IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 26, 2022, accepted May 29, 2022, date of publication June 8, 2022, date of current version June 15, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3181373

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

GEUNHA YOU[1], GYOOSIK KIM[2], SANGCHUL HAN[3], MINKYU PARK[3], AND SEONG-JE CHO[4] (Member, IEEE)

[1]Department of Computer Science and Engineering, Dankook University, Yongin 16890, South Korea
[2]Korea Telecom Infra Laboratory, Seoul 06763, South Korea
[3]Department of Computer Engineering, Konkuk University, Chungju 27478, South Korea
[4]Department of Software Science, Dankook University, Yongin 16890, South Korea

Corresponding author: Minkyu Park (minkyup@kku.ac.kr)

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and Information and Communication Technology (ICT) under Grant 2018R1A2B2004830 and Grant 2021R1A2C2012574.

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, *Deoptfuscator*, to detect obfuscated an Android app and to restore the original source codes. *Deoptfuscator* detects an app control-flow obfuscated by *DexGuard* and tries to restore the original control-flows. *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.

INDEX TERMS Android app, malicious app, obfuscation, deobfuscation, control-flow obfuscation.

I. INTRODUCTION

The Android Operating System occupies 71.45% of the smartphone operating system (OS) market as of May 2022, and the number of apps that appeared in Google Play Store, the official Google app market, records 2.59 million as of March 2022 [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 *MIAFinder* which detected vulnerabilities that could be exploited by the malicious instant app. *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

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

uploaded the apps to VirulTotal 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 *Deobfuscator* 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 *Deopfuscator* 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 *Deopfuscator*, 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 ($P \xrightarrow{T} P'$). 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.

The optimizing compiler can remove Level 1 control-flow obfuscation because it is easy for the compiler to determine if a variable is a local variable and if the variable is compared to a constant.

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. 1 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. 1(a)) is compiled by *R8* compiler with the default options, the produced bytecodes contain the logics for the opaque predicates (Fig. 1(b)). *ReDex*, an Android app optimization tool, can remove the local opaque variables and the simple opaque predicates. Fig. 1(c) is the bytecodes produced by *ReDex*. The resulting bytecodes do nothing and return immediately.

```
1  package com.example.myapplication;
2
3  public class test {
4
5      public void obfuscation_2() {
6          int a = 1;
7          int b = 2;
8          if( b % 128 == 1 ) {
9              if( a % 64 == 0 ) {
10                 System.out.println("Hello Android");
11             }
12         }
13     }
14 }
```

(a) Level 2 control-flow obfuscated Java source codes

```
|[02de80] com.example.myapplication.test.obfuscation_2:()V
|0000: const/4 v0, #int 1 // #1
|0001: const/4 v1, #int 2 // #2
|0002: rem-int/lit16 v2, v1, #int 128 // #0080
|0004: const/4 v3, #int 1 // #1
|0005: if-ne v2, v3, 0012 // +000d
|0007: rem-int/lit8 v2, v0, #int 64 // #40
|0009: if-nez v2, 0012 // +0009
|000b: sget-object v2, Ljava/lang/System;.out:Ljava/io/PrintStream; // field@3e52
|000d: const-string v3, "Hello Android" // string@0448
|000f: invoke-virtual {v2, v3}, Ljava/io/PrintStream;.println:(Ljava/lang/String;)V // method@0127
|0012: return-void
```

(b) Bytecodes compiled from the Java source codes

```
|[064114] com.example.myapplication.test.obfuscation_2:()V
|0000: return-void
```

(c) Bytecodes optimized by *ReDex*

FIGURE 1. Example of level 2 control-flow obfuscation. The opaque variables and opaque predicates can be removed by *ReDex*.

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. 2 shows an example of level 3 control-flow obfuscation. In Fig. 2(a),

```
1  package com.example.myapplication;
2
3  public class test {
4      private static int g_a = 1;
5      private static int g_b = 2;
6
7      public void obfuscation_3() {
8          if( g_b % 128 == 1 ) {
9              if( g_a % 64 == 2 ) {
10                 System.out.println("Hello Android");
11             }
12         }
13     }
14 }
```

(a) Level 3 control-flow obfuscated Java source codes

```
|[02deb8] com.example.myapplication.test.obfuscation_3:()V
|0000: sget v0, Lcom/example/myapplication/test;.g_b:I // field@2c1b
|0002: rem-int/lit16 v0, v0, #int 128 // #0080
|0004: const/4 v1, #int 1 // #1
|0005: if-ne v0, v1, 0015 // +0010
|0007: sget v0, Lcom/example/myapplication/test;.g_a:I // field@2c1a
|0009: rem-int/lit8 v0, v0, #int 64 // #40
|000b: const/4 v1, #int 2 // #2
|000c: if-ne v0, v1, 0015 // +0009
|000e: sget-object v0, Ljava/lang/System;.out:Ljava/io/PrintStream; // field@3e54
|0010: const-string v1, "Hello Android" // string@0448
|0012: invoke-virtual {v0, v1}, Ljava/io/PrintStream;.println:(Ljava/lang/String;)V // method@0128
|0015: return-void
```

(b) Bytecodes compiled from the Java source codes

```
|[0641d0] com.example.myapplication.test.obfuscation_3:()V
|0000: sget v0, Lcom/example/myapplication/test;.g_b:I // field@2c1b
|0002: rem-int/lit16 v1, v0, #int 128 // #0080
|0004: const/4 v0, #int 1 // #1
|0005: if-ne v1, v0, 0015 // +0010
|0007: sget v0, Lcom/example/myapplication/test;.g_a:I // field@2c1a
|0009: rem-int/lit8 v1, v0, #int 64 // #40
|000b: const/4 v0, #int 2 // #2
|000c: if-ne v1, v0, 0015 // +0009
|000e: sget-object v1, Ljava/lang/System;.out:Ljava/io/PrintStream; // field@3e54
|0010: const-string v0, "Hello Android" // string@0448
|0012: invoke-virtual {v1, v0}, Ljava/io/PrintStream;.println:(Ljava/lang/String;)V // method@0128
|0015: return-void
```

(c) Bytecodes optimized by *ReDex*

FIGURE 2. Example of level 3 control-flow obfuscation. The opaque variables and opaque predicates cannot be removed by *R8* and *ReDex*.

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. 2(b) and Fig. 2(c) show the optimized Dalvik bytecodes optimized by *R8* compiler and *ReDex*, repectively. 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)

Android's Dalvik Virtua Machine (DVM) was replaced with Android runtime (ART) from Android 5.0 and ART adopts Ahead-of-Time (AOT) compilation. AOT compilation statically converts all codes to machine codes at installation time, while Just-in-Time (JIT) compilation analyzes Dalvik bytecodes and translates hot spots into optimized native codes during runtime [36]–[39]. Modern ART includes a JIT compiler with code profiling to improve runtime performance of apps [40]–[42]. The advantages and disadvantages of DVM and ART are described in Table 1.

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

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]. 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 (you can find control-flow graph examples in APPENDIX). 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

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. 3 shows the transformation of Java source code when control-flow obfuscation is applied using *DexGuard*. The original code (Fig. 3(a)) is a simple `onCreate()` method without any operation instructions or branch/conditional statements, but the obfuscated code (Fig. 3(b) contains several operation instructions and branch/conditional statements are inserted.

A pair of variables `f65` and `f66`, declared as `private static int` in the class in Fig. 3(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

```
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 = 0;

public void onCreate(Bundle bundle) {
    int i = f66 + 125;
    f65 = i % 128;
    if (i % 2 == 0) {
    }
    super.onCreate(bundle);
    addPreferencesFromResource(R.xml.f140);
    i = f66 + 71;
    f65 = i % 128;
    switch (i % 2 == 0 ? 13 : 87) {
        case 13:
            Object[] objArr = null;
            int length = objArr.length;
            return;
        default:
            return;
    }
}
```

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

FIGURE 3. An original method before obfuscation (a) and the method from which the original method was obfuscated by *DexGuard*.

`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`).

D. ReDex OPTIMIZER

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. 4 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 recoreded 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. This module answers research question 1 in Sec.I. Fig. 5 shows a part of method `onCreate()` which is control-flow obfuscated using *DexGuard* (Fig. 3(b)). In Fig. 5, `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. 6 shows the HIR for the code snippets in Fig. 5. *Deoptfuscator* utilizes this HIR to analyze the variable usage pattern, remove global opaque variables effectively and simplify the control-flow. In Fig. 6, '`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 [l8]`' means that this instruction gets a Java int variable from the field area of the class referred by `l8`.

Fig. 7 shows the HIR instructions converted from obfuscated Java source codes in the example. We explain each Java statement (S1 $\sim$ S4) and its corresponding HIR instructions in a *DexGuard*'s obfuscation pattern.

S1 Get a reference to the class `(l8)` 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 `(l8)`.

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. 8 shows the internal representation for the HIR given in Fig. 6. 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.

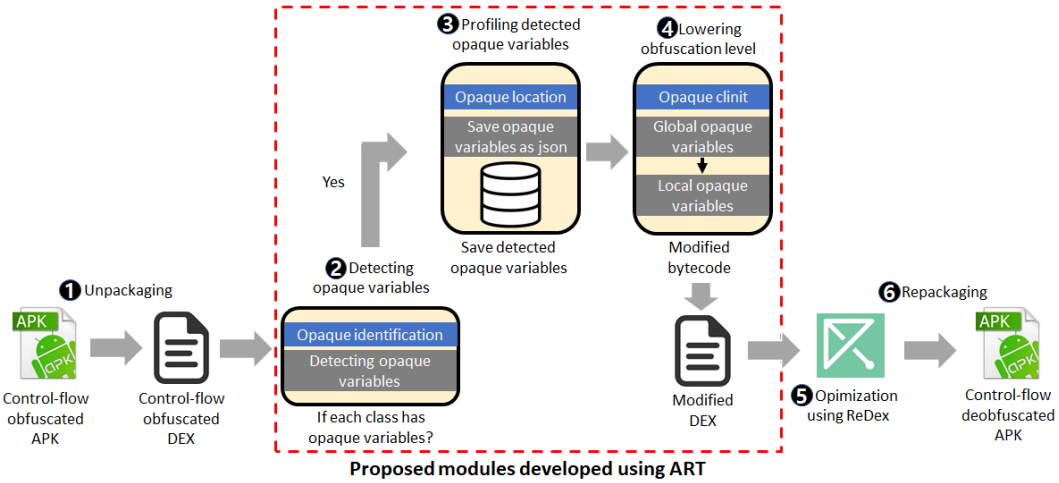


FIGURE 4. Deobfuscation process of *Deoptfuscator*.

```
private static int f65 = 1;
private static int f66 = 0;

public void onCreate(Bundle Bundle)
{
    int i = f66 + 125;
    f65 = i % 128;
    if ( i % 2 == 0 ) {
    }
       ....
```

FIGURE 5. The code snippets from Fig. 5(b).

```
BasicBlock 0, succ: 1
 j7: CurrentMethod
 i10: IntConstant
 i12: IntConstant
 i16: IntConstant
 i18: IntConstant
 i25: IntConstant
 i30: IntConstant
 i39: IntConstant
 i42: IntConstant
 v4: Goto
BasicBlock 1, pred: 0, succ: 9,10
 l8: LoadClass [j7]
 i9: StaticFieldGet [l8]
 i11: Add [i9,i10]
 i13: Rem [i11,i12]
 v15: StaticFieldSet [l8,i13]
 i17: Rem [i11,i16]
 z19: NotEqual [i17,i18]
 v20: If [z19]
```

FIGURE 6. HIR for the code snippets in Fig. 7.

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(i9)`, 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(i9)` → `LoadClass(l8)` → `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(i9)` → `Add(i11)` → `Rem(i13)` → `StaticFieldSet(v15)`, we discover that the result `i13` of the operations is stored to a global variable via instruction `StaticFieldSet(v15)`. *Deoptfuscator* records the locations of `v15` and `i13`. Thereby, `i9` and `i13` 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 `i9`, `i11`, and `i13` are not used in any other part of the current method.

Fig. 9 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

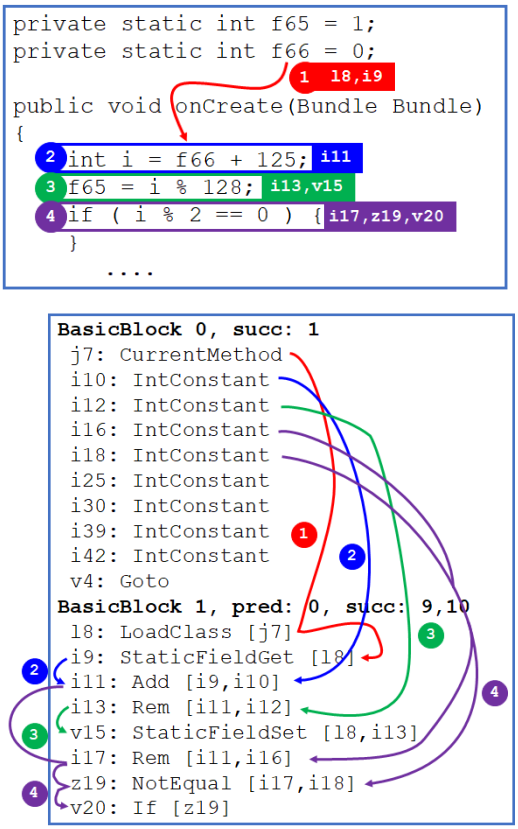


FIGURE 7. Java source code to HIR conversion.

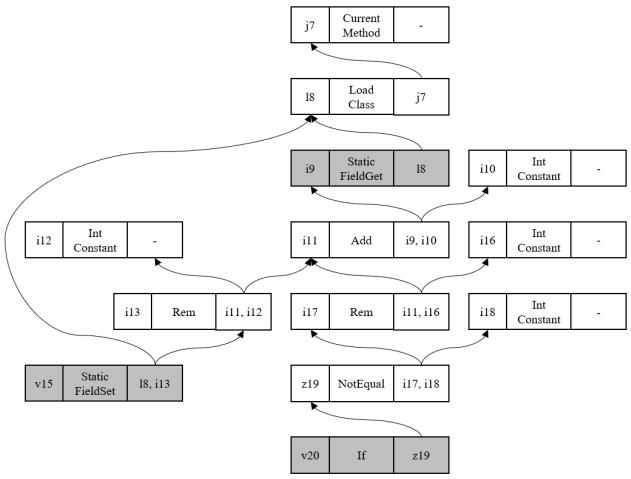


FIGURE 8. Internal representation of HIR.

global variables and the local variable are used only in the obfuscation patterns. We can confirm that `i9` and `i13` 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.

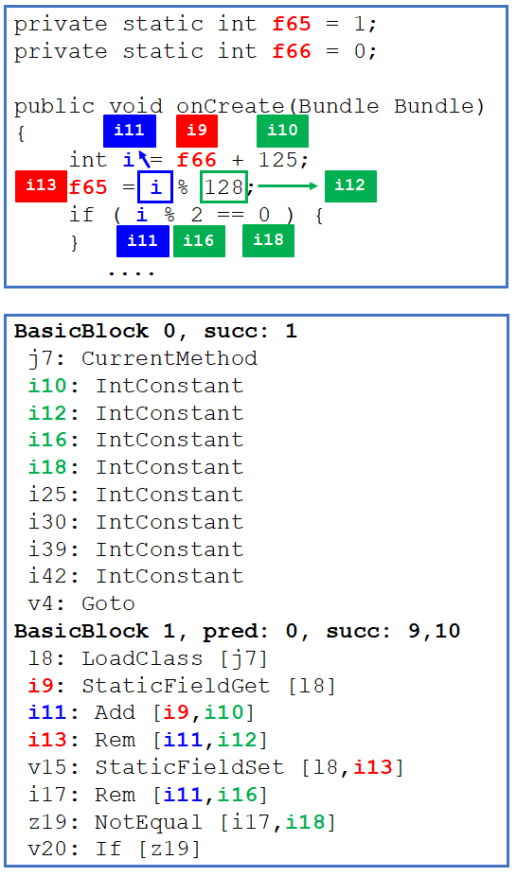


FIGURE 9. Location of opaque variables in HIR.

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 *Obfuscated Bytecode Ratio (OBR)* and is defined as follows:

$$OBR = \frac{\sum_m N_m}{\sum_m L_m} \tag{1}$$

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 `l8` to `v20` in Fig. 8 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. 10. For example, consider a class C with two methods $m1$ and $m2$. Assume *Deoptfuscator* detected one obfuscation pattern in $m1$ and two in $m2$, and that the obfuscation patterns are the same as Fig. 10 (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 $N_{m1} + N_{m2} = 24 + 2 \times 24 = 72$ bytes.

```
00000016  goto                :4C
:18
00000018  sget                v0, iF->f66:I
0000001C  add-int/lit8        v0, v0, 0x7D
00000020  rem-int/lit16       v1, v0, 0x0080
00000024  sput                v1, iF->f65:I
00000028  rem-int/lit8        v0, v0, 2
0000002C  if-nez              v0, :32
:30
00000030  goto                :8
```

FIGURE 10. Android bytecode representation of the detected obfuscation pattern.

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 $m1$ and $m2$ of class C be 100 and 200, respectively. Then $L_{m1} + L_{m2} = 100 + 200 = 300$.

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 θ, *Deoptfuscator* regards the class as obfuscated and deobfuscates it. Otherwise, the detected obfuscation pattern, if any, is regarded as false positive.

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 `clinit`. The `Opaque clinit` changes the instruction to read a global variable (`sget`) and the instruction to write a value to the global variable ('`sput`') to the instruction to assign or get a value of a local variable ('`const/16`'). Then, the `Opaque 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. 11 shows the Java code decompiled from the deobfuscated version of `onCreate()` of Fig. 3(b). You can see that the code of the method has been restored to the same as the original (Fig. 3(a)).

```
public void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    addPreferencesFromResource(R.xml.f140);
}
```

FIGURE 11. The deobfuscated method by *Deoptfuscator*, which is in Fig. 3(b).

In summary, `Opaque clinit` module lowers level 3 control-flow obfuscation to level 2 and *ReDex* eliminates level 2 obfuscation. This answers research question 2.

IV. EXPERIMENTAL SETUP

A. DATASET FOR EVALUATION

We used the Android apps that F-Droid project collected in our experiment. We randomly selected 102 apps among 1426 F-Droid collection (we call these set of apps original in this paper). We generated two more apps for each selected app using *DexGuard*. The first app is generated by performing moderate level obfuscation and optimization, and we call this app a moderately obfuscated app. A high degree of obfuscation and optimization is performed to generate the second app, which is called a highly obfuscated app. After running the original, moderately obfuscated, and highly obfuscated apps on Android Virtual Device (AVD) and physical smartphones (Pixel 2 XL with Android Oreo 9.0), we selected 63 final apps that all three versions were successful.

B. EXPERIMENTAL METHOD

We deobfuscated 63 highly obfuscated apps and 63 moderately obfuscated ones using our proposed To determine how well the *Deoptfuscator* perform, first, we checked whether the deobfuscated app is installed on the AVD and physical smartphone and works successfully. We manually checked whether the app was installed successfully, whether it ran, and whether configuration settings were possible and so on. For example, we installed a calculator app, calculated some

expressions, and changed the settings to a scientific mode or a unit conversion mode.

Next, 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 generated 63 optimized apps for highly obfuscated apps and moderately obfuscated ones, respectively.

The *Deoptfuscator*'s performance depends on the threshold of Eq. 1. The threshold is obtained by experience. Based on the threshold of 0.15, the values of 0.225, 0.075, and 0.015, which are 1.5 times, 0.5 times, and 0.1 times 0.15, were selected. A total of 504 apps were generated, 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. 12).

- 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)

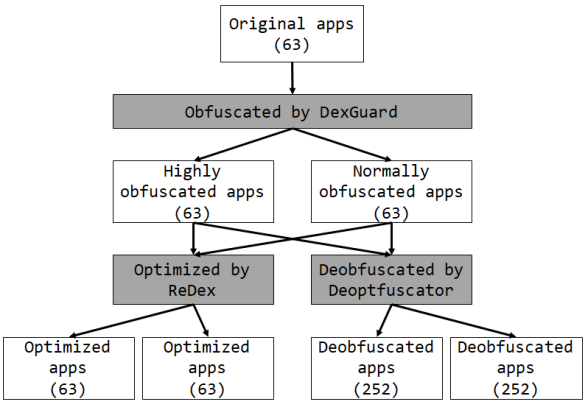


FIGURE 12. All apps used in the experiments.

We compared the deobfuscation performance based on the following criteria.

- Size of dex file
- The number of methods
- The number of basic blocks
- The number of edges in Control-flow Graph (CFG)
- The number of basic blocks per method in CFG
- The number of edges per method in CFG
- Size of insns

Using the *Androsim* module of *Androguard* [50], we measured the similarity between the deobfuscated app and the original app. *Androsim* expresses the bytecode extracted from the Android app as a string and compares the similarity between the two apps on a method-by-method basis.

By comparing the code optimized with *ReDex*, which can be considered as a minimal deobfuscation tool, and measuring the similarity to the original app, it is possible to judge how successful *Deoptfuscator* is. This will answer the research question 3.

V. PERFORMANCE EVALUATION

To measure the deobfuscation ability of the proposed tool, we analyzed the characteristics of the apps according to the criteria mentioned above. The numerical value presented is a normalized value based on the value of the original app.

A. HIGHLY OBFUSCATED APPS

Fig. 13 shows the results of measurements for highly obfuscated apps. In the legend of the figure and table, '*original*' is the original apps, '*DexGuard*' is the obfuscated apps, '*ReDex*' is the apps optimized by *ReDex*. 'θ=n' represents the apps deobfuscated by setting the threshold to n.

Looking at the number of methods in the highly obfuscated app, we can see that it has decreased by about 30% compared to the original app. In contrast, the number of basic blocks and the number of CFG edges increase significantly by 8.23 times and 12.8 times, respectively. Naturally, it can be seen that the number of basic blocks and edges for each method also increases 11.5 times and 17.8 times, respectively, and 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 θ=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.

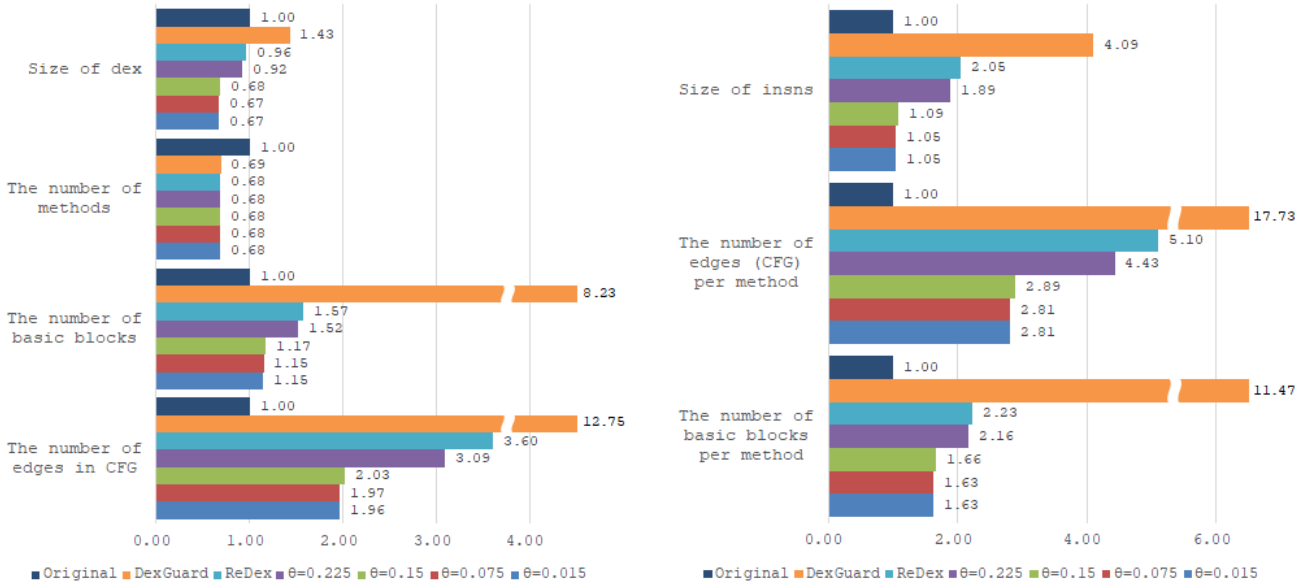


FIGURE 13. Comparison of performance using *Deoptfuscator* and *ReDex* for highly obfuscated apps.

As shown in Fig. 13, if the threshold is greater than 0.15, the effect of intrinsic deobfuscation almost disappears.

B. MODERATELY OBFUSCATED APPS

Fig. 14 shows 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.

Fig. 15 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.

Fig. 16, which compares the similarity of a moderately obfuscated app, shows the same pattern although the figure is slightly higher. A high number indicates that the strength of the obfuscation is weaker than that of the highly obfuscated case.

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`, (2) restore the original strings of encrypted ones by analyzing string encryption and decryption processes, (3) obtain the

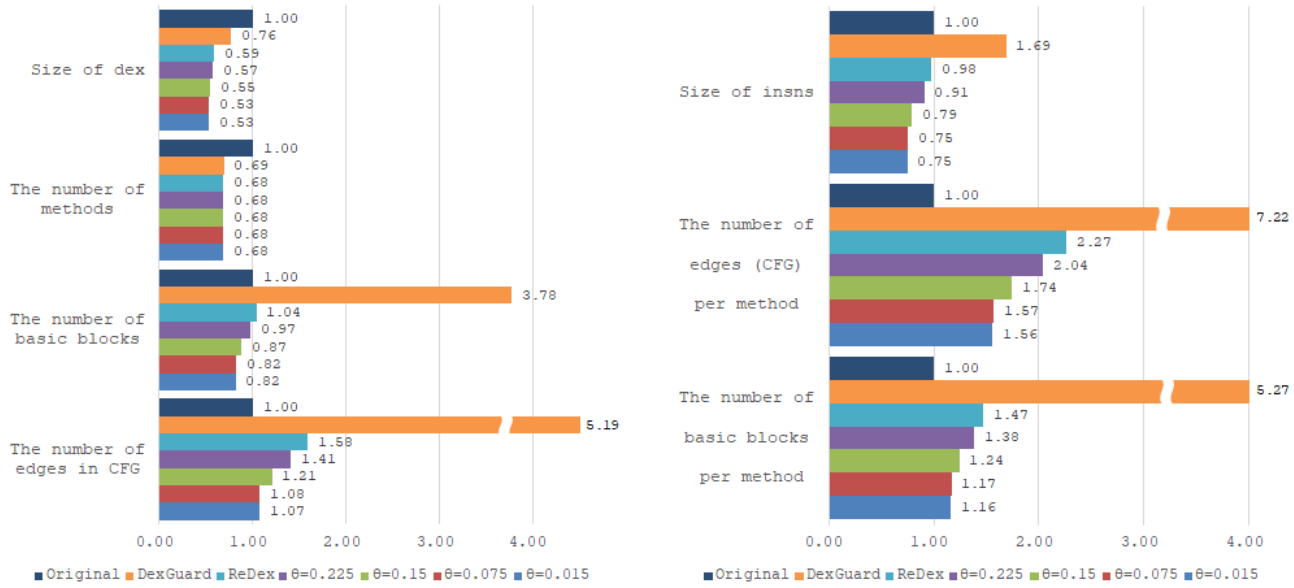


FIGURE 14. Comparison of performance using *Deoptfuscator* and *ReDex* for moderately obfuscated apps.

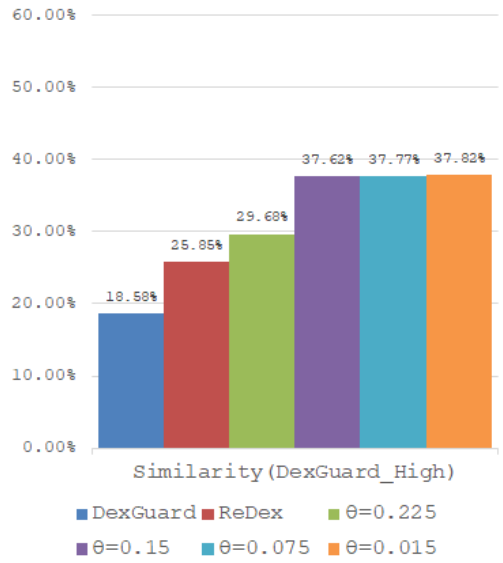


FIGURE 15. Comparison of similarity for highly obfuscated apps (`DexGuard_High`).

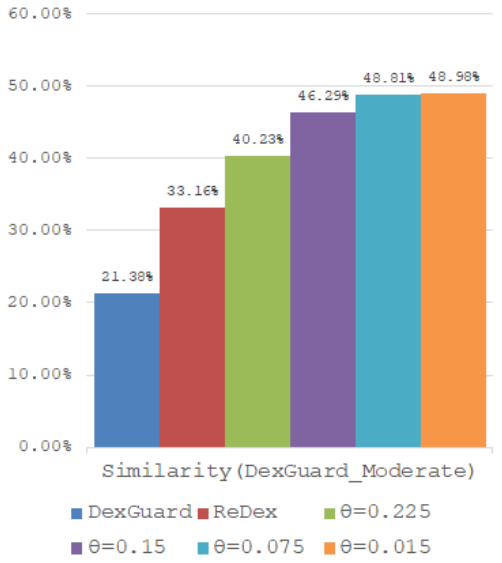


FIGURE 16. Comparison of similarity for moderately obfuscated apps (`DexGuard_Moderate`).

disassembled smali code logic that is the same as original source code logic by breaking class encryption feature, and (4) remove the tamper detection routines or skip it and remake a fake app. They mentioned that the hardest parts of analyzing the weakness of *DexGuard* was to remove the opaque predicates or understand the reordered opcodes generated by control-flow randomization. Finally, they presented a server-based obfuscation technique to securely protect the encrypted classes and the tamper detection routine.

Simplify [51] uses a virtual machine sandbox for executing an app to understand its behavior. *Simplify* analyzes the execution graphs from the virtual machine sandbox and applies optimizations such as constant propagation, dummy code removal, reflection removal, etc. If these optimizations are applied together repeatedly, they will decrypt strings, remove reflection, and simplify code that is easier for humans to understand. *Simplify* does not rename identifiers.

We conducted some experiments with *Simplify* for the obfuscated apps used in this paper. As an output, *Simplify* produced only 9 deobfuscated apps of the 63 `DexGuard_High` apps and 16 deobfuscated apps of the 63 `DexGuard_Moderate` apps, and an error occurred during deobfuscation for the remaining apps. Since *DexGuard* changed the identifier name to special characters, it seems that the error occurred due to the failure of the string processing. In the case of the 25 apps deobfuscated by *Simplify*, the control-flow obfuscation of *DexGuard* could

not be handled, and all of them did not run in our experimental environment (AVD and real smartphone). Therefore, we do not include these experiment results in the performance evaluation.

DeGuard [52] and *Anti-ProGuard* [53] are deobfuscation tools which targets identifier renaming obfuscation applied by *ProGuard*. Bichsel *et al.* [52] developed *DeGuard*, a statistical deobfuscation tool for Android which could reverse the layout obfuscation performed by *ProGuard* and rename obfuscated program elements of Android malware. Their approach phrases the problem of predicting identifier names renamed by the layout obfuscation as a structured prediction with probabilistic graphical models for identifiers that are based on the occurrence of names. *DeGuard* predicted 79.1% of the obfuscated program elements for open-source Android apps obfuscated by *ProGuard* and Android malware samples.

Anti-ProGuard [53] also aims to deobfuscate the identifier renaming technique. It requires smali files as input, and then uses a database storing obfuscated snippets and their original counterparts. *Anti-ProGuard* employs similarity hashing not pursuing exact matches for accuracy improvement. It could successfully identify over 50% of known packages in Android apps.

Java-Deobfuscator [54] is a tool that deobfuscates obfuscated Java bytecodes and makes them much readable. It can handle Java bytecodes (JAR) obfuscated by commercially available Java obfuscators such as *Zelix*, *Klassmaster*, *Stringer*, *Allatori*, *DashO*, *DexGuard*, etc. Since *Java-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 obfucation techniques.

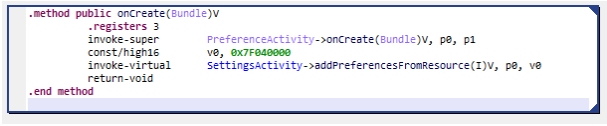
In our previous work, we analyzed the performance of tools for obfuscating, deobfuscating, and optimizing Android apps [15]. We chose *R8* compiler and *Obfuscapk* for obfuscators, *DeGuard* for a deobfuscator, and *R8* compiler and *ReDex* for optimizers. As the default compiler for Android apps, *R8* 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 *Obfuscator-LLVM* (*O-LLVM*). *O-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.

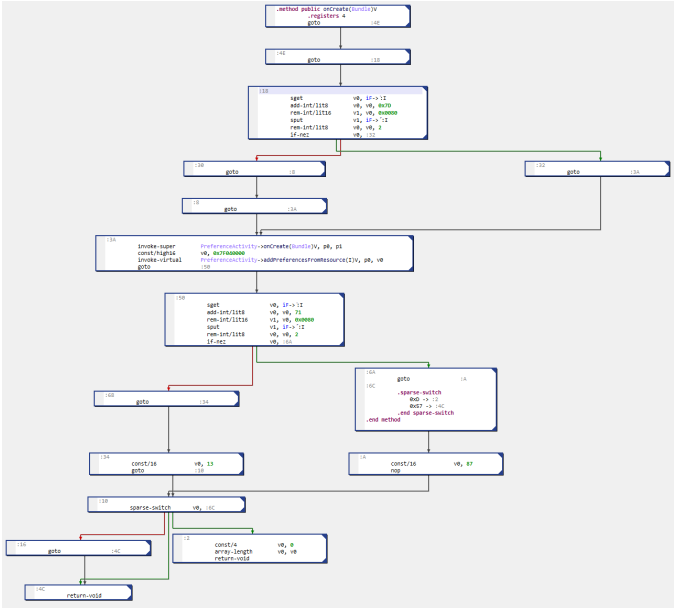
On the one hand, Ming *et al.* [34] tried to detect obfuscation techniques based on opaque predicates. Pointing out that existing researches were not sufficient to detect opaque predicates in terms of generality, accuracy, and obfuscation-resilience, They suggested a Logic Oriented Opaque Predicate (*LOOP*) 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

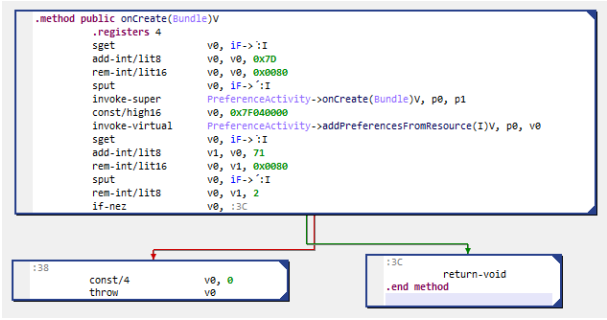
In our previous work [15], we compared optimizers and deobfuscators for Android apps, and evaluated their performance. Program optimization is a technique aimed at improving program execution speed by reducing the use of resources as well as by eliminating redundant instructions, unnecessary branches, and null-checks. On the other hand, program deobfuscation focuses on removing or mitigating the obfuscation techniques applied to the program and restore



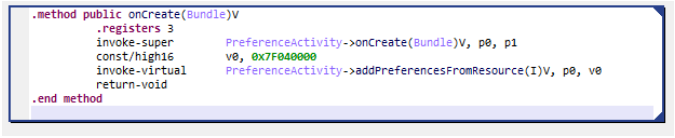
(a) The control-flow graph of an original code



(b) The control-flow graph of the code obfuscated from the original with *DexGuard*



(c) The control-flow graph of the code optimized from the obfuscated with *ReDex*



(d) The control-flow graph of the code deobfuscated from the obfuscated with *Deoptfuscator*

FIGURE 17. 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).

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 `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 debofuscation techniques after calculating the proportion of patterns identified that control-flow obfuscation are applied among instructions within one class through *OBR*.

Our work has some limitations. The proposed technique can only handle control-flow obfuscation by *DexGuard*, and does not consider control-flow obfuscation by other obfuscators including *DashO* and *Allatori*. Codes written by developers can accidentally match *DexGuard*'s control-flow obfuscation pattern. To prevent *Deoptfuscator* from arbitrarily removing opaque variables used in these patterns, the opaque variables are selected as one of the deobfuscation candidates. Then, if it is confirmed that the opaque variable is not used outside of the obfuscation pattern, it is removed, otherwise it is not removed. Separately, we analyzed codes to see if there are any apps that accidentally contain *DexGuard*'s control-flow obfuscation pattern. No such benign apps were found within the scope of our analysis.

The experiments with 63 apps prove the effectiveness of the *Deoptfuscator*. We, however, need to experiment with a wider variety of apps and larger numbers of apps, and check if there is room for improving the performance of *Deoptfuscator*.

All apps that have been deobfuscated by *Deoptfuscator* are executable on both an AVD and a physical 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 debofuscation.

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

APPENDIX

Fig. 17 shows 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*. 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 *DexGuard*'s identifier renaming.

REFERENCES

[1] Statcounter. *Mobile Operating System Market Share Worldwide*. Accessed: Jun. 2021. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide/monthly-202002-202102

[2] Statista. *Number of available applications in the Google Play Store from December 2009 to Sep. 2021*. Accessed: Jun. 2021. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[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. 72182–72196, 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," *Softw., Pract. Exper.*, 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 Inst. Technol., Cambridge, MA, USA, Tech. Rep., 2014. [Online]. Available: https://courses.csail.mit.edu/6.857/2014/files/17-cyrbritt-webbhorn-specter-dmiao-hacking-fitbit.pdf

[6] W. Zhou and S. Piramuthu, "Security/privacy of wearable fitness tracking IoT devices," in *Proc. 9th Iberian Conf. Inf. Syst. Technol. (CISTI)*, Jun. 2014, pp. 1–5.

[7] J. Rieck, "Attacks on fitness trackers revisited: A case-study of unfit firmware security," 2016, *arXiv:1604.03313*.

[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 *Proc. 17th Int. Workshop Mobile Comput. Syst. Appl.*, Feb. 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 *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Nov. 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. Wireless Mob. Netw. 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," *Comput. Sci. Rev.*, vol. 39, Feb. 2021, Art. no. 100365.

[12] G. You, G. Kim, J. Park, S.-J. Cho, and M. Park, *Reversing Obfuscated Control Flow Structures in Android Apps Using ReDex Optimizer*. New York, NY, USA: ACM, 2020, pp. 272–276, doi: 10.1145/3426020.3426089.

[13] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proc. 25th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 1998, pp. 184–196.

[14] J. Park, H. Kim, Y. Jeong, S. Cho, S. Han, and M. Park, "Effects of code obfuscation on Android app similarity analysis," *J. Wireless Mob. Netw. Ubiquitous Comput. Dependable Appl.*, vol. 6, no. 4, pp. 86–98, Dec. 2015.

[15] G. You, G. Kim, S.-J. Cho, and H. Han, "A comparative study on optimization, obfuscation, and deobfuscation tools in android," *J. Internet Serv. Inf. Secur.*, vol. 11, no. 1, pp. 2–15, 2021.

[16] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android obfuscation techniques: A large-scale investigation in the wild," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.* Cham, Switzerland: Springer, 2018, pp. 172–192.

[17] 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.

[18] VirusTotal. *Analyze Suspicious Files, Domains, IPs and URLs to Detect Malware and Other Breaches, Automatically Share Them With the Security Community*. Accessed: Jun. 2021. [Online]. Available: https://www.virustotal.com/

[19] G. Kim, G. You, and S.-J. Cho. *Deoptfuscator: Automated Deobfuscation of Android Bytecode using Compilation Optimization*. Accessed: Aug. 2020. [Online]. Available: https://www.blackhat.com/us-20/arsenal/schedule/index.html#deoptfuscator-automated-deobfuscation-of-android-bytecode-using-compilation-optimization-19958

[20] Gyoonus. *Deoptfuscator*. Accessed: Jun. 2021. [Online]. Available: https://github.com/Gyoonus/deoptfuscator

[21] G. Arboit, "A method for watermarking Java programs via opaque predicates," in *Proc. 5th Int. Conf. Electron. Commerce Res. (ICECR)*, 2002, pp. 102–110.

[22] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Dept. Comput. Sci., Univ. Aukcland, Auckland, New Zealand, Tech. Rep. 148, 1997.

[23] A. Majumdar and C. Thomborson, "Manufacturing opaque predicates in distributed systems for code obfuscation," in *Proc. 29th Australas. Comput. Sci. Conf.*, 2006, pp. 187–196.

[24] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM—Software protection for the masses," in *Proc. IEEE/ACM 1st Int. Workshop Softw. Protection*, May 2015, pp. 3–9.

[25] Google. *Shrink, Obfuscate, and Optimize Your App*. Accessed: Jun. 2021. [Online]. Available: https://developer.android.com/studio/build/shrink-code

[26] GuardSquare. *Shrink Your Java and Android Code*. Accessed: Jun. 2021. [Online]. Available: https://www.guardsquare.com/proguard

[27] PreEmtive. *Smart App Protection—A Professional App Shielding and Hardening Solution*. Accessed: Jun. 2021. [Online]. Available: https://www.preemptive.com/

[28] Licel. *Multi-Layered RASP Solution That Secures Your Android and IOS Apps Against Static and Dynamic Analysis, Illegal Use and Tampering*. Accessed: Jun. 2021. [Online]. Available: https://dexprotector.com/

[29] GuardSquare. *Full Spectrum Protection for Android Apps*. Accessed: Jun. 2021. [Online]. Available: https://www.guardsquare.com/dexguard

[30] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *Proc. 12th Work. Conf. Reverse Eng. (WCRE)*, 2005, p. 10.

[31] 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.
[32] D. Uppal, V. Mehra, and V. Verma, "Basic survey on malware analysis, tools and techniques," *Int. J. Comput. Sci. Appl.*, vol. 4, no. 1, p. 103, 2014.
[33] Z. Kan, H. Wang, L. Wu, Y. Guo, and G. Xu, "Deobfuscating Android native binary code," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Companion (ICSE-Companion)*, May 2019, pp. 322–323.
[34] J. Ming, D. Xu, L. Wang, and D. Wu, "LOOP: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 757–768.
[35] D. Xu, J. Ming, and D. Wu, "Generalized dynamic opaque predicates: A new control flow obfuscation method," in *Proc. Int. Conf. Inf. Secur.* Cham, Switzerland: Springer, 2016, pp. 323–342.
[36] P. Faruki, H. Fereidooni, V. Laxmi, M. Conti, and M. Gaur, "Android code protection via obfuscation techniques: Past, present and future directions," 2016, *arXiv:1611.10231*.
[37] Y. Piao, J.-H. Jung, and J. H. Yi, "Server-based code obfuscation scheme for APK tamper detection," *Secur. Commun. Netw.*, vol. 9, no. 6, pp. 457–467, Apr. 2016.
[38] P. Schwermer, "Performance evaluation of Kotlin and Java on Android runtime," M.S. thesis, Dept. School Comput. Sci. Comm., KTH Royal Inst. Technol., Stockholm, Sweden, 2018.
[39] X. Xu, K. Cooper, J. Brock, Y. Zhang, and H. Ye, "ShareJIT: JIT code cache sharing across processes and its practical implementation," *Proc. ACM Program. Lang.*, vol. 2, pp. 1–23, Oct. 2018.
[40] Google. *Configuring ART*. Accessed: Jun. 2021. [Online]. Available: https://source.android.com/devices/tech/ dalvik/configure
[41] Google. *Implementing ART Just-In-Time (JIT) Compiler*. Accessed: Jun. 2021. [Online]. Available: https://source.android.com/devices/tech/dalvik/jit-compiler
[42] Google. *Android 5.0 Behavior Changes*. Accessed: Jun. 2021. [Online]. Available: https://developer.android.com/about/versions/lollipop/android-5.0-changes
[43] M. Sun, T. Wei, and J. C. S. Lui, "TaintART: A practical multi-level information-flow tracking system for Android RunTime," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 331–342.
[44] M. Backes, S. Bugiel, O. Schranz, P. Von Styp-Rekowsky, and S. Weisgerber, "ARTist: The Android runtime instrumentation and security toolkit," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS P)*, Apr. 2017, pp. 481–495.
[45] O. Schranz, "ARTist - A novel instrumentation framework for reversing and analyzing Android apps and the middleware," Black Hat USA 2018, Las Vegas, NV, USA, Tech. Rep., 2018. [Online]. Available: http://i.blackhat.com/us-18/Thu-August-9/us-18-Schranz-ARTist-A-Novel-Instrumentation-Framework-for-Reversing-and-Analyzing-Android-Apps-and-the-Middleware-wp.pdf
[46] V. Costamagna and C. Zheng, "ARTDroid: A virtual-method hooking framework on Android art runtime," in *Proc. IMPS ESSoS*, 2016, pp. 20–28.
[47] M. Y. Wong and D. Lie, "Tackling runtime-based obfuscation in Android with TIRO," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1247–1262.
[48] F. Engineering. *Optimizing Android ByteCode With ReDex*. Accessed: Jun. 2021. [Online]. Available: https://engineering.fb.com/2015/10/01/android/optimizing-android-bytecode-with-redex/
[49] F. Engineering. *Redex—An Android Bytecode Optimizer*. Accessed: Jun. 2021. [Online]. Available: https://fbredex.com/
[50] A. Desnos. *Reverse Engineering, Malware Analysis of Android Applications and More (NINJA)*. Accessed: Jun. 2021. [Online]. Available: https://github.com/androguard/androguard
[51] C. Fenton. *Generic Android Deobfuscator*. Accessed: Jun. 2021. [Online]. Available: https://github.com/CalebFenton/simplify
[52] 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.
[53] R. Baumann, M. Protsenko, and T. Müller, "Anti-ProGuard: Towards automated deobfuscation of Android apps," in *Proc. 4th Workshop Secur. Highly Connected IT Syst.*, 2017, pp. 7–12.
[54] *Java-Deobfuscator*. Accessed: Jun. 2021. [Online]. Available: https://github.com/java-deobfuscator/deobfuscator
[55] Y. Moses and Y. Mordekhay, "Android app deobfuscation using static-dynamic cooperation," presented at VB 2018, Montreal, Canada, 2018.
[56] M. de Vos and J. Pouwelse, "ASTANA: Practical string deobfuscation for Android applications using program slicing," 2021, *arXiv:2104.02612*.
[57] Z. Kan, H. Wang, L. Wu, Y. Guo, and D. X. Luo, "Automated deobfuscation of Android native binary code," 2019, *arXiv:1907.06828*.

GEUNHA YOU received the B.E. degree from the Department of Software Science and the 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 systems.

GYOOSIK KIM received the B.E. degree in applied computer engineering and the M.E. degree in computer science and engineering from Dankook University, South Korea, in 2016 and 2018, respectively. He is currently a Research Engineer with the Infra Research and Development Laboratory, Korea Telecommunication. His current research interests include computer security and software intellectual property protection.

SANGCHUL HAN received the B.S. degree in computer science from Yonsei University, in 1998, and the M.E. and Ph.D. degrees in computer engineering from Seoul National University, in 2000 and 2007, respectively. He is currently a Professor with the Department of Computer Engineering, Konkuk University. His research interests include real-time scheduling and computer security.

MINKYU PARK received the B.E., M.E., and Ph.D. degrees in computer engineering from Seoul National University, in 1991, 1993, and 2005, respectively. He is currently a Professor with Konkuk University, South Korea. He has authored and coauthored several journals and conference papers. His research interests include operating systems, real-time scheduling, embedded software, computer system security, and HCI.

SEONG-JE CHO (Member, IEEE) 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, South Korea, where he is currently a Professor with the Department of Computer Science and Engineering (Graduate School) and the Department of Software Science (Undergraduate School). He was a Visiting Research Professor at the Department of EECS, University of California, Irvine, USA, in 2001, and the Department of Electrical and Computer Engineering, University of Cincinnati, USA, in 2009. His current research interests include computer security, mobile app security, operating systems, and software intellectual property protection.