ANDROID APP PROTECTION THROUGH ANTI-TAMPERING AND ANTI-DEBUGGING TECHNIQUES

by

JIA WAN

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
January 2018

Copyright © Jia Wan, 2018
Abstract

Android devices remain an attractive mobile malware target in recent years. Android applications (or simply apps) in the device are vulnerable to different attacks which can tamper with the execution of an app to change app behavior so that it performs harm to users or can debug an app to steal private data (source code, user data and behavior). Android app protection is necessary to defend app behavior integrity and protect app privacy.

The app cache, where the app actually runs, is vulnerable to being tampered with. Cache tampering allows for the same behavioral changes as piggybacking. Piggybacking an app is to repackage an legitimate app with extra code that can perform malicious acts after installation, such as stealing user sensitive data or displaying unsolicited advertisements. The cache loading process of Android Runtime (ART) can be exploited by cache tampering attacks without rebooting the device. Security-Enhanced Linux (SELinux) full enforcement has been deployed in the Android platform since Android 5, which enhances the security of Android platform and decreases the security concerns apps should take care of at the same time. Therefore, apps are vulnerable to being debugged in an insecure Android environment such as an emulator or a device with a rooted Android ROM.

We present a comprehensive app protection approach using anti-tampering and
anti-debugging techniques. We implement separate solutions in terms of two protections against tampering and debugging. We maintain the integrity of app cache and implement a lightweight cache protection solution for anti-tampering. We collect debugging points of ART and protect them at runtime from being tampered with. Our solution can be deployed easily across different Android ART-based platforms with little effort. App developers are able to use our techniques to protect their apps.
Acknowledgments

First, I am thankful my supervisor Prof. Mohammad Zulkernine for his support, guidance, patience and motivation through my research at Queen’s.

I would like to dedicate the thesis to my family for their constant support. I am even more grateful for the opportunity to pursue my dream, and to get the best education I could have ever asked for.

I also would like to thank the Canada Research Chairs (CRC) and Mitacs which partially funded this research. Thanks to Clifford Liem, Phil Eisen, and Ben Gardiner for their guidance to my research during my internship.
Contents

Abstract i

Acknowledgments iii

Contents iv

List of Tables vi

List of Figures vii

Chapter 1: Introduction 1
  1.1 Motivation .............................................. 1
  1.1.1 App Behavior Breach ................................. 1
  1.1.2 App Dynamic Analysis ............................... 3
  1.2 Overview of the Proposed Approach ..................... 4
  1.2.1 Cache Protection ..................................... 5
  1.2.2 Dynamic Anti-debugging ............................ 5
  1.3 Contributions ........................................ 7
  1.4 Organization of Thesis ................................. 9

Chapter 2: Background and Related work 10
  2.1 ART ..................................................... 10
  2.2 Cache File ............................................. 12
  2.3 OAT Structure ........................................... 14
  2.4 Compiler Filter ....................................... 16
  2.5 Related Work .......................................... 18
  2.5.1 Cache in an App .................................... 18
  2.5.2 App Anti-tampering ................................. 19
  2.5.3 Static Anti-debugging .............................. 20
  2.5.4 Dynamic Anti-debugging ............................ 20
  2.6 Summary .............................................. 22
List of Tables

5.1 Time consumption for each operation in integrity checks . . . . . . 61
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>App cache tampering</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Debugging an app through an Android instrumentation toolkit</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Cache protection solution</td>
<td>6</td>
</tr>
<tr>
<td>1.4</td>
<td>Dynamic anti-debugging solution</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>Caching loading in ART</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>App cache layout in ART</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Cache loading check in ART</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Cache tampering attack operation</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>The three layers to find the native code of a Java method</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>Vtable tampering</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>ArtMethod entry point tampering</td>
<td>30</td>
</tr>
<tr>
<td>3.6</td>
<td>Native code tampering of a method</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>App cache anti-tampering</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Signing components in the host</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Secure store generation process in the host</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Cache file IV process in the device</td>
<td>40</td>
</tr>
<tr>
<td>4.5</td>
<td>App dynamic anti-debugging approach</td>
<td>42</td>
</tr>
</tbody>
</table>
4.6 Android APIs list .................................................. 42
4.7 Values of Layer 1 in the secure store ............................. 43
4.8 IV pseudo code in Layer 1 ........................................ 45
4.9 IV pseudo code in Layer 2 ........................................ 45
4.10 IV pseudo code in Layer 3 ....................................... 46
5.1 Experimental cache tampering attack launching process ...... 52
5.2 Vtable hooking attack ............................................. 55
5.3 Vtable hooking methods .......................................... 56
5.4 An experiment app ................................................ 57
5.5 Vtable hooking result ............................................. 57
5.6 ArtMethod EP hooking ........................................... 58
5.7 loadUrl() registration in inspector.apk ........................ 59
Chapter 1

Introduction

1.1 Motivation

The Android operating system has a large share in the global smartphone market. The open nature of the operating system leads to more attempts by attackers to target Android applications (or simply apps) [17]. Techniques to protect the integrity of app behavior and privacy should be explored.

1.1.1 App Behavior Breach

Piggybacked apps are popular in third party app markets where legitimate apps are repackaged and leveraged to make profit for attackers [60]. Some research has been done to get apps repackaged for policy enforcement or software analysis [35, 55], but obfuscation technology applied on original apps makes repackaging unrealistic. Also, static signature-based detection can filter out repackaged apps with malicious features and frustrates the attempts of piggybacking [37, 44, 49, 57]. As attackers become much stealthier, an app’s cache may be tampered with to perform the same malicious acts as a repackaged app by exploiting the vulnerabilities in the cache mechanism of
1.1. MOTIVATION

The reliance on an app’s cache to load an app in ART, the default Java virtual machine in Android since Android 5, may be exploited by attackers. Similar to repackaging an app by modifying its source code, an app’s cache can be delicately crafted to achieve the same objective. A cache tampering attack can be launched without the user’s notice and without restarting the device. However, the replacement of an app’s copied Android Package (APK) file (base.apk in the app’s sandbox folder) is only effective after rebooting the device and then Android framework brings it to its user’s attention. A cache tampering attack may occur even the user is careful to install apps from Google’s official app store. Since each installed app runs in a sandbox, the attack must break into the apps sandbox to modify the cache. Cache tampering makes app behavior different as shown in Figure 1.1. The attack is effective when the target app’s process restarts. Hence, when the user taps the app’s icon next time, a malign cache designed to mimic the app’s UI may steal the users account information in the background. Hardened apps may be made through packing services to make reverse engineering more difficult by applying dynamic loading. However, the app’s cache can still be generated once the original Dalvik EXecutable (DEX) format file is loaded dynamically [59]. The DEX file is filled with Dalvik bytecode from Java class files recompilation by Dalvik dx compiler. Therefore, cache protection in ART is necessary for app anti-tampering.

Figure 1.1: App cache tampering
1.1.2 App Dynamic Analysis

App anti-debugging can protect app behavior from being leaked to attackers, thus reducing the potential possibilities of being attacked. Software debugging to monitor program behavior is applicable on Android and even more helpful due to the open-source nature of Android. Android app behavior can be traced and analyzed by software analysis techniques. Software analysis tools used to analyze malware can be leveraged to inspect benign apps which imposes a potential threat to app protection. Once an attacker understands the execution path of an app, an attack can be designed to exploit the target app from the tracing information.

There are static and dynamic analysis techniques for Android. Static analysis techniques employ reverse engineering techniques [26, 29, 39]. However, code obfuscation or other techniques provided by the app packing service like dynamic loading and DEX content encryption frustrate reverse engineering. Dynamic analysis techniques monitor app runtime behavior to get information about the app instead of inspecting the app source code statically. For example, reference hijacking changes the app running environment by modifying app source code and enforcing customized system libraries access for the app to compromise app privacy [55]. Our app cache protection can defeat static code injection used to change the running environment of an app. Virtualization-based analysis approach DroidScope modifies different layers of Android operating system to implement different tracers and aims to provide insight about information leakage. It traces a running app and exposes the execution path of the app [54]. The target app has to run on a special platform with DroidScope extension, which can be defeated by the emulator detection techniques. Another type of Dynamic analysis approach uses an Android runtime instrumentation tool to divert
method execution (or hook methods) to monitor app runtime behavior. Those instru-
mentation frameworks hijack the target app process and hook interesting methods to
analyze input and output for app debugging. Android instrumentation frameworks
do not modify Android operating system or the static app installation file. They are
flexible to be used on different Android devices to monitor benign apps. As shown in
Figure 1.2, these Android instrumentation toolkits are given some debugging points
and hook those points to get debugging information from the app at runtime.

1.2 Overview of the Proposed Approach

A complete app protection approach should address the two threats that perform
malice to breach the integrity of app behavior and trace app behavior to steal app
privacy dynamically. If a cache tampering attack happens, the original app’s behavior
will be modified without restarting the device on which the app is running. Cache
protection is able to defeat cache tampering attacks and defend the integrity of app
behavior. If the dynamic hooking attack happens, the app behavior can be tampered
with without restarting the app process. Our check point protection of sensitive
methods can defeat the attacks. The solution can be deployed easily on different Android ART-based platforms. App developers are able to use our technique to protect apps’ behavior from being tampered with and analyzed dynamically.

1.2.1 Cache Protection

An Optimized ART (OAT) file is the app cache which is large in size, e.g., YouTube’s cache file is nearly 253 MB in the device. By simply encrypting or signing the cache file into hash signature as a secure store, performance overhead may affect the protected app due to CPU usage for computing, extra memory and storage space, and long app loading time that shortens the device’s battery lifespan. Also, to get a signature for the whole cache file is not necessary as not all parts of the cache are targeted by a cache tampering attack. Furthermore, protection should be applied for the vulnerable part that can be reached by attackers to make anti-tampering more targeted. Figure 1.3 shows the solution we provide for app developers. An app developer can submit the app into our cache protection system. The cache protection system runs in the host to generate a secure store which will be attached with an app. We put the time consuming tasks running in the host and try not to impact an app’s performance too much. A native library is provided to do lightweight cache Integrity Verification (IV) in the device.

1.2.2 Dynamic Anti-debugging

App protection includes anti-debugging techniques to protect app privacy (source code, data and behavior) and thus protect an app’s intellectual property. We assume that the target app has applied some app packing techniques like code obfuscation to
1.2. OVERVIEW OF THE PROPOSED APPROACH

prevent static tampering of the app’s source code and data. To counteract dynamic app debugging, we deploy a dynamic anti-debugging technique to thwart Android runtime instrumentation. Previously, an app was able to use PTRACE_TRACEME in a child process to trace itself for anti-debugging and prevent another processes using ptrace to control it since one process can only be traced by one debugger [19, 31]. Since SELinux has been set in enforcing mode by default since Android 5 [23], ptrace is not allowed to be used in apps in devices with Android 5 or higher versions. That means apps cannot have the PTRACE_TRACEME anti-debugging capability in their released versions. Therefore, apps are vulnerable for ptrace debugging when being run on an attacker-friendly system. With root privilege in an emulator or a customized device ROM, an attacker can change SELinux into permissive mode where a ptrace-based instrumentation toolkit can be used for app debugging.

Instead of sticking to ptrace anti-debugging for apps, we build check points for IV operation. If a process uses method hooking for app debugging, the value of some check points will be altered. The IV operation will detect the situation and generate an alert. We have observed that many open-source hooking toolkits can
1.3. CONTRIBUTIONS

Figure 1.4: Dynamic anti-debugging solution

be found online to trace sensitive Android APIs to collect app behavior at runtime [3, 22, 25, 38]. Their solutions are considered effective to analyze apps in specific ART versions. We are not sure if they are effective in the latest ART 8. We analyze these ART instrumentation tools and collect the check points in different layers for our anti-debugging approach. Figure 1.4 shows that a native library attached with the app will do anti-debugging at runtime. The anti-debugging native library will generate secure store and do IV at runtime. The native library will be delivered to app developers for app anti-debugging protection.

1.3 Contributions

We summarize our contributions below:

- We perform a systematic analysis about OAT structure, factors influencing an app’s cache file generation, and cache loading process to explore the possibilities of cache protection in ART. We perform a systematic analysis about method invocation and entry points of different layers in ART 8 (the latest Android version at the time of this writing).

- We propose a defense mechanism for an app’s cache protection to defend an installed app’s behavior [51]. We launch a cache tampering attack to exploit
the vulnerabilities of ART’s cache mechanism that can be leveraged to tamper with the target app’s behavior. The attack is used to assess the effectiveness of our proposed anti-tampering solution.

• We propose an anti-debugging mechanism to prevent a debugging process from tampering with these entry points dynamically [52]. We make adjustments for these open-source ART instrumentation tools and launch these debugging attacks in the latest ART version. The results show the feasibility of the app anti-debugging solution.

• We implement a lightweight IV shared library integrated into the target app in the device and deploy the time-consuming secure store generation operation in a separate powerful server (host). On-device IV is available to detect tampering activities and generate alerts. We also do the performance evaluation of the proposed solution that shows its effectiveness and efficiency.

• We implement a native library integrated into the target app in the device running Android 8. The shared native library will do check point collection and IV at runtime. IV will detect hooking activities and generate alerts to exit the app, thus protecting the app. We also do the performance evaluation of the proposed solution which shows its efficiency.

• The comprehensive app protection approach is implemented on the latest Android version and it is compatible with most of the recent Android versions.
1.4 Organization of Thesis

The rest of the thesis is organized as follows. In Chapter 2, we discuss background information about ART cache in Android 7 (the latest version at the time when the experiments were conducted). Then, we investigate some related work and discuss why our anti-tampering and anti-debugging approach is more advanced in comparison with the previous work. Chapter 3 introduces some threats that can breach the integrity of app behavior and leak app privacy. We detail our implementation for the two solutions and present that the app protection approach can be deployed compatibly in most recent Android versions in Chapter 4. Chapter 5 presents the evaluation results. We conclude and outline the limitations and future work in Chapter 6.
Chapter 2

Background and Related work

The chapter describes background knowledge for understanding the underlying principles of the app anti-tampering and anti-debugging techniques proposed in this thesis. Background information about Android cache mechanism will be elaborately explained. Since there are two files `base.odex` and `base.art` in an app’s sandbox folder, we will demonstrate the two file loading procedures that expose the vulnerabilities. The Optimized ART (OAT) content varies in app cache among different Android versions. We illustrate the OAT file structure in Android 7 and describe how `-compiler-filter` option for `dex2oat` influences the formation of the content in an OAT file. After that, the related research on app anti-tampering and app static and dynamical anti-debugging will be discussed.

2.1 ART

Android apps are developed in Java and compiled into Dalvik bytecode. Android previously used Dalvik as the Java virtual machine to interpret Dalvik EXecutable (DEX) content for execution and introduced ART in Android 4 as an option. DEX content is the Dalvik bytecode in a DEX file. Since Android 5, Dalvik has been
replaced by ART with the aim to improve app performance by executing native code from `base.odex` instead of interpreting Dalvik bytecode. ART is now the default Android Java Virtual Machine (JVM) for Android 5 and the higher versions.

During an app’s installation, `installd` triggers `dex2oat` compiler to create app cache in OAT format. `installd` is the process in the device to receive commands from Android framework for apps’ installation. The compilation operation is time-consuming and the time depends on the size of the app as compilation from Dalvik bytecode to native code takes time. An OAT format file is named `base.odex` in a specific folder and only the app is able to access (except root privilege). This folder is the app’s sandbox folder. When an app starts next time, ART will load the app’s cache file into the memory of its own process rather than the reinstallation of the app’s APK file to reduce startup time and improve runtime performance. The cache file contains the app’s Dalvik bytecode for runtime interpretation or native code for direct execution.

In Android 7, an app’s compiling mode is decided by `-compiler-filter` option. Three compiling modes are introduced to save power, improve runtime performance, and reduce installation time that is mainly occupied by native code compilation on both Android 5 and 6. They are Ahead Of Time (AOT), Just In Time (JIT), and interpreter for Java runtime. AOT translates Dalvik bytecode into native code during an app’s installation by the on-device compiler `dex2oat`, while JIT is much more flexible to compile Dalvik bytecode at runtime. A compilation daemon is used to compile collected classes or methods in a profile file when the device is idle and charging [10]. The profile guided compilation will store frequently executed methods into an app’s image file (`base.art`), which avoids JIT compilation again [12, 15]. The interpreter interprets Dalvik bytecode for execution without consuming time for
2.2. CACHE FILE

The cache file is a special Executable and Linkable Format (ELF) file with an OAT structure and stored in an app’s sandbox folder. For example, YouTube’s cache is /data/app/com.google.android.youtube-1/oat/arm64/base.odex created by dex2oat compiler. /data/app/com.google.android.youtube-1 is the app’s sandbox folder (the app’s data folder). base.odex is actually an OAT file with .odex extension. An image file base.art in the folder consists of compiled frequently used methods, which improves runtime class lookup performance.

A background thread is used to collect resolved classes and methods in ART and store indices of them in a profile file in /data/misc/profiles permanently when they are compiled by JIT or interpreted by interpreter and accessed frequently enough to exceed a threshold. The background thread is called ART’s ProfileSaver thread [5]. The image file (base.art) is generated to store the compilation information of frequently used methods according to the profile by a compilation daemon triggered by many conditions, e.g., charging, idle. The compilation information includes locations of compiled classes and methods in the OAT file (base.odex). The compilation daemon uses dex2oat to compile according to the records in the profile.

Fig. 2.1 shows how ART loads cache into memory when an app runs after its installation. For installed apps, PathClassLoader is the class loader to load them into memory. When cache loading is invoked from Android framework, the path of base.apk in the app’s data folder is passed to ART that tries to load base.odex first. If base.odex does not exist, ART will roll back to load DEX content in the
2.2. CACHE FILE

Figure 2.1: Caching loading in ART

APK file. If base.odex exists, ART will check the existence of base.art and update ClassTable to accelerate linking of methods when class linker looks up classes. ClassTable is a sophisticated structure to record already found classes into memory. Otherwise, ART reads DEX content from base.odex and uses DefineClass (a representation to find a class by traversing all included Android framework’s cache) which is slower than ClassTable searching to link classes in terms of runtime performance.

An attacker is able to exploit the cache loading process by removing base.art that is harder to tamper with and by modifying static base.odex. The operation may incur performance penalty as ClassTable is removed. However, it is possible to get DEX content or native code from base.odex and the performance is better than an APK’s reinstallation. A cache tampering attack is made to remove base.art and tamper with base.odex. ART will load the modified app cache (base.odex) into the
2.3. OAT STRUCTURE

memory of the app’s process, which may modify the target app’s behavior.

2.3 OAT Structure

As Dalvik is the JVM on older Android releases, app cache only contains optimized DEX content. Now ART introduces OAT structure embedded in an ELF cache file. Fig. 2.2 shows the OAT file format where OAT occupies two segments. oatdata in ELF’s .rodata segment stores OAT data content while oatexec in ELF’s .text segment is filled with platform-specific native code. The native code is generated when an app is installed and compiled by dex2oat. ART supports seven types of instruction architectures: Mips, Mips64, X86, X86_64, Arm, Arm64, and Thumb2. It means that cache files produced by the same Android release on different instruction architecture platforms are different.

Four kinds of sections reside in oatdata segment: OATHHeader, OatDexFile, DexFile, and OatClass. OATHHeader contains important fields like instruction set of the device and the number of DEX structures in the cache file that is equal to the number of classes*.dex files in the APK file. adler32_checksum in OATHHeader specifies the checksum of the current OATHHeader and all DEX content. image_file_location_oat_checksum, the other checksum is used to verify the legitimacy of the cache file that will be discussed in Secion 3.1. key_value_store specifies command line of dex2oat to create an app’s cache. The command line involves many options like –oat-file and –compiler-filter.

OatDexFile is a small structure mainly to specify the offsets of both DexFile and OatClass at oatdata segment. An OatDexFile structure also contains a checksum field (dex_file_location_checksum) specifying the origin of the corresponding DexFile
structure. Multiple DEX files have been supported in Android 5 and the higher versions [11], which means that an OAT file may contain multiple DexFile structures and thus many OatDexFile structures. For example, there may be many DEX files in an APK such as classes.dex and classes2.dex. We use classes*.dex to represent DEX content in an APK.

DexFile is the optimized content of classes*.dex [8]. Optimization happens only in the bytecode section of DexFile. Optimized DEX has the same DEX structure as an APK’s classes*.dex. The checksum in OatDexFile is the same as the CRC32 checksum of the corresponding classes*.dex, even though DEX content is optimized and the checksum of the optimized DEX content is supposed to be different from the
2.4. COMPILER FILTER

checksum of the original DEX content. A DexFile structure contains several fields such as constant string index list and method index list. These fields help locate methods in OatClass section. If the target app’s behavior is expected to behave differently in a cache tampering attack, bytecode in DexFile needs to change. Our cache protection mechanism will extract the target app’s DexFile structures from the app’s cache as DexFile reflects an attack’s modification. DexFile content is the vulnerable part of an app’s cache file.

OatClass contains the description of one class with method locations to locate native code in oatexec segment. An OatClass structure describes one class in each DexFile. type in one OatClass indicates the compilation state of the class. There are three compilation states: non-compiled, some-compiled, and all-compiled. Non-compiled means that the method is interpreted by Dalvik bytecode interpreter. All-compiled means that the method is compiled by AOT. Some-compiled means that OatClass uses a bitmap to record a compiled method index to locate native code. method_pointer records the offsets of methods’ native code in oatexec segment.

2.4 Compiler Filter

The concept of compiler filter was introduced in Android 7 and will continue to exist in its higher versions. The idea is to compile apps or framework libraries in different modes with regard to Android runtime performance and device hardware conditions. Some scenarios that have to be considered are: AOT compilation consumes too much time during an app’s installation, or a mobile device may be short of space to store large cache files of compiled native code. Hence, many compilation options are provided to expedite apps’ startup, improve user experience and save battery and space.
An app is installed with `–compiler-filter` of `dex2oat` set to `interpret-only`, which removes compilation time and reduces app installation time for better user experience. However, it sacrifices the app’s runtime performance since Dalvik bytecode interpretation is slower than native code execution. The selection of compiler filter options is a trade off between app’s runtime performance, app’s installation experience, and device conditions.

There are twelve compiler filters in Android 7, while four are officially supported in Android 8 [12]. This will be discussed in Section 4.1.2. There are two categories in terms of compilation options in Android 7: one is for system image configuration and the other is for app compilation. In this thesis, we discuss only the app compilation category, that is the `–compiler-filter` option. For example, `speed-profile` takes advantage of profile-guided compilation. `interpret-only` optimizes some Dalvik instructions of DEX content to get better interpreter performance. `speed` does AOT compilation for all methods to increase app execution speed [8, 12].

The `DexFile` in the cache is not the exact Dalvik bytecode in `classes*.dex` of the APK. Different `–compiler-filter` options generate different cache files. For example, `verify-profile` does not have DEX optimized and the cache file contains the exact Dalvik bytecode in the APK. `interpret-only`, `speed` and `space` do DEX-to-DEX optimization differently. The differences among all compiler filter options for DEX content optimization will not be covered in this thesis.

Android uses different compiler filter options to compile apps depending on platforms’ configuration. Therefore, how an app is compiled is uncertain across different devices. We deploy the time consuming compilation process in the host to generate apps’ secure data (secure stores) by applying all possible compiler filter options.
2.5 Related Work

In this section, we present the related research about app anti-tampering and app anti-debugging respectively. For app anti-tampering, many researchers have put effort into signature-based schemes for malware detection and to develop obfuscation techniques to mitigate the effects of piggybacking to detect modified apps before installation, thus protecting apps. However, little has been done to protect apps after their installation especially to protect app cache. Lest et al. introduced the anti-debugging technique to protect software [41]. Anti-debugging techniques help malware evade detection both on desktop and mobile platforms [31], but can be used to protect benign apps too. They include static and dynamic anti-debugging. Anti-debugging approaches can be categorized in the same way as app analysis methods. Therefore, for app anti-debugging, we examine the existing techniques to counteract app debugging statically and dynamically.

2.5.1 Cache in an App

Finley et al. [40] presented a cache cleaner to remove apps’ cache and keep users’ privacy from being leaked. This app cache is about sensitive data from web browsers, network connections or emails that will not change app behavior. This kind of app data is different from Android app cache which we are concerned about. The cache we want to defend acts as an app’s execution file once the app is installed. The app cache contains an app’s Dalvik bytecode and executable instructions.
2.5. RELATED WORK

2.5.2 App Anti-tampering

Sabanal [21] demonstrated the possibility of replacing an ART generated cache with a modified OAT file by running `dex2oat` in the device manually to change the behavior of apps and the framework. The research inspired us to defend an app’s behavior integrity from the ART cache. Reference hijacking [55] exploits the startup process of an Android app and repackages app to load malicious system libraries without root privilege. The attack can evade the detection of static malware analysis technology. However, our cache protection proposal can defeat reference hijacking attack since a repackaged app will result in the modification of an app’s cache and breach the integrity of the original app [45].

Schulz [48] proposed obfuscation techniques to build apps that need attackers’ more effort to analyze and piggyback apps [30]. Jeong et al. [43] proposed to encrypt an app’s essential part to prevent the app’s source code from being attacked. It makes pirating these apps more difficult, but it cannot guarantee the consistency of the app’s runtime behavior. Packing services adopt obfuscation technology and dynamic loading that make static analysis more difficult, while researchers present approaches to unpack apps to dump DEX files of the apps and make it possible for attackers to reverse engineer, modify and repackage apps [42, 53, 56, 59]. Moreover, instead of tempting users to install malicious apps, cache tampering attacks target installed apps by modifying target apps’ cache. Even though an attacker cannot analyze a hardened app, cache tampering allows to modify the app’s behavior totally by replacing the target app’s cache with a malicious app’s OAT file. The app’s runtime behavior has been modified while the user still think that the legitimate app is running in the device.
2.5. RELATED WORK

2.5.3 Static Anti-debugging

App static anti-debugging means that app can defend its source code from being inspected by different techniques like obfuscation techniques. Some researchers have put efforts to make apps’ source code more secure [36, 43, 47, 56]. Those techniques are put in the app packing service. Cho et al. [33] implemented an approach to prevent a debugger from debugging Android apps. They delved into how a commercial debugger IDA interacting with an app for dumping the app’s DEX content even if the app has been statically protected by a packing tool DexProtector. They presented an anti-debugging scheme to protect app from being analyzed by cutting off the debugging communication between IDA and its JDWP-based debugging support tool running in the device. The technique is implemented in Dalvik which is now replaced by ART. To counteract the packing service, research to unpack the protected app DEX content is emerging. Lim et al. introduced an approach to extract malware DEX content out of a packing shell [46]. Xue et al. used check points in ART to reconstruct original app source code [53]. Packing and unpacking are evolving in an arms race.

2.5.4 Dynamic Anti-debugging

Compared to static app anti-debugging for source code protection, dynamic app protection defends app data and behavior leak at runtime. Zhang et al. proposed App Guardian to analyze all third party apps on a device, aiming to find apps with malicious permission and abnormal runtime behavior [58]. App Guardian utilizes side-channels from /proc interface to analyze an app’s process workload or network transaction which can infer an app’s behavior. It protects the legitimate app supposed to access the data by detecting other apps’ abnormal data collecting behavior.
and killing suspicious malware at runtime. However, much research is focused on app behavior debugging. Some malware behavior analysis techniques can be used to debug benign apps. DroidScope modifies Android all layers to insert its tracers from Android API to native instruction. Its JVM tracer only supports Dalvik which has not been supported since Android 5. These tracers can specify certain app behavior like SMS message and HTTP connection and only run on top of an emulator [54, 28]. In addition, reference hijacking can change app accessed framework libraries to customized ones for app behavior analysis by repackaging the app [55]. However obfuscation technique can make repackaging an app hard. Besides, the modification to repackaging an app breaks the integrity of the original app and can be defeated by our cache protection approach.

Some researchers introduced new techniques to instrument app dynamically without modifying an Android system by controlling the target app’s process and injecting code into its memory [27, 34, 38]. These techniques can trace an app by choosing interesting APIs for different purposes flexibly. To counteract dynamic hooking, desktop operating system employs self-debugging that uses a child process attached to the parent target process to make the debugger unable to hook the target process. However, most of the recent Android versions utilize SELinux for security which disallows self-debugging. Some researchers collected an emulator’s features to conduct anti-virtualization of malware which makes the app conceal its malicious behavior [32, 50] when the app runs in the emulator. However, anti-virtualization to prevent being debugged is defeated by changing the environment into a real device. Our dynamic app anti-debugging aims to protect apps running both on an emulator and real devices for our app developers. The anti-debugging solution expects to defeat Android
2.6. SUMMARY

In this chapter, we introduce some background information before we detail our proposed app protection approach. First, through the explanation of cache structure, the vulnerable part is exposed. Therefore, we protect the vulnerable DEX content in app cache instead of the whole app cache to make our app anti-tampering approach more effective since the DEX content would change if a cache tampering attack is launched. We explore the compiler filter options to see whether we can generate app cache in the server since compiling an app to get app cache is time consuming. App cache compilation in the host makes our cache protection approach efficient.

Next, we summarize the related work associated with the research. We present that static debugging and anti-debugging techniques are in an arms race. We assume that app would be packed to prevent static debugging. Therefore, we will not delve into developing a static anti-debugging approach in this thesis. We discuss some instrumentation tools at runtime.

Except hooking debugging points through the ArtMethod class’s invocation process, the whole ArtMethod class can be replaced in memory at runtime by ArtMethod class of another method [7, 18, 24]. Many companies deploy this hooking technique for app patching (hot fix) like alibaba’s AndFix [1]. This kind of Java method hooking makes a hooked method work instantly once its being invoked without restarting the target app to make the patch work. The whole Java class modification results in the change of other information in the class like declaring.class and dex.method.index which will be verified if the method is invoked by reflection. If the check fails, this type of hooking will result in the target app crash.
dynamic debugging approaches need to run in a specific environment. Some Android instrumentation tools are used to debug apps dynamically but there is a lack of methods to counteract debugging. The anti-debugging research is mostly inspired by those open-source Android instrumentation tool kits.
Chapter 3

Threat Model

This chapter shows the attacks performed to change and debug app behavior. We describe those threat models to demonstrate how those attacks can compromise the integrity of app behavior and app privacy. We will introduce threat models for app tampering and app debugging in the respective sections. Section 3.1 will detail the attack to tamper with app cache. The layered hooking attacks will be elaborated in Section 3.2.

3.1 Cache Tampering Attack

ART checks the legitimacy of a cache file (base.odex) to ensure that the cache is generated by dex2oat compiler. An app’s cache can be loaded into the memory of the app’s process when an app starts up and its process is newly created. Fig. 3.1 shows the cache loading check in ART.

ART calculates CRC32 for each classes.dex in the APK (base.apk) and compares them with the checksum in OatDexFile one by one since there may be multiple DEX files. This makes sure that the cache file is made originally from the APK. If the check passes, image_file_location_oat_checksum in base.odex’s OATHeader is
3.1. CACHE TAMPERING ATTACK

Figure 3.1: Cache loading check in ART

extracted to compare with \texttt{adler32 checksum} in \textit{OATHeader} of on-device \texttt{boot.oat} (Android framework's cache). This operation ensures that the cache file is generated in the device. If the check passes, \texttt{base.odex} is legitimate.

In Section 2.4, we mentioned that DEX-to-DEX optimization may change DEX content in an OAT file by \texttt{dex2oat}. However, checksums in \texttt{base.odex}'s \textit{OatDexFiles} still keep the CRC32s of the original \texttt{classes*.dex} in the APK.

The checking process is vulnerable as checksums can be replaced by the right ones to satisfy the check if the attacker is proficient about OAT structure. Sabanal [21] presents an approach to launch the attack stealthily. The attacking process is shown in Fig. 3.2. An attacker can do reverse engineering for one APK, modify
the smali code (an intermediate code representation generated by Baksmali disassembler) as desired and repackage the modified code into \texttt{app-T.apk} by APK tool. \texttt{base-T.odex} is generated by on-device \texttt{dex2oat} to get the right \texttt{adler32_checksum} from \texttt{boot.oat} in the device. The operation can be done in the host as long as the Android Open Source Project (AOSP) environment is built to execute \texttt{dex2oat} with necessary framework \texttt{jars} in the right Android version. If \texttt{dex2oat} compiler is operated in the host, \texttt{image_file_location_oat_checksum} of \texttt{OATHeader} in the newly built \texttt{base-T.odex} should be replaced by the value of \texttt{adler32_checksum} in \texttt{OATHeader} of the target device’s \texttt{boot.oat}. The next step is to modify \texttt{base-T.odex} with the original \texttt{base.odex}’s checksum of \texttt{OatDexFile}. At last, \texttt{base-T.odex} is put into the
app’s cache folder to replace the original `base.odex`. The attacker has to acquire access to the app’s sandbox folder by jailbreaking the device. As a result, when the victim app starts up with its newly created process, a cache tampering attack can be made without the user’s notice.

3.2 Dynamic method hooking

Open-source Android hooking instrumentation tools are available at github [4, 6, 22]. The authors of these tools use different hooking points to achieve the goal of debugging an app. We categorize them according to the hooking depth from the perspective of a method invocation. Fig. 3.3 shows the layers that the method invocation has to go through to find the native code of a Java method. The figure assumes that the ART is not running in the interpreter mode where the Java method is interpreted for execution and no native code will be generated. A Java method may be compiled into native code stored in the cache file in the speed mode which is one of the four app compilation modes promoted to enhance app runtime performance in Android 8 [12].

For example, ARTDroid [34] changes the entry point in the virtual table (vtable) that dispatches virtual methods since it only hooks sensitive Android APIs, most of which are virtual methods. We assume that hooking attacks would not replace the whole `ArtMethod` class of a Java method like AndFix [1] and do not cause app crash. Therefore, we define this point as Layer 1 as it is the potential top point to dispatch a method. ProbeDroid [22] explores deeper into the entry point of a Java method which is the execution of the method. We set this kind of entry point as Layer 2. ARTIST [38] is an instrumentation framework that tampers with native code of the
Java method which we see as the bottom layer of method invocation. We set the
native code as \textit{Layer 3}. We will choose a typical hooking instrumentation tool for
each layer and illustrate their mechanisms in detail. The description of the three
layers are based on the implementation of ART in Android 8.

![Diagram of the three layers to find the native code of a Java method]

Figure 3.3: The three layers to find the native code of a Java method

\subsection*{3.2.1 Layer 1}

Each Java method can be described as an \texttt{ArtMethod} class in ART, the default Java
Virtual Machine (JVM) in Android 5 and higher. The vtable in a class is used to find
a virtual method for invocation. The \texttt{ArtMethod} class has a data member to specify
the declaring class of the Java method. A data member of the declaring class is the
\texttt{vtable}, an address pointing to an \texttt{ArtMethod} table. The members of the table are
addresses of \texttt{ArtMethod} classes for all virtual methods except the offset. The vtable
has an offset before those \texttt{ArtMethod} addresses. The offset varies among different
ART versions.

Fig. 3.4 shows how vtable tampering happens. The \texttt{ArtMethod} class pointer can be
obtained through JNI interfaces by inputting class name, method name and method
3.2. DYNAMIC METHOD HOOKING

signature as arguments such as `FindClass()` and `GetMethodID()`. `method_index_` in `ArtMethod` class is used to locate the virtual method in the vtable. ARTDroid makes the attack by tampering with the target method slot of the table. The target `ArtMethod` class address in the vtable is replaced by the hooking method address. Therefore, next time the target method is invoked somewhere, the vtable of its class is searched and the hooking method address will be returned to make the attack work.

![Figure 3.4: Vtable tampering](image)

3.2.2 Layer 2

This layer focuses on the internal structure of `ArtMethod` class. `entry_point_from_quick_compiled_code` records native code offset of a Java method after Class Linker (CL) links the method by searching native code offset from the cache. For Android APIs, most of their implementation is in `boot.art`, an image file storing bytecode of pre-loaded classes. Android framework’s cache is `boot.oat` where the native code of compiled methods may be found. In Android 5, Ahead-of-Time (AOT) is introduced
to translate Dalvik bytecode into native code during an app’s installation for the improvement of app runtime performance. However, the native code compilation is configurable and set according to the compiler filter setting of device manufacturers [13]. We assume that compiler filter option is set as speed for the system image, thus native code are stored in boot.oat and mapped into memory of an app’s process. Therefore, CL can obtain the native code address and assign it to the invoked method during class loading.

Fig. 3.5 shows assigning an offset of hooking code (malicious code) can modify the execution of the original method. The hooking code may be a snippet of assembly code like what has been done in ProbeDroid that branches to a C\C++ method to complete tracing the arguments, invoking the original method and getting the return value of the original method for app behavior monitoring.

3.2.3  Layer 3

The native code in boot.oat is explored in this layer. Instead of the complete native code of a Java method being modified, only the first instruction is changed and hooked by a malicious instruction. For example, the first 2B of native code in Thumb or the
first 4B in ARM. We assume that the debugged Android APIs are fully compiled in the system image, thus the native code exists in `boot.oat`. If the method is owned by the target app, its native code should be searched in the app’s cache `base.odex` while the app’s DEX content resides in `base.vdex` in the app’s sandbox folder.

![Diagram of DEX and OAT structures](image)

**Figure 3.6: Native code tampering of a method**

Since DEX content of the Android framework jar files is in `boot.vdex` in Android 8 instead of the OAT file in previous ART versions. Searching the method native code
3.2. DYNAMIC METHOD HOOKING

needs to wade through Optimized ART (OAT) and DEX structures. That means visiting the memory region of `boot.oat` and `boot.vdex` in the target app’s process is necessary to locate a method’s native code. Fig. 3.6 demonstrates each structure that needs to be examined. OatDexFile contains two fields specifying the offset of its DexFile in `boot.vdex` and the offset of OatClassOffset array that includes offsets of OatClass structures. OatClass structures describe the classes of DexFile. There may be several `classes*.dex` files in an app or framework libraries [14], so are the DexFile structures.

Each DexFile represents one `classes*.dex` file. To find a method’s native code, the class name of the method is the first input to get the string index in string_ids array. After searching through type_ids array and then class_defs array, class index is obtained to locate OatClass offset in OatClassOffset array. OatClass contains methods_pointer that is filled with native code offsets of methods in the class. The next step is to find method index to locate the native code offset of the method in methods_pointer. Like class searching, method name and signature are input to get method index by wading through string_ids array, proto_ids array, and method_ids array. After native code of the traced method is located, an illegal instruction is used to replace the first instruction of the native code, which aims to trigger SIGILL. The original instruction is saved to restore execution after the debugging work. ARTIST has two kinds of trapping instructions: illegal instruction and breakpoint instruction for both ARM and Thumb. Native code of the debugged methods are assumed to be executed; therefore, the kernel would trigger SIGILL or SIGTRAP once they are invoked. Signal action function would be called to divert the original method’s execution and trace arguments.
3.3. SUMMARY

The three layers specify the vulnerable method locations that can be hooked for app behavior debugging. Those hooking techniques are called “native hooking” and exist in most Android instrumentation tools. We aim to analyze these hooking tools and design a targeted approach to defend app anti-debugging against them.

3.3 Summary

The ART cache mechanism is vulnerable to a cache tampering attack that aims to change app behavior. We dive into checksum checking operation in the cache loading process to expose the vulnerability before we design a approach to detect the attack.

We analyze those open-source hooking attacks that aim to hook some methods in a certain layer for app debugging. Therefore, we summarize that there are three layers vulnerable to be hooked during a Java method’s invocation and will design a specific approach to protect them.

All these attacks need root privilege to access the app sandbox folder or app process. For app cache tampering, although app cache access needs special permission, we cannot guarantee the root status of a device that has the app installed. We want to assure that the app behavior is harmless even the device is rooted. The same condition applies for app debugging: an attacker may debug the target app in an insecure environment like an emulator or an rooted device. We want to keep app privacy secure even in an insecure environment.
Chapter 4

Implementation

This chapter describes how our app protection proposal can curb the harm of app behavior tampering and app information leakage. We describe the detailed implementations for both our app anti-tampering solution and app anti-debugging solution. Both solutions follow the same paradigm which generates a secure store first and verifies integrity by comparing with one item in the generated secure store. Section 4.1 illustrates app cache protection to prevent app from being tampered with. The check point protection in our dynamic anti-debugging scheme can defeat app analysis by app runtime instrumentation, as in Section 4.2. Our cache protection approach is implemented based on Android 7 (before Android 8 was released), but the cache protection methodology can still be applied on Android 8. Section 4.1.2 explains that our cache protection approach is compliant with the developing trend of ART.

4.1 Cache Protection

As the vulnerable cache exploitation was demonstrated in Section 3.1, new technology should be explored to defeat cache tampering attacks effectively and efficiently and
should be deployed easily across different Android platforms. In this section, we illustrate the design and implementation of a technology that protects app behavior integrity by anti-tampering the app’s cache. Section 4.1.1 introduces the basic concept of our design. We describe the compatibility for the next version of Android 8 in Section 4.1.2. Section 4.1.3 and Section 4.1.4 elaborate our cache protection system implemented in both host and device.

4.1.1 Basic Idea

We assume that if an attacker can tamper with the source code or smali code of an app after reverse engineering, *classes*.dex (APK’s Dalvik bytecode) in the tampered malicious app will be different from the ones in the original APK. The cache generated from the malicious APK by dex2oat compiler will be different from the original cache. The difference is in the DEX content of OAT structure. Our design goal is to make the user aware of cache tampering by sending an alert. Our implementation is forward compatible and can be updated easily as ART changes in each Android release. We make performance affected tasks run in the host and try not not to impact an app’s runtime performance.

We decide to use a secure store generated in the host which is actually a file with DEX content signature in a presumably secure format. The file will be attached with an app as an asset. Since ART is the default Android runtime in Android 5 and higher, an OAT file is generated as cache for booting the app instead of re-installing the original APK each time when the app starts up after its initial launch. The OAT file should be protected appropriately to guarantee the app’s behavior integrity when the app boots up. We perform a lightweight cache Integrity Verification (IV)
4.1. CACHE PROTECTION

Figure 4.1: App cache anti-tampering

operation every time the app starts up. The idea is shown in Fig. 4.1. The cache is base.odex, an OAT file which contains optimized DEX content [8].

App developers may apply app hardening technology to load sources at runtime. Packing services adopt special ClassLoader to dynamically load the APK to assure that attackers cannot take advantage of the APK file. However, ART may generate a cache file for the loadee when the protected app hardened inside a shell is loaded [59]. Hence, cache file protection is still needed.

4.1.2 Compatibility

Our approach is based on the latest Android release (Android 7) at the time of this writing. In Android 8, three cache files are expected to be in an app’s cache folder instead of two (base.odex and base.art) in Android 7. They are .vdex, .odex and .art, where .vdex has DEX code of the APK. The method of cache protection will be the same. We will keep an eye out for Android’s version update and reflect the changes appropriately in our proposed system. Furthermore, four compiler filter options will
be supported officially in Android 8 rather than twelve in Android 7. The work of these four app compilation modes are more definite [12]. Our protection technique may be compatible with future Android releases.

4.1.3 On-host Secure Store Generation

A signing system shown in Fig. 4.2 is designed to generate secure stores for apps. A secure store for an app contains DEX signatures of different compiler filter options on all possible instruction architecture platforms. The signing system utilizes AOSP environments to build OAT files that need Android framework jar files to link classes and optimize Dalvik bytecode inside an OAT structure. The idea is to deploy AOSP environments in different Android versions in the host and generate a secure store of an app for different instruction architecture platforms such as Mips, Mips64, X86, X86_64, Arm, Arm64, and Thumb2. On-host dex2oat compiler generates secure stores. oat2dex is implemented to extract DEX content from an OAT file and may be compatible with different Android versions since OAT structure evolves gradually. The host is a server with different AOSP building environments to build framework jar files for different platforms.

Fig. 4.3 demonstrates DEX signing and a secure store formation working process. The signing system runs in the host to generate DEX signatures, encrypt or hash them, and store them in a secure store. The secure store will be attached in the target app. The signing system uses the target app as an input. In our experiments, we use the adler32 algorithm to get one signature for each DEX file in the target APK. dex2oat built in an AOSP environment runs in the host to optimize original
4.1. CACHE PROTECTION

Figure 4.2: Signing components in the host

Figure 4.3: Secure store generation process in the host
4.1. CACHE PROTECTION

classes*.dex according to different compiler filter options. Corresponding DEX signatures will be generated for IV operation in the device. From the experiments, we find that DEX content are different because of -compiler-filter options. For example, DEX content in speed-profile is different from speed and both are different from verify-profile. How compiler filters optimize DEX is not discussed in this paper.

A secure store is organized as a map involving instruction sets, compiler filter options, and corresponding DEX signatures. The target app will attach the secure store and verify the integrity of the app’s cache when the app starts up. A signing system is implemented to gather OAT files of different compiler filter options under different instruction sets. Four bytes’ signatures is used for each DexFile in these OAT files in our experiments.

4.1.4 Integrity Verification

Fig. 4.4 illustrates an app’s cache IV process. When an app is installed, Android installld process will trigger dex2oat to create a cache file in the app’s cache folder. The cache file is an OAT file named base.odex. oat2dex is implemented in a native library to analyze the OAT file and extract DEX content from it. The compiler filter option and instruction set in the OAT file can be obtained from OATHeader. A secure store is put in the target app’s asset folder. The target app uses the native library to generate DEX signatures and look up the secure store to find a match with the series of DEX signatures when it starts up. If the cache is tampered with and replaced by malicious one, the IV native library will check and send an alert.

For example, an app owner can submit an app into our system to get a secure store which will be attached with the app. A native shared library will be delivered
4.2 Dynamic Anti-debugging Scheme

We assume that the target app has been packed by a packing service to make the app difficult to reverse engineer. Therefore, class name, method name and method signature of defined methods in the app are hard to find. Though using unpacking service to unpack an app is possible for static analysis and debugging [53], we will only discuss check point protection of Android APIs for dynamic anti-debugging in this thesis. Android apps use Android APIs to perform different features like sending text messages and dynamic loading DEX content. The goal of protecting an app from being debugged or traced dynamically is to provide a method for dynamic integrity
4.2. DYNAMIC ANTI-DEBUGGING SCHEME

checks at runtime compatible with different ART versions, and exit the app when method debugging is detected.

We have two parts in our anti-debugging approach. One is secure store generation, the other is integrity checks that try to find matches of the records in the secure store through the three layers. Our anti-debugging approach is shown in Fig. 4.5. *libanti-debugging.so* is implemented to complete the task. If one check fails, the app will exit and its process will be killed to prevent the attacking from exerting further harm on the app. The target app has a secure store generated every time the device boots up to record the correct values, since the memory may be displaced for *boot.oat* and *boot.vdex* different from previous times, and memory allocation for the *ArtMethod* class by ART may be different. A secure store records the values of the vulnerable method check points in these three layers before the app icon is firstly tapped. The work of integrity checks goes through the layered method locations separately, gets the vulnerable values, and compares them with the ones in the secure store. We implement our approach in a native library and provide the anti-debugging scheme for app developers who can list the Android APIs they use in a file. We assume that the file can be encrypted in the device and kept in a safe place accessed by our anti-debugging library.

The file is named *methods.list* and the format is shown in Fig. 4.6. Class name, method name and method signature of each protected API should be listed in the file. For example, *getMethod()* is used for Java reflection. The method name can be traced through hooking this API. *loadUrl()* is to load an URL in a WebView. The link of the URL can be debugged. Protection of those methods used in the app can prevent an app’s privacy being leaked to attackers and thus reduce the attack surface.
4.2. DYNAMIC ANTI-DEBUGGING SCHEME

Figure 4.5: App dynamic anti-debugging approach

```json
{ "list": {
    "AndroidMethods": [
    {
      "class-name": "java/lang/Class",
      "method-name": "getMethod",
      "method-sig": "(Ljava/lang/String;Ljava/lang/Class;)Ljava/lang/reflect/Method;"
    },
    {
      "class-name": "android/webkit/WebView",
      "method-name": "loadUri",
      "method-sig": "(Ljava/lang/String;)V"
    },
    ...
  ]
}}
```

Figure 4.6: Android APIs list
4.2. DYNAMIC ANTI-DEBUGGING SCHEME

4.2.1 Secure Store Generation

We collect these check points of each Android API in a layered way as we analyzed in Section 3.2. The secure store is actually a file with a format we assume secure. How to get the values of the three layers depends on the version of ART in the device since the structure of ArtMethod class and Class class varies among Android 5 - 8 (ART is the default JVM since Android 5). We discuss the implementation of our approach in Android 8.

Layer 1

The value of a vtable slot is the address of an ArtMethod class. If a vtable hooking attack happens, the value in the slot will change. We record the addresses of ArtMethod class for Android APIs in methods.list. Fig. 4.7 lists part of the secure store for this layer.

\begin{verbatim}
Layer1
android/webkit/WebViewloadUrl(Ljava/lang/
String;)V=2928656884
java/lang/Class.getMethod(Ljava/lang/String;
Ljava/lang/Class;)Ljava/lang/reflect/Method;=1872806204
...
\end{verbatim}

Figure 4.7: Values of Layer 1 in the secure store

Layer 2

The entry point in an ArtMethod class is the address of its native code. If the native code of one method does not exist (some method may not be compiled by AOT), it will point to the address of the method’s bytecode for interpretation. If ART is forced to enter interpreter mode, the pointer will point to a bridge function
like `art_quick_to_interpreter_bridge` to go for interpretation. For this layer, we record the address of the target method’s execution code.

**Layer 3**

We search for the memory range of `boot.oat` and `boot.vdex` in the target app’s process memory to find the address of a Java method’s native code. The first instruction will be stored into the secure store.

These original values collected from the three layers for Android APIs in the `methods.list` file are stored in the secure store. Since the memory addresses will change after device reboot and the hooking attacks could be launched any time the device is on, the secure store must be updated every time the device reboots. We suggest that app developers to use a service reacting to the `android.intent.action.BOOT_COMPLETED` broadcast from the Android framework. Therefore, the secure store will be updated in a timely way.

### 4.2.2 Integrity Checks

The operation of IV should run from time to time to keep the app vigilant about the danger of being debugged. The checking intensity is decided by app developers.

**Layer 1**

Fig. 4.8 shows how to get the address from a vtable slot. The code relies on the right structure of `ArtMethod` class and `Class` class. We use `method_index` to locate the method slot in the vtable. But the offset is unspecified. In our experiments, we dump the whole vtable of the class. The size of the vtable can be obtained by subtracting
4.2. DYNAMIC ANTI-DEBUGGING SCHEME

virtual_methods_offset_ from copied_methods_offset_. At the vtable, we seek the address of the target method’s ArtMethod class. After some calculation with method_index_, we get the offset fixed in this Android version. If the address in the slot does not match the one in the secure store, we will quit the app to make it safe.

```java
jclass c = env->FindClass(class_name);
jmethodID method = env->GetMethodID(c, method_name,
method_signature);
Class declaring_class = (ArtMethod) method -
>declaring_class, method_index = method->method_index_
>vtable = declaring_class->vtable
slot_value = *(vtable + offset + method_index * pointer_size)
if (slot_value != method_from_layer1_secure_store)
    quit_app
```

Figure 4.8: IV pseudo code in Layer 1

Layer 2

Fig. 4.9 introduces how to get a method’s execution point in an ArtMethod class. Getting the value of entry_point_from_quick_compiled_code_ is very easy by accessing the ArtMethod class.

```java
jclass c = env->FindClass(class_name);
jmethodID method = env->GetMethodID(c, method_name,
method_signature);
EP = method-
.ptr_sized_fields_entry_point_from_quick_compiled_code_
if (EP != method_from_layer2_secure_store)
    quit_app
```

Figure 4.9: IV pseudo code in Layer 2
Layer 3

The memory region of the process can be analyzed from the /proc/self/maps interface. Fig. 4.10 shows the process to get the first instruction of the method’s native code. Both `findClass()` and `findMethod()` use description of the class and the method to find method index that can locate native code address in the `methods_pointer` of `OatClass`. The value in the native code address that is the first instruction of the method is compared with the one in the secure store.

![Figure 4.10: IV pseudo code in Layer 3](image)

The checking is launched following layer depth of a method invocation. For example, the secure store is generated when the device boots up. App developers can set the check intensity that is the time interval of integrity checks. The checking will go from `Layer 1` to `Layer 3` and exit app immediately if the checking fails.

4.3 Summary

The two approaches (anti-tampering and anti-debugging) utilize a secure store to have signatures of those content that are vulnerable to attacks. The IV operation runs dynamically to check the integrity of the protected content. The app anti-tampering approach protects the vulnerable DEX content in an app’s cache to ensure that app
behavior will not be changed at runtime by cache tampering. We also analyze that the cache protection solution can work on the Android 8 (the latest Android version at the time of this writing) with some adjustment. Furthermore, we propose an anti-debugging approach to defend layered check points against being inspected by hooking.
Chapter 5

Evaluation

This chapter presents the evaluation of our app anti-tampering solution and app anti-debugging solution. As we did those experiments in different periods, we describe the evaluation methods and results in different sections.

5.1 Cache Protection

In this section, we evaluate our cache protection approach in terms of effectiveness in fending off a cache tampering attack and the overhead it introduces on the utility of an app. The signing system runs on a server as secure store generation can be done in advance and without synchronizing with the app’s IV operation (Section 5.1.1). The efficiency is measured with respect to the impacts of IV operation on an app’s performance (Section 5.1.2).

The device we use for the evaluation is a Google Nexus 5X phone running Android 7 with kernel version 3.10.73-g43154bf. The build number is NRD90M. Our OAT file format version is 079. Our technique also considers compatibility for OAT different versions. The AOSP building environment in our signing server is Android-7.0.0_r1. Our signing server runs on Ubuntu 15.04 with 250GB hard drive and 4GB memory.
5.1. CACHE PROTECTION

5.1.1 Effectiveness

We make a cache tampering attack in the device to demonstrate the effectiveness of our cache protection mechanism. The attack targets an Android app for the experiments to show that the app’s behavior can be changed through cache modification. The target app is implemented to show the results of adding two numbers. 

TestAdd.java and MainActivity.java are Java source code of the target app. An IV native library and a secure store generated in the host are put in the APK. Once the target app is installed in the device, base.odex is generated in the app’s cache folder /data/app/com.testadd.experiment.testadd-1/oat/arm64/. base.odex is the app’s cache used to boot the app every time when the app’s process is created. The following target app is designed to show “9” on the device’s window view:

Example of the target app

TestAdd.java

    public class testAdd {
        public int add(int a, int b) {
            int c;
            c = a + b;
            return c;
        }
    }

MainActivity.java

    protected void onCreate(..) {
        ...
    }
5.1. CACHE PROTECTION

```java
testAdd t = new testAdd();
TextView tx = new TextView(this);
tx.setText(Integer.toString(t.add(4, 5)));
...}

(The target app for addition)

We tamper with the target app behavior to do multiplication instead of addition by modifying the source code of the method `add` in class `testAdd`. We build the attacking app that uses on-device `dex2oat` to generate the attacking `base.odex`. An `oatparser` working in the host is implemented to change the attacking app’s cache file with the checksum in `OatDexFile` structures obtained from the target app’s cache file. The operation can pass ART cache checksum check. The checksum can also be acquired by calculating CRC32 from `classes*.dex` in `base.apk` that is a copy of the target APK put into the app’s data folder by Android’s PackageManagerService after the target app’s installation. The cache of the modified attacking app replaces the target app’s cache to be `base.odex` in the target app’s cache folder. When the app starts next time (app’s process is re-created), ART will load the tampered cache, which means that the attack will be successfully launched. The window view of device shows “20” after cache tampering.

*Simple modification of the target app*

`TestAdd.java`

```java
public class testAdd {
    public int add(int a, int b) {
        int c;
```
5.1. CACHE PROTECTION

```c
    c = a * b;
    return c;
```

(Modification to do multiplication)

Fig. 5.1 demonstrates the experiments to show the effectiveness of cache protection. The result shows that the target app’s behavior is changed after cache manipulation. We put the attacking app doing multiplication into the device. The malicious `base.odex` will be obtained and then changed with checksum in `OatDexFile` structures of the target app’s `base.odex` in the host. The new `base.odex` will be put into the original app’s cache folder to launch the attack. In our experiments, we use root privilege to manipulate the target app’s cache. The malicious cache will stay effective before system upgrade that replaces all apps’ OAT files.

Cache file tampering attack can be made successfully when the checksums in headers are tampered with carefully. ART will check two kinds of checksums. One is the checksum in `OATHeader` that should be equal to Android framework `boot.oat`’s checksum to make sure that the cache file is created in the device. The other one is in `OatDexFile` structures. These checksums should be the same as the ones calculated through corresponding `classes*.dex` in `/data/app/com.testadd.experiment.testadd-1/base.apk`.

The IV code is in a native library of the target app and it is not easy to reverse and modify for the safety of our protection code. We assume our cache protection code is in a safe place. When the target app starts with the secure store, IV will generate the signature for the modified cache and check if there is a match with the
one in the secure store with the same compiler filter option and instruction set. If the cache IV operation finds that there is no match in the secure store, it will send out an alert shown on the app. Since the target app’s cache is being tampered with, no match will be found in the secure store.

Our signing server and the lightweight app’s cache IV operation are able to anti-tamper and protect the cache from an app’s behavior modification by alerting users about the attack. However, if the target app has to include Google extra libraries like \texttt{com.google.android.maps.jar} or other libraries not included in AOSP, the secure store will not match cache’s signature since AOSP environment does not contain Google extra \texttt{jar} files. We found that both Facebook and Amazon need to insert Google map \texttt{jar} as \texttt{classpath} (an environment path for reference). For a target app inserted with additional Google’s \texttt{jar} files, we suggest to put the \texttt{jar} files into our signing server to get right cache signatures.

5.1.2 Efficiency

A target app puts the secure store into the asset folder and adds one native library to do IV operation. The performance impacts lie in the size of the secure store and
the native library and the execution time of IV operation.

In our experiments, we use the adler32 algorithm to sign each DEX file and generate four bytes for each. Each compiler filter option occupies one byte and there are twelve compiler filter options for \textit{dex2oat} compiler. There are seven kinds of instruction set architectures for mobile devices. It means that the size of a secure store for seven platforms with a specific instruction set is

\[ 7 \times 12 \times (1 + 4 \times n) \]

bytes, where \( n \) is the number of \textit{classes*.dex} in the target app. For example, the size of Facebook APK is nearly 75 MB. The APK has eleven \textit{classes*.dex} files. Our signing system would produce a secure store of 3,780 bytes. The IV native library is 739 KB. They are trivial compared to the size of an app.

In the IV native library, the additional time consumed by IV operation is 20 ms in our experiments with respect to a baseline of the app’s startup scenario. The time is mainly spent on DEX signature generation from an OAT cache. The time for looking up in a secure store is trivial. The adler32 algorithm is used in our experiments. However, more efficient hash algorithms can be explored and applied in our cache protection system.

5.2 Dynamic Anti-debugging

In this section, we evaluate how well our dynamic anti-debugging approach is defeating different hooking attacks in a realistic environment and the overhead it induces on the utility of an app. The performance is measured with respect to the impact of IV operation on an app’s performance.
5.2. DYNAMIC ANTI-DEBUGGING

The environment we use for evaluation is a Nexus 5 with Android 8 and an Android 8 emulator. We would like to have insecure environments (with root privilege) for debugging a released APK. The ROM on Nexus 5 is from [2] with build number OPR6.170623.013, which is an Android 8 AOSP build with root privilege. The OAT file format version is 124. Our technique also considers compatibility for different OAT versions that represent different Android releases.

5.2.1 Applicability

We launch three dynamic method hooking attacks to monitor app behavior. We reuse the source code of the three open-source projects and make adjustments so that they run in Android 8. An app is implemented as the target app that uses the Android APIs listed in the methods.list. We use ptrace to control the target app’s process and inject the hooking library or DEX file into the target app’s process. SELinux has been deployed on Android since Android 5. SELinux stands for Security-Enhanced Linux which has three modes: enforcing, permissive and disabled. Enforcing mode is effective in SELinux policies and restricts the use of ptrace debugging. The permissive mode can be seen as a temporarily disabled SELinux mode. The switch between enforcing and permissive modes is possible with little effort. In order to make ptrace available in Android 8 where SELinux is enforcing, we need root privilege to use setenforce 0, which will change SELinux from Enforcing mode to Permissive mode that allows the use of ptrace.

In Android 7, linking an external library dynamically is not allowed [9]. Therefore, when we use ptrace to inject a native library, the library should be put into an app’s sandbox folder so as to make the Android system believe the injected library is packed
with the app. Besides, the injected library will use `dlopen()` to open `libart.so` and `dlsym()` to find `JNI_GetCreatedJavaVMs` symbol used to get a `JavaVM` structure and attach the current thread into the JVM environment of the process. Both of them will be rejected by the Android system and the injected library will fail to perform its debugging functionality since apps are not allowed to use functions (`dlopen()`, `dlsym()`) that are not included in the Android NDK. We explore another way to get symbols from `libart.so` like searching the library ELF file to find a symbol for a specified name [20].

**Layer 1**

ARTDroid is designed in Android 4.2 [34, 4]. We refer to part of its code to launch the attack in Android 8. The `ArtMethod` class and `Class` class have changed a lot in the latest ART version. The vtable hooking attack process is shown in Fig. 5.2.

![Figure 5.2: Vtable hooking attack](image)

The offset in vtable is obtained through experiments in the real device. The `inject`
5.2. DYNAMIC ANTI-DEBUGGING

The process uses ptrace to control the target app’s process and inject a native library (libvtablehook.so) and DEX content (hookMethods.apk) into the memory of the target app. Ptrace has to find the address of dlopen() and dlsym() in the target app’s memory. Instead of using the relative offset between the function address and the address of /system/bin/linker in the inject process in Android 4.2, we apply the address of /system/lib/libdl.so to find the address of dlopen() and dlsym() in the target app’s memory in Android 8.

```java
public Method getMethod(String name, Class<?>[] parameterTypes) {
    String key = "java/lang/Class.getMethod(Ljava/lang/String;Ljava/lang/Class;)Ljava/lang/reflect/Method;";
    Log.d(NAME_MODULE, "getMethod hooked: "+name);
    Object[] args = new Object[] {name, parameterTypes};
    Method d = (Method)callOriginalMethod(this, key, args);
    return d;
}

public void loadUrl(String url) {
    String key = "android/webkit/WebViewLoadUrl(Ljava/lang/String;)V"; //hash
    Log.d(NAME_MODULE, "loadUrl hooked String: "+url);
    Object[] args = new Object[] {url};
    callOriginalMethod(this, key, args);
}

public static native Object
callOriginalMethod(Object object, String key, Object[] args);
```

Figure 5.3: Vtable hooking methods

Two hooking methods are shown in Fig. 5.3 and implemented in the injected DEX content. The addresses of their ArtMethod classes replace the value of the original
5.2. DYNAMIC ANTI-DEBUGGING

methods in the vtable slots. The original methods’ execution will be diverted to the execution of the hooking methods. An app implements a function to load an APK and get its method using `getMethod()` function. Also, it invokes `loadUrl()` to link a website. After the attack was launched, we did “LOAD URL” and “DEXCLASSLOADER” operations shown in Fig. 5.4.

![Figure 5.4: An experiment app](image)

The hooking result in Fig. 5.5 shows that the attack is effective to debug the behavior of the app under experimentation. Our anti-debugging scheme ran in the background during the attack. As vtable records the addresses of virtual methods in

```
hookclass: getMethod hooked: sayHello
hookclass: loadUrl hooked String: http://www.google.com
```

![Figure 5.5: Vtable hooking result](image)

The hooking result in Fig. 5.5 shows that the attack is effective to debug the behavior of the app under experimentation. Our anti-debugging scheme ran in the background during the attack. As vtable records the addresses of virtual methods in
a Java class, a vtable tampering will tamper with the content of vtable. The Layer 1 check detected the vtable tampering and exited the experiment app.

Layer 2

ProbeDroid claims to support ART instrumentation in Android 5 [22]. It provides hooking framework that users can employ to add interesting components for different debugging purposes. We made some modification and generated a native library libprobedroid.so to do the Layer 2 diverting operation in Android 8. An apk (inspector.apk) is developed to register debugged Android APIs and create a class to implement debugging methods before and after the original methods’ invocation. It uses ptrace to inject the two files into the target app’s memory. The operation is the same as in Fig. 5.2.

![Figure 5.6: ArtMethod EP hooking](image)

The Entry Point (EP) in an ArtMethod class of each Android API will be tampered with to point to a snippet of assembly code (trampoline). When a debugged Android API is invoked, trampoline will be executed and do the debugging work before and after invoking the original method. The principle of the attack is shown in Fig. 5.6. Each debugged Android API has to be registered as in Fig. 5.7.

After the native library and DEX content were injected into the target app’s
memory, we performed the same operation as in Fig. 5.4. The app did not exit immediately after the operation, which was not expected. We checked the value of `entry_point_from_quick_compiled_code_` which should be the address of a method’s native code. We found that all EPs point to the same address. The address resided in the memory range of `libart.so`. We got the offset of the EP’s address in `libart.so` and utilized the Linux `readelf` command to read symbols of `libart.so`. We found that the offset was the location of function `art_quick_to_interpreter_bridge` on both the device and the emulator, which may mean that the Android environment is forced to be in interpreter mode. Therefore, we are unsure since there was no property specifying the makefile options of the system image in the device [13]. Since an `ArtMethod` class’s EP is expected to pointer to a method’s native code, the method’s behavior would change too if the native code changed. However, if the Android environment is forced to be in the interpreter mode, the native code change will not affect the app’s behavior.

```java
nameClass = "android/webkit/WebView";
nameMethod = "loadUrl";
signatureMethod = "(Ljava/lang/String;)V";
LoadUrl loadUrl = new LoadUrl(true, false);

public class loadUrl extends MethodBundle {
    @Override
    public void beforeMethodExecute(Object[] objects)
    {
        String url = (String) objects[0];
    }
    public void afterMethodExecute(Object o)
    {..}
}
```

Figure 5.7: `loadUrl()` registration in inspector.apk
Layer 3

ARTIST provides an instrumentation framework to hook a method by repackaging the target APK, attaching the framework into the app and thus gathering statistics for the compiled methods of APK and Android framework libraries [38] when the app runs. It claims to run in Android 5.1.1. The OAT structure and files generated by 

\textit{dex2oat} compiler have changed a lot in the latest version. We utilized the source code of ARTIST and aligned the OAT layout with Android 8 [6]. We injected \textit{libartist.so} into the target app’s memory by ptrace. The Nexus 5 is based on Thumb instruction set; therefore, we tampered with the first two bytes of the native code’s address. After doing the operation in Fig. 5.4, we found that the app did not exit. That may explain the ineffectiveness of the \textit{Layer 2} attack. The Android 8 ROM and emulator are forced in interpreter mode. Hence, the \textit{Layer 3} attack will have no effect on app behavior too.

Our dynamic anti-debugging method can defend app’s privacy even though the insecure environment is in interpreter mode and the \textit{Layer 2} attack and the \textit{Layer 3} attack behave differently in Android 8.

5.2.2 Efficiency

An app develop has to provide a file named \textit{methods.list} to record Android APIs an app would like to protect to prevent the app’s behavior from being leaked. A native library is provided to integrate into the app to perform the anti-debugging technique. The size of \textit{methods.list} depends on the number of protected Android APIs and is trivial to count. The size of our native library is less than 600 KB. The size of the secure store generated at runtime is small.
5.3. SUMMARY

The other impacts our method imposes on the target app are the time for a secure store generation and the time for integrity checks. Since secure store generation is triggered when a device boots up, the time will not impact an app at runtime. We separate the time for different operations. In our experiments, we protect 9 Android APIs in the methods.list. The time distribution of checking 9 methods is shown in Table 5.1.

Table 5.1: Time consumption for each operation in integrity checks

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>methods.list analysis</td>
<td>2</td>
</tr>
<tr>
<td>Layer 1</td>
<td>1</td>
</tr>
<tr>
<td>Layer 2</td>
<td>1</td>
</tr>
<tr>
<td>/proc/self/maps analysis</td>
<td>40</td>
</tr>
<tr>
<td>Layer 3</td>
<td>10</td>
</tr>
</tbody>
</table>

The integrity checks will run in the background when the target app starts. methods.list analysis and memory range (/proc/self/maps) analysis will happen once when the app process is created. For integrity checks on Layer 1 and Layer 2, each method consumes less than 1 ms. The time for the integrity check on Layer 3 varies: loadUrl() check consumes 4 ms, while others take 1 ms on average. The IV time depends on the number of protected methods. The frequency of IV is decided by app developers. As the check time for each layer is trivial, the impact to the target app is little.

5.3 Summary

We did these experiments to demonstrate that cache modification can change app behavior and process hijacking in hooking attacks can expose app information. The
experiments for app anti-tampering and app anti-debugging protection were done in different periods on the latest Android versions available at that time. Our anti-tampering approach can protect app cache against being tampered with, thus protecting app behavior. Even though the evaluation results show that both the Layer 2 attack and the Layer 3 attack cannot be launched successfully in the interpreter mode based Android platform, our anti-debugging approach can still effectively prevent the Layer 1 debugging by Android instrumentation toolkits in an attacker-friendly environment. Both approaches have a trivial impact on app performance based on our experiments.
Chapter 6

Conclusion and Future Work

6.1 Conclusion

The importance of protecting the integrity of app behavior and app privacy is rising with growing cases of new attacks. We analyze those attacks and design targeted app protection approach. In this thesis, we assume that the static APK file is well packed or obfuscated to prevent app from being tampered with (repackaged) and being debugged statically. Therefore, we protect app cache to prevent app from being tampered with after an app’s installation instead of protecting an app’s APK file. Since the target app is assumed to be anti-debugging statically because the app packing techniques have been applied, we propose an approach to anti-debugging an app dynamically. We summarize the two techniques in Section 6.1.1 and Section 6.1.2 respectively. We will combine the cache protection approach with the anti-debugging technique to form a comprehensive protection solution for Android apps.
6.1. CONCLUSION

6.1.1 App Cache Anti-tampering

We propose to mitigate the risk of the exploitation of ART cache mechanism and thus defend app behavior integrity. We know that an app’s cache is an executable file loaded into memory after the app is installed. A cache tampering attack can modify cache to change an app’s behavior when the app’s process restarts. Our solution is able to prevent a legitimate app’s behavior from being tampered with by protecting the vulnerable part of the app’s cache. In this thesis, we conduct a systematic investigation about an app’s cache by analyzing ART cache loading process and cache structure, and by assessing the feasibility of signing an app’s classes*.dex in the host.

We implement a lightweight and app-level cache protection mechanism against cache tampering. We deploy the time-consuming compilation process in the host and implement an IV native library to defend app behavior integrity in the device. The signing host applies different compiler filter options to generate secure stores for apps. The host has to insert extra Google libraries to make sure that right secure stores can be generated if target apps need additional Google libraries that do not exist in AOSP. Furthermore, our experimental results show defense effectiveness and efficiency of our proposed approach. The cache protection system is compatible with most of the recent Android versions (5 to 8). Our cache protection technique is able to defend the integrity of app behavior by alerting users about cache tampering attacks.

6.1.2 Dynamic Anti-debugging

Software analysis techniques are largely used to debug app for aiding software development or monitor malware and expose malicious execution. They may be exploited
to monitor benign apps, which is detrimental to app privacy defense. We propose a dynamic app anti-debugging method to mitigate the risk of being analyzed and thus defend app privacy. Through the analysis of current open-source debugging framework tool kits on ART, we summarize the debugging points into three layers. Our approach protects the value of the check points in three layers and perform anti-debugging by verifying their integrity at runtime.

We implement a lightweight and app-level anti-debugging mechanism against Android runtime instrumentation. Though the secure store generation and integrity checks run in the device, our experimental results show its applicability and efficiency. Our approach is compatible with most of the recent Android versions (5 to 8).

6.2 Limitations and Future Work

The attacks that our app protection techniques can defeat need the root privilege to launch. The assumption to some extent diminishes the generality of our work. The next stage is to investigate attacks launched on unrooted devices and propose app protection approaches to defeat them.

Our research about providing a complete Android app protection solution is conducted under the assumption that attackers may fail to reverse engineer the static APK file because of app packing techniques. The next step of our research is to delve into app bytecode protection in the APK file.
Bibliography


