Evaluation of Obfuscation Scheme Focusing on Calling Relationships of Fields and Methods in Methods

Kazuhide FUKUSHIMA
Graduate School of Information Science
and Electrical Engineering
Kyushu University
6-10-1 Hakozaki, Higashi-ku
Fukuoka, 812-8581 Japan
email: fukusima@itslab.csce.kyushu-u.ac.jp

Toshihiro TABATA, Kouichi SAKURAI
Faculty of Information Science
and Electrical Engineering
Kyushu University
6-10-1 Hakozaki, Higashi-ku
Fukuoka, 812-8581 Japan
email: {tabata,sakurai}@itslab.csce.kyushu-u.ac.jp

ABSTRACT
Recently, Java has been spread widely. However, Java has a problem that an attacker can reconstruct Java source codes from Java classfiles. Therefore many techniques for protecting Java software have been proposed, but, quantitive security evaluations are not fully given. This paper proposes an obfuscation scheme for Java source codes by destructing the encapsulation. In addition, we propose an evaluation scheme on the number of accesses to the fields and the methods of the other classes. We try to realize tamper-resistant software with the certain quantitive basis of security using our evaluation.

KEY WORDS
Obfuscation, Software security, Java, Software metrics

1 Introduction

1.1 Background

Recently, cost of hardware is dramatically decreasing by innovations in techniques. On the other hand, cost of software is increasing relatively. It is one cause that the scale of software became large. We come to be able to perform complicated processing by improvement in the capability of computers. Besides, software development relies on manpower though we use today's technologies. That is another cause.

In case software circulates widely through networks, we should give careful consideration to protection of copyrights. Protection against stealing of key algorithm and secret data in a program is true for this. Software development requires many man-days. However, an attacker can steal the software easily. That is why information security technology to protecting software is required.

Java has spread widely. Java source codes are compiled to object files called Java classfile. The classfiles are executed on the executer called Java Virtual Machine (JVM). The same classfiles can run on different platforms because the JVM for each platform is prepared. Java programs can run on portable phones and small information terminals such as Personal Digital Assistants (PDA).

On the other hand, Java classfiles contain information such as name of class, name of super class and names of methods and fields defined in the class file. This information may be a clue to analyze the classfile. Moreover, the description of class file can be divided into description of field and methods[2]. Therefore, Java classfiles have high readability. As the result, an attacker can obtain Java source codes easily by decompiling Java classfiles. He can steal secret data and key algorithms by his reverse engineering of the obtained source code.

Obfuscation makes software difficult to analyze, while keeping the functionality. We can make the theft of program difficult by applying obfuscation.

1.2 Our contribution

1.2.1 Obfuscation scheme focusing on the encapsulation of Java

We propose an obfuscation scheme for Java source codes focusing on properties of object-oriented languages. We transfer local variables and instructions groups in an arbitrary method to the other classes. Static variables and static methods in Java belong to a class itself instead of an object of the class. Therefore, the variables and methods can be accessed from any classes in form of "classname. identifier". We change local variables into static variables, and instructions groups into static methods for transferring them to another class.

Generally, fields and methods is defined as a whole in Java program. We can not massage private filed without using public method from an external class. Thus, each class have high independence. For the reason, it is easy to understand a Java program by analyzing each classfile constructing the Java program. On the other hand, encapsulation structure is destroyed in a Java program obfuscated by our scheme. The independency of each class becomes weaker. In this situation, an analysis for every classfile

does not make sufficient sense. That is, in order to analyze a classfile, we must investigate the other classes contains static fields and static methods accessed from the class. As the result, it is inefficiently to analyze each classfile individually. That is, in order to analyze a classfile, we must investigate the other classes contains static fields and static methods accessed from the class. Therefore, the breaking time of obfuscated program is longer than that of the source program.

1.2.2 Evaluation scheme

We propose the evaluation scheme for our obfuscation scheme. The evaluation scheme is based on the number of accesses to fields and methods of the other classes. First, complexity of the class is defined as the total number of fields and methods declared in the other classes accessed from the class. Next, complexity of the Java program is defined as the total complexity of each class declared in the Java program. Finally, the effectiveness of the obfuscation (*Effect*) is defined as difference the complexity of the obfuscated program and the complexity of the original program.

1.2.3 Experiments

We evaluate the proposed obfuscation scheme by our evaluation scheme. We investigate a correlation between and the *Effect*s, software metrics, and breaking times of programs. Our results shows *Effect* and software metrics showing the complexity of a program increase by applying our obfuscation scheme repeatedly. And, the increase rate of breaking time to that of *Effect* tend to becomes high as *Effect* increases. Finally, we confirm that execution time increases only 54% even when the breaking time become 10 times.

2 Related works

Many techniques for protecting software have been proposed. In this section, we explain encryption, server-side execution, and obfuscation.

We can protect programs by encrypting them. However, encrypted programs can not run as they are. Therefore, they must be decrypted before execution or be executed on the executor with decoder.

In server-side execution, a program are divided into a private part, which executes on the server, and a public part, which runs locally on the user's computer[3]. The private part of the program can be protected entirely, because an attacker can not get it. However, server-side execution requires communication between the server and clients. Moreover, in case many user use this program at the same time, the load of the server will increase.

Obfuscation makes a program difficult to analyze, while functionality are preserved. Obfuscated program can be run on a computer, as well as original program. We can mitigate the danger that users or third persons analyze by distributing obfuscated program.

Many obfusacation schemes have been proposed. Monden et al. proposed an obfuscation scheme for C programs contain loops[4]. Collberg et al. proposed an obfuscation scheme for Java programs by injecting dummy codes and complicating data structures and control flows[5]. Few obfuscation schemes have basis of security. Wang et al[6] showed that statically determining precise indirect branch addresses is a NP-complete problem in the presence of general pointer. And, they proposed an obfuscation scheme using global arrays and pointers of C programs in their scheme. Ogiso et al[8] proposed an obfuscation scheme that makes interprocedual analysis of C programs difficult using function pointers. They showed that the problem of determining the address a function pointer points to is NP-hard. Sakabe et al. proposed an obfuscation scheme for Java programs using properties of object-oriented languages. Their schemes is based on difficulty of Java programs containing interfaces and method overloads[9]. Their obfuscation scheme has the same security as obfuscation schemes[6, 9].

In case we protect digital contents with information security technology, we should realize the security according to worth of contents. When we protect software by obfuscation scheme, we have to realize the security according to worth of software. In order to prove this security, it is required to be able to evaluate the effect of obfuscation quantitatively. However, quantitive security evaluations of obfuscation schemes are not fully given.

3 Proposed obfuscation scheme

3.1 Our view

This paper proposes an obfuscation scheme for Java source codes focused on properties object-oriented languages. Data structures (fields) and operations to them (methods) of a Java classes are intimately tied together. We call it encapsulation. We can operate private fields only by using public methods from external classes. As the result, each classfile has high independency. We obfuscate Java source codes by destruction of the encapsulation.

Static variables and static methods in Java belong to a class itself instead of a specific object of the class. Therefore, the variables and methods can be accessed from any classes in form of "classname. identifier". In our proposed obfuscation scheme, local variables and instructions groups are transferred to the other classes as static fields and static methods. In this situation, an attacker has to know structures of all classes when he analyzes a Java program by reverse engineering. It means that the Java source code become difficult to analyze.

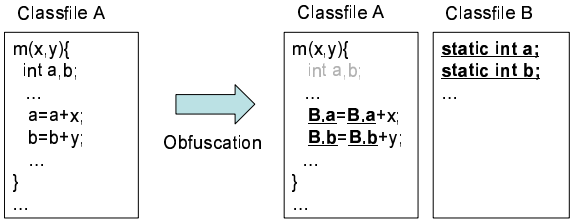

Figure 1. Transference of local variables

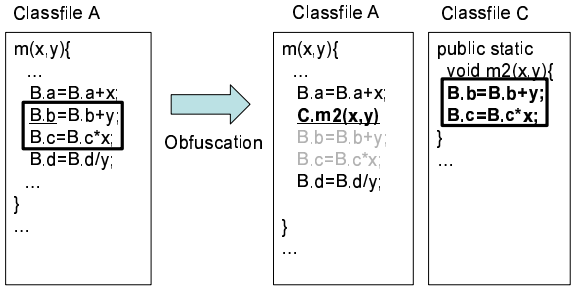

Figure 2. Transference of instructions groups

3.2 Transference of local variables in a method

This technique is applicable to an arbitrary method `m` of an arbitrary class `C`.

1. Choose an arbitrary local variable `v`, which is transferred to the other classes, from method `m`.

2. Choose an arbitrary class `C'` where we place the variable. And, declare the variable `v` as a new static field in the class `C'`.

3. Correct the statements which access the local variable chosen `v` in method `m`. That is, static fields declared in (2) is accessed from method `m` in form of "`C'.v`".

4. Finally, delete the variable declaration of local variable `v` in method `m`.

Fig.1 shows an example of obfuscation by transferring of local variables. In this example, two local variables `a` and `b` in class `A` are transferred to class `B`. New static fields `a` and `b` are declared in class `B`.

3.3 Transference of instructions groups in a method

This technique is applicable to an arbitrary method `m` of an arbitrary class `C`.

1. Choose an arbitrary instructions group `(s0, s1, ..., sn)` , which are transferred to the other classes, from the statements set of `m` except assignment statements for local primitive variables.

2. Choose an arbitrary class `C'` where we place the instructions group. And, declare the a new static method `m'` consist of instructions group `(s0, s1, ..., sn)`.

3. Describe call to static method `m'`, in the spot in front of instructions group `(s0, s1, ..., sn)` in `m`. Static method `m'` is called in form of "`C'.m'` (arguments)".

4. Correct the access modifications of all fields, accessed from static method `m'`, to **public**, so that `m` can access to these fields.

5. Finally, delete instructions group `(s0,s1,..., sn)` in method `m`.

In Java, primitive arguments are called by value[1]. Therefore, if assignment statements for local primitive variables of method `m` are transferred to the other classes, the variables can not be changed from static method `m'`. This is why assignment statements for local primitive variables are excluded in (1).

Fig.2 shows the example of obfuscation by transferring of instructions group. In this example, two statements `B.b=B.b+y` and `B.c=B.c*x` in class `A` are transferred to class `C`. A new static method `m2` is declared in class C and method call to `m2` is described in class A.

3.4 Consideration

Monden et al. claim that a obfuscation τ must satisfy the following two properties and shuld satisfy a propety [4]. We confirm that our obfuscation scheme satisfies the properties.

- The output of obfuscated program τ(P) is the same as that of source program P.
- The breaking time of τ(P) is longer than that of P.

These two properties must be saticefied.

- The execution time of τ(P) is not much longer than that of P.

This property is not nessesary but desirable to satisfy.

3.4.1 Equivalence of output

Even if local variables in a method are transferred to another class as static fields, the method can access the variables by specifying the class name and the field name. Moreover, even if instructions groups are transferred to

another class as static methods, the procedure does not change. The difference of the scope of variables can solved by passing arguments. Therefore, the equivalence of the programs are preserved.

3.4.2 Breaking time of program

Java class is generally designed to realize a specific functionality. The variables and statements for realizing the functionality are defined as a class. We have only to analyze each class file to understand the program. On the other hand, encapsulation structure of obfuscated program is destroyed. As the result, it is inefficiently to analyze each classfile individually. That is, in order to analyze a classfile, we must investigate the other classes contains static fields and static methods accessed from the class. Therefore, the breaking time of obfuscated program is longer than that of the source program.

3.4.3 Execution time

Even if local variables in a method are transferred to another class as static fields, the access speed hardly changes. We confirm the fact by measuring access time to local variables and static field in Java program. Moreover, even if instructions groups are transferred to another class as static fields, the access speed hardly changes. Because the procedure of the instruction groups statements does not change unless dummy codes are injected or control flows are complicated.

4 Evaluation

4.1 Proposed evaluation scheme

We give an evaluation scheme for our proposed obfuscation scheme. The evaluation scheme is based on the number of accesses to fields and methods of the other classes. $e(\mathrm{C})$ represents the complexity of the class C. e(C) is defined as the total number of fields and methods declared in the other classes accessed from C. $E(\mathrm{P})$ represents the complexity of the Java program P. $E(\mathrm{P})$ is defined as the total complexity of each class declared in P. That is,

$$E(\mathrm{P}) = \sum_{\mathrm{C} \in \mathrm{P}} e(\mathrm{C}).$$

Finally, $\mathit{Effectiveness}(\tau)$ represents the effectiveness of the obfuscation τ. $\mathit{Effectiveness}(\tau)$ is defined as difference the complexity of the obfuscated program and the complexity of the original program. That is,

$$\mathit{Effectiveness}(\tau) = E(\tau(\mathrm{P})) - E(\mathrm{P}).$$

Table 1. The properties of source codes

	P0	P1	P2	P3	P4	P5
Size (B)	369	563	924	959	1428	1939
Lines	26	39	48	54	78	105
Methods	2	4	4	6	10	19
Classes	2	3	4	5	5	6
Complexity	2	4	19	29	34	45
Effectiveness	0	2	17	27	32	43
Execution time (10^{-6} s)	539	664	673	747	817	831

Table 2. The breaking times

	P0	P1	P2	P3	P4	P5
subject1	360	240	350	550	600	1600
subject2	150	400	580	1020	1120	1780
subject3	120	320	560	840	1230	1880
subject4	120	380	620	940	1150	2120
subject5	180	350	650	730	1250	2660
average	186	338	552	816	1070	2008

4.2 Experiments

4.2.1 Procedure of experiment

We investigate a correlation between each subject's breaking time of obfuscated programs and the *Effectiveness*es of them in order to confirm that our evaluation scheme is suitable. The procedure of the experiment is as follows.

1. We make a source code P0 of the program that outputs first 20 terms of the Fibonacci sequence.

2. We make source codes P1, P2, P3, P4, and P5. These source codes are obtained by obfuscating the P0. **Table.1** shows the detail of each source code. The output of these program is same, but, the *Effectiveness*es of them are different respectively.

3. The six sources codes (P0 to P5) are passed to five subjects. The subjects break the programs. We regard the breaking of a program is completed when the subject understands the execution path, operations and data flows along the path of the program. The subjects records the breaking times of source codes P0 to P5. **Table.2** shows the results.

4.2.2 Results

Fig.3 shows a correlation between the breaking times and the *Effectiveness*es of the obfuscations. The breaking times and the *Effectiveness*es of the obfuscations have a positive correlation, though there are individual differences. Next, **Fig.4** shows the correlation between the average of five

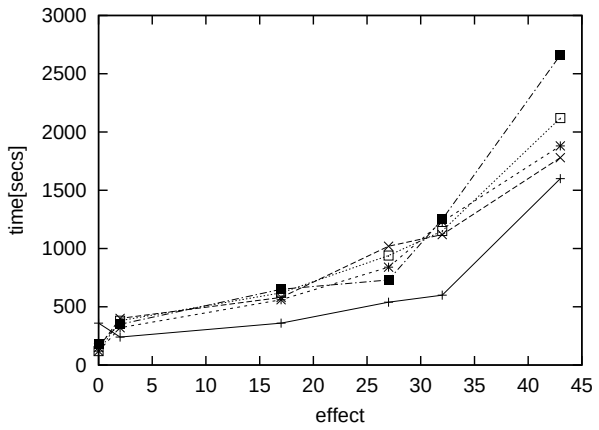

Figure 3. Correlation between each subject's breaking time of obfuscated programs and the *Effectiveness*s of obfuscations

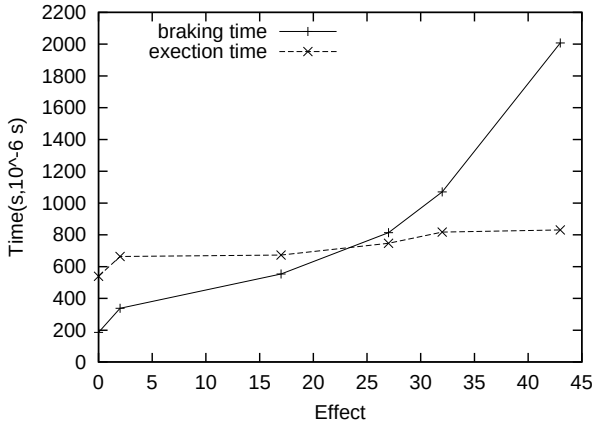

Figure 4. Correlation between the average of five subjects' breaking time, the breaking time, and the *Effectiveness*s of obfuscations

subjects' breaking time, and the *Effectiveness*es of the obfuscation. The breaking time grows more rapidly than the *Effectiveness*es of obfuscations.

4.3 Evaluation by Software Metrics

4.3.1 Metrics

We evaluate six source codes (P0 to P5). We use following six object oriented software metrics.

- Number of message sends (NOM)
  We can measure the size of a method without deviation. Mutual relation with other classes increases as number of message sends increases. Thus, the analysis of the program becomes more difficult.

- Depth of inheritance tree (DIT)
  The possibility of a method being overwritten or being extended becomes high as the inheritance tree becomes deep. Thus, the analysis of the program becomes more difficult.

- Number of instance method (NIM)
  Number of objects, which cooperates with instance, increases as number of instance increases. The class with many instance methods will do many processing. This class is complicated.

- Number of class method (NCM)
  Number of class methods shows the quantity of the common procedure which can be done to all instances. If services, applied by each instance, are offered by the class itself, there are many condition branches based on the data type. In this case, class is complicated.

- method complexity (MCX)
  We calculate the method complexity using the following weight. The method complexity becomes larger as the method becomes more complicated.

  - API calls (5.0)
  - Substitution (0.5)
  - Arithmetical operation (2.0)
  - Messages with arguments (3.0)
  - Nested expression (0.5)
  - Arguments (0.3)
  - Primitive calls (7.0)
  - Local variables (0.5)
  - Messages without arguments (1.0)

- Class cohesion (CCO)
  Class cohesion is the number of kinds of messages. Class cohesion increases as cooperative relation with other classes become increases. The classfiles with strong dependence is difficult to analyze, since it is inefficiently to analyze each classfile individually.

4.3.2 Results

Table.3 shows software metrics for source codes (P0 to P5). NOM and CCO increased because the number of messages and the number of kinds of them increased. The increase of NOM and CCO indicates relations between classes become difficult. NCM increased because we redefine all transferred instructions groups as class methods. The increase of NCM indicates class becomes complicated. Moreover, MCX increases because number of messages and arguments increase. DIT and NIS did not increase, because our proposed obfuscation scheme does not focus on these properties.

Table 3. Software metrics of source codes

	P0	P1	P2	P3	P4	P5
NOM	3	5	5	7	12	24
DIT	1	1	1	1	1	1
NIM	1	0	0	0	0	0
NCM	1	3	3	5	9	18
MCX	24.8	32.7	42.0	51.0	65.3	151.4
CCO	3	5	5	7	10	20

4.4 Consideration

All the obfuscated programs (P1 to P5) output the first 10 terms of the Fibonacci sequence as same as original program (P0). According to the fact, we confirm the equivalence of the programs. Next, we confirm that *Effectiveness*es of the programs increase as applying our obfuscation scheme repeatedly. Software metrics which indicate complexity of relations between classes also increase. The increasing rate of the *Effectiveness* tends to increase as the *Effectiveness* increases. Moreover, the breaking times and the *Effectiveness*es have a positive correlation. Thus, we can see that breaking time of the obfuscated programs are longer than that of original program. Finally, it is hoped that the execution time of the obfuscated programs is almost equal to that of the original program. Unfortunately, However, the execution time of the program (P5) increases only 54% while the breaking time becomes 10 times.

5 Conclusion

We propose an obfuscation scheme for Java source codes focusing on properties of object-oriented languages. Classfiles in obfuscated program by our scheme have strong dependence. That is, in order to analyze a classfile, we must investigate the other classes contains static fields and static methods accessed from the class. Therefore, the breaking time of obfuscated program is longer than that of the source program. As the result, it is inefficiently to analyze each classfile individually.

In addition to, we propose the evaluation scheme for our obfuscation scheme. The evaluation scheme is based on the number of accesses to fields and methods of the other classes.

Moreover, we investigate a correlation between and the *Effect*s, software metrics, and breaking times of programs by experiments. Our results shows *Effect* and software metrics showing the complexity of a program increase by applying our obfuscation scheme repeatedly. And, the increase rate of breaking time to that of *Effect* tend to becomes high as *Effect* increases. Finally, we confirm that execution time increases only 54% even when the breaking time become 10 times.

References

[1] J. Gosling, B. Joy, G. Steele, and Gilad Bracha, *The Java Language Specification Second Edition*, Pearson Education Company, 2000.

[2] T .Lindholm and F.Yellin", *The Java Virtual Machine Specification*, Pearson Education Company, 1999.

[3] D. J. Albert and S. P. Morse, "Combating software piracy by encryption and key management," *IEEE Computer*, pp. 68–73, 1984.

[4] A. Monden, Y. Takeda, and K. Torii, "Methods for scrambling programs containing loops," *IEICE Transactions on Information and Systems*, vol. J80-D-I, no. 7, pp. 644–652, 1997.

[5] C.Collberg, C.Thomborson, and D.Low, "A taxonomy of obfuscating transformations," Technical Report of Deptartment of Computer Science 148, University of Auckland, New Zealand, 1997.

[6] J.Knight C.Wang, J.Hill and J.Davidson, "Software tamper resistance: obfuscating static analysis of programs," Technical report sc-2000-12, Department of Computer Science, New Zealand, 2000.

[7] M .Lorenz, J .Kidd "Object-Oriented Software Metrics," Pearson Education POD, 1994.

[8] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, "A new approach of software obfuscation based on the difficulty of interprocedural analysis," *IEICE Transactions on Fundamentals*, vol. E86-A, no. 1, pp. 176–186, 2003.

[9] Y. Sakabe, M. Soshi, and A. Miyaji, "Software obfuscation for object oriented languages," Technical report of IEICE No.95, Vol.42 , pp. 38–43, 2002.