공학석사학위논문

Breaking Ad-hoc Runtime Integrity Protection Mechanisms in Android Financial Apps

안드로이드 금융 어플리케이션들의 임기응변적인 실행환경 무결성 검사 분석

2017 년 2 월

서울대학교 대학원

컴퓨터공학부

김 태 훈

Abstract

Breaking Ad-hoc Runtime Integrity Protection Mechanisms in Android Financial Apps

Taehun Kim
Department of Computer Science and Engineering
The Graduate School
Seoul National University

To protect customers' sensitive information, many mobile financial applications include steps to probe the runtime environment and abort their execution if the environment is deemed to have been tampered with. This paper investigates the security of such self-defense mechanisms used in 76 popular financial Android apps in Republic of Korea. Our investigations found that existing tools fail to analyze these Android apps effectively because of their highly obfuscated code and complex, non-traditional control flows. We overcome this challenge by extracting a call graph with a self-defense mechanism, from a detailed runtime trace record of a target app's execution to generate. To generate the call graph, we use the causality between the Android APIs and system calls used for integrity checks and for alert dialogs, or to kill the app itself. Our analysis of 76 apps shows that once we obtain a causality graph, we can pinpoint methods to bypass most self-defense mechanisms. We successfully bypassed 67 out of 73 apps that check the platform integrity and 39 out of 44 apps that check the binary integrity of the host app, which shows the inefficiency of checking the integrity at the app level. We also present in-depth studies of the top five security libraries used in the aforementioned

apps to provide their self-defense mechanisms and their weaknesses. Because financial mobile applications should not run during tampered runtimes, our results clearly demonstrate the necessity of a platform-level solution for integrity checks.

Keywords: Application Security, Android, Reverse Engineering

Student Number: 2015-21236

Contents

List of Tables

List of Figures

List of Code

Chapter 1

Introduction

Mobile financial applications (hereafter referred to as *apps*) are gaining popularity, such as banking apps, retailer apps (e.g., CurrentC [1]), credit card apps, and payment modules embedded in messaging apps. In the United States, smartphone users who use mobile banking and mobile payments has increased annually, reaching 53% and 28% in 2015, respectively [2].

To protect users of financial services, mobile financial apps use an extra layer of security, such as two-factor authentication [3], one-time passwords [4], or secure communication via secure socket layer/transport layer security (SSL/TLS) [5–7]. Although these security measures are beneficial to a certain degree, if an app's platform is compromised, any application-level measure can be bypassed, thereby becoming ineffective against various known attacks, including root-exploit attacks [8], app-repackaging attacks [9], keyboard-hooking attacks [10, 11], and memory-dumping attacks [12]. Therefore, apps must check the integrity of the platform to make other security measures effective.

In principle, no bullet-proof solutions exists that allow Android apps to determine whether the device on which they are running has been rooted or whether the binary of the app itself has been modified. Nonetheless, we observe that many Android financial apps appear to employ some mechanisms that check for evidence of tampering and then abort an execution with a warning message if it detects something suspicious. Throughout the paper, we refer to these mechanisms as *self-defense mechanisms*. Despite their wide use, little is

known about how these mechanisms are designed and whether they are effective in practice.

This paper examines the effectiveness of the self-defense mechanisms used in Android financial apps. We specifically focus on the following research questions:

- What information do apps collect (and how do they collect it) to determine whether a device has been rooted or if the app's binary has been tampered with?
- Once self-defense mechanisms are identified, what are the steps needed to bypass them in order to continue executing the app as if the integrity check had passed?

Analyzing Android *financial* apps is challenging for several reasons. First, we observed that these apps often heavily use code obfuscation, making it difficult to gain an understanding of actual control flows through static analysis. For example, obfuscation tools change the names of an app's methods and classes to meaningless ones to conceal their roles [13]. Second, the control flows of Android apps tend to be complex, involving native code, and some are not connected directly. In Android, an app's components (e.g., Activity, Service, BroadcastReceiver, ContentProvider) or threads can communicate with each other using a message object. An Android app's control flow cannot be captured entirely without those indirect relationships. However, they are not connected with a traditional caller–callee relationship, leading many existing dynamic analysis tools to fail to detect them.

To overcome these difficulties, we collected the Java method invocations and the system calls brought up by native methods in the runtime and constructed a call graph based on the collected data. To get the call graph, we first recorded a detailed runtime trace of a target app's execution, including the indirect relationships between threads and components. We then identified the Android APIs and the system calls that the self-defense mechanisms must use and found the causality between the *environment checking*, which checks device rooting and app tampering, and the *execution termination*, which blocks an app's execution. Even if the app's code is obfuscated, they cannot hide which APIs or system calls they bring up in runtime because they are defined within the system. By finding the causal connection, we can

pinpoint the self-defense mechanisms among the thousands of methods in the target app.

We implemented the steps described above in MEthod Recorder with Connecting Indirect relations (MERCIDroid), the enhanced Android platform that tracks an app's control flow. The tool identified 92.9% of the self-defense mechanisms in the studied apps and narrowed the scope of the methods we investigated in the execution path to 3.7% on average.

Using MERCIDroid, we analyzed Android financial apps to investigate the safeness and the effectiveness of self-defense mechanisms adopted in the apps. For the analysis, we selected 200 free apps randomly selected from 400 apps in the Finance category of Google Play available in Republic of Korea. Of the 200 apps, we found that 73 perform a device rooting check, and 44 perform an app integrity check. Based on the observations of various self-defense mechanisms used in these apps, we constructed strategies for bypass attacks. Following these strategies, our bypass attacks successfully bypassed the device rooting checks in 67 out of the 73 apps and the app integrity checks in 39 out of the 44 apps.

Our analysis shows that once self-defense mechanisms are identified, bypassing them only requires modifying a few lines of bytecode or native code. We also conducted in-depth case studies of five popular security libraries used in our studied apps, demonstrating how our bypass attacks defeated a wide variety of tactics and complex control flows, which sometimes involved external servers.

Our contributions are twofold:

- We present an empirical study of the self-defense mechanisms employed in 76 popular Android financial apps. This was possible using MERCIDroid, an enhanced Android platform that traces method calls and constructs runtime call graphs. MERCIDroid captures not only direct method calls but also indirect method relationships, such as inter-thread and inter-component communications. These features are necessary to locate self-defense mechanisms more precisely.

- We show that our bypass attacks are effective in detouring most of the self-defense

mechanisms observed above with simple binary modifications. We detoured 67 out of 73 device rooting checks and 39 out of 44 app binary integrity checks.

The rest of the paper is structured as follows: Section 2 presents an overview of self-defense mechanisms. Section 3 describes the design and the implementation of MERCIDroid. Section 4 presents an empirical study of apps using MERCIDroid. Section 5 evaluates MERCIDroid's effectiveness and limitations. Section 6 discusses ways to improve self-defense mechanisms and the implications of our findings. Finally, Section 7 discusses related work, followed by our conclusions in Section 8.

Chapter 2

Design Characteristics of Self-Defense Mechanisms

To explain the typical structure of self-defense mechanisms, we begin by discussing an example of those used in a real app, AppC. The anonymized code in Listing 2.1 illustrates two distinct steps: The first checks whether the device is rooted, and the second displays an alert dialog if detected. We next describe each step in detail.

```
1  //Common ancestor
2  public class AppCActivity extends Activity {
3    public void onResume() {
4      int i = Lib0.check(this);
5      AppCLogger.e("AppCActivity", i);
6      if (i > 0) {
7        PopupDialog.showAlert(this);
8        setOccurError(true);
9        return;
10     }
11     if(SecureManager.checkRooting()) {
12       PopupDialog.showAlert(this);
13       setOccurError(true);
14       return;
15     }
16   }
17 }
18
19 //Environment checking
20 public class Lib0 {
```

```
21    private static int check(Context paramContext) {
22      return checkNative(paramContext);
23    }
24    private static int checkNative(Context paramContext) {
25      int i = Lib0Native();
26      return i;
27    }
28
29    private static native int Lib0Native() ;
30  }
31
32  //Execution termination
33  public class PopupDialog {
34    Dialog dialog;
35
36    public static PopupDialog showAlert(Activity paramActivity) {
37      PopupDialog alertDialog = new PopupDialog(paramActivity);
38      alertDialog.setType(100);
39      return alertDialog.show();
40    }
41    public void PopupDialog show() {
42      if (getType() == 100) {
43        this.dialog = alertDialogSetter();
44        this.dialog.show();
45      }
46    }
47    private AlertDialog alertDialogSetter() {
48      AlertDialog.Builder localBuilder = new AlertDialog.Builder();
49      localBuilder.setTitle(getTitle());
50      localBuilder.setMessage(getMessage());
51      return localBuilder.create();
52    }
53  }
```

Listing 2.1: AppC's device rooting check methods (decompiled with jd-gui [14]): We simplified the code for readability.

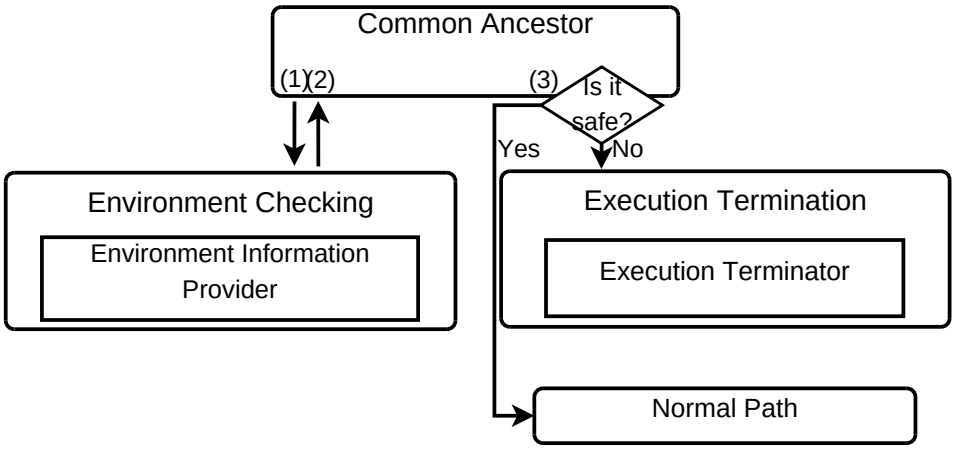
Step 1: Checking device rooting. `AppCActivity .onResume()` calls the native

method `Lib0.check()` to check whether the device is rooted (Line 4). `onResume()` then logs the result (Line 5). If `check()` returns a positive integer, implying that the device is rooted, the method calls `PopupDialog.showAlert()` (Lines 6–7) to show a dialog as described in Step 2. If `check()` does not detect that the device is rooted, `onResume()` calls another method to check device rooting (Line 11).

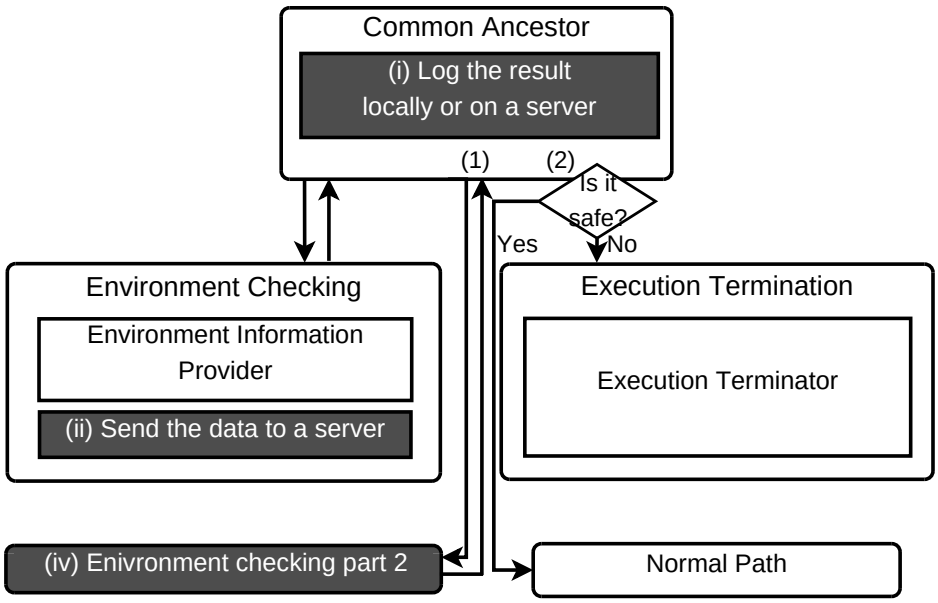
Step 2: Displaying an alert dialog. `PopupDialog.showAlert()` called in Step 1 calls `PopupDialog.show()`, and it calls `PopupDialog.alertDialogSetter()`. (Lines 7, 39, and 43, respectively). `alertDialogSetter()` finally sets the *AlertDialog* that warns the user before aborting the app (Lines 48–50).

Figure 2.1 shows two types of high-level structures of self-defense mechanisms observed from analyzing a large number of Android financial apps. Listing 2.1 fits the first type (Figure 2.1a). First, a method, which we label a common ancestor, calls a method to investigate the app binary and the platform on which the app is installed (Figure 2.1a(1)). This environment checking then calls **environment information providers**, which provide the system environment variables necessary for the check. Based on the values returned, the environment checking decides the app's integrity and/or device rooting and returns the result to the common ancestor method (Figure 2.1a(2)). If the execution environment is deemed unsafe, the common ancestor method calls the code to terminate the app (Figure 2.1a(3)). Typically, this execution termination calls an **execution terminator** that aborts the app by displaying an alert dialog that terminates the app process, or by finishing the app process directly. In the case of AppC, `AppCActivity.onResume()`, `Lib0.check()`, `PopupDialog.showAlert()` correspond to the common ancestor, the environment checking, and the execution termination, respectively. At least 93% of the self-defense mechanisms in our analysis followed this design (see Section 4).

Although self-defense mechanisms mostly follow the main execution path as shown in Figure 2.1a, many contain several extra steps that complicate the analysis. In the following, we describe the details of each block in the figures and explain why knowing the causal



(a) Type 1: Basic frame of a self-defense mechanism.



(b) Type 2: Basic frame with additional tasks. The additional tasks, which are optional, are represented as gray boxes.

Figure 2.1: The structure of self-defense mechanisms.

Type	Environment information providers	How to check the device rooting using the API [15]	Flagging conditions
Android API	`Runtime.exec()`	Run the *su* command	The command name is "su"
	`Package Manager. getPackage Info()`	Find whether a root-related app is installed	The package name is related to device rooting (e.g., eu.chainfire.supersu)
System call	`File.exists()`	Investigate the existence of *su* binary and app packages related to device rooting	The file name is "su" or APK file related to the device rooting
	`stat()`		
	`access()`		
	`open()`		
	`open()`	Check the *adbd* process' user ID	The file name is "/proc/[pid]/status"
	`opendir(), readdir(), chdir()`	Examine every file and directory	-

Table 2.1: Environment information providers for device rooting checks and their usage.

relationship between environment checking part and execution termination part is important to identify self-defense mechanisms.

2.1 Environment checking

The role of environment checking is to obtain information about the execution environment (e.g., the existence of a specific file, package information), decide whether the environment is unsafe, and return the result. For example, an app checks for device rooting by scanning for the existence of the *su* binary or permission management apps. To check app integrity, an app may check for the path of the Android application package (APK) file and its signature. To get information about the execution environment, the environment checking calls environment information provider(s). Thus, we can use the presence of environment

Type	Environment information providers	How to check the app tampering using the API [15]	Flagging conditions
Android API	`PackageManager.getPackageInfo()`	Get the signature of an app package	The flags include *GET_SIGNATURES*
	`Context.getPackageCodePath()`	Get the path of the app's package file	-
	`File.<init>()`	Handling the app's package file	The handled file is the APK file of the app
	`ZipFile.<init>()`		
	`RamdomAccessFile.<init>()`		
System call	`open()`		
Member variable	`ApplicationInfo.sourceDir`	The path of the app's package file	-
	`ApplicationInfo.publicSourceDir`		

Table 2.2: Environment information providers for the app integrity checks and their usage: Note that some Android APIs do not have the flagging condition. This means we need additional context to determine whether the APIs have indeed been called as part of the environment checking.

information provider(s) to locate the environment checking.

The environment information providers can be categorized into two types. The first type is Android APIs. For example, an app may use `File.exists()` to investigate the existence of specific files like the *su* binary. To get the APK file path, an app may call `Context.getPackageCodePath()` or read the `ApplicationInfo.sourceDir` variable. Based on the survey of known techniques [16, 17], we selected the Android APIs shown in Tables 2.1 and 2.2 as environment information provider(s). However, because these APIs can be called from parts other than self-defense mechanisms, we only consider them as the environment information providers when they met the conditions described in the tables.

Type	Execution terminators	How to use to terminate an app [15]	Terminating conditions
Android API	`AlertDialog.setMessage()`	Set a warning message for an AlertDialog	The message contains specific keywords(e.g., “rooting,” “tamper,” “integrity”)
	`TextView.setText()`	Set a warning message for a custom alert dialog	
	`Toast.makeText()`	Set a warning message for a Toast message	
	`Intent.setAction()`	Launch the application uninstaller	The package name to uninstall is that of the app itself.
	`Process.killProcess()`	Kill the app’s process	Process ID to kill is that of the app itself
	`System.exit()`		-
System call	`exit()`	Kill the app’s process in a native function	-

Table 2.3: Execution terminators and their usage. `System.exit()` and `exit()` do not have a terminating condition because an app calls them to terminate only the app itself.

The second type is system calls. If an app uses a native library, the app can get execution environment information regardless of Android APIs, since the Android platform is based on the Linux kernel, and every process must use system calls, mostly through wrapper C functions, to get the system information from the kernel. Therefore, we trace the system calls to locate the environment checking. Similar to the Android APIs, we list the system calls in Tables 2.1 and 2.2 as environment information provider(s). We explain how apps use the Android APIs and system calls in their self-defense mechanisms in more detail in Sections 4.2 and 4.4.

2.2 Execution termination

If an environment checking part detects a tampered environment, the common ancestor calls the execution termination to prevent the app from executing in the environment. A common way to abort the execution is to show an alert dialog with a button to terminate the app. The alert dialog typically contains a warning message that describes the reason for the termination. Some apps directly kill the running processes or uninstall themselves from the device without showing the warning message to the user. Table 2.3 shows all execution terminators found after analyzing our target financial apps' behaviors. Similar to environment information providers, we consider the APIs or the system call as execution terminators only when they meet the terminating condition.

2.3 Challenges to locating self-defense mechanisms

Even if the environment checking and execution termination parts have their own characteristics, finding just one of the parts is not enough to locate a self-defense mechanism. For example, syntactically searching for environment information providers alone is insufficient because these can be called for several reasons other than self-defense mechanisms, even if they meet the conditions described in Tables 2.1 and 2.2. For example, an app reads its package file not only to check its integrity, but also to get resources. We need to know if the call of an environment information provider leads to the call of an execution terminator. To find it, we should be able to connect the environment information provider and execution terminator during runtime.

Similar to the environment checking, finding only execution terminators is not sufficient to locate the self-defense mechanism. First, an app displays an alert dialog not only when a self-defense mechanism detects a tampered environment but also when it has other problems (e.g., network or memory problems). An alert message set in the dialog is necessary to filter alert dialogs that are not related to self-defense mechanisms. For example, the warn-

ing message related to a self-defense mechanism typically contains specific keywords such as “rooted” and “forged.” Second, just skipping execution terminators does not make the app enter the normal control flow because the branch point between the execution termination and the normal path lies in the common ancestor (Figure 2.1b(2)). In order to bypass self-defense mechanisms, we should modify the branch point or the checking part.

In addition to above difficulties to locate a self-defense mechanism, apps often employ extra steps beyond a single checking part and a standard execution termination step as shown in Figure 2.1b. First, the checking part may communicate with an external server for the integrity check (Figure 2.1b(ii)). For example, one possible way to check the app binary integrity is to send a hash value of the app’s APK file and ask the server to verify it. Second, a common ancestor may log the result returned from the checking part by using Android Logcat or by sending it to an external server, as shown in Line 5 in Listing 2.1 (Figure 2.1b(i)). These steps add more method calls between environment checking and execution termination, making it difficult to find causality between them. Third, a common ancestor sometimes calls more than one environment checking as shown in Figure 2.1b(1) and (iv). The common ancestor then collects results from the environment checkings and makes a decision. For example, Listing 2.1 shows that `AppCActivity.onResume()` calls two environment checkings: `Lib0.check()` and `SecureManager.checkRooting()`. These extra tasks increase the number of method calls between the checking part and the execution termination part, which makes understanding the connection between the two parts complicated.

Chapter 3

Analysis Methodology

This section describes a tool that we developed to locate the code relevant to self-defense mechanisms in Android apps. In particular, we focus on how our tool traces various indirect method calls and native calls at runtime.

3.1 Key Insight

To narrow down the code path to investigate, we can start with finding known environment information providers or execution terminators. However, searching for them separately is not efficient and often generates false positives. For example, a financial app that we analyzed contained approximately 30,275 methods on average and, therefore, false positives would significantly slow our analysis as each case needs to be manually examined.

The key insight for improving the accuracy is to find the common ancestor between the environment information provider(s) and the execution terminator in the control flow graph recorded at runtime. To construct a call graph, we perform the following steps. First, we record all the invoked (Java method and native function) calls and returns while a target app executes. While tracing, we record additional information to connect indirect relationships between threads and Android components and to identify environment information providers. Second, when an execution terminator is called, we flush the recorded data into a file. Finally, we parse the records and find common ancestors between the identified environment information providers and the execution terminator, and locate the common ancestor that is closest

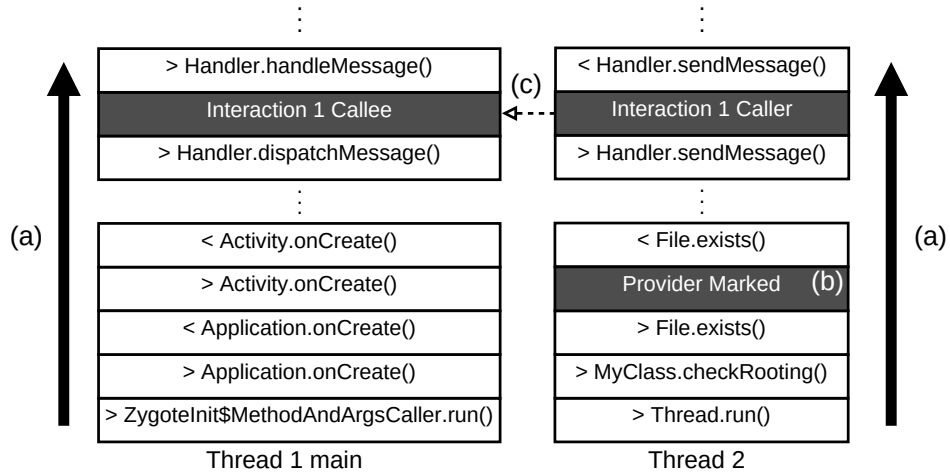


Figure 3.1: Record of method invocations/returns in each thread. Each block is the recorded method invocations/returns from bottom to the top (a). An environment information provider that meets the condition is marked (b). The indirect caller and callee are connected through an interaction ID (c).

to the execution terminator.

Why not taint tracking? One well-known way of analyzing an app is using the taint tracking methodology, which patches the taint tag to data and traces it by detecting the tag. If we use this methodology, we can consider a way to add a taint marking to a return value of an environment information provider and detect it from an execution terminator. However, we cannot use this methodology because the return value of the environment information provider is normally used as a condition for a *if-else* statement in the common ancestor, thus the taint tag is not propagated in the statement. For example, in the case of the device rooting check shown in Listing 2.1, a taint tag patched to the return value of the `Lib0Native()` (Line 25) would be propagated to *int i* in Line 4, but not to an argument of the `PopupDialog.showAlert()` because the two values are independent of each other. Therefore, we cannot detect the taint tag from the execution terminator of the device rooting check. This means that we can only locate an environment checking using the methodology, which is not enough as described in Section 2.1. To overcome this limitation,

the taint tracking system should support the control flow propagation. However, according to the TaintDroid research [18], developing such a mechanism for Android apps is difficult because Dalvik bytecode does not maintain the branch structure. We conclude to trace the control flow of an app by enhancing the Android platform instead of using the taint tracking methodology.

We next explain the design and implementation of the steps necessary to construct a call graph.

3.2 Tool Design and Implementation

In order to locate common ancestors between environment information providers and execution terminators, we need to track caller-callee relationships across threads, components, and process boundaries. MERCIDroid has two parts: 1) collecting control flow information by tracing calls at runtime; and 2) extracting a call graph containing the methods related to a self-defense mechanism.

3.2.1 Recording method calls at runtime

We modified Android 4.4.4 to record method calls during the execution of a target app. The modified system also records additional information to identify the relationships between method calls across threads, components, and processes.

Recording method calls per thread. Using a similar approach as in Compac [19] and the Method Trace function in Android Monitor [20], we modified the Dalvik Virtual Machine (VM) to record the method calls in each thread. We modified a portable C implementation of *mterp*, the interpreter that interprets and executes Dalvik bytecode. In particular, we focused on *invoke* and *return* instructions, which are related to method invocations and returns, respectively. To record the instructions, we first modified the Dalvik VM to allocate the additional space in each thread to store invocation and return histories. We also modified the

portable C code related to the instructions to record the method invocations and returns in the additional space as shown in Figure 3.1(a).

Flagging relevant environment information provider calls. While recording method calls, MERCIDroid flags the calls for the environment information providers, as described in Tables 2.1 and 2.2. MERCIDroid uses runtime arguments to check the marking conditions. If the condition holds, MERCIDroid adds an extra record between the method invocation and the method return to flag the method call as shown in Figure 3.1(b). Because `ApplicationInfo.sourceDir` and `ApplicationInfo.publicSourceDir` are member variables but not Java methods, we instead flag a method call that accesses them. Also, to flag environment information providers whose type is a system call, MERCIDroid executes *strace* and parses the result.

Connecting indirect relationships of method calls. Tracing direct caller-callee relationships only is insufficient to generate a call graph that contains a self-defense mechanism. For example, a thread that manipulates the UI must send a *Message* to the UI thread using `Handler.sendMessage(Message msg)` [15]. Then, in the UI thread, `Handler.dispatchMessage(Message msg)` calls another method to handle the message. In this case, `sendMessage(Message msg)` and `dispatchMessage(Message msg)` are not in a direct caller–callee relationship. Thus, to link these two method calls, we need more information. Figure 3.2 illustrates many types of indirect caller–callee relationships that we need to track. The first three cases in the figure correspond to typical call graphs representing interactions between threads. The last three cases show communications between Android components (Activity, BroadcastReceiver, Service). Those relationships are not connected by direct method calls because the Android system mediates the communication.

To connect the indirect caller-callee relationships, MERCIDroid allocates a unique ID to a message object that both the caller and the callee use. MERCIDroid adds a unique ID to a `Message` object such that we can link the method calls with a `Message` object by

checking its ID. To track interactions between Android app components (Activity, Service, and Broadcast Receiver), MERCIDroid adds a unique ID in an *Intent* object [15]. Figure 3.1(c) illustrates how this added ID can be used to link the indirect method calls.

Storing method call records. When an app is stopped by execution terminators, the system stores the method call records in a file. We manually identify the execution terminators that are likely to be related to a self-defense mechanism by examining the dialog messages in the apps. For example, in many cases, the dialog message includes keywords such as "rooted" and "forged." We then modify the Android APIs so when such an execution terminator is called, the system stores all the records collected thus far in all threads in a file. With the collected data, we can backtrace the control flow from an execution terminator.

3.2.2 Constructing an SDMGraph

We implemented a call graph generator which parses the records, constructs a call graph including the aforementioned indirect relationships, and extracts a self-defense mechanism graph (SDMGraph), which contains only the methods relevant for identifying the self-defense mechanisms.

Once a call graph is constructed, the generator extracts the SDMGraph in which the root node is the common ancestor of the execution terminator and the environment information provider(s). To find the common ancestor, it first recursively checks the environment information provider(s) that are flagged at runtime and their ancestors. Then, it tracks the ancestors of the execution terminator, until the generator finds the one that has been already flagged. The first flagged ancestor node encountered during this backtracking step from the execution terminator is the closest common ancestor. It then constructs this SDMGraph using *graphviz* [21]. By narrowing the scope of the methods to those in the SDMGraph, we can manually disassemble and analyze a small number of methods to find one that we can modify to bypass the self-defense mechanisms.

Here we describe the SDMGraph construction process with AppZ as an ex-

ample. Figure 3.3 shows an example SDMGraph of a device rooting check performed by AppZ. The app uses `ProcessManager.exec()` as an environment information provider and `AlertDialog$Builder.setMessage()` as an execution terminator. To find the common ancestor of the two method calls, the script runs the following steps. First, it checks `ProcessManager.exec()`, already flagged at runtime, and its ancestors (Figure 3.3(a)). Second, it tracks the ancestors of `AlertDialog$Builder.setMessage()` (Figure 3.3(b)). It finally constructs a graph in which the root node is `MainActivity$1$1.run()`, the first flagged ancestor node. By considering methods only under the common ancestor, `MainActivity$1$1.run()`, we can identify the relationship between the environment information provider and the execution terminator. As shown in the figure, the Interaction ID connects `Handler.enqueueMessage()` and `Handler.dispatchMessage()` (Figure 3.3(c)).

3.2.3 Handling inter-process communication

Android components can communicate across different processes. For example, an app can execute its service in a separate process using the *android:process* manifest attribute. An app can also request the self-defense mechanisms of a separate security app. Android processes use *Intent* to execute an Android component in another app, and *Parcel* to send a data to another process. Therefore, we use instrumented Intent and Parcel to connect two processes' communication methods.

To capture inter-process communications, we parse collected method calls from all the processes running in the system. Therefore, when an execution terminator is called in one process, the other processes should be aware of this event and flush the recorded methods into a file. To support this scenario, we added a new system service, MERCIService, and when a process begins execution, it registers itself with it. If one of the registered processes is terminated by an execution terminator, MERCIService triggers each registered process to

store the records in a file.

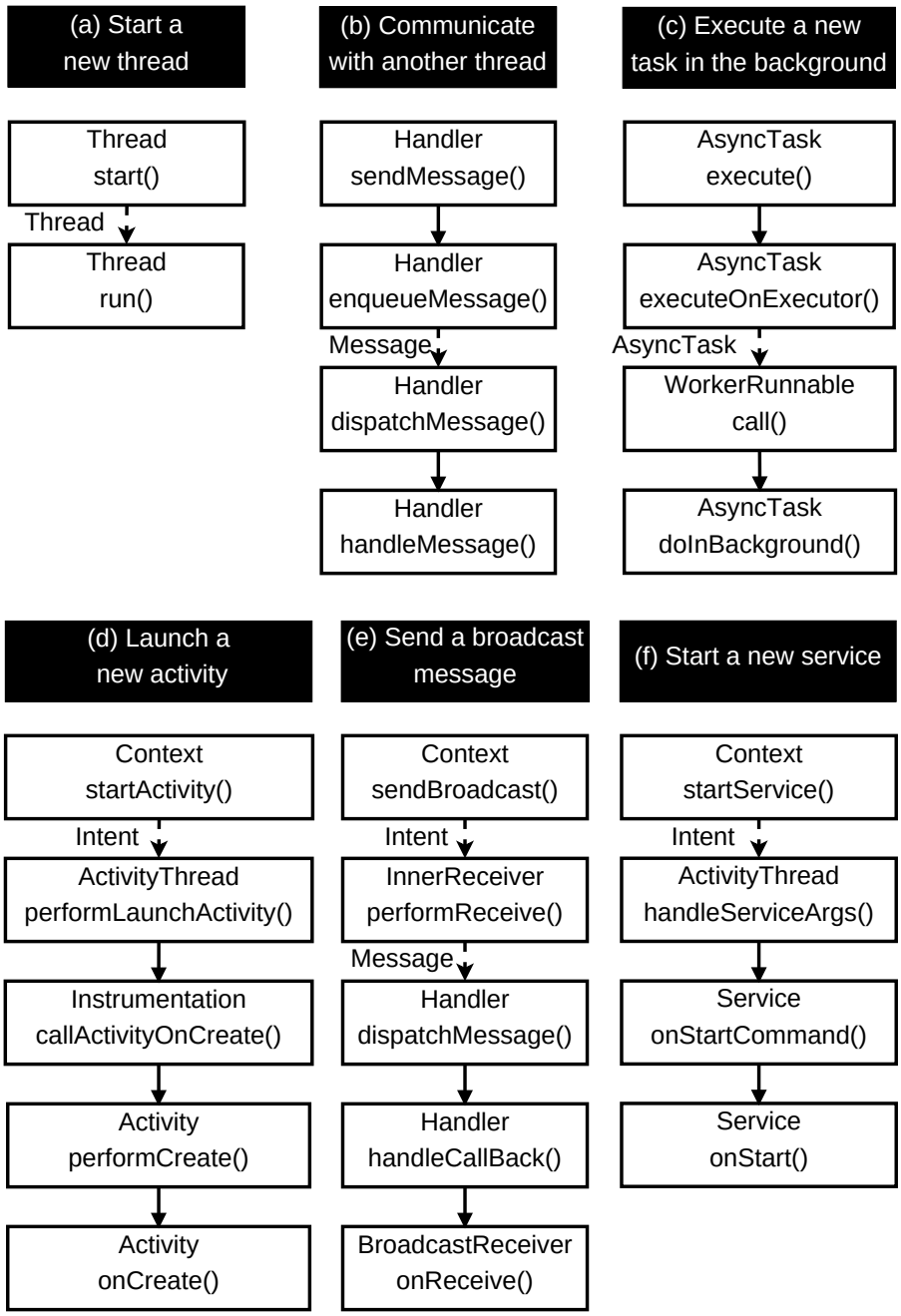


Figure 3.2: Indirect caller-callee relationships.

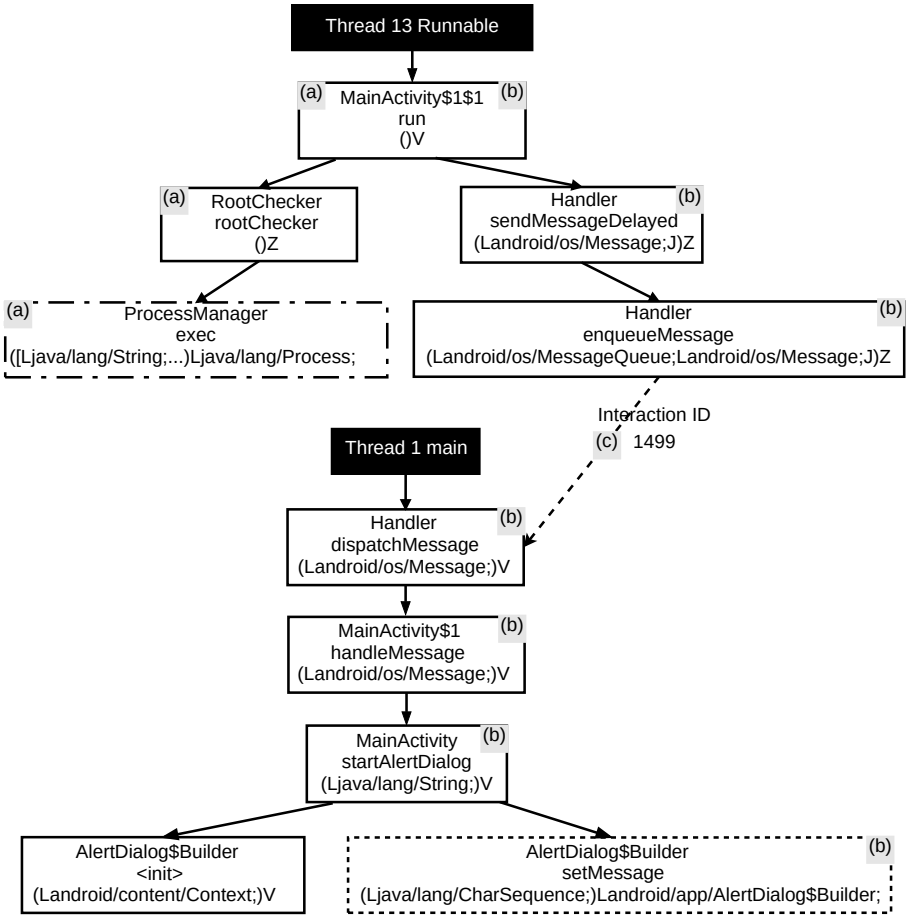


Figure 3.3: An SDMGraph of the device rooting check of AppZ. We simplified the code for readability.

Chapter 4

App Analysis & Bypass Attacks

This section shows our analysis results of selected Android apps using MERCIDroid. We first describe how we chose the apps to analyze. Based on observed characteristics of self-defense mechanisms, we next discuss strategies that we constructed to bypass self-defense mechanisms and show how we actually bypassed them using these strategies. We then introduce security libraries used in the studied apps and present case studies of the top five popular libraries, demonstrating how effectively our bypass attacks defeat seemingly complicated self-defense mechanisms employed by those libraries.

4.1 App Selection

In January 2016, we crawled 400 apps in the "Finance, Free" category of Google Play in a country where mobile banking is widely adopted. We then randomly selected 200 apps for analyses. To find apps that perform self-defense mechanisms, we used the following steps. First, to find apps that check the device rooting, we first installed each app on a rooted device and executed it. If the activity had a button, such as "Press to start," we pressed the button to see whether the app proceeded to the next activity. If the app displayed an alert dialog with a message like "This device is rooted. This app will be terminated," or it killed or uninstalled itself, we concluded that the app checks device rooting and included it in our dataset. To find apps that perform an app binary integrity check, we disassembled the app binary, added an empty class to the code, and reassembled a DEX file of the app. We then executed the

Device rooting check: 73 apps		
SDMGraphs generated: 67	R_Group_API	14
	R_Group_Native_1	57
	R_Group_Native_1	6
SDMGraphs not generated: 6	Limit_Case_2	1
	Limit_Case_3	5

(a) Analysis result for the device rooting checks.

App tampering check: 44 apps		
SDMGraphs generated: 39	I_Group_Predictable_Return	16
	I_Group_Signature	2
	I_Group_APK	14
	I_Group_Context	10
SDMGraphs not generated: 5	Limit_Case_1	3
	Limit_Case_2	2

(b) Analysis result for the app integrity checks.

Table 4.1: Analysis result for the device rooting checks and app integrity checks. Note that multiple methodologies are applicable to bypass the self-defense mechanisms of apps.

modified app binary, and if the app showed an alert dialog with a message like "This app is tampered. This app will be terminated," or if it killed or uninstalled itself, we include it in our dataset.

Additionally, we excluded the following apps for further analysis: (a) 8 apps without a main activity: these apps run like a daemon process or need another app to run together; (b) 14 malfunctioning apps: these apps crash after app repackaging upon launching; (c) 1 app that cannot be disassembled or reassembled with the *apktool* and the *smali* tools; and (d) 1 app requiring an Android platform whose version is higher than 4.4 (Section 5.2 Limit_Case_4).

This left 76 apps that perform one or more self-defense mechanisms: 73 of the 76 apps check device rooting, and 44 of the 76 apps check app integrity. We ran all 76 apps using MERCIDroid to generate a SDMGraph and apply our bypass attacks.

4.2 Bypass Attacks

Using MERCIDroid, we successfully constructed the SDMGraphs for 67 out of 73 apps that check device rooting and 39 out of 44 apps that check app integrity. We then successfully

```
new-instance v0, Ljava/io/File;
const-string v1, "su"
const-string v1, "us"
invoke-direct {v0, v1}, Ljava/io/File;->
<init>(Ljava/lang/String)V;
invoke-virtual {v0}, Ljava/io/File;->
exists()Z
move-result v2
```

(a) Modify an argument for an Android API

```
new-instance v0, Ljava/io/File;
sget-object v1, Lcom/execution/environment;->su;
invoke-direct {v0, v1}, Ljava/io/File;->
<init>(Ljava/lang/String)V;
invoke-virtual {v0}, Ljava/io/File;->
exists()Z
move-result v2
const v2, 0x0
```

(b) Overwrite a return value

```
#.method public native integritycheck()Z;
.method public integritycheck()Z;
const v0, 0x0
return v0
.end method
```

(c) Change a native method declaration to a Java method, which returns a fixed value

```
.method public integritycheck()Z;
const v0, 0x0
return v0
# The rest are ignored...
.end method
```

(d) Fix a return value

Figure 4.1: Example smali [22] code to bypass self-defense mechanisms. The lines added to bypass the self-defense mechanism are presented in bold.

```
1 invoke-virtual {p0}, Lcom/execution/MainActivity;->
2 getPackageCodePath()Ljava/lang/String;
3 move-result-object v0
4 invoke-static p0, Lcom/execution/FakeActivityManager;
      ->getUntamperedPackageCodePath
      (Landroid/content/Context;)
      Ljava/lang/String;
5 move-result-object v0
6 invoke-static {v0}, Lcom/execution/environment;->
7 integritycheck(Ljava/lang/String;)Z;
```

(e) Generate a file path for the original app package

```
1 new-instance v0, Lcom/execution/FakeContext;
2 invoke-direct v0, p0, Lcom/execution/FakeContext;-><init>
      (Landroid/content/Context;)V
3 invoke-static {v0}, Lcom/execution/environment;->
4 integritycheck(Landroid/content/Context)Z;
```

(f) Generate a FakeContext

Figure 4.1: Example smali [22] code to bypass self-defense mechanisms (cont.). The lines added to bypass the self-defense mechanism are presented in bold.

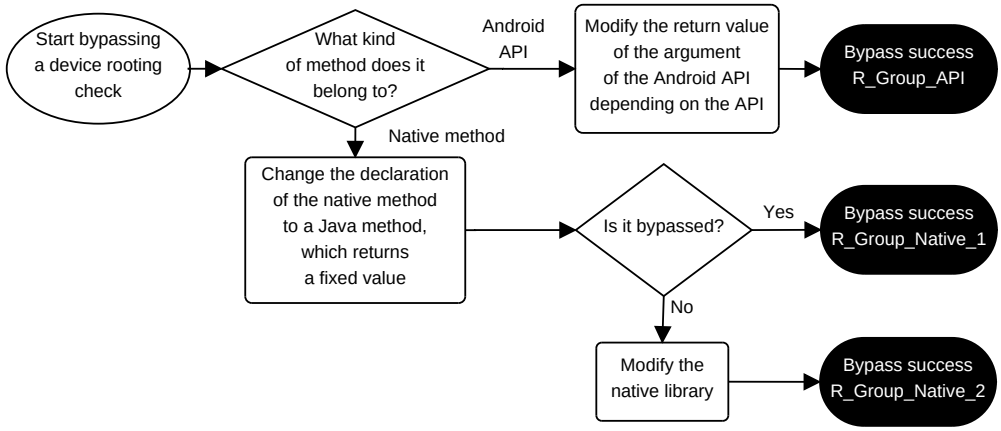


Figure 4.2: A flowchart for bypassing device rooting checks.

bypassed *every* self-defense mechanism for which an SDMGraph has been generated. See Section 5 for more details on the effectiveness and limitations of MERCIDroid.

To bypass a self-defense mechanism, we run through each marked method in a SDMGraph by following the flowcharts shown in Figures 4.2 and 4.3. We constructed the flowcharts based on our sample data and our trial and error experience. We first tried an easier technique to bypass an identified self-defense mechanism. If the technique fails, we progressively tried more difficult techniques. In each step, we rewrote the app's Dalvik bytecode using techniques described in Figure 4.1. If we succeed in bypassing a self-defense mechanism, we placed the app in a corresponding group.

Table 4.1 shows the number of apps included in each group. Note that the sum of the number of apps in the groups included in each flowchart is greater than the number of apps that perform the same self-defense mechanism. Therefore, some apps check the device rooting and app tampering more than once; we bypassed each one of them using a technique suited for each self-defense mechanism. Below, we explain the flowcharts and describe the characteristics of the self-defense mechanisms in each group and how we bypass them.

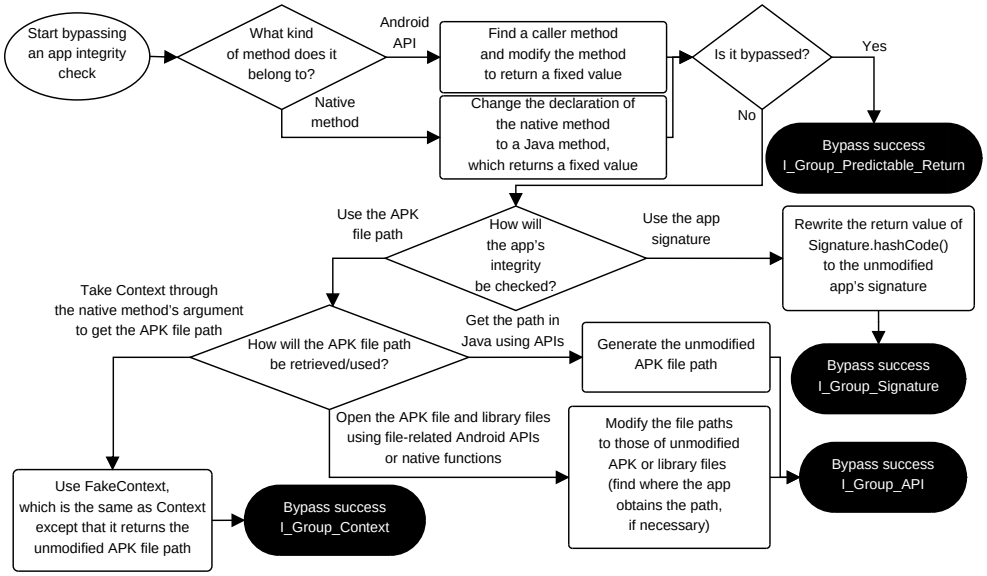


Figure 4.3: A flowchart for bypassing app integrity checks.

4.2.1 Bypassing device rooting checks

Figure 4.2 shows strategies used to bypass device rooting checks. We first identified whether the marked method is an Android API or a native method. If the method is an Android API, we modified the return value or the argument of the method (R_Group_API). Otherwise, we fixed the return value of the native method (R_Group_Native_1). If both fail, we patched some bytes in a native library (R_Group_Native_2). Below, we explain each attack in detail.

R_Group_API. Apps in this group use Java methods for checking device rooting. They use Android APIs in Table 2.1 to detect the existence of binaries or apps related to device rooting. For example, some apps check the presence of the "su" binary using the `File.exists()` API. Similarly, some apps detect the availability of the "su" command in the system using `Runtime.exec()`. If the device is rooted, the method calls without an error. Otherwise, the method throws an exception because it cannot find the "su" command. Apps also use `PackageManager.getPackageInfo()` to check whether apps related to device rooting (e.g., SuperSU) have been installed.

For these apps, we modify an argument or the return value of an Android API (Figure 4.1a or 4.1b). We change the name of a file the apps try to find (e.g., "su" to "us") or fix the return value to a specific value such as *false*, based on the property of the API.

R_Group_Native_1. This group contains apps that check device rooting in native libraries. The Java code in the apps includes the declaration of native methods mapped to functions in native libraries. The apps use the native functions by calling the native methods. For example, apps use `open()` or `stat()` to check for the existence of the "su" binary. We list the native functions used by the self-defense mechanisms at the native level in Tables 2.1 and 2.2.

In this case, we change the native method's declaration to a fake Java method that returns a fixed value (Figure 4.1c). This is made possible because it is easy to predict the return values of the native methods when a device is not rooted. For example, if a native method returns a boolean value, the value only can be *true* or *false*. A return value of an integer or string type is also not hard to predict if the value does not contain a random value. For example, obtaining integer zero or String "0" or "0000" for the return value means "No Problem" in most cases.

R_Group_Native_2. Device rooting checks for apps in this group are similar to those in R_Group_Native_1, except that the native methods return unpredictable values. If a native method returns a byte array, or a string that contains randomly generated values, predicting the value when a device is not rooted and overwriting the return value is difficult. In this case, we modify the native code using IDA Pro [23] to rewrite bytes in the text section to skip the check.

4.2.2 Bypassing app integrity checks

Figure 4.3 shows steps needed to bypass app integrity checks. Contrary to the device rooting check, modifying an argument or the return value of an Android API is more difficult than predicting and fixing the caller's return value because Android APIs used by app integrity checks handle an APK file path or an app's signature (see Table 2.2). These values

Name	# of apps that adopt the libs.			Additional features			Server communication to check/report the result
	All	Device rooting check	App integrity check	Malware detection	Anti-reverse engineering	Other features	
Lib0	30	30		O			
Lib1	11	11		O			
Lib2	10	7	8		O	Secure session ID	O
Lib3	7	7	7		O	Cryptography algorithms	O
Lib4	6	5	6			One-time verification token	O
Lib5	4	4		O			
Lib6	3	3		O			O
Lib7	2	2		O			
Lib8 §	2	2	2		O		O
Lib9	3		3				O
Lib10	1	1	1		O		O
Lib11	1	1			O		O

§ Written in Java. Other libraries are native libraries.

Table 4.2: List of libraries that contain self-defense mechanisms.

are app-specific, so we should generate the values for the unmodified apps to modify the values. Therefore, we first attempted to fix its caller's return value before modifying the API's argument or return value. When the marked method is a native method, we try to change its declaration to a Java method that returns a fixed value (I_Group_Predictable_Return). When we are unable to fix the return value of the method, we modify the app's signature (I_Group_Signature) or APK file path (I_Group_APK, I_Group_Context) to an unmodified app's signature or APK file path. We describe the characteristics of each group in more detail below.

I_Group_Predictable_Return. Similar to the *R_Group_Native_1*, checking part methods in the apps in this group returns predictable values, such as *false* or 0, when the apps are not

Name	Device rooting check	App tampering check	Malware detection	Anti-reverse engineering	Other security features
AppA	Lib0	S, Lib2	Lib0	Lib2	Lib2
AppB	S, Lib7	S	Lib7		
AppC	S, Lib0		Lib0		
AppD	Lib5, Lib4	Lib4	Lib5		Lib4
AppE	Lib0	S, Lib2	Lib0	Lib2	Lib2
AppF	Lib0, Lib4	Lib4	Lib0		Lib4
AppG	S, Lib2	Lib2		Lib2	Lib2
AppH	S, Lib1		Lib1		
AppI	Lib1, Lib8	Lib8	Lib1	Lib8	
AppJ	Lib1, Lib8	Lib8	Lib1	Lib8	
AppK	S, Lib3, Lib1	Lib3	Lib1	Lib3	Lib3
AppL	Lib2, Lib0	Lib2	Lib0	Lib2	Lib2
AppM	Lib0, Lib3	Lib3	Lib0	Lib3	Lib3
AppN	S, Lib0		Lib0		
AppO	Lib2, Lib1	Lib2	Lib1	Lib2	Lib2
AppP	Lib2, Lib1	Lib2	Lib1	Lib2	Lib2
AppQ	Lib2, Lib1		Lib1	Lib2	Lib2
AppR	Lib1, Lib3	Lib3	Lib1	Lib3	Lib3

S: Self-implemented device rooting checks/app tampering checks

Table 4.3: List of apps that check self-defense mechanisms multiple times.

forged.

When the marked method is a native method, we change the method's declaration to a Java method that returns a fixed value (Figure 4.1c). In the case of an Android API, we find the caller and make the method returns a fixed value (Figure 4.1d).

I_Group_Signature. This group includes apps that try to verify their signature. Apps can get their signature using `PackageManager.getPackageInfo()` with a *GET SIGNATURES* flag (see Table 2.2). The signature of the rewritten app is different to that of the unmodified one. Therefore, examining the signature can be a way to check app integrity.

To bypass the app integrity checks using signatures, we obtain the pure app's signature

and overwrite the return value of `Signature.hashCode()` to the obtained signature.

I_Group_APK. This group contains apps that check app integrity by reading the app's package file or library. The apps get the file path of the APK file or library file using Android APIs, such as `Context.getPackageCodePath()`, `ApplicationInfo.sourceDir`, and `ApplicationInfo.publicSourceDir`, as shown in Table 2.2. The apps then read the file using *File*, *RamdomAccessFile*, or *ZipFile* classes to compute the hash value (see Table 2.2). The apps send the computed hash value to an external server to check their tampering.

To allow the tampered app to read the unmodified app's package and native library file instead, we first copy the unmodified files into the tampered APK file. Then, we implement the `getUnmodifiedPackageCodePath()`, which copies the unmodified files to its private storage and returns the file path. We use this method to bypass the checks (Figure 4.1e).

I_Group_Context. Apps in this group perform checks through a native code that takes a *Context*, which contains the APK file path, as an argument. The APK file path in the Context can be obtained with `Context.getPackageCodePath()`, `Context.getApplicationInfo().sourceDir`, or `Context.getApplicationInfo().publicSourceDir`. The native code then reads the APK file and checks for app tampering, similar to *I_Group_APK*.

For such cases, we implement *FakeContext*, which is exactly same as *Context* except that it contains the path to the unmodified APK file generated by `getUnmodifiedPackageCodePath()` (Figure 4.1f).

4.3 Third-party Libraries

On close examination, we found that 60 (out of 76) apps integrated one or more security libraries that implement self-defense mechanisms. We used the Java package names and the file name of the libraries to infer the security company who produced the library.

Table 4.2 shows those libraries and additional security features that the libraries support. All libraries except Lib8 are written in native code. From the library code and the security companies' web pages, we discovered that the libraries provide various security functions other than self-defense mechanisms. The most common function provided by the libraries is malware detection. The libraries scan malicious apps and processes directly or by transacting with security apps developed by the same vendor. Also, the libraries support various features, such as code obfuscation, anti-decompile, and emulator detection, to prevent the reverse engineering of apps.

We found that some libraries communicate with an external server while running self-defense mechanisms. As described in Section 2, the libraries send data or the examination result to the server to be validated, which results in a complex call graph.

Why would some apps execute the same self-defense mechanism multiple times? We found that some apps check device rooting or app tampering more than once as shown in Table 4.3, because they adopt security libraries include the checks. Some apps first call self-implemented self-defense mechanisms, which are simply duplicates of the checks already performed in the security libraries included in the app. In addition, some apps include multiple security libraries that call the same self-defense mechanism.

Risk of using security libraries. If many apps share the same library, and the library has not been mutated across them, these apps would be vulnerable to the same bypass attack. For example, as shown in Table 4.2, 30 apps use Lib0 to perform device rooting checks and our single bypass attack defeated 17 out of these 30 apps. The rest needed more than one type of bypass attacks.

4.4 Case Studies

Given the popularity of security libraries, we did an in-depth analysis of self-defense mechanisms implemented in those libraries. Specifically, we chose the top five libraries, Lib0, Lib1,

Lib2, Lib3, and Lib4. Although the libraries use various ways of checking device rooting and app integrity, we bypassed them using the app rewriting techniques described above.

4.4.1 How to check device rooting

The libraries use various system calls to check device rooting. Normally, they concentrate on finding executable binary files and app package files related to device rooting, but some libraries use other ways to examine platform integrity.

Lib0. We found two ways that Lib0 checks device rooting. First, the library checks predictable file paths in the su binary (e.g., */system/xbin/su*, */system/bin/su*). To check whether a su binary exists in the file path, the library uses the `stat()` [24] system call. The system call takes the file path as the first argument, stores the status of the file in a buffer provided by the second argument, and returns the existence of the file. The library checks the existence of the su binary by calling `stat()` for each predictable file path.

Second, the library checks the user ID of the *adbd* process. The library scans the */proc/* directory, which contains numerical subdirectories for running processes [24], which is named by the processes' ID, to find adbd's process ID. The library then reads the */proc/[pid]/status* file, which contains the user ID of the process, using `read()` system call. If the device is not rooted, the user ID of adbd is 2,000, which is the *shell* user's ID. Otherwise, the user ID of adbd is zero, which is the *root* user's ID. Therefore, the library can check device rooting using adbd's user ID.

Lib1. To check device rooting, Lib1 tries to open or access files or directories related to device rooting. First, it calls the `access()` system call with an *F_OK* flag to check whether private directories for the apps related to device rooting (e.g., *com.marutian.quckunroot*, *com.noshufou.android.su*) are accessible. If a private directory exists, `access()` returns zero, which means an app that uses the directory is installed. Otherwise, it returns -1.

Second, the library tries to open the *su* binary and root-related apps' APK files using the `open()` system call with an *O_RDONLY*(Read-Only) flag. If the file can be opened as

read-only, `open()` returns its file descriptor number. Otherwise, if the file does not exist, `open()` returns -1. Using the system call, the library checks whether an su binary exists in the predictable path and whether root-related apps' APK files are stored (e.g., */system/app/-superuser.apk*, */data/app/com.noshufou.android.su-2.apk*).

Lib2, Lib3. Similar to Lib1, Lib2 and Lib3 try to open the su binary and root apps' APK files using the `open()` system call.

Lib4. Lib4 employs various ways of examining the platform's integrity in addition to checking root-related apps and binaries, like other libraries. First, the library finds the system property *ro.kernel.qemu*'s value. If the value is one, the method considers that the app runs in an emulator. Second, the library searches every file and directory under the root directory recursively using `opendir()`, `readdir()`, and `chdir()` system calls, except for some unimportant directories such as */sdcard/* and */mnt/*. If the library finds a file whose name is "su" or that contains a string like "noshufou" or "supersu," it concludes that the platform has been tampered with.

4.4.2 How to check app integrity

The libraries use a similar methodology to check an app's integrity; they take the app's APK file path, read the file, calculate the hash value, and compare the value with the one stored in an external server. They are always able to successfully obtain the APK file path using Java methods because the path varies by device.

Lib2. To check app integrity, Lib2 reads the app's APK file and library file. The library takes the files' paths as string-type arguments. The library first checks whether the APK file's path is valid. If the file is stored in either */data/app/*, */data/app-private/*, or */mnt/asec*, and the file's name starts with the app's package name, the library considers the path valid. The library then reads the files, calculates the hash values, and sends them to the app developer's server for the validation of app integrity.

Lib3. Similar to Lib2, Lib3 examines the app's integrity by reading the app's APK file and computing its hash value. The library then sends the data to the app developer's private server.

Lib4. Lib4 takes a *Context* as an argument. The library then gains the app's APK file path from `Context.getApplicationInfo().sourceDir` and computes the hash value of the file. The library sends the hash value to the app developer's private server to check the app's integrity.

4.4.3 How to check whether the library function is not bypassed

We find that AppL, which has integrated Lib2, uses an external server to probe whether the self-defense mechanism has been called. So, if we skip the checks, then the app can get the signal from the server. Therefore, simply rewriting the native method's declaration to a Java method does not work for AppL, and we used a different bypass attack in this case.

4.4.4 How to terminate an app running in an unsafe condition

The security libraries, except Lib3, return the checking result to a Java method instead of directly showing an alert dialog. Android supports *NativeActivity*, which can be implemented purely in native code, for apps that require high performance, such as gaming apps [15]. However, using it only for displaying an alert dialog is ineffective. Therefore, the libraries just return the result instead of launching their own activity, and the app's main code shows an alert dialog.

Lib3. Lib3 has characteristics other than libraries in terms of how it terminates the app when the checks fail. If the library judges that the app is running under unsafe conditions, unlike other libraries, it does not return and terminates the app using the `exit()` system call. To store the method call records before termination, we had to register a wrap-up function using the `atexit()` function, which registers a function that should be called in normal process termination [24].

4.4.5 How to return the result and how to bypass

Self-defense mechanisms in libraries do not deviate from those described in Section 4.2, so we can bypass them using the techniques described in the same section. In this section, we introduce which values the self-defense mechanisms in each library returns and how to bypass them in more detail.

Lib0. If the device is not rooted, Lib0's native method returns zero. Otherwise, the method returns -1. We bypass the check by editing the native method's declaration to a Java method, which returns zero.

Lib1. Lib1's native method for a device rooting check returns a string "0" when the device is not rooted. Otherwise, the method returns "1." We bypass the check by editing the native method's declaration to a Java method, which returns "0".

Lib2. If the device is not rooted and the app has not been tampered with, Lib2's native method returns zero. Otherwise, the method returns an error number as an integer. Normally, we can bypass the checks by editing the native method's declaration to a Java method, which returns zero. In the case of AppL, we must modify the library file and the APK file path instead of changing the native method's declaration.

Lib3. When a device rooting check and an app tempering check pass, Lib3's native method returns a string "0000." If they do not, the library terminates the app immediately or returns a four-digit number as a string, such as "9002" when the device is rooted, or "9001" when the app has been tampered with. In both cases, we bypass the checks by editing the native method's declaration to a Java method, which returns "0000."

Lib4. Because Lib4's native method returns a byte array, it is hard to predict the return value when an app runs in an untampered environment. Therefore, we cannot rewrite the method's declaration to a Java method. Instead, we modify the native library file to bypass the device rooting check and use *FakeContext* as described in Section 4.2.2 to bypass the app integrity check.

4.5 Summary

We have shown that the apps and libraries use diverse Android APIs and system calls to detect evidence for tampering, but they mainly focus on only a few characteristics of tampered systems and apps. This characteristic makes it easy to bypass the self-defense mechanisms and we have shown that the strategies we put together for bypass attacks are effective against these apps and libraries.

Chapter 5

MERCIDroid Effectiveness and Limitation Evaluation

This section describes the effectiveness and limitations of MERCIDroid.

5.1 Effectiveness of MERCIDroid

We investigate the effectiveness of MERCIDroid in two aspects. We first evaluate how many apps MERCIDroid can generate SDMGraphs for. SDMGraphs are generated when an SDMGraph contains both an environment information provider(s) and an execution terminator. MERCIDroid constructs SDMGraphs for 67 out of 73 apps that check device rooting and 39 out of 44 apps that check app integrity. Overall, MERCIDroid constructs SDMGraphs for 130 out of 140 self-defense mechanisms, which means that at least 130 self-defense mechanisms follow the self-defense mechanism structure described in Section 2.

We also evaluate the effectiveness of our tool in narrowing the scope of the methods we should analyze. For each self-defense mechanism for which we can generate an SDMGraph, we count the number of the methods defined in the app in the SDMGraph. We compare this number with the total number of methods in the execution trace of each self-defense mechanism. On average, the number of methods in each app is approximately 30,894. The number of methods that are called at least once in the execution trace of running a self-defense mechanism for analysis is approximately 1,079, except Android APIs. Finally, the number of methods in the SDMGraph for self-defense mechanisms is around 40. This means that we

need to consider only 3.7% of the methods in each execution trace and 0.13% of the methods in each app to analyze how robust the self-defense mechanisms are. The SDMGraphs for 130 self-defense mechanisms show that the control flows of 88 of them contain indirect relationships among threads and Android components. Without connecting them, we may only be able to analyze 32% of the self-defense mechanisms.

5.2 Limitations of MERCIDroid

While analyzing the target apps, we found that MERCIDroid is unable to identify the self-defense mechanisms in some cases. First, MERCIDroid cannot identify self-defense mechanisms that do not follow the design pattern described in Section 2 (Limit_Cases 1, 2). Second, MERCIDroid has some implementation limitations (Limit_Cases 1, 3, 4). Despite these limitations, MERCIDroid is able to identify about 92.2% of the self-defense mechanisms, as described above. Tables 4.1a and 4.1b show the number of apps included in Limit_Cases 1 to 3.

Limit_Case_1. An environment checking and an execution termination are called from different entry points. MERCIDroid cannot connect an environment checking and an execution termination when they are called from different entry points. In Android, every app component (e.g., Activity, Service, ContentProvider, BroadcastReceiver) can be an entry point to be used in the Android system [25]. Therefore, an environment checking and an execution termination can be called from different entry points. Then, MERCIDroid cannot connect the parts because their common ancestor is located in the Android system, where MERCIDroid is unable to trace the control flow. Therefore, the components exchange checking results through a class' member variable. We found three apps that do this. To overcome this limitation, MERCIDroid should trace the control flow between an app and the Android system.

Limit_Case_2. An app uses WebView. MERCIDroid cannot analyze three apps that use *WebView* to provide the apps' services. The apps display an execution termination using a

website instead of the Android APIs mentioned in Table 2.3. Therefore, MERCIDroid cannot trace the execution termination part.

Limit_Case_3. A native method uses multi-threads. MERCIDroid can trace multi-threads at the Java level but not at the native level. If we want to trace a new thread executed by the target thread, we should use *strace* with the *-f* option. However, in MERCIDroid, the option makes *strace* sleep. Therefore, MERCIDroid cannot trace system calls from the new thread. This situation arose in five of the apps selected for analysis. We can solve this limitation by making MERCIDroid trace the native level inside a process instead of using an external process like *strace*.

Limit_Case_4. The required Android platform version is higher than MERCIDroid's based Android version. MERCIDroid is based on Android 4.4, the most widely used version [26] and the latest version that uses Dalvik VM as the Android Runtime. Currently, only one of our target apps requires the higher version of the Android platform; therefore, we cannot analyze this app. If the number of apps that require the higher version of Android increases, the coverage of MERCIDroid would decrease. We can solve this limitation by porting MERCIDroid to a higher version of the Android platform that uses ART instead of Dalvik VM as the Android Runtime.

Chapter 6

Discussion

It is security critical to ensure that financial Android apps only run on a non-rooted platform. With the lack of a platform-level support, we find that many financial apps employ various checks themselves, only to be proven ineffective by our analysis. We first discussed a few existing platform-level approaches that can benefit apps which need to ensure platform integrity before executing security critical functions. We then discussed missing pieces and challenges making these approaches impractical.

It is not uncommon for users to root their device. They may decide to install benign apps that require rooting without fully understanding security risks. They may get tricked into installing a root-kit that results in a compromised kernel. To help those users avoid further harm, it is desirable for financial apps to have the capability to abort security sensitive functions if the kernel is deemed compromised. However, since an app runs under the control of the kernel, an app alone is fundamentally insufficient to determine the integrity of the kernel. One option is to leverage a hardware capability called remote attestation introduced by Trusted Computing Group [27]. Nauman et al. [28] discuss ways that an Android platform can integrate remote attestation. If a trusted third-party service validates the attestation value, the service end involved with an app can determine whether to continue the transaction from a particular device. This approach, however, requires hardware changes, new services for validating attestation, and new protocols to orchestrate a somewhat complicated flow between a device, a financial app, a service, and the server end of the financial app.

Another approach is adding a barrier to prevent tampering of the boot chain and critical kernel components. Verified boot, also known as trusted boot or secure boot, is a technique that ensures that only authorized boot loaders and kernel modules are loaded into a device [29, 30]. Android 4.4 and later support verified boot for device manufacturers [31]. Google provides SafetyNet API [32] as part of Google Play services for apps to query whether the device running the app has passed the Android compatibility testing. Although SafetyNet seems to collect various information, fundamentally, it shares the same limitation with other cases shown in this paper as long as a compromised kernel finds ways to present fake data that make it appear unchanged when probed by SafetyNet.

Chapter 7

Related Work

Financial security. Much research has been done to diagnose the security-level of financial services accessed on various platforms such as mobile apps and web browsers. Recent studies investigated the security of branchless banking apps, an emerging financial service [33, 34]. Joel et al. showed that security images, which prevent phishing attacks on online banking systems, are ineffective due to user negligence [35]. To improve the security of financial services, many studies have proposed more practical solutions. Several works propose a multi-factor authentication scheme such as a one-time password [36, 37], or a location-based authentication solution [3, 38]. While the existing works analyze the already well-known security factors, our work focuses on the self-defense mechanisms of mobile financial apps, of which a very little is known.

Android app security. Our research applies to the security of Android apps, in general, which other studies are also interested in. Enck et al. analyzed various Android apps and uncovered several pervasive misuses of personal information, as well as instances of deep penetration from advertising networks [39]. Many researchers have examined Android interactions and identified security risks within permission systems and communication systems [40–42]. Several other studies have investigated SSL/TLS security, or the lack thereof, in Android apps [5–7]. Egele et al. studied the misuse of cryptographic APIs, which secure data such as passwords and personal information [43]. Taint tracking is one of the famous methodologies for Android security research, which taints important information and tracks

it. A number of previous studies use the taint tracking methodology to trace sensitive data or analyze malware [18, 44–46]. Much research focuses on securing the user input, a primary source of privacy-sensitive user data. Chen et al. studied data leakage in a third-party input method editor (IME) app, which can be configured as a system keyboard [11]. UIPicker [47] and SUPOR [48] are static analysis tools that automatically identify sensitive information among input data entered via the UI. Our research analyzed how mobile financial apps protect themselves from attacks using root permission or app tampering, and how we can bypass these checks, which is a security aspect missed out in other research.

Tool for tracking control flow. Our research also contributes to the creation of a tool to track the control flow of Android apps. Cao et al. [49] propose EdgeMiner to detect indirect control flow transitions, such as registering and executing callback methods, in static analysis. However, EdgeMiner's limitation is that it cannot use information which can be acquired only during runtime. Compac [19] and the Method Trace function in Android Monitor [20] both present an idea for modifying the Dalvik VM to record method calls, but they do not consider indirect control flows. By combining their strengths and adding in new findings, MERCIDroid overcomes their shortages, tracking the indirect relationships between threads or components through the runtime information.

Chapter 8

Conclusion

In this paper, we presented an analysis of 76 Android financial apps to investigate their self-defense mechanisms, the extra security measures that aim to protect the apps. To analyze self-defense mechanisms, we built MERCIDroid, an enhanced Android platform that traces the control flow within a thread, across threads, and across Android components. MERCIDroid constructs a minimal control flow graph that joins checking and termination. Using MERCIDroid, we have shown that we can locate self-defense mechanisms efficiently. Furthermore, we have shown that self-defense mechanisms are easy to bypass by rewriting a small portion of app code in many cases. Thus, self-defense mechanisms employed in Android financial apps are not effective. Our work proves the need for financial apps to employ secure ways that ensure platform integrity.

Responsible Disclosure

On October 31, 2016 we shared our results (non-anonymized), including all the technical details, with the organizations that oversee the security issues of software developed in Republic of Korea. To protect affected apps and libraries, we anonymize their names in this paper.

Bibliography

[1] N. Bose, "Retailer-backed mobile wallet to rival Apple Pay set for test." http://www.reuters.com/article/2015/08/12/us-currentc-mobile-payment-idUSKCN0QH1RY20150812.

[2] "FRB: CM: 2016 Introduction." http://www.federalreserve.gov/econresdata/mobile-devices/2016-Introduction.htm.

[3] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun, "Smartphones as practical and secure location verification tokens for payments," in *ISOC NDSS*, 2014.

[4] "How secure the mobile payments are?." https://storify.com/williamjohn005/how-secure-the-mobile-payments-are.

[5] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," in *ACM CCS*, pp. 50–61, 2012.

[6] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *ACM CCS*, pp. 38–49, 2012.

[7] L. Onwuzurike and E. De Cristofaro, "Danger is My Middle Name: Experimenting with SSL Vulnerabilities in Android Apps," *arXiv preprint arXiv:1505.00589*, 2015.

[8] C. Mulliner, W. Robertson, and E. Kirda, "VirtualSwindle: an automated attack against in-app billing on android," in *ACM ASIA CCS*, pp. 459–470, 2014.

[9] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE Security and Privacy (Oakland)*, pp. 95–109, 2012.

[10] E. Miluzzo, A. Varshavsky, S. Balakrishnan, and R. R. Choudhury, "Tapprints: your finger taps have fingerprints," in *ACM MobiSys*, pp. 323–336, 2012.

[11] J. Chen, H. Chen, E. Bauman, Z. Lin, B. Zang, and H. Guan, "You Shouldn't Collect My Secrets: Thwarting Sensitive Keystroke Leakage in Mobile IME Apps," in *USENIX Security*, 2015.

[12] P. Stirparo, I. N. Fovino, M. Taddeo, and I. Kounelis, "In-memory credentials robbery on android phones," in *IEEE WorldCIS*, pp. 88–93, 2013.

[13] "Shrink Your Code and Resources — Android Studio." `https://developer.android.com/studio/build/shrink-code.html`.

[14] "Java Decompiler." `http://jd.benow.ca/`.

[15] "Android Developers." `https://developer.android.com`.

[16] E. Gruber, "Android Root Detection Techniques." `https://blog.netspi.com/android-root-detection-techniques/`.

[17] "android - Determine if running on a rooted device - StackOverflow." `http://stackoverflow.com/questions/1101380/determine-if-running-on-a-rooted-device`.

[18] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM TOCS*, vol. 32, no. 2, p. 5, 2014.

[19] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du, "Compac: Enforce component-level access control in Android," in *ACM CODASPY*, pp. 25–36, 2014.

[20] Google, “Method Trace — Android Developer.” http://developer.android.com/intl/ko/tools/help/am-methodtrace.html.

[21] “Graphviz — Graphviz - Graph Visualization Software.” http://www.graphviz.org/.

[22] “smali - An assembler/disassembler for Android’s dex format.” https://github.com/JesusFreke/smali.

[23] “IDA Debugger.” https://www.hex-rays.com/.

[24] “Linux Manual Page.” http://man7.org/.

[25] “Application Fundamentals — Android Developers.” https://developer.android.com/guide/components/fundamentals.html.

[26] “Dashboard — Android Developers.” http://developer.android.com/intl/ko/about/dashboards/index.html.

[27] “Trusted Computing Group — Open Standards for Security Technology.” http://www.trustedcomputinggroup.org/.

[28] M. Nauman, S. Khan, X. Zhang, and J.-P. Seifert, “Beyond kernel-level integrity measurement: enabling remote attestation for the android platform,” in *Trust and Trustworthy Computing*, pp. 1–15, Springer, 2010.

[29] “Verified Boot - The Chromium Projects.” http://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot.

[30] “U-Boot Verified Boot.” http://git.denx.de/cgi-bin/gitweb.cgi?p=u-boot.git;a=blob;f=doc/uImage.FIT/verified-boot.txt.

[31] “Verified Boot — Android Open Source Project.” https://source.android.com/security/verifiedboot/.

[32] "Checking Device Compatibility with SafetyNet — Android Developers." http://developer.android.com/intl/ko/training/safetynet/index.html.

[33] B. Reaves, N. Scaife, A. Bates, P. Traynor, and K. Butler, "Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World," in *USENIX Security*, 2015.

[34] A. Harris, S. Goodman, and P. Traynor, "Privacy and security concerns associated with mobile money applications in Africa," *Wash. JL Tech. & Arts*, vol. 8, p. 245, 2012.

[35] J. Lee, L. Bauer, and M. L. Mazurek, "The Effectiveness of Security Images in Internet Banking," *Internet Computing, IEEE*, vol. 19, no. 1, pp. 54–62, 2015.

[36] RSA, "RSA SecurID." http://www.emc.com/security/rsa-securid/index.htm.

[37] PayPal, "PayPal Security Key." https://www.paypal.com/webapps/mpp/security/security-protections.

[38] F. S. Park, C. Gangakhedkar, and P. Traynor, "Leveraging cellular infrastructure to improve fraud prevention," in *IEEE ACSAC*, pp. 350–359, 2009.

[39] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *USENIX Security*, vol. 2, p. 2, 2011.

[40] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *ACM MobiSys*, pp. 239–252, 2011.

[41] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *ACM CCS*, pp. 627–638, 2011.

[42] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission Re-Delegation: Attacks and Defenses," in *USENIX Security*, 2011.

[43] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *ACM CCS*, pp. 73–84, 2013.

[44] L.-K. Yan and H. Yin, "DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis," in *USENIX Security*, pp. 569–584, 2012.

[45] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN*, pp. 259–269, 2014.

[46] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *ACM SIGPLAN*, pp. 1–6, 2014.

[47] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "UIPicker: User-Input Privacy Identification in Mobile Applications," in *USENIX Security*, 2015.

[48] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps," in *USENIX Security*, 2015.

[49] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework," in *ISOC NDSS*, 2015.

국문초록

많은 모바일 금융 애플리케이션들은 사용자의 개인 정보를 보호하기 위해 실행 시간 환경을 검사하고, 실행 환경이 변조된 것으로 판단되는 경우 자신의 실행을 종료한다. 이 논문은 76개의 대한민국에서 대중적인 안드로이드 금융 앱에서 사용되는 이와 같은 자기방어 메커니즘에 대해 연구하였다. 그 과정에서 우리는 앱들의 매우 난독화된 코드와 복잡하고 비전통적인 제어 흐름 때문에 기존의 도구들로는 앱을 분석하는 데 한계가 있음을 파악하였다. 우리는 이를 실행 시간에 저장된 대상 앱의 자세한 실행 흐름으로부터 자기방어 메커니즘이 포함된 콜 그래프를 추출하는 방식으로 해결하였다. 이 콜 그래프를 생성하기 위해, 우리는 무결성 검사를 위해 사용된 안드로이드 API와 시스템 콜들, 그리고 경고 창을 띄우거나 앱을 종료하기 위해 사용된 안드로이드 API와 시스템 콜들 간의 인과관계를 이용하였다. 76개 앱에 대한 우리의 분석은 우리가 일단 인과관계 그래프를 얻기만 하면 대부분의 자기방어 메커니즘을 우회할 때 수정해야 하는 메소드를 정확히 찾아낼 수 있음을 보여주었다. 우리는 73개의 플랫폼 무결성을 검사하는 앱 중 67개, 44개의 앱 바이너리의 무결성을 검사하는 앱 중 39개를 우회하는 데 성공함으로써 앱 수준에서의 무결성 검사의 비효율성을 증명하였다. 우리는 또한 앞서 언급된 앱들이 가장 많이 사용하는 5개 라이브러리에 대해 면밀하게 분석함으로써 그들의 자기방어 메커니즘과 그들의 취약성을 보여주었다. 모바일 금융 앱들은 절대로 변조된 실행 환경에서는 실행되면 안 되기에, 우리의 연구 결과는 무결성 검사를 위한 플랫폼 수준에서의 솔루션의 필요성을 보여주었다.

주요어: 애플리케이션 보안, 안드로이드, 역공학
학번: 2015-21236