IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 23, 2018, accepted November 9, 2018, date of publication November 16, 2018,
date of current version December 18, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2881699

DexMonitor: Dynamically Analyzing and Monitoring Obfuscated Android Applications

HAEHYUN CHO[1], JEONG HYUN YI[2], (Member, IEEE), AND GAIL-JOON AHN[1,3], (Senior Member, IEEE)

[1]Center for Cybersecurity and Digital Forensics, Arizona State University, Tempe, AZ 85281, USA
[2]Cyber Security Research Center, Soongsil University, Seoul 06978, South Korea
[3]Samsung Research, Samsung Electronics, Seoul 06765, South Korea

Corresponding author: Jeong Hyun Yi (jhyi@ssu.ac.kr)

This work was supported in part by the Institute for Information and Communications Technology Promotion through the Korea Government under Grant MSIT 2017-0-00168, in part by the Automatic Deep Malware Analysis Technology for Cyber Threat Intelligence, and in part by the Global Research Laboratory Program through the National Research Foundation of Korea through the Ministry of Science, ICT, and Future Planning under Grant NRF-2014K1A1A2043029.

ABSTRACT Both Android application developers and malware authors use sophisticated obfuscation tools to prevent their mobile applications from being repackaged and analyzed. These tools obfuscate sensitive strings and classes, API calls, and control flows in the Dalvik bytecode. Consequently, it is inevitable for the security analysts to spend the significant amount of time for understanding the robustness of these obfuscation techniques and fully comprehending the intentions of each application. Since such analyses are often error-prone and require extensive analysis experience, it is critical to explore a novel approach to systematically analyze Android application bytecode. In this paper, we propose an approach to address such a critical challenge by placing hooks in the Dalvik virtual machine at the point where a Dalvik instruction is about to be executed. Also, we demonstrate the effectiveness of our approach through case studies on real-world applications with our prototype called DexMonitor.

INDEX TERMS Bytecode monitoring, Android application analysis, mobile security.

I. INTRODUCTION

Smartphones are exploding in popularity and functionality. Mobile operating systems that run on smartphones allow third-party developers to develop mobile applications that take advantage of the features of mobile devices. These mobile applications have influenced diverse sectors such as financial, government, entertainment, and healthcare sectors.

Among all the mobile operating systems, Google's Android leads the largest market share and the reports indicate that the Google Play, which is the official application store for Android, has around 1.4 million applications available with over 50 billion application downloads [19], [40].

Android applications are typically written in Java, which are compiled to bytecode that runs on a Java virtual machine, called the Dalvik Virtual Machine (Dalvik VM). Such a Dalvik bytecode is typically "simpler" for static analysis, compared to the traditional binary code for desktop systems (x86 and and x86-64), which results in a rise of Dalvik *decompilers* that transform Dalvik bytecode back to Java code [9], [14]. Also, Dex files containing bytecode can be converted into smali code using a disassembler such as bakSmali [16]. The smali code, containing a lot of information such as Android API information within the Dalvik's instructions [6], becomes a major target of app analysis.

Unfortunately, malicious actors also take advantage of the ease of decompiling Android applications and can perform *repackaging attacks* on Android applications [27], [30]. A repackaging attack happens when a paid or free application in the market is illicitly reverse-engineered, modified, and then redistributed by attackers rather than its original developers. The goal of the attacker is to either insert malicious code in the repackaged application or modify the advertising library to use the attacker's own code so that the attacker can obtain monetary gains from the advertisements. To address such issues, Android application developers use various obfuscation techniques to make their applications more difficult for attackers to dissect. Consequently, rogue application developers can also use these obfuscation techniques to hide malicious behaviors in an application.

The most commonly used obfuscation techniques on Android applications [35] include: (1) *string encryption* that encrypts sensitive strings used in applications [1], [10], [11], [17]; (2) *class encryption* that hides an entire class

by encrypting it and removing it from `classes.dex` [10], [11]; (3) *identifier renaming* that makes reverse-engineered programs less readable by changing the identifiers in an application [1], [10], [11], [17], [36]; (4) *control flow randomization* that makes the control flow of a program difficult to understand [28]; (5) *API hiding* that hides invocations of sensitive APIs, such as cryptographic functions, by using Java reflection [10], [11]; and (6) *virtualization-based protection* that encodes Android bytecode to virtual instructions and uses special virtual machines to execute such instructions [44]. These techniques are widely used in Android application obfuscators, including Stringer [17], Allatori [1], DexProtector [11], DexGuard [10], and DIVILAR [44].

Static analysis of obfuscated Android applications is tedious and error-prone, and requires extensive analysis experience. In addition, static analysis can be easily defeated by encryption-based obfuscation techniques. Therefore, it is imperative to develop novel ways to analyze obfuscated applications. In addition, it would help achieve the following goals: (1) extraction of malicious applications' hidden bytecode prior to any in-depth analysis; (2) measurement of the effectiveness of the obfuscators; and (3) understanding of design requirements for building more robust obfuscation techniques.

In this paper, we propose an approach called DexMonitor to place hooks in the Dalvik VM at the point where a Dalvik instruction is about to be executed. Due to the nature of program execution, obfuscated segments must be revealed by the application itself at this point. By finding this point and intercepting the code and data when the program counter reaches this point, we can generate a view of the disclosed code and data without knowing how the applications were obfuscated. As a consequence, DexMonitor can obtain applications analyzable by providing code and data under the situation where the code is concealed.

The main contributions of this paper are the following:

- We propose a novel approach to analyze Android applications by selectively intercepting Dalvik instructions.
- We implement a prototype, called DexMonitor, by modifying the Dalvik VM.
- We demonstrate the effectiveness of DexMonitor through case studies.

The remainder of the paper is organized as follows. In Section II, we overview the background knowledge and compare existing obfuscators. We articulate our proposed system, DexMonitor to automatically deobfuscate Android applications in Section III. We discuss the implementation and evaluation results in Section IV. In Section V, we describe the challenges and limitation of our work. We conclude the paper in Section VII.

II. BACKGROUND

A. DALVIK VIRTUAL MACHINE

Dalvik is the virtual machine used in Android operating system. At the booting stage of Android, a process `zygote` waits for the request of launching a new application. When such a request arrives, `zygote` forks itself and initializes Dalvik VM for the new application, and then the Dalvik VM will load the application's `classes.dex` file. Normally, dex files are optimized by the Dex Optimizer `dexopt`, that is a component of Dalvik, before being executed [8]. `dexopt` sets byte ordering and structure alignment, replaces certain instructions, such as `invoke-virtual`, and performs method inlining during the dex file optimization. Because method inlining changes the instructions for invoking operations, it must be avoided to faithfully record what has been executed in an application. The interpreter, which is the main part of Dalvik VM, interprets Dalvik bytecode into architecture-specific binary code. From the version 2.2 of Android, Dalvik has a just-in-time compiler (JIT) for improving the runtime performance. While an application is running, the JIT analyzes Dalvik bytecode and actively translates hot parts into the optimized native code [7], [23]. Therefore, it is necessary to disable JIT for the monitoring purpose.

The main feature of the Dalvik VM is the execution of the dex file. Instructions executed in the Dalvik VM are called Dalvik instructions. The Dalvik VM actually converts Dalvik instructions into machine language compatible to the CPU architecture of the mobile device. Figure 1 shows how the Dalvik VM executes a dalvik instruction. Dalvik VM uses 16-bit instruction set. Thus, the Dalvik program counter increases 16-bit and the pointed instruction executes according to its defined operation where operands are parsed and the program counter is changed for the next instruction.

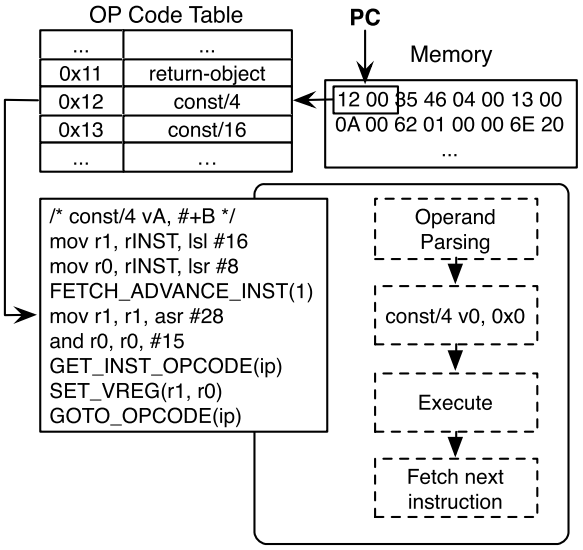


FIGURE 1. Bytecode execution process on Dalvik VM.

B. DALVIK BYTECODE OBFUSCATION

Since Dalvik is the process VM in Android, we use the terms Davik bytecode and Android application bytecode interchangeably in the rest of this paper. Obfuscation techniques have been proposed for Dalvik bytecode to hinder reverse engineering and APK repackaging. The most commonly

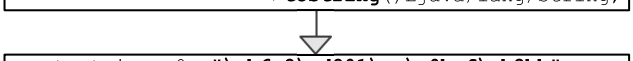
TABLE 1. Comparing android application obfuscators.

	Stringer	Allatori	DexProtector	DexGuard	DIVILAR
String Encryption	√	√	√	√	
Class Encryption			√	√	
Identifier Renaming	√	√	√	√	
Control Flow Randomization		√			
API Hiding			√	√	
Virtualization-based Protection					√
Tamper Detection			√	√	

used obfuscation techniques [35] include string encryption, class encryption, identifier renaming, control flow randomization, API hiding, and virtualization-based protection. Some Android application obfuscators in the market or from academia include Stringer [17], Allatori [1], DexProtector [11], DexGuard [10], and DIVILAR [44]. Table 1 compares the features of these tools. Since DexProtector and DexGuard provide most features that are offered by the other tools, we will revisit these obfuscation schemes along with DexMonitor in the Section III.

DexProtector works directly on Dalvik bytecode, and it provides both string encryption and class encryption. The string encryption feature encrypts strings used in an application to protect sensitive information, such as a program's license information or hash value for tampering detection by using AES for string encryption. DexProtector can also hide APIs called within an application. An application obfuscated by DexProtector can intentionally trigger a system error to force the application to be closed, when the application is repacked. DexGuard provides similar features as DexProtector does. However, it is a backend compiler that comes with Android SDK and works on Java source code instead of bytecode. Figure 2 shows the examples of obfuscated code snippets which were generated by DexProtector and DexGuard respectively. As shown in the Figure, obfuscation schemes make the static analysis very difficult by concealing information that is critical for understanding behaviors of applications such as strings, API names, and even identifiers.

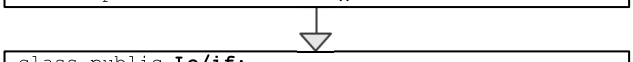
```
const-string v1, "Hello World"
invoke-virtual {v0}, Landroid/widget/TextView;
                  ->getText()Ljava/lang/CharSequence;
move-result-object v2
invoke-interface {v2}, Ljava/lang/CharSequence;
                       ->toString()Ljava/lang/String;
```



```
const-string v0, "\ub6e8\ud801\……\u0baf\ub2bb"

invoke-static {v0}, Lcom/dexprotector/ha;
      ->gbeab(Ljava/lang/Object;)Ljava/lang/Object;

invoke-static {v2}, Lcom/dexprotector/ha;
      ->oodab(Ljava/lang/Object;)Ljava/lang/Object;
```

(a)

```
.class public Lcom/example/SecretClass;
.super Ljava/lang/Object;

.field private final PASSWD:Ljava/lang/String;

.method public secretMethod()Z
```



```
.class public Lo/if;
.super Ljava/lang/Object;

.field private final `:Ljava/lang/String;

.method public `()Z
```

(b)

FIGURE 2. Obfuscation examples. (a) String encryption and API hiding (DexProtector). (b) Identifier renaming (DexGuard).

C. OTHER PROTECTION METHODS

With the threat of the repackaging attack, various protection schemes such as tamper detection, anti-analysis schemes have been emerged [21]. Tamper detection scheme is to prevent tampering of applications. In case of that tamper detection scheme is applied, detecting module of the application checks the application to verify its integrity and, based on the result, the module determine whether to execute. In addition, to secrete the tamper detecting routine, obfuscation methods such as class encryption could be used and there is a tamper detection scheme using the server in which the detecting routine is located [35]. Analyzers sometime insert logging code in the classes.dex file to figure out specific values or conditions. However, with the tamper detection schemes, repackaging the APK file is hard pressed.

On the other hand, there are methods protecting applications from the analysis. For examples, method concealment [20], the manifest file modification [4], and altering the header of ZIP file [13] schemes are used to prevent static analysis. Also, anti-debugging methods [24] and emulator detecting methods [29], [34], [38] could be employed against dynamic analysis. Because malicious applications, of course, can use those protection schemes, we need an analysis system which can be harnessed effectively on various protection methods.

III. DexMonitor

The major objective of DexMonitor is to make an application analyzable by providing executed bytecode with files. Given that a lot of protection techniques prevent applications from being analyzed. Especially encryption-based protection

techniques prohibit analyzers from acquiring concealed code and data.

The outputs, produced on Dalvik VM, of DexMonitor are instructions as a form similar to smali with the detailed information, in which the instructions are included, loaded dynamically. With the outputs, encrypted bytecode, string or any dynamically loaded bytecode can be revealed. Therefore, based on the outputs, we can easily analyze protected applications. In addition, DexMonitor extracts all executables dynamically loaded. By providing hidden executable files which cannot be found statically, it makes a deep analysis possible easily.

Even though similar ideas have appeared in previous approaches to decrypt online streaming content [39] and sensitive malware strings [43], applying it to analyze Dalvik bytecode has some unique technical challenges. First, it requires a complete understanding of Dalvik VM. We build DexMonitor based on the accurate bytecode execution process of Dalvik VM and its related components for having a view of the revealed code and data without knowing how an application's obfuscation and encryption algorithms work. Another challenge is to construct practical outputs for an effective analysis. To achieve this, DexMonitor produces instructions that has a smali-like format comparable to the original smali instructions disassembled from dex files, which helps analyzers to see the original form of executed instructions. On the other hand, DexMonitor provides detail data of an instruction's operands such as strings or numerical values of them. Therefore, analyzers can grasp the context of an instruction and can easily find critical values that affect execution states of an application.

A. DESIGN GOALS

We first articulate a list of design goals that are considered and accommodated in DexMonitor.

1) IN-THE-BOX DESIGN

The *Dalvik instruction tracer* of DroidScope [12], [42], that is a virtualization-based and out-of-the-box approach, can be modified to realize the similar basic idea by intercepting code and data at the same point of program execution. However, an in-the-box design that modifies Dalvik VM directly instead of using virtual machine inspection has the following advantages.

i) Anti-emulation, anti-debugging proof. Android applications thwart dynamic analysis by detecting their running environments. If an emulator is detected, an application could change its behavior or simply crash itself. Recent studies have shown that there exist many ways for Android applications to tell if it is running in an emulator [29], [34], [38]. In addition, anti-debugging methods can hinder an analysis by checking the debugging environment [24]. By directly changing the Dalvik VM and running it on bare-metal, we can minimize of risk of being detected by the analyzed applications. Note that our approach still allows analysts to run the modified VM in an emulator which provides then an option to use bare-metal analysis. Analysts can still run our system of the modified Dalvik VM in an emulator if they need.

ii) No semantic gap. Direct modification of the Dalvik VM is easier to implement, since the proposed approach runs with the Dalvik VM and sees all the symbol information that is necessary for monitoring and analysis. On the other hand, a virtualization-based approach has to reconstruct the semantic gap between the low-level view from the virtual machine monitor and the Dalvik-level view.

iii) Minimal performance overhead. Virtual machine inspection introduces significant performance degradation. Large-scale Android application analysis may not be feasible due to such a critical limitation. By directly modifying the Dalvik VM and selectively monitoring, it is guaranteed to introduce minimal performance overhead.

2) SELECTIVE MONITORING

If a system simply outputs all executed Dalvik instructions as the *Dalvik instruction tracer* of DroidScope [42] does, it will produce many instructions that are not part of Android applications. These instructions may come from Android application framework, Android Libraries, etc. Outputting such instructions without knowing where they are from would make subsequent analysis steps more complicated. To cope with this challenge, it is necessary to know which Android application, thread, class, or method is executing on the Dalvik VM and only outputs the traces of analyst-specified applications, classes or methods. With such a design, we can also check the method flow of applications by monitoring only `invoke` and `return` instructions.

B. BUILDING BLOCKS OF DexMonitor

To meet the aforementioned design goals, DexMonitor monitors the execution of Dalvik instructions at run time and outputs the observed information for further analysis. DexMonitor consists of several modules that reside in Dalvik VM. As shown in Figure 3, DexMonitor is comprised of

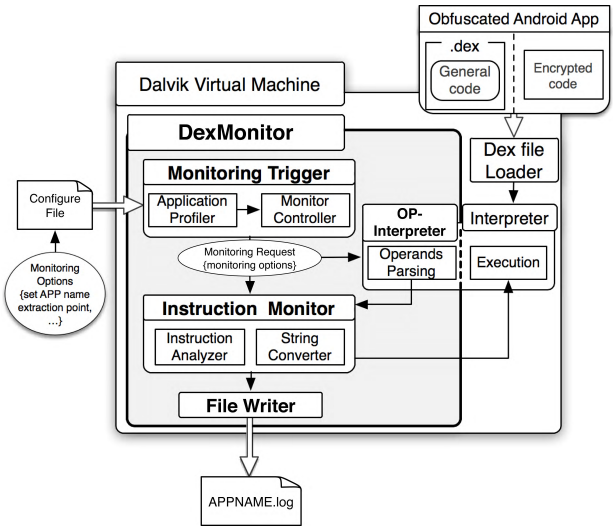


FIGURE 3. The architecture of DexMonitor.

three major components: Monitoring Trigger, OP-Interpreter, and Instruction Monitor.

1) MONITORING TRIGGER

To achieve selective monitoring, DexMonitor takes a configuration file as an input, which includes the applications, classes, and methods a security analyst wants to monitor. Monitoring trigger is responsible for reading this configure file. Each line of this configuration file is a string that indicates a package, class, or method to be monitored. For example, **com.example | 1 | com/example/SecretClass | all** in a configuratin file means DexMonitor should monitor all methods of the class **com/example/SecretClass** in the thread `1` of the application its package name is **com.example**. The number of thread is a number managed by Dalvik VM and if the value of thread is `0`, it indicates all threads are monitored. Furthermore, there are 3 on/off options: extracting executable files, extracting parameters when invoking method, and extracting only `invoke` and `return` instructions. For instance, options of **extractFile=off, extractParam=on, extractInvokeOnly=on** signify that DexMonitor does not extract the executable files containing bytecode and outputs only instructions regarding method call and return with information of parameters. If the option **extractFile** is **on**, DexMonitor extracts all executable files (in dex, jar, APK, ZIP and so format) loaded by the application.

Upon the launch of an application, DexMonitor reads the configure file and compares the package names written in the file with the launched application's package name before the Dalvik VM invokes the interpreter. A global structure that defines the monitoring range is declared in the `dalvik/vm/Globals.h`. The global variable is initialized in the `dvmInitAfterZygote` function in the `dalvik/vm/Init.cpp` file, so it can be ready before the interpreter is started.

2) OP-INTERPRETER

In the Dalvik VM, the `dvmInterpret` function in the `dalvik/vm/interp/Interp.cpp` file is the entry point of the interpreter. It calls an actual interpreter based on which execution mode it is in. For example, if the `WITH_JIT` variable that is declared in `Android.mk` is `TRUE`, the `ModeJit` interpreter is chosen at compile time and the `dvmMterpStd` interpreter is called by function `dvmInterpret` at runtime. We introduce a hook in the `dvmInterpret` function to invoke our OP-Interpreter.

Before an instruction is executed, OP-Interpreter determines whether the instruction should be recorded or not. The parameters of `invoke` or `return` instructions are also taken into consideration if such an instruction should be recorded. If a target method invokes some Android APIs, such as `loadLibrary`, it does not extract the instructions of `loadLibrary` but just the instruction that invokes `loadLibrary` and the instruction `return` that has the return value, instead. To do so, DexMonitor checks the parameters of `invoke` or `return` instructions to get information about which method is called or where to return. To get such information, DexMonitor uses the class descriptor and method descriptor. The class descriptor would be obtained from `ClassObject` structure which is declared in the `dalvik/vm/oo/Object.h` file. The method name would be found from the `method` structure which is declared in the same file. The Method descriptor can be also extracted by calling the `dexProtoCopyMethodDescriptor` function.

3) INSTRUCTION MONITOR

Instruction Monitor actually outputs the monitored instructions with the thread ID managed by Dalvik and other information. Especially when instructions are associated with string object or in case of returning the string object, it can output the character string which is managed by the string object. For a method call statement, the class descriptor, method name, method descriptor of the callee method and parameters are generated. For a return statement, the return value is generated as well. string object along with the execution code.

4) EXECUTABLE FILE EXTRACTOR

To get concealed executable files, we placed hooks on the scattered points that load executable files such as `dvmJarFileOpen` or `dvmLoadNativeCode` to extract dynamically loaded files including APK file. Those hooks copy the files to the predesignated directory when the functions load executable files if the options is set in the configure file.

IV. IMPLEMENTATION AND EVALUATION

We developed a prototype of DexMonitor on Android version 4.4.4 [3]. Some modules of DexMonitor, such as OP-Interpreter and Instruction Monitor, were modified from existing Dalvik VM source codes, whereas other modules were developed from scratch. DexMonitor consists of 1,042 SLoC. The modified Android system was flashed to a Nexus 5 phone for experiments. For aforementioned reasons, we disabled JIT and modified the `dexopt` to prevent it from rewriting `invoke-virtual`, `invoke-static`, and `invoke-direct` instructions.

In addition, we developed SmaliParser which filters the extracted outputs to be compatible for analysis. DexMonitor output code is sent to SmaliParser, which is comprised of Thread Parser and Callstack Analyzer. Thread Parser increases the readability of the outputted code for effective analysis. Because Dalvik virtual machine is used on many threads, a procedure that separates the outputted code by thread is needed. To the end, Thread Parser sorts out the code outputted by DexMonitor by thread number. Callstack Analyzer distinguishes the calling stack between the caller method and callee method and is structured to display through an indentation after the callee method code and caller method code.

To evaluate the effectiveness of DexMonitor, we selected three commercial applications which are protected by obfuscation and tamper detection schemes to prevent the repackaging attacks. Target applications are one mobile antivirus application, two mobile banking applications and mobile malware. We inspected these applications with DexMonitor to find a vulnerability of their tamper detection schemes and then we emasculated them. In the figures used for showing analysis results of DexMonitor, all lines beginning with '#' refer to comments for explanations on the following instructions. The other lines are DexMonitor's outputs.

A. ANTIVIRUS APPLICATION: QIHOO 360 SECURITY V.3.4.2

Qihoo antivirus application is top ranked mobile security application in the Android play store. It scans a mobile device for searching malware in real-time as well as updates its antivirus database dynamically, also it can boost speed of the mobile device by cleaning caches or freeing memory. If this application could be hacked and distributed by adversaries, they can bypass the malware detection module to consider their malware as a normal application. Thus, in case of antivirus applications, strong protection schemes are essential.

Protection schemes applied to the application are obfuscation, anti-rooting, and tamper detection and it also loads jar files dynamically. When we just repacked the application to observe the tamper detection scheme, a crash report was appeared repetitively by its tamper detection routine as shown in Figure 4.

To find a tamper detection routine of the application, we first used ''invoke-only'' option of DexMonitor to get original control flow of methods and repacked one before the crash report was shown. Based on a simple comparison, we could figure out a method which returns a Boolean value. The method returned `1` for original one and `0` for repacked one. Also, that method was called by using the reflection because the method is not in the `classes.dex`, a main executable file of an APK file, but in the `oclt.jar` file loaded dynamically. Hence, the tamper detection routine could not be found through static analysis of the `classes.dex` file.

Furthermore, we were able to modify the method in the `oclt.jar` file to subvert the application's tamper detection scheme.

B. MOBILE BANKING APPLICATIONS

1) CASE I: H BANKING APPLICATION V.4.41

In the case of banking application, security is the prime concern for users to prevent problems such as [30]. H banking application uses obfuscation schemes including string encryption and anti-rooting schemes to protect the application. The application is also protected by a tamper detection scheme using a server so that if the application is tampered, application process will be halted along with the error report.

Server based tamper detection scheme checks integrity of the application by responding the application's request.

```
# Method flow until invoking a integrity checking
method

Lcom/qihoo/b/a/a/b;->a(…)…
Lcom/qihoo/b/a/a/b;->a(…)…
Lcom/qihoo/b/a/a/e;->a(…)…
Ljava/lang/reflect/Method;->invoke(…)…

# Methods in oclt.jar file

Lcom/qihoo360/plugin/clear/Entry; -> getModule(…)…
Lcom/qihoo360/mobilesafe/opti/dex/fx;->a(…)Z
```

(a)

```
# Loading the oclt.jar file
dvmJarFileOpen ->
       /data/data/com.qihoo.security/files/oclt.jar
…
move-result-object v4        (v4=0x42a41180)

# Invoking a integrity checking method
invoke-static args=1 @0x04bb {v4, v0, v0, v0, v0}
Lcom/qihoo360/mobilesafe/opti/dex/fx;->a(…)Z
paramCnt : 1, args 1
param1 Ljava/lang/String;
                  fec53268a38f029357056d46098c9384
# Original app returns 1
# Repacked app returns 0

retval=0x00000001
return to Lcom/qihoo360/mobilesafe/opti/dex/fx;
                                          ->a(…)Z
move-result v4          (v4=0x00000001)
if-eqz v4,-
const/4 v0,#0x01

return v0
retval=0x00000001
return to Lcom/qihoo360/plugin/clear/Entry;
                                    ->getModule(…)
```

(b)

FIGURE 4. Analysis results of Qihoo360. (a) Method flow of Qihoo 360. (b) The tamper detection routine.

However, it has a vulnerability that the application decides whether to be executed or not based on the response of the server at the method in the application. Therefore, if we find the method which deals with the response from the server and find the value from the server, we can bypass the tamper detection routine, although the integrity checking routine is operated by the server separately.

We found a branch point through DexMonitor and important values made by the method which receives the response from the server. Firstly the server sends the response using JSON object and then the result of integrity checking is saved on the device.

As shown in Figure 5, application's integrity is decided by the values stored in the register `v0` and `v1`. Therefore, the tamper detection scheme of the application could be neutralized by inserting just two lines of smali code to modify the values.

2) CASE II: W BANKING APPLICATION V.1.1.8

This banking application uses more complicated tamper detection scheme, which uses a attestation server and a sub process, than other applications tested in our case study.

```
# Executed code after receiving the JSONobject
invoke-virtual args=2 @0x7009 {v0, v1, v0, v0, v0}
(NATIVE)Ljava/lang/String;->equals(…)Z
retval=0x00000000
return from native Ljava/lang/String;->equals(…)Z
to Lcom/h-bank/ebk/channel/android/h-bank/app/
H-Intro;->onJsonReceived(…)V
# Original app returns 0
# Repacked app returns 1

move-result v1          (v1=0x00000000)

sput v1,sfield@0x2ee8
+ SPUT 'isAppProtect'=0x00000000

invoke-virtual args=1 @0x705b {v0, v0, v0, v0, v0}
Ljava/lang/StringBuilder;->toString()…
# Original app returns "result==> success"
# Repacked app returns "result==> error"

retStr = result==>> success
return to to Lcom/h-bank/ebk/channel/android/
h-bank/app/H-Intro;->onJsonReceived(…)V

move-result-object v0          (v0=0x42ef7ee0)

# Invoking a integrity checking method
invoke-static args=2 @0x514c {v1, v0, v0, v0, v0}
Leeeeee/avaava;
->…04390439…(Ljava/lang/String;Ljava/lang/String;)V
```

FIGURE 5. Analysis result of H banking application.

Sub process of W banking application interacts with the server and with the app process using the binder. The application's attestation process is comprised of challenge response authentication protocol between the server and the sub process. In their protocol, a challenge is designated as CODE_CHALLENGE and a response is designated as CODE_RESPONSE. When the W banking application is tampered, the server does not permit a user to login so that the user cannot use any banking service with the tampered application. Analyzed tamper detection architecture through DexMonitor is shown in Figure 6.

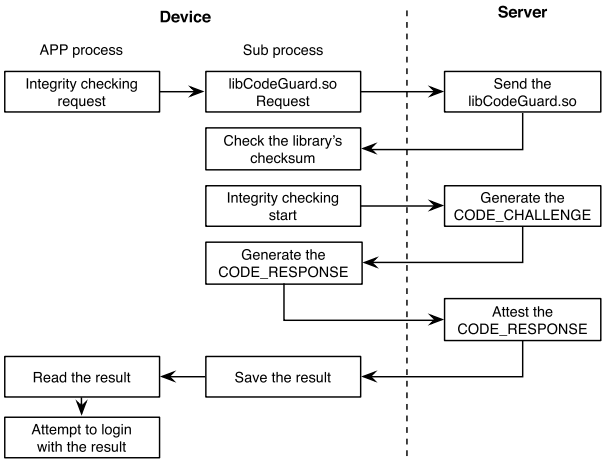


FIGURE 6. The Tamper detection architecture of W banking application.

As illustrated in Figure 6, when the applications are started, it starts its tamper detection scheme by sending a request to its sub process and then the sub process attempts to download a native library from the server. We could find the URL for downloading the library and obtain the file by monitoring with DexMonitor. The sub process starts to inspect the integrity of the application after the library's checksum is checked. It tries to get the CODE_CHALLENGE with the application's information such as name and version from the server. For the next step, the sub process sends the CODE_RESPONSE to the server as soon as the downloaded native library generates it. The sever shows examination result of the CODE_RESPONSE and the application saves the result.

When a user login to the server, the saved result is sent to the server appending with the CODE_RESPONSE, user ID, password, device ID and so on. Hence, if the server tracks the CODE_RESPONSE with respect to the CODE_CHALLENGE to attest the application, the W banking application's tamper detection scheme would not be bypassed. However, the application's tamper detection scheme is subverted by the replay attack, because the server did not record the code_challange issued by the server with the code_response. The server only checks a CG_SIGNATURE field of the result, they deny access to the server just in case of the "CODEGUARD_VERIFICATION_TOKEN_FAIL" string is included.

In addition, we found that the data used for login could be used again on other devices as shown in Figure 7. Therefore, if we make a tampered application which steals user's secret data, we can login with the stolen data on a device since the data contains encrypted password and other secret values.

```
# JSON Data used for login
_JSON_DATA={"_REQ_DATA":
{"PWD":"6b763534cc6cbe10346d869f8df0fb6…",
"_COM_SMT_UNIQUEID_FDS":"@A0201Nexus 5…",
"SEC_KEYPAD_KEYVAL":"2a5d092ac1a8dc14",
"EMP_NO":"",
"USER_ID":"ma…",
"MCN_UNQ_INF_ID":"APA91bHeaw2S8M65dNd9…",
"_CUR_HPHONE_NO":"",
"_COM_SMT_UNIQUEID":"MzUyMTM2MDYyNjAyMDA0…",
"_CUR_VERSION_NO":"1.1.6",
"CUR_APP":"PIB",
"TWO_ERR_RELOGIN_YN":"N"},
"H_LANG":"KO"}
```

FIGURE 7. JSONobject containing user's secret data.

C. REAL-WORLD MALWARE

We used DexMonitor to analyze a real-world malware which is called google app stoy [18]. When google app stoy is installed, it shows an icon similar to Google play store on the home screen. When a user clicks this icon to launch google app stoy, it shows a message 'This app is uninstalled because of Program error' and appears to be terminated. However, it still runs some Android services.

To analyze the malware, we used DexMonitor to print out the executed code of google app stoy. DexMonitor output

```
# The Target file to open and decrypt
const-string v19 string@0x00b9, string = ds
 …
# The output path and file name
retStr = /data/data/com.sdwiurse/x.zip
 …
# Calling the decrypt function
invoke-virtual args=2 {v5, … }
Lcom/kbstar/kb/android/star/DesUtils;
                                    ->decrypt([B)[B
 …
# The class name includes target method to invoke
const-string v18 string@0x00ac,
                      string = dalvik.system.DexFile
 …
invoke-static-range args=1 @0x0039 {…}
Ljava/lang/Class;
     ->forName(Ljava/lang/String;)Ljava/lang/Class;
 …
# The method name that will be invoked
const-string v19 string@0x015a, string = loadDex
 …
invoke-virtual args=3 {v0, …}
Ljava/lang/Class;
         ->getMethod(L…;)Ljava/lang/reflect/Method;
 …
# Invoke the loadDex method
invoke-virtual args=3 {v0, v1, v3, v0, v0}
Ljava/lang/reflect/Method;
                    ->invoke(L…;)Ljava/lang/Object;
 …
# The malware delete the decrypted file after
loading
invoke-virtual-range args=1 @0x0030 {…}
Ljava/io/File;->delete()Z
```

(a)

```
# Setting the target app name : 'aname'
  (The Ahnlab V3 is a Vaccine application)
const-string v0 string@0x0336,
                    string = AhnLab V3 Mobile Plus 2.0
sput-object v0,sfield@0x0799
+ SPUT 'aname'=0x42639310
 …
# Getting the installed app information
invoke-direct args=1 @0x1a3f {v0, …}
L…/UninstallerService;->getSoftName()V
 …
invoke-virtual args=1 @0x1a3e {v9,…}
L…;->getPackageManager()L…;
 …
invoke-virtual args=2 @0x0106 {v5,…}
L…r;->getInstalledApplications(I)L…;
 …
invoke-virtual args=2 @0x00ff {v0, …}
L…;->loadLabel(L…;)L…;
 …
# Getting the 'aname' field
sget-object v7,sfield@0x0799
+ SGET 'aname'=0x42639310
 …
# Comparing the 'aname' with the installed app
invoke-virtual args=2 @0x227e {v3, …}
L…;->equalsIgnoreCase(L…;)Z
retval=0x750508b000000000
return to L…/…;->getSoftName()V
 …
move-result v7          (v7=0x00000000)
 …
# If the result is not TRUE(1),searching other
installed app until the result is TRUE
if-eqz v7,+0x0026
> branch taken
```

(b)

FIGURE 8. Analysis results on a real-wrold malware. (a) The process of loading the encrypted classes. (b) The CODE_CHALLENGE.

shows google app stoy first opens a file named `ds` inside the Asset directory, which is a file that includes encrypted malicious classes. Figure 8-(a) shows how `ds` file is decompressed and the malicious classes are loaded to the memory.

After reading `ds` file inside the Asset directory, the secret key, directory where the decrypted files should be stored, file name, and the encryption algorithm must be used as parameters to decrypt the `ds` file. Also, by using Java reflection, `loadDex` method is called and loads the decrypted classes into the memory and executes them.

Since classes inside `ds` file are android services, general users cannot realize that the service is running. Even if the users realize that the services are running, it is hard to remove it completely because a deletion prevention scheme is applied to the app.

Continuously, DexMonitor was used to analyze after loading processes of the decrypted classes. As a result, `classes.dex` which is a output from the `ds` file starts services such as autoRunService, upload ContentService, UninstallerService, SoftService, and uploadPhone.

Each service removes installed vaccine programs and sends information such as phone number list, certificate, etc. to the attacker's Gmail, while sending personal information included inside the SMS to the attacker's server.

As shown in Figure 8-(b) we can confirm what the google app stoy app does by using DexMonitor.

D. PERFORMANCE EVALUATION

To evaluate the performance overhead of DexMonitor, we used two benchmark applications, namely Benchmark for Android [2] and SciMark [15]. Both applications measure the CPU and memory performance of the modified Android system by running multiple benchmark tests. The suite includes several popular benchmarks such as WhetStone, LinPack, QSort, Ackermann and Sieve of Eratosthenes. Each test is run in a native context, coded in C++, and in a managed/Dalvik context, coded in Java. In addition, both tools are used to measure efficiency of Dalvik VM and CPU.

Evaluating the performance of DexMonitor is correlated with only Dalvik VM. Thus, we ran the tools for benchmarking Dalvik, and compared the performance differences between monitored state and non-monitored stated. A monitored state means that we ran the benchmark tool with DexMonitor as monitoring all methods of the applications.

All experiments were performed on a LG Nexus 5 built with the same platform mentioned in the earlier section. The units in Android Bench are milliseconds (ms) and in SciMark are millions of floating point operations per second (Mflops). The results are shown in Figures 9 and 10.

When DexMonitor monitors instructions, the performance degradation is noticeable. This result ensures that all proposed tasks are reasonably performed as illustrated in in Figures 9 and 10. Even though the user of DexMonitor

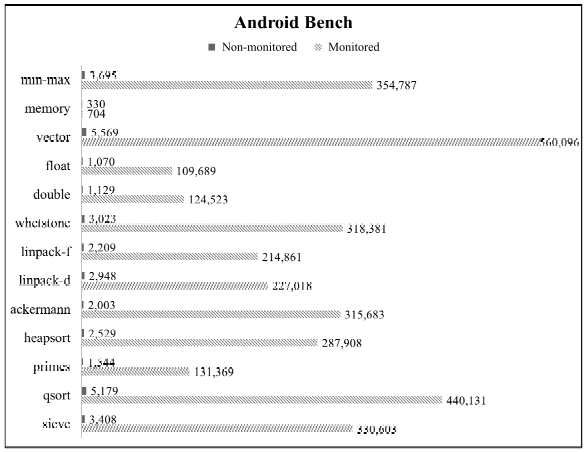


FIGURE 9. Benchmark results using Android bench.

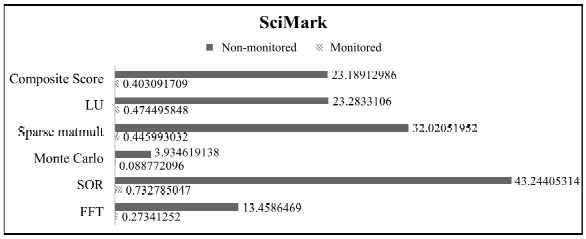


FIGURE 10. Benchmark results using SciMark.

selects a small region from the monitor, DexMonitor should monitor all methods calling instructions to determine whether the instruction is contained within the region selected by a user or not. Also, if the region includes the instruction, it needs additional work such as finding a string, printing out the instructions and so on.

V. DISCUSSION

Lipp *et al.* [33] has recently demonstrated that a malicious android application without any permission or privilege can monitor keystroke. These sophisticated attack techniques are driving the need for the sophisticated analysis methods. The first purpose of DexMonitor is to make an application analyzable. By tracing the Dalvik instruction to be executed, DexMonitor can automatically generate a trace of the actual Dalvik instructions that are executed by an application and can output its executable files. Hence, DexMonitor effectively can be used for analyzing applications which employ protection schemes based on code concealment such as code encryption and dynamic loading, which cannot be analyzed by static analysis. Providing unveiled executable files regardless of location of files or protection methods, it would be great service to analyzers.

There are several challenges that must be tackled to transform this approach into a complete analysis system. First, there is the issue of native code execution. Since it executes outside the Dalvik VM, DexMonitor is unable to see and trace it. However, we can reveal JNI code which calls Android APIs or user methods. Second, as addressed in traditional dynamic analysis environment for malware [26], malware can use timing analysis to detect analysis environments. As DexMonitor has significant overhead, it could be possible for malware to use this overhead as a timing channel. However, as our initial implementation of DexMonitor is a prototype, we believe that the performance could be optimized and improved in the future. Finally, there is the significant challenge of analyzing Virtualization-based Protection which is another critical challenge in traditional desktop software and needs more considerable attention.

Since the Lollipop (Android version 5.0), Dalvik VM has been replaced with the ART runtime to improve the performance. The ART runtime uses an OAT file produced by ahead-of-time (AOT) compilation instead of a dex file. However, android applications are still implemented by using Java and packed as the APK file format containing a dex file which is the input of AOT compilation. In addition, android uses the Dalvik VM in the ART runtime since some techniques cannot be executed on the ART runtime [5]. Therefore, there is a distinct possibility in implementing DexMonitor on a higher version of android than the current prototype. Our prototype of DexMonitor can be even used to analysis up-to-date applications if those applications are implemented with minimum SDK version lower than 4.4.4.

VI. RELATED WORK

We believe that DexMonitor can represent a first step toward a fully automated deobfuscation system, since it can reveal all hidden code and can generate the detailed tracing results such as parameters, fields, retrun values, strings as well as executable files. Deobfuscation is an important part of the mobile application security ecosystem, even though the challenge of automatically deobfuscating a mobile application is difficult. It is clear that there will be an arm race between obfuscators and deobfuscators. The centralized application store model allows mobile operating system developers to vet applications (unlike the traditional desktop computing environment). To evaluate potential applications for being included the in centralized market, applications should be analyzed, both statically and dynamically, against malicious behaviors. Obfuscation tools allow malicious developers to hide and mask malicious behaviors, thus bypassing the market vetting process. Therefore, automated deobfuscation is an important technique to reveal the hidden behaviors of applications so that traditional static analysis techniques can be further applied for protecting users.

DexMonitor's related works as follows. RAMSES [25] is a static analysis tool for characterizing malware by using constant strings. Even though this tool can be used as a first analysis to filter benign applications out, it cannot be utilized when the string encryption scheme is simply applied on applications. Li *et al.* [32] proposed a systematic procedure for recovering malicious events of Android malware. However, they manually recovered encrypted strings which is one of the most important data containing server addresses, command names, etc. While DexMonitor can output decrypted strings automatically.

TraceDroid framework [37] was proposed for dynamic analysis of Android applications to detect suspicious, possibly malicious applications. However, it only traces API calls, and thus, it cannot provide detail information on an executable itself for a complete analysis. Kim *et al.* [31] designed DWroidDump to extract the main executable code from the memory as an effective analysis preliminary. DexMonitor has this functionality in more efficient way, not requiring complicated memory analysis technique. Furthermore, DexMonitor can be used to track more detail information regarding dynamically loaded executable such as where it comes from or when it is loaded.

Bichsel *et al.* [22] proposed a method to deobfuscate layout obfuscations schemes such as the identifier renaming. It showed very promising results by predicting names of obfuscated identifiers through their statistical model. The most recently, Wong and Lie [41] have presented TIRO, which can automatically detect and reverse language-based and runtime-based obfuscation via dynamic instrumentations. We believe that DexMonitor can be utilized generally as a groundwork for research works similar to the above ones by providing accurate information on executed instructions and executable files.

VII. CONCLUSION

Since statically analyzing (potentially malicious) Android applications is tedious and requires tremendous expertise, we proposed an new approach to analyze Android bytecode. The core idea of our proposed approach is to place hooks in the Dalvik VM at the point where a Dalvik instruction is about to be executed. We have chosen an in-the-box design for its advantages over a virtual machine inspection solution by directly modifying the Dalvik VM. We have shown the effectiveness and performance of our approach by evaluating on various Android applications.

APPENDIX A
OUTPUTS OF DexMonitor

A. EXECUTED BYTECODE

DexMonitor prints out all executed bytecode in the rage of a user selects. It can handle all kinds of bytecode of Dalvik VM. Also, it provides the detailed information depending on the operation code such as invoke, return, and so on. Listing 1 shows an example of bytecode that DexMonitor generated.

Basic form of the output bytecode is like |Thread number|Operations. In addition, as you can see in Listing 1, all executable files loaded dynamically are remained in the log with the function name which loads the file and, when the option for parameters is set, parameters are printed like lines 5-6. In cases of operations regarding fields such as sget or sput, DexMonitor provides type, name and value of the field. Also, if the branch or jump operations are occured, DexMonitor puts "branch taken" string. Besides, we can know new values stored in registers by move operations as indicated in lines 21 and 25.

```
1  dvmJarFileOpen -> /data/app/com.example.
       dexmonitor.apk
2  dvmJarFileOpen -> /data/data/com.example.
       dexmonitor/files/example.jar
3  dvmLoadnativeCode -> /data/data/com.example.
       dexmonitor/files/libexample.so
4  |1|invoke-static args=1 @0x04b9 {v1, v0, v0, v0
       , v0}
5  |1|[fp=0x6d455c04] Lcom/example/dexmonitor/a;->
       a(Landroid/content/Context;)Z
6  |1|paramCnt : 1, args 1
7  |1|param1 Lcom/example/dexmonitor/a/
       exampleService; 0x42a7f2b0
8  |1|sget-object v0,Lcom/example/dexmonitor/a;->
       exampleField:Ljava/lang/String;
       sfield@0x0001
9  |1|+ SGET 'exampleField'=0x42ec3b30
10 |1|const/4 v0,#0x00
11 |1|invoke-static args=1 @0x04bb {v4, v0, v0, v0
       , v0}
12 |1|[fp=0x6d455bd8] Lcom/example/dexmonitor/a;->
       b()Ljava/lang/String;
13 |1|retStr=This is an example string
14 |1|return to Lcom/example/dexmonitor/a;->a(
       Landroid/content/Context;)Z [fp=0x6d455c04]
15 |1|move-result-object v1          (v1=0x429f1d18
       )
16 |1|const-string v2 string@0x0015, string =
       Another String Example
17 |1|invoke-virtual args=2 @0xbd06 {v1, v2, v0,
       v0, v0}
18 |1|[fp=0x6d427d5c] (NATIVE)Ljava/lang/String;->
       equals(Ljava/lang/Object;)Z
19 |1|retval=0x00000001
20 |1|return from native Ljava/lang/String;->
       equals(Ljava/lang/Object;)Z to Lcom/example
       /dexmonitor/a;->a(Landroid/content/Context
       ;)Z [fp=0x6d455c04]
21 |1|move-result v1          (v1=0x00000000)
22 |1|if-eqz v1,+0x0004
23 |1|> branch taken
24 |1|const/16 v3,#0x0001
25 |1|move/from16 v0,v3       (v0=0x00000001)
26 |1|return v0
27 |1|retval=0x00000000
```

Listing 1. Example of bytecode extracted by DexMonitor

B. EXECUTABLE FILES

Listing 2 shows an example of executable files of DexMonitor. The prefix number is an order of loaded files and the rest of it is a full path of the file on Android device; to save the files in the designated directory, a forward slash was changed to an underline.

```
1  shell@hammerhead:/data/am$ ls
2  0_data_app_com.example.dexmonitor.apk
3  1_data_data_com.example.
       dexmonitor_files_example.jar
4  2_data_data_com.example.
       dexmonitor_files_libexample.so
```

Listing 2. Example of executable files extracted by DexMonitor

APPENDIX B
SCREENSHOTS OF CASE STUDIES

Figure 11, 12, and 13 show the evaluation results of each application mentioned in Section IV. It illustrates the

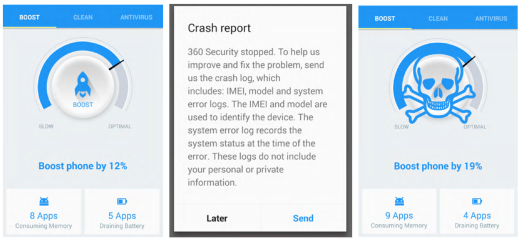


FIGURE 11. Screenshots of analyzed Qihoo 360 App.

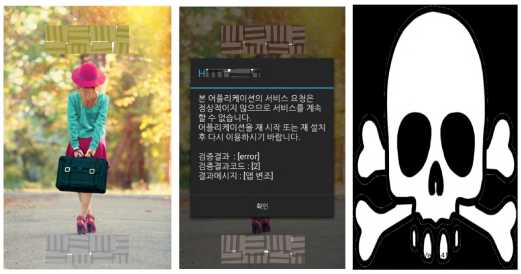


FIGURE 12. Screenshots of analyzed H Banking App.

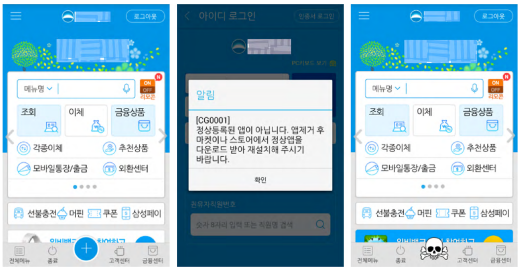


FIGURE 13. Screenshots of analyzed W Banking App.

snapshots of the original application, the tamper detection error message, and repacked application by using DexMonitor, respectively.

REFERENCES

[1] *Allatori Java Obfuscator*. Accessed: Sep. 6, 2018. [Online]. Available: http://www.allatori.com/

[2] *Android Bench*. Accessed: Sep. 6, 2018. [Online]. Available: https://play.google.com/store/apps/details?id=net.warp7.nativevsdalvik

[3] *Android Open Source Project*. Accessed: Sep. 6, 2018. [Online]. Available: https://source.android.com/

[4] P. Schulz and F. Matenaar, "Android reverse engineering and defenses," Bluebox Labs, Tech. Rep., 2013.

[5] *Art and Dalvik*. Accessed: Sep. 6, 2018. [Online]. Available: https://source.android.com/devices/tech/dalvik/index.html

[6] *Dalvik Instruction*. Accessed: Sep. 6, 2018. [Online]. Available: https://source.android.com/devices/tech/dalvik/instruction-formats.html

[7] *Dalvik JIT*. Accessed: Sep. 6, 2018. [Online]. Available: http://android-developers.blogspot.kr/2010/05/dalvi k-jit.html

[8] *Dalvik Optimization and Verification With Dexopt*. Accessed: Sep. 6, 2018. [Online]. Available: http://newandroidbook.com/code/android-5.1.1_r24/dalvik/docs/dexopt.html

[9] Dex2jar. *Dex2jar Tool*. Accessed: Sep. 6, 2018. [Online]. Available: https://github.com/pxb1988/dex2jar

[10] *DexGuard*. Accessed: Sep. 6, 2018. [Online]. Available: https://www.saikoa.com/dexguard

[11] *Dexprotector by Licel*. Accessed: Sep. 6, 2018. [Online]. Available: http://dexprotector.com/

[12] *DroidScope*. Accessed: Sep. 6, 2018. [Online]. Available: https://code.google.com/p/decaf-platform/wiki/DroidScope

[13] *Fake Encryption Sample*. Accessed: Sep. 6, 2018. [Online]. Available: https://github.com/blueboxsecurity/DalvikBytecodeTampering

[14] Java Decompiler. *JD Project: Java Decompiler*. Accessed: Sep. 6, 2018. [Online]. Available: http://jd.benow.ca

[15] SciMark. *SciMark 2.0: A Java Benchmark for Scientific and Numerical Computing*. Accessed: Sep. 6, 2018. [Online]. Available: http://math.nist.gov/scimark2/index.html

[16] Smali. *Smali: An Assembler for the Dex Format*. Accessed: Sep. 6, 2018. [Online]. Available: https://code.google.com/p/smali/

[17] *Stringer JAVA Obfuscator*. Accessed: Sep. 6, 2018. [Online]. Available: https://jfxstore.com/stringer/

[18] *What are you Doing?—DSEncrypt Malware*. Accessed: Sep. 6, 2018. [Online]. Available: https://www.fireeye.com/blog/threat-research/2014/06/what-are-you-doing-dsencrypt-malware.html

[19] AppBrain. *Number of Available Android Applications*. Accessed: Sep. 6, 2018. [Online]. Available: http://www.appbrain.com/stats/number-of-android-apps

[20] A. Apvrille, "Playing hide and seek with Dalvik executables," Hacktivity, Budapest, Hungary, 2013.

[21] D. Aucsmith, "Tamper resistant software: An implementation," in *Information Hiding*. Cambridge, U.K.: Springer, 1996, pp. 317–333.

[22] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of Android applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 343–355.

[23] B. Cheng and B. Buzbee, "A JIT compiler for Android's Dalvik VM," in *Proc. Google I/O Developer Conf.*, 2010, pp. 1–32.

[24] H. Cho, J. Lim, H. Kim, and J. H. Yi, "Anti-debugging scheme for protecting mobile apps on Android platform," *J. Supercomput.*, vol. 72, no. 1, pp. 232–246, 2016.

[25] L. Dolberg, Q. Jérôme, J. François, R. State, and T. Engel, "RAMSES: Revealing Android malware through string extraction and selection," in *Proc. Int. Conf. Secur. Privacy Commun. Netw.* Singapore: Springer, 2014, pp. 498–506.

[26] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, p. 6, 2012.

[27] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of Android application security," in *Proc. USENIX Secur. Symp.*, vol. 2, 2011, p. 2.

[28] T. W. Hou, H. Y. Chen, and M. H. Tsai, "Three control flow obfuscation methods for Java software," *IEE Proc. Softw.*, vol. 153, no. 2, pp. 80–86, Apr. 2006.

[29] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu, "Morpheus: Automatically generating heuristics to detect Android emulators," in *Proc. 30th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2014, pp. 216–225.

[30] J.-H. Jung, J. Y. Kim, H.-C. Lee, and J. H. Yi, "Repackaging attack on Android banking applications and its countermeasures," *Wireless Pers. Commun.*, vol. 73, no. 4, pp. 1421–1437, 2013.

[31] D. Kim, J. Kwak, and J. Ryou, "Dwroiddump: Executable code extraction from Android applications for malware analysis," *Int. J. Distrib. Sensor Netw.*, vol. 11, no. 9, p. 379682, 2015.

[32] J. Li, D. Gu, and Y. Luo, "Android malware forensics: Reconstruction of malicious events," in *Proc. 32nd Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, Jun. 2012, pp. 552–558.

[33] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *Proc. 25th USENIX Secur. Symp.* Berkeley, CA, USA: USENIX, 2016, pp. 549–564.

[34] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of Android malware," in *Proc. 7th Eur. Workshop Syst. Secur.*, 2014, p. 5.

[35] Y. Piao, J.-H. Jung, and J. H. Yi, "Server-based code obfuscation scheme for APK tamper detection," *Secur. Commun. Netw.*, vol. 9, no. 6, pp. 457–467, 2014.

[36] Z. Tang, X. Chen, D. Fang, and F. Chen, "Research on java software protection with the obfuscation in identifier renaming," in *Proc. 4th Int. Conf. Innov. Comput., Inf. Control (ICICIC)*, Dec. 2009, pp. 1067–1071.

[37] V. Van Der Veen, H. Bos, and C. Rossow, "Dynamic analysis of Android malware," M.S. thesis, Dept. Comput. Sci., Internet Web Technol., VU University Amsterdam, Amsterdam, The Netherlands, 2013.

[38] T. Vidas and N. Christin, "Evading Android runtime analysis via sandbox detection," in *Proc. 9th ACM Symp. Inf., Comput. Commun. Secur. (ASIACCS)*, 2014, pp. 447–458.
[39] R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Steal this movie: Automatically bypassing DRM protection in streaming media services," in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 687–702.
[40] C. Warren. (Jul. 2013). *Google Play Hits 1 Million Apps*. [Online]. Available: http://mashable.com/2013/07/24/google-play-1-million/, .
[41] M. Y. Wong and D. Lie, "Tackling runtime-based obfuscation in Android with TIRO," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1247–1262.
[42] L.-K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. USENIX Secur. Symp.*, 2012, pp. 569–584.
[43] Z. Zhao, G.-J. Ahn, and H. Hu, "Automatic extraction of secrets from malware," in *Proc. 18th Work. Conf. Reverse Eng. (WCRE)*, Oct. 2011, pp. 159–168.
[44] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang, "Divilar: Diversifying intermediate language for anti-repackaging on Android platform," in *Proc. 4th ACM Conf. Data Appl. Secur. Privacy (CODASPY)*, 2014, pp. 199–210.

HAEHYUN CHO received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, South Korea, in 2013 and 2015, respectively. He is currently pursuing the Ph.D. degree in computer science with the School of Computing, Informatics and Decision Systems Engineering, Arizona State University, with a focus on information assurance. His research interest lies in the field of systems security that is to address and discover security concerns stemmed from insecure designs.

JEONG HYUN YI received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, South Korea, in 1993 and 1995, respectively, and the Ph.D. degree in information and computer science from the University of Californian at Irvine in 2005. He was a Member of Research Staff with the Electronics and Telecommunications Research Institute, South Korea, from 1995 to 2001. From 2000 to 2001, he was a Guest Researcher with the National Institute of Standards and Technology, Gaithersburg, MD, USA. He was a Principal Researcher with the Samsung Advanced Institute of Technology, South Korea, from 2005 to 2008. He is currently an Associate Professor with the School of Software and the Director of the Cyber Security Research Center, Soongsil University, Seoul. His research interests include mobile security and privacy, IoT security, and applied cryptography.

GAIL-JOON AHN is currently a Professor in computer science and engineering and the Director of the Center for Cybersecurity and Digital Forensics, Arizona State University (ASU).

Prior to joining ASU in 2008, he was an Associate Professor with the College of Computing and Informatics and the Founding Director of the Center for Digital Identity and Cyber Defense Research, The University of North Carolina, Charlotte. He has authored more than 150 refereed research papers. His research was supported by NSF, NSA, DoD, ONR, DoE, DoJ, the Bank of America, CISCO, GoDaddy, Hewlett Packard, Google, Microsoft, and the Robert Wood Johnson Foundation. His research interests include security analytics and big data driven security intelligence, vulnerability and risk management, access control and security architecture for distributed systems, identity and privacy management, cyber crime analysis, security-enhanced computing platforms, and formal models for computer security device. He is currently the Information Director of the ACM Special Interest Group on security, audit, and control. He was a recipient of the U.S. Department of Energy Early Career Principal Investigator Award, the Educator of the Year Award from the Federal Information Systems Security Educators' Association, and the Best Researcher Award from CIDSE. He is the Steering Committee Chair of the ACM Symposium on Access Control Models and Technologies. He is also the General Chair of the ACM Conference on Computer and Communications Security 2014. He serves as the Associate Editor-in-Chief of the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING and an Associate Editor of the *ACM Transactions on Information and Systems Security*. He is on the Editorial Board of *Computers and Security*.