Hindawi
Mobile Information Systems
Volume 2020, Article ID 8813243, 11 pages
https://doi.org/10.1155/2020/8813243

Research Article

Self-Controllable Mobile App Protection Scheme Based on Binary Code Splitting

Sungtae Kim,[1] Taeyong Park,[1] Geochang Jeon,[2] and Jeong Hyun Yi[2]

[1]*School of Computer Science and Engineering, Soongsil University, Seoul 06978, Republic of Korea*
[2]*School of Software, Soongsil University, Seoul 06978, Republic of Korea*

Correspondence should be addressed to Jeong Hyun Yi; jhyi@ssu.ac.kr

Received 17 July 2020; Revised 27 August 2020; Accepted 23 September 2020; Published 10 October 2020

Academic Editor: Navuday Sharma

Mobile apps are booming with the expansion of mobile devices such as smartphones, tablet PCs, smartwatches, and IoT devices. As the capabilities of mobile apps and the types of personal information required to run apps have diversified, the need for increased security has grown. In particular, Android apps are vulnerable to repackaging attacks, so various code protection techniques such as obfuscation and packing have been applied. However, apps protected with these techniques can also be disabled with static and dynamic analyses. In recent years, instead of using such application level protection techniques, a number of approaches have been adopted to monitor the behavior of apps at the platform level. However, in these cases, not only incompatibility of system software due to platform modification, but also self-control functionality cannot be provided at the user level and is very inconvenient. Therefore, in this paper we propose an app protection scheme that can split a part of the app code, store it in a separate IoT device, and self-control the split code through the partial app. In the proposed scheme, the partial app is executed only when it matches the split code stored in the IoT device. It does not require complicated encryption techniques to protect the code like the existing schemes. It also provides solutions to the parameter dependency and register reallocation issues that must be considered when implementing the proposed code splitting scheme. Finally, we present and analyze the results of experimenting the proposed scheme on real devices.

1. Introduction

Since the advent of mobile technologies, mobile apps have expanded very rapidly. According to IDC's smartphone market share report [1], smartphone shipments are expected to increase from 1.3 billion units in 2020 to 1.5 billion units in 2024 due to the launch of new devices and 5G plans. Of these, Android devices are predicted to occupy 87% of the 1.5 billion units. With the increase in the number of apps, their functions and personal information required from users are diversifying. Apps that require a variety of personal information such as smart banking, social network service (SNS), e-mail, and so on generally store users' IDs and passwords for convenience so that they automatically remain logged in. However, if a device is unlocked or infected with a virus due to an Android vulnerability [2], malware can access or steal confidential information and leak it to an attacker.

Currently, various authentication schemes [3–7], such as password, pattern, and biometric information authentication, are provided with Android smartphones. However, once the authentication is made, apps can be run without any restrictions until the smartphone is locked. In other words, unauthorized users can access personal information if they manage to pass the authentication process. In particular, Android apps are vulnerable to repackaging attacks [8], so various code protection techniques such as obfuscation and packing have been applied. However, apps protected with these techniques can also be disabled with static and dynamic analyses.

To deal with these problems, many techniques [9–11] are introduced to protect the app by modifying the platform or using root privileges. Typically, a monitor function is inserted inside an app that contains a lot of sensitive personal information to trace and control the behaviour of the app. However, this approach of modifying the app itself is

very inconvenient to apply directly at the user level. In order to overcome these shortcomings, techniques that allow users to directly protect apps by utilizing a private launcher are recently introduced [12, 13].

Therefore, in this paper, we propose a self-controllable mobile app protection scheme that can freely split binary code and authenticate using the split code to resolve smartphone security issues. The proposed scheme randomly splits the code of the target app through a launcher app, stores it in a separate IoT device, and reinstalls it after reconfiguring it as an executable app with the rest, except for the missing split code. With the proposed scheme, an app can only be run through the proposed private launcher. When the app is executed, the proposed launcher can receive the split code from a separate IoT device and deliver the split code to the app for execution. At this moment, a code-based authentication scheme is used, so only authenticated code can be run in the app and there is no need for a complicated cryptographic authentication. By using this scheme, only the user who has the split code can run the app, thereby improving the security of the smartphone by preventing the unauthorized user from running the app. In addition, the proposed scheme can be applied at the app level, so no platform modification and root privileges are required. The user can simply improve the security of personal information by installing the app.

In order to implement the code splitting function, which is the core part of the proposed scheme, a parameter dependency problem and a register reallocation problem inevitably occur. In this paper, solutions to these problems are presented in detail along with sample codes. It also describes the results of measuring feasibility and performance overhead of the proposed scheme on real Android device and smartwatch.

This paper is organized as follows. Section 2 addresses the related work. Section 3 provides the background and motivation behind the proposed scheme. Section 4 describes the design of the proposed scheme. Section 5 describes issues that arise when implementing the proposed scheme and their solutions. Section 6 demonstrates the experimental results with the proposed scheme. Finally we conclude the paper in Section 7.

2. Related Work

Protecting mobile apps by modifying the platform or using root privileges is inconvenient and difficult for users to apply directly. Many techniques [9–11] have been developed to modify and protect the app itself. I-arm-droid [9] identifies security-sensitive API methods and specifies security policies for the app. It also improves security by rewriting bytecodes in policies by monitoring apps. Aurasium [10] does not require modification of the Android OS to provide the security and policy desired by the user. This tool also monitors behaviour for privacy breaches, such as attempts to retrieve sensitive information from users or access malicious IP addresses. However, in such methods, a monitor function should be inserted inside an app that contains a lot of sensitive personal information to control the behaviour of the app. However, this method of modifying the app itself is very inconvenient to apply directly at the user level.

Recently, many protection schemes have been introduced through the launcher app [12, 13] to help users manage the app comfortably. In general, Android launcher refers only to a program that runs a home screen in the user interface (UI) [14–16]. In most cases, it consists of home screens and app drawers, and it can be seen that it is included in the Android UI. In addition, the app is always running while the terminal is running, and additionally, the home screen area can be provided to arrange shortcut icons or widgets of the app so that the developer can execute the desired function, such as executing or deleting other apps. The manufacturer's launcher is designed as the default launcher from booting, but as the new launcher is installed, a selection window pops up from the home button and allows selecting the installed launcher. To change the default launcher that is already specified, we can use the launcher to clear the default task or use a separate app. Boxify [17] is a representative example of a protection technique through a launcher app. It executes the target app through the launcher app, which is an isolated process with minimal privileges, and monitors it through hooking to control untrusted apps from doing actions that cause damage such as personal information leakage.

In addition, an example of applying the code splitting technique to Robot OS (ROS), an embedded software for smart cars, was recently introduced [18]. This study applied the code splitting scheme for the purpose of secure booting to prevent an attacker from remotely controlling the smart car. In addition, while this uses code splitting for native code, the proposed scheme is applied for Android bytecode. Except for the concept, the detailed underlying techniques such as parameter dependency checking and register reallocation are completely different.

3. Background

3.1. Android App. Android apps are provided in a single file called the Android package (apk) file. The apk file is in zip format and consists of classes.dex, which contains not only the app's code, but also resource files that contain configuration information such as the app's icons, images, and strings. Each apk file contains an AndroidManifest.xml file containing the app's components and permission information. The main language of the Android app is Java. Java code is compiled into the Dalvik bytecode and consists of a file called classes.dex. The generated bytecode is executed on the Dalvik virtual machine. In addition, developers can use native library (.so) written in the C or C++ language. These native library codes run directly on the processor of the device, not on the Dalvik virtual machine.

All apps can be identified by a unique package name and are self-signed by the developer's private key [19]. Android apps consist of different types of components: Activity, Service, Broadcast Receiver, and Content Provider. The Activity represents functions that are performed through a UI. A single app can consist of several activities. In contrast, the Service runs in the background without a UI. For

example, a music player app might require a UI for selecting songs, but no additional UI is required while music is playing. This task can be implemented as a Service. The Broadcast Receiver is a function that can receive the message service and perform the corresponding action when a system event occurs in Android. Finally, the Content Provider is used to provide app data to other apps.

3.2. Android Repository. Android has internal storage and external storage [20]. The internal storage primarily stores systems and apps, while data is stored in the primary external storage. The internal storage can read and write data only in the apps, and the external storage is used as a common area. Also, the data in the internal storage is deleted when the app is deleted. The external storage contains photos, videos, and other files. With permission, it is possible to read and write data, which is in the external storage, from other apps. There is a cache area, a database area, and a file area in the internal storage that exists for each app. Since it is troublesome to find the necessary path whenever the path of each area is required, Android provides an API that easily obtains the main path where data is stored.

3.3. ASMDEX. ASMDEX [21] is an open-source project which parses the dex file and organizes it into a tree. It allows the user to modify, add, or delete the generated tree and rebuild it as a dex file. The tree structure created by ASMDEX is shown in Figure 1. When constructing a dex file as a tree, the root node is represented by ApplicationNode. ApplicationNode has member variables called classes, which represents a list of ClassNodes classes. ClassNode parses and holds all information, such as name, authority, and method for every class in a dex file. A member variable, method, represents a list of MethodNodes classes. MethodNode is the information of method contained in ClassNode. Similar to ClassNode, MethodNode parses and contains information about method, such as name, descriptors, exceptions, and number of registers used. Unlike ClassNode, MethodNode may have a duplicate name. In such a case, the method is identified through a descriptor. The MethodNode class has a member variable, instruction, which is a class called InsnList that implements a double linked list for AbstractInsnNode. AbstractInsnNode is an abstract class, and a method inherits the corresponding abstract class and executes each instruction.

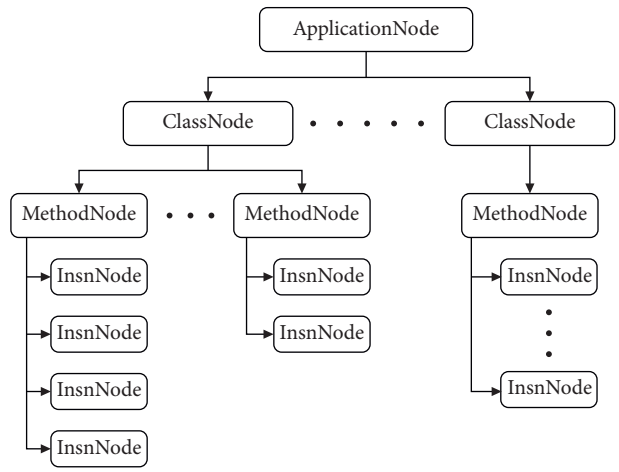

Figure 1: The dex file tree generated from ASMDEX.

4. Proposed Scheme

When the user authentication is performed once before use, anyone can run all the apps installed in a smartphone until it is locked, thereby allowing personal information leakage. This section proposes a scheme to protect personal information by implementing app execution environment through the self-controllable private launcher.

4.1. Concept. The basic idea of the proposed scheme is to split and manage a part of the binary code of the app safely and separately and to take the split code at runtime and functionally assemble it to operate the same as the original code. More specifically, a part of the binary code of the app is split and stored in the IoT device, and each time the app is executed, the split code stored in the IoT device is taken and assembled functionally through a code-based authentication. We call this launcher app that provides this functionality an AppContainer, which is provided in two modes: Normal or Protected modes. If the target app is given as input to the AppContainer, it will enter the Normal mode by default. As shown in Figure 2, when the target app is executed in the Normal mode, the binary code splitting function is operated first. A part of the binary code of the app is randomly selected, split, and then stored in an IoT device. The rest of the code is incomplete, but apparently reconstructed to take the form of the app and reinstalled on the smartphone. This incomplete-but-normal-looking app is called a partial app in the rest of this paper. At this time, the partial app can be run only in Protected mode. That is, after switching to Protected mode, all apps displayed in the AppContainer are partial apps. When running this partial app in the Protected mode, the corresponding split code is received from the IoT device. Then, it operates in the same way as the original app through the code-based authentication protocol.

The dex file can be decompiled into smali code at any time, so it is possible to parse the method in the executable. There can be up to 65536 methods in one dex file. Currently, methods are randomly selected. Even if the proposed scheme is applied to the same app several times, different pairs of split and remaining codes can be generated each time. Thus, by uniquely creating a split code, only the owner of the IoT device can run the app. In addition, if you select the core logic and then store it in the split code and configure only the less important code in the partial app, the core logic of the original app can still be protected even if the partial app is exposed to reverse engineering.

4.2. Design Details. The proposed scheme consists of two main phases: binary code splitting for the target app and applying the code-based authentication to the partial app.

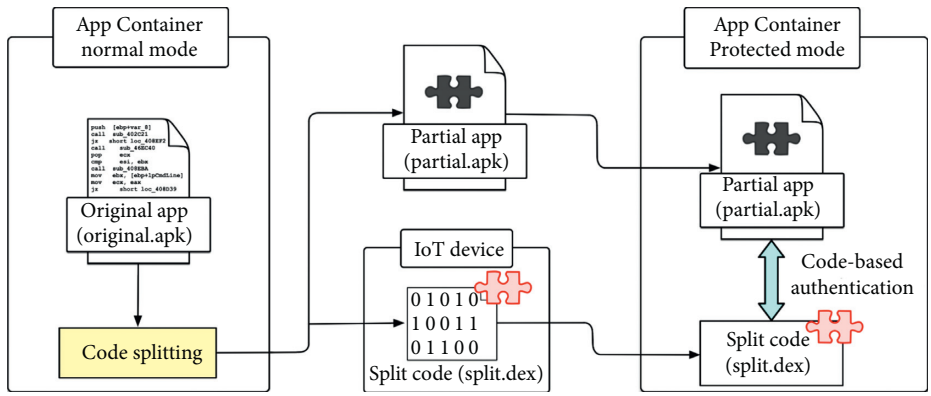

Figure 2: Concept of proposed AppContainer scheme.

4.2.1. Binary Code Splitting. Like the general launcher app, except for system apps, the list of apps installed by the user is displayed in the form of icons in the Normal mode. At this time, when a specific app is selected, the code of the app is split, the rest of the app is rebuilt as a partial app, and a code-based authentication function is additionally applied. This rebuilt partial app is only displayed in the Protected mode and no longer visible in the Normal mode. The outline of how the Normal mode operates is shown in Figure 3. A target app (original.apk) can be selected to apply the code splitting scheme. The package name of the target app goes to the internal memory path/data/app/"packagename"/. This path can only read the original.apk files of the target apps because read-only permission exists for other apps. Then, the original.apk file is copied to external storage such as an/sdcard. To use external storage, the AppContainer must have READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE permissions. Next, the imported original.apk file is unzipped into the original.dex file. After the code splitting process, the original.dex file is split into the partial.dex and the split.img files. The split.img file is reconstructed into the split.dex file with a wrapping function. When the reconstruction is complete, a hash value on the split.dex file is created with the application name. It will be used later for code-based authentication. Then the split.dex file is sent to the connected IoT device. In the other side, the folder containing the partial.dex file is recompressed to create in form of a partial.apk file. This partial.apk file is resigned with the user's private key. Then, the original.apk file is replaced with the partial.apk file in the internal storage. After completing the code splitting process, all files used in the/sdcard path are deleted.

The detailed process of the code splitting scheme is given in Algorithm 1. Given the original.apk file, the original.dex inside the original.apk file is turned into a tree through ASMDEX. It traverses the created tree and randomly selects (we note that it is impossible to split any part of the Android application. This is always applicable only for user-defined classes and methods. It does not apply to classes or methods with framework code or system dependencies. Also, the onCreate function of the MainActivity class, which is the basis for app running, or a class that is automatically created by the system such as R$ should not be selected. However, there is no problem for practical application because sensitive or secret code logic, which is the main target to be split in this proposed scheme, is all user-defined classes and methods) specific splitClass and splitMethod. When the split node selection is complete, ASMDEX is used to create a new tree. The selected splitMethod is added to the splitClass node to create a new dex tree. Since the selected splitMethod disappears from the existing tree, it is necessary to modify the caller part and splitMethod body part. If the selected splitMethod is Static, the splitMethod body part needs to be modified; otherwise, the caller part needs to be modified. If it is not Static method, the splitMethod and splitClass are changed into Abstract, and all splitMethod bodies are deleted. The reason for changing to Abstract is to allow a splitClass with a splitMethod to inherit the existing class and use the undefined methods and field values of the existing class.

Consequently, it traverses the existing targetTree and finds all caller parts of the splitMethod. If a splitMethod is used as an existing class, it is changed to an Abstract method, and thus a splitMethod cannot be used by creating an existing class. The existing class part should be replaced by the stub code, which finds a splitClass that inherits an existing class. A split.dex file is created including the splitClass and then is returned. In the case of Static method, the caller should be replaced with the contents of calling the stub code without removing the caller part. In the split code, the following actions are performed: find and create a splitClass that inherits the existing class, execute the splitMethod immediately, and return the result value of the splitMethod. Inserting or deleting other codes may cause parameter and register dependency problems that may conflict with existing registers because the Dalvik bytecode is register-based, not stack-based. To solve this problem, we have to deal with a parameter dependency checking and a register reallocation, which are explained in Sections 5.1 and 5.2, respectively.

4.2.2. Split Code Integrity Checking. In the Normal mode, when AppContainer separates the split.dex and transmits it to an IoT device, it stores IMEI (International Mobile Equipment Identity) information of the device in the

```
Input: target app (original.apk)
Output: remaining target tree and split code tree
(1)  Tree targetTree, newTree;
(2)  Class splitClass;
(3)  Method splitMethod;
(4)
(5)  targetTree = makeTree (original.dex);
(6)  selectSplittingTarget (targetTree, splitClass, splitMethod);
(7)
(8)  ASMDEX.init (newTree);
(9)  ASMDEX.makeClassNode (newTree, splitClass);
(10) ASMDEX.makeMethodNode (newTree, splitMethod);
(11)
(12) if splitMethod.Type = = STATIC then
(13)    convertMethodCalleeToStub (targetTree, newTree, splitClass, splitMethod);
(14) else
(15)    convertToAbstract (targetTree, splitClass, splitMethod);
(16)    deleteMethodBody (targetTree, splitMethod);
(17)
(18)    for Class class:targetTree.Classes do
(19)       if findMethodCaller (class, splitMethod) = = true then
(20)          convertToStub (class, newTree, splitMethod);
(21)       end if
(22)    end for
(23) end if
```

Algorithm 1: Pseudocode for splitting original app.

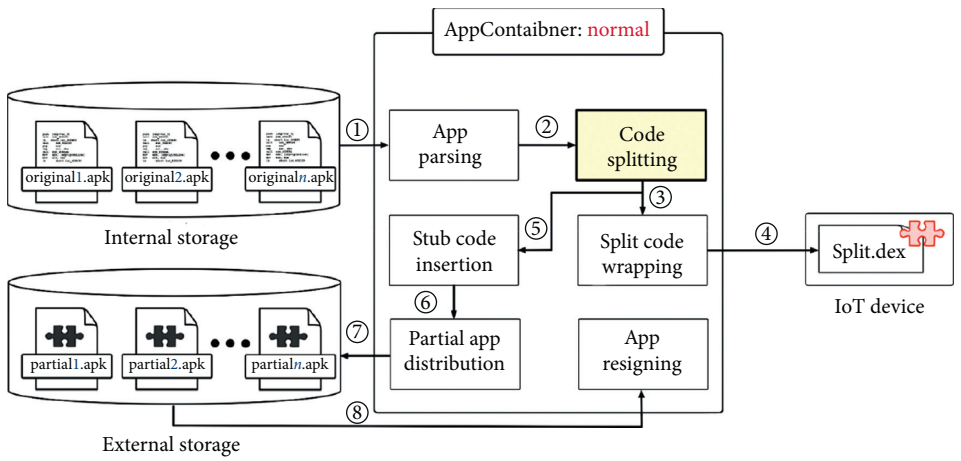

Figure 3: Normal mode operation of the AppContainer.

internal storage. After that, when AppContainer requests split.dex from the IoT device, the IoT device creates a hash value using not only split.dex, but also IMEI and salt. The IoT device transmits the remaining split.dex, salt, and hash value excluding IMEI information to the AppContainer. Then, AppContainer calculates the hash value using split.dex, salt received from the IoT device, and IMEI stored in the internal storage and then checks whether it matches the hash value received from IoT. If the two hash values match, AppContainer proves that it has received the split.dex file from a trusted IoT device and that the integrity of split.dex is verified.

4.2.3. Code-Based Authentication. In the Protected mode, only partial apps with code splitting scheme are displayed in the form of icons on the home screen area. As shown in Figure 4, when the partial.apk starts, it requests its corresponding split.dex to the IoT device. The partial.apk remains on standby until the split.dex file is transmitted from the IoT device to AppContainer. Upon downloading the split.dex file to AppContainer, as explained in Section 4.2.2, AppContainer checks the integrity of the split code and stores it in the internal storage (/data/data/"packagename"/). Then, check if split.dex and corresponding original.dex work normally. If it is wrong, the partial.apk does not work

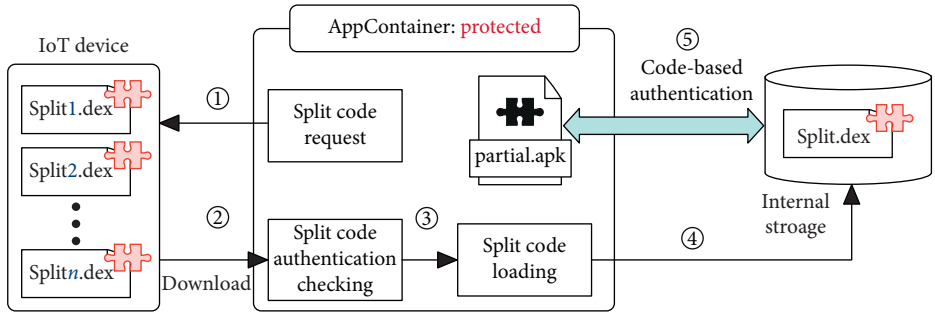

Figure 4: Protected mode operation of the AppContainer.

anymore and is terminated. When the partial.apk terminates abnormally, the split.dex files created during download are deleted. After that, when the partial.apk receives the split.dex file stored in the internal storage through the Intent, the partial.apk can be normally executed. Once again, if you try to run a partial.apk on a general launcher other than AppContainer, the partial.apk does not work because its corresponding split.dex file does not exist.

5. Implementation Issues

In this section, we present several issues and solutions to implement the code splitting scheme described above. In regard to code-based authentication, since there are no implementation problems, we focus on the issues for the code splitting scheme.

5.1. Parameter Dependency Checking. As shown in Figure 5, given the original.dex file, it is divided into the partial.dex and split.img files. When the partial.dex is transformed to the partial.apk, there are important implementation issues on the parameter dependency checking and the register reallocation.

To perform the same operation as before splitting, the splitClass instead of an existing class should be created since the splitMethod is replaced with an Abstract method. In the stub code, a new class that inherits the existing class is created instead (refer to Figure 6). Since the splitClass has always different shape, the type and number of parameters required for class creation are different, so the number of registers used is different. To generically solve this problem, the parameter dependency should be resolved by adding three registers to the method that contains the caller part. Reusing an existing register can cause conflicts with the other code, so only the new register is used. The first register is a register containing name information of a splitClass that inherits an existing splitClass. The second register is an object array that can hold the constructor parameters. The reason for using an object array is that the number of registers used is different because the number and type of parameters in the splitClass constructor are different each time. Therefore, several parameter registers are managed as one register and sent to the stub code to generically fix the caller part. When creating an array of objects, a register is created using the init() in the original code and moved to the second register. The last register is the index register that controls the object array. In addition, parameters of primitive types such as Integer and Double cannot be put directly into the object array, but they must be converted to Integer and Double types using the valueof() function. Therefore, before putting it into the object array, the type is converted into the array by using the register used as a parameter register.

5.2. Register Reallocation. The register dependency problem occurs because it does not match the number of registers previously used. To solve this problem, register reallocation is additionally needed. This task is to solve the index conflict caused by three registers added to resolve the parameter dependency. As shown in Table 1, method registers used in the Dalvik bytecode [22] can be divided into local registers and parameter registers. Local registers are numbered from the beginning, and in the case of parameter registers, the last register is used in all registers. The register before the parameter register is this register that represents the method itself. Thus, adding three registers changes the total number of registers and may cause a malfunction during the execution because the modified register is accessed; thus, the relocation of the registers is necessary.

To solve this problem, at the start of the splitMethod, this register and the parameter registers are returned to the register position before adding the register. As shown in Table 2, when 5 registers are used in the splitMethod and 2 parameters are used, v3 and v4 registers have first and second parameters, and v2 register has this register. If three registers are added, the first parameter goes into the v6 register, the second parameter goes into the v7 register, and this register goes into the v5 register. In this case, if v2, v3, or v4 is used in the original code, an error occurs because the desired value is not included. Therefore, the values of v5, v6, and v7 are put back to v2, v3, and v4. Then the added v5, v6, and v7 registers are used to resolve parameter dependencies.

If Double and Long of the parameter register type are used, two registers are used instead of one. Also, by adding registers, the total number of registers used in the splitMethod may be over 16. In some cases, more than 16 registers of the existing splitMethod may be used. For example, invoke-virtual should be used when using registers less than 16, but invoke-virtual/range should be used when using registers above 16. In addition, the number of registers to be used must

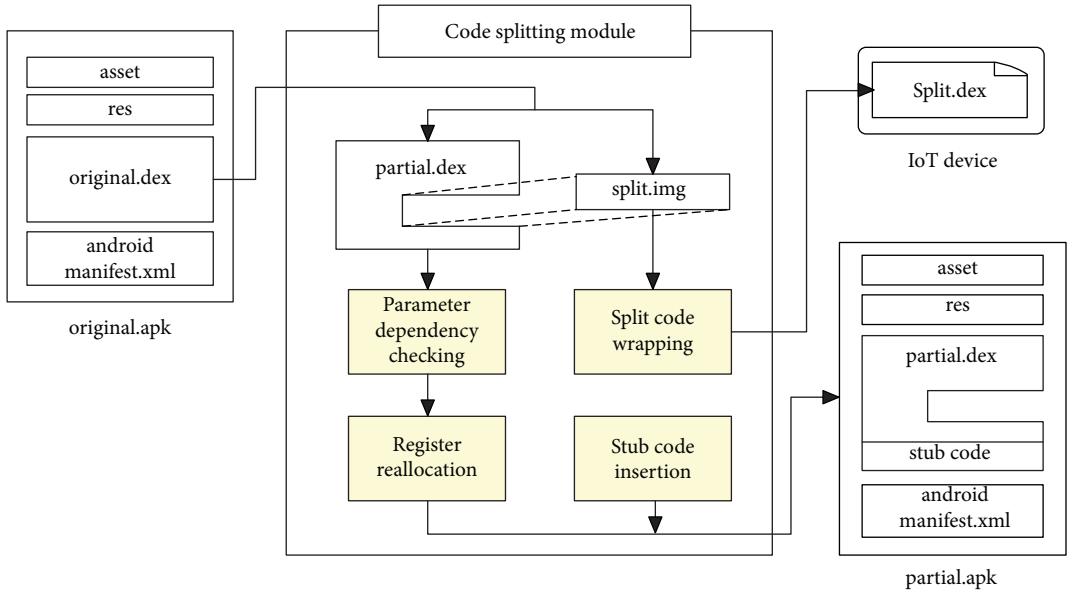

Figure 5: Implementation issues with code splitting.

```
SplitCodeCaller(…)
{
    …
    RandomClass a = new RandomClass(…)
    a.RandomMethod(…)
    …
}
```

⇨

```
SplitCodeCaller(…)
{
    …
    RandomClass a = (RandomClass)StubcodeCall(…)
    a.RandomMethod(…)
    …
}
```

Figure 6: Method caller change for solving parameter dependency problem.

Table 1: Dalvik register allocation (before).

Variables	Parameters	Description
v0		Local register
v1		Local register
v2	p0	This-register
v3	p1	First parameter register
v4	p2	Second parameter register

Table 2: Dalvik register allocation (after).

Variables	Parameters	Description
v0		Local register
v1		Local register
v2		Class name register
v3		Parameter information register
v4		Index register
v5	p0	This-register
v6	p1	First parameter register
v7	p2	Second parameter register

be sequential. When invoke-virtual is available, three registers such as v5, v8, and v3 are available. But when invoke-virtual/range is available, the registers should be v5, v6, and v7. In addition, more than 16 registers cannot new-array and thus cannot create object arrays. The object array is created and relocated using the init() command register, which uses the register below 16 unconditionally, as described above.

5.3. Stub Code Insertion. The stub code needs to be injected in two cases. The first is the case that the AppContainer needs to get the split.dex and store it in internal memory when the app first starts. The second is necessary to load the split.dex from the internal memory when the split.dex is called and to execute the splitClass from the split.dex. The first case analyzes the AndroidManifest.xml and inserts stub code into the Activity class that starts first when the app is run. You also need to modify the beginning of the onCreate() function to add a call to the inserted stub code when the app starts. Moreover, we need to check whether the split.dex received from the AppContainer is the corresponding the split.dex to the partial.apk. If the checking is correct, save the split.dex in the internal memory. If not, terminate the program. The code that loads the split.dex is inserted by adding a splitClass node to the tree created by ASMDEX. The stub code is executed when the caller invokes the split.dex. We create a DexClassLoader object and load the split.dex stored in the internal memory into the DexClassLoader object. Find the splitClass in the created DexClassLoader object and execute the splitMethod normally. Therefore, it is an object that has a class name and constructor parameter value. It finds the desired splitClass by using the reflection API provided by Java. In the case of the Static method, the method finds and executes the method through the object that has the class name, method name, and method parameter value.

5.4. Resigning. When all the code splitting procedures are done, the folder containing the partial.dex is recompressed to create an partial.apk file. Using ASMDEX, a new tree created with split.img is created as a split.dex file. This split.dex file is distributed to the connected IoT device. The Android app must be digitally signed before distribution. Since the original.apk file was modified during the splitting process, the previous signature is useless, so resigning is required to install the app. Therefore, the user's signing key stored in the AppContainer is used. When all the resigning is done, the existing original.apk is deleted, and the partial.apk is reinstalled.

6. Experimental Results

In this section, we describe the results of evaluating performance of the proposed scheme. We implement and measure the performance on an Android version 6 or later and Galaxy Gear for an IoT device.

6.1. Sample Codes with Code Splitting. When the code splitting scheme is applied, the target method is randomly chosen from all the methods. As shown in Figure 7, the addFont method of jxL/biff/Fonts class is selected among all methods and converted into an Abstract method, and the method body disappeared and its size became zero.

Figure 8 shows the caller part of a splitMethod. Previously, only 7 registers from v0 to v6 were used. Three registers were added to modify 10 registers from v0 to v9. We also reallocated the parameter register and this-register through the move-object at the start of the method to avoid register conflicts.

Figure 9 is the part that creates class before calling the splitMethod. It creates an object array using the register used to execute the Init() function and stores the object array in the added register, v8. The register v9 was not used because there were no parameters in the constructor of the splitClass, and the object[] array was also created with a size of zero.

The name of the class to create is stored in the register v7. The class name and the object array to be created are sent with the parameters for calling the stub code, and the/range command is used in case the register number becomes 16 or more. The generated class is cast to the original class and stored in the register v0 because the original class uses the register v0 in the code before modification.

Figure 10 shows the splitClass and splitMethod in the split.dex file stored in the IoT device. In the example code above, there is a splitMethod in the newly named class that inherits the selected class.

If the static method is selected for splitting as in Figure 11, the change of the caller part is not necessary and only the body of the splitMethod is changed. The changed code executes the method by sending class name, method name to execute, and parameters of the method to stub code. It then processes the parameter information received by the method, converts the result to the original return format, and delivers it.

```
com/example/secrettest/Secret4 extends java/lang/Object
    method: <init>()V [public constructor] size:4
    method: sum1(I I)I [public] size:3
    method: sum2(I I I)I [public] size:3
```

⇩

```
com/example/secrettest/Secret4 extends java/lang/Object
    method: <init>()V [public constructor] size:4
    method: sum1(I I)I [public] size:3
    method: sum2(I I I)I [public abstract] size:0
```

Figure 7: Method change with code splitting.

```
# Parameters:
# - Local registers:v0…v11
# - v12:android.view.View
#
# - return:void

onClick-BB@0x0 :
    const/4                          v10, 1
    const/4                          v9, 3
    const/4                          v8, 2
```

⇩

```
# Parameters :
# - Local registers :  v0…v14
# - v15 : android.view.View
#
# - return:void

onClick-BB@0x0 :
    move-object                      v11, v14
    move-object                      v12, v15
    const/4                          v10, 1
    const/4                          v9, 3
    const/4                          v8, 2
```

Figure 8: Register rearrangement followed by register addition.

6.2. Performance Overhead. We tested whether the proposed scheme is properly applied to real apps in the Google Play Store and evaluated the execution overhead by comparing the launching time of apps with the proposed scheme and apps without it.

Table 3 shows the launching speed of the app before and after applying the proposed scheme to five apps by category in Google Play Store. Experimental results show that the proposed scheme has a delay time of 138 milliseconds on average, although the delay time is different for each app. This delay is caused by the time required to load the split code when the app starts and to check the authenticity of the split code received from the AppContainer. Because each app has different size and functions, its launching time before and after applying the proposed scheme is different. The fastest launching time is 163 milliseconds, and the slowest one is 975 milliseconds. Looking through the experimental results, it can be seen that the overhead due to the proposed scheme increases by about 15% to 2 times. However, the average increase of 138 milliseconds is reasonable, making the launching delay of the proposed scheme acceptable.

```
invoke-virtual      v3, v5, v6, Lcom/example/secrettest/Secret3;->e(Ljava/lang/
new-instance        v4, Lcom/example/secrettest/Secret4;
invoke-direct       v4, Lcom/example/secrettest/Secret4;-><init>()V
const/4             v5, 6
invoke-virtual      v4, v9, v8, v5, Lcom/example/secrettest/Secret4;->sum2(I I
```

⇩

```
invoke-virtual      v3, v5, v6, Lcom/example/secrettest/Secret3;->e(Ljava/lang/
const/4             v4, 0
new-array           v4, v4, [Ljava/lang/Object;
move-object         v14, v4
const-string        v13, 'com.example.secrettest.rStUVNacyAlA'
invoke-static/range v13 … v14, Ledu/ssu/msec/sot/a;->gI(Ljava/lang/String;[java
move-result-object  v13
move-object         v4, v13
check-cast          v4, Lcom/example/secrettest/Secret4;
const/4             v5, 6
invoke-virtual      v4, v9, v8, v5, Lcom/example/secrettest/Secret4;->sum2(I I
```

Figure 9: Caller part of changed method with code splitting.

```
Lcom/example/secrettest
    com/example/secrettest/rStUVNacyALA extends com/example/secrettest/Secret4
        method: <init> () [public constructor] size:4
        method: sum2 (I I I)I [public] size:3
```

Figure 10: Split code stored in IoT devices.

```
sum-BB@0x0 :
    (00000000) add-int          v0, v1, v2
    (00000004) return           v0
```

⇩

```
sum-BB@0x0 :
    const-string        v0, 'com.example.secrettest.FpWcUBZqSvRa'
    const-string        v3, 'sum'
    const/4             v1, 2
    new-array           v1, v1, [Ljava/lang/Object;
    invoke-static       v4, [Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
    move-result-object  v4
    const/4             v2, 0
    aput-object         v4, v1, v2
    invoke-static       v5, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
    move-result-object  v5
    const/4             v2, 1
    aput-object         v5, v1, v2
    invoke-static       v0, v3, v1, Ledu/ssu/msec/sot/a;->gStatic(Ljava/lang/S
    move-result-object  v0
    check-cast          v0, Ljava/lang/Integer;
    invoke-virtual      v0, Ljava/lang/Integer;->intValue()I
    move-result         v0
    return              v0
```

Figure 11: Static method definition part with the proposed scheme.

Table 3: Comparison of launching time.

	Lifestyle apps	Banking apps	Finance apps	Education apps	Test apps
Original app	0.31 (sec)	0.98 (sec)	0.19 (sec)	1.00 (sec)	0.14 (sec)
App with split code	0.44 (sec)	1.14 (sec)	0.32 (sec)	1.13 (sec)	0.28 (sec)

Table 4: Feature comparison between code protection solutions.

	DexProtector	DexGuard	AppContainer
Class protection	Encryption	Encryption	Code splitting
Method protection	Encryption	Encryption	Code splitting
Reversing resistance	Static	Static	Dynamic
Side effect	—	—	Device authentication

6.3. Feature Comparison. Table 4 shows the feature comparison of AppContainer with typical commercial tools for software code protection for Android. The existing tools such as DexGuard [23] and DexProtector [24] adopt encryption to protect methods and classes, but the proposed scheme utilizes code splitting to protect the code without

encryption. Since Android bytecode can automatically recover encrypted code by using advanced dynamic analysis tools [25], the existing tools with encryption can prevent static analysis, but have the disadvantage of being exposed to dynamic analysis. On the other hand, the proposed AppContainer does not expose the complete code even when attempting dynamic analysis of the partial app because a part of the code exists in the external device. Therefore, the proposed scheme can resist dynamic analysis as well as static analysis without applying any encryption techniques. Recall that the split code is physically stored on an external device, and the partial app is stored on a smartphone. In the proposed scheme, since the code works normally only when the pair of partial app and split code must match each other, the app runs normally means that the external device that stores the split code can be trusted. In other words, this means that device authentication is obtained as a side effect.

7. Conclusion

As many apps require personal information, such as smart banking, SNS, and e-mail, the importance of personal information protection is also increasing. However, most users keep their auto-login status by storing their ID and password even though they are apps with sensitive personal information for convenience. Smartphones are protected by various authentication methods such as the PIN, patterns, and biometric information authentication, but they fall short of providing the utmost security of personal information. Thus, we proposed a scheme that protects the app from unauthorized users by assigning control of app execution by merely installing the app without modifying the platform of the smartphone.

The AppContainer is designed to meet the following design goals. First, an app with the proposed scheme requires user authentication before running the app so that unauthorized users cannot run the app itself. The AppContainer is responsible for receiving the split code from the IoT device and communicating it with the app. Therefore, only authenticated users who have a split code on the IoT device can run the app through the AppContainer. Secondly, it can be applied simply as an app-level protection technique rather than a platform modification. Existing protection techniques have enhanced the security by changing the platform of the smartphone, but since the proposed scheme does not require any platform change, it can be used on any platform by any user.

In addition, the AppContainer shows a list of apps with code splitting so that the user can recognize which apps have code-based authentication. Just in case, if the misbehaving app is reinstalled due to a repackaging attack, it is excluded from the list so that users can easily recognize that it is not an existing app. In conclusion, the proposed AppContainer is expected to prevent personal information leakage by effectively avoiding app execution by unauthorized users. As a future work, we intend to expand and develop the proposed scheme by applying the code splitting technique not only to Android but also to various embedded software such as smart vehicles, robots, and drones.

Data Availability

All data generated or analysed during this study are included in this published article.

Disclosure

The authors disclose that this manuscript is an expanded and improved version of the master's thesis [26] by the first author, S. Kim.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported in part by the Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea Government (MSIT) (No. 2017-0-00168, Automatic Deep Malware Analysis Technology for Cyber Threat Intelligence) and in part by the Mid-Career Researcher program through the National Research Foundation of Korea (NRF) funded by the MSIT (Ministry of Science and ICT) under Grant NRF-2020R1A2C2014336.

References

[1] M. Chau and R. Reith, "Smartphone market share," 2020, https://www.idc.com/promo/smartphone-market-share/.

[2] W. Winder, "28 million android phones exposed to "eye-opening" attack risk," 2020, https://www.forbes.com/sites/daveywinder/2019/08/03/28-million-android-phones-exposed-to-eye-opening-attack-risk/#761afc4a7b74.

[3] M. A. Ferrag, L. Maglaras, A. Derhab, and H. Janicke, "Authentication schemes for smart mobile devices: threat models, countermeasures, and open research issues," *Telecommunication Systems*, vol. 73, no. 2, pp. 317–348, 2020.

[4] Z. Lin, W. Meng, W. Li, and D. S. Wong, "Developing cloudbased intelligent touch behavioral authentication on mobile phones," in *Deep Biometrics*, pp. 141–159, Springer, Berlin, Germany, 2020.

[5] A. O. Ekpezu, E. E. Umoh, F. N. Koranteng, and J. A. Abandoh-Sam, "Biometric authentication schemes and methods on mobile devices: a systematic review," in *Modern Theories and Practices for Cyber Ethics and Security Compliance*, W. Yaokumah, M. Rajarajan, J. Abdulai, I. Wiafe, and F. A. Katsriku Eds., IGI Global, Hershey, PA, USA, pp. 172–192, 2020.

[6] Q. Li, P. Dong, and J. Zheng, "Enhancing the security of pattern unlock with surface EMG-based biometrics," *Applied Sciences*, vol. 10, no. 2, p. 541, 2020.

[7] M. Guerar, L. Verderame, A. Merlo, F. Palmieri, M. Migliardi, and L. Vallerini, "CirclePIN: a novel authentication mechanism for smartwatches to prevent unauthorized access to IoT devices," *ACM Transactions on Cyber-Physical Systems*, vol. 4, no. 3, pp. 1–19, 2020.

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

[9] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-armdroid: a rewriting framework for in-app reference monitors

for android applications," *Mobile Security Technologies*, vol. 2012, no. 2, pp. 1–7, 2012.

[10] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: practical policy enforcement for android applications," in *Proceedings of the 21st USENIX Security Symposium*, Bellevue, WA, USA, 2012.

[11] B. Davis and H. Chen, "RetroSkeleton: retrofitting android apps," in *Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, Taipei, Taiwan, 2013.

[12] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard-real-time policy enforcement for third-party applications," Technical report A/02/2012, Saarland University, Saarbrücken, Germany, 2012.

[13] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, "NJAS: sandboxing unmodified applications in non-rooted devices running stock android," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, Denver, CO, USA, 2015.

[14] Go Launcher, 2020, http://www.goforandroid.com/.

[15] ADW Launcher, 2020, http://jbthemes.com/anderweb/.

[16] LauncherPro, 2020, http://www.launcherpro.com/.

[17] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: full-fledged app sandboxing for stock android," in *Proceedings of 24th USENIX Security Symposium*, Washington, DC, USA, 2015.

[18] J. Yoo and J. H. Yi, "Code-based authentication scheme for lightweight integrity checking of smart vehicles," *IEEE Access*, vol. 6, pp. 46731–46741, 2018.

[19] Oracle, "Understanding signning and veification," 2020, https://docs.oracle.com/javase/tutorial/deployment/jar/intor.html.

[20] H. Kim, N. Agrawal, and C. Ungureanu, "Examining storage performance on mobile devices," in *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*, Cascais, Portugal, 2011.

[21] ASMDEX, 2020, http://asm.ow2.org/doc/tutorial-asmdex.html.

[22] Android Open Source Project, 2020, https://source.android.com/index.html.

[23] DexGuard, 2020, https://www.guardsquare.com/en/products/dexguard.

[24] DexProtector, 2020, https://dexprotector.com/.

[25] 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.

[26] S. Kim, "Self-controllable mobile app protection scheme based on binary code splitting," Master degree thesis, Soongsil University, Seoul, Republic of Korea, 2017.