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

Taehun Kim
Seoul National University
th_kim@snu.ac.kr

Hyeonmin Ha
Seoul National University
hyeonmin.ha@snu.ac.kr

Seoyoon Choi
SAP Labs, Korea
seoyoon.choi@sap.com

Jaeyeon Jung
Samsung Electronics
jyjung@gmail.com

Byung-Gon Chun [*]
Seoul National University
bgchun@snu.ac.kr

ABSTRACT

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 the Republic of Korea. Our investigation 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 the call graph, we identify the causality between the system APIs (Android APIs and system calls) used to check device rooting and app integrity, and those used to stop an app's execution. Our analysis of 76 apps shows that we can pinpoint methods to bypass a self-defense mechanism using a causality graph in most cases. We successfully bypassed self-defense mechanisms in 67 out of 73 apps that check device rooting and 39 out of 44 apps that check app integrity. While analyzing the self-defense mechanisms, we found that many apps rely on third-party security libraries for their self-defense mechanisms. Thus we present in-depth studies of the top five security libraries. Our results demonstrate the necessity of a platform-level solution for integrity checks.

CCS Concepts

•**Security and privacy → Software and application security;** *Mobile and wireless security; Software reverse engineering;*

Keywords

Application Security; Android; Reverse Engineering

[*]Corresponding author.

ASIA CCS '17, April 02 - 06, 2017, Abu Dhabi, United Arab Emirates
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4944-4/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3052973.3053018

1. INTRODUCTION

Mobile financial applications (hereafter referred to as *apps*) are gaining popularity, such as banking apps, retailer apps (e.g., CurrentC [22]), 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 [8].

To protect users of financial services, mobile financial apps use an extra layer of security, such as two-factor authentication [40], or one-time passwords [10]. 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 [41], app-repackaging attacks [52], and memory-dumping attacks [49]. Therefore, apps must check the integrity of the platform to make other security measures effective.

In principle, there is no bullet-proof solution that assists 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 obtain (and how do they collect it) to determine whether a device has been rooted or if the app's binary has been tampered with? How can we identify self-defense mechanisms precisely?
- Once self-defense mechanisms are identified, what are the steps needed to bypass them in order to continue executing the app as if the self-defense mechanism 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 [15]. 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 implemented MEthod Recorder with Connecting Indirect relations (MERCIDroid), an enhanced Android platform that tracks an app's control flow. To get the call graph, the tool first records a detailed runtime trace of a target app's execution, including the indirect relationships between threads and components. From the recorded data, the tool finds the causality between the *environment investigation*, which checks device rooting and app tampering, and the *execution termination*, which blocks an app's execution, by identifying the Android APIs and the system calls that each part uses. By finding the causal connection, MERCIDroid can pinpoint the self-defense mechanisms among the thousands of methods in a target app. 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 effectiveness of self-defense mechanisms adopted in the apps. For the analysis, we selected 200 apps randomly from top 400 apps in the Finance category of Google Play available in the Republic of Korea, as of January 2016. Of the 200 apps, we found that 73 apps perform a device rooting check, and 44 apps perform an app integrity check. Based on the understanding of various self-defense mechanisms used in these apps, we constructed strategies for bypass attacks: we listed techniques to bypass self-defense mechanisms, and first tried to apply the easiest one per each self-defense mechanism. If the technique did not work, we tried more difficult techniques. Following these strategies, we 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 the apps look at only a limited set of characteristics of rooted platforms and tampered apps, such as the existence of files related to device rooting or the hash value of app package files. In addition, once self-defense mechanisms are identified, bypassing them only requires modifying a few lines of bytecode or native code. In the analysis, we found that many apps use several third-party security libraries for their self-defense mechanisms, so the apps share same self-defense mechanism codes. To shed insights on the effectiveness of these security libraries, we conducted in-depth case studies of five popular security libraries used in our studied apps. We found that most libraries share the same weaknesses. Some libraries use techniques that other self-defense mechanisms do not use, such as checking a system process's user ID and investigating a system property to check device rooting. Nevertheless, the libraries cannot prevent themselves from being bypassed through app rewriting.

Our contributions are threefold:

- 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, including indirect method relationships such as inter-thread and inter-component communications, and constructs runtime call graphs. MERCIDroid enables us to locate self-defense mechanisms precisely.
- Based on the analysis of the self-defense mechanisms, we show that the apps and third-party libraries try to detect malicious system manipulation attempts using various techniques and system functions. However, they mostly leverage only a few characteristics of rooted systems and tampered apps.
- We demonstrate that our bypass attacks are effective in detouring most of the self-defense mechanisms observed above with simple code modifications, which proves that self-defense mechanisms are not effective as it is. 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.

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 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.

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 8).

Step 2: Displaying an alert dialog. `PopupDialog.showAlert()` calls `PopupDialog.show()`, and it calls `PopupDialog.alertDialogSetter()`. (Lines 7, 29, and 34, respectively). `alertDialogSetter()` finally sets the *AlertDialog* that warns the user before aborting the app (Lines 40–42).

Figure 1 shows high-level structure of self-defense mechanisms observed from analyzing a large number of Android financial apps. Listing 1 fits the structure. First, a method, which we label a common ancestor, calls an environment investigation investigate the app binary and the platform on which the app is installed (Figure 1(1)). This environment investigation then calls **environment information providers**, which provide the system environment variables necessary for the check. Based on the values returned, the environment investigation decides the app's integrity and/or device rooting and returns the result to the common ancestor method (Figure 1(2)). If the execution environment is deemed unsafe, the common ancestor method calls the code to terminate the app (Figure 1(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 investigation, and the execution termination, respectively.

```
//Common ancestor
public class AppCActivity extends Activity {
  public void onResume() {
    int i = Lib0.check(this);
    AppCLogger.e("AppCActivity", i);
    if (i > 0)
      PopupDialog.showAlert(this);
    else if(SecureManager.checkRooting())
      PopupDialog.showAlert(this);
  }
}

//Environment investigation
public class Lib0 {
  private static int check(Context paramContext) {
    return Lib0Native();
  }

  private static native int Lib0Native() ;
}

//Execution termination
public class PopupDialog {
  Dialog dialog;

  public static PopupDialog showAlert(Activity
      paramActivity) {
    PopupDialog alertDialog = new
        PopupDialog(paramActivity);
    alertDialog.setType(100);
    return alertDialog.show();
  }

  public void PopupDialog show() {
    if(getType() == 100) {
      this.dialog = alertDialogSetter();
      this.dialog.show();
    }
  }

  private AlertDialog alertDialogSetter() {
    AlertDialog.Builder localBuilder = new
        AlertDialog.Builder();
    localBuilder.setTitle(getTitle());
    localBuilder.setMessage(getMessage());
    return localBuilder.create();
  }
}
```

Listing 1: AppC's device rooting check methods (decompiled with jd-gui [12]): We simplified the code for readability.

The typical structure of self-defense mechanisms assumes that environment investigation and execution termination are connected by a common ancestor. We found that 93% of the self-defense mechanisms in our analysis follow this structure (see Section 4). In the following, we describe the details of the environment investigation and execution termination. We then explain challenges to identifying self-defense mechanisms.

2.1 Environment investigation

The role of environment investigation is to obtain information about the execution environment (e.g., the existence of a specific file, package information), and decide whether the environment is unsafe. For example, an app checks for device rooting by scanning for the existence of the *su* binary or permission management apps such as *SuperSu*. 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 investigation calls environment information provider(s). Thus, we can use the environment information provider(s) to locate the environment investigation.

The environment information providers can be categorized into

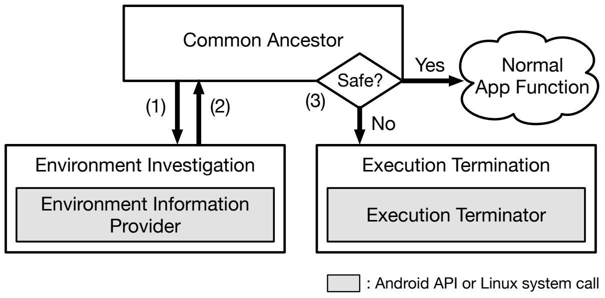

Figure 1: Control flow of a self-defense mechanism.

two types: Android APIs and Linux system calls. An app may use specific Android APIs and system calls (mostly through C wrapper functions) to retrieve information required to investigate the environment, such as file paths of binaries or app packages. Based on the survey of known techniques [1, 35], we selected the Android APIs and system calls shown in Tables 1 and 2 as environment information providers. However, because these APIs and system calls can be called from parts other than self-defense mechanisms, we only consider them as environment information providers when additional conditions are satisfied, described in the tables.

2.2 Execution termination

If an environment investigation part detects a tampered environment, the common ancestor executes an 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. The warning message contains specific keywords such as "rooted" and "forged". To reduce false negatives, we consider various keywords in English and Korean. We also cover app termination that does not display an alert dialog. For example, some apps directly kill the running processes of themselves using Android APIs or `exit()` system call. Some other apps launch the application uninstaller to uninstall themselves from the device. Table 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

Although self-defense mechanisms mostly follow the main execution path as shown in Figure 1, many self-defense mechanisms contain extra steps that complicate the analysis. Therefore, even if the environment investigation 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 investigation 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 1 and 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.

Finding and modifying only the execution termination part is not sufficient to bypass the self-defense mechanism because the branch point between the execution termination part and the normal execution path is located outside of the execution termination part.

Type	Environment information providers	How to check the device rooting using the API [2]	Investigating conditions
Android API	`Runtime.exec()`	Run the *su* command	The command name is "su"
	`PackageManager.getPackageInfo()`	Find whether a root-related app is installed	The package name is related to device rooting (e.g., eu.chainfire.supersu)
	`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
System call	`stat()`		
	`access()`		
	`open()`		
		Check the *adbd* process' user ID	The file name is "/proc/[pid]/status"
	`opendir(), readdir(), chdir()`	Examine every file and directory	-

Table 1: Environment information providers for device rooting checks.

Type	Environment information providers	How to check the app tampering using the API [2]	Investigating 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: Environment information providers for the app integrity checks.

Since the branch point lies in the common ancestor, just skipping execution terminators does not make the app enter the normal control flow (Figure 1(3)). Therefore, we should modify the branch point or the checking part in order to bypass the self-defense mechanisms.

In addition to the above difficulties to locate a self-defense mechanism, apps often employ extra steps beyond a single environment investigation and a standard execution termination as described below. First, 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 1 (Figure 2(i)). These steps add more method calls between environment investigation and execution termination, making it difficult to find causality between them. Second, the checking part may communicate with an external server for the integrity check (Figure 2(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. Third, a common ancestor sometimes calls more than one environment investigation as shown in Figure 2(iii). The common ancestor then gathers results from the environment investigations and makes a decision. For example, Listing 1 shows that `AppCActivity.onResume()` calls two environment investigations: `Lib0.check()` and `SecureManager.checkRooting()`. These extra tasks make the analysis of a self-defense mechanism complicated. In the next section, we describe how we overcome these difficulties by exploiting the causality between the environment investigation and the execution termination.

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.

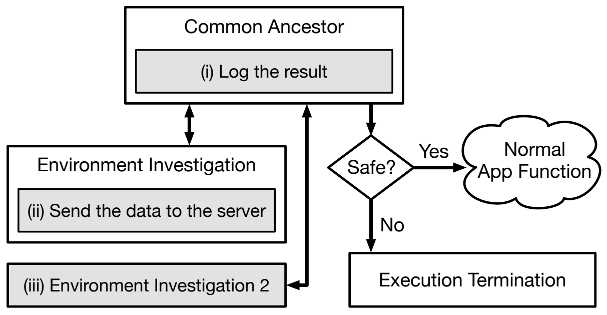
Common Ancestor
(i) Log the result
Environment Investigation
(ii) Send the data to the server
(iii) Environment Investigation 2
Safe?
Yes
No
Normal App Function
Execution Termination

Figure 2: Control flow of a self-defense mechanism with optional, additional tasks. The tasks are presented as gray boxes.

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 of locating self-defense mechanisms is using the causality between the environment information providers 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

Type	Execution terminators	How to use to terminate an app [2]	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 3: Execution terminators for self-defense mechanisms. `System.exit()` and `exit()` do not have a terminating condition because they are only used to kill an app itself.

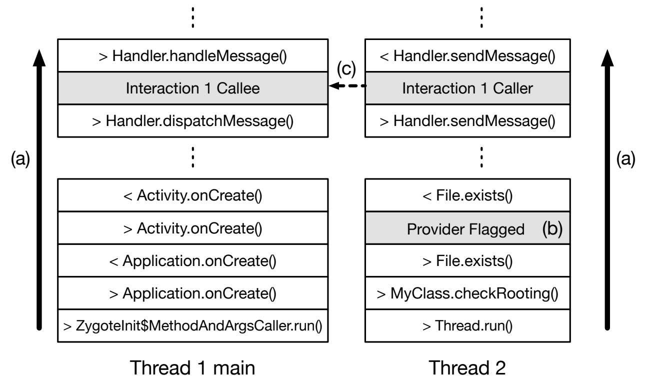

Figure 3: Recording of method invocations in each thread. Each method call/return is stored from the bottom to top (a). An environment information provider that meets the condition is flagged (b). The indirect caller and callee are connected through an interaction ID (c).

the execution terminator, and locate the common ancestor that is closest to the execution terminator.

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. In this paper, we focus on using MERCIDroid to analyze financial app's self-defense mechanisms, but MERCIDroid can also be used for other purposes that can benefit from identifying causality between the system APIs.

3.2.1 Recording method calls at runtime

To record method calls of an app at runtime, we modified Android 4.4.4. We focused on Dalvik Virtual Machine (VM), Android Runtime that Android 4.4.4 uses because Android 4.4.4 is still the most widely used version [7]. Higher versions of Android use Android Runtime (ART), replacing Dalvik VM. However, the underlying runtime does not affect our analysis. Since ART is compatible with Dalvik [5], most Android apps work on both runtimes. We had only one financial app that requires version higher than 4.4.

The modified system records method calls and indirect relationships between threads and components as described in Figure 3.

Recording method calls per thread. Using a similar approach as in Compac [50] and the Method Trace function in Android Monitor [34], we modified the Dalvik 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 allocated 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(a).

Flagging relevant environment information provider calls. While recording method calls, MERCIDroid flags the calls for the environment information providers, as described in Tables 1 and 2. MERCIDroid uses runtime arguments to check the investigating 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(b). Also, to flag environment information providers whose type is a system call, MERCIDroid executes *strace* and parses the result.

Finding 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)` [2]. 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.

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 [2]. Figure 3(c) illustrates how this added ID can be used to link the indirect method calls. Appendix A describes the list of indirect relationships in more detail.

Storing method call records. When an app is stopped by an execution terminator shown in Table 3, 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 a call graph

We implemented a call graph generator which parses the records, constructs a call graph including the aforementioned indirect relationships. Once a call graph is constructed, the generator extracts a self-defense mechanism graph (SDMGraph), which contains only the methods relevant for identifying the self-defense mechanisms. The root node of the SDMGraph is the common ancestor of the execution terminator and the environment information provider(s). 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. We describe the construction process of an SDMGraph in more detail in Appendix B.

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 instrumented Intent and Parcel to trace between two processes.

To capture inter-process communications, we parse collected method calls from all the processes related to self-defense mechanisms. 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.

4. APP ANALYSIS & BYPASS ATTACKS

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

4.1 App Selection

We started with the top 400 apps in the “Finance, Free” category of Google Play in the Republic of Korea, as of January 2016. To cover a diverse set of apps from popular to infrequently used apps, we randomly selected 200 apps from the 400 apps instead of selecting top 200 apps. Note that analyzing financial apps requires setting up credit or bank accounts, thus we had to limit the number of apps we analyzed. To find apps that contain self-defense mechanisms, we went through the following steps. First, to find apps that check device rooting, we installed each app on a rooted device and executed it. To find apps that perform an app binary integrity check, we disassembled the app binary, added an empty class to the code, reassembled a DEX file of the app, and executed the modified app binary. 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’s execution was blocked, we concluded that the app performs a self-defense mechanism and added it to the list of apps to analyze.

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 to run with another app; (b) 14 malfunctioning apps: these apps crash after app repackaging upon launching; (c) 1 app that could not 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.

We found 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 analyze and bypass the self-defense mechanisms.

Groups		**# of apps**
SDMGraphs generated: 67	R_Group_API	14
	R_Group_Native_Predictable_Return	57
	R_Group_Native_Unpredictable_Return	6
SDMGraphs not generated: 6	Limit_Case_2	1
	Limit_Case_3	5

(a) Device rooting checks

Groups		**# of apps**
SDMGraphs generated: 39	I_Group_Predictable_Return	16
	I_Group_Signature	2
	I_Group_APK_Path	14
	I_Group_APK_Path_Context	10
SDMGraphs not generated: 5	Limit_Case_1	3
	Limit_Case_2	2

(b) App integrity checks

Table 4: The number of apps in each group for the device rooting checks (73 apps in total) and the app integrity checks (44 apps in total). Note that some apps check device rooting and app integrity more than once. We categorize the self-defense mechanisms for which a SDMGraph is generated into X_Group_Y, where X is R for device rooting check and I for app integrity check, and Y is a characteristic of the group. We explain the Limit_Case groups in Section 5.2.

4.2 Traits of Self-defense Mechanisms

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 (See Section 5 for more details). We then successfully bypassed *every* self-defense mechanism for which an SDMGraph has been generated.

We found that the self-defense mechanisms can be grouped by how they use the Android APIs and the system calls described in Tables 1 and 2 and how difficult they were to bypass. Based on the flowcharts described in Appendix C, we ran through each marked method in an SDMGraph, bypassing the self-defense mechanisms by rewriting the app’s Dalvik bytecode using the techniques described in Figure 4 and then grouped the apps. Table 4 shows the number of apps included in each group.

Next, we describe the traits of the self-defense mechanisms in each group and how we bypassed them.

4.2.1 Device rooting checks

R_Group_API.

Apps in this group use Java methods to check device rooting. They use the Android APIs in Table 1 to detect the existence of binaries or apps related to device rooting. For example, some apps check the presence of the “su” binary or command using the

```
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

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

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

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

(f) Generate a FakeContext

Figure 4: Example smali [16] code to bypass self-defense mechanisms. The lines added to bypass the self-defense mechanism are presented in bold.

`File.exists()` or `Runtime.exec()` methods, respectively. Apps also use `PackageManager.getPackageInfo()` to check whether apps related to device rooting (e.g., SuperSU) have been installed.

How to bypass. We modify an argument or the return value of an Android API (Figure 4a or 4b). We change the name of a file the apps try to find (e.g., "su" to "us") or fix the return value to *false*, based on the property of the API.

R_Group_Native_Predictable_Return.

This group contains apps that check device rooting in native libraries. The Java code in the apps includes the declaration of the 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 1 and 2.

How to bypass. We change the native method's declaration to a fake Java method that returns a fixed value (Figure 4c). This is made possible because it is easy to predict the return values of the native methods when a device is not rooted.

R_Group_Native_Unpredictable_Return.

Device rooting checks for apps in this group are similar to those in *R_Group_Native_Predictable_Return*, 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.

How to bypass. In this case, we modify the native code directly using IDA Pro [11] to rewrite bytes in the text section to skip the check.

4.2.2 App integrity checks

I_Group_Predictable_Return.

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

How to bypass. When the marked method is a native method, we change the method's declaration to a Java method that returns a fixed value (Figure 4c). In the case of an Android API, we find the caller and make the method return a fixed value (Figure 4d).

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). The signature of the rewritten app is different from that of the unmodified one. Therefore, examining the signature can be a way to check app integrity.

How to bypass. 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_Path.

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, as shown in Table 2. The apps then read the file using *File*, *RamdomAccessFile*, or *ZipFile*

Name	# of apps that adopt the libraries			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 5: List of libraries that contain self-defense mechanisms.

classes to compute the hash value (see Table 2). The apps send it to an external server to check their tampering.

How to bypass. 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 4e).

I_Group_APK_Path_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_Path*.

How to bypass. 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 4f).

4.3 Third-party Security 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 that produced the library.

Table 5 shows those libraries and the 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 by getting the lists of installed apps and running processes from the system, or they scan by transacting with security apps developed by the same vendor. Also, the libraries support various features, such as code obfuscation, anti-decompiling, 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.

Risk of using security libraries. If many apps share the same libraries, these apps are vulnerable to the same bypass attacks. For example, as shown in Table 5, 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.

4.3.1 Case Studies

Given the popularity of security libraries, we did an in-depth analysis of the 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 methods to check device rooting and app integrity via the system calls described in Tables 1 and 2, we bypassed all checks with the app rewriting techniques described above.

How the libraries 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. However, some libraries use other methods to examine device rooting, such as checking the user ID of a system process and investigating system properties.

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*) using the `stat()` [13] system call. The system call takes the file path as an 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 finds adbd's process ID by scanning the */proc/* directory, which contains numerous subdirectories for running processes named by the process IDs [13]. The library then gets the user ID of the adbd process by reading the */proc/[pid]/status* file via the `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 to check whether private directories for the apps related to device rooting (e.g., *com.marutian.quckunroot*, *com.noshufou.android.su*) are accessible.

Second, the library tries to open the *su* binary and root-related apps' APK files using the `open()` system call. 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 methods to examine the platform's integrity in addition to checking root-related apps and binaries, such as 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 under the root directory recursively except for some unimportant directories, such as */sdcard/* and */mnt/*. To evaluate every file, the library calls `opendir()`, `readdir()`, and `chdir()`. 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.

How the libraries check app integrity.

The libraries use a methodology similar to that described in Section 4.2 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. Even if the libraries are written in native code, they always obtain the APK file path using Android APIs 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, i.e., 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 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* object 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.

How an app finds out whether the self-defense mechanism had been called.

Although the libraries use various methods to check device rooting and app integrity, almost none of the apps or libraries verify after the checks that the checks were performed. 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.

How the libraries terminate an app running in an unsafe environment.

If a self-defense mechanism finds that the device is rooted or the app has been tampered with, the security library stops the app's execution. The security libraries, except Lib3, return the checking result to a Java method instead of directly showing an alert dialog.

Lib3. Lib3 has characteristics different from the other 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 [13].

How the libraries return the result and how we bypassed.

The libraries use diverse techniques to check device rooting and app integrity, and to prevent apps from executing in unsafe environments. However, the techniques do not deviate from those described in Section 4.2, so they cannot prevent attackers from bypassing checks.

Lib0. If the device is not rooted, Lib0's native method returns zero. Otherwise, the method returns -1. We bypass the check by changing the native method's declaration to a Java method that 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 changing the native method's declaration to a Java method that 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. Exceptionally, we modify the library file and the APK file path instead of changing the native method's declaration for AppL.

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.

4.4 Summary

We have shown that the apps and libraries use diverse Android APIs and system calls to detect evidence of tampering, but they mainly focus on only a few characteristics of tampered systems and apps. For example, to check device rooting, apps and libraries try to find binaries and apps related to device rooting using known file paths and app names. To check app integrity, they compute the app package file's hash value and compare it with the correct value. These checks are done in the app itself, which makes it easy to bypass the self-defense mechanisms. We have shown that the strategies we put together for bypass attacks are effective against these apps and libraries. Additionally, we found that some libraries leverage characteristics that other apps or libraries do not consider. For example, Lib0 checks a system process's user ID and Lib4 investigate all files and directories in the system to check device rooting. We also found that one app checks whether the self-defense mechanism has been bypassed, during runtime. However, these attempts are still made in the app, so they cannot completely prevent the bypass attacks. From the findings, we conclude that the currently implemented self-defense mechanisms are ineffective, and thus platform-level verification is required to ensure the integrity of the platform and the apps.

5. MERCIDroid EFFECTIVENESS AND LIMITATIONS

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 the number of apps for which MERCIDroid can generate SDMGraphs. 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 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 the robustness of the self-defense mechanisms. 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 identifying the indirect relationships, we may only be able to analyze 32% of the self-defense mechanisms.

5.2 Limitations of MERCIDroid

MERCIDroid identified more than 92% of self-defense mechanisms in our target apps, but it has a few limitations. As described in Section 2, MERCIDroid relies on some assumptions regarding the typical structure of a self-defense mechanism. For example, we assume that an alert message in an execution termination contains some keywords such as "root" or "forged." We also assume that there is a common ancestor between environment investigation and execution termination. Most self-defense mechanisms can be spotted with such assumptions, but there are some self-defense mechanisms that do not follow the assumptions and thus are not identified by MERCIDroid (Limit_Cases 1, 2). Also, MERCIDroid has some implementation limitations (Limit_Cases 1, 3, 4). Table 4 shows the number of apps included in Limit_Cases 1 to 3.

Limit_Case_1. An environment investigation and an execution termination are called from different entry points. MERCIDroid cannot find a relationship between an environment investigation 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 [4]. Therefore, an environment investigation and an execution termination can be called from different entry points and exchange checking results through a class' member variable. Then, MERCIDroid cannot find their common ancestor because it is located in the Android system process, where MERCIDroid is unable to trace the control flow. We found three apps where this case holds. 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 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. Five apps belong to this category. We can solve this limitation by making MERCIDroid trace the native level inside a process instead of using an external process such as *strace*.

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, which are ineffective as shown by our analysis. We first discuss a few existing platform-level approaches that can benefit apps which need to ensure platform integrity before executing security critical functions. We then discuss the limitations that make 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 platform. To help those users avoid further harm, it is desirable for financial apps to have the capability to abort security sensitive functions if the platform is deemed compromised. However, since an app runs under the control of the platform, an app alone is fundamentally insufficient to determine the integrity of the platform. One option is to leverage a hardware capability called remote attestation introduced by Trusted Computing Group [17]. Nauman et al. [43] 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 system 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 [18, 19]. Android 4.4 and later support verified boot for device manufacturers [20]. However, verified boot is an optional solution for device manufacturers, thus there is no guarantee that a financial app runs on a device that uses verified boot. Google provides SafetyNet API [6] 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 platform finds ways to present fake data that make it appear unchanged when probed by SafetyNet.

7. RELATED WORK

Financial security.

There are several studies related to the security-level of financial services on various platforms such as mobile apps and web browsers. Recent studies investigated the security of branchless banking apps, an emerging financial service [36, 47]. Joel et al. showed that security images, which prevent phishing attacks on online banking systems, are ineffective due to user negligence [39]. 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 [46,48], or a location-based authentication solution [40, 45]. Managing a secret key for a cryptography algorithm is another important issue for financial security. A secret key hard-coded in an app or carelessly managed in the memory can incapacitate the cryptography algorithm no matter how strong the algorithm is. White-box cryptography technology reduces this concern by hiding the secret key in the transformed cryptography algorithm [23]. Recent security solutions for mobile financial apps use the white-box cryptography to protect user's private information the apps hold [3,14]. 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.

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 [29]. Many researchers have examined Android interactions and identified security risks within permission systems and communication systems [26,31,32]. Several other studies have investigated SSL/TLS security, or the lack thereof, in Android apps [30, 33, 44]. Egele et al. studied the misuse of cryptographic APIs, which secure data such as passwords and personal information [27]. Chen et al. studied data leakage in a third-party input method editor (IME) app, which can be configured as a system keyboard [25]. UIPicker [42] and SUPOR [37] 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.

There has been much research on Android security that adopts taint tracking methodologies; taint tracking taints important information and tracks it to trace sensitive data or analyze malware [21, 28, 38, 51]. However, this approach is not effective for analyzing self-defense mechanisms, because it cannot locate an environment investigation and an execution termination at once. A common ancestor normally uses the return value of the environment information provider as a condition for an *if-else* statement, thus the taint tag is not propagated through the statement. To overcome this limitation, the taint tracking system should support the control flow propagation. However, according to the TaintDroid research [28], developing such a mechanism for Android apps is difficult because Dalvik bytecode does not maintain the branch structure. Therefore, we conclude to trace the control flow of an app by enhancing the Android platform instead of using the taint tracking methodology.

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. [24] 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 [50] and the Method Trace function in Android Monitor [34] both present an idea for modifying the Dalvik VM to record method calls, but they do not consider indirect control flows. MERCIDroid overcomes their deficiencies by tracking the indirect relationships between threads or components using the runtime information.

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 an environment investigation and an execution termination. Using MERCIDroid, we have shown that we can locate self-defense mechanisms efficiently. Our analysis of the self-defense mechanisms shows that apps use various techniques to detect that the execution environment has been tampered with. However, the mechanisms are ineffective because they rely on only a few characteristics of the rooted platform and tampered apps and are executed inside the apps. Finally, 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 calls for security mechanisms to ensure platform and app integrity for Android financial apps.

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 the Republic of Korea. To protect affected apps and libraries, we anonymize their names.

Acknowledgment

We thank our shepherd Yubin Xia and the anonymous reviewers for their feedback on this work. This work was supported by BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by National Research Foundation of Korea(NRF) (21A20151113068).

9. REFERENCES

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

[2] Android Developers. https://developer.android.com.

[3] App security for banking & payment apps - Promon SHIELD. https://promon.co/industries/app-security-banking-payment/.

[4] Application Fundamentals | Android Developers. https://developer.android.com/guide/components/fundamentals.html.

[5] ART and Dalvik | Android Open Source Project. https://source.android.com/devices/tech/dalvik/.

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

[7] Dashboard | Android Developers. http://developer.android.com/intl/ko/about/dashboards/index.html.

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

[9] Graphviz | Graphviz - Graph Visualization Software. http://www.graphviz.org/.

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

[11] IDA Debugger. https://www.hex-rays.com/.

[12] Java Decompiler. http://jd.benow.ca/.

[13] Linux Manual Page. http://man7.org/.

[14] Mobile Banking Security, Internet Banking App Security. https://www.whitecryption.com/mobile-banking/.

[15] Shrink Your Code and Resources | Android Studio. https://developer.android.com/studio/build/shrink-code.html.

[16] smali - An assembler/disassembler for Android's dex format. https://github.com/JesusFreke/smali.

[17] Trusted Computing Group | Open Standards for Security Technology. http://www.trustedcomputinggroup.org/.

[18] 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.

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

[20] Verified Boot | Android Open Source Project. https://source.android.com/security/verifiedboot/.

[21] 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*, pages 259–269, 2014.

[22] 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.

[23] W. Brecht. White-box cryptography: hiding keys in software. *NAGRA Kudelski Group*, 2012.

[24] 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.

[25] 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.

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

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

[28] 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*, 32(2):5, 2014.

[29] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security*, volume 2, page 2, 2011.

[30] 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*, pages 50–61, 2012.

[31] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM CCS*, pages 627–638, 2011.

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

[33] 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*, pages 38–49, 2012.

[34] Google. Method Tracer | Android Studio. https://developer.android.com/studio/profile/am-methodtrace.html.

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

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

[37] 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.

[38] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android taint flow analysis for app sets. In *ACM SIGPLAN*, pages 1–6, 2014.

[39] J. Lee, L. Bauer, and M. L. Mazurek. The Effectiveness of Security Images in Internet Banking. *Internet Computing, IEEE*, 19(1):54–62, 2015.

[40] 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.

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

[42] 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.

[43] 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*, pages 1–15. Springer, 2010.

[44] L. Onwuzurike and E. De Cristofaro. Danger is my middle name: experimenting with ssl vulnerabilities in android apps. In *ACM WiSec*, page 15. ACM, 2015.

[45] F. S. Park, C. Gangakhedkar, and P. Traynor. Leveraging cellular infrastructure to improve fraud prevention. In *IEEE ACSAC*, pages 350–359, 2009.

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

[47] 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.

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

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

[50] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du. Compac:

Enforce component-level access control in Android. In *ACM CODASPY*, pages 25–36, 2014.

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

[52] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Security and Privacy (Oakland)*, pages 95–109, 2012.

APPENDIX

A. LIST OF INDIRECT CALLER–CALLEE RELATIONSHIPS

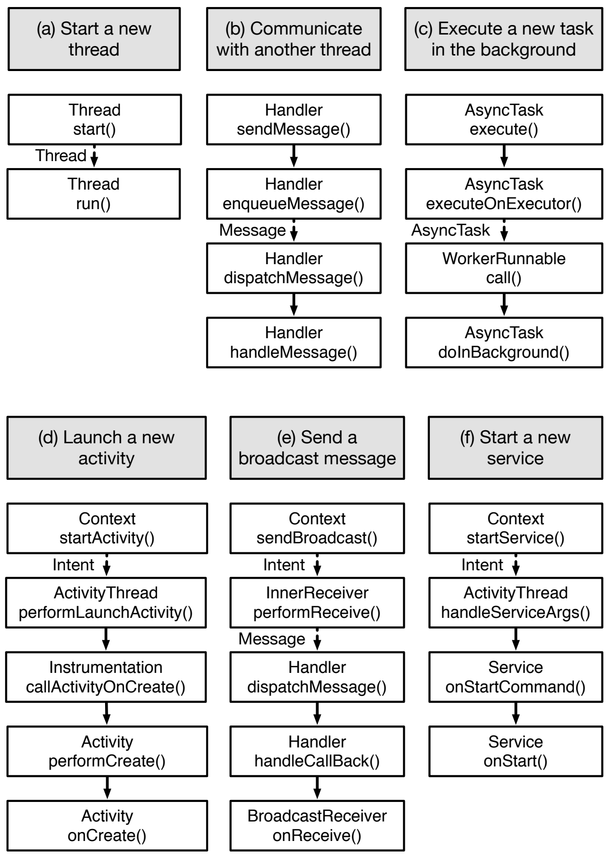
(a) Start a new thread
(b) Communicate with another thread
(c) Execute a new task in the background
Thread start()
Thread
Thread run()
Handler sendMessage()
Handler enqueueMessage()
Message
Handler dispatchMessage()
Handler handleMessage()
AsyncTask execute()
AsyncTask executeOnExecutor()
AsyncTask
WorkerRunnable call()
AsyncTask doInBackground()
(d) Launch a new activity
(e) Send a broadcast message
(f) Start a new service
Context startActivity()
Intent
ActivityThread performLaunchActivity()
Instrumentation callActivityOnCreate()
Activity performCreate()
Activity onCreate()
Context sendBroadcast()
Intent
InnerReceiver performReceive()
Message
Handler dispatchMessage()
Handler handleCallBack()
BroadcastReceiver onReceive()
Context startService()
Intent
ActivityThread handleServiceArgs()
Service onStartCommand()
Service onStart()

Figure 5: Indirect caller–callee relationships.

Figure 5 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.

B. SDMGRAPH CONSTRUCTION PROCESS WITH AN EXAMPLE

In this section, we describe the SDMGraph construction process in detail. To find the common ancestor, the call graph generator described in Section 3.2.2 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* [9].

Figure 6 shows an example SDMGraph of a device rooting check performed by AppZ. The app uses `ProcessManager.exec()` as an environment information provider and `AlertDialog$Builder.setMessage()` as an execution terminator. To find the common ancestor of the two method calls, the script runs the following steps. First, it checks `ProcessManager.exec()`, already flagged at runtime, and its ancestors (Figure 6(a)). Second, it tracks the ancestors of `AlertDialog$Builder.setMessage()` (Figure 6(b)). Finally, it 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 6(c)).

C. FLOWCHARTS FOR BYPASSING SELF-DEFENSE MECHANISMS

Figures 7 and 8 show flowcharts for bypassing device rooting checks and app integrity checks. We constructed the flowcharts based on our sample data and our trial and error experience. We first try an easier technique to bypass an identified self-defense mechanism. If the technique fails, we progressively try more difficult techniques. In each step, we rewrite the app using the techniques described in Section 4.2. If we succeed in bypassing a self-defense mechanism, we place the app in a corresponding success group.

Figure 7 shows the strategies used to bypass device rooting checks. We first identify whether the marked method is an Android API or a native method. If the method is an Android API, we modify the return value or the argument of the method (R_Group_API). Otherwise, we fix the return value of the native method (R_Group_Native_Predictable_Return). If both fail, we patch some bytes in a native library (R_Group_Native_Unpredictable_Return).

Figure 8 shows the steps that must be followed to bypass app integrity checks. Contrary to the device rooting check, when the marked method is an Android API, we first attempt to fix its caller's return value before modifying the API's argument or return value. This is because the Android APIs used by app integrity checks handle an APK file path or an app's signature (see Table 2). These values are app-specific, so we should generate the values for the unmodified apps to modify the values. In many cases, predicting and fixing the caller's return value is easier than handling an APK file path or an app's signature, so we first try the prediction. 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_Path, I_Group_APK_Path_Context) to an unmodified app's signature or APK file path.

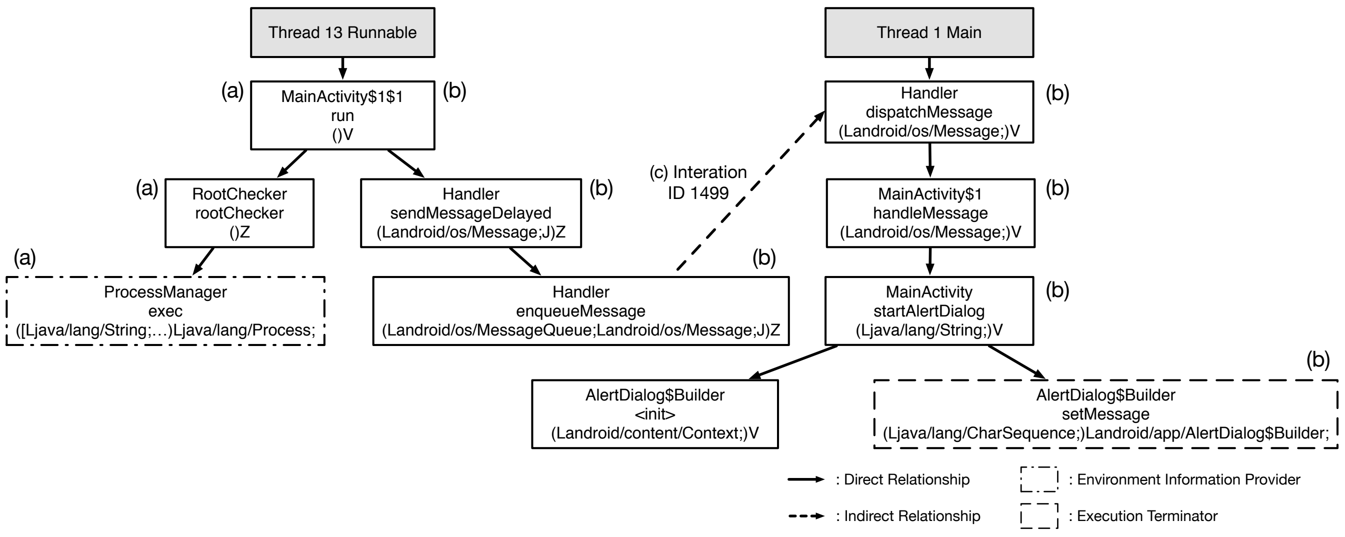

Figure 6: An SDMGraph of the device rooting check of AppZ. We omitted, simplified, and renamed the methods.

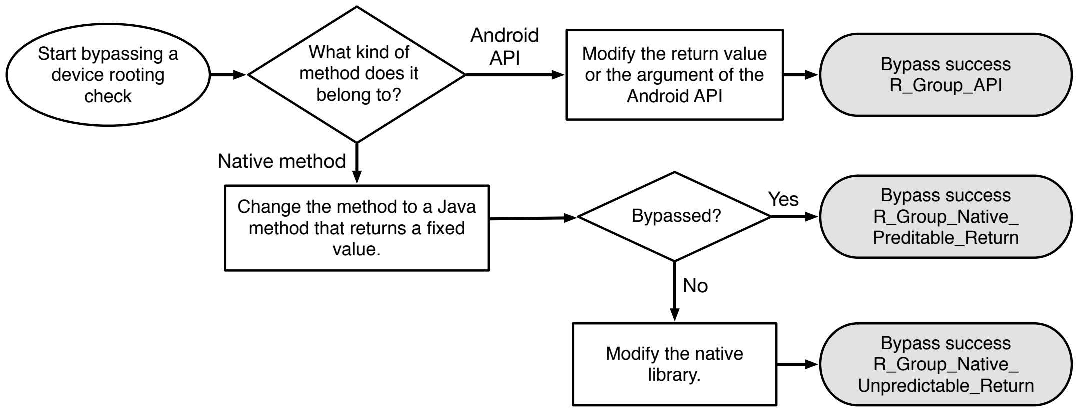
Start bypassing a device rooting check
What kind of method does it belong to?
Android API
Modify the return value or the argument of the Android API
Bypass success R_Group_API
Native method
Change the method to a Java method that returns a fixed value.
Bypassed?
Yes
Bypass success R_Group_Native_ Preditable_Return
No
Modify the native library.
Bypass success R_Group_Native_ Unpredictable_Return

Figure 7: A flowchart for bypassing device rooting checks.

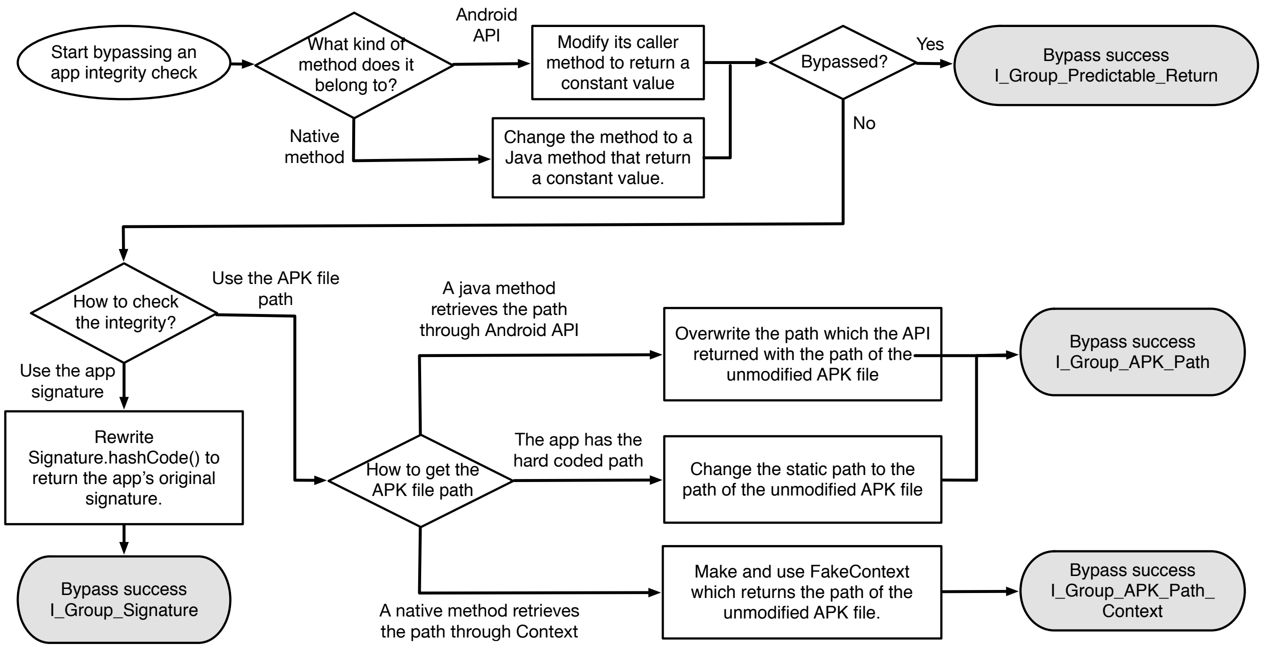
Start bypassing an app integrity check
What kind of method does it belong to?
Android API
Modify its caller method to return a constant value
Native method
Change the method to a Java method that return a constant value.
Bypassed?
Yes
Bypass success I_Group_Predictable_Return
No
How to check the integrity?
Use the APK file path
Use the app signature
Rewrite Signature.hashCode() to return the app's original signature.
Bypass success I_Group_Signature
How to get the APK file path
A java method retrieves the path through Android API
Overwrite the path which the API returned with the path of the unmodified APK file
Bypass success I_Group_APK_Path
The app has the hard coded path
Change the static path to the path of the unmodified APK file
A native method retrieves the path through Context
Make and use FakeContext which returns the path of the unmodified APK file.
Bypass success I_Group_APK_Path_ Context

Figure 8: A flowchart for bypassing app integrity checks.