An Obfuscation Method to Build a Fake Call Flow Graph by Hooking Method Calls

Kazumasa Fukuda  
Division of Frontier Informatics,  
Graduate School of Kyoto Sangyo University.  
Motoyama, Kamigamo, Kita-ku, Kyoto-shi,  
Kyoto, Japan, 603-8555

Haruaki Tamada  
Faculty of Computer Science and Engineering,  
Kyoto Sangyo University.  
Motoyama, Kamigamo, Kita-ku, Kyoto-shi,  
Kyoto, Japan, 603-8555.

Abstract—This paper proposes an obfuscation method against illegal analysis. The proposed method tries to build a fake call flow graph from debugging tools. The call flow graph represents relations among methods, and helps understanding of a program. The fake call flow graph leads misunderstanding of the program. We focus on a hook mechanism of the method call for changing a callee. We conduct two experiments to evaluate the proposed method. First experiment simulates attacks by existing tools: Soot, jad, Procyon, and Krakatau. The Procyon only succeeded decompilation, the others crashed. Second experiment evaluates understandability of the obfuscated program by the hand. Only one subject in the nine subjects answered the correct value. The experiments shows the proposed method has good tolerance against existing tools, and high difficulty of understanding even if the target program is tiny and simple program.

Index Terms—Call Flow Graph, Obfuscation, Java 7, Hook Mechanism

I. INTRODUCTION

Today, many software cracking incidents were reported. Cracking incidents make heavy damages to software industries. For example, a DVD player was cracked and extracted decoding keys of DVD content[1]. The cracking invalidated the protection mechanism of DVD. A Blu-ray player was also cracked to extract decoding keys[2].

Many cracking methods are published and used by crackers. Then, crackers implement a cracking method as tools and distribute them. Tools are used by other crackers. Especially, crackers called script kiddies attack software with existing tools without special knowledge about cracking. It is important to prevent attacks by powerful crackers. In addition, it is also important to prevent attacks by script kiddies with existing tools.

The one of simple cracking methods is to get a call flow graph. IDA Pro[3], Soot[4], and etc. are the debugging tools whose features are to extract a call flow graph from given binary program. Therefore, a method to prevent getting a call flow graph contributes tolerances of a program against attacks.

Many software obfuscation methods are proposed to prevent such cracking. Software obfuscation method increases the cost of understanding the software. For example, changing names is the most famous obfuscation method[5]. Names in a program usually show the meaning of variables and methods, and expose intentions of the program. Therefore, it is simple but effective to change names to meaningless names.

In this paper, we propose a method to camouflage a callee. Adversaries try understanding the relations among methods in the early stage of the attack. Therefore, we entrap adversaries by building a fake call graph. For this, we focus on the hook mechanism of the method call. The key idea of the proposed method is to change a callee before runtime, then, the actual callee is called by the hook method at runtime. Besides, an actual callee and a fake callee are hopefully overloaded. Because, the actual callee analysis becomes more difficult.

II. PROPOSED METHOD

A. Program Obfuscation Method

We describe what is a program obfuscation method before describing the our method. A program obfuscation method transforms a program harder to understand. Collberg et al. give the definition of the program obfuscation method[6]. We re-statement the definition as follows.

Definition 1 (Program Obfuscation Method): Let $p$ be a given program, $I$ be an input set for $p$, and $r(p, I)$ be an output of $p$ with $I$. Let $X$ be a set of information in $p$, and $c(p, X)$ be a cost for extracting $X$ from $p$. Then, the obfuscation of $p$ with respect to $X$ is to transform $p$ into $p'$ with a certain method $f$ ($p' = f(p)$), such that

Condition 1: $r(p, I) = r(p', I)$, and
Condition 2: $c(p, X) < c(p', X)$.

Condition 1 means to keep input/output mapping before and after the obfuscation. This means that the obfuscation must preserve the external specification of a target program. Condition 2 means that understanding $p'$ is significantly more difficult than understanding $p$ for extracting $X$.

B. Key Idea

In this paper, we tackle to camouflage a callee by hooking a method call in order to build a fake call flow graph. In use of hook mechanism, we inject some routine into a program beforehand. Then, hook mechanism performs the routine before executing a target method at runtime.

We describe a runtime procedure of the proposed method. At first, we change a program to call a fake method $m_f$ instead of a target method $m_t$. Next, we install a hook into $m_f$ for calling a hook method $m_h$. Then, $m_h$ changes callee from $m_f$
the obfuscation method must preserve outputs. Therefore, we hook method to call a fake method \( m_f \). However, the program code is described to invoke a fake method \( m_f \) and a hook method \( m_h \) is called before invocation of \( m_f \). Then, \( m_h \) changes a callee from \( m_f \) to \( m_t \). Thereby, \( m_f \) is not invoked.

A relation between \( m_f \) and \( m_t \) is interesting in order to hide the actual call flow graph. We focus on method overloading mechanism, which is features of the object oriented programming languages. The method overloading mechanism is to decide an actual callee by method name and its argument types. Therefore, methods with same names can be defined multiply in a class, whose argument types are different.

C. Methodology

1) The transforming procedure of the proposed method: We summarize terminologies before describing the transforming procedure of the proposed method. Let \( p \) be a given program, \( p' \) be an obfuscated program, \( m_t \) be a target method, \( m_f \) be a fake method, \( m_h \) be a hook method, and \( m_r \) be a method call for \( m_t \). Let \( A_t \) be a set of argument types of method \( m_t \).

The followings are the transforming procedure of the proposed method.

Step 1. select overloaded two methods as \( m_t \) and \( m_f \).
Step 2. change instructions to push variables into the stack in order to change a callee from \( m_t \) to \( m_f \).
Step 3. add \( m_h \) to \( p' \).
Step 4. hook \( m_f \) for calling \( m_h \), and
Step 5. install routine for transforming arguments from \( A_f \) to \( A_t \) \((A_f \rightarrow A_t)\) into \( m_h \).

At first, a target method and a fake method are selected. Based on the key idea, two methods should be chosen preferably, so that the one method is overloaded from another. To invoke a method \( m_f \), the values of \( m_f \)'s arguments must be pushed on the stack before \( m_r \). In other words, changing the stack status turns to change the callee. Thus, in the step 2, we change the stack status from \( A_t \) to \( A_f \) in order to call a fake method \( m_f \). The change means the callee became not the right method \( m_t \), but the fake method \( m_f \). However, the obfuscation method must preserve outputs. Therefore, we install routine for recovering the callee into \( m_h \) in the step 5.

2) The relation between \( A_t \) and \( A_f \): The recovering routine is swapping, adding, and deleting arguments. Practically, the routine depends on the relation between \( A_t \) and \( A_f \). Following shows the relation between \( A_t \) and \( A_f \). Figure 2 also shows relation patterns between \( A_t \) and \( A_f \).

(a) \( A_f \) includes all of elements of \( A_t \) in the same order \((A_t \subset A_f)\).
(b) \( A_f \) includes all of elements of \( A_t \) in the same order \((A_t \subset A_f)\).
(c) All of elements of \( A_t \) and \( A_f \) are the same in the different order \((A_t = A_f)\).
(d) \( A_t \) and \( A_f \) do not have the same elements \((A_t \cap A_f = \emptyset)\).
(e) The part of elements of \( A_t \) and \( A_f \) are the same \((A_t \cap A_f \neq \emptyset \wedge A_t \cup A_f \neq A_t, A_f)\).

The recovering routine in \( m_h \) is implemented according to the relations.

III. IMPLEMENTATION

A. Overview of Implementation

This section describes how to implement the proposed method. The procedure of the proposed method consists of five steps described in Section II-C1. Followings are the transforming algorithm of the proposed method based on the transforming procedure shown in Section II-C.

1) select overloaded two methods as a target method and a fake method,
2) change callee by updating the stack status,
3) install a hooked method to the program,
4) hook the method call, and
5) build the recovering routine and installing it into the hook method.

B. Invokedynamic Instruction

Before describing each step, we introduce invokedynamic instruction[7]. Certainly, we realize the proposed method in use of reflection mechanism. For example, the Java language has \texttt{java.lang.reflect.Proxy} class to introduce hooking methods. However, obfuscator will be quite complex because of stack operations in recovering routine. Actually, more simple way exists in the Java 7 platform in use of invokedynamic instruction.

Invokedynamic instruction is the new instruction to execute a method. Originally, invokedynamic instruction is proposed for the JVM languages such as Jython[8], Scala[9], Groovy[10], and etc. In the JVM languages, method calls are inefficient since the calls must through the interpreter of the language. This indirect call is bottleneck of performance.

![Fig. 1. The key idea of the proposed method](image1)

![Fig. 2. Relation between \( A_t \) and \( A_f \)](image2)
Invokedynamic instruction achieves direct call, which is the solution of the problem in the JVM languages.

Invokedynamic instruction has acceptable additional effect. The effect is that invokedynamic instruction executes a special method called bootstrap method (BSM) before execution of a callee. That is, it is possible to hook a method with invokedynamic instruction.

Figure 3 shows basic behavior of the invokedynamic instruction in the sequence diagram of UML. At first, the JVM calls a corresponding BSM of invokedynamic instruction at executing the instruction from a caller object (Seq 1, and 2 calls a corresponding BSM of invokedynamic instruction at instruction in the sequence diagram of UML. At first, the JVM invokes BSM before execution of invokedynamic instruction. The effect is that invokedynamic instruction executes a special solution of the problem in the JVM languages. Invokedynamic instruction achieves direct call, which is the method called bootstrap method (BSM) before execution of invokedynamic instruction. Invokedynamic instruction has acceptable additional effect.

1) Selecting a target method and a fake method: In the first step, we select two methods as a target method \( m_t \) and a fake method \( m_f \). Hopefully, \( m_t \) should overload \( m_f \) from the key idea in order to obfuscate effectively.

2) Changing a callee: In this step, the proposed method changes arguments on the stack. The change routine depends on the relation between an actual callee and a fake callee, described in Section II-C2.

In the case of relations (a) and (b), the method inserts the push instructions to pile fake arguments on the stack. Because elements in the stack are missing to call \( m_f \) in the relation (a).

In relation (b), straightforward implementation drops surplus arguments to call \( m_f \). However, it turns quite tough situation to get the value again for calling the actual callee \( m_t \). Generally, getting a value is quite complex task, since the value is made from many values. In the relation (c), the method swaps the order of arguments since the order of arguments is different. In the relation (d), the method also inserts the push instructions. The reason of insertion is the same as the relation (b). In the relation (e), the method inserts the push instructions and swaps the order of arguments, since the relation (e) is the composite states of (a) through (d).

3) Installing the hooked method: This step installs the hooked method \( m_h \) to the program \( p \). Generally, the reflection mechanism is used for hooking a method call. However, in the proposed method, we use another approach with invokedynamic instruction for Java 7 platform.

We install the hooked method \( m_h \) to the target program \( p \). The signature (argument types and return type) of BSM is provided from the JVM specification[7]. The return type of BSM is \texttt{java.lang.invoke.CallSite} object. The arguments of BSM are the target method name, a signature of the target method, and argument values gives to the target method. General signature of BSM is shown in Fig. 4.

CallSite object holds target method as a \texttt{java.lang.invoke.MethodHandle} object. The first arguments of BSM, MethodHandle.Lookup object, is a factory class for MethodHandle object. The second arguments of BSM is \texttt{String} object, shows field or method name. The third arguments is MethodType object, shows a signature of the target method. Then, MethodHandle object shows the target method information as a function pointer, includes a method name, a class name of the method, argument types, and a return type.

The minimum routine of BSM is to build MethodHandle object from arguments and to return CallSite object related with MethodHandle object.

4) Hooking the method call: Next step, we hook the method call of \( m_f \) to call the hooked method \( m_h \). For this, we change old invoke instruction to invokedynamic instruction. Besides, we also relate the BSM and invokedynamic instruction for installing recovering routine.

5) Building the recovering routine: Final step builds the recovering routine and injects it into the hooked method \( m_h \). The resultant hooked method changes the fake callee to the actual callee.

The recovering routine depends on the relation between \( A_t \) and \( A_f \), alike in Section III-C2. In use of reflection mechanism, recovering routine operates the stack status. In

```
public static CallSite bootstrapMethod(Lookup lookup,
    String name, MethodType methodType){
    // body of the boot strap method.
} 
```
use of invokedynamic instruction, the useful class is provided. The class called `java.lang.invoke.MethodHandles` can drop, swap, and add arguments accessing the stack.

D. Example of the Proposed Method

For ease of understanding, we show an example of the proposed method. Figure 5 shows a sample program for printing an integer value from command line arguments.

```
1: public class IntegerPrinter {
2: public static void main(String[] args) {
3: Integer i = Integer.parseInt(args[0]);
4: System.out.println(i);
5: }
}
```

Fig. 5. Sample Program for the Proposed Method

```
1: public class IntegerPrinter {
2: public static void main(String[] args) {
3: Integer i = Integer.parseInt(args[0]);
4: return mcs;
5: mcs.setTarget(re);
6: mh, 1, int.class
7: MethodHandles.dropArguments(
8: java.lang.Integer.class, "parseInt",
9: MethodType.methodType(
10: int.class, String.class
11: )
12: );
13: MethodHandle re =
14: MethodHandles.dropArguments(
15: mh, i, int.class
16: );
17: mcs.setTarget(re);
18: return mcs;
19: }
20: public static void main(String[] args) {
21: Integer i = Integer.parseInt(args[0], 16);
22: System.out.println(i);
23: }
24: }
```

Fig. 6. Pseudo Code of Obfuscated Program

```
1: public class IntegerPrinter {
2: public static void main(String[] args) {
3: Integer i = Integer.parseInt(args[0]);
4: System.out.println(i);
5: }
6: }
```

Fig. 5. Sample Program for the Proposed Method

```
1: public class IntegerPrinter {
2: public static void main(String[] args) {
3: Integer i = Integer.parseInt(args[0]);
4: return mcs;
5: mcs.setTarget(re);
6: mh, 1, int.class
7: MethodHandles.dropArguments(
8: java.lang.Integer.class, "parseInt",
9: MethodType.methodType(
10: int.class, String.class
11: )
12: );
13: MethodHandle re =
14: MethodHandles.dropArguments(
15: mh, i, int.class
16: );
17: mcs.setTarget(re);
18: return mcs;
19: }
20: public static void main(String[] args) {
21: Integer i = Integer.parseInt(args[0], 16);
22: System.out.println(i);
23: }
24: }
```

Fig. 6. Pseudo Code of Obfuscated Program

```
1: public class Queue {
2: private List<String> list
3: public void enqueue(String value)
4: list.add(value);
5: return value;
6: }
7: public String dequeue() {
8: String value = list.get(list.size() - 1);
9: list.remove(list.size() - 1);
10: return value;
11: }
12: }
13: }
```

Fig. 7. Target queue program

```
1: public class Queue {
2: private List<String> list
3: public void enqueue(String value)
4: list.add(value);
5: return value;
6: }
7: public String dequeue() {
8: String value = list.get(list.size() - 1);
9: list.remove(list.size() - 1);
10: return value;
11: }
12: }
13: }
```

Fig. 8. Obfuscated program of Figure 7

is `{int.String}. The relation between $A_t$ and $A_f$ is the relation (a) shown in the Section II-C2. Therefore, we drop the fake argument `String` to call $m_t$ at line 13.

Resultant pseudo code is shown in Fig. 6. However, the proposed method is applied to class file. Therefore, the whole changes are not appear in the source code form.

IV. EVALUATION

A. Tool and Obfuscation Sample

We have developed a tool of the proposed method presented in Section III, called mican with ASM[11] for engineering class files. Mican generates obfuscated class files from original class files.

We conducted two experiments to evaluate the proposed method. The first experiment measures tolerance against existing attack tools. The second experiment simulates attack by the hand.

A program for the experiments is shown in Fig. 7, implements a queue structure. The target method is `add(int, Object)` method of `java.util.List` class at line 5 in Fig. 7. In the program, the new element for queue is added into first index in enqueue method, and dequeue method gets and returns the last element of `list` object.

We camouflaged the program to a stack structure. For this, a fake method is `add(Object)` method of `List` class. Figure
8 shows resultant program of the proposed method. The result is shown in source code form for easy to understand.

B. Experiment 1: Tolerance against existing attack tools

In the experiment, we exploit four tools to reverse-engineer the obfuscated program (Figure 8). The tools are Soot[4] for building a call flow graph tool, and jad[12], Krakatau[13], and Procyon[14] for decompilation.

Soot, jad, Krakatau were crashed and produced no results. Because, those tools do not support features of the Java 7 platform, especially invokedynamic. In the near future, we must take countermeasures, since those tools will support features of the Java 7. However, it is difficult task to build a call flow graph from the operand of invokedynamic instruction. Because overcoming the proposed method requires high level analysis of BSM’s body in order to find an actual callee.

Procyon only succeeded decompilation and generated source code. In the next experiment uses the result of this experiment.

C. Experiment 2: Analyzing obfuscated source code by the hand

The experiment is to find the result of the program shown in Fig. 9. The program is based on decompiled source code by Procyon in the experiment 1. We changed symbol names to meaningless names for eliminating understandable clues. Also, we add main method to print a value.

In this experiment, we had nine subjects, who are two master students and seven bachelor students (3rd grade and 4th grade students). Additionally, every subjects studied the Java language and the Java bytecode. We assigned the obfuscated source code to each subject and asked what the result is.

Results are one subject answered the correct value and the other subjects answered the wrong value. The correct subject and the one of the wrong subjects know the specification about invokedynamic instruction. Therefore, the proposed method causes wrong understanding even if the target program is simple and subjects know the specification about invokedynamic instruction.

V. Discussion

A. Stealthiness of the proposed method

In the early stage of the attack, adversaries try exposing what kind of the protection method is used. If the protection method is stealthy, the attack for the program become more hard task. The proposed method uses invokedynamic instruction and classes in java.lang.invoke package, general Java programs rarely used them. That is, the proposed method is easy to identify.

On the other hand, invokedynamic instruction is generally used in the JVM languages. Moreover, scripts written in the JVM languages can be compiled to class files. Therefore, if a program is written in multiple languages (e.g., the Java language, and the JVM languages), stealthiness will be increased. Recently, software written in multiple languages is frequently developed. The JVM language are usable for configuration and glue code for a software written in Java.

Besides, lambda expressions are newly supported in Java SE 8[15]. Lambda expressions represent one method interface in expression form and compiled lambda expression uses invokedynamic instruction. The bytecode of the expressions contains invokedynamic instructions. Therefore, software written in Java 8 will increase stealthiness of the proposed method.

B. Tolerance against dynamic analysis

The proposed method does not tolerate dynamic analysis. For example, AOP can store arbitrary logs when the method was called[16]. The actual call flow graph is built easily from the logs.

1: public class Data {
2:  private List<String> list;
3:  public Data() {
4:  super();
5:  this.list = new ArrayList<String>();
6:  }
7:  public static CallSite bsm(
8:    final MethodHandles.Lookup lookup,
9:    final String s, final MethodType mt)
10:   throws Throwable {
11:    final MethodType mt2 =
12:      mt.dropParameterTypes(0, 1);
13:    final MethodHandle virtual =
14:      MethodHandles.lookup().findVirtual(
15:        mt.parameterType(0), "add", mt2
16:      );
17:    return new ConstantCallSite(MethodHandles.
18:      insertArguments(virtual, 1, 0));
19:   }
20:   public void a(final String n) {
21:     // invokedynamic(add:
22:     // (Ljava/util/ArrayList;
23:     // Ljava/lang/Object;)V,
24:     // this.list, n)
25:   }
26:   public String b() {
27:     final String value =
28:       this.list.get(this.list.size() - 1);
29:       this.list.remove(this.list.size() - 1);
30:     return value;
31:   }
32:   public static void main(String[] args) {
33:     Data data = new Data();
34:     data.a("one");
35:     data.a("two");
36:     data.a("three");
37:     System.out.println(data.b());
38:   }
39: }

Fig. 9. The obfuscated source code from the result of decompilation developed.
However, the proposed method does not conflict with other obfuscation methods. Against dynamic analysis, overload induction reinforces the proposed method, which forcibly makes method overloading relation among unrelated methods[5]. Moreover, other obfuscation methods e.g., flattering control flows[17], and pseudo exchange of method execution order from random value[18], also strengthen the proposed method.

Furthermore, in this paper, the proposed method is implemented as static-transformation. Dynamic transformation will be acceptable for the proposed method by introducing self-modifying code in Java 7 platform[19]. It is rest of our future works how to apply the proposed method into the dynamic transformation.

VI. RELATED WORKS

The program obfuscation is classified according to what kind of information want to hide in the program. The kinds of categories are layout obfuscation, data obfuscation, and control flow obfuscation[20].

Layout obfuscations complicate the information in the program which does not need to its execution. The most famous layout obfuscation is to rename identifier to meaningless name[5]. Data obfuscations affect to data structure in the program by encoding, re-ordering, aggregating, and etc. String encryption is the one of data obfuscations, which is implemented in popular obfuscation tools, such as Zelix KlassMaster[21], and Dash-O[22]. Control flow obfuscations complicate control flow of the program. For example, opaque predicate inserts bogus conditions such as always true or false to boolean expression[23].

The proposed method does not conflict with all of obfuscation methods introduced in this section. Because, the proposed method transforms a invoke instruction to invokedynamic instruction, and the others do not change invokedynamic instruction.

VII. CONCLUSION

This paper proposed the obfuscation method to build a fake call flow graph. The method changes a callee before running a target program, and the program calls the right method by hooking a method call at runtime.

We developed the tool for the proposed method, and evaluated the proposed method. Three-fourth reverse-engineering tools were crashed and got no result. The one of reverse-engineering tool succeeded decompilation. The experiment to analyze the source code from decompilation was conducted. The one-ninth subjects only answered the correct value.

In our future work, we will conduct more large-scale experiment for human subjects. Also, we will examine how to increase the tolerance against certain attacks by combination with other protection methods.

ACKNOWLEDGMENT

Part of this work was supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B), 25730087.

REFERENCES