Defending Application Cache Integrity of Android Runtime

Jia Wan[1] *, Mohammad Zulkernine[1], Phil Eisen[2], and Clifford Liem[2]

[1] School of Computing, Queen's University, Kingston, ON, Canada
{jiawan,mzulker}@cs.queensu.ca
[2] Irdeto Corporation, Ottawa, ON, Canada
{phil.eisen,clifford.liem}@irdeto.com

Abstract. Android malware vendors profit by "piggybacking" on legitimate applications (or simply apps) and inserting malicious code that can steal users' sensitive data or display unsolicited advertisements. A piggybacked app is a repackaged legitimate app with extra code that can perform malicious acts after installation. Many researchers have put effort into signature schemes for malware detection and to develop obfuscation techniques to mitigate the effects of piggybacking. However, little has been done to protect apps after their installation. In particular, the cache, where the app actually runs, is vulnerable to tampering. Cache tampering allows for the same behavioral changes as piggybacking. Cache loading process of Android Runtime (ART) can be exploited by cache tampering attacks without rebooting the device. In this paper, we introduce an approach to protect apps by maintaining the integrity of their cache. We show that cache tampering is possible and propose a lightweight cache protection mechanism to alert users about a cache tampering attack. We describe the approach in detail and present the results of a real implementation. Our evaluation results on Android 7 (the latest version at the time of this writing) show that our cache protection system can detect the abnormal behavior effectively and efficiently.

Keywords: Android Application, ART, Anti-Tampering, Mobile Security

1 Introduction

Android devices remain an attractive mobile malware target in recent years [31]. Piggybacked apps are popular in third party app markets where legitimate apps are repackaged and leveraged to make profit for attackers [1]. Some research have been done to get apps repackaged for policy enforcement or software analysis [7, 8], 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 [14–17]. As attackers become more stealthier, an app's cache may be tampered to perform

* The corresponding author.

the same malice as a repackaged app by exploiting the vulnerabilities in Android cache mechanism [21].

Dalvik has been replaced by ART since Android 5. ART is now the default Android Java Virtual Machine (JVM) for Android 5 and the higher versions [23]. During an app's installation, *installd* triggers *dex2oat* compiler to create app cache in Optimized ART (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 up next time, ART will load the app's cache file into the memory of its own process rather than the reinstallation of the Android Package (APK) 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.

The reliance on app cache to load an app in ART 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. The replacement of an app's copied APK file (`base.apk` in the app's sandbox folder) is only effective after rebooting the device and then Android framework brings notice to its user's attention. The 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 should break into the app's sandbox to modify the cache. 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 user's account information in the background. Moreover, hardened apps are made through packing services to make reverse engineering more difficult by applying dynamic loading. However, the app's cache can be generated once original Dalvik EXecutable (DEX) format file is loaded dynamically [6]. Therefore, cache protection in ART is necessary.

An OAT file is large in size, e.g., YouTube's cache file is nearly 253MB 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 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.

In this paper, we attempt to provide an anti-tampering solution for apps by protecting app cache. 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. Our solution can be deployed easily across different Android ART-based platforms with little effort. App developers are able to use our technique to protect the integrity of their apps' behavior. In summary, we make the following contributions:

- 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. Our findings inspire the research of Android cache protection.
- We propose a defense mechanism for an app's cache protection to defend installed apps' behavior. We launch a cache tampering attack to exploit the vulnerability of ART's cache mechanism that can be leveraged to tamper the target app's behavior. The attack is used to assess the effectiveness of our proposed mechanism.
- We implement a lightweight Integrity Verification (IV) shared library integrated into the target app in the device running Android 7 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.

The rest of the paper is organized as follows. In Section 2, we discuss background information about ART cache in Android 7. Section 3 introduces a general overview of how a cache tampering attack can circumvent ART cache check and do malice in the device. We detail our proposed system in Section 4 including the implementation in both host and device. Section 5 presents the evaluation results. After introducing the related work in Section 6, we conclude in Section 7.

2 Background

Android introduced ART in Android 4 as an option and set it as default Android runtime to execute Dalvik bytecode in the later releases of Android. ART aims to improve the execution performance of Android Java apps by executing native code from `base.odex` instead of interpreting Dalvik bytecode. 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 [24]. The profile guided compilation will store frequently executed methods into an app's image file (`base.art`), which avoids JIT compilation again [22, 25]. The

interpreter interprets Dalvik byte code for execution without consuming time for compilation.

In this section, we provide background information about Android cache mechanism. Section 2.1 includes `base.odex` and `base.oat` loading procedure. OAT content varies among different Android versions. We illustrate OAT file structure in Android 7 in Section 2.2. Section 2.3 describes how *–compiler-filter* option for *dex2oat* influences the formation of OAT content.

2.1 Cache File

The cache file is a special Executable and Linkable Format (ELF) file with 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 [32]. 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. 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 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 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 `base.odex`. ART will load the modified app cache

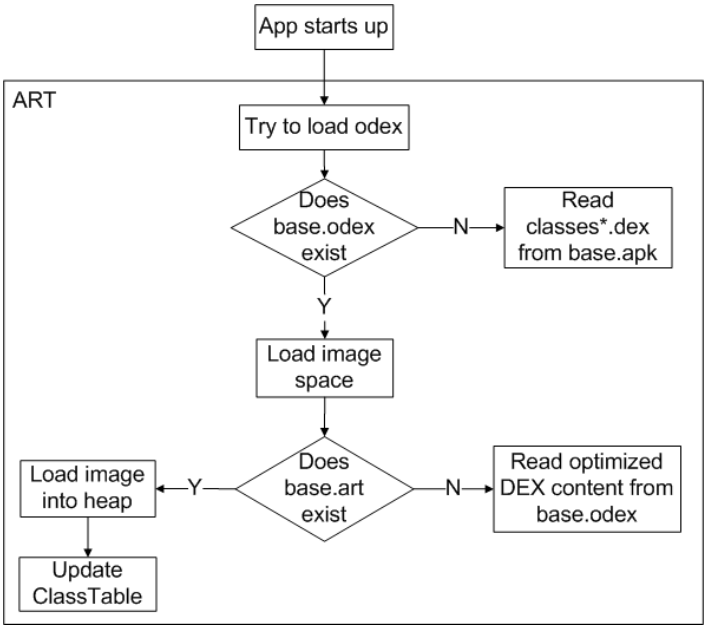

Fig. 1. Caching loading in ART

(`base.odex`) into the memory of the app's process, which may modify the target app's behavior.

2.2 OAT Structure

As Dalvik is the JVM on older Android releases, app cache contains optimized DEX content. Now ART introduces OAT structure embedded in an ELF cache file. Fig. 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: *OATHeader*, *OatDexFile*, *DexFile*, and *OatClass*. *OATHeader* 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 *OATHeader* specifies the checksum of the current *OATHeader* 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. `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 [26], 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.

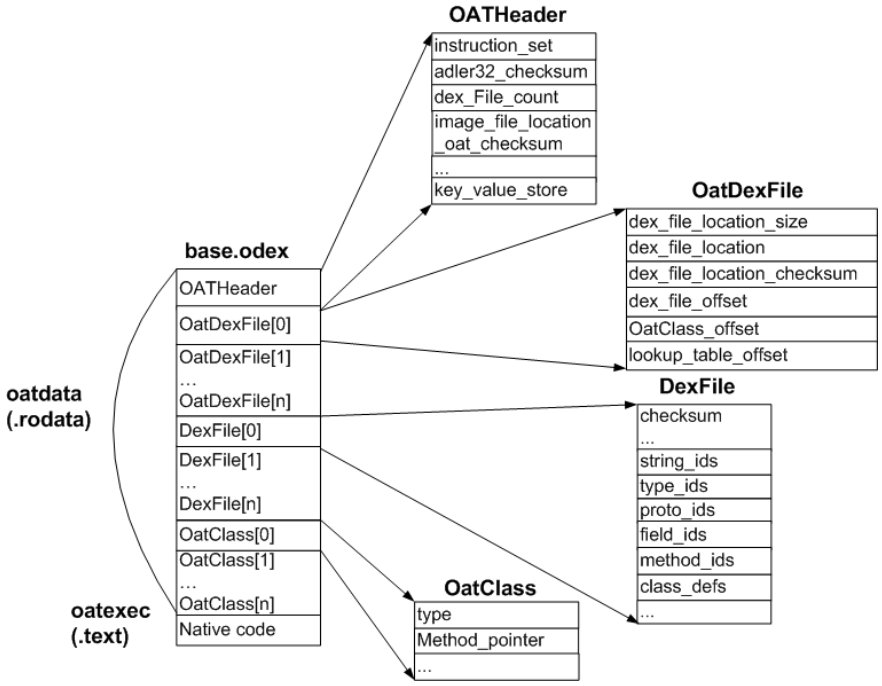

Fig. 2. App cache layout in ART

DexFile is the optimized content of *classes*.dex* [27]. 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 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 status of the class. There are three compilations statuses: 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 compiled method index to locate native code. *method_pointer* records the offsets of methods' native code in *oatexec* segment.

2.3 Compiler Filters

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 have to be considered like AOT compilation consumes too much time during an app's installation and a mobile device may be short of space to store large cache files with 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 [22]. This will be discussed in Section 4.2. There are two categories in terms of compilation options in Android 7: one is for system image configuration and the other is about app compilation. In this paper, we only discuss app compilation category that is *–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 [22, 27].

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 paper.

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.

3 Threat Model

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 shows the cache loading check in ART.

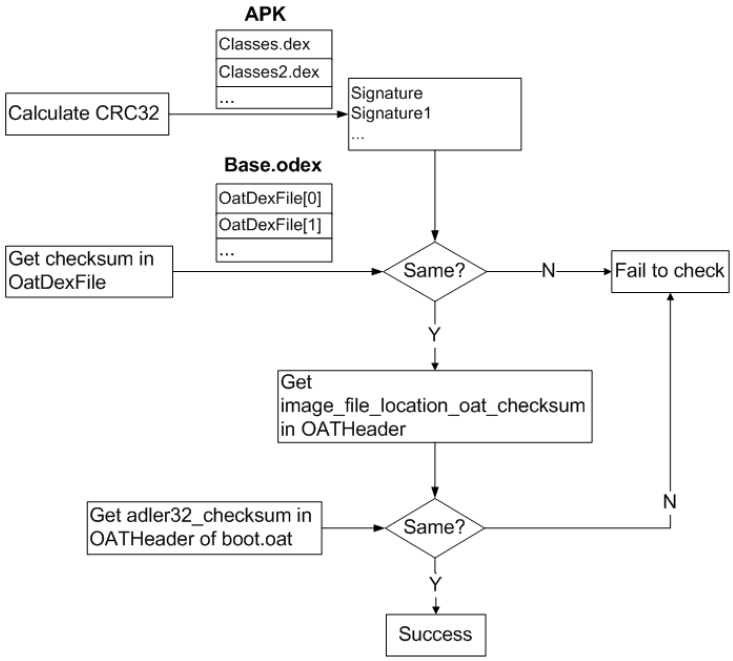

Fig. 3. Cache loading check in ART

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 extracted to compare with `adler32_checksum` in *OATHeader* of on-device `boot.oat` (Android framework's cache). This operation ensures that the cache file is generated in the device. If the check passes, `base.odex` is legitimate.

In Section 2.3, we mention that DEX-to-DEX optimization may change DEX content in an OAT file by *dex2oat*. However, checksums in `base.odex`'s *OatDexFile*s still keep the CRC32s of the original *classes*.dex* in the APK.

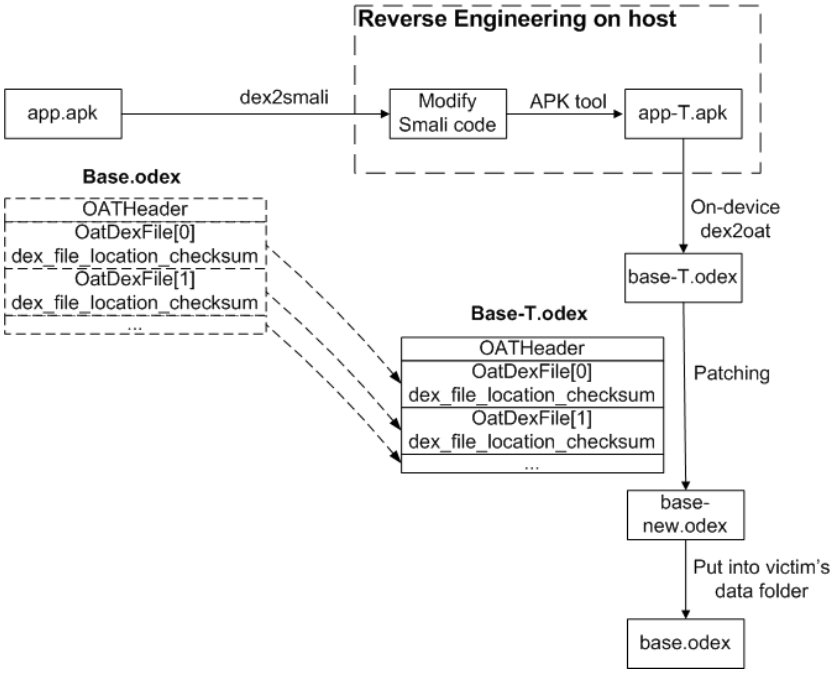

Fig. 4. Cache tampering attack operation

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. 4. 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 `app-T.apk` by APK tool. `base-T.odex` is generated by on-device *dex2oat* to get the right `adler32_checksum` from `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 *dex2oat* with necessary framework *jar*s in the right Android version. If *dex2oat* compiler is operated in the host, `image_file_location_oat_checksum` of *OATHeader* in the newly built `base-T.odex` should be replaced by the value of `adler32_checksum` in *OATHeader* of the target device's `boot.oat`. The next step is to modify `base-T.odex` with the original `base.odex`'s checksum of *OatDexFile*. At last, `base-T.odex` is put into the app's cache folder to replace 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.

4 Cache Protection Approach

As the vulnerable cache exploitation is demonstrated in Section 3, 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 such a technology that protects app behavior integrity by anti-tampering the app's cache. Section 4.1 introduces the basic concept of our proposal. We describe the compatibility for the next version of Android 8 in Section 4.2. Section 4.3 and Section 4.4 elaborate our cache protection system implemented in both host and device.

4.1 Basic Idea

We assume that if an attacker can tamper 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 is different from the original cache. The difference happens in the DEX content of OAT structure. Our design goal is to get the user aware of cache tampering by making 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 do not impact an app's runtime performance too much.

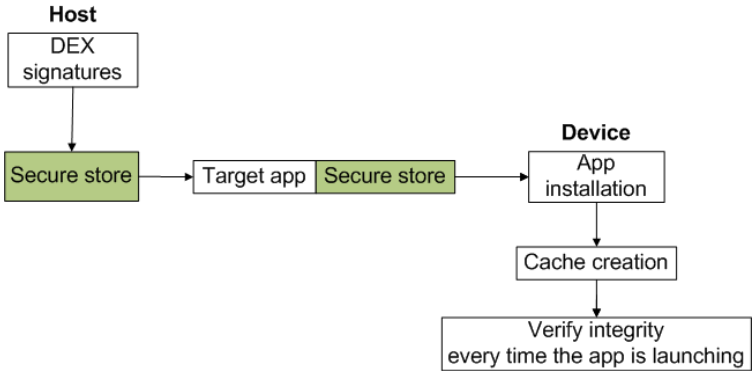

Fig. 5. App cache anti-tampering

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 IV every time the app starts up. The idea is shown in Fig. 5. The cache is `base.odex`, an OAT file which contains optimized DEX content [27].

App developers may apply app hardening technology to load sources at runtime. Packing services adopt special *ClassLoader* to dynamically load APK to assure that attackers may not take advantage of the APK file. However, ART may generate cache file for the loadee when the protected app hardened inside a shell is loaded [6]. Hence, cache file protection is still needed.

4.2 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*, while *.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 [22]. Our protection technique may be compatible with future Android releases.

4.3 Secure Store Generation

A signing system shown in Fig. 6 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. 7 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 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

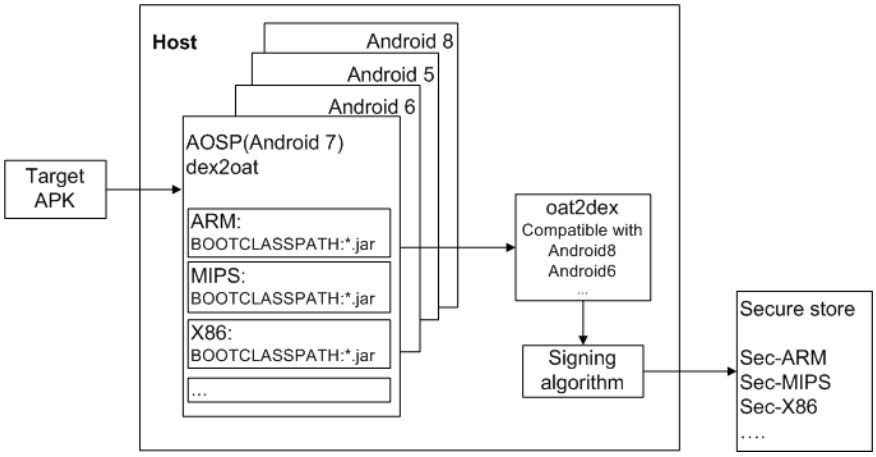

Fig. 6. Signing components in the host

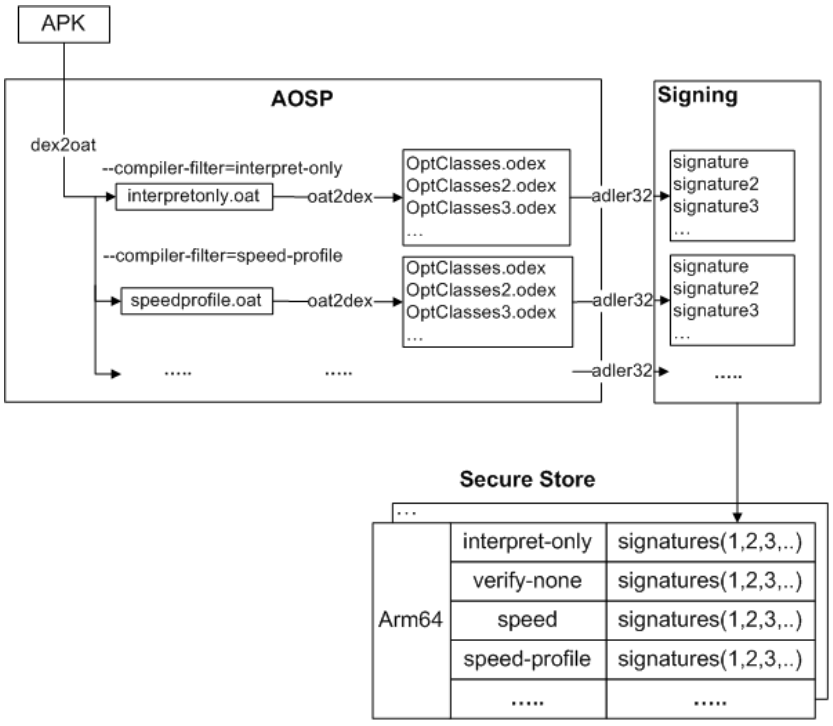

Fig. 7. Secure store generation process in the host

original *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.

4.4 Integrity Verification

Fig. 8 illustrates an app's cache IV process. When an app is installed, Android *installd* 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 and replaced by malicious one, IV native library will check and send an alert.

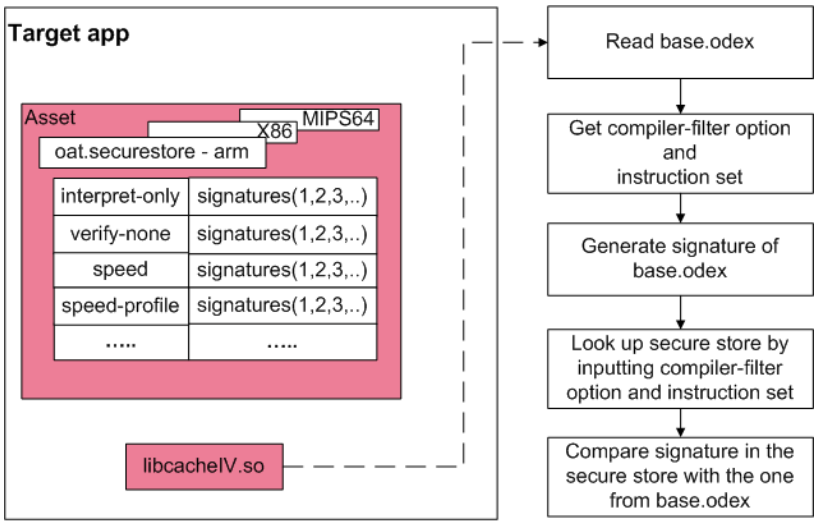

Fig. 8. Cache file IV process in the device

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 to the app owner to be integrated into the app. Both files will not change DEX content of the app. The app will be released with a secure store and an IV library to perform cache IV after the app is installed.

5 Evaluation

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). The efficiency is measured with respect to the impacts of IV operation on an app's performance (Section 5.2).

The device we use for the evaluation is a Google Nexus 5X phone running Android version 7.0 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 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(..) {
    ...
    testAdd t = new testAdd();
    TextView tx = new TextView(this);
    tx.setText(Integer.toString(t.add(4, 5)));
    ...
  }
```

(The target app for addition)

We tamper 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
  public class testAdd {
    public int add(int a, int b) {
        int c;
        c = a * b;
        return c;
    }
  }
```

(Modification to do multiplication)

Fig. 9 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 carefully. ART will check two kinds of checksums. One

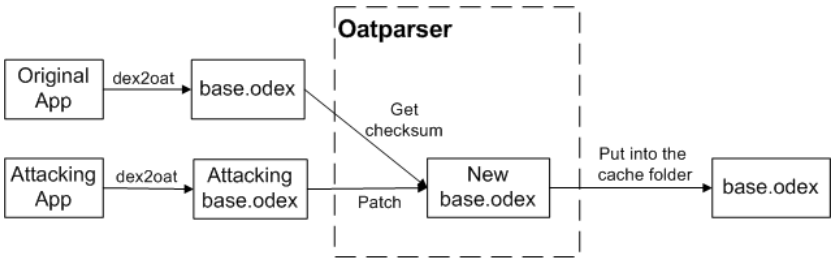

Fig. 9. Experimental cache tampering attack launching process

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 tampered 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 tampered, 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 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 `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 *jar* files. We found that both Facebook and Amazon need to insert Google map *jar* as `classpath` (an environment path for reference). For a target app inserted with additional Google's *jar* files, we suggest to put the *jar* files into our signing server to get right cache signatures.

5.2 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 adler32 algorithm to sign each DEX file and get four bytes for each. Each compiler filter option occupies one byte and there are twelve compiler filter options for *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

$$7 * 12 * (1 + 4 * n)$$

bytes, where n is the number of *classes*.dex* in the target app. For example, the size of Facebook APK is nearly 75 MB. The APK has eleven *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.

6 Related Work

Finley et al. [10] 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 about which we are concerned. In our paper, an app's cache acts as an app's execution file once the app is installed. An app's cache contains an app's Dalvik bytecode and executable instructions.

Sabanal [21] demonstrated the possibility of replacing 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 ART cache. Reference hijacking [7] 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 [18].

Schulz [11] proposed obfuscation techniques to build apps that need attackers' more effort to analyze and piggyback apps [12]. Jeong et al. [13] 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, while 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 [2, 6, 19]. 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 the legitimate app is running in the device.

Some recent work proposed to instrument ART for apps' monitoring. Costamagna et al. [5] proposed a runtime injection approach in ART to monitor app

behavior but the scope of monitoring is limited. ARTDroid diverts the execution of sensitive Android APIs for an app's behavior monitoring by method pointer replacement in class virtual table, while the app's exclusive methods are obfuscated and cannot be easily tampered. ProbeDroid achieves the same result by using *ptrace* mechanism (Linux Process Trace) to inject code into a target app's process and change the entry point of methods in the tracked app [30]. Dresel et al. proposed an instrumentation framework in ART to get the locations of Java classes and methods that are helpful for an attacker to control methods and divert their execution. However, ARTIST [3] needs to inject code into an app and repackage the app, which breaks the integrity of the original app and can be detected by our cache protection technique since repackaging the target app changes the app's cache. Backes et al. modified *dex2oat* and added compilation instrumentation for ART backend compiler, which aims to track an app's runtime execution footprint. ARTist [4] replaces *dex2oat* with the optimized one and updates the instrumented app's cache, but our cache protection may not detect such changes since backend compiler optimization impacts native code generation instead of DEX content. These technologies can be applied to tamper apps' runtime behavior. The runtime instrumentation research has influenced our work to prevent runtime methods from being tampered and employ runtime protection of methods.

We develop *oat2dex* for extracting DEX content from an OAT file in Android 7. Note that Chao [28] provides DEX content extraction from an OAT structure, but it lacks the support in OAT analysis in Android 7.

7 Conclusion

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 by protecting the vulnerable part of the app's cache. In this paper, 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 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 by alerting users about cache tampering attacks. However, if an attack injects malicious code into an app's memory to control the app's methods in ART and diverts methods' execution to malicious code, malice would be done once a tampered method is invoked without restarting the app's process [5, 4, 30]. For a complete Android app anti-tampering design, defending app runtime behavior is necessary. We will extend the app protection mechanism to handle the tampering of methods at runtime.

Acknowledgments. This project is partially funded by Mitacs Canada and Irdeto Corporation.

References

1. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 95-109. IEEE (2012)
2. Yu, R.: Android packers: facing the challenges, building solutions. In: Proceedings of the Virus Bulletin Conference (VB'14), pp. 266-275 (2014)
3. Dresel, L., Protsenko, M., Mller, T.: ARTIST: The Android Runtime Instrumentation Toolkit. In: 2016 11th International Conference on Availability, Reliability and Security (ARES), pp. 107-116. IEEE (2016)
4. Backes, M., Bugiel, S., Schranz, O., von Styp-Rekowsky, P., Weisgerber, S.: ARTist: The Android runtime instrumentation and security toolkit. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 481-495. IEEE (2017)
5. Costamagna, V., Zheng, C.: ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime. In: IMPS@ ESSoS, pp. 20-28 (2016)
6. Zhang, Y., Luo, X., Yin, H.: Dexhunter: toward extracting hidden code from packed android applications. In: European Symposium on Research in Computer Security, pp. 293-311. Springer, Cham (2015)
7. You, W., Liang, B., Shi, W., Zhu, S., Wang, P., Xie, S., Zhang, X.: Reference hijacking: Patching, protecting and analyzing on unmodified and non-rooted android devices. In: Proceedings of the 38th International Conference on Software Engineering, pp. 959-970. ACM (2016)
8. Davis, B., Chen, H.: RetroSkeleton: retrofitting android apps. In: Proceeding of the 11th annual international conference on Mobile systems, applications, and services, pp. 181-192. ACM (2013)
9. Han, J., Yan, Q., Gao, D., Zhou, J., DENG, H. R.: Android or iOS for better privacy protection?. In: International Conference on Secure Knowledge Mangagement in Big-data era (SKM 2014) (2014)
10. Finley, S., Du, X.: Dynamic cache cleaning on Android. In: 2013 IEEE International Conference on Communications (ICC), pp. 6143-6147. IEEE (2013)
11. Schulz, P.: Code protection in android. In: Insititute of Computer Science, Rheinische Friedrich-Wilhelms-Universitgt Bonn, Germany, 110 (2012)
12. Bichsel, B., Raychev, V., Tsankov, P., Vechev, M.: Statistical deobfuscation of Android applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 343-355. ACM (2016)
13. Jeong, Y. S., Park, Y. U., Moon, J. C., Cho, S. J., Kim, D., Park, M.: An anti-piracy mechanism based on class separation and dynamic loading for android applications. In: Proceedings of the 2012 ACM Research in Applied Computation Symposium, pp. 328-332. ACM (2012)

14. Kywe, S. M., Li, Y., Hong, J., Yao, C.: Dissecting developer policy violating apps: characterization and detection. In: 2016 11th International Conference on Malicious and Unwanted Software (MALWARE), pp. 1-10. IEEE (2016)
15. Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P., Blasco, J.: Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. In: Malicious and Unwanted Software (MALWARE), Expert Systems with Applications, 41.4: 1104-1117 (2014)
16. Zhang, M., Duan, Y., Yin, H., Zhao, Z.: Semantics-aware android malware classification using weighted contextual api dependency graphs. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1105-1116. ACM (2014)
17. Deshotels, L., Notani, V., Lakhotia, A.: Droidlegacy: Automated familial classification of android malware. In: Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014, p. 3. ACM (2014)
18. Li, L., Li, D., Bissyand, T.F., Klein, J., Traon, Y.L., Lo, D., Cavallaro, L.: Understanding Android app piggybacking. In: Proceedings of the 39th International Conference on Software Engineering Companion, pp. 359-361. IEEE Press (2017)
19. Xue, L., Luo, X., Yu, L., Wang, S., Wu, D.: Adaptive unpacking of Android apps. In: Proceedings of the 39th International Conference on Software Engineering, pp. 358-369. IEEE Press (2017)
20. Cheng, B., Buzbee, B.: A jit compiler for androids dalvik vm. In: Google I/O developer conference, vol. 201, no. 0. (2010)
21. Sabanal, P.: Hiding behind ART. IBM. `https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART.pdf` (Accessed 4 August 2017)
22. Google Inc.: Configuring ART. `https://source.android.com/devices/tech/dalvik/configure` (Accessed 4 August 2017)
23. Google Inc.: Android 5.0 Behavior Changes. `https://developer.android.com/guide/practices/verifying-apps-art.html` (Accessed 4 August 2017)
24. Google Inc.: Android 7.0 for Developers. `https://developer.android.com/about/versions/nougat/android-7.0.html`(Accessed 4 August 2017)
25. Google Inc.: Implementing ART Just-In-Time (JIT) Compiler. `https://source.android.com/devices/tech/dalvik/jit-compiler` (Accessed 4 August 2017)
26. Google Inc.: Configure Apps with Over 64K Methods. `https://developer.android.com/studio/build/multidex.html`(Accessed 4 August 2017)
27. Github.: DEX-to-DEX Optimisations. `https://github.com/anestisb/oatdump_plus#dex-to-dex-optimisations` (Accessed 4 August 2017)
28. Github.: Oat2dex. `https://github.com/lollipopgood/oat2dex` (Accessed 4 August 2017)
29. Dalvik and ART. `http://newandroidbook.com/files/Andevcon-ART.pdf` (Accessed 4 August 2017)
30. Github.: ProbeDroid. `https://github.com/ZSShen/ProbeDroid` (Accessed 4 August 2017)
31. Symantec.: Internet Security Threat ReportInternet ReportVOLUME. `https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf` (Accessed 4 August 2017)
32. Zhong, X.: ART JIT in Android N. `http://connect.linaro.org/resource/las16/las16-201/` (Accessed 4 August 2017)