PROPOSAL AND EVALUATION OF OBFUSCATION SCHEME FOR JAVA SOURCE CODES BY PARTIAL DESTRUCTION OF ENCAPSULATION

Kazuhide FUKUSHIMA

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

Toshihiro TABATA, Kouichi SAKURAI

Kyushu University
Faculty of Information Science
and Electrical Engineering
6-10-1 Hakozaki, Higashi-ku,
Fukuoka, 812-8581 Japan
{tabata,sakurai}@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 a tamper-resistant software with the certain quantitive basis of security using our evaluation.

1. INTRODUCTION

1.1. Background

Recently, Java has spread widely. Java is an object-oriented language released by Sun Microsystems in 1995. 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) as well as PCs and Workstations.

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[2]. Moreover, the description of class file can be divided into description of field and methods. Therefore, Java classfiles have high readability. As the result, an attacker can obtain Java source codes easily by decompiling Java classfiles. He can crib secret data and key algorithms by his reverse engineering of the obtained source code.

Software developers are frightened by the prospect of a competitor being able to extract secret data and key algorithm in order to incorporate them into their own programs. The competitor may intercept their commercial edge by cutting development time and cost. To make matters worse, it is difficult to detect and pursue the injustice.

1.2. Our contribution

We propose an obfuscation scheme for Java source codes that focuses on properties of object-oriented languages. We transfer local variables and compound statements in an arbitrary method of a class to the other classes. This obfuscation scheme can make a Java source code difficult to read by destruction the encapsulation structure of Java. Moreover, we proposed 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. In addition, we investigate the correlation between breaking time of obfuscated programs and the *Effect*s of obfuscations provided by our evaluation. Our result shows the breaking time grows more rapidly than the *Effect*s of obfuscations. The property is desirable for realization of tamper-proof software with the certain quantitive basis of security.

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 can be broken into a private part, which executes on the server, and a public part, which runs locally on the user's site[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.

Obfuscation makes software difficult to analysis, while its functionality are preserved. 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. proposed an obfuscation scheme based on the fact that the problem of determining precise indirect branch target addresses is NP-hard[6]. They used global arrays and pointers of C programs in their scheme. Ogiso et al. proposed an obfuscation scheme based on the difficulty of interprocedual analysis of C programs. They showed that the problem of determining the address a function pointer points to is NP-hard[7]. 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[8].

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 compound statements 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.

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'`.

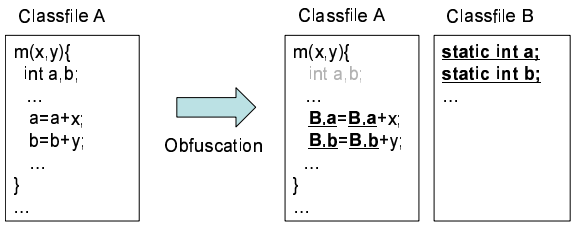

Fig. 1. Transference of local variables

(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 compound statements in a method

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

(1) Choose an arbitrary compound statements `(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 compound statements. And, declare the a new static method `m'` consist of compound statements `(s0, s1, ..., sn)`.

(3) Describe call to static method `m'`, in the spot in front of compound statements `(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 compound statements `(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).

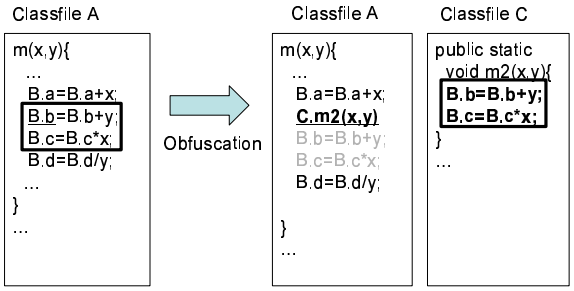

Fig. 2. Transference of compound statements

Fig.2 shows the example of obfuscation by transferring of compound statements. 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 should be satisfy the following three properties[4]. We confirm that our obfuscation scheme satisfies the properties.

- The output of obfuscated program $\tau(\mathrm{P})$ is the same as that of source program P.
- The breaking time of $\tau(\mathrm{P})$ is longer than that of P.
- The execution time of $\tau(\mathrm{P})$ is not much longer than that of P.

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 compound statements 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 impossible 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 compound statements are transferred to another class as static fields, the access speed hardly changes. Because the procedure of the compound statements does not change unless dummy codes are injected or control flows are complicated.

4. PROPOSED EVALUATION SCHEME

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

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

4.2. Experiment results

4.2.1. Procedure of experiment

We investigate the correlation between each subject's breaking time of obfuscated programs and the *Effect*s 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 *Effect*s 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 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
Effect	0	2	17	27	32	43
Execution time (10^{-6} s)	539	664	673	747	817	831

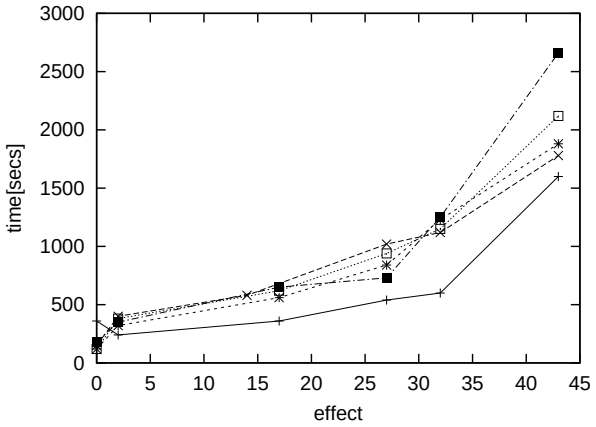

Fig. 3. Correlation between each subject's breaking time of obfuscated programs and the *Effect*s of obfuscations

4.2.2. Result of experiment

Fig.3 shows the correlation between the breaking times and the *Effect*s of the obfuscations. The breaking times and the *Effect*s of the obfuscations have a positive correlation, though there are individual differences. Next, **Fig.4** shows the correlation between the average of five subjects' breaking time, and the *Effect*s of the obfuscation. The breaking time grows more rapidly than the *Effect*s of obfuscations.

4.3. Consideration

We can see that the breaking time grows more rapidly than *Effect*s. Therefore, we can lengthen the breaking time sufficiently by applying our obfuscation scheme repeatedly. The property is desirable for realization of tamper-proof software.

5. CONCLUSION

We proposed an obfuscation scheme for Java source codes. We could obfuscate Java source codes by transferring local variables and compound statements in an arbitrary method of a class file to another class file. And we proposed an evaluation scheme for our obfuscation scheme. We investigate the correlation between each subject's breaking time

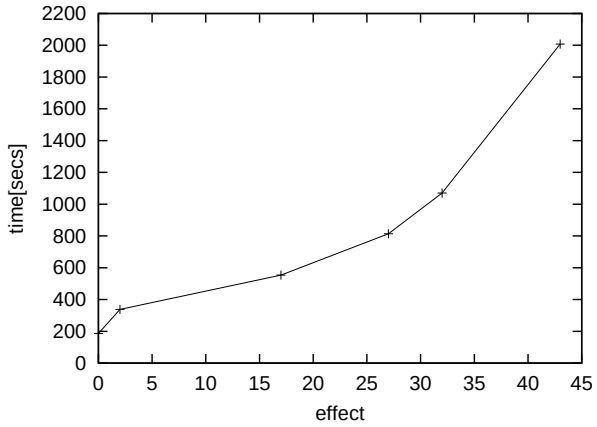

Fig. 4. Correlation between the average of five subjects' breaking time, and the *Effect*s of obfuscations

of obfuscated programs and the *Effect*s of them. As the result, the breaking time grows more rapidly than the *Effect*s of obfuscations.

Our future work is improving our evalustion scheme.

6. 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," *Transaction of IEICE*, 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] 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.

[8] Y. Sakabe, M. Soshi, and A. Miyaji, "Software obfuscation for object oriented languages," Technical report of ieice (ISEC02-6), pp. 38–43, 2002.