An Instruction Folding Method to Prevent Reverse Engineering in Java Platform

Tetsuya Ohdo
Division of Frontier Informatics,
Graduate School of Kyoto Sangyo University.
Motoyama, Kamigamo, Kita-ku, Kyoto-shi
Kyoto, Japan, 603-8555
Email: i1258177@cse.kyoto-su.ac.jp

Haruaki Tamada
Faculty of Computer Science and Engineering,
Kyoto Sangyo University.
Motoyama, Kamigamo, Kita-ku, Kyoto-shi
Kyoto, Japan, 603-8555,
Email: tamada@cc.kyoto-su.ac.jp

Yuichiro Kanzaki
Dept. of Human-Oriented Information Systems
Engineering, Kumamoto National College of Technology.
2659-2 Suya, Koshi-shi,
Kumamoto, Japan 861-1102

Akito Monden
Graduate School of Information Science,
Nara Institute of Science and Technology.
8916-5, Takayama-cho, Ikoma-shi,
Nara, Japan, 630-0192

Abstract—To improve tamper resistance of programs against illegal modification, this paper proposes instruction folding applicable to Java platform. In the proposed method, at first, similar methods are selected in a Java program. Next, these methods are merged into one method and diffs among these methods are stored in the program. Then, at runtime, when one of the merged methods is executed, diffs are restored by self-modification, which is realized by the Java instrumentation mechanism. The proposed method is resilient against tampering of folded method. Even if an adversary modifies the folded method, the program goes crash because the method is repeatedly modified at runtime.

Index Terms—Java; instruction folding; tamper-resistant software; bytecode; software protection

I. INTRODUCTION

Until now, various program obfuscation methods have been proposed to protect programs from illegal modification and analyses. The obfuscation transform a program so that it becomes more difficult to analyze, yet is functionally equivalent to the original one. For example, the name obfuscation replaces names in a program into meaningless ones[2],[3]. The opaque predicate method adds dummy branches to disturb control flow analyses [1]. Also, there are many obfuscation tools available for Java bytecode programs because they are commonly used and are susceptible to decompilation attacks. However, while conventional obfuscation methods focus on adding resilience against static analyses, there are few proposals to disrupt dynamic analyses.

This paper focuses on the dynamic diversity method, which translates a program during its execution to make both static and dynamic analyses difficult. Previous studies proposed methods to protect programs in PE format (Windows platform), ELF format (UNIX platform), and assembly language [4],[5]. These techniques are expected to protect Java programs as well, but we need to somehow introduce self modification mechanism in Java programs to realize dynamic program translation. Therefore, our previous work exploited the Java instrumentation mechanism to realize the self modification in Java platform [6].

In this paper, we introduce to Java platform the instruction folding method proposed by Nishioka et al.[7], which is a powerful dynamic diversity method. The instruction folding merges similar sequences into one sequence and diffs among these sequences are restored during execution. While the original method was proposed for PE format programs in Windows platform[7], this paper introduces the basic idea of this method to Java platform.

The rest of this paper is organized as follows. Section II describes related works. Section III introduces the concept of instruction folding method. Section IV shows implementation and the case study of the proposed method. Section V discusses the tolerance against certain attacks. Finally section VI summarizes the paper and describes future works.

II. RELATED WORKS

A. Analysis Techniques

Tamper-resistant software is to prevent illegal modification from adversaries by supplying some protection mechanisms to software. The one of the goals of these mechanisms is to increase the complexity of software since adversaries need to understand software for meaningful modification.

Analysis technologies are generally categorized into static or dynamic manner. The static analysis analyzes the program structures and/or layouts without running a program. The major static analysis techniques are disassembling and decompiling. To disrupt these techniques, program obfuscation techniques and program encryption techniques were proposed[8]. The program obfuscation technique is to convert a program to more complex and incomprehensible one[1]. The program encryption technique is to encrypt the program beforehand, then it is decrypted and executed at runtime[9].
The dynamic analysis analyzes dynamic behaviors of a program, such as memory status, method call traces, and debug information. The typical techniques of dynamic analyses include memory dump, running a debugger, etc. To prevent these techniques, methods for introducing dynamic diversity to a program were proposed[4], [5].

B. Dynamic Diversity Techniques

The use of self-modification technique is the conventional method of the dynamic diversity. The self-modification technique changes the instruction while execution of the program. Kanzaki et al. proposed the instruction camouflage method for hiding instructions in a program[5]. Kanzaki’s method overwrite instructions with dummy instructions beforehand. Then, the dummy instructions are rewritten back to the original instructions at runtime. After the execution of the original instruction, it is changed back to the dummy instruction again. Another technique using self-modification is dynamic code mutation[10]. This technique changes code slightly in every execution. Therefore, the execution trace extracted from the program is different in each execution.

The common problem of these techniques is that they introduce extra code fragments in the program. Examples of extra fragments are dummy instructions and dummy methods, which are not executed actually. If these extra fragments insertion is performed many times, stealthiness of the protection methods will decrease. Hence, a protection method without extra fragments is required. Note that, generally, we should apply more than two protection methods together to protect a program against tampering.

Dynamic diversity for Java platform was not discussed so far, because conventional Java platform did not support the modifying of bytecode at runtime.

C. Protection Methods in Java Platform

In Java platform, there are many methods to analyze bytecode. The most popular method is decompilation. The rigorous specification of the Java Virtual Machine[11] made it easy to develop powerful decompilers, such as jad[12].

Meanwhile, the dynamic analysis is also commonly used to comprehend bytecode in the Java platform. Because many powerful and usable methods are proposed, such as AOP (Aspect Oriented Programming)[13], and Amida[14].

In AOP, we apply an event such as a method call and calculation as a join-point. Then, given code (advice) is weaved at any timing at runtime. From the point of view of attackers, an important method is selected as a join-point. Next, displaying code of object statuses is weaved. Then running the program exposes secret information.

Amida is a tool to record an execution trace and visualize the trace as a sequence diagram[14]. This tool is used to extract a profile of a program and to visualize program behaviors.

To prevent attacks by above techniques, there is an approach to obfuscate interpreter[15]. This method introduce a finite state machine which gives context-dependent semantics and operand syntax to the encoded instructions in program. Thus, attempts of static analysis of the relation between instructions and their semantics will be failed. Unfortunately, this approach needs the new interpreter installation, and degrades the portability.

From above discussion, this paper proposes a method to prevent dynamic analysis in the modern Java platform. The method is called instruction folding, which introduces a self-modification mechanism to Java bytecode and merges several instruction sequences into one sequences.

III. INSTRUCTION FOLDING

A. Key Idea

Originally, Nishioka et al. proposed the instruction folding method[7] for assembly programs. On the other hand, our target is the Java platform since it is susceptible to many types of attacks as we described. However, Java platform does not support modifying of loaded bytecode[8], [11]. This problem exists, actually, only in conventional Java platform. Modern Java platform (Java 5 platform or later) supports modifying loaded bytecode mechanism at runtime, called the instrumentation mechanism. Our previous work confirmed that self-modification is feasible in modern Java platform[6]. Hence, we can apply an instruction folding method in Java platform from those two works.

Here, we revisit the instruction folding method proposed by Nishioka et al. Figure 1 shows an outline of an instruction folding method. Here we assume in Fig. 1 the original program (A) before folding has three similar instruction sequences a1, a2 and a3. In Fig. 1, after applying the instruction folding, sequences a1, a2 and a3 are folded into vc in the resultant program (B). Differences (diffs) between vc and a1, a2 and a3 are extracted, and they are used to recover original sequences a1, a2 and a3 at runtime by using self-modifying mechanism.

This method has tolerance against illegal modification. If adversaries try to modify vc, the modification leads to crash the restoring of original instruction sequences a1, a2 and a3. It is difficult to remove the protection mechanism because it is unaware to the adversary how many instruction sequences are folded (merged) in vc.

(A) Original program

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction sequence a1</td>
<td>!</td>
</tr>
<tr>
<td>Instruction sequence a2</td>
<td>!</td>
</tr>
<tr>
<td>Instruction sequence a3</td>
<td>!</td>
</tr>
</tbody>
</table>

(B) Folded program

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Restoring sequence r1 (vc to a1)</td>
<td>!</td>
</tr>
<tr>
<td>Common sequence vc of a1, a2, and a3</td>
<td>!</td>
</tr>
<tr>
<td>Restoring sequence r2 (vc to a1)</td>
<td>!</td>
</tr>
<tr>
<td>Restoring sequence r3 (vc to a1)</td>
<td>!</td>
</tr>
</tbody>
</table>

Fig. 1 Outline of an instruction folding method
B. Folding Similar Instructions

Let \( p \) be a given program and \( V_t = \{a_1, a_2, \ldots, a_m\} \) be a list of similar instruction sequences in \( p \). Let \( d(a, b) \) be a function to obtain differences between instruction sequence \( a \) and \( b \). Then, let \( v_c \) be a folded sequence from the elements of \( V_t \). Also, let \( r_i \) be a restoring sequence from \( v_c \) to \( a_i \) \((r_i : v_c \rightarrow a_i)\), and \( V_r \) be a list of restoring sequences, denotes by \( V_r = \{r_1, r_2, \ldots, r_m\} \). By executing \( r_i \), \( v_c \) is changed into \( a_i \) \((0 \leq i \leq m)\).

The instrumentation mechanism of the Java platform can modify loaded bytecode at runtime. Therefore, it enables self modification in the Java platform. However, the mechanism has several limitation. The one of the limitations is that it cannot change bytecode in a currently executed method. Therefore, we identify \( v_c \) as a method level. As a result, the instrumentation mechanism can modify \( v_c \) at runtime.

C. Instrumentation Mechanism

Characteristics of Instrumentation class are as follows.

1) redefined classes are not initialized,
2) instances of the redefined class are not affected, and
3) the active method is not affected.

The first characteristic is to preserve the value of variables before/after redefinition. Also, constructor of redefined classes is not executed, because of this characteristic. The second characteristic is that redefinition does not add, remove, or rename fields or methods, change signature of methods, or change inheritance. However, method bodies, constant pool and attributes can be changed. The third characteristic indicates that although a method being executed is redefined at runtime, the method continues running with original bytecode. The new bytecode is used in the new invocation.

IV. IMPLEMENTATION AND PERFORMANCE EVALUATION

This section describes how we implemented a prototype implementation of the proposed method. We also describe the evaluation of the performance of the proposed implementation.

The platform in this section is Java SE 6.37, OS X 10.8.2 Mountain Lion, 8GB RAM, Intel Core i7 3667U 3.20GHz, MacBook Air.

A. Prototype Implementation

In use of instrumentation mechanism, we must employ Java agent mechanism. A Java agent can change target programs at runtime or before execution without changing source code of the target programs. The Java agent has the premain method executed before the main method. We can build Java agent in the Java language and the C/C++ languages. In this work, we implemented a prototype tool in the Java language with ASM[16], named TATAMI.

Figure 2 shows a target program finding \( n \)th-root of a given value by the Newton’s method. We define \( f(x) = x^n - k \) with given values \( n \) and \( k \). We derive \( f'(x) = nx^{n-1} \) of derivative function of \( f(x) \). By Newton’s method, we use the recurrence relation

\[
x_{m+1} = x_m - \frac{f(x_m)}{f'(x_m)}.
\]

We have straightforwardly implemented Newton’s method as Figure 2 (omit error handling process). This program accepts \( n \) and \( k \) values as inputs and gives \( \sqrt[k]{n} \) as an output. In this program, we define \( f(x) \) method and \( dfx \) method from \( f(x) \) and \( f'(x) \), respectively. Now, we folds the two methods \( f(x) \) and \( dfx \) in the target program.

Figure 3 shows bytecode of \( f(x) \) and \( dfx \), given by compilation. The bold fonts are different parts, and the normal fonts are common sequence \( v_c \). Then, the resultant program is shown in Figure 4. The source code representation is just for understanding (actually, source code is not required.) Difference between Figure 2 and Figure 4 is that \( dfx \) method is defined or not.

In Figure 4, the program calls the method update of the class InstructionFoldingMethod, instead of \( dfx \) method definition. The class InstructionFoldingMethod is an utility class for instruction folding. By calling update method in line 7 and 9 of Figure 4, method \( f(x) \) is changed into method \( dfx \) and vice versa.
Table I

Result of Performance Evaluation

<table>
<thead>
<tr>
<th>Ordinary</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32.1</td>
</tr>
<tr>
<td>Our method</td>
<td>5,268.2</td>
</tr>
</tbody>
</table>

In Figure 5 shows the sequence diagram of Figure 4 behavior. In Figure 5, dashed oval box shows loop structure. The result of the first fx method and the second fx method are different. Because update of InstructionFoldingMethod changes the behavior of fx method into dfx of Figure 2.

B. Inside TATAMI

The TATAMI is implemented as InstructionFoldingMethod class for instruction folding. The procedure of update method of InstructionFoldingMethod class is as follows.

1) get bytecode of the class by ClassLoader class,
2) analyze the class by ASM,
3) update fx method,
4) get new bytecode of FoldedNthRoot by ASM, and
5) redefine FoldedNthRoot by Instrumentation class.

Currently, self modification points are pre-defined in TATAMI. Therefore to use TATAMI for other programs, we need to rewrite TATAMI. In future, we will improve the implementation of TATAMI to increase its usability.

C. Performance Evaluation

We conducted an experiment for evaluating performance of our implementation. In this experiment, we compute the value of \( \sqrt[3]{7} \), with and without our method 10 times. The result of average performance is shown in Table I. From the result, we found that execution time became 160 times larger in our implementation. Since the self-modification was executed four times, one self-modification requires about 1,300 µs.

```
0: dconst_1 0: dconst_1
1: dstore 5 1: dstore 5
2: iconst_0 3: iconst_1
4: istore 7 4: istore 7
6: iload 7 6: iload 7
8: iload_1 8: iload_1
9: if_icmpge 24 9: if_icmpge 24
12: dload 5 12: dload 5
14: dload_3 14: dload_3
15: dmul 15: dmul
16: dstore 5 16: dstore 5
18: iinc 7,1 18: iinc 7,1
21: goto 6 21: goto 6
24: dload 5 24: dload 5
26: iload_2 26: iload_1
27: i2d 27: i2d
28: dsub 28: dmul
29: dstore 5 29: dstore 5
31: dload 5 31: dload 5
33: dreturn 33: dreturn

(a) bytecode of fx
(b) bytecode of dfx
```

Fig. 3. Difference between similar methods

Moreover, we evaluated the delay rate per self-modification counts (the number of executed self-modification). To evaluate the delay rate, we choose input parameters \( n \) and \( k \) for which self-modification counts are from 0 to 20. The relationship between the self-modification count and input parameters is shown in Table II. In the evaluation, we performed TATAMI 10 times in every input and measured the average execution time.

```
1: public class FoldedNthRoot{
2: private static final double DELTA = 0.0001;
3: public double calculate(int n, int k){
4: double startX = 2d;
5: double x = startX;
6: while(Math.abs(fx(n, k, x)) > DELTA){
7: InstructionFoldingMethod.update();
8: double dfx = fx(n, k, x);
9: InstructionFoldingMethod.update();
10: x = x - (fx(n, k, x) / dfx);
11: }
12: return x;
13: }
14: public double fx(int n, int k, double x){
15: double answer = 1;
16: for(int i = 0; i < n; i++){
17: answer = answer * x;
18: }
19: answer = answer - k;
20: return answer;
21: }
22: public static void main(String[] args){
23: int n = Integer.parseInt(args[0]);
24: int k = Integer.parseInt(args[1]);
25: FoldedNthRoot root = new FoldedNthRoot();
26: System.out.printf("sqrt_%d(%d)=%g\n",
27: n, k, root.calculate(n, k))
29: }
30: }
```

Fig. 4. The result of folded program

```
:InstructionFoldingMethod
:FoldedNthRoot

Invoke premain

Invoke main

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main

Invoke premain

Invoke calculate

Loop

Invoke update

Invoke fx

Invoke update

Invoke fx

Invoke fx

Invoke update

Invoke update

Invoke fx

Invoke main
```

Fig. 5. The sequence diagram of figure 4
The result is shown in Figure 6. The horizontal axis represents self-modification count, and the vertical axis plots the execution time in µs. From Figure 6, we found that execution time is delayed about 500 µs per a self-modification is executed. This delay is quite heavy for real-time software. Therefore this method should be applied to a method that is not frequently executed. Also, this method is suitable to non-tight response-time systems, such as GUI applications.

V. DISCUSSION

The proposed method is resilient against tampering of folded method. Even if an adversary identifies the folded method and modifies it, the program goes crash. Because the method is repeatedly modified at runtime.

One concern is that the updated methods could be observed at runtime by analyzing the RAM at runtime (although it may not be easy). One solution is to identify similar sequences that have different execution frequencies. If one of the sequences has very low execution frequency, we expect that it is difficult for an adversary to identify all folded sequences.

Another concern is that the proposed method uses the instrumentation mechanism of the Java platform to introduce self-modification; and, this indicates that the program applied our method contains very special code that uses Instrumentation class. Thus, the instrumentation code potentially becomes a clue of reverse engineering, and an adversary possibly identifies the folded sequence. We should apply other obfuscation methods for hiding self-modification code to improve robustness against static analyses.

Currently, there is no specific way to select sequences (methods) to be folded. For an extreme example, two dissimilar sequences \( a \) and \( b \) like \( a \cap b = \emptyset \) can be folded. Further research is needed to develop a guideline to select sequences.

VI. CONCLUSION

This paper proposed a method to employ instruction folding in the modern Java platform to improve tamper resistance. By using the Java instrumentation mechanism, we could introduce self-modification to realize the instruction folding in Java programs. Then, we implemented the prototype tool of the proposed method and evaluated the method. The result showed that execution time became 160 times larger in our implementation; thus, we should use the proposed method in not time-sensitive programs ore functions.

Finally, we describe our future works. We will conduct major-scale experiments to evaluate the delay mechanism and to reduce the delay. In the current implementation, folding procedure and constructing a restoring-routine are done by human. Automation of these processes is also our important future work.

REFERENCES


