Is Your Android App Insecure? Patching Security Functions With Dynamic Policy Based on a Java Reflection Technique

SUNG-HOON LEE1, SEUNG-HYUN KIM2, JUNG YEON HWANG3, SOOHYUNG KIM2, AND SEUNG-HUN JIN2

1Information Security Engineering, Ph.D. Student, University of Science and Technology, Daejeon 34113, South Korea
2Electronics and Telecommunications Research Institute, Daejeon 34113, South Korea
3Department of Mathematics, Sungshin Women’s University, Seoul 02844, South Korea

Corresponding author: Seung-Hyun Kim (ayo@etri.re.kr)

This work was supported in part by Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.2018-0-01369, Developing blockchain identity management system with implicit augmented authentication and privacy protection for O2O services), and in part by UST Young Scientist Research Program through the University of Science and Technology (No. 2019YS09).

ABSTRACT With the popularization of smart devices, companies are adopting bring-your-own-device or mobile office policies that utilize personal smart devices for work. However, as work data are stored on individual smart devices, critical security threats are emerging, such as the leakage of confidential documents. Enterprises want to address this issue by adapting enterprise mobility management (EMM) solutions. Appwrapping is among the core technologies in EMM solutions, enabling security function insertion or misused code patching without the original application (app) source code. Studies on permission control, misused code patching, security function insertion based on static policies, etc., have been conducted, but there are limitations such as poor user convenience and overhead. In this paper, we propose an AppWrapper toolkit to support dynamic policies. Basically, it can insert security function execution code into apps by using appwrapping technology without the original source code. This code uses Java reflection to invoke security functions dynamically based on preset policies. Accordingly, after the initial appwrapping, the policy can be changed easily. In addition, even when multiple security functions are required, Java reflection can invoke multiple security functions dynamically and simultaneously without conflicting with the existing code. The AppWrapper toolkit also provides a log function to check in real time where the security function is needed. Hence, the policy-setting administrator can check the log in real time and implement the security function where needed. Our experimental results show that this technique improves significantly the efficiency, effectiveness, and convenience of adding security function execution code.

INDEX TERMS Mobile application, mobile computing, security management.

I. INTRODUCTION

Android is an open-source platform used on about 72% of all mobile devices as of December 2019 [1]. Due to the popularization of smartphones, companies are adopting bring-your-own-device (BYOD) policies to improve work efficiency and reduce costs [2]. However, as BYOD practices involve utilizing personal devices for work, security threats such as the leakage of confidential information are emerging [3]–[5]. Companies have addressed security threats with mobile device management (MDM), mobile application management (MAM), mobile content management (MCM), and enterprise mobility management (EMM) solutions. MDM, MAM, and MCM solutions manage devices, applications (apps), and content, respectively, and EMM solves security threats within an enterprise by integrating these solutions and providing various security functions [6].

The underlying technologies of EMM include containers and appwrapping. Appwrapping has many advantages over container technology in the form of virtual sandboxes because it can apply security policies at the app level [7]. Appwrapping technology can be used to patch the vulnerable
code of an app or insert security functions at the bytecode level without the original app source code. Appwrap-
ing, also known as bytecode rewriting [8], involves creating an app by repackaging the bytecode after inserting or patching code from the bytecode obtained by decompiling the app. In Android, the bytecode is small code. Previous studies using this technology to enhance Android app security include research on permission controls for privacy [9]–[11], vulnerable code patching [12], [13], and addition of static policy-based security functions for vulnerable apps [14]. However, these existing studies have the following limitations.

In permission control research [9]–[11], as permission is controlled based on an application programming interface (API), such control occurs whenever the API is called, resulting in user inconvenience and overhead. In code patching studies [12], [13], the misused code has been detected and patched with the appropriate code, but it is not possible to add the security function provided by the EMM solution for apps that require security. When adding a security function to an app based on a static policy [14], only a single security function can be added when an EMM solution needs security. However, there are drawbacks to repackaging the app to change the added security functions, and there are limitations to adding multiple security functions.

There are two methods of adding new security functions to insecure apps for EMM solutions: inserting the security functions directly into the original Android app source code or inserting the security functions into the APK file of the app via bytecode rewriting [15]. In the former case, the original Android source code must be maintained. In addition, if the app is no longer managed, it may be difficult to obtain the original Android app source code. In contrast, bytecode rewriting involves inserting security functions at the bytecode level without using the original Android app source code. This technique enables the insertion, changing, and deleting of the app security functions when its original source code is not properly managed.

When inserting security function-related code into an insecure app, it is difficult to ensure that the app runs smoothly and that there are no conflicts with the existing code [16]. The Android bytecode is Dalvik bytecode. This register-based instruction set allocates an available register range, depending on the number of variables and parameters declared for each method [17]. The register range is divided into 4 bits, 8 bits, and 16 bits [18]. If the allocated register range is exceeded, an error occurs when recompiling.

In this paper, we propose an AppWrapper toolkit that inserts security function execution code on a per-method basis into an insecure app that needs security functions. In addition, the dynamic policy executes and controls the security functions required at the identified method location. The security functions include those provided by EMM solutions. For example, controllable security functions include enhanced user authentication (fingerprinting, facial recognition, etc.), screen capture restrictions, and data usage restrictions. The contributions of the proposed technique are as follows.

- Execution and control of security functions by method level (usability). By inserting security function execution code into the method unit, the security functions are executed and controlled where required.
- Dynamic policy management (efficiency). After inserting the security function execution code and repackaging the app, it is possible to execute and control the security function where required (per method in the activity) according to the dynamic policy. Even when the policy is changed, the new policy can be directly applied by using a Java reflection technique without inserting security execution code and repackaging the app again.
- A user interface is provided (user convenience). Information technology administrators can run apps that require security and check the logs (class and method name) in real time to see where security functions are needed. In addition, the administrators can simply click on the log where it requires security to set a security policy at that location. Administrators can create security policies using the AppWrapper toolkit even without knowledge of bytecode.

The proposed technique can dynamically call security functions using Java reflection, which is a method of dynamically calling classes and methods and has the advantage of being able to call and use classes and methods when they are needed more flexibly than static calling methods that can use only predefined classes in code [19]. Consequently, the proposed method has the following advantages in terms of usability and efficiency compared to the existing static policy-based approaches.

A. CONCISE SECURITY FUNCTION EXECUTION CODE

The security function execution code consists of the minimum code for calling the security function execution library and has only two local variables and one parameter. The security function execution library dynamically calls the security function of the security app that provides the security function using Java reflection. The structure of this code is based on the nature of the smali code, which is Android bytecode. Smali code generates an error at the compile time if the range of registers allocated for each method is exceeded. By simplifying the code, it can be ensured that the range of register areas allocated to the methods in the existing code will not be exceeded. That is, the security function execution code can be inserted into the method level while minimizing conflicts with the existing code. Compared to the static approach [14], our method can provide more than 30% coverage improvement.

B. APPLY SECURITY POLICY SIMPLY BY CHANGING THE POLICY

When changing the policy, the new policy can be applied immediately without adding security function execution code or repackaging the app. The policy contains the security
function classes and methods required to run the security function. Java reflection is used to load the security function classes and methods that are dynamically set in the policy and to execute the security functions as set in the policy. Even if the policy is changed through the Java reflection technique, it can be applied directly to the app.

C. MULTIPLE SECURITY FUNCTIONS

Because security functions can be called and executed dynamically, even if more than one security function is required for each method unit, they can be executed and controlled without conflicting with the existing code. Since the number of local variables and parameters required for execution differs depending on the security function, adding several security function execution codes directly to a method may cause conflict with the existing code due to exceeding register area range, as mentioned above. However, the proposed method dynamically calls the security functions of the app that provides security functions from the security library by using Java reflection, minimizing the conflicts with the existing code even if multiple security functions are called.

The remainder of this paper is organized as follows. Section II summarizes the previous research on security patching techniques and describes its limitations. Section III provides an overview of the proposed technique and discusses each component in detail. Section IV describes the test and evaluation of the success rate of the proposed technique in commercial apps. Section V discusses the performance issues, experimental limitations, optimization issues, and legal issues. Finally, Section VI presents the conclusions.

This paper is an expanded version of a paper published in the ISSRE2018 industry track [20].

II. RELATED WORK

A. BYTECODE REWRITING

Bytecode rewriting is a technique for adding, modifying, or deleting source code at the same bytecode level [15]. On Android devices, smali code, as a type of bytecode, is an intermediate language [21] that is used to protect against permission-based privacy threats, to administer apps by inserting security functions, to track vulnerable codes, and to analyze programs using reverse engineering [22].

B. PERMISSION-BASED PRIVACY THREAT PROTECTION

On Android devices, there could be cases in which more permissions are required than allowed for the app. The average number of APIs per permission is seven, making it difficult to control each API [23]. For example, camera permissions include the TakePicture() and MediaRecorder() APIs. To address these weaknesses, studies have been conducted to control the permissions associated with potentially risky APIs.

Zhang et al. [11] proposed a method of statically analyzing a security-vulnerable app and adding permission control code at the point at which permission-related APIs are called that could potentially leak sensitive information. The permission control then displays a warning window that allows the user to choose whether to grant permission. This method requires preliminary statistical analysis of the app and has the limitation of controlling only permission-related APIs. In addition, in terms of the user interface, there is a lack of convenience as the end user must decide whether to allow permission.

Backes et al. [9] added monitoring code to an app for permission control, which monitors permission-related API calls. Unlike in the preset policy file, a warning window is displayed when the permission-related API declared in the policy file is called. The warning window allows the user to choose at what level the API provides the desired information. For example, in the case of location information, it is possible to select whether to provide approximate or precise location information. However, the monitoring service must be operated in real time for permission control. In addition, as reported previously, only permission-related APIs can be controlled, which is inconvenient for users as they must decide whether to allow permission. Although detailed log information is not provided, the available limited information on the log function grants users access to the log related to the permission calls.

Neisse et al. [10] proposed a method of controlling permissions based on a preset permission control policy. This method sets a policy on how much information to provide when there is an API call with potential privacy threats. The set policy monitors the presence of API calls while the permission control library is running. If there is such a call, the level of information provided is adjusted accordingly. It provides a user interface for nine items, including structure, behavior, threat rule, and role for policy setting, but only policy setting for privacy-threat-related APIs is possible, regardless of the flow of the app. For example, control is generated in all sections of the app where the same API is called, but not in sections that require API control. Research on permission control has not addressed the ability to add security functions to insecure apps.

C. APP MANAGEMENT BY INSERTING SECURITY FUNCTIONS

There has also been research on providing security where needed on a per-method basis instead of adopting permission control. Lee et al. [14] proposed a method of inserting security functions on a method-by-method basis by using appwrapping, which involves inserting security functions at the bytecode level without relying on the original Android source code of the insecure app lacking security functions. It extracts security function execution code and libraries from the EMM app at the bytecode level (smali code) and inserts them where needed. However, it works on a static policy basis such that every time a policy is changed, security functions must be extracted and inserted into the app; in addition, repackaging is necessary. Furthermore, to obtain the location information (app class and method names) required by the security functions, the class and method names must be
analyzed in advance according to the app flow. For example, information about the names of the classes and methods that are executed immediately after the app launches and as the user progresses through the app is necessary.

D. VULNERABLE CODE PATH
Ma et al. [12] proposed a patch at the smali code level for misused cryptographic APIs in Android apps. After setting the misused cryptographic APIs as templates in advance, finding them in bytecode, and modifying them with normal APIs, they studied patching of the misused app code with the appropriate code at the bytecode level. Their approach enables easy code patching by using the local and parameter variables in the existing code. However, patching is only possible for preset misused code templates. In addition, this method does not provide the ability to insert security functions required by the app. Sie et al. [13] proposed a technique to detect vulnerable code (inter-app attack code) and patch it automatically at the smali code level to prevent inter-app attacks.

E. APP ANALYSIS
Furthermore, studies on security assessment [24] as well as the dynamic monitoring and analysis of obfuscated Android apps [25] have been conducted. However, these studies have not addressed the addition of security functions to insecure apps and have only focused on dynamic analysis of apps at the bytecode level or evaluation of whether security elements have been properly applied.

F. OTHER STUDIES
Benjamin et al. [18] and Hao et al. [26] proposed a framework at the bytecode level to identify security-sensitive APIs and to set custom security policies for each app. In addition, Wang et al. [27] studied the prevention of transitive permission attacks, which are classified as privilege escalation attacks, using bytecode rewriting without modifying the Android framework.

III. APPWRAPPER TOOLKIT
This section introduces the proposed technique, the AppWrapper toolkit. To overcome the limitations of previous research, the AppWrapper toolkit inserts security function execution code at the bytecode level into each method unit of the insecure app needing security functions. Additionally, the AppWrapper toolkit provides dynamic policy management to control whether or not security functions are executed in each method. As shown in Fig. 1, the AppWrapper toolkit consists of three parts, namely, automatic app wrapping, real-time app flow checking and policy setting, and dynamic policy management. The objectives of the proposed technique are as follows.

- **Extraction of bytecode level security execution codes and library**: To call the security functions of an app that provides various security functions (for example, an EMM app), the security function execution code and library are extracted at the bytecode level. The security function execution code is connected to its library with its information (class and method name) as input parameters. The library calls the security functions of the security app according to a predefined policy file.
  - **Insertion of a granular and extensible security function at method level**: The security function execution code is inserted into each method unit of an insecure app. The inserted code uses Java reflection to invoke the security functions of the app dynamically.
  - **Dynamic policy management without app repackaging**: After inserting the security function execution code into each method unit, the dynamic policy determines whether or not to execute the security functions at the method. The policy has the information (method and class name) about where the security function should be executed and which security functions should be executed. Whenever the policy is changed (for example, other security functions are needed), the changed policy is applied without repackaging the app.
  - **Policy setting based on app flow**: By adding the log function of each method unit in the app, the administrator in charge of security can check the log information in real time and understand the app flow (class and method name) while the app is running. The administrator can check the current class and method name of the app and define the appropriate security functions to be executed at the method where the security functions are required.

The overall flow of the AppWrapper toolkit is as follows.
1. Decompile the APK file of the insecure app (the app that needs security functions) to obtain the bytecode (smali file), AndroidManifest.xml file, and resource file.
2. Insert the security function execution code (including the log function) and copy the security function library to the smali file created in Step 1.
3. Create a patched APK file by repackaging the smali file with the inserted security function execution code, existing AndroidManifest.xml file, and resource file.
4. Install the generated patched APK (the APK with the inserted security function execution code) file on a phone.
5. When the patched app is executed, the current class and method information (class and method name) is sent to the log view of the user interface of the AppWrapper toolkit.
6. Check the log view, and click on the log that requires policy setting to move to the policy management screen. In addition, create a policy file by setting the necessary security functions on the policy management screen.
7. Once all of the policies have been set, download the policy file to the phone and run the patched app. The patched app works with the downloaded policy file to execute the security function according to the flow of the app.
The security function execution code and its library are commonly inserted at the bytecode level. For this purpose, AppWrapper decompiles the insecure app to obtain small files consisting of bytecode and an AndroidManifest.xml file to check the app structure. The extraction part only requires small files that contain the security function execution code and its library, whereas in the patching part, the AndroidManifest.xml file is needed to check the app structure and the security function insertion location. Finally, the resource file is necessary for app repackaging.

As a preliminary task of step 2 in Fig. 1, the security function execution code and its library must be extracted at the bytecode level from the app (security function call app) that calls the security functions of the security app (for example, an EMM app). The extracted security function execution code and security library are then added to the method unit within the activity class of the insecure app. The information regarding the activity class can be checked through the AndroidManifest.xml file, and the method information can be checked through the smali file of the corresponding activity class.

After the security function execution code is inserted, the security library is copied to the app for repackaging. The inserted security function execution code determines whether or not to execute the security function for each method in the activity according to policies declared on the policy file, and the policy file is copied to the phone. The security function can be managed simply by changing the policy, and no repackaging is required even if the policy is changed. To achieve this objective, the security library is implemented using the Java reflection technique.

The following subsections describe the security function execution code and library extraction, automatic appwrapping, security function operation and dynamic policy management, and real-time log checking and policy setting.

A. SECURITY FUNCTION EXECUTION CODE AND LIBRARY EXTRACTION

To insert security functions into an insecure app, the security function execution code and security library must be extracted from the app that is calling the security functions of the security app at the bytecode level. This process is a preliminary step before extracting the security function execute code and its library in step 2 shown in Fig. 1. A security function call app is implemented to call the security functions of the security app in Android, using the Java reflection technique. Fig. 2 shows the process of implementing the app that calls the security functions. The flow is as follows.

1. Implement the security function call app in the Android source code. The source code consists of the security function execution code and security library. The security function execution code simply invokes the security library with two input parameters: the activity and method names of the method into which the security function execution code is inserted. These parameters are associated with the policy file to determine if there is a policy corresponding to the activity and method names. The security library loads the policy
from the administrator-generated policy file and calls the security functions of the security app according to the policy.

2. Convert into smali code. To obtain the smali code at the bytecode level, the app is decompiled.

3. Extract the security function execution code and security library from the smali file. Even a single Android source code file is divided into several smali files in the process of being converted into smali code; therefore, only the smalli file for the security library and security function execution code should be extracted. Additionally, the security function execution code should be extracted from the smalli file.

The security library is responsible for calling the security functions of the security app. In other words, the security library must be inserted into the insecure app along with the security function execution code. The security function execution code is inserted into every method unit in the activity class declared in the AndroidManifest.xml file. The execution of the inserted security function execution code is determined by the policies contained in the policy file.

The security function execution code requires two input parameters to compare with the policy and to determine whether or not to execute the inserted security functions, as shown in Table 1. When the security function execution code is inserted into an insecure app, the two parameters are saved with the name of the activity and method into which the security function execution code is inserted. The extracted security function execution code and library can be used even when the security app is updated, unless the classes and method names of the security functions of the security app are changed. Therefore, the security function execution code and security library do not have to be extracted every time the security app is updated.

**TABLE 1. Security execution code input parameters.**

| Variable | Parameter | Description |
|----------|-----------|-------------|
| v0       | Activity  | Activity name into which security function execution code is inserted |
|          | Name      | function execution code is inserted |
|          | Method    | Method name into which security function execution code is inserted |

### B. AUTOMATIC APPWRAPPING

This section describes how to insert the security function execution code and security library into an insecure app. The automatic appwrapping process is shown in Fig. 3. This process occurs after decompiling the insecure app.

1. Check the activity class name. Using the activity information declared in the AndroidManifest.xml file obtained by decompiling the APK file, the activity class name into which the security function execution code and security library will be inserted is checked.

2. Insert security function execution code and copy security library smali file. The security function execution code is inserted into each method unit declared in each activity class smalli file through the activity path and name identified in the AndroidManifest.xml file. When the security function execution code is inserted, the information (activity name and method name) about the code insertion location is written as parameter information in the variable v0. This variable is used in conjunction with dynamic policy to determine whether to execute the security function at the location at which the security function execution code has been inserted (the dynamic policy is described in Section II.C).

3. Repackage to generate the patched APK. Repackaging involves generating the patched APK to which the inserted security function execution code and copied security library are applied through step 2. The APK signing process then allows the APK file to be installed on the phone of the user.

4. Install the patched and security APKs. The patched and security APKs that provide security functions need to be installed. The security app can be either be installed directly as an APK file or automatically suggested for download by the patched app when it is loaded for the first time.

5. Update the original app. As in the patched APK installation, the existing original app is updated to provide security functions.
To insert security function execution code into each activity file, it is necessary to find each activity path and name declared in the AndroidManifest.xml file, where they are declared as `<activity android:name = ...>`. After step 1, the activity file as small code is confirmed. Security function execution code is inserted into every method unit of each activity file and is executed later by dynamic management.

In each activity file, methods are declared as `.method - .end method`. After the `.method` line, the security function execution code is inserted. When the code is inserted, the two parameter values are modified with the inserted location information as `ActivityName-MethodName`, as shown in Fig. 3 (after step 2, “ActivityName-MethodName” of the security function execution code is changed to “com.activityM-method A”), and can be useful for managing security functions later in conjunction with the dynamic policy.

To repackage the patched APK with the inserted security function execution code and copied security library, the smali files, AndroidManifest.xml file, and resource file are needed. In addition, a signing key is required for repackage. After signing with the signature key, the app can be distributed and installed on the phone of the user. If the app is already installed on the phone, the app is updated with the patched APK file.

**C. SECURITY FUNCTION OPERATION AND DYNAMIC POLICY MANAGEMENT**

This section explains how security function execution code and security libraries work and how dynamic policies are managed. Security functions work in conjunction with dynamic policies. As shown in Fig. 4, the security function execution code calls the security library, which in turn calls the security functions of the security app after comparing it with the policy file. When the security function is called and the execution of the security function is completed, the original code of the app proceeds as follows.

1. Call the security library. The security function execution code inserted into the method unit of an insecure app invokes the security library function (loadSecurity). When the security function is called, `ActivityName-MethodName` modified with the current flow location information (for example, “ActivityM-public A”) as the input parameter value is passed to the security library.

2. Check the policy. The security library checks whether the policy is set with the parameter value by comparing the input parameter value of `ActivityName-MethodName` with the policy file.

3. Call the security function. If the policy for the parameter is set, the security library calls the security function of the security app set in the policy file. (For example, the policy of ActivityM-method A calls the fast identity online (FIDO) authentication method of the security class on the security app.)

4. Execute the security function. The security function of the security app is executed as set in the policy file.

After the security function is successfully executed, the original source code in the patched app proceeds.

**FIGURE 5. Security function execution code.**

```java
// 1. Assign Current Flow Information (ActivityName-MethodName)
Const-string v0, “ActivityName-MethodName”
.local v0, “inputCurrentMethod”:
Ljava/lang/String;

// 2. Initiate Security Library with JavaReflection Technique
New-instance v1, Ljava/lang/Class;
Invoke-direct {v1}, Ljava/lang/Class;-><init>(Ljava/lang/String;)V
Invoke-virtual {p0}, Ljava/lang/Class;->getName(Ljava/lang/String;)V
Invoke-virtual {v0, v1, p0}, Landroid/content/Context;->getApplicationContext(Landroid/content/Context;Landroid/app/Activity;)V

// 3. Invoke Security Library
Invoke-static {v0, v1, p0}, Ljava/lang/Class;->loadSecurity(Ljava/lang/String;Landroid/content/Context;Landroid/app/Activity;)V
```
location is passed as an input parameter to the security library. The security library has the following three functions:
- Check the received input parameter value
- Compare the parameter value with the policy file
- Call the security function of the security app

After checking the received input parameter value and loading the policy file, the security library checks whether policies are declared with the parameter value in the policy file. If a policy is declared, the security library calls the security function of the security app corresponding to the policy. As shown in Table 2, the security policy contained in the policy file is set with the activities and method locations for executing the security function, security function execution class, and function of the security app.

| #  | Activity Class | Method Name | Security Library | Security Function          |
|----|----------------|-------------|------------------|---------------------------|
| 1  | ActivityM      | publicA     | Security         | FIDO Authentication      |
| 2  | ActivityM      | publicB     | Security         | Capture Lock             |
| 3  | ActivityZ      | publicZ     | Security         | Office Mode              |

The security library uses Java reflection to invoke the security functions of the security app. Java reflection is not a static method of importing classes and calling methods for execution. Instead, the security library dynamically calls the security functions of the security app using Java reflection. Java reflection is a popular means of dynamically loading and using a class [26]. By dynamically managing policies using this technique, various security functions can be called and managed simply. However, there are two caveats when using Java reflection. First, the classes and methods of the security app should be designed to be brief. Second, the security functions should be separated by strings, enabling these functions to be managed simply when setting the policy. The security function of the policy file includes a method name for each security function. For example, policy 1 in Table 2 calls the FIDO authentication method of the security class of the security app.

To call the security function of the security app dynamically using Java reflection, the information about the method and class name of the security function provided by the security app are needed. As mentioned in Section I, our AppWrapper toolkit can be used to apply EMM solutions. In such cases, the EMM app becomes a security app. Therefore, information about the method and class name of the security function can be obtained from the EMM solution manufacturer.

By using Java reflection, even if the security function provided by the security app is updated, it is applied immediately by the updated policy file when the security app is updated unless the method and class name of the security function are changed. If the method and class name of the security function have been changed, it is only necessary to modify the security function class and method name in the policy file. This approach enhances the efficiency of the proposed scheme because the security function can be invoked by modifying only the policy after the initial appwrapping.

### D. REAL-TIME LOG CHECKING AND POLICY SETTING

To insert security function execution code into an insecure app, it is important to understand the flow of the app and to identify where to insert the security function execution code. For this purpose, the AppWrapper toolkit provides a log function to check the flow of the patched app in real time by inserting a log function into the insecure app, as illustrated in Fig. 6. For example, the security administrator can set up a policy by viewing the AppWrapper toolkit log in real time and clicking the log in a location that requires security function. The process is detailed below.

1. Call the log library. The log function inserted into the method units in the insecure app calls the log library. When calling the log library, the current activity name (with path) and method name are passed as input parameters.
2. Send the log. The log library connects to the AppWrapper toolkit with secure socket layer (SSL) and sends the class and method names to the Log View of the AppWrapper toolkit.
3. Use the log to choose where to execute the security function. The administrator checks the logs in real time on the log view screen in the AppWrapper and selects the log that requires security functions.
4. Policy setting. The policy is created after selecting the security functions required for the location (class and

![FIGURE 6. Real-time log checking and policy setting.](image-url)
method name on the log view screen). The created policy is stored in a policy file, and the policy file is downloaded onto a smartphone that has a patched app installed.

To send the app log information to the AppWrapper toolkit securely and in real time, the patched app must be connected to the AppWrapper toolkit on SSL before sending the log. The send logs are shown in the Log View sequentially and can be checked in the Log View until the patched app ends. The administrator can check not only the activity and method flow of the app, but also the location of the security function that is required. Next, the policy is set directly by clicking on the location on the Log View. The policy file is then created to allow the security function to run where it is needed.

By providing the log function to check the activity and method flow of the patched app in real time, the administrator can easily check the app operation flow and obtain a better understanding of where security functions are needed. In addition, it is possible to improve the user convenience by specifying a policy switch of the dynamic policy setting screen when the log is clicked at the required location.

IV. IMPLEMENTATION AND EXPERIMENT
This section describes the method of implementing the AppWrapper toolkit at a level that can be used in real-world scenarios, as well as the experimental environment and procedures. In Section IV.C, we analyze the performance of the AppWrapper toolkit in four respects.

A. IMPLEMENTATION
The AppWrapper toolkit is applied in actual field scenarios involving the insertion of security functions into business apps using EMM solutions in the business app market for companies adapting BYOD. This section describes how to implement automatic appwrapping and introduces real-time log-checking and policy-setting user screens.

For appwrapping, the apktool.jar (ver. 2.0.1) file is used to decompile and recompile the app [27]. Using this file, smali files for extracting the security function execution code and security library are obtained after decompiling the security function call app. After inserting the security function execution code into the test app (our dataset is described in the next section), the test app is recompiled as a patched APK file. This APK file can then be installed on an Android phone by signing it through the sign.jar file [28]. The automatic appwrapping module executes compiling and recompiling are jointly, as shown in Fig. 3.

When a test app is selected, the automatic appwrapping module (Fig. 3) decompiles the app to obtain the Android-Manifest.xml file and smali files, thereby yielding a list of activities declared in the AndroidManifest.xml file. After loading the activities, the security function execution code (including the log function) is inserted on a per-method basis into each smali file of the loaded activities. Among the methods into which the security function execution code is inserted, the inserted code is removed from those that cause errors later. Methods that cause errors will be elaborated upon in Section V. When inserting security function execution code on a per-method basis, the parameters of the security function execution code are modified according to the activity and method names that are inserted.

When the patched APK file is installed and executed on the phone, the log is sent to the Log View screen in real time, as shown in Fig. 7. The logs sent from the phone are transmitted securely via SSL communication. The administrator can check the logs in real time to determine current location (activity and method names). Then, the administrator can set the policy at the dynamic policy setting screen as shown in Fig. 8, with the following flow.

1. Select the activity name in step 1 of Fig. 8.
2. Select the method name that requires security functions from the list of methods declared in step 2 of Fig. 8.
3. Select the security functions required by the class and method name in step 3 of Fig. 8.

The policy is set by the administrator following above 3 steps, and the policy file is created. Alternatively, if the administrator selects a log in the Log View screen, the class and method names of steps 1 and 2 in Fig. 8 are automatically selected, and the administrator can set the policy only through
necessary security functions in step 3. When the generated policy file is downloaded onto the phone of the user and the patched app is executed, the security functions are operated at the location set in the policy and compared with the policy file according to the operating flow of the app.

B. EXPERIMENTAL ENVIRONMENT

To evaluate the performance of our AppWrapper toolkit, experiments were conducted using Android commercial apps. We collected at least three apps in 33 categories to build a dataset of a total of 79 apps in the Google Play (Korea) market, the official Android market. Among the collected apps, apps forged through signature key verification or apps that included repackage blocking were excluded from the dataset, because our AppWrapper toolkit targets the business app market of companies adopting BYOD or mobile office policies. The excluded apps will be discussed in Section V. The apps in the dataset were divided into five file size groups, as shown in Table 3.

| App size | Quantity |
|----------|----------|
| <10 MB   | 30       |
| 10–20 MB | 17       |
| 20–30 MB | 11       |
| 30–40 MB | 10       |
| 40–50 MB | 6        |
| >50 MB   | 5        |

The mobile device used for the experiment was a Samsung Galaxy S6, and the Android version was 7.0. The computer system was Windows 7 64-bit, the CPU was Intel i7-4770 3.5 GHz with a memory of 8 GB.

The experiment confirmed that decompile, appwrapping (inserting security function execution code and copying security library), and repackaging were performed on 79 APK files of the dataset. We also tested whether the security function was executed properly according to the policy after execution of the patched app.

C. EXPERIMENTAL RESULTS

We experimented with the AppWrapper toolkit with the data set of 79 apps, and all 79 apps were successfully installed and running. Fig. 9 shows the app launch screen before (A) and after (B) the security function execution code was inserted. The patched app executed the security functions according to the policy declared in the policy file. As shown in Fig. 9B, we confirmed that FIDO authentication was successfully executed with user authentication. For detailed performance evaluation of the proposed technique, further experiments were conducted with the following objectives.

Q1. How long does it take to create a patched app using the AppWrapper toolkit?

Q2. What is the file size change after appwrapping?

Q3. What is the coverage of the activities and methods into which the security function execution code is inserted?

Q4. How long does it take to execute a security function through the security library using Java reflection?

D. PROCESSING TIME

Using the proposed technique, we measured the time required to create a patched app by automatic appwrapping to each app of our dataset. As shown in Table 3, which is classified by APK file size, the time taken for automatic appwrapping by APK file size was measured. The measured time was divided into decompiling, appwrapping (insertion of security function execution code and copying of the security library), and repackaging (recompiling and signing). The processing time of each app corresponding to the APK file size group was calculated to obtain the average processing times for decompiling, appwrapping, and repackaging.

As shown in Table 4, the appwrapping phase took the least amount of time. The time spent in the overall phase decreased from 3.49% to at least 2.02%. Within a maximum of 2.6 s and a minimum of 1 s, the security function execution code was inserted into all of the methods of the activities declared in the AndroidManifest.xml file of the app. In addition, this time included the time during which the security library was copied to the smali folder of the insecure app. As the APK file size increases, the processing time of each step as well as the appwrapping generally increases.

The time from decompiling to repackaging was the least for the APK files with sizes of less than 10 MB, which averaged approximately 28 s. On the contrary, the 40–50 MB file size group required the most time, at approximately 95 s, owing to the increased decompiling and repackaging times.

The experimental results show that the automatic appwrapping time of the proposed scheme is not critical. In addition, even if the policy is changed after appwrapping, it is not necessary to perform decompiling, appwrapping, and
TABLE 4. Security patch processing time (s).

| App   | Decompile | Appwrapping | Recompile | Sign | Total |
|-------|-----------|-------------|-----------|------|-------|
|       |           |             |           |      |       |
| <10 MB | 7.340     | 1.010       | 18.346    | 2.254| 28.921|
|        | (23.381%) | (3.493%)    | (63.332%) | (7.794%)|       |
| 10–20 MB | 11.994    | 1.516       | 31.351    | 3.951| 48.946|
|        | (24.505%) | (3.097%)    | (64.400%) | (7.799%)|       |
| 20–30 MB | 13.648    | 1.841       | 43.189    | 7.590| 66.268|
|        | (20.595%) | (2.778%)    | (65.173%) | (11.454%)|       |
| 30–40 MB | 17.479    | 2.383       | 55.595    | 10.480| 85.937|
|        | (20.339%) | (2.773%)    | (64.693%) | (12.195%)|       |
| 40–50 MB | 18.068    | 2.662       | 61.248    | 13.886| 95.864|
|        | (18.848%) | (2.777%)    | (63.891%) | (14.485%)|       |
| >50 MB | 16.354    | 1.752       | 52.538    | 15.690| 86.334|
|        | (18.943%) | (2.029%)    | (60.854%) | (18.174%)|       |

TABLE 5. Security patch file size changes.

| App   | APK File Size Original | After Appwrapping | Main Class File Size Original | After Appwrapping |
|-------|------------------------|-------------------|-------------------------------|-------------------|
|       |                        |                   |                               |                   |
| <10 MB | 6,345 kB               | 6,538 kB (+193 kB, 3.0%) | 34 kB (64 KB, 15.7%)          |                   |
| 10–20 MB | 14,696 kB             | 14,942 kB (+247 kB, 1.6%) | 60 kB (+10 KB, 13.8%)        |                   |
| 20–30 MB | 25,737 kB             | 27,155 kB (+417 kB, 5.2%) | 25 kB (+6 KB, 20.5%)         |                   |
| 30–40 MB | 34,974 kB             | 35,533 kB (+579 kB, 1.6%) | 17 kB (+5 KB, 22.8%)         |                   |
| 40–50 MB | 47,285 kB             | 47,620 kB (+335 kB, 0.7%) | 27 kB (+8 KB, 22.3%)         |                   |
| >50 MB | 75,538 kB             | 73,984 kB (+446 kB, 0.6%) | 147 kB (+18 KB, 10.8%)       |                   |

repackaging again. Therefore, our proposed scheme is efficient in terms of processing time. In Section V, not only the processing time, but also other performance evaluation criteria will be discussed in comparison with previously obtained results in existing works.

E. FILE SIZE

We investigated the size changes of the APK files and main class files repackaged before and after appwrapping. When the security function execution code was inserted into every method in every activity class declared in the AndroidManifest.xml file, there was no noticeable difference in file size. Table 5 shows the changes in the APK file and main class file sizes according to the APK file size group.

The APK files smaller than 10 MB showed an average size increase of approximately 3%, whereas the main class files showed an increase of approximately 15.7%. The APK and main class file size growth rates differed for each APK file, with the APK file size increases ranging from 0.6% to 5.2% and the main class file size increases ranging from 10.8% to 22.8%.

F. COVERAGE PERFORMANCE

In this experiment, among the methods into which the security function execution code was to be inserted, we analyzed what percentage of code was actually inserted. Through the proposed technique, we checked whether the required security function could actually operate at the location at which the security function was required (activity and method name) according to the flow of the app. For each app group, we calculated the ratio between the number of methods into which the security function execution code was successfully inserted and the number of methods without the code inserted, and we determined the average of the calculated values, as shown in Table 6.

Of all of the methods in the class declared in the AndroidManifest.xml file, the percentage of methods with security function execution code inserted was 84.033% on average. Methods containing some specific commands produced errors when recompiling. The security function execution code was not inserted into these methods, which occurred at a rate of 15.33% on average. For the methods allocated as 8 bit register, the security function execution code was not inserted; these methods occurred at an average rate of 0.673%. In the discussion section, the methods that did not have the security function execution code inserted to prevent errors owing to the inclusion of specific commands will be addressed.

The experiment described in Table 6 was conducted to assess the performance of every method in every activity declared in the AndroidManifest.xml file. It did not reveal how many of the activities declared in the AndroidManifest.xml file had security code execution code inserted. Further, of all the methods declared in one activity, it was not
possible to check how many methods had the security function execution code inserted successfully.

Therefore, we conducted additional experiments by dividing the activity and method levels. This approach indicated in more detail whether the proposed technique could execute security functions according to the operating flow of the app.

The coverage ratio of the activity level was the percentage of all activities declared in the AndroidManifest.xml file that had the security function execution code successfully inserted, as shown in (1):

\[
\text{CovRa}_{\text{activity}} = \frac{\# \text{ of Wrapping Activity}_\rho}{\# \text{ of All Activities}} \times 100\%.
\]  

Here, \( \rho \) is the conditional value and includes three conditions regarding all of the activities declared in the AndroidManifest.xml file as follows.

- \( \rho = \text{all} \): the security function execution code is inserted into all methods declared in an activity
- \( 1 \leq \rho < \text{all} \): the security function execution code is inserted into at least one of the methods declared in an activity
- \( \rho = 0 \): the security function execution code is not inserted into any of the methods declared in an activity

The activity level coverage results for each app is are in Fig. 10. The x-axis is sorted in ascending order from the left according to the app size. Evidently, 79 apps had the security function execution code inserted successfully, corresponding to an average of 99.6% including the cases with \( \rho = \text{all} \) and \( 1 \leq \rho < \text{all} \). These numbers show that security function execution code could be inserted into almost all of the activities declared in AndroidManifest.xml as the app activity flow.

The coverage ratio of the method level is the percentage of all of the methods declared in the activity that had the security function execution code successfully inserted, as shown in (2):

\[
\text{CovRa}_{\text{method}} = \frac{\# \text{ of Wrapping Method}_\rho}{\# \text{ of All Methods}} \times 100\%.
\]  

This quantity shows how much of the security function execution code was inserted for each activity with \( 1 \leq \rho < \text{all} \) for the activity level coverage. This equation has also \( \rho \) as

### TABLE 6. Number of methods for coverage performance.

| App       | Avg. # of inserted methods | Avg. # of methods not inserted due to containing parameters | Avg. # of excepted methods |
|-----------|----------------------------|----------------------------------------------------------|----------------------------|
| <10 MB    | 251 (79.987%)              | 1.733 (0.552%)                                           | 12.6 (4.015%)              |
|           |                            |                                                          | 45.8 (14.595%)             |
|           |                            |                                                          | 2.667 (0.85%)              |
| 10–20 MB  | 627.647 (75.417%)          | 3.353 (0.643%)                                           | 28.118 (3.379%)            |
|           |                            |                                                          | 161.824 (19.444%)          |
|           |                            |                                                          | 9.294 (1.117%)             |
| 20–30 MB  | 692.933 (84.891%)          | 2.2 (0.27%)                                              | 35.467 (4.345%)            |
|           |                            |                                                          | 81.6 (9.997%)              |
|           |                            |                                                          | 4.067 (0.498%)             |
| 30–40 MB  | 1715.091 (88.644%)         | 9.455 (0.489%)                                           | 121.909 (6.301%)           |
|           |                            |                                                          | 76.273 (3.942%)            |
|           |                            |                                                          | 12.091 (0.625%)            |
| 40–50 MB  | 1237.833 (86.825%)         | 5.333 (0.374%)                                           | 142.5 (9.995%)             |
|           |                            |                                                          | 36.5 (2.56%)               |
|           |                            |                                                          | 3.5 (0.245%)               |
| >50 MB    | 1719.857 (88.438%)         | 7.286 (0.375%)                                           | 164.286 (8.448%)           |
|           |                            |                                                          | 43.857 (2.255%)            |
|           |                            |                                                          | 9.429 (0.485%)             |
| Total Avg.| 84.033%                    | 0.45%                                                    | 6.080%                     |
|           |                            |                                                          | 8.799%                     |
|           |                            |                                                          | 0.637%                     |

**FIGURE 10. Activity level coverage.**
the conditional value and includes the following condition on each activity in the AndroidManifest.xml file.

- $1 \leq \rho < \text{all}$: the security function execution code is inserted into at least one of the methods.

The method level coverage results for the apps are shown as Fig. 11. The x-axis is sorted in ascending order starting from the left according to the app size. Of all of the methods declared in activities with the security function execution code successfully inserted into one or more methods, the code was successfully inserted into 77.17% of the methods on average. In short, for 51.46% of the activities with $1 \leq \rho < \text{all}$ for the activity-level Coverage, 77.17% of the declared methods could execute the security functions through the security function execution code.

**G. EXECUTION TIME**

This section discusses the effectiveness of dynamic policy-based security function execution according to the measured time required for the security function to execute in a patched app. For the time measurements, the start and end of the security function execution code were measured. The average value was found by calculating the differences between the recorded times, and the units of measurement were milliseconds (ms). The experimental results are shown in Table 7.

The average time required to invoke dynamic policy-based security function for the 79 apps with the successfully inserted security function execution code was approximately 4 ms, ranging from 2.9 ms to 5.5 ms depending on the app size. This result shows that the security function is called quickly without causing any overheads in the app operating flow.

**V. DISCUSSION**

This section discusses the performance, apps excluded from dataset, optimization, packaging, and legal issues related to the proposed technique.

**A. PERFORMANCE**

This section compares the performance of the proposed technique with those reported in the existing static policy-based studies [14]. Prior to this analysis, it was difficult to compare the proposed approach directly with the existing techniques. Table 8 lists the processing times, file sizes, and execution times of the proposed and existing methods.

| Category        | AppWrapper | Others       |
|-----------------|------------|--------------|
| Processing Time | 2 s        | 10 s (83.6%) [12] |
| File Size       | + 2.12%    | + 10.45% [11] |
| Execution Time  | 4 ms       | 9.9 ms [14]  |

In terms of processing time, 83.6% of the test apps in a previous code patch study [12] took less than 10 s, while in a static policy-based study [14], it took 0.15 s to insert the security function execution code into the only two methods. In the AppWrapper toolkit, the security function execution code was inserted into every method in every activity declared in the AndroidManifest.xml file, which took an average of 2 s and up to 2.67 s (1,256 methods). In case of the static policy-based approach [14], inserting security function execution code into 1,256 methods took about 92 s (0.15 s × 1,256/2). Hence, compared with previous approaches, our technique showed excellent efficiency.
In terms of file size increase, an existing permission control study [11] showed a 10.45% file size increase, and a static policy-based study [14] showed a 0.49% increase. The static policy-based study showed a small increase because only two methods had security function execution code inserted. The AppWrapper toolkit saw an increase of about 2.12%, even though the security function execution code was inserted to every method in every activity declared in AndroidManifest.xml due to the simplified security function execution code using the Java reflection technique.

In terms of security function execution time, the control code execution time for one API control was about 9.9 ms in the previous study (Samsung Galaxy s5 lollipop version, 1.9 GHz quad-core) [14], but our proposed technique yielded a security function execution time of only 4 ms (Samsung Galaxy S6 Nougat version, 2.1 GHz quad-core). Further, as the security function execution time is determined according to the dynamic policy, even if the security function was to change as the policy was changed, the security function execution time would not differ significantly.

### B. COVERAGE

In our proposed technique, security function execution code that calls the security library, and the security library dynamically calls the security function of the security app by using the Java reflection technique. This approach minimized the conflict with the existing app code when the security function execution code was inserted. As a result, the security function execution code was successfully inserted into 84.033% of the methods (using three local variables), as shown Fig. 12.

![FIGURE 12. Method coverage based on number of local variables.](image)

In the existing static policy-based study, only one security function can be executed into the security function execution code. The number of local variables used at this time depends on which security function is being run. Assuming that five local variables are required to implement one security function, the static policy-based study shows about 81% method coverage. However, if two security functions are being executed, 10 parameters are required in total and the method coverage is 72%. In other words, in a static policy-based study, it is difficult to insert security execution code with multiple local variables at a method location requiring multiple security functions. Specifically, the method coverage of our proposed technique was 84.033%, indicating that the security function execution code was inserted to more methods than in the existing approach. In addition, the Java reflection technique allows multiple security functions to be executed simultaneously in all of the methods to which the security function execution code is inserted.

### C. NOT INSERTED METHODS

The proposed technique inserts security function execution code into all of the methods in all of the activities declared in the AndroidManifest.xml file. However, in cases in which errors occur due to conflicts with the existing code when the security function execution code is inserted into the smali code at the bytecode level, the security function execution code is not inserted. Errors that occur due to conflicts with the existing code may cause repackaging not to be performed, and even if the repackaging is successful, an error may occur due to a collision in the memory area allocated at the time of app execution. To prevent such errors and collisions, exceptions are made when there is a conflict with the existing code or a conflict in the allocated memory area. Exceptions are classified as methods into which security function execution code is not inserted because it includes a specific command (number of methods not inserted in Table 6) and methods into which the security function execution code is not inserted from the beginning (number of exceptions in Table 6).

A method into which no security function execution code is inserted is identified when the number declared as local is 14 or more. When inserting security function execution code, the variables that the code uses include a parameter variable and several local variables. When the code is to be inserted into a method with 14 local variables, the parameter variable of the security function execution code uses up to 15 memory allocations, and the local variables exceed the 15th address memory allocated to the 4-bit registers (0 to 15th memory), causing a register error to occur. Consequently, the security function execution code is not inserted into methods with more than 14 local variables.

If the security function execution code is inserted into a method allocated with 4-bit registers causing the allocated address memory to be exceeded, the instructions in the method must be changed to use 8-bit registers. In the smali language, the 4-bit and 8-bit registers have different instructions [18]. However, some 4-bit register instructions cannot be easily changed to 8-bit register instructions because the memory address area is a different instruction, and a simple change can cause an invalid register instruction error.

To solve this problem, the following approaches can be investigated. By restoring the APK file with Android code
rather than smali code, we can see the code for methods that do not have security function execution code inserted because of the 4-bit and 8-bit register problem. We can insert security function to the method as Android code, recreate the APK file, and decompile it to see how it is converted to smali code. Thereafter, the method assigned to the 4-bit register can be compared to the method assigned to the 8-bit register. Then, it can be determined how best to change from a 4-bit register instruction to an 8-bit register instruction.

There are cases in which the security function execution code is not inserted because it includes a specific command such as pnumber, sget, or param, which are designated as Types 1, 2, and 3, respectively, as shown in Fig. 13. In the Type 1 case, the security function execution code is not inserted due to 4-bit and 8-bit register conflicts. If the sum of the number of local variables in the method and the number of variables in the security function execution code is more than 16, the inserted security function execution code is cancelled because it exceeds the 4-bit register area. In the Type 2 case, a p0 error occurs when the parameter variable p0 is already declared in the method, as shown in Fig. 13. This error occurs because p0 has already been declared, and p0 in the code is duplicated. In this case, the inserted security function execution code is also canceled. The Type 3 case in Fig. 13 shows an error during recompiling when the sget-object v0 command exists in the existing code, the inserted security function execution code is canceled.

As mentioned in Section IV.F regarding the performance, the proposed technique confirmed that the security function execution code was successfully inserted into 84.033% of the methods of all of the activity classes declared in the AndroidManifest.xml file. In terms of coverage issues, 99.6% of the activity classes had one or more methods covered. These values seem to be sufficient to add the required security functions to the app flow. It remains as a topic for future work to enable the security function execution code to be inserted into all of the methods.

D. PACKAGING

Packaging is a technique used to conceal code in Android apps, and malicious apps are often utilized to conceal malware [29]. This technique is commonly employed in malicious apps [30], [31]. AppWrapper toolkit, the technique proposed in this paper, is targeted toward the business app market.

The insider threat of a company security administrator inserting malicious code into a business app through packaging for malicious purposes was not covered in this study. To address such cases, internal code inspection can be performed and countermeasures can be taken against insider attacks [32]. In the future, we plan to expand the AppWrapper toolkit to detect and patch malicious code hidden in apps.

E. LEGALITY

This section discusses whether the Google Play policy on modifying Android apps without Android source code is violated. Our technique can be used to add security functions into insecure commercial apps. Even if the app developer does not support the app further, it can be used to insert security functions. There is no provision for app modification without the Android source code on Google Play. In addition,
the Google Android App Distribution Policy [33] only states what to watch for when distributing apps, and there are no restrictions on app modification and redistribution.

Let us also consider the app developers and users. For app developers, the proposed technique does not seem to limit the redistribution of security patches applied to commercial apps. App developers have their own signature keys, so no app forgery problems are caused by these keys. In the corporate app and third-party markets, it seems that there are no limits, as in Google Play [34].

F. APPS EXCLUDED FROM DATASET

Among the apps collected from Google Play, those with security features were excluded from the dataset. These apps were equipped with anti-repackaging protection or app forgery checks. Apps that are equipped with anti-repackaging protection may not be properly decompiled when using aptool.jar, or errors may occur when recompiling [12]. The app forgery check inspects the signature key employed when signing the app using the app server. Our AppWrapper toolkit is a method of strengthening the security functions of business apps, and each company has its own signature key for its business apps. App repackaging protection can also be inserted later using the AppWrapper toolkit. Therefore, apps with these security features were excluded from the dataset.

VI. CONCLUSION

Appwrapping is one of the key technologies in EMM solutions that solve security problems in enterprises adopting BYOD or mobile office policies. To provide mobile security using appwrapping technology, various studies have been conducted on, e.g., permission control, patching misused code, and inserting security functions based on static policy, but there are limitations in terms of overhead, user convenience, and repackaging for each policy change.

In this paper, we proposed an AppWrapping toolkit that inserts security function execution code into each method unit of the activities declared in the AndroidManifest.xml file and copies the security library for insecure apps. The inserted security library can easily be managed through policy with Java reflection, facilitating security policy control. The security policy or function can also be changed efficiently after the initial appwrapping. In addition, the log inquiry of the user interface is provided so that the security function execution code can be inserted according to the app flow. The security policy administrator can conveniently set the policy by switching to the policy management interface by simply selecting a location requiring security functions in the log interface. The experiments conducted using commercial apps to evaluate the performance of the proposed technique showed that the security function execution code was successfully inserted into the method units and that there was low overhead in the processing time and file size.

In future research, we will study solutions for cases in which the security function execution code could not be inserted using the present technique. Some methods were bypassed due to exceeding the register allocation area, and because of certain other instructions. The solution is to restore the method to its original state in the Android source code and then to create an APK file by inserting the security function execution code into the Android source code. Also, coverage of various apps will be examined by applying the proposed technique to more commercial apps.

REFERENCES

[1] statcounter. Mobile&Tablet Operating System Market Share Worldwide. Accessed: Jan. 21, 2020. [Online]. Available: https://gs.statcounter.com/os-market-share-mobile-tablet/worldwide/monthly-201812-201912
[2] A. B. Garba, J. Armarego, and D. Murray, “Bring your own device organizational information security and privacy,” ARPN J. Eng. Appl. Sci., vol. 10, no. 3, pp. 1279–1287, 2015.
[3] M. Morolong, A. Gamundani, and F. B. Shava, “Review of sensitive data leakage through Android applications in a bring your own device (BYOD) workplace," in Proc. IST-Africa Week Conf. (ISTAfrica), May 2019, pp. 1–8.
[4] H. Romer, “Best practices for BYOD security," Comput. Fraud Secur., vol. 2014, no. 1, pp. 13–15, 2014.
[5] P. Beckett, “The business risks of using smartphones," Netw. Secur., vol. 2015, no. 11, pp. 16–17, 2015.
[6] Wikipedia. Enterprise Mobility Management. Accessed: Jan. 17, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Enterprise_mobility_management
[7] T. Shields. (Jul. 7, 2014), In the Mobile Security Bout of the Year, App Wrapping Beats. [Online]. Available: https://www.forrester.com/report/Bridge+Your+Enterprise+Mobile+Strategy+To+Partners+And+Contractors/-/E-RES142938
[8] R. Gutierrez. Unwrapping the Truth: Analysis of Mobile App Wrapping. Accessed: Jan. 17, 2020. [Online]. Available: https://www.blackhat.com/docs/us-14/materials/us-14-Gutierrez-Unwrapping-The-Truth-Analysis-Of-Mobile-Application-Wrapping-Solutions.pdf
[9] M. Backes, S. Gerling, C. Hamner, M. Maffei, and P. von Styp-Rekowsky, “Appguard-enforcing user requirements on Android apps," in Proc. Int. Conf. TOOLS Algorithms for Construct. Anal. Syst. Rome, Italy: Springer, 2013, pp. 543–548.
[10] R. Neisse, G. Steiri, D. Geneiatakis, and I. N. Fovino, “A privacy enforcing framework for Android applications," Comput. Secur., vol. 62, pp. 257–277, Sep. 2016.
[11] M. Zhang and H. Yin, “Efficient, context-aware privacy leakage confinement for Android applications without firmware modding," in Proc. 9th ACM Symp. Inf., Comput. Commun. Secur. (ASIACCS), 2014, pp. 259–270.
[12] S. Ma, D. Lo, T. Li, and R. H. Deng, “CDBRep: Automatic repair of cryptographic misuses in Android applications," in Proc. 11th ACM Asia Conf. Comput. Commun. Secur., 2016, pp. 711–722.
[13] J. Xie, X. Fu, X. Du, B. Luo, and M. Guizani, “AutoPatchDroid: A framework for patching inter-app vulnerabilities in Android application," in Proc. IEEE Inf. Conf. Commun. (ICC), May 2017, pp. 1–6.
[14] S.-H. Lee, S.-H. Kim, S. Kim, and S.-H. Jin, “Appwrapping providing fine-grained security policy enforcement per method unit in Android," in Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW), Oct. 2017, pp. 36–39.
[15] A. Bartel, J. Klein, M. Monperrus, K. Allix, and Y. Le Traon, “Improving privacy on Android smartphones through in-vivo bytecode instrumentation," Univ. Luxembourg, Luxembourg City, Luxembourg, Tech. Rep., 2012.
[16] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, “I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications," Mobile Secur. Technol., vol. 2012, no. 2, pp. 1–7, 2012.
[17] J. Kim, I. Kim, C. Min, H. K. Joo, S. H. Lee, W.-T. Kim, and Y. I. Eom, “Static dalvik bytecode optimization for Android applications," ETRI J., vol. 37, no. 5, pp. 1001–1011, Oct. 2015.
[18] Dalvik bytecode. Accessed: Jan. 20, 2020. [Online]. Available: https://source.android.com/devices/tech/dalvik/dalvik-bytecode
[19] G. McCluskey. Using Java Reflection. Accessed: Jan. 20, 2020. [Online]. Available: https://www.oracle.com/technical-resources/articles/java/javareflection.html
[20] S.-H. Lee, S.-H. Kim, S. Kim, and S.-H. Jin, “AppWrapper: Patching security functions with dynamic policy on your insecure Android apps,” in Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW), Oct. 2018, pp. 36–41.

[21] H. A. Müller, J. H. Jahneke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, “Reverse engineering: A roadmap,” in Proc. Conf. Future Softw. Eng., 2000, pp. 47–60.

[22] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh, “A comparison of Android reverse engineering tools via program behaviors validation based on intermediate languages transformation,” IEEE Access, vol. 6, pp. 12382–12394, 2018.

[23] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, “Android permissions demystified,” in Proc. 18th ACM Conf. Comput. Commun. Secur. (CCS), 2011, pp. 627–638.

[24] H. Cho, J. H. Yi, and G.-J. Ahn, “DexMonitor: Dynamically analyzing and monitoring obfuscated Android applications,” IEEE Access, vol. 6, pp. 71229–71240, 2018.

[25] T. Cho, H. Kim, and J. H. Yi, “Security assessment of code obfuscation based on dynamic monitoring in Android things,” IEEE Access, vol. 5, pp. 6361–6371, 2017.

[26] Java Reflection. Accessed: Jun. 11, 2019. [Online]. Available: http://www.oracle.com/technetwork/articles/java/javareflection-1536171.html

[27] Apktool.JAR. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[28] Sign.JAR. [Online]. Available: https://github.com/appium/sign

[29] Y. Duan, “Things you may not know about Android (Un) packers: A systematic study based on whole-system emulation,” in Proc. NDSS, 2018.

[30] Y. Zhang, X. Luo, and H. Yin, “Deshunter: Toward extracting hidden code from packed Android applications,” in Proc. Eur. Symp. Res. Comput. Secur. Vienna, Austria: Springer, 2015, pp. 293–311.

[31] Y. Zhou and X. Jiang, “Dissecting Android malware: Characterization and evolution,” in Proc. IEEE Symp. Secur. Privacy, May 2012, pp. 95–109.

[32] N. Elmrbait, S.-H. Yang, and L. Yang, “ Insider threats in information security categories and approaches,” in Proc. 21st Int. Conf. Autom. Comput. (ICAC), Sep. 2015, pp. 1–6.

[33] Google Android Distribution Policy. Accessed: May 31, 2019. [Online]. Available: https://play.google.com/intl/ALL_kr/about/developer-distribution-agreement.html

[34] Google Play App Distribution for Company. Accessed: May 31, 2019. [Online]. Available: https://developer.android.com/distribute/googleplay/work.html?hl=ko#best-practices-for-managed

SUNGHOO LEE received the B.S. degree from the Department of Computer Engineering, Chungju National University, South Korea, in 2011. He is currently pursuing the Ph.D. degree in information security engineering with the University of Science and Technology (UST). He is also a Student Researcher at the Electronics and Telecommunications Research Institute (ETRI), South Korea. His research interests are behavior-based authentication, user authentication, and mobile security.

SEUNG-HYUN KIM was born in Daegu, South Korea, in 1979. He received the B.S. degree in computer engineering from the Kumoh National University of Technology, South Korea, in 2002, the M.S. degree in computer engineering from POSTECH, South Korea, in 2004, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2017. Since 2004, he has been a member of the research staff at ETRI, Daejeon, South Korea. His main research areas are ID management, mobile security, mobile privacy, and personalization service.

JUNG YEON HWANG received the B.S. degree in mathematics, and the M.S. and Ph.D. degrees in information security from Korea University, Seoul, South Korea, in 1999, 2003, and 2006, respectively. He was a Visiting Researcher with Columbia University, New York City, NY, USA, from 2005 to 2006. He was a member of the Information Security Research Division, ETRI, South Korea, from 2009 to 2020. He is currently an Assistant Professor with the Department of Information Security, Soongsil University, Seoul, South Korea. His research interests include financial security, identity management, privacy-enhancing technology, cryptographic protocols, and their applications.

SOOHYUNG KIM received the B.S. and M.S. degrees in computer science from Yonsei University, Seoul, South Korea, in 1996 and 1998, respectively, and the Ph.D. degree in computer science from KAIST, Daejeon, South Korea, in 2016. He is currently the Project Leader of the Information Security Research Division, ETRI, Daejeon. His research interests include biometrics, identity management, blockchain security, and financial security.

SEUNG-HUN JIN received the B.S. degree from the School of Computer Science and Engineering, Soongsil University, Seoul, South Korea, in 1993, the M.S. degree from the School of Computer Science and Engineering, Soongsil University, Seoul, in 1995, and the Ph.D. degree from the Department of Computer Science and Engineering (Information Security), Chungnam National University, Daejeon, South Korea, in 2004. He is currently the Managing Director/Principal Researcher Planning Division at ETRI, Daejeon. His research interests include information security (PKI, authentication/authorization technique, privacy protection), mobile payment system, and computer/network security.

* * *