Hybrid Dynamic Analysis for Android Malware Protected by Anti-Analysis Techniques with DOOLDA

Sunjun Lee, Yonggu Shin, Minseong Choi, Haehyun Cho, Jeong Hyun Yi*

School of Software, Soongsil University, Republic of Korea
starj1024@gmail.com, tls09611@gmail.com, gigacms@gmail.com, haehyun@ssu.ac.kr, jhyi@ssu.ac.kr

Abstract

A lot of the recently reported malware is equipped with the anti-analysis techniques (e.g., anti-emulation, anti-debugging, etc.) for preventing from being the analyzed, which can delay detection and make malware alive for a longer period. Therefore, it is of the great importance of developing automated approaches to defeat such anti-analysis techniques so that we can handle and effectively mitigate numerous malware. In this paper, by analyzing 1,535 malicious applications, we found that 18.31% of them equipped with anti-analysis techniques. Next, we propose a novel, dynamic analyzer, named DOOLDA, for automatically invalidating anti-analysis techniques through dynamic instrumentation. DOOLDA monitors executions of Android applications' entire code layers (i.e., bytecode and native code). Based on monitoring results, DOOLDA finds the code related to anti-analysis techniques and invalidates the anti-analysis techniques by instrumenting it. To demonstrate the effectiveness of DOOLDA, we show that it can invalidate all known anti-analysis techniques. Also, we compare DOOLDA with other dynamic analyzers.

Keywords: Malware analysis, Dynamic analysis, Mobile security

1 Introduction

Mobile malware targeting Android devices is not only increasing in number but also is evolving to avoid various detection techniques. Consequently, it is getting more difficult to automatically analyze them. As various code protection techniques developed to protect mobile applications began to be applied to malware. For example, adversaries started using obfuscation and packing techniques to hinder static analyses [1-5], which derived security experts to use dynamic analysis approaches [6-7]. For thwarting dynamic analyzers, attackers are using anti-analysis techniques so that they can prevent them from being analyzed.

We have observed several security incidents caused by malware implementing anti-analysis techniques. By using such anti-analysis techniques, malware called Skinner could stay in Google Play for two months without detection. For the two months, Skinner tracked a lot of users' locations and actions and can execute code from its command and control server without the users' permission. Specifically, in order to avoid detection, Skinner used anti-emulation and anti-debugging techniques. As another example, Avaddon, which was used in various cyber-attacks in 2020 and leaked more than 574GB of data from 23 companies, exploited anti-debugging techniques to protect itself from being analyzed by security experts.

In this work, our goal is to automatically invalidate anti-emulation and anti-debugging techniques by which we can quickly find malicious behaviors from them and respond to them. To this end, we first investigated of existing anti-analysis techniques. We, then, reclassified anti-analysis techniques based on features collected through the investigation and we designed an invalidation strategy for each technique.

Also, we implemented a dynamic instrumentation tool, called DOOLDA, that monitors and instruments an Android application, for automatically invalidating anti-analysis techniques. Because Android applications can have two different types of code (i.e., bytecode and native code), the architecture of DOOLDA consists of two parts: DaBIDA which instruments the bytecode, and DaNIDA which instruments the native code. DOOLDA monitors an application's execution to find code implementing known anti-analysis techniques. If it found the code, DOOLDA hooks the code for instrumenting. And if not, it just skips the code instrumenting step. In the instrumenting step, DOOLDA instruments the code to invalidate anti-analysis techniques. Also, DOOLDA records each instruction with data to help security analysts.

To demonstrate the effectiveness of DOOLDA, we performed experiments with real-world malware using the Android Malware Dataset (AMD) [8], finding and invalidating anti-analysis techniques implemented in them. Our evaluation results show that how much Android malware uses anti-analysis techniques and DOOLDA can effectively invalidate anti-analysis techniques in real-world malware.

In summary, this paper makes the following contributions: (1) We surveyed the existing studies, reclassified the collected signatures, and established an invalidation strategy for each of the anti-analysis techniques; (2) We propose a novel approach, named DOOLDA, that automatically invalidates anti-analysis techniques used in Android malware; (3) We evaluate DOOLDA with real-world malware.

*Corresponding Author: Haehyun Cho; E-mail: haehyun@ssu.ac.kr
DOI: 10.53106/160792642024032502003

2 Background

2.1 Executables in Android Applications

Android applications are deployed by using the APK file format. The APK file contains has two different types of code: Bytecode, and native code. The bytecode is stored in a Dalvik executable (.dex) file and the native code is in a shared object (.so) file. In Android applications, we use the Java Native Interface (JNI) which defines a way to interact between the bytecode and native code. Android applications can equip anti-analysis techniques (e.g., anti-debugging) in each code layer to prevent being analyzed. Malicious applications, also, use those techniques to make analysis very difficult so that they can hide their internal logic or behaviors. Therefore, to successfully analyze such Android malware, we should identify and instrument both the bytecode and native code.

Listing 1. A motivating example: The Android malware with the anti-emulation techniques

```
1  public class ActivityStart extends Activity {
2    protected void onCreate(Bundle arg5) {
3      ...
4      if(!n.a(this.getApplicationContext())) {
5        n.b(this.getAppcationContext());
6        ...
7      }
8    }
9
10   public class n {
11     public static boolean a (Context arg6) {
12       boolean v1 = false;
13       boolean v4 = arg6.getSystemService("phone")
14                                  .getDeviceId()
15                                  .equals("000000000000000");
16       int v0 = (Build.MODEL.contains("google_sdk")) || ... ?
         1 : 0;
17       int v3 = !Build.DEVICE.startsWith("generic") || ... ?
         0 : 1;
18
19       if ((v4) || (v0 != 0) || (v3 != 0)) {
20         v1 = true;
21       }
22       return v1;
23     }
24
25     public static void b (Context arg7) {
26       if (Build$VERSION.SDK_INT >= 19) {
27         arg7.getSystemService("alarm") ...
28         new Intent(arg7, AlarmReceiverKnock.class), ... );
29       } else {
30         arg7.startService(new Intent(arg7, knock.class));
31       }
32     ...
33 }
```

2.2 Motivating Example

Listing 1 shows a code snippet of real-world Android malware that uses anti-analysis techniques. In the code snippet, the method b of class n enables the alarm receiver to steal the contents of SMS and MMS services of a device. It is invoked by *onCreate* method and *onCreate* method will be invoked when the *ActivityStart* activity is initialized. However, the method b will not be invoked if an application executes on an emulator. The method a implements the anti-emulation techniques that check whether or not the application is running on an emulator because dynamic analyzers usually employ an emulator. To this end, it checks system properties such as *DeviceId*, *Build.MODEL*, *Build.DEVICE*, etc. If the method finds that the application is not running on an actual device, the method b will not be invoked and thus we cannot analyze the application's malicious behavior as a result. The code presented in Listing 1 can effectively prevent being analyzed from existing dynamic analyzers based on the emulator [9-11] and dynamic analyzers based on customized system [6-7, 12].

Other than the anti-emulation techniques such as the given example, the anti-debugging techniques can thwart the dynamic analyzer using debugging features [13-14]. Such techniques check if a debugging process exists in the process list or checks whether the *PTRACE* system call is used or not. In addition, some techniques scan network ports to examine a device is being traced or debugged remotely to avoid debuggers.

3 Overview

In this paper, we aim to provide an automated solution to thwart anti-analysis techniques used in advanced malware. To this end, we propose a dynamic analysis platform, named DOOLDA, that can effectively invalidate anti-analysis techniques. DOOLDA, also, can be used to dynamically instrument the other code of malware.

Figure 1 shows how DOOLDA analyzes Android malware equipped with the anti-analysis techniques via instrumenting its code during runtime. DOOLDA first finds code that implements anti-analysis techniques and invalidates it by dynamically instrumenting it.

The both modules of DoolDA operate based on a Dynamic Binary Instrumentation (DBI) framework. DaBIDA uses Android's built-in instrumentation module, and DaNIDA uses Valgrind, which dynamically instrument code while malware is running. Therefore, there is no need to find the code snippet through reversing malware. Also, there is no need to recompile after the instrumentation. DoolDA simply intervenes between DBI tool's IR (Intermediate Representation) translation and IR compilation phases to monitor and instrument code in malware.

(1) Monitoring Code: In the monitoring phase, DOOLDA examines the IR translated by the DBI tool. This allows DOOLDA to know the actual behavior of the code. DOOLDA monitors the code before it executes using IR and checks whether the code contains anti-analysis techniques or not. If so, DOOLDA hooks the code for instrumenting. Otherwise, it just skips the code instrumentation phase.

(2) Instrumenting Code: In the Instrumenting phase, based on the results inspected in the monitoring phase, DOOLDA instruments the code to invalidate the anti-analysis techniques. For invalidating anti-analysis techniques, DOOLDA replaces code or data used for anti-

analysis techniques with code that makes an application keeps executing. Furthermore, DOOLDA can be used for instrumenting the other code of malware: it can dynamically patch code that will execute at any point while runtime.

The above methodology can also be implemented by using a static analysis instead of instrumentation through DBI. However, there is a limitation that a static analysis-based approach cannot be used when anti-analysis techniques such as obfuscation and packing are applied onto applications. Therefore, we designed DOOLDA based on the dynamic analysis through DBI.

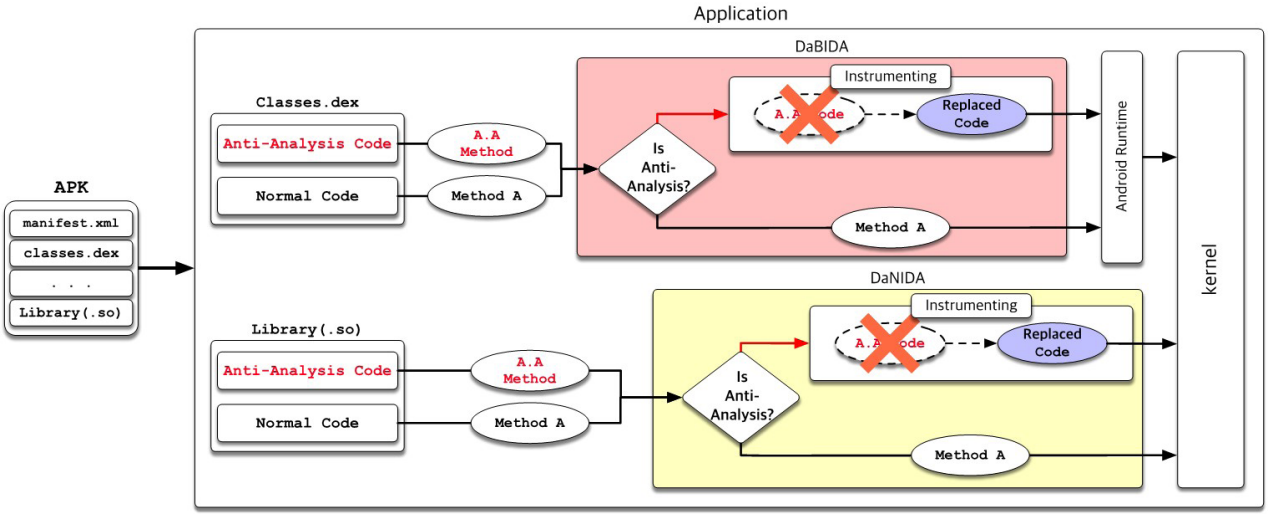

Figure 1. The workflow of DOOLDA

4 DOOLDA

4.1 Anti-Analysis Techniques

To achieve our goal, we first analyzed real-world malware and surveyed state-of-art research work on anti-analysis and anti-anti-analysis techniques of Android applications [7, 15-24]. As the results, we classify the anti-analysis techniques in three-fold as follows.

(1) Anti-rooting techniques: To prevent executions on a rooted device, the anti-rooting techniques are to figure out that the device is rooted or not. The most widely used techniques are checking whether the rooting-related applications are installed (AR1 in Table 1) and checking the existence of binary and directory that only can be seen on a rooted device (AR2 in Table 1). Furthermore, there is a technique that checks the system properties related to root permissions (AR3 in Table 1).

(2) Anti-emulation techniques: In general, the emulation environment is used to test or analyze an application. Most of automated analyzers run on emulators customized for their purposes [6, 10-11, 25]. Hence, for preventing executions on an emulator, the anti-emulation techniques check the traits that imply the emulation environment. An emulator such as Android Virtual Device (AVD) has virtualized hardware and a system with arbitrary data. By checking the configuration data of Android system, an application can identify whether it is running on a real device or not. The well-known techniques are checking the hardware configuration (AE1 in Table 1), checking build information (AE2 in Table 1) and checking system properties (AE3 in Table 1). Also, checking files related to an emulator is also widely used to figure out an emulation environment (AE4 in Table 1), it is because an emulator has specific binaries and directories that cannot be seen on a real device. In addition, an emulation environment has a lot of additional layers to compose emulated components and they make runtime performance low. By using this feature, there is an anti-emulation technique that checks an execution time of a specific task (AE5 in Table 1).

(3) Anti-debugging techniques: On Android, we can check installed or running debugger-related programs (AD1 in Table 1) and the usage of system calls (AD2 in Table 1) to figure out a device is being debugged or not. Also, checking the activation status of JDWP (AD3 in Table 1) and checking debuggable flags (AD4 in Table 1) can be used to identify the existence of a debugger. Moreover, similar to the anti-emulation techniques, there is an anti-debugging technique that checks an execution time (AD5 in Table 1).

The summary of our investigation about the anti-analysis techniques is illustrated in Table 1. As a result, we found that most of the anti-analysis techniques check the device status such as the system properties (AR3, AE2, AE3, AD2, AD3, AD4), hardware information (AE1), running processes, or files existence (AR1, AR2, AE4, AD1). Also, we confirm that the Android malware checks the device status in 2 layers not just native system-level but also Android Framework's bytecode level. This means covering only 1 layer (instrumenting only bytecode level [26] or native system-level [27]) is not sufficient. For obtaining a status of the device, anti-analysis techniques use pre-defined properties and APIs that the Android framework offers, or Linux standard library functions provide. Using this common ground, we set up a strategy in DOOLDA (DaNIDA and

DaBIDA to cover the bytecode and native code) that always returns the correct value collected from a normal real device. The collected data is stored with the related properties, APIs, and functions. Therefore when anti-analysis techniques try to check the device status in a known way, we can deceive the checking process by hooking the related code and returning the stored correct data. In the case of the timing-based anti-analysis techniques (AE5 and AD5), we can deceive it by returning a fixed time value because they usually check that the execution time exceeds the limit or not.

Table 1. The categorized anti-analysis techniques and detailed classification results

Category	Types	Description	Practical cases
Anti-rooting techniques	AR1: Check rooting applications	Checks whether rooting-related applications are installed.	com.devadvance.rootcloak, com.grarak.kerneladiutor, com.jrummy.root.browserfree, com.koushikdutta.superuser, etc
	AR2: Check binary files / directories	Check the existence of files / directories that only can be seen on rooted device.	/system/xbin/su, /system/bin/su, /system/xbin/.../xbin/su, /system/xbin/busybox, SuperSU, Magisk, Luckypatcher, etc
	AR3: Check system properties	Check the value of system properties related to root permission.	ro.secure, ro.debuggable, service.adb.toor, etc.
Anti-emulation techniques	AE1: Check hardware configurations	Checks hardware information of the device	TelephonyManager.getDeviceId(), TelephonyManger.getNetwork*(), TelephonyManager.getSimSericalNumber(), Android device ID, IMEI, IMSI, MAC addresss, /proc/cpuinfo (goldfish is not allowed), etc
	AE2: Checks build Information	Checks whether it has the build information that real devices do not have.	Build.PRODUCT, Build.BOARD, Build.BRAND, Build.DEVICE, Build.FINGERPRINT, Build.ID, Build.MODEL, Build.TAGS, etc.
	AE3: Check system properties	Checks whether it has the system properties that real devices do not have.	ro.bootloader, ro.bootmode, ro.hardware, ro.product.model, ro.product.device, ro.produce.name, init.svc.qemud, etc.
	AE4: Check binary files / directories	Checks whether the specific files that imply the emulation environment exists.	/dev/socket/qemud, /dev/qemu_pipe, /proc/tty/drivers, /system/lib/libc_malloc_debug_qemu.so, /sys/qemu_trace, /system/bin/qemu-props, etc.
	AE5: Timing checking	Checks the delay of execution time caused by emulation environment.	gettimeofday(), currentTimeMillis(), System.nanoTime(), etc.
Anti-debugging technique	AD1: Check debugger programs	Check the debugger programs are running or installed on the device.	/proc/self/maps, ps (for detecting gdb, Frida-agent-*.so, etc).
	AD2: Check system calls C	Checks the usage of system calls such as ptrace, strace, etc.	ptrace, strace, /proc/self/status (for checking TracePid), etc.
	AD3: Checks the usage of JDWP	Checks the JDWP is activated in Dalvik VM.	DVMGlobals, JdwpAdbState, count member of BreakpointSet structure, etc.
	AD4: Checks debuggable flag	Checks the value of debuggable flag of the application.	BuildConfig.BUILD_TYPE, BuildConfig.DEBUG, android:debuggable, etc.
	AD5: Timing checking	Checks the delay of execution time caused by breakpoints.	gettimeofday(), currentTimeMillis(), System.nanoTime(), etc

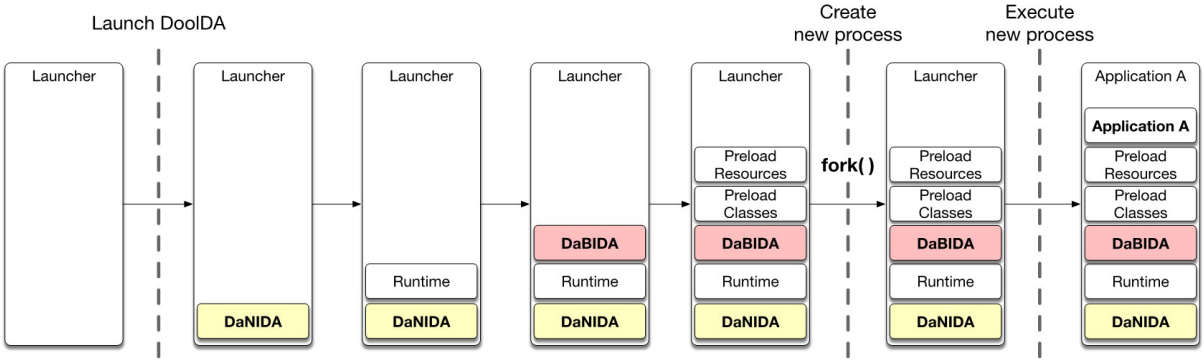

Figure 2. The process of executing a target application

Table 2. Instrumentation events and handler interfaces in ART

Event	Event handler interface
MethodEntered	MethodEntered(Thread* thread, Handle<mirror::Object>this_object, ArtMethod* method, uint32_t dex_pc)
MethodExited	MethodExited(Thread* thread, Handle<mirror::Object>this_object, ArtMethod* method, uint32_t dex_pc, const JValue& return_value)
MethodUnwind	MethodUnwind(Thread* thread, Handle<mirror::Object>, ArtMethod* method, uint32_t dex_pc)
DexPcMoved	DexPcMoved(Thread* thread, Handle<mirror::Object>this_object, ArtMethod* method, uint32_t dex_pc)
FieldRead	FieldRead(Thread* thread, Handle<mirror::Object>this_object, ArtMethod* method, uint32_t dex_pc, ArtField* field)
FieldWritten	FieldWritten(Thread* thread, Handle<mirror::Object>this_object, ArtMethod* method, uint32_t dex_pc, ArtField* field, const JValue& field_value)
ExceptionCaught	ExceptionCaught(Thread* thread, mirror::Throwable* exception_object)
Branch	Branch(Thread* thread, ArtMethod* method, uint32_t dex_pc, int32_t offset)
InvokeVirtualOrInterface	InvokeVirtualOrInterface(Thread* thread, Handle<mirror::Object>this_object, ArtMethod* method, uint32_t dex_pc, ArtMethod* target)

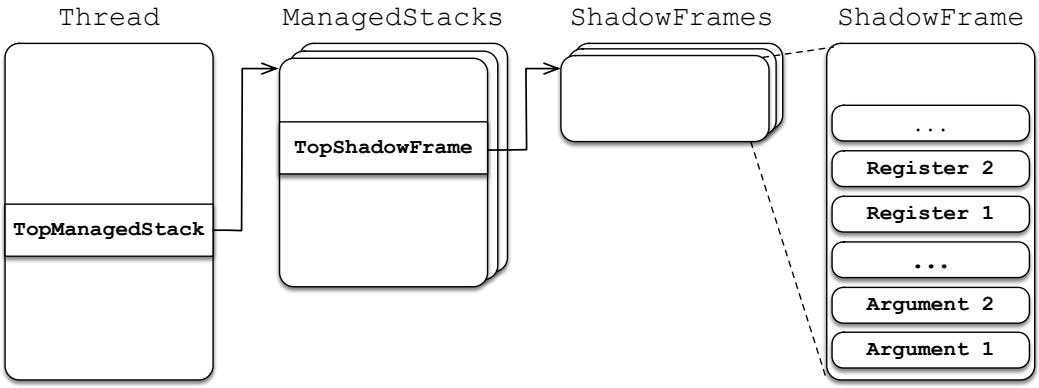

Figure 3. The virtual stack structure in ART

4.2 Application Launcher

DOOLDA uses a dedicated application launcher to dynamically instrument an application's bytecode and native code. The module launches an application after loading DaNIDA and DaBIDA into a process. We implement it by modifying the *app_process* of Android [28] which performs a series of pre-processing to execute an application. With the launcher, DOOLDA can control all executable code of an application.

Figure 2 shows the execution process of DOOLDA's application launcher. First, DOOLDA loads DaNIDA when it starts. Since DaNIDA aims to instrument native code, it needs to be loaded first to gain control overall the various shared libraries which are loaded by an application later. After that, the Android runtime engine, ART [29], is loaded to the process. Then, DOOLDA load DaBIDA that works with the runtime. DOOLDA also loads the pre-compiled dex file, called dummy dex, at this time. The dummy dex contains code and data that are going to be replaced with logic for implementing anti-analysis techniques in an application. Lastly, the launcher loads the target application and starts an application.

4.3 DaBIDA

DaBIDA is a bytecode instrumentation module that operates by an event-driven mechanism. Precisely, DaBIDA catches an event, such as *MethodEntered*, which arises when ART interprets bytecode that starts executing a method, to inspect and instrument an application. Table 2 shows the events and interfaces of the event handlers that DaBIDA uses. DaBIDA traces an application's execution by using the handlers.

Instrumenting an instruction. To trace each instruction being executed, DaBIDA utilizes *ArtMethod* and *dex_pc* that the most of event handlers use except for *ExceptionCaught* and Branch. *ArtMethod* object manages the bytecode of a method and *dex_pc* refers to the offset value of an instruction to be executed. By combining these two data, DaBIDA can trace the bytecode of an application.

In addition, DaBIDA monitors data stored in virtual registers used by the bytecode that DaBIDA is tracing. The virtual registers are stored in a virtual stack called ShadowFrame. To monitor the virtual registers, DaBIDA uses the Thread object that exists in all of the instrumentation events. As presented in Figure 3, Thread object has a field that is pointing to the TopManagedStack and ManagedStack has a field pointing to the TopShadowFrame. Hence, DaBIDA accesses the ShadowFrame by using the Thread object that transferred from the instrumentation event and monitors the virtual register value and other data stored in the virtual stack. The data stored in virtual registers can be simple data such as an integer value or a memory address that points to an object such as String, ArtMethod. If the data is a pointer, DaBIDA finds an object that the pointer is pointing to and an

exact value of it because it can be an important value used in anti-analysis techniques.

DaBIDA repeats the monitoring process for every bytecode instruction of a target application. Through the monitoring process, DaBIDA identifies the instruction related to the anti-analysis techniques such as obtaining property information of a device. Also, the DaBIDA handles it according to the strategy represented in Section 4.1 by replacing it with the dummy method or dummy data to make the anti-analysis techniques fail to prevent analysis.

For the case of the instruction that gets the property information, we instrument the result of the property data after executing the instruction. Figure 4 shows an example that how we instrument such instructions with DaBIDA.

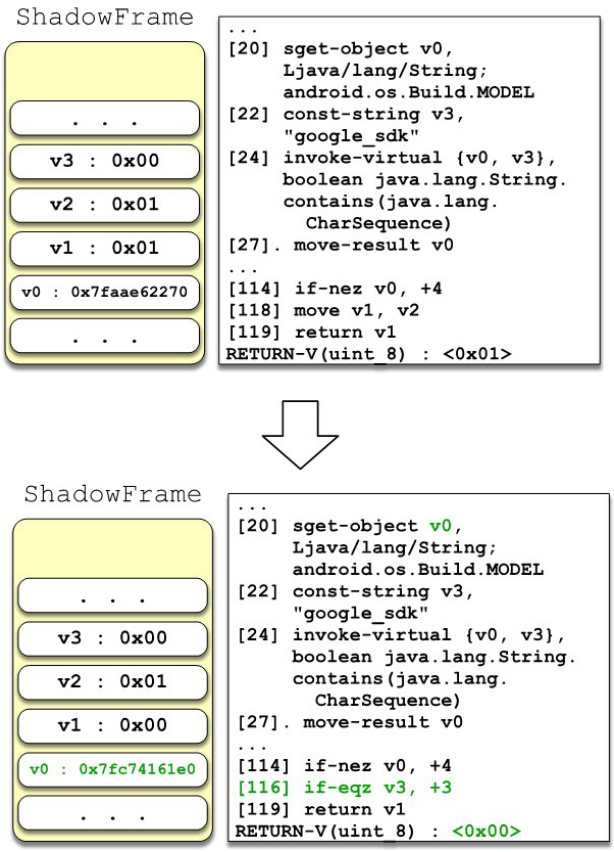

Figure 4. The result of instrumenting virtual registers and opcode using DaBIDA

Before the instrumenting, the result of code *sget-object v0, Ljava/lang/String; android.os.Build.MODEL* on Line 20 is 0x7faae62270 which is the address of a String object that stores *android.os.Build.MODEL* data. However, after the instrumentation, the result of the code on line 20 is 0x7fc74161e0 which is the address of a String object that stores our dummy data collected from the normal device. As the result, the return value of the method becomes 0x00 from 0x01.

Instrumenting an invocation of a method. There are two ways to invoke a method in the bytecode of an Android application. One is to use *dexcache* and another one is to use a dispatch table such as *vtable* or *itftable*. Both of them have similar mechanisms that find a new method using the index of the method represented in the bytecode's operand. DaBIDA handles both cases by using the mechanism that replaces the target method referenced by the operand of the bytecode with the dummy method.

When a method is invoked by using the *dexcache*, DaBIDA instruments the bytecode as presented in Figure 5. Before the instrumentation, the *method[1234]* stored in the *ResolvedMethods* array managed by *dexcache* points to the original *Method@1234* method. However, after the instrumenting, the *method[1234]* stored in the ResolvedMethods array points to the Dummy Method. As the result, when an application invokes the *method@1234* method, *Dummy Method* will be invoked instead of the original one. Likewise, when a dispatch table such as the *vtable* is used for invoking a method, DaBIDA instruments the table as illustrated in Figure 6. After DaBIDA instruments the table, the table entry that pointed to the *method[1234]* points to the Dummy Method.

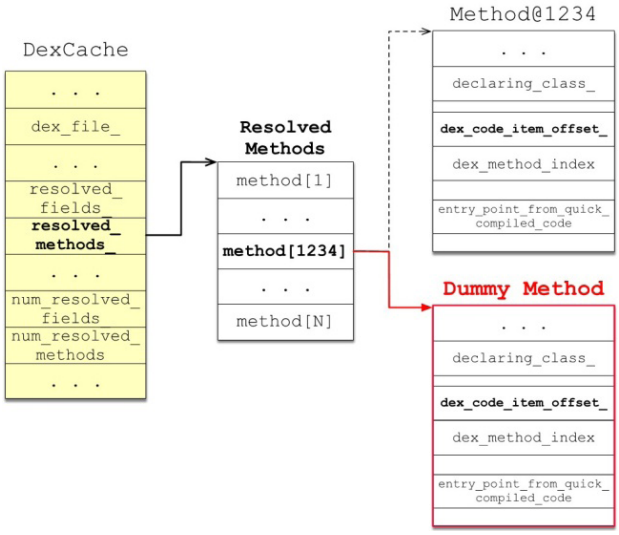

Figure 5. In the case of method invocation with dexcache, the instrumenting result by DaBIDA

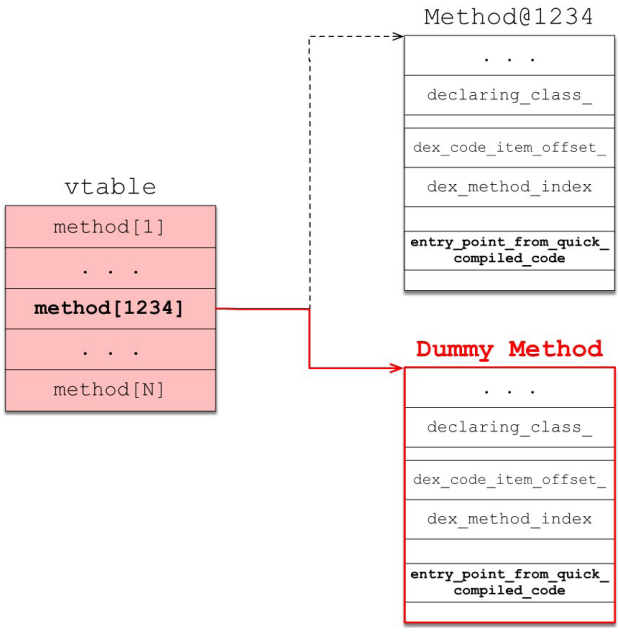

Figure 6. In the case of method invocation with vtable, the instrumenting result by DaBIDA

Figure 7 illustrate an example of how DaBIDA instruments the code for invalidating anti-emulation techniques that checks the *DeviceId* and *Build.MODEL* information. Through the monitoring process, DaBIDA can identify the code related to the anti-emulation techniques that using *android.telephony. TelephonyManager.getDeviceId* and *android.os.Build.MODEL*. The former method is invoked through the *vtable*. Therefore, DaBIDA replaces the element of *vtable* to invoke a dummy method that returns *996156449799883* string which is the device ID obtained from a normal device. Also, after the instruction that gets the *android.os.Build.MODEL* property, DaBIDA overwrites the String object to have Nexus 5 string. As a result, the proper data will be used and the anti-emulation techniques in the application will be invalidated.

```
boolean isEmulator(android.content.Context)
 reg6(obj) - android.app.Application
...
[10] invoke-virtual {v0}, java.lang.String
     android.telephony.TelephonyManager.getDeviceId()
[13] move-result-object v0
[14] const-string v3, "000000000000000"
[16] invoke-virtual {v0, v3}, boolean
     java.lang.String.equals(java.lang.Object)
[19] move-result v4
[20] sget-object v0, Ljava/lang/String; android.os.Build.MODEL
[22] const-string v3, "google_sdk"
[24] invoke-virtual {v0, v3}, boolean java.lang.String.contains
      (java.lang.CharSequence)
[27] move-result v0
...
RETURN-V(uint_8) : <0x01> // TRUE
```

```
boolean isEmulator(android.content.Context)
 reg6(obj) - android.app.Application
...
[10] invoke-virtual {v0}, java.lang.String
     android.telephony.TelephonyManager.getDeviceId()
 String com.dummy.Dummy_method.getDeviceId()
  [0] sget-object v0, com.dummy.Dummy_string.deviceId
  [1] return v0
 RETURN-V(String) : "996156449799883"
[13] move-result-object v0
[14] const-string v3, "000000000000000"
[16] invoke-virtual {v0, v3}, boolean
     java.lang.String.equals(java.lang.Object)
[19] move-result v4
[20] sget-object v0, Ljava/lang/String; android.os.Build.MODEL
      // overwrited to "Nexus 5"
[22] const-string v3, "google_sdk"
[24] invoke-virtual {v0, v3}, boolean java.lang.String.contains
      (java.lang.CharSequence)
[27] move-result v0
...
RETURN-V(uint_8) : <0x00> // FALSE
```

Figure 7. Bytecode instrumenting using DaBIDA

4.4 DaNIDA

In Android, the native code starts executing by a request from the bytecode through the JNI. Therefore, DaNIDA starts monitoring the execution of the native code when the JNI is called by an application. To implement DaNIDA, we used a dynamic binary instrumentation framework: Valgrind. DaNIDA monitors native codes and manages the execution flow of an application. To this end, DaNIDA translates the native code to VEX Intermediate Representation (IR). VEX is an architecture-agnostic, side-effects-free representation of a num ber of target machine languages.

Figure 8 shows the translation result of *Mov ebp, [esp + 16]* instruction to the IR. The IR uses an internal temporary variable on its own, and an IR statement is mapped to a computing operation. Accordingly, one machine instruction can be represented as multiple IR statements. The *IMark* statement stores the address and length of the instruction and the following IRs are as below: (1) Store a value of *ESP* register in *t9*; (2) Add 16 to *t9* and store in *t8*; (3) Store, in *t10*, a value pointed by the address stored in *t8*; (4) Store the value of *t10* in *EBP*; and (5) Insert an address of the instruction to be executed next into *EIP*.

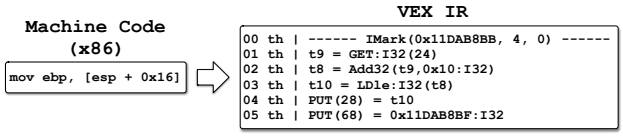

Figure 8. Example of translating machine code to Vex IR

VEX generates and executes one basic block and generates the next basic block according to the execution result of the previous basic block. Figure 9 shows the structure of the basic block used in VEX. One basic block has IR statements corresponding to the plurality of instructions, and each IR is distinguished by Tag. There are two ways for a basic block to find the next basic block. When an address intended to branch is a constant value, the corresponding value becomes the start address of the next basic block and when the address is not a constant value, the execution result of the corresponding basic block is stored in a temporal variable and the result value of the corresponding temporary variable becomes the start address of the next basic block.

After the translation of a basic block is finished, DaNIDA's monitoring module checks if the anti-analysis techniques are implemented in the basic block. To this end, we collect and implement known anti-debugging, -emulation, and -rooting techniques shown in Table 1. Looking at the various signatures collected, we found that most anti-analysis techniques use the form of human-readable strings in the native code. Therefore, in order to use those signatures, DaNIDA checks string constants and function names for detecting anti-analysis techniques.

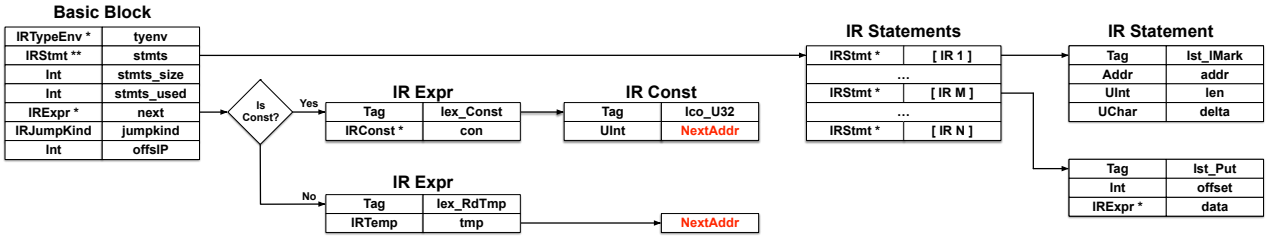

Figure 9. The basic block structure of VEX IR.

Normally, by monitoring IRs translated from the native code, DaNIDA cannot know what the method is or what string is used. There are two main reasons for this. First, there are only addresses or const values in IRs. Second, applications are published after removing symbols unnecessary to execute the native code. Therefore, it is difficult to obtain string information directly from the native code.

To overcome the challenge, we map the address of a basic block monitored by DaNIDA and library functions because the symbol table of a native library contains addresses and names of functions. By using */proc/<pid>/maps* file, we can check where the starting address of the basic block, and we can find out which native library has the current basic block. Similarly, if the library is identified, the offset of the method can be known through the *symbol table* of the library. Through the previously extracted information, it is possible to identify which method in which the library was called based on the starting address of the basic block during the execution. In the monitoring stage, when a function is called, an address value is converted into human-readable information that tells us which function is executing. DaNIDA compares the information with the signatures collected in advance to determine whether anti-analysis techniques are used by an application.

When DaNIDA detects anti-analysis techniques, it selects a invalidating function of the corresponding anti-analysis techniques. DaNIDA's IR instrumentation module changes basic blocks to bypass anti-analysis techniques by modifying execution flows. Figure 10 illustrates how DaNIDA instruments a function by using a code snippet for checking the *su* binary to figure out the device is rooted or not. When the corresponding method is executed, the same result produced as hooking of the corresponding function may be performed by deleting the existing IR, inserting 0 into *EAX* and modifying IR to run the termination instruction. In this way, DaNIDA can change the content of a function before it executes.

```
Function : boolean
com.example.sample.MainActivity.checkForSuBinary()
00 th | ------ IMark(0x11FBF00C, 7, 0) ------
01 th | t18 = GET:I32(24)
02 th | IR-NoOp
03 th | t2 = GET:I32(8)
04 th | IR-NoOp
05 th | IR-NoOp
06 th | ------ IMark(0x11FBF013, 3, 0) ------
07 th | t4 = Sub32(t18,0x2C:I32)
08 th | PUT(24) = t4
09 th | PUT(68) = 0x11FBF016:I32
...
51 th | ------ IMark(0x11FBF037, 2, 0) ------
52 th | t48 = CmpEQ32(t39,0x0:I32)
53 th | t47 = 1Uto32(t48)
54 th | t45 = t47
55 th | t49 = 32to1(t45)
56 th | t40 = t49
57 th | if (t40) { PUT(68) = 0x11FBF06C:I32; exit-Boring }
```

```
Function : boolean
com.example.sample.MainActivity.checkForSuBinary()
00 th | PUT(8) = 0x0:I32
01 th | t50 = 0x11D46055:I32
```

Figure 10. Method hooking with IR using DaNIDA

For another example, using the PTRACE function is a widely-used technique by anti-debugging techniques. If the target process is a process executed by the debugger, PTRACE returns -1 with an error and 0 on success. Anti-debugging techniques can use this feature to whether the process is debugged or not. However, with DaNIDA, if the *PTRACE* function is called, DaNIDA changes IR to return 0 instead of executing the original code of the *PTRACE* function so that detecting a debugger is not possible.

5 Evaluation

In this section, we evaluate DOOLDA with Android malware provided by the argus group [8]. Specifically, we address the following three research questions.

• RQ1. How much Android malware uses anti-analysis techniques?

• RQ2. Can DOOLDA automatically invalidate real-world malware equipped with anti-analysis techniques?

• RQ3. What does DOOLDA do better than the other dynamic analyzer?

5.1 Implementation

We implemented the prototype of DOOLDA with dual instrumentation modules, DaBIDA and DaNIDA. DaBIDA is implemented based on the instrumentation module in ART and DaNIDA is implemented based on the VEX module of Valgrind. DOOLDA's instrumentation modules are loaded into a process and trace the execution of an application, invalidating anti-analysis techniques. Because DOOLDA uses the same virtual memory space with a target application, it can manipulate the target application's data and code without any restriction.

5.2 Experimental Setup

Setup. Our evaluations were performed on the Android Virtual Device (AVD) and a mobile device (Nexus 5) with the Android system (version 5.1). Also, we used the device with remote gdb for detecting malware equipped with anti-debugging techniques.

Malware dataset. Since there was no verified dataset of malware equipped with anti-analysis techniques, we need to extract samples from a reliable malware dataset. We, thus, used Android malware provided by the argus group [8]. The dataset contains 24,650 android malwares (as APK files), and each of them is classified into each category. Among them, we first found malware samples that use anti-analysis techniques and were executable. First, by using the signatures in Table 1, we checked whether the signature exists each application. After that, we checked whether the app is executable and whether the execution result changes according to the execution environment. For example, there are samples that run when it execute on a mobile device, but not on the emulator. Consequently, we could select 1,535 samples in total.

5.3 Measuring Anti-Analysis Techniques

We performed experiments to find how many Android malware is using anti-analysis techniques with the dataset. In this experiment, we directly checked behaviors of malware in various execution environments (i.e., on a normal device, Android Virtual Device (AVD), and with the remote gdb). The classification result presents in Table 3. 16.87% of malware applies anti-emulation techniques, and 2.28% of malware applies anti-debugging techniques. The most commonly used technique is anti-emulation techniques and the malware that applies at least one type of anti-analysis technique accounts for 18.31% of the dataset.

Usually, if an anti-analysis technique is used by an application, it first executes the technique to check whether the application is running on an analyzer. Therefore, we checked whether the main activity of an application normally executes in various environments. We note that this experiment does not aim to observe malicious behaviors of malware, but to analyze whether the anti-analysis techniques were successfully bypassed by DoolDA, and which anti-analysis is used by malware.

Because we use the sampled data, two cases can occur when running a malicious application using DoolDA: Either the app runs because an anti-analysis technique was bypassed by DoolDA, or the app does not execute because DoolDA could not bypass an anti-analysis technique. Also, after launching each malware, we can check the log of DoolDA to see if the anti-analysis technique was successfully bypassed. When we met a case where an application does not run occurs because DoolDA fails to bypass the application's anti-analysis technique, we added a signature so that DoolDA can bypass the anti-analysis technique by manually analyzing the application. By repeatedly performing this process, we performed measurements on all malware samples, adding signatures. In addition, the samples that we chose from the dataset contain only malware that equips anti-analysis techniques. Consequently, it is worth noting that, in our measurement results, there are no false-positives.

In addition to anti-emulation and anti-debugging techniques, there is a well-known anti-analysis technique called anti-rooting. Commercial applications such as banking applications usually use the technique. As shown in Table 4, in our dataset, no malware uses anti-rooting techniques. However, there is 77 malware that checks the root permission. Listing 2 shows a code snippet of malware that requires the root permission (AR2 in Table 1). If there is a file in the path, the FALSE value will be returned, and the application will show a message that the application cannot run because it does not have the root privilege. Such malware checks the root permission and utilize it to do malicious actions.

Table 3. The applying rate of anti-analysis techniques for the dataset

Category	Anti-emulation tech	Anti-debugging tech	Not applied	**Sub-total**
AndroRAT	0 (0%)	0 (0%)	30 (100.00%)	**30**
Minimob	14 (21.53%)	6 (9.23%)	48 (73.84%)	**65**
BankBot	26 (6.87%)	0 (0%)	352 (93.12%)	**378**
FakeDoc	0 (0%)	13 (86.66%)	2 (13.33%)	**15**
Vidro	14 (77.77%)	11 (61.11%)	3 (16.66%)	**18**
Nandrobox	10 (31.25%)	0 (0%)	22 (68.75%)	**32**
Penetho	4 (30.76%)	0 (0%)	9 (69.23%)	**13**
DroidKungFu	1 (2.00%)	0 (0%)	49 (98.00%)	**50**
Utchi	0 (0%)	0 (0%)	10 (100.00%)	**10**
Svpeng	1 (25.00%)	0 (0%)	3 (75.00%)	**4**
Winge	0 (0%)	0 (0%)	5 (100.00%)	**5**
GingerMaster	1 (4.00%)	0 (0%)	24 (96.00%)	**25**
Mtk	7 (28.00%)	0 (0%)	18 (72.00%)	**25**
Lotoor	61 (66.30%)	5 (5.43%)	26 (28.27%)	**92**
Jisut	0 (0%)	0 (0%)	90 (100.00%)	**90**
SimpleLocker	0 (0%)	0 (0%)	56 (100.00%)	**56**
Opfake	2 (50.00%)	0 (0%)	2 (50.00%)	**4**
Triada	1 (3.44%)	0 (0%)	28 (96.55%)	**29**
Youmi	86 (22.63%)	0 (0%)	294 (77.37%)	**380**
Dowgin	31 (14.48%)	0 (0%)	183 (85.51%)	**214**
Total	**259 (16.87%)**	**35 (2.28%)**	**1,254 (81.69%)**	**1,535**

Table 4. The applying rate of anti-rooting techniques for the dataset

Category	Desiring root	Anti-rooting tech	Do not need root	**Sub-total**
AndroRAT	2 (6.66%)	0 (0%)	28 (93.33%)	**30**
Minimob	5 (7.69%)	0 (0%)	60 (92.31%)	**65**
BankBot	0 (0%)	0 (0%)	378 (100.00%)	**378**
FakeDoc	0 (0%)	0 (0%)	15 (100.00%)	**15**
Vidro	0 (0%)	0 (0%)	18 (100.00%)	**18**
Nandrobox	9 (28.12%)	0 (0%)	23 (71.88%)	**32**
Penetho	2 (15.38%)	0 (0%)	11 (84.62%)	**13**
DroidKungFu	15 (30.00%)	0 (0%)	35 (70.00%)	**50**
Utchi	0 (0%)	0 (0%)	10 (100.00%)	**10**
Svpeng	0 (0%)	0 (0%)	4 (100.00%)	**4**
Winge	0 (0%)	0 (0%)	5 (100.00%)	**5**
GingerMaster	8 (32.00%)	0 (0%)	17 (68.00%)	**25**
Mtk	0 (0%)	0 (0%)	25 (100.00%)	**25**
Lotoor	19 (20.65%)	0 (0%)	73 (79.35%)	**92**
Jisut	5 (5.55%)	0 (0%)	85 (94.44%)	**90**
SimpleLocker	2 (3.57%)	0 (0%)	54 (96.42%)	**56**
Opfake	0 (0%)	0 (0%)	4 (100.00%)	**4**
Triada	1 (3.44%)	0 (0%)	28 (96.55%)	**29**
Youmi	7 (1.84%)	0 (0%)	373 (98.16%)	**380**
Dowgin	2 (0.93%)	0 (0%)	212 (99.07%)	**214**
Total	**259 (16.87%)**	**35 (2.28%)**	**1,254 (81.69%)**	**1,535**

Listing 2. Case study - root detection: source code of malware that runs malicious script

```
1  protected void onCreate(Bundle arg10) {
2    ...
3    if(!hasRootPermission()) {
4      v1.setTitle(«Check root permission»);
5      v1.setMessage(«Sorry, you don\'t have root permission ...
«);
6    } else {
7      ...
8      new AppInitializer(((Context)this), ... ).start());
9    }
10 }
11
12 public static boolean hasRootPermission {
13   Boolean v2 = true;
14
15   if(!new File(«/system/bin/su»).exists()
16        && !new File(«/system/xbin/su»).exist()
17        && !new File(«/system/sbin/su»).exist()) {
18          v2 = false;
19   }
20   ...
21
22 public class AppInitializer extends Thread {
23   public void run() {
24     v1.exec(
25       «chmod 755
26        /data/data/com.aps.hainguyen273.app2card/.app2card_
tmp/getinfo.sh\n
27        /data/data/com.aps.hainguyen273.app2card/.app2card_
tmp/getinfo.sh info»
28     )
29   }
30 }
```

5.4 Analyzing the Malware Equipped with Anti-Analysis Techniques using DOOLDA

DOOLDA discovered and invalidated all anti-analysis techniques successfully in 281 malware as in Table 3. In this section, we show how DOOLDA analyzes the malware equipped with anti-analysis techniques through the following case studies. The case studies consist of malware implementing anti-emulation, anti-debugging techniques. DOOLDA defeats each case of the anti-analysis techniques by instrumenting the target malware successfully.

5.4.1 Case Study: Anti-Emulation Techniques

Listing 3 shows the code of malware using anti-emulation techniques. In the code, the method *setAlarm* of class n enables the alarm receiver to steal the contents of SMS and MMS. Therefore, *setAlarm* is the main target to analyze and we have to execute the code to analyze it dynamically. It is invoked by *onCreate* method and *onCreate* method will be invoked when the *ActivityStart* activity is initialized. However, the method *setAlarm* is not always invoked. *setAlarm* method will be invoked according to the result of method *isEmulator* of the class n. It will be invoked according to the result of the method *isEmulator* of the class n. The method *isEmulator* has the code related to the anti-analysis techniques especially blocking the analyzer with a virtual environment. In this case, there are techniques corresponding to AE1 and AE2 in Table 1. It checks the information such as *DeviceId*, *Build.MODEL, Build.DEVICE*, etc. If at least one of the information implies that the application is running on an emulator, the method *isEmulator* will return TRUE and the method *setAlarm* will not be invoked.

Listing 3. Case study of the anti-emulation: malware steals the contents of the message in java code

```
1  public class ActivityStart extends Activity {
2    ...
3    protected void onCreate(Bundle arg5) {
4      super.onCreate(arg5);
5      ...
6      if(!n.a(this.getApplicationContext())) {
7        n.b(this.getAppcationContext());
8        ...
9      }
10   }
11
12   public class n {
13     ...
14     public static boolean isEmulator (...) {
15       boolean v1 = false;
16       boolean v4 = arg6.getSystemService(«phone”)
.getDeviceId()
17                                .equals(«000000000000000»);
18       int v0 = (Build.MODEL.contains(«google_sdk»)) || ... ?
1 : 0;
19       int v3 = !Build.DEVICE.startsWith(«generic») || ... ? 0 :
1;
20
21       if ((v4) || (v0 != 0) || (v3 != 0)) {
22         v1 = true;
23       }
24       return v1;
25     }
26
27     public static void setAlarm (Context arg7) {
28       if (Build$VERSION.SDK_INT >= 19) {
29         arg7.getSystemService(«alarm») ...
30         new Intent(arg7, AlarmReceiverKnock.class), ... );
31       } else {
32         arg7.startService(new Intent(arg7, knock.class));
32       }
34     }
35   }
36 }
```

For a successful analysis, we have to make the result of method *isEmulator* to be FALSE. In this case, DOOLDA invalidated the anti-emulation techniques by instrumenting the bytecode of the *isEmulator* method with DaBIDA to make the method returns FALSE. As a result, with DOOLDA, we can successfully found the malicious behavior as in Figure 11 that llustrates the malware leaks SMS information after an SMS message is received.

For a successful analysis, we have to make the result of method *isEmulator* to be FALSE. In this case, DOOLDA invalidated the anti-emulation techniques by instrumenting the bytecode of the *isEmulator* method with DaBIDA to make the method returns FALSE. As a result, with DOOLDA, we successfully found the malicious behavior as in Figure 11 that illustrates the malware leaks SMS information after an SMS message is received.

```
boolean isEmulator(android.content.Context)
 reg6(obj) - android.app.Application
...
[10] invoke-virtual {v0}, java.lang.String
     android.telephony.TelephonyManager.getDeviceId()
 String com.dummy.Dummy_method.getDeviceId()
  [0] sget-object v0, com.dummy.Dummy_string.deviceId
  [1] return v0
 RETURN-V(String) : "996156449799883"
[13] move-result-object v0
[14] const-string v3, "000000000000000"
[16] invoke-virtual {v0, v3}, boolean
     java.lang.String.equals(java.lang.Object)
[19] move-result v4
[20] sget-object v0, Ljava/lang/String; android.os.Build.MODEL
       // overwrited to "Nexus 5"
[22] const-string v3, "google_sdk"
[24] invoke-virtual {v0, v3}, boolean java.lang.String.contains
      (java.lang.CharSequence)
[27] move-result v0
...
RETURN-V(uint_8) : <0x00> // FALSE

...
void m.a(android.content.Context, java.lang.String)
 reg3(obj) - android.app.Application
 reg4(string) - {"module":"sms","tel":"01002102410",
                 "body":"Your Google verification code is
                  G-554053", "time":"Wed Jul 08 22:58:19 EDT}
...
java.lang.String r.a(java.lang.String, org.json.JSONObject)
 reg5(string) - http://childgura.in/beloado/index.php
 reg6(obj) - org.json.JSONObject
 ...
```

Figure 11. The analysis result of DOOLDA with code instrumenting of DaBIDA

5.4.2 Case Study: Anti-Debugging Techniques

Listing 4 presents the code of malware equipped with anti-debugging techniques. In the code, the method *rootShell* executes shellcode and installs a new application for conducting malicious behaviors. Therefore, *rootShell* is the main target to analyze and we have to execute the code to analyze it dynamically. It is invoked by *onClick* method and *onClick* method will be invoked after the *MainActivity* is created. In the control flow of this application, there is no restriction of invoking *rootShell* method. However, it has a pre-configuration step in the native library. When the *MainActivity* is created, the *DBGChecker* instance is also created. *DBGChecker* is a class defined in a native library and it has the function named *wwx*. The *wwx* function preempt the *PTRACE* system call for denying the analysis with *PTRACE* as in AD2 of Table 1.

DOOLDA can analyze the malware by invalidating the anti-debugging techniques with DaNIDA. If the analysis target program uses the *PTRACE* system call shown in Listing 4, it can be easily detected and invalidated through DaNIDA. First, DaNIDA knows the names and locations of other libraries with the process map. With this information, before the library code is executed, by using the offset of the functions included in the library based on the execution address, it is possible to know which function is scheduled to be executed next. Through this, DaNIDA can notice that the *PTRACE* system call is the next function to be executed.

Then, DaNIDA modifies IR statements in the *PTRACE* system call to return right after the system call executes.

The instrumentation and analysis results are shown in Figure 12. After DaNIDA modified the original *PTRACE* system call's IR statements, the malware returns from the *PTRACE* system call, and then, installs the *systemservice.apk*.

Listing 4. Case study of the anti-debugging: malware installs another malicious application

```
1  // sec.cpp
2  void wwx() {
3    if(ptrace(PTRACE_ATTACH, p_pid, NULL, NULL) == 0)
   {
4      ...
5      ptrace(PTRACE_CONT, p_pid, NULL, NULL);
6    }
7  }
8
9  jint JNI_OnLoad(JavaVM* vm, void* reserver) {
10   …
11   wwx();
12 }
13
14 // MainActivity.java
15 protected void onCreate(Bundle savedInstanceState) {
16   ...
17   DBGChecker v11 = new DBGChecker();
18 }
19
20 public void onClick(DialogInterface arg10, int arg11) {
21   InputStream v2 = this.getAssets().open("assets/libx86.so");
22   FileUtils.copyInputStreamToFile(v2, new File("/sdcard/
     fc.key"));
23   ...
24   new shell_m().rootShell();
25 }
26
27 public void rootShell() {
28   sh.execCommand(new String[]{"mount -o rw,remount /
     system", ...,
29     "mv /sdcard/fc.key/system/app/systemservice.apk",
30     "chmod 644 /system/app/systemservice.apk", "reboot"},
       true);
31   …
32 }
```

```
void com.smsbombardment.MainActivity$100000001
       .onCreate(Bundle savedInstanceState)
 ...
 [17] invoke-direct {v0}, void
com.smsbombardment.scurity.DBGChecker.<init>()

================================================================
Object : /data/app/xxx.zzzz.cccc/lib/x86/libSec.so
0 th | ------ IMark(0x120F8864, 2, 0) ------
...
42 th | ------ IMark(0x120F86C0, 6, 0) ------  // jump to ptrace()
43 th | t30 = Add32(t21,0x20:I32)
44 th | t11 = LDle:I32(t30)

object : /system/lib/libc.so
0 th | ------ IMark(0x4A246C0, 1, 0) ------  // 0x4A246C0 == PTRACE
1 th | PUT(8) = 0x0:I32
2 th | t42 = 0x11D46055:I32 // ret
================================================================
...
void com.smsbombardment.MainActivity$100000001
       .onClick(android.conetent.DialogInterface, int)
 ...
 void com.smsbombardment.shell_m.rootShell()
  reg9(obj) - com.smsbombardment.shell_m
 ...
 [9] const-string v7, "mount -o rw,remount /system"
 [11] aput-object v7, v5, v6
 . . .
 [16] const/4 v6, #+1
 . . .
 [65] const-string v7, "chmod 644 /system/app/systemservice.apk"
 [67] aput-object v7, v5, v6
 ...
 [71] const/4 v5, #+7
 [72] invoke-static {v4, v5},
        com.smsbombardment.shell_m$CommandResult
        com.smsbomardment.shell_m.execCommand(java.lang.String[],
                                              boolean)
```

Figure 12. The analysis result of DOOLDA with code instrumenting of DaNIDA

5.4.3 Case Study: Comparison with Other Dynamic Analyzers

In order to show differences from other dynamic analyzers, we compare analysis results of DOOLDA with other analyzers by using applications to which anti-emulation and anti-debugging techniques are applied. The comparison results are summarized in Table 5.

As presented in Figure 13 to Figure 16, DOOLDA can analyze all the cases successfully. However, DexMonitor [6], DroidScope [11] and Frida [14] failed to analyze some cases of the anti-emulation and anti-debugging techniques.

Table 5. The analyzability of dynamic analyzers including DOOLDA

		DexMonitor	DroidScope	Frida	DoolDA
Anti-emulation techniques	Checking device properties	✓	✗	✓	✓
	Checking system properties	✗	✗	✓	✓
	Checking signature files	✓	✓	✓	✓
Anti-debugging techniques	Checking debugging flag	✓	✓	✓	✓
	Checking process list	✓	✓	✗	✓
	Ports scanning	✓	✗	✗	✓
	System calls	✓	✓	✗	✓

6 Limitations

DOOLDA is a novel system that has dual instrumentation modules to cover the whole code of the Android applications and can handle the malware equipped with anti-analysis techniques. In this paper, we presented the prototype of DOOLDA and proved its superiority in Section 5. However, as in other analyzers, some limitations still exist in DOOLDA and we leave them as future work.

DOOLDA can invalidate only known anti-analysis techniques. For invalidating anti-analysis techniques, DOOLDA has to detect the code related to anti-analysis techniques first. However, the detection is based on the information that was previously reported. Therefore, DOOLDA cannot handle unknown anti-analysis techniques. In addition, DaNIDA cannot support the latest version of the Android system. Because DaNIDA is implemented based on the VEX module of Valgrind and the VEX module supports up to Android system version 5.1. To overcome this limitation, we can change the base module of DaNIDA from Valgrind to Address Sanitizer (ASAN) [30].

7 Related Work

In this section, we discuss previously proposed systems for dynamically analyzing Android malware.

There are dynamic analyzers implemented by modifying the Android operating system [6-7, 12, 25, 31]. Such in-the-box analyzers can directly monitor executions of malware without a semantic gap. Also, as far as they run on bare-metal, they can minimize the risk of being detected by analyzed applications. However, they have a couple of limitations. First off, in-the-box analyzers have Android system version dependency as they run in a specific version of Android system. For example, if the system is updated, there is a hassle of analyzing the updated Android system code and re-creating an analysis tool based on it. In addition, execution environments for using such analyzers can be limited to actual devices. This is because, if they run on an emulator, anti-emulating techniques can hinder analysis.

Dynamic analysis tools based on the emulator have the advantage of being able to configure environmental conditions [9-11]. Also, another advantage is that they can even analyze the most privileged attacks because they analyze malware in a controllable sandbox. However, their downsides are that (1) they should rebuild the semantic information; and (2) they can be thwarted by anti-emulation techniques.

Dynamic code instrumentation approaches do not have environmental limitations related to executions of applications as in the in-the-box and emulator-based analyzers [14, 32-33]. However, as far as they use a debugging bridge or system calls for debugging such as *PTRACE*, those analyzers cannot analyze applications implementing anti-debugging techniques without bypassing them.

8 Conclusion

In this paper, we propose DOOLDA, a novel system using dual instrumentation modules for handling both the native code and bytecode, to analyze the malware equipped with anti-analysis techniques. Through our evaluations, we showed that DOOLDA can invalidate anti-analysis techniques by automatically instrumenting code that is about to be executed. We believe that DOOLDA can be used as an effective dynamic malware analysis framework for analyzing advanced malware.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT), and Future Planning under Grant NRF-2020R1A2C2014336 and Grant NRF-2021R1A4A1029650.

References

[1] B. Bichsel, V. Raychev, P. Tsankov, M. Vechev, Statistical Deobfuscation of Android Applications, In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, Vienna Austria, 2016, pp. 343–355.

[2] A. Dinaburg, P. Royal, M. I. Sharif, W. Lee, Ether: malware analysis via hardware virtualization extensions, *Proceedings of the 15th ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, 2008, pp. 51-62.

[3] A. P. Felt, E. Chin, S. Hanna, D. Song, D. A. Wagner, Android permissions demystified, *Proceedings of the 18th ACM Conference on Computer and Communications Security*, Chicago, Illinois, USA, 2011, pp. 627–638.

[4] S. Rasthofer, S. Arzt, M. Miltenberger, E. Bodden, Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques, In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, 2016, pp. 1–15.

[5] P. Royal, M. Halpin, D. Dagon, R. Edmonds, W. Lee, PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware, *Proceedings of the 22nd Annual Computer Security Applications Conference*, Miami, FL, USA, 2006, pp. 289–300.

[6] H. Cho, J. H. Yi, G.-J. Ahn, DexMonitor: Dynamically Analyzing and Monitoring Obfuscated Android Applications, *IEEE Access*, Vol. 6, pp. 71229–71240, November, 2018.

[7] L. Xue, H. Zhou, X. Luo, Y. Zhou, Y. Shi, G. Gu, F. Zhang, M. H. Au, Happer: Unpacking Android Apps via a Hardware-Assisted Approach, *2021 IEEE Symposium*

on Security and Privacy, San Francisco, CA, USA, 2021, pp. 1641–1658.

[8] Y. Li, J. Jang, X. Hu, X. Ou, Android Malware Clustering Through Malicious Payload Mining, *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses*, Atlanta, GA, USA, 2017, pp. 192–214.

[9] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, X. Wang, Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation, *Proceedings of the 2018 Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, 2018, pp. 1–15.

[10] K. Tam, S. J. Khan, A. Fattori, L. Cavallaro, CopperDroid: Automatic Reconstruction of Android Malware Behaviors, *Proceedings of the 2015 Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, 2015, pp. 1–15.

[11] L. K. Yan, H. Yin, DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis, *Proceedings of the 21st USENIX Security Symposium*, Bellevue, WA, USA, 2012, pp. 569–584.

[12] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, D. Gu, AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware, *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses*, Kyoto, Japan, 2015, pp. 359–381.

[13] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, R. B. Lee, Catalyst: Defeating last-level cache side channel attacks in cloud computing, *Proceedings of the 22nd IEEE Symposium on High Performance Computer Architecture*, Barcelona, Spain, 2016, pp. 406–418.

[14] O. A. V. Ravnås, *Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers*, 2021, https://frida.re/.

[15] M. K. Alzaylaee, S. Y. Yerima, S. Sezer, EMULATOR vs REAL PHONE: Android Malware Detection Using Machine Learning, *Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics*, Scottsadale, AZ, USA, 2017, pp. 65–72.

[16] S. Berlato, M. Ceccato, A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps, *Journal of Information Security and Applications*, Vol. 52, Article No. 102463, June, 2020.

[17] B. Brenner, *Android malware anti-emulation techniques*, 2017, https://news.sophos.com/en-us/2017/04/13/android-malware-anti-emulation-techniques/.

[18] A. Druffel, K. Heid, DaVinci: Android App Analysis Beyond Frida via Dynamic System Call Instrumentation, *Proceedings of the 2020 International Conference on Applied Cryptography and Network Security*, Rome, Italy, 2020, pp. 473–489.

[19] D. Jang, Y. Jeong, S. Lee, M. Park, K. Kwak, D. Kim, B. B. Kang, Rethinking anti-emulation techniques for large-scale software deployment, *Computers & Security*, Vol. 83, pp. 182–200, June, 2019.

[20] K. Lim, Y. Jeong, S.-J. Cho, M. Park, S. Han, An Android Application Protection Scheme against Dynamic Reverse Engineering Attacks, *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, Vol. 7, No. 3, pp. 40–52, September, 2016.

[21] D. Maier, T. Müller, M. Protsenko, Divide-and-Conquer: Why Android Malware Cannot Be Stopped, *Proceedings of the 9th International Conference on Availability, Reliability and Security*, Fribourg, Switzerland, 2014, pp. 30–39.

[22] N. Miramirkhani, M. P. Appini, N. Nikiforakis, M. Polychronakis, Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts, *2017 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, 2017, pp. 1009–1024.

[23] V. Sihag, M. Vardhan, P. Singh, A survey of android application and malware hardening, *Computer Science Review*, Vol. 39, Article No. 100365, February, 2021.

[24] S.-T. Sun, A. Cuadros, K. Beznosov, Android rooting: Methods, detection, and evasion, *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, Denver, CO, USA, 2015, pp. 3–14.

[25] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. D. McDaniel, A. Sheth, TaintDroid: An Information Flow Tracking System for Real-time Privacy Monitoring on Smartphones, *ACM Transactions on Computer Systems*, Vol. 57, No. 3, pp. 99–106, March, 2014.

[26] X. Wang, S. Zhu, D. Zhou, Y. Yang, Droid-AntiRM: Taming Control Flow Anti-Analysis to Support Automated Dynamic Analysis of Android Malware, Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC), Orlando, FL, USA, 2017, pp. 350–361.

[27] L. Xue, Y. Zhou, T. Chen, X. Luo, G. Gu, Malton: Towards On-Device Non-Invasive mobile malware analysis for ART, *Proceedings of the 26th USENIX Security Symposium*, Vancouver, BC, 2017, pp. 289–306.

[28] Google git, *Main entry of app_process*, 2021, https://android.googlesource.com/platform/frameworks/base/+/0419ab9e4a7761e2aac7e1bec73057b3beba97ec/cmds/app_process/app_main.cpp.

[29] Android Developer, *Android Runtime (ART) and Dalvik*, 2021, https://source.android.com/devices/tech/dalvik.

[30] Android Developer, *Address Sanitizer*, 2021, https://developer.android.com/ndk/guides/asan.

[31] M. Sun, T. Wei, J. C. S. Lui, TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime, *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, Vienna, Austria, 2016, pp. 331–342.

[32] V. Costamagna, C. Zheng, ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime, *Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security*, London, UK, 2016, pp. 20-28.

[33] L. Xue, X. Luo, L. Yu, S. Wang, D. Wu, Adaptive unpacking of Android apps, *Proceedings of the 39th*

International Conference on Software Engineering, Buenos Aires, Argentina, 2017, pp. 358–369.

Biographies

Sunjun Lee received the B.S and M.S degrees in Computer Science and Engineering from Soongsil University in 2019 and 2021, respectively. His research interests include binary analysis, reverse engineering, system security and mobile security.

Yonggu Shin received the B.S and M.S degrees in Computer Science and Engineering from Soongsil University in 2019 and 2021, respectively.

Minseong Choi received the B.S and M.S degrees in Computer Science and Engineering from Soongsil University in 2019 and 2023, respectively.

Haehyun Cho is an Assistant Professor in the School of Software at Soongsil University, Seoul, Korea. He received the B.S. and M.S. degrees in computer science from Soongsil University and the Ph.D. degree in computer science at School of Computing, Informatics and Decision Systems Engineering of Arizona State University.

Jeong Hyun Yi is a Professor in the School of Software at Soongsil University, Seoul, Korea. He received the B.S. and M.S. degrees in computer science from Soongsil University, in 1993 and 1995, and the Ph.D. degree in information and computer science from the University of California, Irvine, in 2005.

Appendix

```
boolean isEmulator(android.content.Context)
 reg6(obj) - android.app.Application
...
[10] invoke-virtual {v0}, java.lang.String android.telephony.TelephonyManager.getDeviceId()
 String com.dummy.Dummy_method.getDeviceId()
  [0] sget-object v0, com.dummy.Dummy_string.deviceId
  [1] return v0
 RETURN-V(String) : "996156449799883"
[13] move-result-object v0
[14] const-string v3, "000000000000000"
[16] invoke-virtual {v0, v3}, boolean java.lang.String.equals(java.lang.Object)
[19] move-result v4
[20] sget-object v0, Ljava/lang/String; android.os.Build.MODEL
       // overwrited to "Nexus 5"
[22] const-string v3, "google_sdk"
[24] invoke-virtual {v0, v3}, boolean java.lang.String.contains(java.lang.CharSequence)
[27] move-result v0
...
RETURN-V(uint_8) : <0x00> // FALSE

...
void m.a(android.content.Context, java.lang.String)
 reg3(obj) - android.app.Application
 reg4(string) - {"module":"sms","tel":"01002102410", "body":"Your Google verification code is
                 G-554053", "time":"Wed Jul 08 22:58:19 EDT}
...
java.lang.String r.a(java.lang.String, org.json.JSONObject)
 reg5(string) - http://childgura.in/beloado/index.php
 reg6(obj) - org.json.JSONObject
 ...
```

(a) The Analysis result of DOOLDA

```
|1|[fp=0x6d445c84] Ln;->isEmulator(Landroid/content/Context;)Z
|1|retval=0x414dff6800000000
|1|return to Lzzzzzz/xxxxxx/cccccc/ActivityStart;->onCreate(Landroid/os/Bundle;)V [fp=0x6d445cb4]
|1|BYTE_CODE0a00
|1|move-result v0          (v0=0x00000001)
|1|BYTE_CODE39001700
|1|if-nez v0, #0x0c
|1|BYTE_CODE0e00
|1|return-void
|1|retval=0x713c651800000000
|1|return to Landroid/app/Activity;->performCreate(Landroid/os/Bundle;)V [fp=0x6d445ce0]
```

(b) The analysis result of DexMonitor

Figure 13. The analysis result of the malware with anti-emulation of DOOLDA and DexMonitor.

```
[c0050f28] 7100 7d03 0000  |c0058c44: invoke-static{}, n.isEmulator // method@03d7
. . .
[c0051020] 0fc0            |c0058a84: return v192(0xbfeb0f88)
[c0051d28] 0a00            |c0051d28: move-result v192(0xbfeb0e48)
[c0051d2c] 3900 1700       |c0051d2c: if-nez v192(0xbfeb0e48), c0051dfc // +0034
[c0051dfc] 0e00            |c0051d34: return-void
[c002b020] 0fc0            |c0058a84: return v192(0xbfeb0c2c)
. . .
```

(a) The analysis result of DroidScope

```
*** entered zzzzzz.xxxxxx.cccccc.n.isEmulator()
 arg[0]: android.app.Application@c0a8085
 . . .
*** exiting zzzzzz.xxxxxx.cccccc.n.isEmulator()
 retval: false
 . . .
*** entered zzzzzz.xxxxxx.cccccc.n.setAlarm()
 arg[0]: android.app.Application@c0a8085
 . . .
*** entered zzzzzz.xxxxxx.cccccc.m.a()
 arg[0]: android.app.Application@c0a8175
 arg[1]: "module":"sms","tel":"01002102410", "body":"Your Google verification code is G-714003"
         ,"time":"Tue SEPTEMBER 14 20:58:19 EDT"}
 . . .
*** entered zzzzzz.xxxxxx.cccccc.r.a
 arg[0]: http://childgura.in/beloado/index.php
 arg[1]: org.json.JSONObject
 . . .
```

(b) The analysis result of Frida

Figure 14. The analysis result of the malware with anti-emulation of DroidScope and Frida.

```
void com.smsbombardment.MainActivity$100000001.onCreate(Bundle savedInstanceState)
 ...
 [17] invoke-direct {v0}, void com.smsbombardment.scurity.DBGChecker.<init>()

================================================================================================
Object : /data/app/xxx.zzzz.cccc/lib/x86/libSec.so
0 th | ------ IMark(0x120F8864, 2, 0) ------
...
42 th | ------ IMark(0x120F86C0, 6, 0) ------  // jump to ptrace()
43 th | t30 = Add32(t21,0x20:I32)
44 th | t11 = LDle:I32(t30)

object : /system/lib/libc.so
0 th | ------ IMark(0x4A246C0, 1, 0) ------  // 0x4A246C0 == PTRACE
1 th | PUT(8) = 0x1:I32
2 th | t42 = 0x11D46055:I32 // ret
================================================================================================
...
void com.smsbombardment.MainActivity$100000001.onClick(android.conetent.DialogInterface, int)
 ...
 void com.smsbombardment.shell_m.rootShell()
  reg9(obj) - com.smsbombardment.shell_m
 ...
 [9] const-string v7, "mount -o rw,remount /system"
 [11] aput-object v7, v5, v6
 . . .
 [16] const/4 v6, #+1
 . . .
 [65] const-string v7, "chmod 644 /system/app/systemservice.apk"
 [67] aput-object v7, v5, v6
 ...
 [71] const/4 v5, #+7
 [72] invoke-static {v4, v5}, com.smsbombardment.shell_m$CommandResult
        com.smsbomardment.shell_m.execCommand(java.lang.String[], boolean)
```

(a) The analysis result of DOOLDA

```
|1|[fp=0x6d445c70] Lcom/smsbombardment/security/DBGChecker;-><init>(Lcom/smsbombardment/
                                                                  MainActivity;)V
. . .
|1|BYTE_CODE1a07072e
|1|const-string v7 string@0x2e07, string = mount -o rw, remount /system
. . .
|1|BYTE_CODE1a0707ea
|1|const-string v7 string@0xea07, string = chmod 644 /system/app/systemservice.apk
. . .
|1|BYTE_CODE712005985400
|1|invoke-static args=2 @0x9805 {v4, v5, v0, v0, v0}
|1|[fp=0x6d445c5c] Lcom.smsbomardment.shell_m;->execCommand(Ljava.lang.StringBuffer;I)V
```

(b) The analysis result of DexMonitor

Figure 15. The analysis result of the malware with anti-debugging of DOOLDA and DexMonitor.

```
[c006541c] 7000 0701 0000  |c0058c44: invoke-direct{},
                                      com.smsbombardment.security.DBGChecker.<init>
                                      // method@0107
. . .
[c00654b8] 1a07 072e       |c0058a28: const-string v7(0xbfeb0 f88), "mount -o re, remount /
                                                                     system" // string@2e07
. . .
[c00654dc] 1a07 07ea       |c0058a3c: const-string v7(0xbfeb0f88), "chmod 644 /system/app/
                                                                    systemservice.apk"
                                                                    // method@ea07
. . .
[c0065504] 7120 0598 5400  |c0058a84: invoke-static {v4(0xbfeb0f88), v5(0xbfeb0f88)},
                                       com.smsbomarment.shell_m.execCommand // method@9805
. . .
```

(a) The analysis result of DroidScope

```
(py3) Userui-iMac:~ user$ frida -U com.aora.anti_debugger_sample
     ____
    / _  |   Frida 12.11.12 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://www.frida.re/docs/home/

Failed to attach: unable to access process with pid 12868 due to system restrictions;
try `sudo sysctl kernel.yama.ptrace_scope=0`, or run Frida as root
```

(b) The analysis result of Frida

Figure 16. The analysis result of the malware with anti-debugging of DroidScope and Frida.