Program Obfuscation Scheme Using Random Numbers to Complicate Control Flow

Tatsuya Toyofuku[1], Toshihiro Tabata[2], and Kouichi Sakurai[3]

[1] Graduate School of Information Science and Electrical Engineering, Kyushu University, 6-10-1 Hakozaki, Higashi-ku, Fukuoka, Japan 812-8581
toyofuku@itslab.csce.kyushu-u.ac.jp
[2] Graduate School of Natural Science and Technology, Okayama University, 3-1-1 Tsushima-naka, Okayama, Japan 700-8530
tabata@it.okayama-u.ac.jp
[3] Faculty of Information Science and Electrical Engineering, Kyushu University, 6-10-1 Hakozaki, Higashi-ku, Fukuoka, Japan 812-8581
sakurai@csce.kyushu-u.ac.jp

Abstract. For the security technology that has been achieved with software in the computer system and the protection of the intellectual property right of software, software protection technology is necessary. One of those techniques is called obfuscation, which converts program to make analysis difficult while preserving its function. In this paper, we examine the applicability of our program obfuscation scheme to complicate control flow and study the tolerance against program analysis.

1 Introduction

Recently, Java, the object oriented programming language has been rapidly widespread. Java is executable in different hardware, OS, and furthermore small information terminals such as cellular phones and PDA. Described ahead, Java has a big feature of portability that it is executable on many platforms.

Java has a serious problem, however. Java program is distributed in the style called class file which is executed on a virtual machine. There is a technique called decompile that converts binary code into source code. As for the Java class file, we can easily get program code which is close to original source code. Analyzing decompiled source code, an attacker can steal algorithm used in the program code. Java has another big feature. Class file created in a certain program can be reused in the part of another program. Abusing this feature, the attacker is able to steal class file, make new program using that file, and insist on the property right of the program.

To solve these problems, software protection technique is necessary. One of those techniques is called software obfuscation. Obfuscation is a technique that converts program into another program which is difficult to analyze while preserving its function.

In this paper, we propose obfuscation scheme using random numbers to complicate control flow. We introduce how to obfuscate program control flow and study the tolerance against program analysis.

2 Related Works

Many obfuscation schemes have been proposed. The easiest scheme that an automatic application to the program (we call this auto-application) is called name conversion. This is a scheme of concealing what value each variable maintain and what kind of operation each function does by changing variable identifiers and function names into a quite meaningless character string. Monden et al. proposed scheme obfuscating program includes loop [1]. Ogiso et al. proved that pointers address decision problem is NP-Hard, and proposed scheme to complicate function calling by using function pointer. This scheme has theoretical proof of safety against program analysis [2]. These schemes are for obfuscating C program.

For Java program, Fukushima et al. introduced scheme to make analysis difficult by destroying the encapsulation by distributing methods. This is the scheme to destroy encapsulation which is one of the features of object oriented program and we can erase class information by this scheme [3]. This scheme is applicable to any object oriented program. Another scheme is to conceal relation between variables by linear transformation [4]. Some schemes obfuscate program by complicating control flow. For example, paper [5] proposed scheme to make analysis hard by inserting if-sentence which always returns true (or false).

To apply obfuscation scheme into huge amount of program, auto-application is required, but some scheme is difficult to do this. It is necessary to find the part where we can change program execution order while preserving original program's functionality to apply scheme introduced in paper [1]. But automation of this judging process is difficult. About scheme in paper [2], application itself is difficult because some programming language does not have a pointer.

In this paper, we propose obfuscation scheme complicating control flow. Our scheme is applicable to program written in object oriented language (we call this object oriented program), and auto-application is possible. We propose obfuscation scheme by complicating control flow and we study about execution efficiency and tolerance to the attack.

3 Proposed Scheme

3.1 Complicating Control Flow by Random Numbers

We explain our scheme using Java program. The purpose of this scheme is to complicate control flow in `main` function which exists in object oriented program, and make analysis of program's entire execution difficult.

We consider obfuscating program in figure 1. We can know an execution order of each method by analyzing `main` function part. Our scheme complicates this part and make program's entire execution flow analysis difficult. In the process of complicating control flow, we use random numbers which are difficult to predict by static analysis which is a technique to analyze program by only seeing source code. We explain an algorithm to achieve this in the following.

```
public static void main(){
(A) (B)
    method1();    method2();
    method3();    method4();}
static void method1(){
(C)
definition of method1}
//define the other methods
```

Fig. 1. Basic program

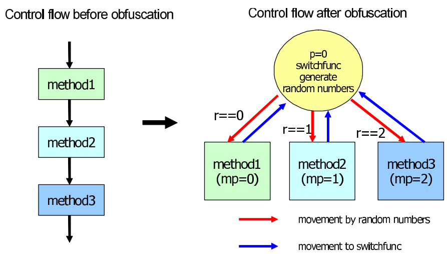

Fig. 2. Outline of method point algorithm

3.2 Method Point Algorithm

This algorithm consists of three steps. We introduce a detail of each step. Figure 2 shows an outline of this algorithm.

1. Setting point variable and method points
2. Generating a random number
3. Comparison of `p` and `mp`

1. Setting point variable and method points
Add point variable `p` (initial value is 0) in (A) , and give a method point `mp` to each method. `Mp1` stands for `mp` of method1.

Example
`mp1`=0, `mp2`=1, `mp3`=2, `mp4`=3

2. Generating a random number
Generate a random number `r` in (B). Decide to which method to move by `r`.

Example
Move to `method1()` if `r` is 0, `method2()` if `r` is 1, `method3()` if `r` is 2, and `method4()` if `r` is 3.

3. Comparison of `p` and `mp`
In the method moved in step 2, compare `p` and `mp` at (C). If both values match, execute that method, increase `p`, and do step 2 again. If those do not match, return to step2 without executing that method.

Repeat step 2 and 3 until `p` becomes 4 (the number of methods).

Explanation of sample behavior
There are some patterns of program behavior. We will explain them respectively.

Case 1: When `p` matches `mp` and `p` does not become 4 after executing method (Ex. When `p`=0 and `r`=0)
Since `mp`=0 and `p`=`mp`, execute `method1` and `p` becomes 1. Then, regenerate random number `r` since `p` is not 4.

```
public static void main(String[] args){
        int p = 0;//initialize point variable
        switchfunc(p);}
static void method1(add variable p){
        if(p==0){//if p matches mp, execute method
        p++;//increase p}
        switchfunc(p);}
//add variable p to the other methods similarly
static void switchfunc(int p){
    if (p<4){//generate random number if program is not finished
   int r =(int)(Math.random()*4);
    switch(r){ //move to method allocated by r
    case 0:        method1(); break;    case 1:        method2(); break;
    case 2:        method3(); break;    default:       method4(); break;}}}
```

Fig. 3. Outline of obfuscated program

Case 2: When `p` does not match `mp` (Ex. When `p`=1 and `r`=2)
Since `mp`=2 and `p`≠`mp`, `method3` will not be executed. Regenerate random number `r`.
Case 3: When `p` matches `mp` and `p` becomes 4 after executing method (Ex. When `p`=3 and `r`=3)
Since `mp`=3 and `p`=`mp`, execute `method4` and `p` will become 4. End program because `p` is now 4 and this means every method is executed.

By this algorithm, we can obtain obfuscated program which has original program's function and complicate control flow. Figure 3 is an outline of program applying our scheme to program in figure1.

4 Expanding Proposed Scheme

We examined obfuscating program control flow by method point algorithms in the case that a program does not have such a complicate structure as branch or loop in `main` function. We call this structure simple control flow. In this section, we expand our scheme to make application possible to such a complicate control flow explained above. We consider applying our scheme to control flow in figure 4 and 5.

4.1 Obfuscating Branch Program

Consider the case of program control flow in figure 4. Program in figure 6 shows the sample program which has control flow in figure 4. To apply our scheme, we convert complicate program enclosed with frame into simple control flow. We can achieve this conversion of program which have branch by executing steps below.

1. Embedding branch condition
2. Allocating method point
3. Setting `switchfunc`

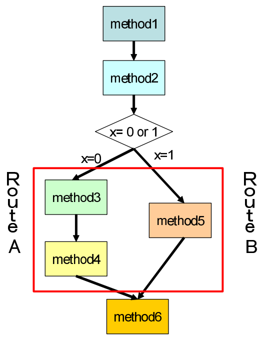

Fig. 4. Example of program control flow having branch

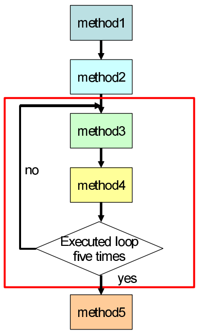

Fig. 5. Example of program control flow having loop

1. Embedding branch condition
Before framed part in figure 4 exists branch condition to decide whether go to route A or B. Embed this condition into method located in just before the condition (in this case, embed into `method2`). We also embed executing condition of each route into framed methods.

Example of Embedding Condition
`int x = (int)(Math.random()*2);`(I)
`if (x==0){route A} else {route B}`(II)

Suppose branch condition is the one written above. This means if random number `x` is 0, then execute route A, if `x` is 1, then route B. Embed condition (I) into `method2` and (II) into `method3`, `method4` (methods executed in route A) and `method5` (method executed in route B).

2. Allocating method point
Allocate method point in executing order just like in the scheme explained in section 3.2 until reach branch point.

From the part where the branch starts to where branched execution routes join, method points are allocated from the continuation of the `mp` allocated in the method just before the branch point respectively.

In method executed after joining, choose the biggest `mp` among points allocated in the method just before joining, and allocate from the continuation of that value. Thus, each method's `mp` becomes the value written below.

Example of allocating `mp`
`mp1`=0, `mp2`=1, `mp3`=2, `mp4`=3, `mp5`=2, `mp6`=4

3. Setting `switchfunc`
Finally, decide which method to execute by `r` and introduce `p`.

We have one point to consider. The value in `p` after executing each route differs. In this case, `p` after executing route A is 4, while after route B is 3. Method

```
public static void main(){
    method1();    method2();
int x=(int)(Math.random()*2);
  //x=0 or 1
if(x==0){
    method3();    method4();}
else {method5();}}
  method6(); }
static void method1(){
  definition of method1}
//define the other methods
```

Fig. 6. Sample program having branch

```
static void method2(int p, int x){
  if(p==1){execution of method2
    p++;} //increase p
  (*)x=(int)(Math.random()*2);
  switchfunc(p,x);}
static void method3(int p, int x){
  (*)if(x==0){
    if(p==2){ //compare p and mp
    execution of method3
    p++;} //increase p
  switchfunc(p,x);}
static void switchfunc(int p, int x){
    define switchfunc as usual
}
```

Fig. 7. Outline of obfuscated program having branch

point `mp` in method executed next is 4, but after executing route B, method whose `mp` is 3 will be executed. To avoid this case, when route B is executed, adjust value added to `p` to become the next method's `mp` in the last method. If route B is executed in figure 4, `p`=2 before `method5`, so add 2 after executing method. As we see in this example, if multiple routes meet after branched, we need to adjust a value added to `p` in the method whose `mp` is smaller than other one to make `p` next method's `mp` in method just before joining of each route.

By these steps, we can apply our scheme to program which has branch structure. Figure 7 shows an outline of obfuscated program. Operation (*) is an embedded program.

4.2 Obfuscating Loop Program

In this section, we consider applying our scheme to a program such have a loop repetition structure like in the part enclosed with the frame in figure 5. In this control flow, repeat `method3()`, `method4()` 5 times after executing `method1()` and `method2()`. Then, execute `method5()`. Figure 8 shows the sample program.

We can apply our scheme to control flow like figure 5 by following steps.

1. Allocating method point
2. Embedding loop finishing condition

1. Allocating method point
First, allocate method point to each method as usual scheme without considering loop.

Example
`mp1`=0, `mp2`=1, `mp3`=2, `mp4`=3, `mp5`=4

```
public static void main(){
    method1();    method2();
for(int loop=0; loop<5; loop++){
    method3();    method4();}
    method5();}}
static void method1(){
    definition of method1}
//define the other methods
```

Fig. 8. Sample program having loop

```
static void method4(int p, int loop){
  if(p==3){
    (**)if(loop<4){
    (**)p--;
    (**)loop++;}
else{p=p+1;}//end loop}
  switchfunc(p,loop);}
static void switchfunc(int p, int loop){
  if(p<5){int r =(int)(Math.random()*5);
    define switchfunc as usual }}
```

Fig. 9. Outline of obfuscated program having loop

2. Embedding loop finishing condition
Embed loop finishing condition into method executed in the last of the loop (`method4` in the case of figure 5). Explain this by loop written in `for` sentence. The `for` sentence is written in a style of (initial state; finishing condition; continuance processing). Thus, in figure 5, loop condition is `for(int loop=0; loop<5; loop++)`. This means loop will be repeated 5 times. In `method4`, compare point variable `p` and method point `mp`. If both values match, to judge whether loop is over or not after executing method, embed finishing condition and continuance processing as follows.

```
if(loop<4/*finishing condition*/){p--;
loop++;/*continuance processing*/} else {p++;}
```

At `if` sentence, whether to continue loop or not will be judged. If continuing loop is necessary, `method3` must be executed again. Executing `method3` is impossible, however, in the time when `method4` is executed. Because at that moment, `p` is 3 and it does not match `method3`'s method point `mp3`(=2). Therefore, we introduce new operation for `p`. If a loop must be repeated, decrease `p`. A value to decrease is equal to the number of methods in loop structure before method where the finishing condition is embedded. In this case, there are 2 methods repeated and only 1 method before `method4` where finishing condition is embedded, so we subtract 1 from `p`. Thus, embedded condition is written before. Figure 9 is an outline of obfuscated program. Operation (**) is an embedded program.

5 Evaluating Proposed Scheme

5.1 Attacking Program

Attack on program is divided mainly into 2 types: static analysis which analyze program only by seeing source code, and dynamic analysis by executing program. First, we examine the tolerance against static analysis. Our scheme has a feature

```
a=6, b=3
c=9
d=3
e=18
f=2
```

Fig. 10. Output result before dynamic analysis

```
static void sub(){
//method operates subtraction
  if(p==2){
(***)System.out.println("sub");
    execution of sub
    p++}
  switchfunc(p);}
```

Fig. 11. Sample of dynamic analysis

that generates random number and decides method executing next. To analyze an obfuscated program, an attacker must judge whether the called method is executed or not. He tries to find out method executing order by analyzing `p` and `mp`. Suppose analyzing `p` and `mp` is hard. In this case he takes a strategy of arranging methods suitably and analyzing the execution order. If there are N methods in the program, there are N! probable execution order. Thus, the more the method numbers are, the more the probable control flow increases, cost to static analysis grows extremely high.

Next, we consider the dynamic analysis using program computes the four basic operations of arithmetic. Figure 10 is an result of execution. For example, in figure 11, insert a program (***) which outputs method name when `p` matches `mp` and that method is executed. In this case, the string ”sub” is displayed when executing method `sub`. Insert this program in every method changing output name. By this attack, method’s name is displayed like figure 12 when each method is executed, and an attacker can know method execution order.

A method in figure 13 is considered as a countermeasure against dynamic analysis. When applying scheme, insert dummy method which has no influence on program execution result. There is no limitation in the number of dummy methods, and it can be executed many times. Example of dummy method is given in figure 13. Dummy method in figure 13 executes the follwing. If the condition is true, call `method4`, and if it is false, operate complicate operation for `p`. But this condition always returns false and no method is called. And complicate operation actually returns `p` itself. Moreover, the frequency of dummy method calling changes every time, analyzing method execution order using strategy considered in this section becomes difficult.

5.2 Program Execution Time

We applied our scheme to program computes the four basic operations of arithmetic and measured execution time. P0’ is the program which has branch and executes route A (executes `method3` and `method4`) in figure 4, and P0” is the program which has loop and repeats framed part 5 times in figure 5. P1 and P2, P1’ and P2’, P1” and P2”, are programs obfuscated P0, P0’, P0” respectively. The frequency of random number generation differs. Random numbers are not

```
a=6, b=3
add //method operates addition
c=9
sub
d=3
mul //method operates multiplication
e=18
div //method operates division
f=2
```

Fig. 12. Output result after dynamic analysis

```
static void dummy(int p){
  int x=(int)(Math.random()*100);
  if(((2*x+1)%2)==0){
//condition which is always false
     method4();}
  else{p=2*(p+3)-p+6;}
//dummy operation for p
  switchfunc(p);}
```

Fig. 13. Outline of dummy method

Table 1. Measurement result of program execution time

Source code	Frequency of random Number generation	Execution time (10^{-6}s)	Increase rate of Execution time(%)
P0	-	605	-
P1	8	619	2
P2	40	627	4
P0'	-	725	-
P1'	10	736	2
P2'	50	744	3
P0"	-	2,574	-
P1"	10	3,000	17
P2"	50	3,000	17

the same in each execution, so we gave a number sequence consists of probable value which will be generated during execution. Table 1 shows the result. The experimental environment is as follows.

– Processor:Intel Pentium III, 1GHz
– Memory:512MB RAM
– Windows 2000 Service Pack 4
– j2sdk-1_4_2_06-windows-i586-p.exe

From table 1, the difference of the execution time between obfuscated programs and original program is less than 1/1000 seconds. Thus, we can say the frequency of random number generation has a litte influence to the execution efficiency, and execution time between original program and obfuscated program by our scheme.

6 Conclusion

In this paper, we introduced software obfuscation scheme using random numbers. We explained how to obfuscate control flow and extended this scheme to apply to the program having complicate control flow.

After introducing our obfuscation scheme, we studied about the scheme. In our scheme, the more the method number is, the bigger the cost of static analysis becomes. And we confirmed the influence of random number generation on execution efficiency is small. We found out that our scheme is vulnerable to dynamic analysis, then explained a countermeasure that inserting dummy method which has no influence on program execution result. In the future work, we consider scheme to make distinguishing dummy method and original method difficult, and study evaluation about quantitive security analysis of proposed scheme.

References

1. Akito Monden, Yoshihiro Takada, Kouji Torii, "Methods for Scrambling Programs Containing Loops," IEICE Trans. D-I Vol.J80-D-I No.7 pp.1-11, July 1997.
2. T. Ogiso, Y. Sakabe, M. Soushi, "Software obfuscation on a theoretical basis and its implementation," IEICE Trans. Fundamentals, Vol.E86-A,No.1, pp.176-186, 2003.
3. Kazuhide Fukushima, Toshihiro Tabata, Kouichi Sakurai, "Proposal and Evaluation of Obfuscation Scheme for Java Source Codes by Partial Destruction of Encapsulation," Proc. of International Symposium on Information Science and Electrical Engineering 2003 (ISEE 2003), pp.389-392 (11, 2003).
4. Hirotsugu Sato, Akito Monden, Ken-ichi Matsumoto, "Program Obfuscation by Coding Data and Its Operation," Technical Report of IEICE, Techinical Group on Information Theory, Vol. IT2002-49, pp.13-18, Mar. 2002.
5. D. Low, "Java control flow obfuscation", Master of Science Thesis, Department of Computer Science, The University of Auckland, 1998.