Comparative Analysis of Mobile App Reverse Engineering Methods on Dalvik and ART

Geonbae Na, Jongsu Lim, Kyoungmin Kim, and Jeong Hyun Yi*
Soongsil University, Seoul, 06978, Korea
{nagb, jongsu253, mseckkm, jhyi}@ssu.ac.kr

Abstract

The runtime system for the Android platform has changed to ART. ART differs from previously used Dalvik in that it is to be a runtime environment for the application's machine code. As a result, ART does not execute Dalvik bytecode through an interpreter but executes the machine code itself, leading to high performance and many other benefits. This change in runtime system also has many implications for mobile security. While we can anticipate with certainty the resurgence of modified malicious activity or malicious applications previously used with Dalvik or the emergence of completely new structures of malicious techniques, we can no longer ascertain the feasibility of the analysis techniques and analysis tools used against these malicious applications that operated in Dalvik. To combat future potential malicious techniques for ART, we must first have a clear understanding of ART and, with this foundation, to effectively and accurately utilize the correct analysis technique. Thus, this paper serves to introduce an analysis on the operating method and architecture of ART and, based on this information, address the executable feasibility of the analysis techniques in ART. Furthermore, we present the test results of running these analysis tools and techniques in ART.

Keywords: Android runtime, reverse engineering, dynamic analysis

1 Introduction

The Android platform has been using Dalvik Virtual Machine (Dalvik) as the runtime environment but, due to limitations in performance from of its structural characteristic of using a virtual machine and other complex factors such as hardware resource performance improvement, has begun to use a new runtime environment called ART (Android runtime) instead of Dalvik starting from Version 5.0. The architecture of Dalvik uses the interpreter method to run the application based on Dalvik bytecode while ART uses the compiler method to run the application using machine code obtained through ahead-of-time complication of the Dalvik bytecode.

We are at a point where, following this change in runtime environment, a preliminary examination of the new needs and limitations of the existing android application analysis techniques and tools is needed. As of now, because versions 4.1-4.3 (Jellybean) and 4.4 (Kitkat), which use Dalvik, take up over 50% of the Android platform, the existing dynamic analysis tools used in Dalvik can be used for reverse engineering. However, in the soon to be upgraded versions 5.x (Lollipop) and version 6.0 (Marshmallow), most of the applications are run in ART.

As a result, there may be cases in the future where vulnerabilities or the structural characteristics in ART can be manipulated so that a code runs as normal code in Dalvik but as malicious code in ART. In this case, it will prove difficult to use as is the dynamic analysis tools based on QEMU (Quick Emulator)[7] which heavily relies on Dalvik for code analysis.

*Corresponding author: School of Software, Soongsil University, Seoul, 06978, Korea, Tel: +82-2-828-7360

Since its advent, Android has been the main target of continuous malicious behavior such as privilege escalation attack[13], information leakage[16], rootkit[11], etc. Furthemore, the number of malicious code targeting Androids will not soon diminish, and with the runtime system currently changing, there is the possibility of a resurgence of mobile malicious applications that had been previously weeded out in addition to the emergence of malicious applications with new techniques. It is most likely that developers of malicious applications will discover and abuse vulnerabilities in the new runtime system and bypass the existing application analysis tools that are suited for Dalvik to induce false positives and false negatives.

In response, to address ahead of time mobile malicious code that is soon to appear, this paper sets out to examine the operational method and architecture of ART through source code analysis. We analyze and confirm through running experiments whether the various static and dynamic analysis schemes used in Dalvik can effectively run in the ART environment. Furthermore, we introduce the latest ART based malicious code analysis techniques and tools.

This paper is organized as follows. In Section 2, we analyze the architecture of the ART system and, in Sections 3 and 4, we explain each of the existing static analysis techniques and dynamic analysis techniques and the feasibility of using these technique in ART. Section 5 presents the experiment results of running static and dynamic analysis in the ART environment. Lastly, we make our conclusion in Section 6.

2 Android Runtime

In this section, we examine the ART based operating structure that has been applied since Android version 4.4 (KitKat) and compare it to Dalvik to analyze the differences in-between. While ART was experimentally applied to Android version 4.4 (KitKat), under that version the default runtime system was still Dalvik, and ART could be selected as the runtime system as an option for the developer. Afterwards, starting from Android version 5.0 (Lollipop), Dalvik was completely gone, and ART began to be used as the default runtime system.

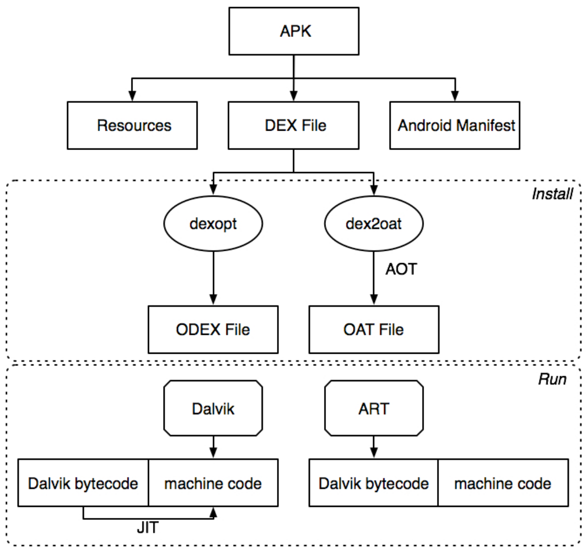

Figure 1: Comparison on installation and run methods of Android applications

With the change in runtime system, the biggest changes in application installation and execution times are shown in Figure 1. In the case of Dalvik, the DEX file in the distributed APK was optimized using the dexopt tool and the ODEX file generated, while for ART, the DEX file is compiled using the dex2oat tool and the OAT file is generated. The ODEX file is the actual executable file with Dalvik and, while almost identical to the original DEX file, is partially optimized so that opcode is partly changed. On the other hand, the OAT file in ART is a file in ELF format, completely different from the original DEX file, and includes the machine code generated through the compilation process.

2.1 Dex File Format

Because the format of the input file for both Dalvik and ART is DEX, in this section we provide a brief overview of the DEX file structure. The DEX file is composed of many various sections, as shown in Figure 2.

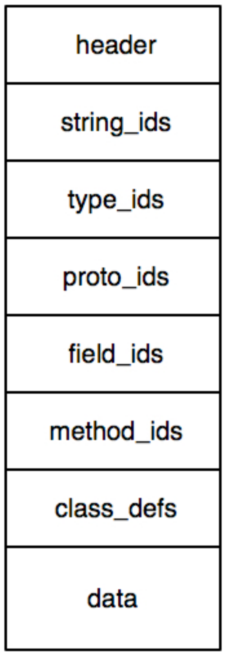

Figure 2: DEX File Format

The header section contains the overall information of the DEX file while the `strings_ids` section contains all the file's strings. All the information regarding type is included in the `type_ids` section, and all the information regarding prototype is in the `proto_ids` section. Information regarding the field, method, and class is in the `field_ids`, `method_ids`, `class_defs` sections respectively. Lastly, the `data` section, the constant pool of the java class file compiled by javac, contains information such as the string or method name.

2.2 OAT File Format

The OAT file, a file compiled through AOT (Ahead-of-Time) compilation, is an execution file that replaces the ODEX file for Dalvik.

In the case of ART, while an application is being installed, it goes through the AOT compilation process where the DEX file is converted into the OAT file using `dex2oat` within the device. More precisely, the file generated using `dex2oat` is not the OAT file itself but the ELF formatted file that includes the

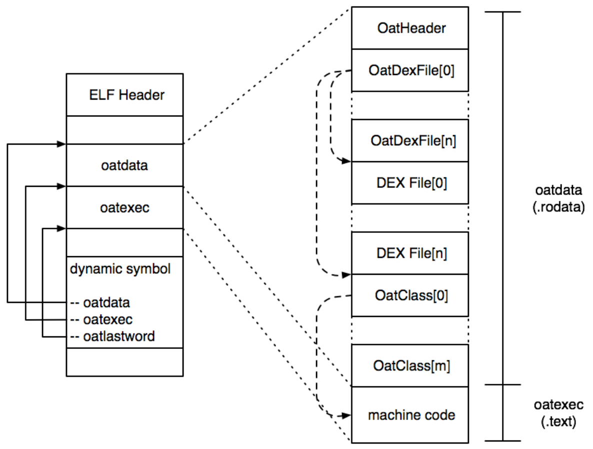

Figure 3: OAT File Format

compiled OAT file[9]. This file, shown in the left side of Figure 3, is comprised of ELF Header, and the `oatdata` and `oatexec` sections[14][15], while the three symbols oatdata, oatexec, oatlastword exist in the dynamic symbol table while each points to the beginning of the `oatdata` section, the beginning of the `oatexec` section, and the end of the `oatexec` section respectively (See Figure 4).

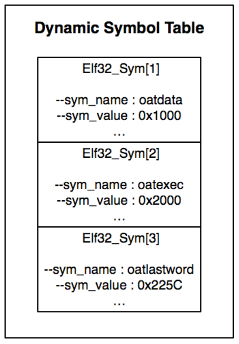

Figure 4: Dynamic Symbol Table

`OatHeader`: The OAT file, made of the oatdata section and oatexec section, has its own header. `OatHeader` contains the magic value, file version information, OatHeader's alder-32 checksum, DEX file count, instruction set, and the offset information of the `oatexec` section. The `oatexec` section includes the machine code compiled based on the Dalvik bytecode.

The OAT file version for Android 6.0 (Marshmallow) is Version 064 and for Android 5.1 (Lollipop MR1) is Version 045. With no more portable mode from Android 6.0 onward, there have also been changes in `OatHeader`. As can be seen in Figure 5, there are fields relating to portable mode in the current Version 045 such as `portable_imt_conflict_trampoline_offset` and `portable_resolution_trampoline_offset`, while in Version 064 the corresponding field is completely gone.

In addition, we can identify the command used to generate the corresponding OAT file using `key_value_store`, the last field of `OatHeader`. Figure 6 shows the content of `key_value_store` from extracting the OAT file of the test application installed in Android 5.1. `--zip-location` represents APK file's path while `--oat-location` represents the location of where the OAT file

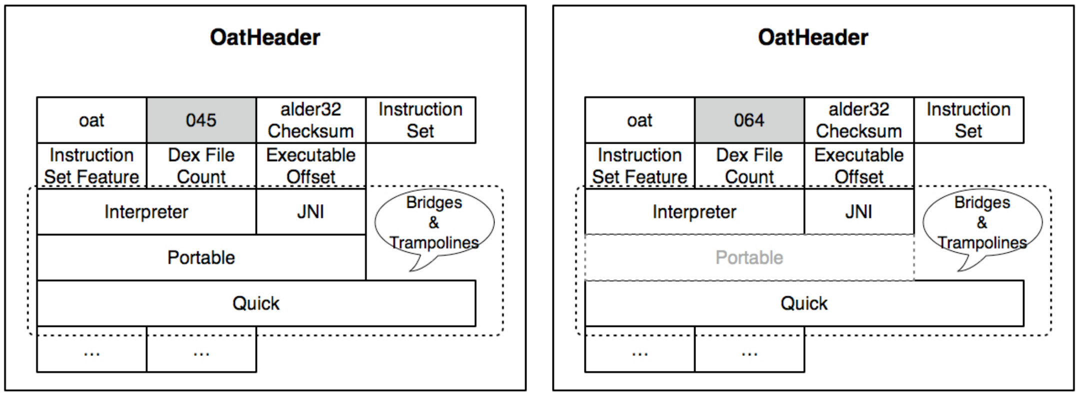

Figure 5: OatHeader Structure Comparison

generated through the compilation process will be placed. Next, `--instruction-set` is an option that indicates to compile instructions as a specific instruction set. The basic value is arm, but arm64, mips, mips64, x86, and x86_64 are also supported. However, depending on the Android version, sometimes only part of the enumerated instruction set is supported.

```
dex2oat-cmdline = --zip-fd=6 --zip-location=/data/app/zto.smali-
hookmanager-1/base.apk  --oat-fe=7  --oat-location=/data/dalvik-
cache/arm/data@app@zto.smalihookmanager-1@base.apk@classes.dex
--instruction-set=arm  --instruction-set-features=div  --runtime-
arg -Xms64m --runtime-arg -Xmx512m --swap-fd=8
```

Figure 6: Example of `dex2oat` command line

`OatDexFile:` Right after `OatHeader` is one or more `OatDexFiles` – as many as the number of DEX files there are in `oatdata`. The number of DEX files in the OAT file can be known using the `dex_file_count` field in the aforementioned `OatHeader`.
`OatDexFile` is comprised of brief information regarding the DEX file and includes the path, CRC32 checksum, and offset of the DEX file as well as the `OatClass` offset for the DEX file in array form (See Figure 7).

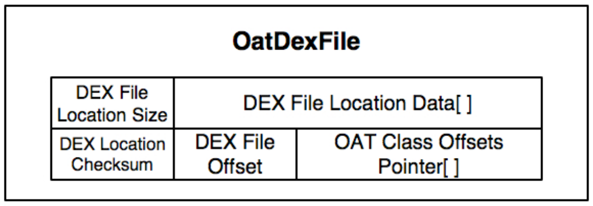

Figure 7: `OatDexFile` Structure

`DEX File:` One or more DEX file may exist between OatDexFile and OatClass, as in Figure 8. If an application is installed, the classes.dex file is included in the oatdata section of the OAT file in the APK file. While generally only one DEX file exists, for boot.oat files including Android frameworks such as core-libart.jar, framework.jar, etc., many DEX files exist and, as explained before, there are as many OatDexFiles containing information regarding DEX files as there are DEX files.

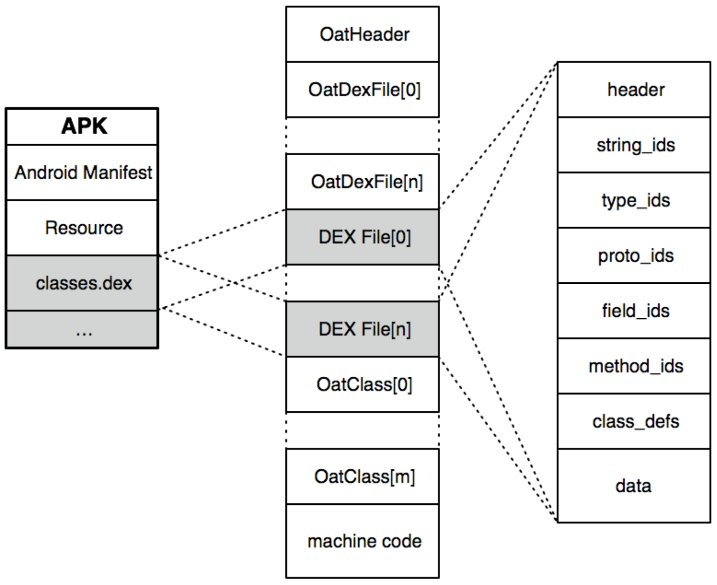

Figure 8: DEX File in `oatdata` section

`OatClass`: After the DEX file, there is one or more `OatClass`, the total number being however as many DEX file classes there are. In the DEX file header, there is a `class_defs_size` field that represents the number of classes contained in the DEX file, which also indicates the number of `OatClass`.

`OatClass` contains general information regarding the class and method of the application. Along with information on whether the class and method were both compiled error free during the compilation process or whether only a part of the method was compiled, there is also information on the type and member method of the class. The member method information is managed in array form using `methods_pointer` field. Each element of the array, in OatMethod form, is made of information directly related to the machine code such as the actually AOT compiled machine code of the method and the method header.

2.3 Compilation Methods

With Dalvik, JIT (Just-in-time) compilation was used. JIT compilation is a method of improving the application's execution performance by dynamically compiling the Dalvik bytecode into machine code every time the application is run. JIT with Androids, to minimize the resulting overhead from the machine code compilation process, does not compile the application's entire Dalvik bytecode but compiles into machine code a uniform number of parts that are repeatedly executed.

On the other hand, ART uses AOT (Ahead-of-time) compilation to compile the Dalvik bytecode into machine code in advance when the application is installed. While JIT compilation compiles only parts of the application into machine code, AOT compilation targets the entire application for compilation.

As the abovementioned process occurs during installation of an application, there are drawbacks in having a longer installation time than the current time as well as in needing a greater amount of storage space. However, compared to the early mobile devices with limited internal storage capacity, most of today's mobile devices support sufficient storage capacity, so the issue of storage space is no longer counted as a drawback. Rather, because the machine code is compiled in advance, the machine code no longer needs to be dynamically generated every time the application runs, resulting in greater memory efficiency and longer battery life as well as better run performance.

2.4 File Analysis Tools

The DEX file, compared to the OAT file, relatively has more information on file structure. Even formally, the DEX file structure is disclosed, and a number of analysis and research materials have been collected over the years. As a result, the various existing tools explained in Section 3 (eg., `apktool`[1], `baksmali`[2], etc.) can be used to analyze the DEX file. Even if a tool does not exist, it is possible to run an analysis up to a certain point by solely using information on the structure of DEX file.

On the other hand, the OAT file, carrying limited information compared to DEX file, can be analyzed using Android's basic system utility `oatdump`. The main function of `oatdump` is to process the content in OAT file format and to provide it in a format that is discernable to the user. `oatdump` largely provides information on `OatHeader` and `OatClasses`. Because the `OatHeader`, as explained before, includes information regarding the OAT file version, instruction set, machine code start position, checksum, commands and options used when generating the OAT file, etc., we can determine general information regarding the OAT file by dumping `OatHeader` information. Containing important information such as bytecode and machine code, mapping table, etc. through the compiled information of each class and the methods within each class, `OatClasses` can be considered the focus of core analysis. However, because `oatdump` simply dumps all of the OAT file's content, if the file size gets bigger, the dumped content becomes so large that analysis becomes somewhat difficult. Furthermore, with information regarding the entire DEX file structure, excluding Dalvik bytecode, not separately provided, analysis from various angles can become complicated.

`oatdump++`[6], modified and enhanced from `oatdump`, provides a greater variety of functions than does the current `oatdump` such as class list extraction, method list extraction, filtered classes dump, filtered methods dump, DEX file extraction, etc. While `oatdump++` is based on `oatdump` but offered separately, because the main functions of `oatdump++` are currently merged in AOSP (Android Open Source Project), the various functions provided by `oatdump++` can be used starting from Android 6.0. Comparison of Dalvik and ART can be summerized as Table 1.

	Dalvik	ART
File Format	DEX	OAT + DEX
Compilation Method	JIT	AOT
File Analysis Tool	apktool, baksmali, etc	oatdump(oatdump++)

Table 1: Comparison of Dalvik and ART

3 Static Analysis

Static analysis refers to analysis of the application without running the application using the application's source code or object code. This section investigates whether the static analysis methods used in Dalvik can also be applied in the ART environment.

3.1 Analysis Tools

Android static analysis tools can be grouped into diassemblers and decompilers and, depending on whether they support analysis of Dalvik bytecode and machine code, can be organized as Table 2.

Disassemblers	Description	Bytecode	Machine code
Apktool	Disassembles Dalvik bytecode into Smali code	Yes	No
Dedexer[3]	”	Yes	No
Oatdump	OAT file dump	Yes	Yes
Oatdump++	Provides additional functions than Oatdump	Yes	Yes
IDA	Disassembles Dalvik bytecode and machine code	Yes	Yes

Decompilers	Description	Bytecode	Machine code
Dex2jar & jad	Reconstruct Dalvik bytecode into Java source code	Yes	No
JEB	Provicdes automated decompilation function	Yes	No

Table 2: Static Analysis Tools for Android

Using analysis tools that targeted the DEX file, the analysis methods can be applied on the DEX file included in the OAT file or the DEX file in the distributed APK. To analyze the new OAT file structure and machine code, `oatdump` or GDB can be used.

As analysis tools for OAT files, as explained in Section 2.4, there is `oatdump` and `oatdump++`. Fundamentally they provide the disassemble function for Dalvik bytecode and machine code and additionally provide information of the OAT file format, thus supporting analysis of OAT file.

Another static analysis tool besides those two is commercial software IDA. In the case of IDA, it provides the disassemble function for both machine code and Dalvik bytecode. When the input is an OAT file, IDA provides a disassemble function for machine code regardless of the Dalvik bytecode located in the `oatdata` section within the OAT file while if the input is an APK file, IDA disassembles the DEX file within the APK file. Thus, while IDA provides disassemble functions for both machine code and Dalvik bytecode, with one input, it cannot identify the two code formats simultaneously. To analyze both machine code and Dalvik bytecode, it needs both the APK (or DEX) file and OAT file.

3.2 Applicability on ART

For Android applications, the smali code obtained using `apktool` or `baksmali` or the java code restored using `dex2jar` becomes the target of static analysis. In the Dalvik environment, the main static analysis method was to, after either extracting the APK file from the application or downloading it, to disassemble the DEX file (`classes.dex`) in the APK file using the `apktool`, while simultaneously decoding the resources to analyze not only the application's code[12] but the various elements of the application's architecture. This kind of method is also still effective in the current ART environment due to the following reason.

Figure 9 has taken Figure 1, which showed the change during the installation and execution of the application, and has focused on the differences during the time of nd the run stage, there are no differences in distribution. In ART as well, the application that the developer finally distributesdistribution. While Dalvik and ART have quite stark differences during the installation stage of the application onto the device a is still distributed in the APK file format, and users install them onto their devices from a variety of markets and websites. Because application developers and users, regardless of the type of runtime system, are utilizing the familiar, existing mechanism, due to the nature of static analysis where the actual execution of the application is unnecessary, existing methods and tools can be applied for analysis in the ART environment as well.

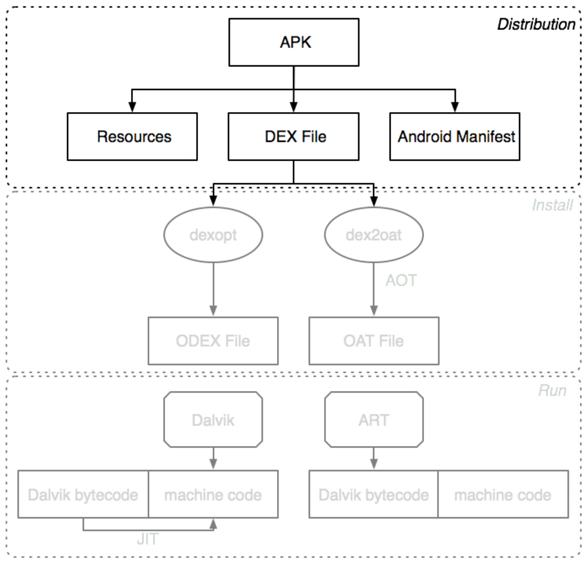

Figure 9: Difference between Dalvik and ART in application distribution

4 Dynamic Analysis

Dynamic analysis is the analysis of the application's operational behavior by running the application. In this section, we examine whether the existing Dalvik-based dynamic analysis methods for Android applications are applicable in the ART environment.

4.1 Analysis Tools

The main dynamic analysis tools that have been used are shown in Table 3.

Name	Description	Bytecode	Machine code
NetBeans	Provides Dalvik bytecode debugging function	Yes	No
GDB	Provides machine code debugging function	No	Yes
IDA	Provides Dalvik bytecode & machine code debugging function	Yes	Yes
DroidScope[5]	Provides real-time Dalvik bytecode extraction function	Yes	No

Table 3: Dynamic Analysis Tools for Android

4.2 Applicability on ART

One of the reasons for continuing to leave the entire Dalvik bytecode into the OAT file despite having generated the machine code based on the Dalvik bytecode during application installation in the ART environment can be understood by relating it to dynamic debugging.

When an ART-based Android application is generally run, it is made to run using the compiled machine code, while in the case of methods with events such as `break` point or `watch` configured in an environment in the process of debugging, it is made to reset the method's entry point so that the application is run using Dalvik bytecode. This structure would run in Dalvik-based dynamic debugging environments for Androids using JDWP (Java Debug Wire Protocol) which, we surmise, was to maintain the system structure as is. Consequently, dynamic analysis of an application in ART or the usage of existing JDWP-based debugging tools should be possible.

5 Experiments

5.1 Static Analysis

Static analysis experiments show that the new runtime system and the previous existing system run with the same exact mechanism with no mentionable difference. The steps for static analysis are delineated below[10].

First, the application's APK file is extracted (or downloaded). While there are slight differences in the APK file pathway according to the Android version, they are similar overall.

Second, obtain the smali code or Java code using the prepared in advance diassembler and decompiler. Excluding cases where a separate protection mechanism is applied to the application, the majority of the tools will allow the analyzer to obtain the code in any form he or she wants. Using the obtained code as the basis, the analysis can now begin in earnest as shown in Figure 10.

The two steps mentioned above are applicable in both Dalvik and ART, the reason being that, during distribution time, the executable file format is the same DEX file within the APK file for both.

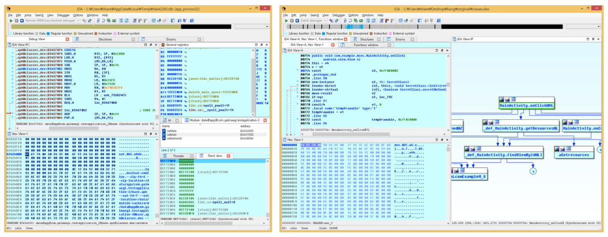

Figure 10: Disassembled output using IDA: machine code (Left) and Dalvik bytecode (Right)

5.2 Dynamic Analysis

NetBeans provides dynamic debugging for Dalvik bytecode, while GDB provides dynamic debugging for machine code. Operating through JDWP, Netbeans was usable in Dalvik, which allows JDWP. Because, fundamentally, the method is executed based on machine code in ART and ART also allows JDWP, we tested for whether, when debugging, the method would be once again executed using Dalvik bytecode. We modified the OAT file so that the Dalvik bytecode and machine code of a method that outputs the log would each output a different string, as seen in Figure 11.

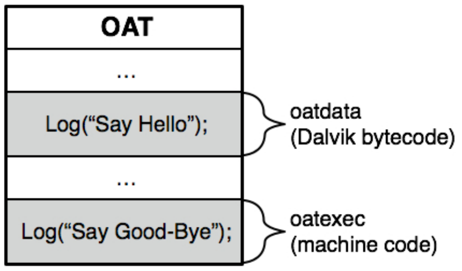

Figure 11: Modified bytecode and machine code example

While the string "`Say Good-Bye`" is output until the method's events are triggered using dynamic debugging tools such as Netbeans or IDA, we were able to confirm that the string "`Say Hello`" was output when a debugging event is configured. Judging from the code shown in the tool as well as the string “`Say Hello`” as seen in Figure 12, we could see that it was Dalvik bytecode rather than method’s machine code that functioned as the basis of dynamic debugging.

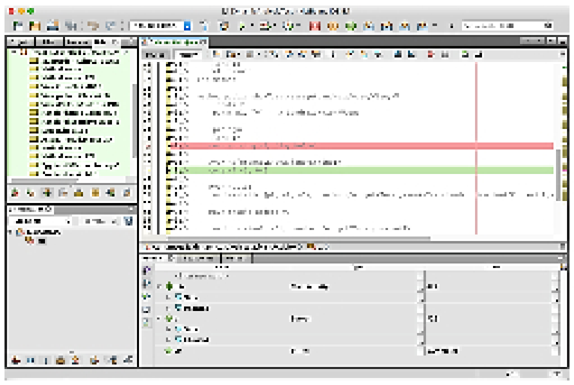

Figure 12: Dalvik bytecode dynamic debugging using Netbeans in ART environment

Thus, in the case of dynamic debugging methods using JDWP, the debugging function is provided by forcibly changing the entry point of the method in ART, while with using GDB on the other hand, the actually executed machine code of the method is able to be debugged as shown in Figure 13.

```
(gdb) shell cat / proc/18908/maps | grep dex
b3433000-b3437000  r--p  00000000 b3:1c 106051      /data/dalvik-cache/arm/data@app@com.galmaegi.testapplication-2@base.apk@classes.dex
b3437000-b3439000  r-xp  00004000 b3:1c 106051      /data/dalvik-cache/arm/data@app@com.galmaegi.testapplication-2@base.apk@classes.dex
b3439000-b343a000  rw-p  00006000 b3:1c 106051      /data/dalvik-cache/arm/data@app@com.galmaegi.testapplication-2@base.apk@classes.dex
b343a000-b343b000  r--p  00000000 b3:1c 106051      /data/dalvik-cache/arm/data@app@com.galmaegi.testapplication-2@base.apk@classes.dex
(gdb) x/8c 0xb3434000
0xb3434000:    111 ‘o’ 97 ‘a’ 116 ’t’ 10 ‘\n’ 48 ‘0’ 52 ‘4’ 53 ‘5’ 0 ‘\000’
(gdb) x/20i 0xb3437025
   0xb3437025:  stmdb   sp!, {r5, r6, lr}
   0xb3437029:  sub     sp, #20
   0xb343702b:  adds    r6, r0, #0
   0xb343702d:  str     r0, [sp, #0]
   0xb343702f:  adds    r5, r1, #0
   0xb3437031:  movw    lr, #57461     ; 0xe075
   0xb3437035:  movt    lr, #29402     ; 0x72da
   0xb3437039:  movw    r0, #41320     ; 0xa168
   0xb343703d:  movt    r0, #28840     ; 0x70a8
   0xb3437041:  adds    r1, r5, #0
   0xb3437043:  blx     lr
```

Figure 13: Dynamic debugging of machine code using GDB

Because this experiment forcibly constructed Dalvik bytecode and machine code, it showed the differences in results when debugging while, generally, the Dalvik bytecode and machine code of OAT files are only different in form and will generate the same results. In addition, QEMU based dynamic analysis tools such as DroidScope, Andrubis, DroidBox[4], Taintdroid[8] provide functions such as extracting Dalvik instruction as well, they all having a strong dependency on Dalvik.

6 Conclusion

In this paper, to smoothly deal with the new malicious applications soon to appear due to the Android platform’s new change in runtime system, we analyze the main changes in the runtime system and introduce related static analysis methods and dynamic analysis methods. We were able to confirm that, with the exception of dynamic analysis tools that use the QEMU emulator, which heavily depends on Dalvik, most of the existing analysis tools and methods were still effective with the change in runtime system. The fact that the existing analysis tools are effective also means that the current various analysis prevention methods are also effective. However, in the future, we predict that malicious applications that abuse the vulnerabilities in ART or take advantage of its structural characteristic and such will gradually emerge. Thus, with the appearance of new malicious code in the future, the ART system structure analysis results presented in this paper will help contribute toward effective analysis of the malicious code.

Acknowledgments

This research was supported by the Global Research Laboratory (GRL) program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT, and Future Planning (NRF-2014K1A1A2043029).

References

[1] Apktool. `http://ibotpeaches.github.io/Apktool` [Online; Accessed on August 5, 2016].

[2] Baksmali. `https://github.com/JesusFreke/smali` [Online; Accessed on August 5, 2016].

[3] Dedexer. `http://dedexer.sourceforge.net/` [Online; Accessed on August 5, 2016].

[4] Droidbox. `https://github.com/pjlantz/droidbox/` [Online; Accessed on August 5, 2016].

[5] Droidscope. `https://github.com/sycurelab/DECAF/` [Online; Accessed on August 5, 2016].

[6] Oatdump++. `https://github.com/anestisb/oatdump_plus/` [Online; Accessed on August 5, 2016].

[7] Qemu. `http://wiki.qemu.org/` [Online; Accessed on August 5, 2016].

[8] Taintdroid. `http://www.appanalysis.org/` [Online; Accessed on August 5, 2016].

[9] M. Backes, O. Schranz, and P. von Styp-Rekowsky. Poster: Towards compiler-assisted taint tracking on the android runtime (art). In *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15), Denver, Colorado, US*, pages 1629–1631. ACM, October 2015.

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

[11] S. Kim, J. Park, K. Lee, I. You, and K. Yim. A brief survey on rootkit techniques in malicious codes. *Journal of Internet Services and Information Security (JISIS)*, 2(3/4):134–147, November 2012.

[12] S. W. Park and J. H. Yi. Multiple device login attacks and countermeasures of mobile voip apps on android. *Journal of Internet Services and Information Security (JISIS)*, 4(4):115–126, November 2014.

[13] Y. Park, C. Lee, J. Kim, S.-J. Cho, and J. Choi. An android security extension to protect personal information against illegal accesses and privilege escalation attacks. *Journal of Internet Services and Information Security (JISIS)*, 2(3/4):29–42, November 2012.

[14] P. Sabanal. State of the art exploring the new android kitkat runtime. Hack In The Box Security Conference, 2014. `https://www.corelan.be/index.php/2014/05/29/hitb2014ams-day-1-state-of-the-art-exploring-the-new-android-kitkat-runtime/?output=pdf` [Online; Accessed on August 5, 2016].

[15] P. Sabanal. Hiding behind art. Black Hat, 2015. `https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp.pdf` [Online; Accessed on August 5, 2016].

[16] S. Sakamoto, K. Okuda, R. Nakatsuka, and T. Yamauchi. Droidtrack: Tracking and visualizing information diffusion for preventing information leakage on android. *Journal of Internet Services and Information Security (JISIS)*, 4(2):55–69, May 2014.

Author Biography

Geonbae Na received the B.S. degree in Computer Science and Engineering and the M.S. degree in Software Convergence from Soongsil University, Seoul, Korea, in 2014 and 2016, respectively. His research interests include mobile application security, mobile platform security and Reverse Engineering.

Jongsu Lim received the B.S. degree in Computer Science and Engineering from Soongsil University, Seoul, Korea, in 2016. Currently he is taking a master's course in Software Convergence from Graduate School of Soongsil University. His research interests include mobile application security, mobile platform security and Reverse Engineering.

Kyoungmin Kim received the B.S. degree in Computer Science and Engineering from Soongsil University, Seoul, Korea, in 2016. Currently he is taking a master's course in Software Convergence from Graduate School of Soongsil University. His research interests include mobile application security, mobile platform security and Reverse Engineering.

Jeong Hyun Yi is an Associate Professor in the School of Software and a Director of Mobile Security Research Center at Soongsil University, Seoul, Korea. He received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, Korea, in 1993 and 1995, respectively, and the Ph.D. degree in information and computer science from the University of California, Irvine, in 2005. He was a Principal Researcher at Samsung Advanced Institute of Technology, Korea, from 2005 to 2008, and a member of research staff at Electronics and Telecommunications Research Institute (ETRI), Korea, from 1995 to 2001. Between 2000 and 2001, he was a guest researcher at National Institute of Standards and Technology (NIST), Maryland, U.S. His research interests include mobile security and privacy, IoT security, and applied cryptography.