IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 20, 2021, accepted August 13, 2021, date of publication August 24, 2021, date of current version September 3, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3107601

Automated Multi-Layered Bytecode Generation for Preventing Sensitive Information Leaks From Android Applications

GEOCHANG JEON, MINSEONG CHOI, SUNJUN LEE, JEONG HYUN YI, (Member, IEEE), AND HAEHYUN CHO

School of Software, Soongsil University, Seoul 06978, South Korea
Cyber Security Research Center, Soongsil University, Seoul 06978, South Korea

Corresponding author: Haehyun Cho (haehyun@ssu.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) Grant through the Korean Government (MSIT) under Grant NRF-2021R1A4A1029650.

ABSTRACT Sensitive information leakages from applications are a critical issue in the Android ecosystem. Despite the advance of techniques to secure applications such as packing and obfuscation, a lot of applications are still under the threat of repackaging attacks that inject malicious code and re-distribute applications. Also, as we are becoming more dependent on mobile technologies, more sensitive information is used on our mobile devices. Hence, it is of great importance to reduce the risk of such sensitive information leaks. In this paper, we first present a threat model that attempts to leak users' sensitive information by using the repackaging attack, named ReMaCi attack. By analyzing the top 8,546 applications downloaded from Google Play Store, we show that 50% of them are really vulnerable to the ReMaCi attack. We, thus, propose a novel, automated static anti-analysis tool, called AmpDroid, for preventing sensitive information leaks. AmpDroid identifies sensitive dataflows and isolates the code that handles the sensitive data from an application. To demonstrate the effectiveness of AmpDroid, we perform the security and performance evaluation of AmpDroid, comparing it with other obfuscation tools.

INDEX TERMS Sensitive information leaks, obfuscation, android security, code protection.

I. INTRODUCTION

As we are becoming more dependent on mobile technologies, the amount of sensitive information such as user names, phone numbers, e-mail addresses, and credit card numbers used on our mobile devices has been dramatically increased [1], [2]. Not to mention, currently, sensitive information leakages from applications are a critical issue in the Android ecosystem [3], [4]. Therefore, it is of great importance to reduce the risk of such sensitive information leaks. Especially, to reduce the risk, we must secure sensitive data flows in applications [3]. If attackers identified such sensitive data flows in Android applications, they can leak the sensitive data of users by conducting repackaging attacks [5]–[7]. Attackers abuse the repackaging policy of Android that allows an application to re-distribute to third-party markets with a different developer's signature. Therefore, it is possible to inject malicious code leaking confidential information to an application and re-distribute it.

However, despite the advance of techniques to secure applications such as packing and obfuscation from attacks, a lot of applications are still under the threat of repackaging attacks [8]. Especially, such techniques are ineffective when attackers use dynamic analysis methods to reveal sensitive data flows that should not be exposed [9], [10]. It is worth noting that simply encrypting sensitive data cannot prevent its leakages. This is because, by the nature of program execution, encrypted data must be decrypted by the application itself before the data is used. Therefore, if attackers can identify where encrypted sensitive data is decrypted, a sensitive information leak can occur by employing the repackaging attack. The repackaging attack is to redistribute an application to third-party markets after an attacker injects malicious codes to leak sensitive information from the application or to access unpermitted files.

In this paper, we focus on the threat: (1) We first present the threat model, named *Repackaging with Malicious Code*

The associate editor coordinating the review of this manuscript and approving it for publication was Gaurav Somani.

Injected (ReMaCi) attacks, and show that roughly 50% of top-downloaded applications in the Google Play Store are really vulnerable to the attack model. (2) We propose an automated anti-analysis system, named AmpDroid, that generates multi-layered bytecode for preventing the ReMaCi attack. AmpDroid identifies sensitive data flows and isolates code that handles the sensitive data from an application so that attackers cannot analyze the application. The isolated code is separately managed and be provided to the application when it executes.

To demonstrate the effectiveness of AmpDroid, we implement a proof-of-concept of it with sub-applications and thoroughly evaluate AmpDroid. Our evaluation results show that AmpDroid can effectively mitigate the ReMaCi attack and show that AmpDroid imposes a reasonable performance and memory overhead on applications in comparison with other state-of-the-art obfuscation tools (i.e., Liapp [11], Obfuscapk [12], and Dexprotector [13]). In the spirit of open science, we have released the source code of AmpDroid that we developed as part of our research.[1]

In summary, this paper has the following contributions.

- We demonstrate the severity of private information leakage in real-world applications.
- We present an automated multi-layered bytecode generation system that identifies sensitive data flows and prevents sensitive information leaks by isolating code handling the sensitive information.
- We implement the proof-of-concept of AmpDroid and thoroughly evaluate it in comparison with state-of-the-art obfuscation tools.

II. BACKGROUND

In this section, we introduce sensitive data flows in commercial applications and how the sensitive data can be leaked, and look around obfuscation techniques.

A. SENSITIVE DATA FLOW

A data flow is a term to show how data passes from one component to another, through an application. In this work, we define a sensitive data flow as that shows how sensitive information such as credit card number, phone number, e-mail, and International Mobile Equipment Identify (IMEI), is passed among variables or methods. We introduce real-world examples of the sensitive data flow in Android applications that we identified.

Listing 1 shows Smali[2] instructions that we traced by using a dynamic analyzer of a commercial application (Payment teacher manager app in Korea, 1.0.15ver [14]). `Reg5` of the `com.google.gson.Gson.a` method contains not only the message interface between the server with the client but also unencrypted personal information as shown from Line 7 to 17. This private information

```
1  Tid : 18433
2  java.lang.Object com.google.gson.Gson.a(
3    java.lang.String,
4    java.lang.reflect.Type )
5
6    reg4(obj) - com.google.gson.Gson
7    reg5(string) - {
8      "msg":"Success.",
9      "code":"0000",
10     "info"{
11         "member_id":"csec@naver.com",
12         "member_phone":"01012345678",
13         "member_nm":"TestName",
14         "member_email":"csec@naver.com",
15         "use_yn":"Y",
16         "is_exist_partner":"N",
17         "member_idx":"US8A41FFF9" } }
18   reg6(obj) - java.lang.Class
19
20   const/4 v0, #+0
21   new-instance v1, java.io.StringReader //
       type@TypeIndex[5130]
22   invoke-direct {v1, v5}, void java.io.
       StringReader.<init>(java.lang.String) //
       method@22887
23   new-instance v5, com.google.gson.stream.
       JsonReader // type@TypeIndex[1210]
24   invoke-direct {v5, v1}, void com.google.gson.
       stream.JsonReader.<init>(java.io.Reader) //
       method@8044
25 Method Exit
```

Listing 1. Getting user information from server in payment application downloaded from google play store.

including `member_id`, `member_phone`, `member_nm`, and `member_e-mail`, `member_idx` is stored into the internal string object for displaying it on the screen (Line 21 to 24).

Listing 2 shows another Smali instruction that sets the text of the `TextView` with a phone number to display it (same application with Listing 1). `Reg1` containing an unencrypted phone number in Line 8 is transferred as the first argument of the `onTextChanged` function in Line 13.

If attackers identified such sensitive data flows in Android applications, they can leak the sensitive data of users by performing repackaging attacks [5]–[7]. We discuss this attack model in detail (in Section III).

B. OBFUSCATION TECHNIQUES

We introduce obfuscation techniques that make reverse engineering very difficult, which in turn can be used to prevent exposures of sensitive data flows.

1) IDENTIFIER RENAMING

Well-defined identifiers of applications help attackers understand the internal logic of the code and its semantics [15], [16]. Identifier renaming is an obfuscation technique to change identifiers such as package, class, method, and field names to random strings [17], [18].

2) CONTROL FLOW OBFUSCATION

Control flow obfuscation is to alters control flows of an application program without altering its semantics [19], [20].

[1] https://github.com/ssu-csec/code-AmpDroid

[2] Smali is a disassembler implementation for the dex format used by Android runtime (Dalvik virtual machine).

```
Tid : 18583
void v.b.p.z.onTextChanged(
  java.lang.CharSequence,
  int,
  int,
  int )

  reg1(string) - 01012345678
  reg2 - 0
  reg3 - 0
  reg4 - 18

  invoke-super {v0, v1, v2, v3, v4}, void
    android.widget.TextView.onTextChanged(java.
    lang.CharSequence, int, int, int) //
    method@3864
  iget-object v1, v0, Lv/b/p/y; v.b.p.z.d //
    field@21723
  if-eqz v1, +19
  sget-boolean  v2, Z v.i.n.b.a // field@22942
  if-nez v2, +15
  return v0
Method Exit
```

Listing 2. **Updating user phone number in TextView.**

This technique makes it difficult for making static analysis very difficult to determine the original control flows.

3) API HIDING

API hiding is a frequently used obfuscation technique for Java applications for preventing being statically analyzed by hiding which APIs are called by an application. APIs invoked by the application are dynamically determined during the runtime based on the context. Albeit API hiding can effectively hinder a static analysis, we can easily reveal APIs used in applications by directly monitoring instructions executed [21]–[26].

4) DATA ENCRYPTION

Any type of data used in an application including code can be stored as *encrypted* data [27], [28]. Applications using encrypted data must decrypt the data when they actually use it. Therefore, similar to the API hiding technique, encrypted data can effectively impede static analysis methods.

However, unfortunately, all of the above obfuscation techniques are not able to completely protect sensitive data flows from being revealed [29].

III. REMACI ATTACK

In this work, we assume that an attacker attempts to leak sensitive information in an android application through the repackaging attack. The repackaging attack is to re-distribute applications after inserting malicious code to an original application downloaded from application stores such as Google Play Store [30]. We define such attacks for leaking sensitive information as the *Repackaging with Malicious Code Injected (ReMaCi)* attack.

To perform ReMaCi attacks, the attacker needs to inject malicious code that leaks private data such as e-mail or phone numbers to the compromised server after identifying sensitive data flows in applications. Then, the attacker repackages the application and redistributes it. To this end, the attacker first needs to perform dynamic analysis and/or a static analysis to identify sensitive data flows. Once the attacker identified sensitive data flows, it is straightforward to find the exact locations of code that handles the sensitive data. We note that, albeit applications are heavily obfuscated, the attacker can accomplish this step by using a dynamic analyzer that can monitors instructions executed [31], [32]. Therefore, ReMaCi attacks can be applied to any application regardless of the protection mechanisms applied to them. Once the flow of sensitive data was revealed from a dynamic analyzer and/or static analyzer, an attacker injects malicious code that leaks the sensitive data. Finally, the tampered application is *repackaged* with the attacker's private key and *re-distributed* to a third-party app store. Later, if potential victims download and use the application, their personal information such as a phone number will be leaked to the attacker.

In Section VII-A, we demonstrate the severity of the threat by showing real-world applications downloaded from Google Play Store are really vulnerable to the ReMaCi attack.

IV. AMPDROID

We demonstrated that the exposure of sensitive data flows can lead to sensitive data leakages in Section III. Therefore, the flow of sensitive data must be *hidden* to attackers for protecting users' private information. To this end, we segmentalize our goal as follows and propose an automated multi-layered bytecode generation system for preventing sensitive information leaks.

G1: Identifying Sensitive Data flows.
To protect sensitive data flows in applications, we first should identify them. Such data flows can be expressed by using the following three components. The entry method is the start method fetching sensitive data from storage or the network into a process (called a *source*). Next, the end method sends data out of the process (called a *sink*). Furthermore, execution paths between the start method and end method (called *tainted paths*). For the rest of this paper, we call the above three components a *stem*. Our first goal is to find the *stems* in applications. There can be numerous *stems* in an application. The more *stems* we find, the more we can prevent data leakage by hiding them. Also, we should decide which stem will be used to hide among *stems* (called a *selected stem*). We demonstrate how we identify and select *stems* to hide them in Section V-A.

G2: Generating Multi-Layered Bytecode.
The next goal is to generate bytecodes of an application into multi-layered bytecode. The original dex file will be divided into 3 dex files which are the sensitive code Dex (called *SC dex*), all of the other user-defined classes Dex (called *UDC dex*),

and `classes.dex` containing library classes such as `java`, `android`, and `kotlin`. By dividing the original dex file, we can manage them separately. Specifically, if we can hide the SC dex files securely, attackers will not discover sensitive data flows from applications. We discuss how we can achieve this goal in Section V-B.

V. DESIGN

We aim to design an automated anti-analysis mechanism to protect applications against the *ReMaCi attack*. In this section, we introduce the design of our automated multi-layered bytecode generation system, codenamed AmpDroid, that can identify sensitive data flows and prevent information leaks by isolating code that handles sensitive data.

A. IDENTIFICATION OF SENSITIVE DATA FLOWS

1) FINDING THE STEMS

To discover the *stems* mentioned in Section IV, we employ a taint analysis tool, Flowdroid [33]. Flowdroid defines functions that find device-dependent sensitive information such as IMEI and serial numbers and the other internal messages used in an application as a *source*. Flowdroid then defines functions that send sensitive information over the network or save it to storage as a *sink*. Next, Flowdroid finds the sources in the application to check if the application uses sensitive data obtained by using APIs. If Flowdroid discovers a source, it performs a taint analysis to discover execution paths that end up with transferring the sensitive data (e.g., functions sending data over the network) from the source by using the Inter-procedural Control-Flow Graph (ICFG). Identified execution paths include a sequential procedure that sensitive data is passed into variables or functions from source to sink.

Flowdroid is a widely-used static taint analyzer to analyze data flows in an Android application. Tofighi-Shirazi *et al.* [34] demonstrated that Flowdroid with Iccta shows the noticeable accuracy to find information flow in mobile applications compared to other tools. Also, because Flowdriod has been steadily updated by users up to these days, it is quite stable and can be used for applications that run on the most recent version of the Android platform. Therefore, we believe that we can effectively find the stems from applications by employing Flowdroid.

As such, analysis results of Flowdroid are sound, and thus, we can provide a proper defense against ReMaCi attacks by generating multi-layered bytecode based on them.

2) SELECTING A CLASS TO ISOLATE

After discovering the stems, we select a class that will be isolated from an application based on the identified stems where the sensitive data is handled. AmpDroid, currently, supports one class to be isolated from the application. For the effectiveness of the isolation, AmpDroid finds a class that has the most instructions in stems. A stem is a series of instructions from the start of processing for data to the end of sending data outside of an application. Therefore, a stem may consist of instructions over multiple classes in an application.

To choose a class, we analyze an application to find which class has the most instructions in stems. To this end, we statically find the number of instructions contained in stems in each class by comparing instructions between a class and each stem. If more than or equal to two classes have the same number of instructions in stems, AmpDroid picks a class randomly. It is worth noting that even though AmpDroid isolates one class, we can successfully mitigate ReMaCi attacks because an AmpDroid-protected application cannot be statically analyzed and dynamically analyzed (it cannot execute without the isolated class). Specifically, if a class can access sensitive data with few instructions that were not be selected for isolating from the application, the few instructions may be exploited by the ReMaCi attack. To this end, the application must execute for leaking sensitive data, but it conclusively cannot start executing without the isolated class. Consequently, the remaining instructions that use sensitive information cannot be exploited. We note that AmpDroid-protected applications can execute only with the isolated class. For the rest of the paper, we call a class that will be isolated by AmpDroid as *sensitive code*.

B. MULTI-LAYERED BYTECODE GENERATION

`classes.dex` of an application includes user-defined classes, third-party libraries, android platform API, etc. To generate multi-layered bytecode, AmpDroid divides `classes.dex` into three dex files: sensitive code dex (called SC dex), user-defined classes dex (called UDC dex), and the rest code for the `classes.dex`.

1) SC DEX

To prevent *ReMaCi attack*, we hide the sensitive code so that attackers cannot reveal it through reverse engineering applications. To this end, AmpDroid isolates the sensitive code as an SC dex file from the `classes.dex`. The isolated sensitive code will be dynamically loaded and used when an application executes. Also, we need to manage the SC dex to not exist in the application. This is because as far as the SC dex is in an application, attackers are able to monitor the execution of the sensitive code and reveal sensitive data flows even if SC dex was heavily obfuscated.

To isolate the sensitive code, AmpDroid first decompiles the `classes.dex`. It, then, parses decompiled Smali files to obtain and manage all `packages`, `classes`, `methods`, and `fields` defined in the application. Next, AmpDroid finds the sensitive code and stores it in a separate file. Also, AmpDroid deletes the sensitive code from the decompiled Smali files. Furthermore, AmpDroid finds all instructions invoking methods in the sensitive code to patch them.

To be specific, AmpDroid creates a `DexClassLoader` object in the `classes.dex` to dynamically load the SC dex. By using the method, an application dynamically loads the class isolated in the SC dex when it executes. Also, because

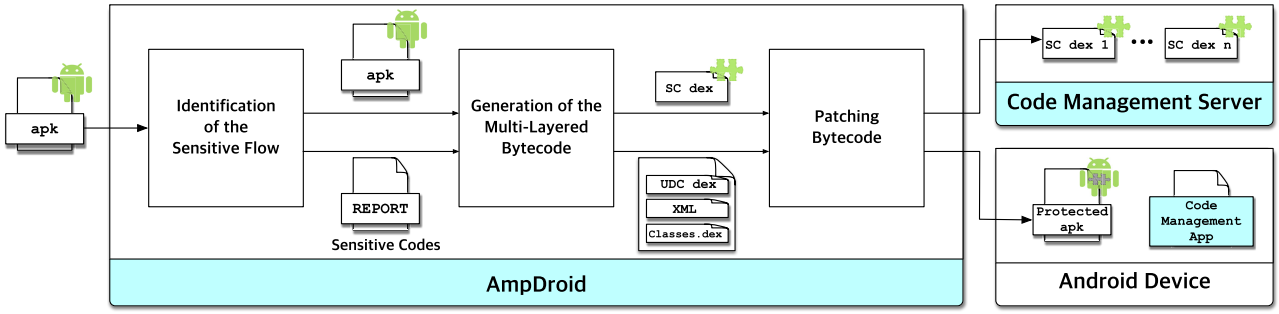

FIGURE 1. Overview of the proposed system.

the sensitive code was separated from the `classes.dex`, we should patch method call sites that invoke methods in the sensitive code as well.

Remaining code in classes.dex has to invoke isolated methods by using the Reflection of Java. To invoke a method with the reflection, we need the procedure to get and access objects and methods to invoke. AmpDroid, thus, must patch all the calling sites invoking methods defined in the isolated class. However, patching all the invoke instructions can increase the size of code unnecessarily. To minimize the redundancy, we design a Generic Reflection Call Method (GRCM) that can invoke any method in the sensitive code with reflection. GRCM is designed to easily provide the procedure to the remaining code without heavily modifying it. GRCM takes the following three arguments: (1) the name of a class object that contains a method to invoke; (2) an array including the name of the method to be called and objects to be passed as parameters when calling the method; and (3) the number of parameters used in the method call. By using GRCM, we can minimize the code redundancy for calling isolated methods. Lastly, AmpDroid re-compiles the Smali files to generate a new `classes.dex`.

2) UDC DEX

After isolating the sensitive code, AmpDroid finds all user-defined classes including `MainActivity` and isolates them into a UDC dex. The AmpDroid uses the package names of Android libraries to distinguish between user-defined classes and android library classes. Android provides libraries such as `java`, `org/apache`, `com/google`, `android`, and `kotlin`, so the other packages can be recognized as user-defined classes and can be split into the UDC dex.

Because an application must calls the `onCreate` method of `MainActivity` in the UDC dex to execute, AmpDroid patches the `classes.dex` to call the method by using the reflection. After the `onCreate` method is executed, the other user-defined methods can be directly invoked.

AmpDroid isolates the UDC dex for making static analyses difficult on user-defined classes. By isolating the UDC dex from classes.dex file, an attacker cannot easily analyze it because the code is separated and hidden even though the file exists in the APK file.

3) CLASSES.DEX

After splitting the sensitive code and all user-defined classes, `classes.dex` has classes of Android libraries. Also, AmpDroid inserted code for loading dynamically other dex files with GRCM in the `classes.dex`.

VI. IMPLEMENTATION

We implement a proof-of-concept of AmpDroid that prevents information leaks by isolating the sensitive code and its sub-applications. Sub-applications are a sensitive code management server (CMS) that saves and provides the SC dex and a sensitive code management application (CMA) that sends a request to the server and provide the SC dex to an AmpDroid-protected application. AmpDroid is currently publicly available at https://github.com/ssu-csec/code-AmpDroid.

A. MANAGING THE SC.DEX

To protect applications against the ReMaCi attack, we should securely manage SC dex files isolated from applications. In this work, we focus on identifying and isolating the sensitive code, leaving the secure management of the sensitive code as future work.

In our proof-of-concept implementation, we used the CMS and CMA to manage the sensitive code. After splitting the sensitive code, AmpDroid calculates a hash value of the application to check its integrity. It, then, sends the SC dex file and the hash value to the code management server to save it and it manages them with pairs. When the code management server receives a request from the CMA for the SC dex, the CMA calculates and sends the application's hash value as well. By doing so, we can check the integrity of the application (whether or not the application is modified). After the CMA receives the sensitive code, the CMA delivers it via `Intent`.

To securely transfer intents between AmpDroid-protected applications and the CMA, every intent is encrypted. To this end, AmpDroid and the CMA use a secret key-based encryption algorithm such as AES to transfer intent securely. When AmpDroid isolates a class from an application, it generates a secret key and stores the key into the application

and the isolated class. When the AmpDroid-protected application requests the SC dex by using an intent to the CMA, the AmpDroid-protected application encrypts the intent with the secret key and sends it. In the case of delivering the SC dex to the AmpDroid-protected application, the CMA also encrypts the intent and then sends it. By doing so, the intent can be delivered securely.

As we described the management of the SC dex, AmpDroid makes AmpDroid-protected applications executable by dynamically loading the SC dex received from the CMA. The CMA provides the SC dex when AmpDroid-protected applications start executing and request it. Once the CMA first downloads the SC dex from the server, CMA can store it on the device. Therefore, CMA does not need to download the same SC dex several times.

B. AMPDROID-PROTECTED APPLICATIONS

The AmpDroid-protected application executes with checking whether the CMA is installed on a device or not, and loads the UDC dex only if the CMA is installed (If not installed, exit the process). After loading the UDC dex, the application sends an intent for obtaining the SC dex to the CMA. The CMA calculates the hash value of the application and provides the SC dex to it. The application, then, can load the SC dex. Once loading the SC dex has completed, the application executes the `onCreate` method of the `MainActivity`.

When calling the `onCreate` method using reflection APIs, AmpDroid-protected applications can mitigate that an attacker resolves the static arguments of reflection APIs by using dynamic analysis tools. The calling `onCreate` method is the next step after the SC dex is dynamically loaded. The attacker without the CMA cannot obtain the SC dex, and thus, the attacker cannot execute the `onCreate` method even if the attacker uses a dynamic analyzer such as Frida.

AmpDroid can also mitigate the collusion attack, two or more applications collaborate to perform stealthy malicious actions, by restricting the execution of the application. To perform the collusion attack, colluded applications must be executed and malicious code which accesses private data must be inserted in colluded applications. But, the application protected by AmpDroid must be able to execute only with SC dex which includes sensitive data flows which can be exploited by an attacker's malicious code. AmpDroid uses also an encryption algorithm to prevent a hijacking when it sends SC dex using an intent that can be exploited by colluded application.

VII. EVALUATION

In this section, we first demonstrate how many recent applications are vulnerable to ReMaCi attacks in Seciton VII-A. We, also, present how effectively AmpDroid protects applications from sensitive information leaks in Section VII-B, Section VII-C, and Section VII-D, comparing with other obfuscation tools. Also, we demonstrate the performance impact of AmpDroid in Section VII-E.

TABLE 1. The number of real-world applications vulnerable to the ReMaCi attack.

The number of applications vulnerable to the ReMaCi attack	
Tested applications number	8,546
Average count of source	6
Average count of sink	60
Average count of tainted paths	26
Applications with more than 1 stem	4,008 (46%)
Applications with more than 100 sinks	2,020 (23%)

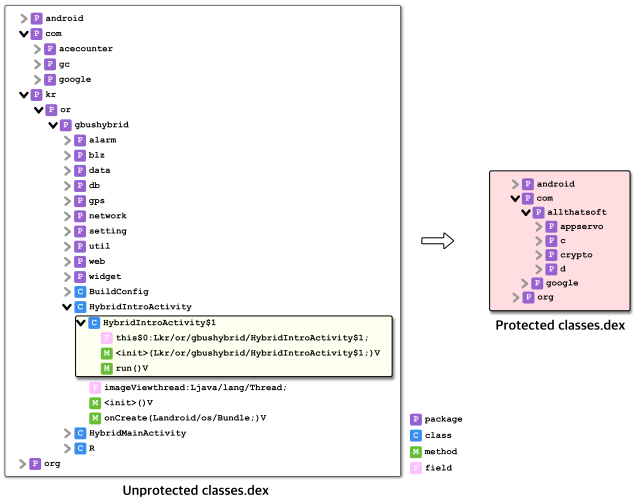

FIGURE 2. Comparison of the class tree of unprotected application (left figure) and protected application by AmpDroid (right figure).

Our evaluation addresses the following research questions:

RQ1: Are real-world applications really vulnerable to the ReMaCi attack?

RQ2: Does AmpDroid effectively prevent ReMaCi attacks?

RQ3: What is the runtime performance overhead imposed by AmpDroid in comparison with other obfuscations tools?

1) EVALUATION SETUP

We collected 8,546 applications from the Google Play Store within the last 2 years and used them for demonstrating the threat of ReMaCi attacks. Our dataset includes quite famous apps that have 1 million+ downloads such as "com.tflat.phatamtienganh.apk", "com.bianf.avatars.couple.dance.apk" to provide an effect of our approach on a variety of applications. The size of our dataset is about 160GB and includes applications from 13KB to 376MB in size and various categories such as Education, Business, Finance, and Lifestyle.[3]

To evaluate the performance of AmpDroid, we used 1,870 random applications (in Section VII-D and Section VII-E) published from 2019 through ApkPure [35].

[3]We provide our dataset to everyone via a public repository at https://github.com/ssu-csec/code-AmpDroid/tree/main/doc

FIGURE 3. SC class tree of protected application by AmpDroid.

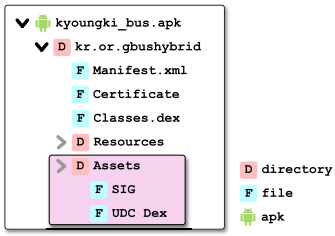

FIGURE 4. File structure of protected application by AmpDroid.

FIGURE 5. UDC class tree of protected application by AmpDroid.

We performed our evaluations on the Google Pixel 2XL device of the android 8.1.0 version. The detailed specifications of the device are composed of a CPU of 2.35GHz x 4 & 1.9GHz x 4 Quad-Core Processor, and 4GB of RAM.

A. REAL-WORLD APPLICATIONS VULNERABLE TO ReMaCi ATTACKS

To demonstrate real-world Android applications are really vulnerable to the ReMaCi attack, we first selected the top-selling 8,546 applications. We then used AmpDroid to find the stems from the applications. Each stem represents an identifiable execution path that accesses the sensitive information and sends the data out of the application. If we can identify such stems in an application, we can inject malicious code that transfers the data to an arbitrary server or other devices in the application and repack the application for distributing it through third-party markets. Therefore, the existence of identifiable stems in an application can imply the application is vulnerable to the ReMaCi attack.

The experimental result in Table 1 shows that around 50% of applications have more than one stem, and thus, they are vulnerable to the ReMaCi attack. Also, 23% of our dataset has more than 100 sinks to transfer sensitive data to outside, and thus, they can be valuable targets from attackers. If an application has a lot of sensitive data flows, the sensitive data can be easily identified and repacked by attackers. In other words, those applications really need proper protection to prevent sensitive information leaks.

The accuracy of identifying stems or sensitive code relies on Flowdroid we employed for implementing AmpDroid. Tofighi-Shirazi *et al.* [34] showed Flowdroid has an accuracy of about 90% and 100% for DroidBench and ICC-bench, which is higher than other taint analysis tools. Therefore, we believe that our identification module would have high accuracy as well. Unfortunately, we do not have the ground truth of how many sensitive data flows exists in our dataset because we downloaded applications from the Google Play Store (we do not have the source code of them), and thus, it is not feasible to evaluate the accuracy of the identification module against the dataset.

B. EFFECTIVENESS OF AMPDROID

To conduct the *ReMaCi attack*, attackers need to discover sensitive data flows via static analysis and/or dynamic analysis. To evaluate how effectively AmpDroid can mitigate the attack, we applied AmpDroid to a real-world application, Kyoungki Bus App, vulnerable to the ReMaCi attack. We, then, performed static and dynamic analyses for the applications to identify sensitive data flows.

1) STATIC ANALYSIS

Figure 2 shows the comparison of class tree of unprotected `classes.dex` and protected `classes.dex` analyzed by JEB [36]. The unprotected class tree contains the `kr/or/gbushybrid/HybridIntroActivity$1` class marked with yellow rectangular box. However, in protected `classes.dex`, the protected `classes.dex` contains only the android and google libraries and custom packages created by AmpDroid as described in Section V-B. Because the sensitive code and UDC dex have been isolated from class.dex, they do not exist in the class.dex.

Figure 2 also illustrates that we cannot only observe the isolated `HybridIntroActivity$1` class but also all the user-defined classes such as `com/acecounter`, `com/gc`, and `kr` as root of selected class. The isolated sensitive code is in the SC dex as shown in Figure 3. On the other hand, Figure 4 shows the file structure of the protected APK, in which there is the UDC dex under the `Assets` folder. AmpDroid makes it difficult to statically analyze the application by isolating user-defined classes from the `classes.dex`. UDC dex has all of the user-defined packages such as `acecounter`, `gc`, and `kr` isolated from the

```
1  Tid : 6831
2  void com.allthatsoft.c.B.<init>()
3
4     reg0(obj) - com.allthatsoft.c.B
5
6     invoke-direct {v0}, void android.app.Application.<init>() // method@77
7     return-void-no-barrier
8
9     • • •
10    const-string v0, "amp" // string@8199
11    invoke-static {v0}, void java.lang.System.loadLibrary(java.lang.String) // method@16081
12
13    • • •
14    iput-object-quick v3, v6, // offset@32
15    const-string v3, "onInitializeFailure" // string@12605
16    invoke-virtual-quick {v0, v3, v1}, // vtable@47
17
18    • • •
19    sput-object v2, Landroid/content/Context; com.allthatsoft.c.B.a // field@4008
20    sput-object v1, Landroid/app/Application; com.allthatsoft.c.B.b // field@4009
21    return-void-no-barrier
22 RETURN-V(void) : VOID
23
24 Method Exit
```

FIGURE 6. Smali instructions of protected application monitored by a dynamic binary instrumentation tool.

original `classes.dex` as shown in the class tree of UDC dex (Figure 5).

2) DYNAMIC ANALYSIS

To demonstrate AmpDroid can protect sensitive data flows from being revealed by attackers through dynamic analyses, we performed a dynamic analysis on the same application used in Section VII-B1. To this end, we employed a dynamic binary instrumentation tool (DBI) for Android applications proposed by Wong and Lie *et al.* [37], by which we can directly monitor Smali instructions executed.

Figure 6 shows Smali code extracted by the DBI when the application starts executing. The application executes its `<init>` function in the `com.allthatsoft.c.B` class in Line 2. Then, the application loads a native library that checks whether the code management application was installed or not on the device at the native level in Line 10 to 11. Thus, the DBI tool cannot monitor the checking process because the module is implemented with the native code. This makes it difficult to infer the behaviors of the application. If the code management application has not been installed on the device, the library returns false and the application fails to initialize. In the log, it can only see the callback method of the checking process received from the library in Line 15 to 16. Finally, the application calls the `onInitializeFailure` function and finishes executing.

C. IDENTIFIED SENSITIVE DATA FLOWS

We demonstrate that how AmpDroid identifies the sensitive data flows in an application and then finds a class that has the most instructions among the identified data flows as described in Section V-A. To this end, we used twenty applications and count the Class, Method, and Instruction identified by AmpDroid as the sensitive data flows and the most instruction class to isolate.

Section 2 shows the experimental result. AmpDroid found that each application has 15 sensitive classes, 38 sensitive methods, and 1,393 sensitive instructions on average. Then, AmpDroid selected a class, which has the most instructions, including 3 methods and 291 instructions on average. Finally, AmpDroid isolates them from applications.

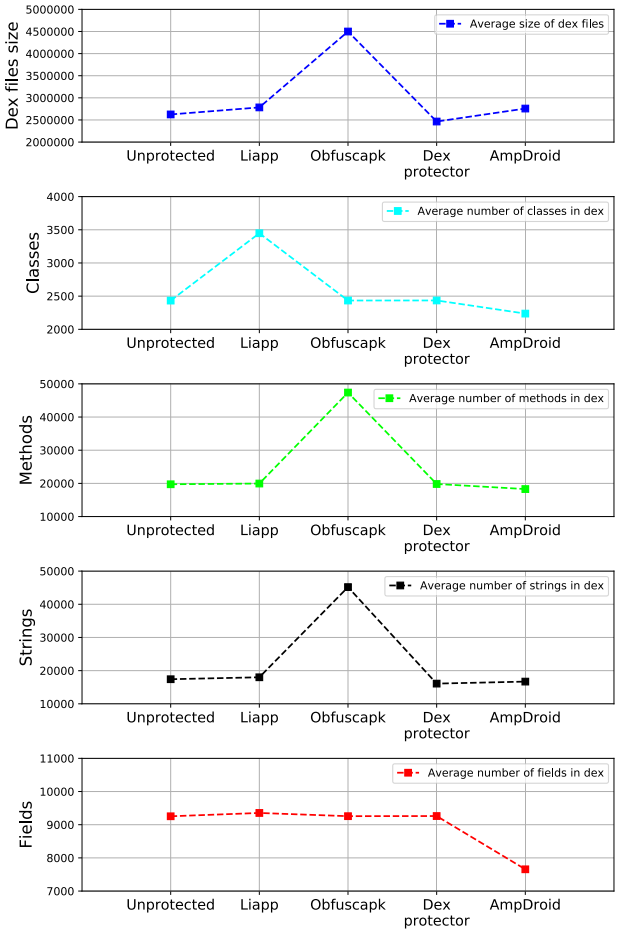

FIGURE 7. Average numbers of identifiers in dex files with 1,870 applications between obfuscation tools.

In Table 2, the Rate column means a rate that the instructions of the selected class occupy in the identified sensitive data flows. AmpDroid selected the most instruction class taking 32.74% of sensitive data flows on average.

D. RECOGNIZABLE IDENTIFIERS

We show that recognizable identifiers in `classes.dex` protected by AmpDroid, comparing with other obfuscation tools. To this end, we used the `CallIndirection` and `Reflection` of obfuscapk [12] and the `code protection` of Liapp [11] and the `class encryption` of Dexprotector [13]. Then, we used a static analysis tool, AndroGuard [38], to extract identifiers from `classes.dex` files protected by AmpDroid and the other obfuscation tools.

Figure 7 illustrates the evaluation result where we can find the number of recognizable identifiers (Dex size, Class, Method, String, Field) between the `classes.dex` files. The `classes.dex` file protected by AmpDroid has the lowest number of recognizable identifiers with Class, Methods, and Field as shown in Figure 7 because of the isolated class. Having lower identifiers implies that AmpDroid

TABLE 2. Identified sensitive data flows and the most instructions class to isolate targeting. We used twenty applications to measure how many sensitive data flows exist in applications and in a class that is going to be isolated by AmpDroid.

APK Name	Identificated Sensitive Data Flows			Selected Class to Isolate				Application Total Information		
	Classes	Methods	Instructions	Class Name	Methods	Instructions	Rate	Classes	Methods	Size (MB)
com.bms.romanceg.apk	60	189	5,536	dummyMainClass	8	736	13.29%	1,243	11,971	1
com.bms.xdfyangchung.apk	27	57	1,845	shelfEbdManager	1	168	9.11%	219	1,699	2
com.bns.bible.lite.KBSBible.apk	15	28	196	CompareActivity	2	21	10.71%	4,229	30,970	28
com.boneit.android.telink050.apk	2	3	11	b	2	7	63.64%	4,523	35,755	11
com.book.rmxs.apk	6	14	88	OI	5	13	14.77%	7,453	63,283	14
com.daimler.mbdcatalog.android.apk	19	53	5,762	LibraryActivity	11	1,893	32.85%	4,200	29,905	13
com.grandstream.wave.apk	29	66	3,005	ContactsListFragment	8	138	4.59%	6,579	55,760	17
com.haenam.gunmin.apk	2	4	22	MainActivity	2	11	50.00%	2,681	22,124	1
com.pointbank.rimansale.apk	11	20	786	u	1	150	19.08%	966	8,889	1
com.sac.app.apk	1	1	13	a	1	13	100.00%	182	981	1
com.sds.square.sso.agent.apk	19	27	796	b	1	98	12.31%	1,301	10,472	1
com.semaphore.bookplayer.apk	18	45	2,157	ActChangePwd	3	659	30.55%	6,573	56,435	52
com.slksolution.carauction.apk	11	18	792	x	3	174	21.97%	1,194	10,868	1
com.socialnmobile.colordict.apk	13	34	466	a	5	140	30.04%	2,401	16,012	1
com.sromai.sromailibrary.apk	13	33	2,511	RequestParams	3	499	19.87%	6,376	51,433	21
com.usefultools.electricstungun.igun.apk	2	2	22	SettingsActivity	1	11	50.00%	397	4,464	1
kgm.check.apk	44	105	1,333	DB	2	76	5.70%	1,731	18,236	1
kr.webadsky.daema.apk	1	2	11	Statics	2	11	100.00%	1,352	15,287	1
lcms.or.kr.angel.apk	9	22	326	TwilightManager	4	76	23.31%	1,589	16,789	1
ru.lithiums.callsblockerplus.apk	15	53	2,200	DBDataSource	4	945	42.95%	8,321	64,323	4
Average	15	38	1,393		3	291	32.74%	3,175	26,282	8

can protect applications better than the other tools against ReMaCi attacks from being statically analyzed.

E. RUNTIME PERFORMANCE EVALUATION

We evaluated the runtime performance of AmpDroid with 1,870 applications, comparing it with the performance impacts of the other obfuscation tools. Specifically, we first measured loading times of the applications until the `MainActivity` executes because the biggest performance impact of using obfuscation tools and AmpDroid is when an application is being loaded. Secondly, we evaluated the CPU and memory usages of loading times by using Simpleperf [39] and Dumpsys [40]. To this end, we measured not only the main process's CPU and memory usages, but also one of the sub-processes to reasonably conduct the evaluation.

Figure 8 shows the evaluation result of loading times between AmpDroid and the other obfuscations tools. The average loading time of applications protected by AmpDroid is lower than Liapp but higher than the other obfuscation tools. This is because AmpDroid requires additional processes for dynamically loading the UDC dex and SC dex. Also, our proof-of-concept implementation of AmpDroid requires the communication process for requesting and receiving the SC dex from the code management application.

Next, Figure 9 shows the overhead of CPU and memory usages while the applications are loaded. In Figure 9–(a), CPU Cycles refer to the number of cycles to process instructions from the CPU. Instructions refer to the number of instructions executed from the CPU. Task-clock refers to the usage time of the CPU. Context-switch refers to the time used to replace the CPU's register values. In Figure 9–(B), Private total refers to all memory sizes allocated in the process. Private dirty refers to the used memory. Private clean refers to the memory size allocated to the process, excluding dirty. SwapPss Dirty refers to the swapped size to obtain memory.

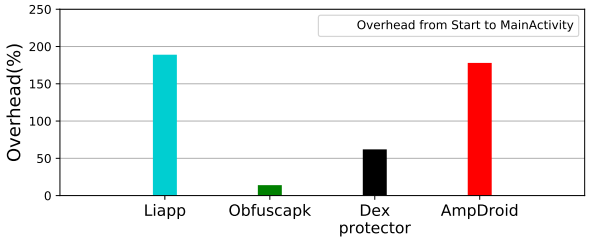

FIGURE 8. Overhead of the load time from start to calling MainActivity between obfuscation tools.

On average, for loading applications, applications protected by AmpDroid used almost 170% of CPU cycles and 300% of instructions more than unprotected ones. Also, AmpDroid-protected applications showed the highest CPU usages among the obfuscation tools. We note that after loading an application has been completed, the performance overhead of AmpDroid-protected applications is nearly zero because no additional task is required to execute them. While the memory overhead of AmpDroid-protected applications is 16% higher than unprotected applications. However, the memory overhead of applications protected by AmpDroid lower than ones obfuscated by the other tools.

These performance evaluation results demonstrate that AmpDroid takes the second most overhead of the load time and shows the highest CPU usages but the lowest memory usages among the protection tools.

F. LIMITATION

AmpDroid relies on the static analysis result for selecting the sensitive code to isolate from an application. As a result, if our static analyzer fails to analyze sensitive data flows, AmpDroid cannot provide a proper defense to an application. Unfortunately, in our experiments using the real-world dataset, we cannot analyze 50% of applications and cannot start to execute 30% of applications after protecting them

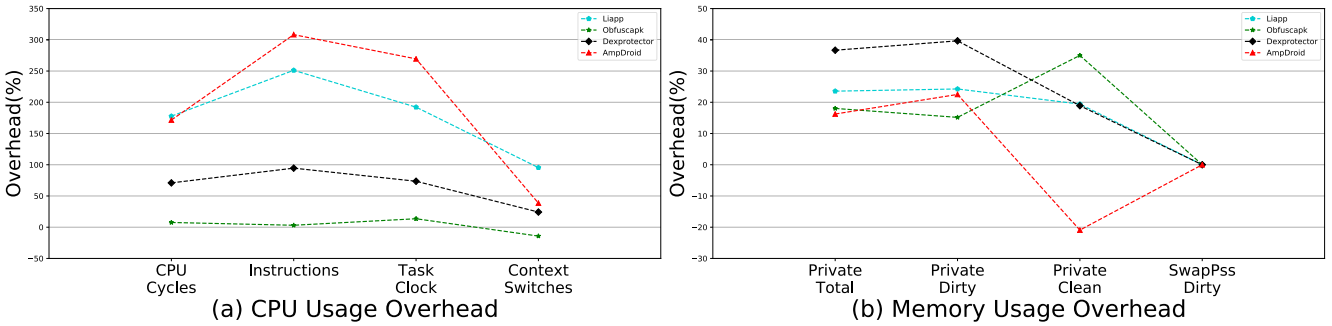

FIGURE 9. The runtime average overhead of the usage of CPU (a) and memory (b) between obfuscation tools.

by using AmpDroid. Our dataset consists of applications downloaded from the Google Play Store within the last two years and most applications in our dataset were protected by various obfuscation schemes such as class encryption and string encryption. Therefore, AmpDroid failed to analyze around 50% of our dataset and to start to execute 30% of our dataset. To protect sensitive data on unanalyzable applications, we must identify sensitive data flows on them even though their codes are strongly obfuscated. But, we note that statically analyzing obfuscated applications is a challenging problem and leave this problem as our future work. However, except for such cases, AmpDroid can be generally used for protecting the application.

VIII. RELATED WORK

We proposed AmpDroid, a system for preventing sensitive information leaks for Android applications. In this section, we introduce other protection mechanisms that can be used against the ReMaCi attack.

Obfuscapk [12], which obfuscates bytecode to difficult the understanding of code stream such as our system, is a static obfuscation tool and proposes various functions such as `CallIndirection` and `LibEncryption` to secure the code. Also, they transform the identifier such as package, class, and method name to confuse the understanding of the code. Further, they insert a new method that invokes the original method (`IndirectionCall`) and supports encryption to protect library and asset files. It's effective protection on static analysis. However, at runtime, decrypted files are loaded in memory and can be tracked from a dynamic analyzer.

Dexpro [41] can confuse the flow of register data and combines opaque predicates to make it difficult to understand the control flow. Fundamentally, the control flow is obfuscated by inserting dummy code or by switching codes. Liapp [11], which is used as a commercial tool, is also a static obfuscation tool that receives an APK file as input. It not only protects the source code against anti-analysis but also supports anti-repackage so that the app cannot be decompiled. Although Liapp's code protection hides the class in a specific location, when the binary is loaded on memory, the flow of sensitive data can be monitored from the dynamic analyzer (see Appendix).

Another commercial tool, Dexprotector [13], provides code protection such as class encryption, string encryption, and API Hiding. Dexprotector uses encryption to protect bytecode such as Obfuscapk. Our protection scheme has differed from Dexprotector in that it identifies the flow of sensitive data and isolates the code, not encryption.

Zhang *et al.* [42] proposes the application code protection scheme that acquires a memory area where the binary is loaded and relocates it to a special area. Since it is relocated into the kernel area, the only user who can access the kernel is permitted to control the memory of the application. It is difficult to trace or penetrate because the data is hidden in the kernel. However, to apply this approach, the kernel of the Android device must be customized and it is difficult to use it in general.

App Guardian [43] is a technique that provides immediate protection when an app is installed and does not need to change the operating system or target app. It is similar to our system in a point of achieving the goal of preventing sensitive information leaks. However, App Guardian focuses on finding some suspicious background processes to block gathering runtime information of the target app. It protects the runtime information gathering of the target app by pausing all suspicious background processes when the target app is running.

On the other hand, Zhao *et al.* [44] introduced the Virtual Machine (VM)-based MultiDex and share object (SO) protection approach. They used the newly stack-based native code system to protect apps from reverse engineering techniques. They prevent reverse engineering by executing the app on different execution tracks every time. The key is to prevent dynamic cumulative attacks by running different VMs each time.

IX. CONCLUSION

In this paper, we first assess real-world Android applications to illustrate that are really vulnerable to attacks for leaking sensitive information. We, then, proposed a novel system, AmpDroid, that can prevent sensitive information leaks by

```
1  Tid : 21174
2  void com.example.sensitivecode.UIMain.onClick(
       android.view.View)
3  reg4(obj) - com.example.sensitivecode.UIMain
4  reg5(obj) - androidx.appcompat.widget.
       AppCompatButton
5  invoke-virtual-quick {v5}, // vtable@223
6  move-result v5
7  constv0, #+2131165218
8  if-nev5, v0, +57
9  sget-object v5, Landroid/widget/EditText; com.
       example.sensitivecode.UIMain.etID //
       field@11389
10 invoke-virtual-quick {v5}, // vtable@1061
11 move-result-object v5
12 invoke-virtual-quick {v5}, // vtable@7
13 move-result-object v5
14 sget-object v0, Landroid/widget/EditText; com.
       example.sensitivecode.UIMain.etPWD //
       field@11390
15 invoke-virtual-quick {v0}, // vtable@1061
16 move-result-object v0
17 invoke-virtual-quick {v0}, // vtable@7
18 move-result-object v0
19 sget-object v1, Lcom/example/sensitivecode/
       MainActivity; com.example.sensitivecode.UIMain
       .main // field@11391
20 new-instance v2, java.lang.StringBuilder //
       type@TypeIndex[1976]
21 invoke-direct {v2}, voidjava.lang.StringBuilder.<
       init>() // method@15724
22 const-string v3, "id: " // string@11184
23 invoke-virtual-quick {v2, v3}, // vtable@77
24 invoke-virtual-quick {v2, v5}, // vtable@77
25 const-string v5, " pwd: " // string@215
26 invoke-virtual-quick {v2, v5}, // vtable@77
27 invoke-virtual-quick {v2, v0}, // vtable@77
28 invoke-virtual-quick {v2}, // vtable@7
29 move-result-object v5
30 const/4 v0, #+0
31 invoke-static {v1, v5, v0}, android.widget.
       Toastandroid.widget.Toast.makeText(android.
       content.Context, java.lang.CharSequence, int)
       // method@3058
32 move-result-object v5
33 invoke-virtual {v5}, void android.widget.Toast.
       show() // method@7
34 return-void
35 RETURN-V(void) : VOID
```

Listing 3. Monitoring the obfuscated code of Liapp.

generating Multi-Layered Bytecode. AmpDroid selects the sensitive code by identifying sensitive dataflows and isolates the code. We demonstrated the effectiveness of our system. In addition, we show that our system can be used for Android applications.

APPENDIX

A. THE TRACKING OF THE OBFUSCATED CODE BY DYNAMIC ANALYZER

The obfuscation technique that makes it difficult to understand the code is one of the enhancing security of an application. Obfuscation can effectively prevent static analysis, but it can still be traced against dynamic analysis can monitors executable code. We used the Lipp and Obfuscapk tools to show that.

```
1  Tid : 23549
2  void com.example.sensitivecode.UIMain.onClick(
       android.view.View)
3   ...
4  sget-object v5, Landroid/widget/EditText; com.
       example.sensitivecode.UIMain.etID //
       field@11390
5  invoke-static {v5}, android.text.Editable com.
       example.sensitivecode.UIMain.NImOqzKvYLLigOdV(
       android.widget.EditText) // method@45997
6  Tid : 23549
7  android.text.Editable com.example.sensitivecode.
       UIMain.NImOqzKvYLLigOdV(android.widget.
       EditText)
8  invoke-virtual-quick {v1}, // vtable@1061
9  Tid : 23549
10 android.text.Editable androidx.appcompat.widget.
       AppCompatEditText.getText()
11 ...
12 invoke-static {v2}, android.text.Editable androidx
       .appcompat.widget.AppCompatEditText.
       ZZtAqVSOzJsVoDXM(android.widget.EditText) //
       method@11515
13 Tid : 23549
14 android.text.Editable androidx.appcompat.widget.
       AppCompatEditText.ZZtAqVSOzJsVoDXM(android.
       widget.EditText)
15 ...
16 invoke-super {v1}, android.text.Editable android.
       widget.EditText.getEditableText() //
       method@2600
17 RETURN-V(object) : Landroid/text/Editable;
18 move-result-object v5
19 ...
20 invoke-static {v1, v5, v0}, android.widget.Toast
       com.example.sensitivecode.UIMain.
       AQDoDorjlPfwHwkX(android.content.Context, java
       .lang.CharSequence, int) // method@45994
21 Tid : 23549
22 android.widget.Toast com.example.sensitivecode.
       UIMain.AQDoDorjlPfwHwkX(android.content.
       Context, java.lang.CharSequence, int)
23 ...
24 invoke-static {v1, v2, v3}, android.widget.Toast
       android.widget.Toast.makeText(android.content.
       Context, java.lang.CharSequence, int) //
       method@3072
25 ...
26 invoke-static {v5}, void com.example.sensitivecode
       .UIMain.EJdjYBXhzSyuFCDO(android.widget.Toast)
        // method@45995
27 ...
28 Tid : 23549
29 void com.example.sensitivecode.UIMain.
       EJdjYBXhzSyuFCDO(android.widget.Toast)
30 invoke-virtual {v5}, void android.widget.Toast.
       show() // method@7
```

Listing 4. Monitoring the obfuscated code of Obfuscapk.

1) LIAPP

Listing 3 shows the execution log that a sample of Liapp (commercial obfuscation tool) applied code protection was dumped by a dynamic analyzer. The dynamically measured code is part of the `onClick`. Line 9 shows the `etID` of the `EditText` is moved to `v5`. And, the `etPWD` of the `EditText` is moved to `v0` in Line 14. The value of `etID` and `etPWD` is copied to `v5` through Line 20 to Line 29. The user account transformed with String is passed as a second argument of `Toast`'s `makeText` function in Line 31.

Finally, we can see that the ID and password are sinking to the outside through `Toast`'s `show`. The codes executed at the runtime of Liapp are traceable because plain text is exposed.

2) OBFUSCAPK

Listing 4 shows the execution code of another tool, Obfuscapk's `CallIndirection` and `Reflection` function. The id entered by the user is transferred to `v5` in Line 4. Line 5 shows that the method name is replaced with `NImOqzKvYLLigOdV` due to the `CallIndirection` of Obfuscapk. This method cast the type to String over multiple function calls. The `v5` (user-id) having the result of casting is passed via the `CallIndirection` of Obfuscapk. Finally, the user-id can be tracked by a dynamic analyzer that is sunk to the outside via the `Show` method of `Toast` Object.

The above two samples show that the flow of sensitive data can be tracked by a dynamic binary instrumentation tool. Obfuscation, which makes the code difficult to understand, still has the limit that the original executable code can be dumped, and the monitored log can be used for tracking the flow of sensitive data. Thus there remains still a potential risk of sensitive data leaks.

REFERENCES

[1] R. Salvia, A. Cortesi, P. Ferrara, and F. Spoto, "Intents analysis of Android apps for confidentiality leakage detection," in *Advanced Computing and Systems for Security*. Singapore: Springer, 2021, pp. 43–65.

[2] N. Williams, "Persistent private information," *Econometrica*, vol. 79, no. 4, pp. 1233–1275, 2011.

[3] *OWASP Mobile Top 10*, Accessed: Feb. 25, 2021. [Online]. Available: https://owasp.org/www-project-mobile-top-10

[4] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *Proc. Int. Conf. Trust Trustworthy Comput.* Berlin, Germany: Springer, 2012, pp. 291–307.

[5] J.-H. Jung, J. Y. Kim, H.-C. Lee, and J. H. Yi, "Repackaging attack on Android banking applications and its countermeasures," *Wireless Pers. Commun.*, vol. 73, no. 4, pp. 1421–1437, Dec. 2013.

[6] Y. Lee, S. Woo, J. Lee, Y. Song, H. Moon, and D. H. Lee, "Enhanced Android app-repackaging attack on in-vehicle network," *Wireless Commun. Mobile Comput.*, vol. 2019, pp. 1–13, Feb. 2019.

[7] Z. He, G. Ye, L. Yuan, Z. Tang, X. Wang, J. Ren, W. Wang, J. Yang, D. Fang, and Z. Wang, "Exploiting binary-level code virtualization to protect Android applications against app repackaging," *IEEE Access*, vol. 7, pp. 115062–115074, 2019.

[8] S. Liu, R. Guo, B. Zhao, T. Chen, and M. Zhang, "APPCorp: A corpus for Android privacy policy document structure analysis," 2020, *arXiv:2005.06945*. [Online]. Available: http://arxiv.org/abs/2005.06945

[9] L. Xue, H. Zhou, X. Luo, Y. Zhou, Y. Shi, G. Gu, F. Zhang, and M. H. Au, "Happer: Unpacking Android apps via a hardware-assisted approach," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, May 2021, pp. 1641–1658.

[10] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 674–691.

[11] *Liapp*, Accessed: Feb. 25, 2021. [Online]. Available: https://liapp.lockincomp.com

[12] S. Aonzo, G. C. Georgiu, L. Verderame, and A. Merlo, "Obfuscapk: An open-source black-box obfuscation tool for Android apps," *SoftwareX*, vol. 11, Jan. 2020, Art. no. 100403.

[13] *Dexprotector*. Accessed: Feb. 25, 2021. [Online]. Available: https://dexprotector.com

[14] *payment Manager App*. Accessed: Jun. 12, 2021. [Online]. Available: https://play.google.com/store/apps/details?id=com.paymint.payssam_aos_manager&hl=ko&gl=US

[15] R. Baumann, M. Protsenko, and T. Müller, "Anti-ProGuard: Towards automated deobfuscation of Android apps," in *Proc. 4th Workshop Secur. Highly Connected IT Syst. (SHCIS)*, 2017, pp. 7–12.

[16] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "TriggerScope: Towards detecting logic bombs in Android applications," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 377–396.

[17] R. Tofighi-Shirazi, M. Christofi, P. Elbaz-Vincent, and T.-H. Le, "DoSE: Deobfuscation based on semantic equivalence," in *Proc. 8th Softw. Secur., Protection, Reverse Eng. Workshop (SSPREW)*, 2018, pp. 1–12.

[18] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of Android applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 343–355.

[19] Z. Wang, Y. Shan, Z. Yang, R. Wang, and S. Song, "Semantic redirection obfuscation: A control flow obfuscation based on Android runtime," in *Proc. IEEE 19th Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Dec. 2020, pp. 1756–1763.

[20] V. Balachandran, D. J. J. Tan, and V. L. L. Thing, "Control flow obfuscation for Android applications," *Comput. Secur.*, vol. 61, pp. 72–93, Aug. 2016.

[21] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[22] Y. Zhang, X. Luo, and H. Yin, "Dexhunter: Toward extracting hidden code from packed Android applications," in *Proc. Eur. Symp. Res. Comput. Secur.* Cham, Switzerland: Springer, 2015, pp. 293–311.

[23] M. Y. Wong and D. Lie, "IntelliDroid: A targeted input generator for the dynamic analysis of Android malware," in *Proc. NDSS*, vol. 16, 2016, pp. 21–24.

[24] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 1–29, 2014.

[25] L. Xue, H. Zhou, X. Luo, L. Yu, D. Wu, Y. Zhou, and X. Ma, "PackerGrind: An adaptive unpacking system for Android apps," *IEEE Trans. Softw. Eng.*, early access, May 21, 2020, doi: 10.1109/TSE.2020.2996433.

[26] C. Sun, H. Zhang, S. Qin, J. Qin, Y. Shi, and Q. Wen, "DroidPDF: The obfuscation resilient packer detection framework for Android apps," *IEEE Access*, vol. 8, pp. 167460–167474, 2020.

[27] S. Inshi, R. Chowdhury, M. Elarbi, H. Ould-Slimane, and C. Talhi, "LCA-ABE: Lightweight context-aware encryption for Android applications," in *Proc. Int. Symp. Netw., Comput. Commun. (ISNCC)*, Oct. 2020, pp. 1–6.

[28] A. Skillen, D. Barrera, and P. C. van Oorschot, "Deadbolt: Locking down Android disk encryption," in *Proc. 3rd ACM Workshop Secur. Privacy Smartphones Mobile Devices (SPSM)*, 2013, pp. 3–14.

[29] 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.

[30] *Google Play Store*, Accessed: Feb. 25, 2021. [Online]. Available: https://play.google.com/store

[31] V. Costamagna and C. Zheng, "Artdroid: A virtual-method hooking framework on Android art runtime," in *Proc. IMPS@ ESSoS*, 2016, pp. 20–28.

[32] *Frida*. Accessed: Feb. 25, 2021. [Online]. Available: https://frida.re

[33] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[34] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2018, pp. 176–186.

[35] *Apkpure*. Accessed: Feb. 25, 2021. [Online]. Available: https://apkpure.com/kr

[36] *JEB*. Accessed: Feb. 25, 2021. [Online]. Available: https://www.pnfsoftware.com

[37] S. Lee, "Bytecode instrumentaion scheme for evading anti-analysis techniques based on method hooking," M.S. thesis, Soongsil Univ., Seoul, South Korea, 2021.

[38] *Androguard*. Accessed: Feb. 25, 2021. [Online]. Available: https://androguard.readthedocs.io

[39] *Simpleperf*. Accessed: Feb. 25, 2021. [Online]. Available: https://developer.android.com/ndk/guides/simpleperf

[40] Google. *Dumpsys*. Accessed: Feb. 25, 2021, available:. [Online]. Available: https://developer.android.com/studio/command-line/dumpsys

[41] B. Zhao, Z. Tang, Z. Li, L. Song, X. Gong, D. Fang, F. Liu, and Z. Wang, "Dexpro: A bytecode level code protection system for Android applications," in *Proc. Int. Symp. Cyberspace Saf. Secur.* Cham, Switzerland: Springer, 2017, pp. 367–382.
[42] X. Zhang, Y.-A. Tan, C. Zhang, Y. Xue, Y. Li, and J. Zheng, "A code protection scheme by process memory relocation for Android devices," *Multimedia Tools Appl.*, vol. 77, no. 9, pp. 11137–11157, May 2018.
[43] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang, "Leave me alone: App-level protection against runtime information gathering on Android," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 915–930.
[44] M. A. R. Khan and M. K. Jain, "Protection Android app with multiDEX and SO files from reverse engineering," *Mater. Today, Proc.*, Jan. 2021.

GEOCHANG JEON received the B.S. degree in computer science from Myongji University, Seoul, South Korea, in 2015. He is currently pursuing the master's degree with the School of Software, Soongsil University. He was a Junior Researcher at the Network Laboratory, Telefield Company, South Korea, from 2018 to 2019. His research interests include android security, embedded systems, network security, and system software security.

MINSEONG CHOI received the B.S. degree in computer science from Soongsil University, in 2019, where he is currently pursuing the master's degree with the School of Software. His research interests include mobile security and system security.

SUNJUN LEE received the B.S. and M.S. degrees in computer science and engineering from Soongsil University, in 2019 and 2021, respectively. He is currently a Research Staff with the Cyber Security Research Center. His research interests include binary analysis, reverse engineering, system security, and mobile security.

JEONG HYUN YI (Member, IEEE) received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, South Korea, in 1993 and 1995, respectively, and the Ph.D. degree in information and computer science from the University of California at Irvine, Irvine, in 2005. He was a Principal Researcher at Samsung Advanced Institute of Technology, South Korea, from 2005 to 2008, and a member of Research Staff with the Electronics and Telecommunications Research Institute (ETRI), South Korea, from 1995 to 2001. From 2000 to 2001, he was a Guest Researcher with the National Institute of Standards and Technology (NIST), MD, USA. He is currently a Professor with the School of Software and the Director of the Cyber Security Research Center, Soongsil University. His research interests include mobile security and privacy, the IoT security, and applied cryptography.

HAEHYUN CHO received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, South Korea, in 2013 and 2015, respectively, and the Ph.D. degree from the School of Computing, Informatics and Decision Systems Engineering of Arizona State University, majoring in computer science, and especially concentrating on information assurance. He is currently an Assistant Professor with the School of Software and the Co-Director of the Cyber Security Research Center, Soongsil University. His research interests include the field of systems security, which is to address and discover security concerns stemmed from insecure designs and implementations. He is passionate about analyzing, finding, and resolving security issues in a wide range of topics.