An Adaptive Approach to Recommending Obfuscation Rules for Java Bytecode Obfuscators

Yanru Peng* Yuting Chen* Beijun Shen†
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
†School of Software, Shanghai Jiao Tong University, China
{sherlloo, chenyt, bjshen}@sjtu.edu.cn

Abstract— **Bytecode obfuscation is an essential technique for protecting intellectual property and defending against Man-At-The-End (MATE) attacks to Java/Android applications. Several bytecode obfuscators have been developed for modifying or refactoring Java bytecode (.class) so that it becomes hard to understand but remains fully functional. These obfuscators usually integrate a variety of obfuscation rules, allowing obfuscation algorithms to be combined and enforced on the applications. Meanwhile, it still remains a difficulty: Given a bytecode file f, which obfuscation rule(s) need to be applied such that f can get obfuscated sufficiently?**

This paper presents ORChooser (Obfuscation Rule Chooser), an adaptive approach to recommending a small number obfuscation rules for Java bytecode obfuscators. The key idea of ORChooser is, given a bytecode obfuscator, to (1) randomly select/unselect obfuscation rules for the obfuscator, and (2) calculate the *obfuscation distance* between the bytecode before and after obfuscation. Furthermore, ORChooser takes *an iterative process* to adaptively obfuscate the bytecode file f such that the obfuscated code is far away from f.

We have implemented ORChooser and evaluated it on a state-of-the-art bytecode obfuscators: Android R8. The evaluation results clearly show the strength of ORChooser. In particular, within 5 iterations, ORChooser chose about 25% of obfuscation rules for R8, reducing more than 29% of the bytecode size. The similarity between the bytecode files before and after obfuscation is less than 27%, indicating that the ORChooser-supported obfuscators have obfuscated bytecode sufficiently and reduce its comprehensibility significantly.

Index Terms—**bytecode obfuscation, distance calculation, obfuscation rules, obfuscation assessment**

I. Introduction

Bytecode obfuscation is an essential technique for protecting intellectual property and defending against Man-At-The-End (MATE) attacks to Java applications [1], [2]. Bytecode obfuscation transforms a bytecode file (.class), say f, into another, say f', through renaming, inserting opaque predicates [3], modifying its control flow [4], [5], *etc.* The obfuscated code f' thus becomes obscure, as it contains variables with meaningless names, redundant/inversed control flow, incomprehensible functions, *etc.* Many bytecode obfuscators have also been designed and developed, including Android R8 [6], ProGuard [7], JavaGuard [8], and yGuard [9]. As Figure 1a shows, these obfuscators modify or restructure Java bytecode so that it becomes hard to understand but remains fully functional.

Every mature obfuscator, such as ProGuard [10] or Android R8 [11], supports a variety of obfuscation rules. Each rule corresponds to either an algorithm designed for obfuscating the objective bytecode, or a property the resulting code needs to hold. For instance, R8 can take some obfuscation rules in Figure 1b. These rules can either specify the libraries on which the obfuscated code depends (*e.g.*, `-libraryjars`), or the strategies the obfuscator takes to obfuscate (*e.g.*, `-useuniqueclassmembernames`). The obfuscated bytecode, which is wrapped in a .dex (Dalvik Executable) file, becomes obscure and its size shrunk.

Meanwhile, it still remains a problem when an engineer chooses and employs an obfuscator:

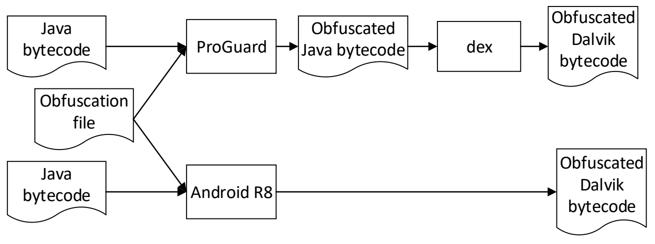
Problem Description: *Given a bytecode file f, which obfuscation rule(s) need to be applied such that f can get obfuscated sufficiently?*

Indeed, the rules significantly increase the obfuscation space. For instance, ProGuard supports 59 obfuscation rules. Let each rule be selected and/or unselected and the enforcement of the rules be unsorted. The obfuscation space will be at least 2^{59}, which is beyond the capillarities of many human engineers. On the other hand, illegal bytecode can be created due to inappropriate combinations of rules. For example, as Figure 2 shows, when some `-keep` rules are missing, R8 may obfuscate an application incorrectly—the application will crash at runtime and output a `ClassNotFoundException`.

Instead, some engineers prefer to apply all of the rules, expecting that the bytecode can thus get obfuscated much more sufficiently. However, it may also raise another two problems. First, the efficiency of the obfuscation process can get decreased, as the process does contain many trivial code transformations. Second, some rules are dependent on the others or conflict to each other. A complete list of obfuscation rules may not help the obfuscator achieve the mostly incomprehensible code. A detailed evaluation will be given in Section V.

To solve the above problems, we propose in this paper ORChooser (Obfuscation Rule Chooser), an adaptive approach to recommending obfuscation rules for Java bytecode obfuscators. The key idea of ORChooser is, given a bytecode obfuscator, to (1) randomly select/unselect obfuscation rules for the obfuscator, and (2) calculate the *obfuscation distance* between the bytecode before and after obfuscation. Furthermore, ORChooser takes *an iterative process* to adaptively obfuscate bytecode and maximize the obfuscation distance.

This paper makes the following contributions:



(a) Some typical obfuscation processes. Note that (1) R8 takes the ProGuard obfuscation rules; (2) R8 produces .dex files running on Android runtime.

```
-injars Sum.jar
-libraryjars lib/rt.jar
-keepclasseswithmembers class * {
public static void main(java.lang.String[]);
}
-ignorewarnings
-allowaccessmodification
-useuniqueclassmembernames
-overloadaggressively
-repackageclasses ''
-keepparameternames
```

(b) Obfuscation rules.

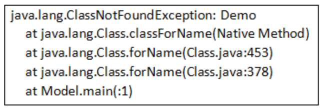
Fig. 1: Bytecode obfuscation process and rules.

```
//Demo.java
public class Demo{
  //do sth.
}

//EmptyClass.java
public class EmptyClass{
  //do sth.
}

//Model.java
public abstract class Model{
  public static void main(String[]
          args){
  Class<?> demo1 = null;
  try{
      demo1 = Class.forName("Demo"
          );
  } catch (Exception e){
    e.printStackTrace();}
  System.out.println("ClassName
        " + demo1.getName());
  EmptyClass ec = new EmptyClass()
        ;}}
```

(a) Original source

```
.class public abstract LModel;
.super Ljava/lang/Object;\label{
        sec:app}
.source ""
// direct methods
.method public static main([Ljava/
        lang/String;)V
  ...
.end method

.class public La;
.super Ljava/lang/Object;
.source ""
// direct methods
.method public constructor <init
        >()V
  ...
.end method
```

(b) Obfuscated code



```
java.lang.ClassNotFoundException: Demo
   at java.lang.Class.classForName(Native Method)
   at java.lang.Class.forName(Class.java:453)
   at java.lang.Class.forName(Class.java:378)
   at Model.main(:1)
```

(c) Error message

Fig. 2: A simple example of obfuscation conflict.

1) **Fitness function**. ORChooser employs a fitness function, which calculates the obfuscation distance between the bytecode before and after obfuscation. Such a distance can thus be adopted to evaluate whether a bytecode file has been obfuscated sufficiently: the larger the distance, the obscurer the obfuscated code, and the more sufficient the obfuscation. Having the fitness function, we thus cast the difficult problem of choosing rules as an optimization problem.
2) **Approach**. ORChooser is an adaptive approach to recommending obfuscation rules. It selects and/or unselects obfuscation rules, and takes an iterative process for accepting/rejecting the selections. An improved genetic algorithm is designed for speeding up the process and maximizing the obfuscation distance between the original and the obfuscated bytecode.
3) **Implementation and evaluation**. We have implemented ORChooser and evaluated it on a state-of-the-art bytecode obfuscators: Android R8. The evaluation results clearly show the strength of ORChooser. In particular, within 5 iterations, ORChooser chose about 25% of obfuscation rules for R8, reducing more than 29% of the bytecode size. The similarity between the bytecode files before and after obfuscation is less than 27%, indicating that the ORChooser-supported obfuscators have obfuscated bytecode sufficiently and reduce its comprehensibility significantly.

The remainder of the paper is organized as follows: Section II explains the main functionalities of obfuscators and obfuscation rules. Section III formalizes the notion of obfuscation distance. Section IV presents the details of the ORChooser approach. Section V evaluates ORChooser on R8. Section VI surveys related work and Section VII concludes.

II. Background

As a variety of bytecode obfuscators exist and it might not be possible for ORChooser to support all of them, we choose two state-of-the-art obfuscators, ProGuard and R8, for explaining the principles of ORChooser. Next describes the bytecode obfuscators and their functionalities followed by a detailed explanation about the obfuscation rules.

A. *Bytecode Obfuscator*

A typical obfuscator, such as ProGuard, does not only obfuscate code, but also shrinks, optimizes, and pre-verifies the obfuscated code. ProGuard obfuscates code by renaming the classes, fields, and methods using short, meaningless names, shrinks bytecode by detecting and removing unused classes, fields, methods and attributes, and optimizes the resulting bytecode using some classical data flow analysis techniques. Thus ProGuard can make the code small, efficient and difficult to understand.

Android R8 is a Java program shrinking and minification tool that converts Java bytecode to optimized dalvik bytecode. R8 is supported on Android Studio 3.2+ as a replacement to ProGuard. In particular, it does all of shrinking, desugaring and dexing in one step; it also supports tree-shaking the program to remove unneeded code and supports minification of the program names to reduce the code size.

B. *Obfuscation Rules*

ProGuard and R8 accept the same obfuscation rules. These rules, as Table I shows, can be divided into four categories: (1) core rules, (2) application-specific rules, (3) dependence rules, and (4) the others.

TABLE I: Classification of ProGuard's obfuscation rules.

Category	Rules
Core Rules	-skipnonpubliclibraryclasses, -forceprocessing, -target, -dontskipnonpubliclibraryclasses, -overloadaggressively, -dontskipnonpubliclibraryclassmembers, -dontshrink, -dontoptimize, -optimizationpasses, -dontobfuscate, -allowaccessmodification, -mergeinterfacesaggressively, -keepparameternames, -dontusemixedcaseclassnames, -useuniqueclassmembernames, -flattenpackagehierarchy, -repackageclasses
Application Specific Rules	-keepdirectories, -keep, -keepclasseswithmembers,-if, -keepnames, -keepclassmembers, -keepclassmembernames, -keepclasseswithmembernames, -assumenosideeffects, -assumenoexternalsideeffects, -assumenoescapingparameters, -assumenoexternalreturnvalues, -keeppackagenames, -keepattributes, -adaptclassstrings, -adaptresourcefilenames, -adaptresourcefilecontents
Dependence Rules	-include, -basedirectory, -injars, -outjars, -libraryjars
Other Rules	-printseeds, -printusage, -whyareyoukeeping, -printmapping, -printconfiguration,-applymapping, -obfuscationdictionary, -classobfuscationdictionary, -packageobfuscationdictionary, -renamesourcefileattribute, -dontpreverify, -microedition, -android, -verbose, -dontnote, -dontwarn, -ignorewarnings, -dump, -addconfigurationdebugging

a) Core Rules.: Some rules are generally purposed. For example, the rule `-overloadaggressively` specifies that the obfuscator needs to aggressively overload when obfuscating—one name may be assigned to multiple fields and methods, making the program difficult to read. These rules also support the obfuscator to shrink the obfuscated code.

b) Application Specific Rules.: Some `-keep` rules specify which classes and class members (fields and methods) need to be preserved. It is used to resolve rule conflicts. For example, Figure 3 shows that the obfuscating rules are disabled for `ClassA` but enabled for the others.

c) Dependence Rules.: ProGuard and Android R8 must take `-injars`, `-outjars`, and `-libraryjars` rules for specifying inputs, outputs, and libraries used, respectively. At least one `-keep` rule must be added. In addition, an obfuscator must specify the entry point of an application (*e.g.*, a class with the `main` method); if the entry point is not specified, ProGuard fails to output, and R8 refuses to obfuscate.

d) Other Rules.: Other rules are used for debugging, rather than for obfuscation. For example, when `-printmapping` is taken, the obfuscator records a name mapping between bytecode before and after obfuscation.

Note that R8 does not support all of the ProGuard rules [11]. For example, `-skipnonpubliclibraryclasses` requires non-public classes to be skipped when library jars are met, and thus the obfuscator can be accelerated and memory usage be reduced [12]. However, R8 has not yet supported this rule.

III. Obfuscation Distance

We next formalize the notion of bytecode obfuscation followed by defining obfuscation distance.

Definition 1: (**Bytecode Obfuscation**) A bytecode obfuscation $P \xrightarrow{\Delta} P'$ is a transformation of a bytecode file P to a target P'. Here P and P' need to be semantically equivalent, but syntactically different. If P and P' are semantically different, we call Δ an over obfuscation.

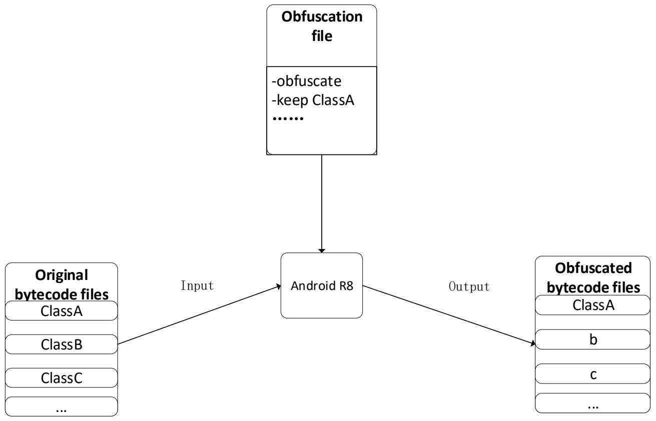


Fig. 3: Rules are enabled for all the classes except ClassA, since a `-keep` rule is used.

Meanwhile, in practice it is not easy, and many times not possible, to determine two bytecode files semantically equivalent. Thus we assume that each obfuscator runs correctly, but uses *obfuscation distance* to measure the obfuscation.

Definition 2: (**Obfuscation Distance**) An obfuscation distance between two bytecode files P and P', say $dis(P, P')$, represents the syntactical and semantical differences between the two files.

In this paper, we let the obfuscation distance be associated with (1) the differences between the two bytecode files in their constructs (*e.g.*, classes and methods), (2) the differences between their function call graphs (CGs), and (3) the differences between the tokens used.

First, each program is composed of a set of classes and methods. Thus the distance can be computed using these features—class names, the number of methods, the number of fields, *etc.* Method features, such as method names, parameter types and names, and method sizes, can also be used for computing distances. For simplifying calculation, we let

$$distance_1 = \sum_i \frac{|feature^i_P - feature^i_{P'}|}{feature^i_P} \tag{1}$$

where $feature^i$ is the i_{th} feature of the bytecode file, and $|feature^i_P - feature^i_{P'}|$ computes the Jaro-Winkler distance between the string features (*e.g.*, class names) [13].

Second, let each program be abstracted into a call graph that usually contains dozens of nodes. The distance between two CGs are the edit distance (*i.e.*, the Levenshtein distance [14]). It calculates the minimum number of graph edit operations (inserting, deleting, and modifying) to transform one graph to another. Let $CG \bullet CG_1 \bullet CG_2 \bullet \cdots \bullet CG_{n-1} \bullet CG'$. Let the CG of P has $size'$ nodes. We have

$$distance_2 = \frac{n}{size'} \tag{2}$$

Third, let each program corresponds to a token set $Token$. The distance between the tokens of two programs can be calculated using

$$distance_3 = 1 - \frac{|Token_P| + |Token_{P'}| - |Token_P \cup Token_{P'}|}{|Token_P \cup Token_{P'}|} \tag{3}$$

It denotes the rate of the tokens owned by either program, but not by both.

Having the above distances, the obfuscation distance between the two bytecode files can be computed using:

$$distance = \alpha \times distance_1 + \beta \times distance_2 + \gamma \times distance_3 \tag{4}$$

Here α, β, and γ are weights that need to be tuned for each bytecode file under obfuscation. They are all set to 1 at the beginning, but will be tuned in the following process.

R8's Obfuscation. R8 obfuscates bytecode files when transforming them into a dex file. Let $P \xrightarrow{\Delta} P'$ and $P \xrightarrow{\psi} P''$ where P is a set of bytecode files, P' a set of obfuscated ones, P'' is the .dex file, and Δ is an obfuscation operation and ψ is an operation that converts Java bytecode files to a .dex file. The obfuscation distance can be calculated after P' is converted into a .dex file, or P" is decomposed into a set of Java bytecode files.

IV. Approach

This section describes the ORChooser approach to recommending obfuscation rules.

A. Overview

ORChooser takes an iterative process to select/unselect obfuscation rules. An overview of ORChooser is shown in Figure 4, which can be divided into three parts:

1) **Initialization.** This step randomly selects a set of core rules for bytecode obfuscation. We let these rules compose an initialization set.
2) **An iterative process.** ORChooser then performs an iterative process. Each iteration selects and/or unselects some obfuscation rules, and accepts or rejects the selection by taking some strategies. A random strategy or a greedy strategy can be employed here for maximizing the obfuscation distance. Here an elitism algorithm is specially designed for directing the iterative process.
3) **Conflict detection and resolution.** Some rules can be conflict with each other, which can lead to exceptions (such as `ClassNotFoundException` and `NoSuchMethodException`) at runtime. We detect and resolve rule conflicts in the rule combination.

B. Distance Calculation

Having the obfuscation distance defined, we let the original bytecode file be the center of the obfuscation space. As Figure 5 shows, each point in the obfuscation space presents an obfuscated file, and the one that is farthest from the center is the one we expect to obtain.

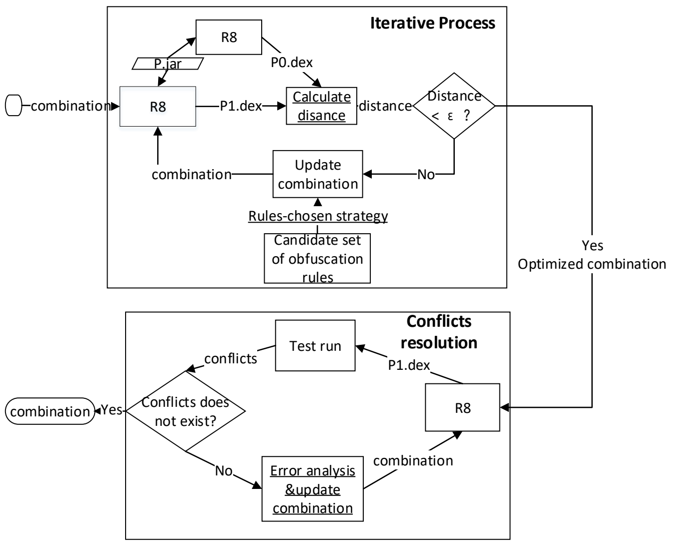


Fig. 4: An Overview of ORChooser.

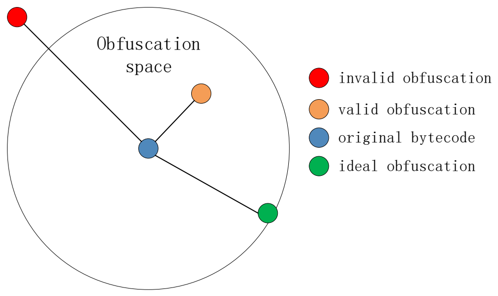


Fig. 5: An illustration of the obfuscation space.

The process of calculating obfuscation distance is show in Figure 6. It consists of three steps:

Step 1. Obtaining bytecode features. Having two bytecode files (an original file and an obfuscated one), we obtain their features. These features include class names, the method numbers of classes, method names and sizes, the parameter names and types, *etc.*

Step 2. Establishing mappings between bytecode. Potential mappings can be established on the bytecode files. A mapping process is much more like clone detection or code search—given a class or a class method in one file, it searches for the corresponding bytecode segment in another file. Some mapping records during the obfuscation process facilitates the mapping process.

Step 3. Calculating the obfuscation distance. Having these mapping files, ORChooser calculates the syntactical and token

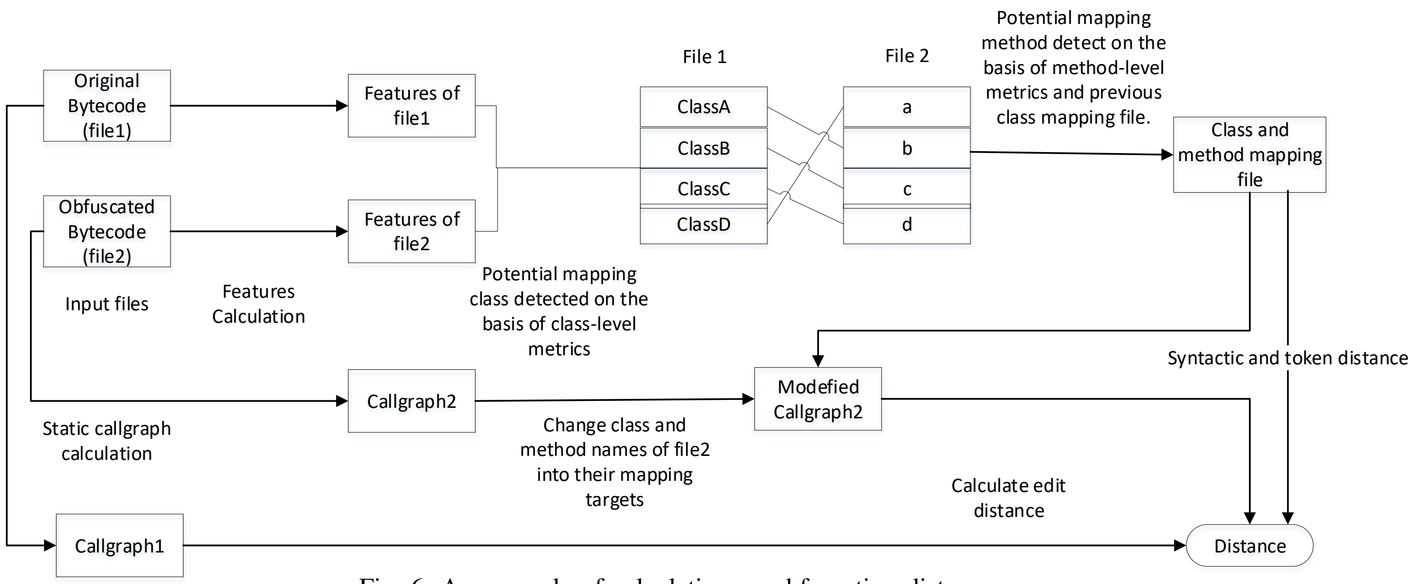

Original Bytecode (file1)
Obfuscated Bytecode (file2)
Input files
Features of file1
Features of file2
Features Calculation
File 1
File 2
ClassA
ClassB
ClassC
ClassD
a
b
c
d
Potential mapping class detected on the basis of class-level metrics
Potential mapping method detect on the basis of method-level metrics and previous class mapping file.
Class and method mapping file
Syntactic and token distance
Callgraph2
Modefied Callgraph2
Static callgraph calculation
Change class and method names of file2 into their mapping targets
Calculate edit distance
Callgraph1
Distance

Fig. 6: An example of calculating an obfuscation distance.

Algorithm 1 An Iterative Process of Selecting Obfuscation Rules

```
Input: The original program $P$; Android R8; the obfuscated program $P'$
Output: A rule combination for P
1: combination $\leftarrow$ getInitial()
2: while distance $< \epsilon$ do
3:     $P' \leftarrow$ R8($P$, combination)
4:     distance $\leftarrow$ calculateDistance($P$, $P'$)
5:     combination $\leftarrow$ selectRules()
6: end while
7: return combination
```

differences. We also build the call graphs of the two bytecode files and then calculate the edit distance between them. Note that some mappings retrieved in Step 2 can be employed for mapping the nodes of the two call graphs. The obfuscation distance can then be calculated for the bytecode before and after obfuscation.

C. An Iterative Solution

After the initial combination is obtained, ORChooser iteratively selects/unselects obfuscation rules for maximizing the obfuscation distance between the bytecode P and the obfuscated bytecode P'. The iterative algorithm is shown in Algorithm 1. Here ϵ is a user-defined threshold, `getInitial()` is to initialize the rule combination. R8 then outputs the obfuscated program P' (line 3). `calculateDistance(`P`,` P'`)` computes the obfuscation distance. `selectRules()` adds or removes obfuscation rules into the rule combination by taking an *elitism* algorithm.

The elitism algorithm is a variant of the traditional genetic algorithm [15] [16]. A new population selected by this variant

Algorithm 2 Conflict Resolution

```
Input: $Rules$, a rule combination that may raise conflicts; the original program $P$; the obfuscated program $P'$
Output: A rule combination in which conflicts have been resolved
1: while $n > 0$ do
2:     pick up by random one rule in $Rules$ and delete it
3:     $P' \leftarrow R8(Rules, P)$ //use R8 to obfuscate $P$
4:     $conflict \leftarrow dalvikvm(P')$ //run $P'$ on dalvikvm
5:     if conflict does not exist then
6:         return $Rules$
7:     else
8:         $n \leftarrow n - 1$
9:     end if
10: end while
11: return $Rules$
```

is to keep alive such that it can be carried over to the next generation [17]. This strategy is known as an elitist selection and guarantees that the solution quality obtained by the GA will not decrease in the next generation. The elitism algorithm is launched in case that the genetic algorithm converge slowly.

D. Conflict Resolution

We detect and resolve the conflicts by stepwise adjusting obfuscation rules[1]. Algorithm 2 describes the process of conflict resolution. Here n is the number of iterations. If no conflicts are revealed, the rule combination is kept. Otherwise, a rule is removed. Note that errors, once reported, facilitate engineers to reproduce and eliminate the conflicts.

[1]Mainly by adopting Application Specific Rules.

V. Evaluation

We have conducted two evaluations: one for investigating the necessaries of combining different obfuscation rules and another for evaluating the effectiveness of ORChooser on R8 (ver. 1.4.9). The both evaluations were conducted on Ubuntu16.04 x86_64 with OpenJDK 1.8.1_151 and Java HotSpot(TM) 64-Bit Server VM.

A. The First Evaluation

1) Setup: We used R8 to investigate the effectiveness of combinations of obfuscation rules. The evaluation was designed to answer the following research question:

- **RQ1.** Whether will the obfuscated code be different when different combinations of rules are employed?

Benchmark. Different rule combinations often lead to different obfuscated bytecode. In order to investigate the effects of different rules and their combinations, we prepared a mini project: it contains a standard `ArrayList.java` file and a `Test.java` file that operates an ArrayList object (*e.g.*, initialization, inserting, appending, and deleting). The project is used so that the relevance between the number of code smells and different rule combinations can be observed.

Rule combinations. We picked up three obfuscation rules (`-dontshrink`, `-dontoptimize` and `-dontobfuscate`). Thus we had eight rule combinations—each corresponds to an obfuscated code file. Next shows the combinations of the rules: 0 and 1 are to disable and enable a functionality (*i.e.*, shrinking, optimization, or obfuscation), respectively.

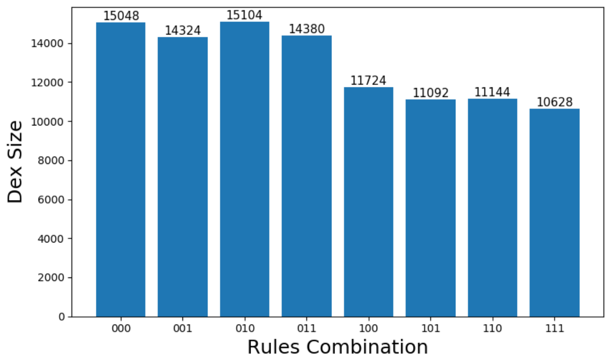
`shrink`	0	0	0	0	1	1	1	1
`optimize`	0	0	1	1	0	0	1	1
`obfuscate`	0	1	0	1	0	1	0	1
Obfuscated code	⓪	①	②	③	④	⑤	⑥	⑦

Two metrics are adopted to evaluate different rule combinations.

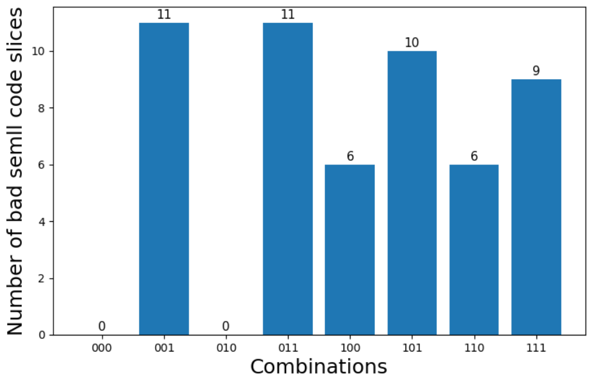
Metric 1—Dex sizes. R8 obfuscates the mini project and produced the obfuscated dex files ⓪–⑦. Dex size is chosen as a metric—R8 obfuscates, shrinks, and optimizes the code, making the code size vary.

Metric 2—Numbers of code smells. In order to check whether the obfuscated code is incomprehensible, we counted the numbers of code smells in the code files ⓪–⑦—the more code smells, the obscurer the code[2]. We adjust the checking rules, making code⓪ have no smells to eliminate the influence of decompiling process. Thus we decompiled the dex code using dex2jar [18], [19] and *JD-GUI* [20] and used `sonar` [21] to check each decompiled file. Dex2jar is a freely available tool that is commonly used in Android reverse engineering. JD-GUI is a standalone graphical utility that displays Java source codes of .class files. sonar is a code smell checking tool.

[2]The existence of code smells usually indicates that the objective system is error prone.



(a) Sizes of obfuscated Dalvik bytecode.



(b) Numbers of code smells in each decompiled code.

Fig. 7: Dex sizes and numbers of code smells.

2) Results: The results for the first evaluation are shown in Figure 7.

As Figure 7a shows, the rule for code shrinking significantly reduces the code size from 14324 bytes to 15048 bytes; code optimization slightly increases the Dex sizes by 0.37%, as it is designed for speeding up the project by sacrificing memory. When all of the three rules are taken, the size of the obfuscated Dalvik bytecode becomes the smallest and drops by 29.37%.

The numbers of code smells in the decompiled code files are shown in Figure 7b. Compared with the case that no rules are enforced (the first column in the figure), the number of code smells increases if any obfuscation rule is enabled except the one for optimization. R8 creates the most code smells when using the obfuscation rule—As long as it is enabled, the number of code smells is always larger than 9. The code smells are mostly relevant to names, *e.g.*, constant names, function names, and class names should conform to a naming convention, a field's name should not be same as the class name, *etc.*.

We draw out the first finding.

Finding 1: Different rule combinations lead to differences among obfuscated bytecode.

B. The Second Evaluation

1) Setup: The evaluation was designed to investigate the effectiveness of ORChooser. In particular, the following research questions need to be answered:

- **RQ2.** Is distance a good fitness function for directing the obfuscation process?
- **RQ3.** Which strategy is suitable for choosing rules?
- **RQ4.** Can ORChooser resolve the rule conflicts?

Benchmark. Scimark 2.0 [22] and ErsBlocks [23] are two benchmarks we chose in our evaluation. SciMark 2.0 is a Java benchmark for scientific and numerical computing. It has 2850 lines of code. SciMark 2.0 consists of five computational tasks: Fast Fourier Transform (FFT), Jacobi Successive Over-relaxation (JOR), Sparse matrix-multiply, Monte Carlo integration, and dense LU factorization. ErsBlocks is a small game in Java. It has 1300+ lines of code.

We chose Scimark 2.0 and ErsBlocks because (1) the benchmarks are independent from users, and thus they can be executed without explicit user inputs; (2) the benchmarks can be executed both on JVM and DalvikVM; and (3) the benchmarks do not contain reflection calls that can crash the benchmarks at runtime.

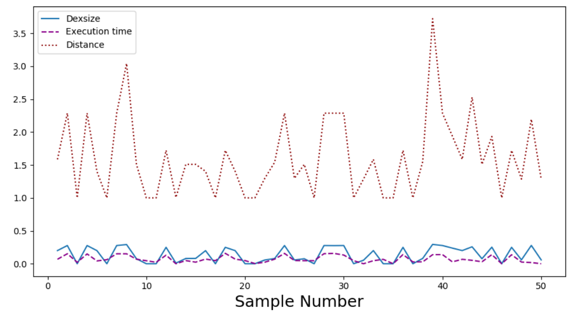
We also used SPECjvm2008 [24] to evaluate ORChooser. SPECjvm2008 is a benchmark suit for measuring the performance of a Java Runtime Environment (JRE), containing several real-world applications and benchmarks focusing on core Java functionality. The benchmarks chosen in SPECjvm2008 are `compress`, `crypto_aes`, `crypto_rsa`, `crypto_signverify` and `serial`.

Metrics. We evaluated the obfuscation quality of ORChooser in three respects: potency, resilience and cost [25], [26]. Potency shows the level of obscurity a specific obfuscator gives. Resilience measures how well an obfuscator holds up against reverse engineering attacks. Cost is the performance and size penalties incurred by the obfuscation. Metrics that are used for evaluating the obfuscation quality are: the code size, the accuracy of reverse engineering, and the execution time of an obfuscator (*i.e.*, R8).

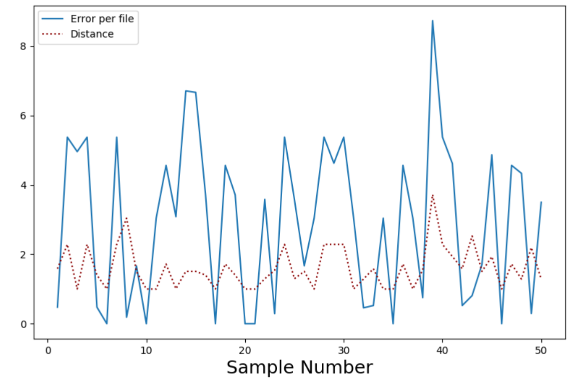
Quality Metric	Explanation	Ways to measure
Potency	Size of bytecode file and execution time	Recoding bytecode size and time consumed
Resilience	Accuracy of reverse engineering	Using Dex2jar, JD-GUI and sonar
Cost	Execution time of R8	Recording time spent

Comparisons of rule recommendation algorithms. To answer RQ3, we compared the elitism algorithm in ORChooser to a random algorithm, a greedy algorithm, and a traditional genetic algorithm.

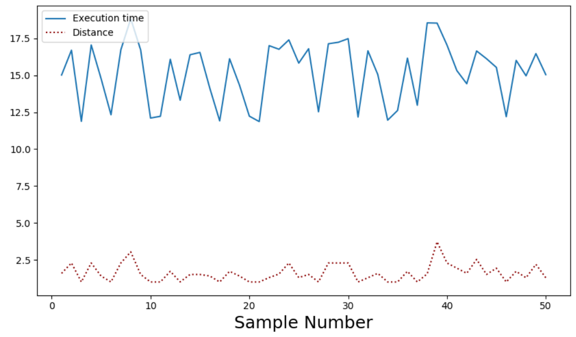
1) **A random algorithm.** The random algorithm randomly chooses obfuscation rules.
2) **A greedy algorithm.** The greedy algorithm is used to determine whether some rule selection is kept. A rule selection is accepted only when it leads to a longer obfuscation distance.
3) **A traditional genetic algorithm.** Every rule in candidate set is a gene and the whole set is taken as chromosome. With the help of crossover and mutation of gene, a diversity of rule combinations can be obtained.
4) **An elitism algorithm.** The elitism algorithm keeps the fittest organism alive, guaranteeing that it can be passed to the next generation.



(a) Potency (dex size, execution time) and distance.



(b) Resilience (Code smells per file) and distance.



(c) Cost (Execution time of R8) and distance.

Fig. 8: Distances *w.r.t.* 50 rule combinations.

2) Answers to RQ2: We used 50 randomly chosen obfuscation rule combinations. The evaluation results are shown in Figure 8.

Figure 8a shows the potency of obfuscation. Here we normalize the dex size and execution time by:

$$new\_size(i) = \frac{|original\_size - size(i)|}{original\_size}$$

where $new\_size(i)$ is the resulting size of the i_th sample, $original\_size$ is the bytecode size of unobfuscated bytecode file and $size(i)$ that of the i_th sample. By comparing the dex size, the execution time, and the distances, we can conclude that distance is strongly relevant to the obfuscator's potency.

Figure 8b shows the resilience of obfuscation that focuses on the obfuscator capabilities of preventing against MATE attacks. In this figure, except for some examples (*e.g.*, the last few samples), the changes of distances and numbers of code smells are similar. It clearly indicates that distance is relevant to the obfuscator's resilience. Exceptions may be caused because sonar is not designed for measuring obfuscation—it is able to catch code smells and measuring the comprehensibility, while not be sensitive to code obfuscation.

Figure 8c shows the cost of R8 on the 50 rule combinations. We ran each combination for five times on Ubuntu and calculated the average execution time. The figure indicates that the cost of the obfuscator can increase along with the increase of the distance. The main reason is that given an obfuscator, the higher the obfuscation level, the more time the obfuscator will spend on obfuscation.

Distance changes are consistent with the changes of dex size, execution time, numbers of code smells and cost of the obfuscator. It leads to the following finding.

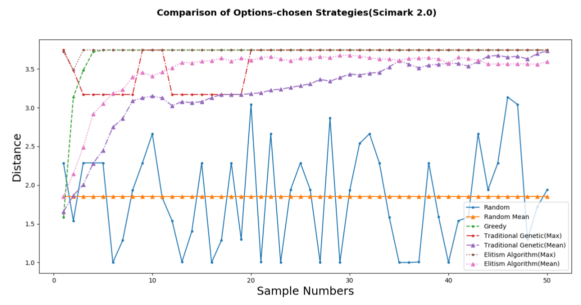
Finding 2: Distance is a useful metric for directing the obfuscation process.

3) Answers to RQ3: The results for different strategies are shown in Figure 9a and Figure 9b. The average distance of the random strategy is around 1.85 and the max is 3.74375 (let Figure 9a be an example). The greedy strategy works well and can converge within 10 iterations—the distance does not increase after 10 iterations.

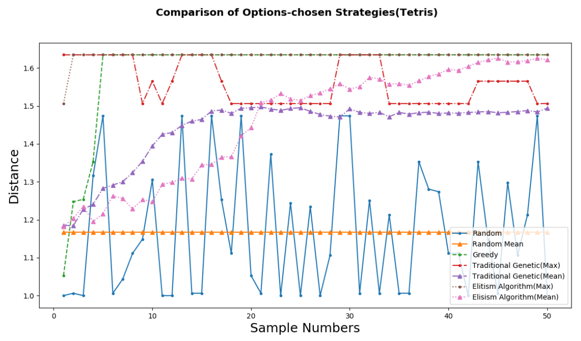
However, for the greedy algorithm, the resulting combination is not necessarily the optimal. Therefore, we apply the genetic algorithm to get a solution. The traditional genetic algorithm achieves an optimal core-rule combination after 50+ iterations. Thus we took the elitism algorithm in which the default selector of the traditional genetic algorithm is slightly modified. Typically, the elitism algorithm converges within five iterations—it is more quickly than the traditional genetic algorithm, and even more quickly than the greedy algorithm.

Here we draw out the third finding:

Finding 3: The elitism algorithm converges more quickly than the other algorithms. It usually returns the optimal core-rule combination within five iterations.



(a) Performance on the scimark2 benchmarks.



(b) Performance on the ErsBlocks benchmark.

Fig. 9: Performance of four options-chosen strategies within fifty iterations. Here the random algorithm is with solid line, the greedy algorithm is with dashed line, the general genetic algorithm is with dash-dot line, and the elitism algorithm is with dotted line.

4) Answers to RQ4: We compared ORChooser against non-rule obfuscation and all-rule combination. The results are shown in Table II.

In this table, ORChooser can reduce the bytecode sizes of resulting files by 29%—37% compared with those obtained under non-rule obfuscation. However, for ProGuard (ver. 6.0.3), the bytecode sizes increase dramatically. ORChooser can reduce bytecode size by 37%—59%. That means ProGuard does more changes while obfuscating. ORChooser can decrease the bytecode sizes for almost all of the bytecode files, compared with those obtained using the all-rule combination. Besides, ORChooser can solve nearly 100% of conflicts.

Thus we draw out the fourth finding:

Finding 4: ORChooser's rule combinations are more effective than the all-rule combination in reducing code size and resolving conflicts.

VI. Related Work

This section discuss two strands of related work: code difference analysis and recommendation techniques.

a) Code Difference Analysis: Code difference analysis is an important research field in program analysis. Traditional code difference analysis techniques are text-based, token-based,

TABLE II: Bytecode sizes of the obfuscated SPECJVM benchmarks. Here each cell in gray indicates that rule conflicts exist.

	Bytecode Size (byte)							
	Android R8				ProGuard			
	Non	All	ORChooser	ORChooser (conflict resolved)	Non	All	ORChooser	ORChooser (conflict resolved)
scimark2lib	40812	29420	28804	28804	33898	21580	21356	24709
compress.jar	188340	115656	114168	120428	172335	83560	82762	84786
crypto_aes.jar	182288	116924	117092	118004	162804	78529	73878	80384
crypto_rsa.jar	181984	117036	115332	116180	192490	78523	78188	79233
crypto_ signverify.jar	182204	116912	115196	116124	162434	78286	77953	79043
serial.jar	197084	116232	114540	130000	184130	78206	77927	96753

abstract syntax tree (AST) based, program dependence graph (PDG) based and metric-based.

In recent years, new code difference analysis techniques emerge. Nimrod *etc.* compute programs semantic differences by abstract interpretation [27]. The most important part is to abstract relationships between variables in two programs. Yaniv David*etc.* propose an approach to decomposing a code into smaller comparable fragments [28]. Then similarity and difference of fragments can be located through customizing semantic definitions.

Besides, by combining traditional code difference analysis techniques, the code difference can be evaluated much more precisely. Saed *etc.* propose Semantic Integrated Graph (SIG) that integrates control flow graph, register flow graph, and function call graph, and then use the representations for identifying functions of binary code [29]. Execution Flow Graph (EFG) that integrates dependence graph and control flow graph has also been used for describing the semantics of binary code [30], which can also be employed for code difference analysis. In addition, the techniques that describe text information can be utilized. For BinShape, instruction-level characteristics are adopted [31]. Comparatively, ORChooser calculates obfuscation distances in three dimensions, allowing bytecode differences to be evaluated precisely.

b) Recommendation Techniques: Some recommendation techniques exist. Collaborative filtering methods are designed for collecting and analyzing a large amount of information on users' behaviors, activities or preferences and predicting what users will like based on their similarities to the other users [32]. A collaborative filtering method can be designed either on the memory [33], [34] or on some model [35]. However, to the best of our knowledge, collaborative filtering is not suitable for our purpose.

Content-based filtering is mainly based on a description of the item and a profile of the user's preferences [36], [37]. Many algorithms recommend items similar to the past ones. ORChooser is in fact a content-based filtering technique. It evaluates obfuscated file and maximizes the obfuscation distance iteratively.

VII. Conclusion

ORChooser is an adaptive approach to recommending obfuscation rules for Java bytecode obfuscators. It calculates obfuscation distance between the bytecode before and after obfuscation, and then employs an iterative process to maximizes the distance. The evaluation results show that ORChooser successfully recommends obfuscation rules for R8—it allows a small number of obfuscation rules to be chosen, while the obfuscated code becomes shrunk and obscure.

As for future work, we plan to extend ORChooser in three aspects. First, explore the influence of different α, β and γ. Second, the obfuscation distance between two bytecode files can be measured using many other features, such as word embeddings, traces and data flow. It remains our future work in selecting bytecode features for calculating distances, as different distance measurements may benefit ORChooser in recommending obfuscation rules. Third, the principle of ORChooser can be applied to many bytecode obfuscators. We would make ORChooser practical and compatible to other bytecode obfuscators.

Acknowledgments

We would like to thank the anonymous COMPSAC reviewers for valuable feedback on earlier drafts of this paper. Yuting Chen is the corresponding author. This research was sponsored in part by the National Key Research and Development Program of China (Project No. 2018YFB1003903), the National Nature Science Foundation of China (Grant No. 61572312, 61472242). Yuting Chen was also partially supported by Shanghai Municipal Commission of Economy and Informatization (No. 201701052).

References

[1] A. Akhunzada, M. Sookhak, N. B. Anuar, A. Gani, E. Ahmed, M. Shiraz, S. Furnell, A. Hayat, and M. K. Khan, "Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions," *J. Network and Computer Applications*, vol. 48, pp. 44–57, 2015.

[2] C. S. Collberg, J. W. Davidson, R. Giacobazzi, Y. X. Gu, A. Herzberg, and F. Wang, "Toward digital asset protection," *IEEE Intelligent Systems*, vol. 26, no. 6, pp. 8–13, 2011.

[3] C. S. Collberg, C. D. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, 1998, pp. 184–196.

[4] C. Wang and J. Knight, *A security architecture for survivability mechanisms*. University of Virginia, 2001.

[5] D. Low, "Java control flow obfuscation," Ph.D. dissertation, Citeseer, 1998.

[6] Google, "Android r8," https://r8.googlesource.com/r8, accessed 2018.

[7] GuardSquare, "The open source optimizer for java bytecode," https://www.guardsquare.com/en/products/proguard, accessed 2018.

[8] glurk, "Javaguard," https://sourceforge.net/projects/javaguard, accessed 2018.

[9] yWorks, “yguard-java bytecode obfuscator and shrinker,” https://www.yworks.com/products/yguard, accessed 2018.

[10] Y. Piao, J.-h. Jung, and J. H. Yi, “Structural and functional analyses of proguard obfuscation tool,” *The Journal of Korean Institute of Communications and Information Sciences*, vol. 38, no. 8, pp. 654–662, 2013.

[11] L. Sei, “R8, the new code shrinker from google, is available in android studio 3.3 beta,” *https://android-developers.googleblog.com/2018/11/r8-new-code-shrinker-from-google-is.html*, 2018.

[12] E. Lafortune, “Proguard manual,” *https://www.guardsquare.com/en/products5/proguard/manual/usage*, 2004.

[13] K. Dreßler and A. N. Ngomo, “On the efficient execution of bounded jaro-winkler distances,” *Semantic Web*, vol. 8, no. 2, pp. 185–196, 2017.

[14] Y. Li and B. Liu, “A normalized levenshtein distance metric,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 6, pp. 1091–1095, 2007.

[15] G. Rudolph, “Convergence analysis of canonical genetic algorithms,” *IEEE Trans. Neural Networks*, vol. 5, no. 1, pp. 96–101, 1994.

[16] J. Zhang, H. S. Chung, and W. Lo, “Clustering-based adaptive crossover and mutation probabilities for genetic algorithms,” *IEEE Trans. Evolutionary Computation*, vol. 11, no. 3, pp. 326–335, 2007.

[17] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, “A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii,” in *International Conference on Parallel Problem Solving From Nature*. Springer, 2000, pp. 849–858.

[18] B. Alll and C. Tumbleson, “Dex2jar: Tools to work with android. dex and java. class files,” *Octeau D, Jha S, McDaniel R Retargeting*.

[19] pxb1988, “Tools to work with android .dex and java .class files,” https://github.com/pxb1988/dex2jar, accessed 2018.

[20] E. Dupuy, “Jd-gui: Yet another fast java decompiler,” *URL http://java. decompiler. free. fr*, 2012.

[21] G. Campbell and P. P. Papapetrou, *SonarQube in action*. Manning Publications Co., 2013.

[22] R. Pozo and B. Miller, “Scimark 2 (2016),” *URL http://math. nist. gov/scimark2*, vol. 98, p. 118.

[23] Studently, “Ersblocksgame,” https://github.com/Studently/ErsBlocksGame, accessed 2018.

[24] P. Lengauer, V. Bitto, H. Mössenböck, and M. Weninger, “A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008,” in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L’Aquila, Italy, April 22-26, 2017*, 2017, pp. 3–14.

[25] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[26] E. Hillert, “Obfuscate java bytecode: an evaluation ofobfuscating transformations using jbco,” 2014.

[27] N. Partush and E. Yahav, “Abstract semantic differencing via speculative correlation,” in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, 2014, pp. 811–828.

[28] Y. David, N. Partush, and E. Yahav, “Statistical similarity of binaries,” pp. 266–280, 2016.

[29] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, “SIGMA: A semantic integrated graph matching approach for identifying reused functions in binary code,” *Digital Investigation*, vol. 12, no. Supplement-1, pp. S61–S71, 2015.

[30] J. Qiu, X. Su, and P. Ma, “Using reduced execution flow graph to identify library functions in binary code,” *IEEE Trans. Software Eng.*, vol. 42, no. 2, pp. 187–202, 2016.

[31] P. Shirani, L. Wang, and M. Debbabi, “Binshape: Scalable and robust binary library function identification using function shape,” in *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, 2017, pp. 301–324.

[32] J. S. Breese, D. Heckerman, and C. M. Kadie, “Empirical analysis of predictive algorithms for collaborative filtering,” in *UAI ’98: Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, University of Wisconsin Business School, Madison, Wisconsin, USA, July 24-26, 1998*, 1998, pp. 43–52.

[33] S. Ghazarian and M. A. Nematbakhsh, “Enhancing memory-based collaborative filtering for group recommender systems,” *Expert Syst. Appl.*, vol. 42, no. 7, pp. 3801–3812, 2015.

[34] F. Ortega, B. Zhu, J. Bobadilla, and A. Hernando, “CF4J: collaborative filtering for java,” *Knowl.-Based Syst.*, vol. 152, pp. 94–99, 2018.

[35] M. A. Ghazanfar, A. Prügel-Bennett, and S. Szedmák, “Kernel-mapping recommender system algorithms,” *Inf. Sci.*, vol. 208, pp. 81–104, 2012.

[36] C. C. Aggarwal, *Recommender Systems - The Textbook*. Springer, 2016.

[37] P. Brusilovsky, A. Kobsa, and W. Nejdl, Eds., *The Adaptive Web, Methods and Strategies of Web Personalization*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4321.