Deoptfuscator: Defeating Advanced Control-flow Obfuscation Using ART (Android Runtime)

GEUNHA YOU\textsuperscript{1}, GYOOSIK KIM\textsuperscript{2}, SANGCHUL HAN\textsuperscript{3}, MINKYU PARK\textsuperscript{3}, AND SEONG-JE CHO\textsuperscript{4}

\textbf{ABSTRACT} Code obfuscation is a technique that makes it difficult for code analyzers to understand a program by transforming its structures or operations while maintaining its original functionality. Android app developers often employ obfuscation techniques to protect business logic and core algorithm inside their app against reverse engineering attacks. On the other hand, malicious app writers also use obfuscation techniques to avoid being detected by anti-malware software. If malware analysts can mitigate the code obfuscation applied to malicious apps, they can analyze and detect the malicious apps more efficiently. This paper proposes a new tool, \textit{Deoptfuscator}, to detect obfuscated an Android app and to restore the original source codes. \textit{Deoptfuscator} detects an app control-flow obfuscated by \textit{DexGuard} and tries to restore the original control-flows. \textit{Deoptfuscator} deobfuscates in two steps: it determines whether an control-flow obfuscation technique is applied and then deobfuscates the obfuscated codes. Through experiments, we analyze how similar a deobfuscated app is to the original one and show that the obfuscated app can be effectively restored to the one similar to the original. We also show that the deobfuscated apps run normally.

\textbf{INDEX TERMS} Android app, Malicious app, Obfuscation, Deobfuscation, Control-flow Obfuscation.

\section*{I. INTRODUCTION}

The Android Operating System occupies 71.9\% of the smartphone operating system (OS) market as of February 2021, and the number of apps that appeared in Google Play Store, the official Google app market, records 2.97 million as of December 2020 [1], [2]. According to the increased availability of smartphones, various mobile services such as SNS, streaming, banking, shopping, or healthcare comprise a mobile ecosystem, and people frequently use these services.

This situation furnishes that mobile apps in those services increasingly handle users’ credit cards or private/sensitive information. Simultaneously, the number of malicious apps that hack/steal such sensitive information are rising continuously [3]–[8]. For example, Tang et al. [9] (1) described an attack that could steal sensitive information by connecting URL links in smartphones where a malicious instant app was installed, and (2) proposed a tool called \textit{MIAFinder} which detected vulnerabilities that could be exploited by the malicious instant app. \textit{MIAFinder} collected 400,000 apps, 200,000 from both the Google Play Store and Tencent Myapp respectively, and showed that 228,207 among the 400,000 apps were vulnerable to attacks by the malicious instant app.

Because Android apps can be decompiled easily, app developers use obfuscation techniques to protect the app’s business logic, internal structure, and code that handles sensitive data. Code obfuscation refers to a technique that increases the cost of program analysis such as reverse engineering by transforming the control-flows and data structures or identifiers of the program while preserving its original semantics and behaviors.

On the other hand, malware authors also apply obfuscation to avoid malware detection [10]–[17]. Aonzo et al. [17]...
showed that obfuscated malicious apps could evade anti-malware systems. They made their own tool to obfuscate Android apps, obfuscated Android malicious apps, and uploaded the apps to VirusTotal system [18] to check if the apps could be accurately classified as malicious. The test results showed that the performance of detecting obfuscated malicious apps was significantly lower than that of detecting the original malicious apps to which obfuscation was not applied. Therefore, in order to effectively detect an obfuscated malicious app, it is necessary to deobfuscate the obfuscated malicious app. There are several forms of obfuscation techniques for Android apps: identifier renaming, control-flow obfuscation, string encryption, class encryption, API hiding (Java reflection), etc. We focus on control-flow obfuscation and its deobfuscation in this paper.

We implement a new Android deobfuscation tool, Deoptfuscator, to determine whether the control-flow of an Android app is obfuscated by DexGuard, and then to deobfuscate the control-flow obfuscated apps. We also evaluate the performance of Deoptfuscator with respect to ReDex.

Among various issues related to Android deobfuscation techniques, we try to answer the following three research questions.

- **RQ1** How to detect and determine whether the control-flow of a given Android app is obfuscated or not?
- **RQ2** How to effectively deobfuscate a control-flow obfuscated app?
- **RQ3** How can we confirm that our deobfuscation approach was really successful?

In summary, the main contributions of this paper are the following:

- **Deoptfuscator** is the first tool for Android apps to detect and deobfuscate high-level control-flow obfuscation patterns of DexGuard.
- The effectiveness of Deoptfuscator is demonstrated by checking whether the deobfuscated app also runs the same as the original app which the control-flow obfuscation was not applied.
- The source code of Deoptfuscator has been published in a public repository on GitHub. Thus, it can be freely accessed and used by anyone [19], [20].

Our paper is organized as follows: Section 2 describes the characteristics and patterns of control-flow obfuscation and ART (Android Runtime) in Android. Section 3 explains the design and implementation of Deoptfuscator, and its deobfuscation strategy. Section 4 presents the experimental method to evaluate Deoptfuscator, and section 5 evaluates its performance. Section 6 describes the related studies and discusses the limitation of our study. Finally, section 7 concludes this work.

## II. BACKGROUND

Obfuscation is a technique that increases the time and cost required for program analysis while keeping the program’s functionality. Suppose an original program $P$ is transformed (obfuscated) to $P'$ using a transformation technique $T (P \rightarrow P')$. Then, the functionality of $P$ and $P'$ are the same, but the analysis complexity of $P'$ is much higher than $P$ [10]–[17], [21]–[24]. Popular obfuscation tools for Android apps include R8 [25], a compiler suite that incorporates ProGuard’s [26] obfuscation functions, DashO [27], DexProtector [28], and DexGuard [29].

Obfuscation techniques can be classified into four types as follows.

- **Identifier renaming** changes the name of the identifiers such as package, class, method and variable to meaningless symbols.
- **String encryption** encrypts and stores string literals, and decrypts them at runtime, restoring the original strings.
- **Control-flow obfuscation** changes the control-flow of a program by inserting dummy codes or exception handling codes (try-catch phrase), modifying branch/condition statements, etc.
- **Reflection obfuscation** hides the name of invoked methods using Java reflection (a.k.a. API hiding).

### A. CONTROL-FLOW OBFUSCATION

Control-flow obfuscation is a technique that hinders efficient program analysis by inserting dummy codes and exception handling codes, or modifying branch/condition statements, consequently complicating the order of code execution or function invocation. However, control-flow obfuscated codes can be simplified or removed by modern compilers. Recent compilers, such as R8 compiler, are equipped with excellent optimization techniques that can remove unnecessary codes [15].

Opaque predicates and opaque variables are useful in effective control-flow obfuscation. An opaque predicate is a conditional expression that is composed of complex operations, so that it is difficult to tell whether the result of the expression is true or false. The result of opaque predicate becomes known at runtime. An opaque variable is a variable used in opaque predicates [21]–[24], [30]–[35].

The usage pattern of the opaque variable and opaque predicate can divide the code obfuscation into three levels. The higher the level, the harder it is for the optimization tool to remove the obfuscated code.

#### 1) LEVEL 1

Level 1 control-flow obfuscation has the following form.

- Opaque variables are declared as local variables.
- Opaque predicates test whether a opaque variable is identical to a constant.

Fig. 1 shows an example of level 1 control-flow obfuscation. Fig. 1(a) is the obfuscated source codes, where ‘a’ and ‘b’ are opaque variables and ‘b == 1’ and ‘a == 2’ are opaque predicates. Since the conditional expressions at line 8 and 9 are always false, the compiler removes the conditional statement and the two local variables while the
functionality of the method ‘Obfuscation_1()’ is not changed. Fig. 1(b) is the Dalvik bytecodes compiled from the source codes by R8 compiler. It shows that the method ‘Obfuscation_1()’ does nothing and returns immediately.

2) LEVEL 2
Level 2 control-flow obfuscation has the following form.
- Opaque variables are declared as local variables.
- Opaque predicates consist of mathematical operations (e.g. positive/negative decision, odd/even decision, ...)

Fig. 2 shows an example of level 2 control-flow obfuscation. It differs from level 1 control-flow obfuscation in that the two opaque predicates employ modulo operations (‘b % 128 == 1’ and ‘a % 64 == 0’) instead of just comparing opaque variables with a constant. Again, since the opaque predicates at line 8 and 9 are always false, a compiler produces bytecodes that do nothing and just return if it optimizes the codes perfectly. However, when the source code (Fig. 2(a)) is compiled by R8 compiler with the default options, the produced bytecodes contain the logics for the opaque predicates (Fig. 2(b)). ReDex, an Android app optimization tool, can remove the local opaque variables and the simple opaque predicates. Fig. 2(c) is the bytecodes produced by ReDex. The resulting bytecodes do nothing and return immediately.

3) LEVEL 3 (Advanced Control-flow Obfuscation)
Level 3 control-flow obfuscation has the following form.
- Opaque variables are declared as global variables.
- Opaque predicates consist of mathematical operations (e.g. positive/negative decision, odd/even decision, ...)

Level 3 control-flow obfuscation is also called as advanced control-flow obfuscation. Even optimizers of recent compilers cannot easily optimize level 3 obfuscation. Fig. 3 shows an example of level 3 control-flow obfuscation. In Fig. 3(a), the opaque variables (‘g_a’ and ‘g_b’) are global within class test. Although the opaque predicates at line 8 and 9 are always false, neither R8 compiler nor ReDex removes the opaque variables and the opaque predicates. Fig. 3(b) and Fig. 3(c) show the optimized Dalvik bytecodes optimized by R8 compiler and ReDex, respectively. In this example, the two Dalvik bytecodes produced by R8 compiler and ReDex are exactly the same.

ReDex does not remove global opaque variables. Since a global variable may be used in several methods, ReDex regard global variables as non-opaque variables. To deobfuscate level 3 control-flow obfuscated codes, we should remove global opaque variables. If a global variable is used only in a method and opaque predicates, the global variable and predicates can safely be removed.

B. ANDROID RUNTIME (ART)
Ahead-of-Time (AOT) compilation statically translates codes before an execution of an app, while Just-in-Time (JIT) compilation dynamically translates codes during runtime [36]. AOT converts all codes to machine code at installation time, so app installation speed is slow compared to JIT. JIT converts frequently used bytecodes to machine code during runtime and app installation time is fast compared to AOT.
TABLE 1: The comparison of DVM and AOT compiler

|                    | JIT       | AOT       |
|--------------------|-----------|-----------|
| App installation time | Fast      | Slow      |
| Size of installed apps | Small    | Large     |
| Architecture supported | 32-bit only | 32- and 64-bit |
| Usage of CPU, memory | High      | Low       |
| Multidex            | Do not support | Support   |
| Battery consumption  | High      | Low       |

to continually improve execution performance of apps [40]–[42]. The JIT compiler complements ART’s AOT compiler and reduce storage space, and speeds app updates. ART also improves the AOT compiler by avoiding recompilation of apps during over-the-air (OTA) updates or system slowdown during automatic app updates.

ART’s **dex2oat** is an on-device compiler suite with several compilation backends, code generators for hardware platforms, etc. It is responsible for the validation of apps and their compilation to native code [32]. Fig. 4 shows the compilation process of the **dex2oat** compiler using **optimizing** backend. When a **.dex** file in an APK is given as input to the **dex2oat**, it checks the validity of the input file (**.dex**). Then, the code in the **.dex** is converted into an **.oat** file through Hydrogen Intermediate Representation (**HIR**). The **.oat** file is the AOT binary for the **.dex** file. The **HIR**, also called **optimizing**’s intermediate representation (**IR**), is a control-flow graph (**CFG**) on the method level which is denoted as **HGraph**. The **HGraph** is used as the single IR of the app code. When the **HGraph** is created, the **dex** instructions of the app’s bytecode are examined one after another, and the corresponding **HInstructions** are generated and interconnected with the current basic block and the graph. It is transformed into **single static assignment form**(SSA) for complex optimizations.

Typical optimizations using **HGraph** are as follows [43]–[47]:

- Class Hierarchy Analysis (**CHA**) guard elimination
- Bounds check elimination
- Global Value Numbering (**GVN**)
- Dead Code Elimination (**DCE**)
- Constant folding
- Loop optimization
By modifying the optimization part of ART, we develop three modules: Opaque identification module to identify opaque variables, Opaque location module to record the location of the identified opaque variables, and Opaque clinit module to remove the opaque variables. The detailed description of these modules are given in Section III.

C. DEXGUARD’S CONTROL-FLOW OBFUSCATION
The Android tool DexGuard provides obfuscation equivalent to Advanced control-flow obfuscation (level 3). This section describes the advanced control-flow obfuscation (level 3) used by DexGuard. Fig. 5 shows the transformation of Java source code when control-flow obfuscation is applied using DexGuard. The original code (Fig. 5(a)) is a simple onCreate() method without any operation instructions or branch/conditional statements, but the obfuscated code (Fig. 5(b)) contains several operation instructions and branch/conditional statements are inserted.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    addPreferencesFromResource(R.xml.settings);
}
```

(a) An original method before control-flow obfuscated with DexGuard

```
private static int f65 = 1;
private static int f66 = 6;
public void onCreate(Bundle bundle) {
    int i = f65 + 125;
    f65 = i % 128;
    if (i % 2 == 0) {
        super.onCreate(bundle);
        addPreferencesFromResource(R.xml.f148);
    }
    f66 = i + 79;
    switch (i % 2 == 8 ? 15 : 87) {
        case 13:
            Object[] objArr = null;
            int length = objArr.length;
            return;
        default:
            return;
    }
}
```

(b) The control-flow obfuscated method from (a) with DexGuard

**FIGURE 5:** An original method before obfuscation (a) and the method from which the original method was obfuscated by DexGuard

A pair of variables f65 and f66, declared as private static int in the class in Fig. 5(b), are opaque variables. The obfuscated onCreate() method adds a literal to f66 and stores the result in the local variable i. The result of modulo operation with another literal on the value of i is stored in f65. The variable i is used as part of the opaque predicate in the conditional expression of the if statement. Similar codes exist before the next switch-case statement. Through code analysis such as this, we can find that f66 affects f65 through simple arithmetic operations and local variable (i). This shows that f65 and f66 are global opaque variables and used in pairs. Also, it can be confirmed that the local variable i is an important variable that determines the true/false of the opaque predicate in the conditional expression of branch/conditional statements (if, switch-case). DexGuard’s control-flow obfuscation uses these patterns.

D. REDEX OPTIMIZER
Our proposed tool, Deoptfuscator detects the DexGuard’s control-flow obfuscation patterns described in Section II.C, lowers their obfuscation level to Level 2 from Level 3, and then optimizes them using ReDex. ReDex is an Android bytecode optimizer developed by Facebook Engineering team, which was released as open source [12], [15], [48], [49]. It takes a dex file as input and outputs the dex file with optimized bytecode. ReDex uses several modules to optimize dex files. Of them, we are interested in the followings:

- Inlining
- Dead Code Elimination (DCE)
- Peephole

Inlining is the process of replacing a function call at the point of call with the body of the function being called, thus reduces the overhead of a function call. DCE walks all branches and method invocations from the entry points of an app and removes any code that is unreachable. Peephole optimization involves replacing a small code patterns with an equivalent pattern that performs better. It performs a string search of the code for known inefficient sequences and replaces them with more efficient code. It can remove redundant load/store instructions and perform algebraic simplification, etc. Each module can be processed independently of each other.

Based on analyzing the characteristics of the optimization modules, we find out that ReDex can effectively remove the control-flow obfuscation of Level 2 defined in Section II-A, while it cannot handle the advanced control-flow obfuscation (Level 3) directly.

III. DESIGN OF DEOPTFUSCATOR
We propose Deoptfuscator, a tool that can deobfuscate Android apps. It can deobfuscate advanced control-flow obfuscation. It can be used alone in a user’s PC or as a part of ART compilation process. Deoptfuscator consists of three modules:

- Opaque identification
- Opaque location
- Opaque clinit

The Opaque identification module detects global opaque variables. The Opaque location module records
the location of opaque variables detected by the Opaque identification module. The Opaque clinit module changes the property of opaque variables appropriately.

A. OVERVIEW OF DEOPTFUSCATOR

Fig. 6 depicts the deobfuscation steps of Deoptfuscator. Deoptfuscator proceeds in the following order.

1) Unpackaging
Given a control-flow obfuscated APK, it unpackages the input APK using APKTool.

2) Detecting opaque variables
Using the Opaque identification module, it identifies the opaque variables.

3) Profiling detected opaque variables
Using the Opaque location module, the locations of opaque variables detected in step 2 are recorded in json format.

4) Lowering obfuscation level
Change the global opaque variables recorded in step 3 to local opaque variables, which means that the obfuscation level is lowered from level 3 to level 2.

5) Optimizing DEX
Using Redex, remove local opaque variables and opaque predicates.

6) Repackaging
Repackage the DEX file. The resulting APK is control-flow deobfuscated.

B. OPAQUE IDENTIFICATION

This section describes the process of the Opaque identification module of Deoptfuscator in detail using an example. Fig. 7 shows a part of method onCreate() which is control-flow obfuscated using DexGuard (Fig. 5(b)).

In Fig. 7, f65 and f66 are global opaque variables, and i is a local variable used as a bridge between f66 and f65 and between opaque variables and opaque predicates. Variable i is also used in a conditional expression (an opaque predicate). Using a local variable as a bridge between global opaque variables and opaque predicates increases the program complexity and prevents compilers or optimizers from removing control-flow obfuscation.

Fig. 8 shows the HIR for the code snippets in Fig. 7. Deoptfuscator utilizes this HIR to analyze the variable usage pattern, remove global opaque variables effectively and simplify the control-flow. In Fig. 8, ‘pred’ and ‘succ’ indicate the basic block numbers before and after the current basic block. BasicBlock 0 is the first basic block of method onCreate(), so there is no previous block and the subsequent block number is 1. BasicBlock 1 indicates that the previous block number is 0, and can branch to block 9 or 10. The label of each instruction denotes the return data type of the instruction and the execution order in a method. Alphabet ‘j’, ‘l’, ‘i’, ‘v’, and ‘z’ stand for ‘Java long’, ‘Java reference’, ‘Java int’, ‘Java void’, and ‘Java boolean’, respectively. For example, ‘i9: StaticFieldGet [18]’ means that this instruction gets a Java int variable from the field area of the class referred by 18.

Fig. 9 shows the HIR instructions converted from obfuscated Java source codes in the example. We explain each Java statement (S1 ~ S4) and its corresponding HIR instructions in a DexGuard’s obfuscation pattern.

S1 Get a reference to the class (18) that contains the current method (j7), and get the class variable f66 of the class (i9).

S2 Add f66 obtained from i9 and constant 125 (i10), and store the result in local variable i (i11).

S3 Perform modulo operation by dividing i11 (i) by constant 128 (i12), and store the result (i13) to the class variable f65 (v15) using class reference (18).

S4 Perform modulo operation by dividing i (i11) by constant 2 (i16), and the result (i17) is compared with constant 0 (i18) by NotEqual operation (z19). The result of the NotEqual operation is used as the conditional expression of If operation (v20).

Deoptfuscator analyzes the variable usage pattern based on HIR to detect global opaque variables. Fig. 10 shows the internal representation for the HIR given in Fig. 8. The analysis is performed as follows.

1) Deoptfuscator traverses all basic blocks from the BasicBlock 1 of the method. BasicBlock 0 is the initialization part of the method and BasicBlock 1 is where the actual instruction starts. Deoptfuscator checks whether the last instruction of basic block is If. Due to the nature of the basic block, branch-related instructions (if, goto, throw, return, etc.) are located in the last of each basic block. If the last instruction of the block is If, Deoptfuscator marks the instruction (If (v20)) and records the location.

2) Deoptfuscator traces the operands from the marked If, finds the StaticFieldGet through backward tracing and temporarily records the location as well. This is because a global variable can be an opaque variable only when it affects the decision of the opaque predicate of If. Through the trace of If (v20) → NotEqual (z19) → Rem (i17) → Add (i11) → StaticFieldGet (i19), it can be seen that the global variable obtained by StaticFieldGet is used in a series of operations (such as Add and Rem) and affects the decision of the opaque predicate of If.

3) It is necessary to check whether i9 is a class variable in the field area of the class to which the current method belongs. This is because DexGuard’s control-flow obfuscation does not use other classes’ opaque variables, but defines global opaque variables for each class and uses them only within a class. This step can be done by tracing StaticFieldGet (i19) → LoadClass (18) → CurrentMethod (j7). Now we have a global variable that determines a opaque predicate and i9 is a candidate for a opaque variable.

4) This step finds the buddy of global opaque variable i9. Through forward tracing StaticFieldGet (i19) → Add (i11) → Rem (i13) → StaticFieldSet (v15), we discover that the result i13 of the op-
Deoptfuscator records the locations of $v_{15}$ and $i_{13}$. Thereby, $i_9$ and $i_{13}$ become a pair of global opaque variable candidates.

5) This step confirms that the global opaque variable candidates are actually the opaque variables, i.e., opaque variables are not used anywhere except the obfuscation pattern. This step is necessary because fatal errors might occur if a developer unfortunately writes codes similar to obfuscation patterns and global variables are removed carelessly. Through forward tracing, we can
be convinced that the values of \(i_9\), \(i_{11}\), and \(i_{13}\) are not used in any other part of the current method.

Fig. 11 displays the constants (green), global variables (red), and a local variable (blue) of the Java source, as well as their location in the HIR. We can see that the global variables and the local variable are used only in the obfuscation patterns. We can confirm that \(i_9\) and \(i_{13}\) are global opaque variables. The information for the confirmed global opaque variables is stored in a temporary file for each method.

The above process is repeated for each method in a class. Note that opaque variables can be used in many methods in a class.

**C. OPAQUE LOCATION**

The Opaque location module collects the temporary files containing the information for the confirmed opaque variables and records the information in json format. Specifically, the information includes class name, method name, the field indexes of global opaque variables and the locations of instruction ('sget' and 'sput') that accesses global opaque variables. The instruction locations are the distance from the method’s offset in DEX file and can be calculated using the location of StaticFieldGet and StaticFieldSet in the HIR.

**D. OPAQUE CLINIT**

The Opaque clinit module removes the detected advanced control-flow obfuscation by lowering the obfuscation level. To decide whether to remove the detected control-flow obfuscation from a class, we measure the ratio of bytecodes matching the obfuscation pattern to the entire bytecodes of a class. We call this ratio \(OBR\) and is defined as follows:

\[
OBR = \frac{\sum_m N_m}{\sum_m L_m}
\]  

where \(N_m\) is the length of bytecodes of control-flow obfuscation patterns detected in method \(m\) and \(L_m\) is the length of bytecodes of method \(m\).

The series of instructions from \(l_8\) to \(v_{20}\) in Fig. 10 is a control-flow obfuscation pattern in HIR. Its corresponding pattern in bytecodes is a series of instructions from ‘sget’ to ‘if-nez’ in Fig. 12. For example, consider a class \(C\) with two methods \(m_1\) and \(m_2\). Assume Deoptfuscator detected one obfuscation pattern in \(m_1\) and two in \(m_2\), and that the obfuscation patterns are the same as Fig. 12 (from ‘sget’ to ‘if-nez’). Since the length of a bytecode instruction is 4 bytes, the length of an obfuscation pattern is 24 bytes. Thus, the total length of the obfuscation patterns detected in class \(C\) is \(N_{m_1} + N_{m_2} = 24 + 2 \times 24 = 72\) bytes.

\(L_m\) is the length of bytecodes of method \(m\) and can be obtained from DEX file. Among the items of DEX file, there are \(insns\) and \(insns\_size\) fields in the code item area. \(insns\) is an array containing the bytecode of a method, and \(insns\_size\) indicates the length of \(insns\). In other words, \(insns\_size\) is the total length of the bytecode of a method. Let \(insns\_size\) of method \(m_1\) and \(m_2\) of class \(C\) be 100 and 200, respectively. Then \(L_{m_1} + L_{m_2} = 100 + 200 = 300\).

A high \(OBR\) implies that obfuscation patterns are found
in a class many times. Such a class is likely to be control-flow obfuscated since obfuscators tend to insert obfuscation patterns into a class many times. If the OBR of a class is higher than a threshold \( \theta \), Deoptfuscator regards the class as obfuscated and deobfuscates it. Otherwise, the detected obfuscation pattern, if any, is regarded as false positive.

The threshold \( \theta \) is selected empirically. Using the threshold, we can control how aggressively we deobfuscate classes. As the threshold decreases, the number of classes to which deobfuscation is applied increases (aggressive deobfuscation). As the threshold increases, the number of classes to which deobfuscation is applied decreases (passive deobfuscation). If the threshold is 0, Deoptfuscator deobfuscates all classes. For example, assuming \( \theta = 0.15 \), the OBR of class \( C \) above is calculated as follows and Deoptfuscator deobfuscates \( C \).

\[
OBR = \frac{\sum_{m} N_{m}}{\sum_{m} L_{m}} = \frac{24 + 48}{100 + 200} = 0.24 > \theta (= 0.15)
\]

For a class with \( OBR > \theta \), Deoptfuscator lowers its control-flow obfuscation level from 3 to 2 by converting global opaque variables to local variables. These global variables are defined in method \texttt{clinit}. The Opaque \texttt{clinit} changes the instruction to read a global variable (\texttt{sget}) and the instruction to write a value to the global variable (\texttt{put}) to the instruction to assign or get a value of a local variable (\texttt{const/16}). Then, the Opaque \texttt{clinit} module removes the codes that declare the global opaque variable pairs. Since there is no place where global opaque variables are used in the class through the previous processes, removing them does not cause errors.

E. OPTIMIZING DEX

Deoptfuscator optimizes the modified bytecodes (DEX file) utilizing ReDex. As explained in Section II-A, ReDex can remove level 2 control-flow obfuscation. Fig. 13 shows the Java code decompiled from the deobfuscated version of \texttt{onCreate()} of Fig. 5(b). You can see that the code of the method has been restored to the same as the original (Fig. 5(a)).


\begin{verbatim}
public void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    addPreferencesFromResource(R.xml.f140);
}
\end{verbatim}

FIGURE 13: The deobfuscated method by Deoptfuscator, which is in Fig. 5(b)

A. DATASET FOR EVALUATION

We used the Android apps that F-Droid project collected in our experiment - we will call them original apps in this paper. Using one original app, we generated two more apps by applying control-flow obfuscation and optimization of DexGuard. We created two obfuscated apps for each original app. One was applied the high-level obfuscation/optimization option and the other applied the moderate-level obfuscation/optimization option. We select 63 original apps that all three versions of apps run normally on AVD (Android Virtual Device) and an actual smartphone (Pixel 2 XL with Android Oreo 9.0).

B. EXPERIMENTAL METHOD

As described above, 63 highly obfuscated apps and 63 moderately obfuscated apps were generated from 63 original apps. Theses apps were deobfuscated using our proposed Deoptfuscator. In order to determine how well the Deoptfuscator perform, the deobfuscated apps were compared with ones optimized using the ReDex optimization tool. The optimizer does not aim to deobfuscate apps, but it can be considered as minimal deobfuscation in that it eliminates meaningless or unnecessary comparisons and loops. We created 63 optimized apps for highly obfuscated apps and moderately obfuscated ones, respectively.

The Deoptfuscator’s performance depends on the threshold of Eq. 1. Therefore, we deobfuscated apps by changing this threshold to 0.015, 0.075, 0.15, and 0.225. A total of 504 apps were created, two sets of 252 (63×4) apps for highly and moderately obfuscated apps. Therefore, the list and the number of apps used in our experiment are as follows (Fig. 14).

- original apps (63)
- highly obfuscated apps (63)
- moderately obfuscated apps (63)
- ReDex-optimized versions of highly obfuscated apps (63)
- ReDex-optimized versions of moderately obfuscated apps (63)
- Deobfuscated versions of highly obfuscated apps according to the thresholds (252)
- Deobfuscated versions of moderately obfuscated apps according to the thresholds (252)

We compared the deobfuscation performance based on the following criteria.

- Size of dex file
- The number of methods

IV. EXPERIMENTAL SETUP
ProGuard aims to minimize the storage space by shortening identifier names. However, ProGuard identifier renaming obfuscation, which renames identifiers such as classes, methods, and variables to meaningless shorter ASCII names. Naturally, it can be seen that the number of basic blocks and edges for each method also increases. The value of insns, which represents the number of bytecode instructions, also increases more than 4 times. Despite the significant increase in the number of basic blocks and instructions, related to executable code, the reason why the size of the dex file has increased by about 43% is due to optimizations such as identifier renaming and unnecessary method removal.

Let’s see the result of optimizing the obfuscated app with Redex. First, if you look at the change in the number of methods, you can see that there is little difference because DexGuard removes unused methods along with obfuscation. The number of basic blocks and CFG edges is about 1.57 times and 3.6 times that of the original app, which correspond to about 19% and 28% of the highly obfuscated app. The number of basic blocks and edges per method shows a similar trend. It can be confirmed that the length of the bytecode is also about 50% of the obfuscated one.

When deobfuscating with Deoptfuscator, the larger the threshold, the fewer classes to which deobfuscation is applied, and the smaller the threshold, the more it increases. When the number of classes to which deobfuscation is applied is small, most classes are only optimized by Redex, so the results of deobfuscation and optimization show a similar result. As an example, you can find the result of deobfuscation with \( \theta = 0.225 \) is almost similar to that of optimizing with Redex. When the other three thresholds were set, the number of basic blocks was 1.15 times the original, and the number of edges was about twice. The length of the bytecode was about 1.06 times, which was almost the same size as the original. As shown in Fig. 15, if the threshold is greater than 0.15, the effect of intrinsic deobfuscation almost disappears.

### B. MODERATELY OBfuscated APPS

Fig. 16 and Table 3 show the experimental results for an app that is moderately obfuscated. With moderately obfuscated apps, the results show the same tendency as highly obfuscated apps, but the numbers are small because DexGuard applies the same optimization but a subset of obfuscation.

The number of methods yielded almost the same result as for highly obfuscated apps. In other words, it can be confirmed once again that the decrease in the number of methods is a result of DexGuard’s optimization. The number of basic blocks and CFG edges increased to about 3.78 times and 5.19 times of the original, respectively. The number of basic blocks and CFG edges per method also increased. The length of bytecode increased by about 69%, but the size of the .dex file was reduced to about 76% due to optimization.
ReDex optimization reduces the number of basic blocks and CFG edges to about 28% and 30% of the moderately obfuscated app, which correspond to about 1.04 times and 1.58 times that of the original app. The number of basic blocks and edges per method shows a similar trend. It can be confirmed that the length of the bytecode is also about 57% of the obfuscated one.

The deobfuscated app has 0.84 times the original basic blocks, and 1.12 times the number of edges. The length of the bytecode was about 1.06 times, which was almost the same size as the original.

### C. THE DEGREE OF SIMILARITY

How similar the deobfuscated app is to the original app will best indicate the effectiveness of a deobfuscation tool. Since control-flow obfuscation is performed on a method-by-method basis, it is reasonable to measure similarity on a method-by-method basis. We use Androguard’s Androsim module to calculate the similarity between an original app, one obfuscated with DexGuard, one optimized with ReDex, and one deobfuscated with Deoptfuscator.

As described above, the number of methods in the obfuscated app is about 68% of that of the original one, so the expected similarity to the original is about 68%. In addition, the similarity will be lower because methods that are not obfuscated can be modified by optimization.

---

**TABLE 2:** Analysis of highly obfuscated apps

|                  | original | DexGuard | ReDex | \(\theta = 0.225\) | \(\theta = 0.15\) | \(\theta = 0.075\) | \(\theta = 0.015\) |
|------------------|----------|----------|------|-------------------|-------------------|-------------------|-------------------|
| Dex size         | 1        | 1.43     | 0.96 | 0.92              | 0.68              | 0.67              | 0.67              |
| # of methods     | 1        | 0.69     | 0.68 | 0.68              | 0.68              | 0.68              | 0.68              |
| # of basic blocks| 1        | 8.23     | 1.57 | 1.52              | 1.17              | 1.15              | 1.15              |
| # of CFG edges   | 1        | 12.75    | 3.60 | 3.09              | 2.03              | 1.96              | 1.96              |
| # of edges per method | 1   | 17.73    | 5.10 | 4.43              | 2.89              | 2.81              | 2.81              |
| # of basic blocks per method | 1    | 11.47    | 2.23 | 2.16              | 1.66              | 1.63              | 1.63              |
| insns size       | 1        | 4.09     | 2.05 | 1.89              | 1.09              | 1.05              | 1.05              |

**TABLE 3:** Analysis of moderately obfuscated apps

|                  | original | DexGuard | ReDex | \(\theta = 0.225\) | \(\theta = 0.15\) | \(\theta = 0.075\) | \(\theta = 0.015\) |
|------------------|----------|----------|------|-------------------|-------------------|-------------------|-------------------|
| Dex size         | 1        | 0.76     | 0.59 | 0.57              | 0.55              | 0.53              | 0.53              |
| # of methods     | 1        | 0.69     | 0.68 | 0.68              | 0.68              | 0.68              | 0.68              |
| # of basic blocks| 1        | 3.78     | 1.04 | 0.97              | 0.87              | 0.82              | 0.82              |
| # of CFG edges   | 1        | 5.19     | 1.58 | 1.41              | 1.21              | 1.07              | 1.07              |
| # of edges per method | 1   | 7.22     | 2.27 | 2.04              | 1.74              | 1.56              | 1.56              |
| # of basic blocks per method | 1    | 5.27     | 1.47 | 1.38              | 1.24              | 1.16              | 1.16              |
| insns size       | 1        | 1.69     | 0.98 | 0.91              | 0.79              | 0.75              | 0.75              |
You et al.: Deoptfuscator: Defeating Advanced Control-flow Obfuscation Using ART (Android Runtime)

Fig. 16: Comparison of performance using Deoptfuscator and ReDex for moderately obfuscated apps

Fig. 17 shows similarity for highly obfuscated apps. The average similarity of highly obfuscated apps is about 19%, and the average similarity of apps optimized with ReDex is about 26%. It can be said that the similarity increased because the optimization tool can remove some obfuscated codes. Looking at the similarity with the app deobfuscated with Deoptfuscator, the larger the threshold, the less the number of methods to which deobfuscation is applied, which is closer to the ReDex result.

VI. RELATED WORK AND DISCUSSION
A. RELATED WORK
Piao et al. [37] first inspected both the weakness and the obfuscation process of DexGuard. For an app obfuscated by DexGuard, they could (1) rename classes of a DEX to deobfuscate the identifier renaming technique by analyzing the renaming dictionary of DexGuard and using dex2jar,
Deobfuscator is not a tool for Android apps, several processes are required to use it for Android apps. That is, it is necessary to (1) convert the obfuscated DEX file of a given Android app into a JAR file, (2) apply Java-Deobfuscator to the JAR file, and then (3) convert the deobfuscated JAR file into a DEX file again. However, since there is a loss in the process of converting the obfuscated DEX file into a JAR file, it is difficult to expect Java-Deobfuscator to work properly, and it is very hard to correctly create and run an Android app with the finally deobfuscated DEX file.

Moses and Mordekhay [55] utilized both static and dynamic analysis to defeat two obfuscation techniques: string encryption and dynamic method binding via reflection. Their deobfuscation solution was tested on 586 Android apps, containing strings encrypted by DashO obfuscator. They identified decryption calls and extracted argument values, executed the decryption calls, and obtained the decryption results. They found out that the argument values were retrieved for 99% of the decryption calls on average. They mentioned that it is necessary to handle string encryption even in case that the decryption logic is not included in a single function for further research.

De Vos and Pouwelse [56] proposed a string deobfuscator, ASTANA, to identify the deobfuscation logic for each string literal and execute the logic to recover the original string values from obfuscated string literals in Android apps. ASTANA uses program slicing to seek for an executable code snippet with proper statements to handle a obfuscated strings.

According to the study of Wong and Lie [47], language-based and full-native code obfuscation techniques include reflection, value encryption, dynamic loading, native methods, and full-native code obfuscation. In addition to the traditional obfuscations, Wong and Lie [47] described a set of runtime-based obfuscations in ART such as DEX file hooking, class data overwriting, ArtMethod hooking, etc. They then developed a hybrid iterative deobfuscator, TIRO (Target-Instrument-Run-Observe), which is a framework to deobfuscate malicious Android apps. TIRO employed both static instrumentation and dynamic information gathering, and could reverse language-based and runtime-based obfuscation techniques.

In our previous work, we analyzed the performance of tools for obfuscating, deobfuscating, and optimizing Android apps [15]. We chose R8 compiler and Obfuscapck for obfuscators, DeGuard for a deobfuscator, and R8 compiler and Redex for optimizers. As the default compiler for Android apps, R8 has various features including optimization (removing unused codes, inlining) and obfuscation (identifier renaming). We examined the characteristics of the four tools and compare their performance. R8 showed better performance than Redex in terms of the number of classes, methods, and resources.

An Android app can contains native code binaries written in C or C++. Thus, there was a study to deobfuscate Android native binary code rather than the Android Dalvik bytecode. Kan et al. [57] proposed an automated system to
deobfuscate native binary code of an Android app obfuscated by O\textsuperscript{2}\textsuperscript{-LLVM}. O\textsuperscript{2}\textsuperscript{-LLVM} is a popular native code obfuscator which provides three obfuscations: instruction substitution, bogus control-flow and control-flow flattening. Kan et al. could recover the original control-flow graph of native binary code using taint analysis and flow-sensitive symbolic execution. For example, they used taint analysis for global opaque predicate matching to remove dead branches.

On the one hand, Ming et al. [34] tried to detect obfuscation techniques based on opaque predicates. Pointing out that existing researches were not sufficient to detect opaque predicates in terms of generality, accuracy, and obfuscation-resilience. They suggested a Logic Oriented Opaque Predicate (LOOP) detection tool for obfuscated binary code, which developed based on symbolic execution and theorem proving techniques. Their approach captured the intrinsic semantics of opaque predicates with formal logic, and could even detect intermediate contextual and dynamic opaque predicates.

B. DISCUSSION

In our previous work [15], we compared optimizers and deobfuscators for Android apps, and evaluated their performance. Program optimization is a technique aimed at improving program execution speed by reducing the use of resources as well as by eliminating redundant instructions, unnecessary branches, and null-checks. On the other hand, program deobfuscation focuses on removing or mitigating the obfuscation techniques applied to the program and restore the obfuscated codes to the same or similar states as the original. Traditional control-flow obfuscation contains call indirection by substituting existing methods and adding new methods, junk-code insertion (insertion of useless computations), abuse of \texttt{goto} instructions, etc. Thus, deobfuscating control-flow obfuscated codes seems similar to optimization because it may also improve program execution performance. However there is a difference in that its key purpose is to restore the control-flow obfuscated app to the original.

In this paper, we devised a new approach to deobfuscating control-flow obfuscated Android apps, and verified its effectiveness based on various evaluations and similarity measurements. In addition, our approach is flexible and scalable because it allows users to determine whether to apply aggressive or passive deobfuscation techniques after calculating the proportion of patterns identified that control-flow obfuscation are applied among instructions within one class through \textit{OBR}.

Our work has some limitations. The proposed technique can only handle control-flow obfuscation by \textit{DexGuard}, and does not consider control-flow obfuscation by other obfuscators including \textit{DashO} and \textit{Allatori}. If a developer accidentally writes an app includes the control-flow obfuscation patterns of \textit{DexGuard}, a problem may arise if \textit{Deoptfuscator} removes the global variables to deobfuscate the app. To prevent this problem, we divided the detected opaque variable into a candidate and a confirmation stage by checking whether the opaque variable was used in a part other than the obfuscation pattern through data-flow analysis. Separately, we checked whether any of many benign apps contains the code obfuscated with the control-flow obfuscation technique of \textit{DexGuard}, but there was no such app.

All apps that have been deobfuscated by \textit{Deoptfuscator} are executable on both a AVD and a real smartphone. The research on apps with an anti-tampering protection is out of the scope of this paper. Thus, if an obfuscated app is equipped with an integrity protection mechanism, the execution of its deobfuscated app cannot be guaranteed because the code has been changed due to the deobfuscation.

VII. CONCLUSION

We defined the three levels of control-flow obfuscation according to the usage patterns of opaque variables and the type of opaque predicates used in Android apps. \textit{DexGuard}, a powerful obfuscation tool for Android, offers the level 3 (advanced control-flow obfuscation) obfuscation, which uses global variables as opaque variables. Existing deobfuscators or optimizers have a difficulty of removing the level 3 obfuscation codes because if the global variables are arbitrarily removed from the obfuscated app, a fatal error may occur during execution.

We have then developed \textit{Deoptfuscator} that can effectively detect and deobfuscate the codes added by the control-flow obfuscation of \textit{DexGuard}. The \textit{Deoptfuscator} analyzes variable usage patterns to confirm global opaque variables are used only in opaque predicates. We evaluated its performance with respect to Re\textit{Dex} and demonstrated the effectiveness by showing that the apps deobfuscated by \textit{Deoptfuscator} can run normally on both a real device and the AVD. We have published the source code of \textit{Deoptfuscator} at the public repository GitHub, which helps malware analysts to reverse control-flow obfuscated malicious Android apps.

APPENDIX A

Fig. 19 shows four control-flow graphs: the graph of an original code (a), the graph of the code obfuscated from the original with \textit{DexGuard} (b), the graph of the code optimized from the obfuscated with Re\textit{Dex} (c), and the graph of the code deobfuscated from the obfuscated with \textit{Deoptfuscator}. The name of apk and method is ‘An.stop_9.apk’ and ‘An.stop.SettingsActivity.onCreate()’, respectively. Four control-flow graphs are the same method, but the name of the package and class has been changed due to \textit{DexGuard}’s identifier renaming.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science and ICT (no. 2018R1A2B2004830) and Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science and ICT (no. 2017R1A2B4004089). If a developer accidentally writes an app includes the control-flow obfuscation patterns of \textit{DexGuard}, a problem may arise if \textit{Deoptfuscator} removes the global variables to deobfuscate the app. To prevent this problem, we divided the detected opaque variable into a candidate and a confirmation stage by checking whether the opaque variable was used in a part other than the obfuscation pattern through data-flow analysis. Separately, we checked whether any of many benign apps contains the code obfuscated with the control-flow obfuscation technique of \textit{DexGuard}, but there was no such app.

All apps that have been deobfuscated by \textit{Deoptfuscator} are executable on both a AVD and a real smartphone. The research on apps with an anti-tampering protection is out of the scope of this paper. Thus, if an obfuscated app is equipped with an integrity protection mechanism, the execution of its deobfuscated app cannot be guaranteed because the code has been changed due to the deobfuscation.

VII. CONCLUSION

We defined the three levels of control-flow obfuscation according to the usage patterns of opaque variables and the type of opaque predicates used in Android apps. \textit{DexGuard}, a powerful obfuscation tool for Android, offers the level 3 (advanced control-flow obfuscation) obfuscation, which uses global variables as opaque variables. Existing deobfuscators or optimizers have a difficulty of removing the level 3 obfuscation codes because if the global variables are arbitrarily removed from the obfuscated app, a fatal error may occur during execution.

We have then developed \textit{Deoptfuscator} that can effectively detect and deobfuscate the codes added by the control-flow obfuscation of \textit{DexGuard}. The \textit{Deoptfuscator} analyzes variable usage patterns to confirm global opaque variables are used only in opaque predicates. We evaluated its performance with respect to Re\textit{Dex} and demonstrated the effectiveness by showing that the apps deobfuscated by \textit{Deoptfuscator} can run normally on both a real device and the AVD. We have published the source code of \textit{Deoptfuscator} at the public repository GitHub, which helps malware analysts to reverse control-flow obfuscated malicious Android apps.

APPENDIX A

Fig. 19 shows four control-flow graphs: the graph of an original code (a), the graph of the code obfuscated from the original with \textit{DexGuard} (b), the graph of the code optimized from the obfuscated with Re\textit{Dex} (c), and the graph of the code deobfuscated from the obfuscated with \textit{Deoptfuscator}. The name of apk and method is ‘An.stop_9.apk’ and ‘An.stop.SettingsActivity.onCreate()’, respectively. Four control-flow graphs are the same method, but the name of the package and class has been changed due to \textit{DexGuard}’s identifier renaming.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science and ICT (no. 2018R1A2B2004830) and Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science and ICT (no. 2017R1A2B4004089).
The control-flow graph of an original code (a).
The control-flow graph of the code obfuscated from the original with DexGuard (b).
The control-flow graph of the code optimized from the obfuscated with ReDex (c).
The control-flow graph of the code deobfuscated from the obfuscated with Deoptfuscator (d).

FIGURE 19: Four control-flow graphs: the graph of an original code (a), the graph of the code obfuscated from the original with DexGuard (b), the graph of the code optimized from the obfuscated with ReDex (c), and the graph of the code deobfuscated from the obfuscated with Deoptfuscator (d).

REFERENCES

[1] Statcounter, “Mobile Operating System Market Share Worldwide,” https://gs.statcounter.com/os-market-share/mobile/worldwide/monthly-202002-202102, accessed: 2021.06.
[2] Statista, “Number of available applications in the Google Play Store from December 2009 to September 2021,” https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, accessed: 2021.06.
[3] B. Kim, K. Lim, S.-J. Cho, and M. Park, “Romadroid: A robust and efficient technique for detecting android app clones using a tree structure and components of each app’s manifest file,” IEEE Access, vol. 7, pp. 72 182–72 196, 2019.
[4] Q. Do, B. Martini, and K.-K. R. Choo, “Is the data on your wearable device secure? An android wear smartwatch case study,” Software: Practice and Experience, vol. 47, no. 3, pp. 391–403, 2017.
[5] B. Cyr, W. Horn, D. Miao, and M. Specter, “Security analysis of wearable fitness devices (fitbit),” Massachusetts Institute of Technology, vol. 1, 2014.
[6] W. Zhou and S. Piramuthu, “Security/privacy of wearable fitness tracking int devices,” in 2014 9th Iberian Conference on Information Systems and Technologies (CISTI). IEEE, 2014, pp. 1–5.
[7] J. Rieck, “Attacks on fitness trackers revisited: A case-study of unit firmware security,” arXiv preprint arXiv:1604.03313, 2016.
[8] A. K. Das, P. H. Pathak, C.-N. Chuah, and P. Mohapatra, “Uncovering privacy leakage in ble network traffic of wearable fitness trackers,” in Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications, 2016, pp. 99–104.
[9] Y. Tang, Y. Sui, H. Wang, X. Luo, H. Zhou, and Z. Xu, “All your app links are belong to us: understanding the threats of instant apps based attacks,” in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 914–926.
[10] M. Park, G. You, S.-j. Cho, M. Park, and S. Han, “A framework for identifying obfuscation techniques applied to android apps using machine learning,’’ J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl., vol. 10, no. 4, pp. 22–30, 2019.
[11] V. Sihag, M. Vardhan, and P. Singh, “A survey of android application and malware hardening,” Computer Science Review, vol. 39, p. 100365, 2021.
[12] G. You, G. Kim, J. Park, S.-J. Cho, and M. Park, Reversing Obfuscated Control Flow Structures in Android Apps Using ReDex Optimizer.
D. Xu, J. Ming, and D. Wu, “Generalized dynamic opaque predicates: A new control flow obfuscation method,” in International Conference on Information Security. Springer, 2016, pp. 323–342.

P. Faruki, H. Fereidooni, V. Laxm, M. Conti, and M. Gaur, “Android code protection via obfuscation techniques: past, present and future directions,” arXiv preprint arXiv:1611.10231, 2016.

Y. Piao, J.-H. Jung, and J. H. Yi, “Server-based code obfuscation scheme for apk tamper detection,” Security and Communication Networks, vol. 9, no. 6, pp. 457–467, 2016.

P. Schw nerme r, “Performance evaluation of kotlin and java on android runtime,” 2018.

X. Xu, K. Cooper, J. Brock, Y. Zhang, and H. Ye, “Sharejit: Jit code cache sharing across processes and its practical implementation,” Proceedings of the ACM on Programming Languages, vol. 2, no. OOPSLA, pp. 1–23, 2018.

Google, “Configuring ART,” https://source.android.com/devices/tech/dalvik/configure, accessed: 2021.06.

——, “Implementing ART Just-In-Time (JIT) Compiler,” https://source.android.com/devices/tech/dalvik/jit-compiler, accessed: 2021.06.

——, “Android 5.0 Behavior Changes,” https://developer.android.com/about/versions/lollipop/android-5.0-changes, accessed: 2021.06.

M. Buckes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, “Artist: The android runtime instrumentation and security toolkit,” in 2017 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 2017, pp. 481–495.

M. Sun, T. Wei, and J. C. Lui, “Taintart: A practical multi-level information-flow tracking system for android runtime,” in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 331–342.

O. Schranz, “Artist—a novel ins-trumentation framework for reversing and analyzing android apps and the middleware,” 2019.

V. Costamagna and C. Zheng, “Ardroid: A virtual-method hooking framework on android art runtime.” in IMPoS ESSoS, 2016, pp. 20–28.

M. Y. Wong and D. Lie, “Tackling runtime-based obfuscation in android with [TIRO],” in 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 1247–1262.

F. engineering, “Optimizing Android bytecode with ReDex,” https://engineering.fb.com/2015/10/01/android/optimizing-android-bytecode-with-redex, accessed: 2021.06.

——, “Redex - An Android Bytecode Optimizer,” https://fbreadex.com/, accessed: 2021.06.

A. Desnos, “Reverse engineering, Malware analysis of Android applications ... and more (ninja) !” https://github.com/androguard/androguard, accessed: 2021.06.

C. Fenton, “Generic Android Deobfuscator,” https://github.com/CalebFenton/simplify, accessed: 2021.06.

B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, “Statistical deobfuscation of android applications,” in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 343–355.

R. Baumann, M. Protsenko, and T. Müller, “Anti-proguard: Towards automated deobfuscation of android apps,” in Proceedings of the 4th Workshop on Android Security and Privacy in Communication Systems. Springer, 2018, pp. 172–192.

B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, “Statistical deobfuscation of android apps,” in Proceedings of the 4th Workshop on Android Security and Privacy in Communication Systems. Springer, 2018, pp. 172–192.

F. engineering, “Optimizing Android bytecode with ReDex,” https://engineering.fb.com/2015/10/01/android/optimizing-android-bytecode-with-redex, accessed: 2021.06.

——, “Redex - An Android Bytecode Optimizer,” https://fbreadex.com/, accessed: 2021.06.

A. Desnos, “Reverse engineering, Malware analysis of Android applications ... and more (ninja) !” https://github.com/androguard/androguard, accessed: 2021.06.

C. Fenton, “Generic Android Deobfuscator,” https://github.com/CalebFenton/simplify, accessed: 2021.06.

B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, “Statistical deobfuscation of android applications,” in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 343–355.

R. Baumann, M. Protsenko, and T. Müller, “Anti-proguard: Towards automated deobfuscation of android apps,” in Proceedings of the 4th Workshop on Security in Highly Connected IT Systems, 2017, pp. 7–12.

——, “Android 5.0 Behavior Changes,” https://developer.android.com/about/versions/lollipop/android-5.0-changes, accessed: 2021.06.

J. Meng, D. Xu, L. Wang, and D. Wu, “Loop: Logic-oriented opaque predicate detection in obfuscated binary code,” in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 757–768.

D. Xu, J. Ming, and D. Wu, “Generalized dynamic opaque predicates: A new control flow obfuscation method,” in International Conference on Information Security. Springer, 2016, pp. 323–342.
GEUNHA YOU received a B.E. degree in Dept. of Software Science, and an M.E. degree in Computer Science and Engineering from Dankook University, South Korea, in 2020, and 2022, respectively. His current research interests include computer security, mobile security, reverse engineering and embedded system.

GYOOSIK KIM received a B.E. in Applied Computer Engineering, and an M.E. in Computer Science and Engineering from Dankook University, South Korea in 2016, and 2018, respectively. He is currently a research engineer in infra R&D Lab at Korea Telecommunication. His current research interests include computer security and software intellectual property protection.

SANGCHUL HAN received his B.S. degree in Computer Science from Yonsei University in 1998 and his M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 2000 and 2007, respectively. He is now a professor in the Dept. of Computer Engineering at Konkuk University. His research interests include real-time scheduling and computer security.

MINKYU PARK received the B.E., M.E., and Ph.D. degree in Computer Engineering from Seoul National University in 1991, 1993, and 2005, respectively. He is now a professor in Konkuk University, Rep. of Korea. His research interests include operating systems, real-time scheduling, embedded software, computer system security, and HCI. He has authored and co-authored several journals and conference papers.

SEONGJE CHO received the B.E., M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 1989, 1991 and 1996, respectively. In 1997, he joined the faculty of Dankook University, Korea, where he is currently a Professor in Department of Computer Science and Engineering (Graduate school) and Department of Software Science (Undergraduate school). He was a visiting research professor at Department of EECS, University of California, Irvine, USA in 2001, and at Department of Electrical and Computer Engineering, University of Cincinnati, USA in 2009 respectively. His current research interests include computer security, mobile app security, operating systems, and software intellectual property protection.

***