Comparative Analysis of Code Obfuscation Approaches to Protect Software Products

Hyun-Il Lim

***Abstract*—As the recent improvement in information technology, the role of software becomes more and more important in computing environments. Software is generally released as a form of binary information, such as instructions and data for executing programs. Although such software is protected by software licenses, it may be analyzed or reverse-engineered to employ in developing other software without permission. Code obfuscation is an approach to making original code harder to analyze or understand by transforming its original code into different form with preserving semantics or execution results. Code obfuscation is helpful to protect against reverse engineering of software. In this paper, we introduce several approaches to obfuscating program code to make software harder to analyze or understand. We introduce measures for evaluating code obfuscation and compare the performance of each code obfuscation methods according to the measures.**

***Index Terms*—Code obfuscation, program transformation, binary code analysis, software protection.**

I. Introduction

As the recent improvement in information technology, the role of software becomes more and more important in computing environments. The development cost of software is also increasing. So, various open source software is good references for developing new software. In other hand, many of software is protected by software license or copyright law, and they are not permitted to be used in developing other software without permission.

Software is generally released as a form of binary information, such as instructions and data for executing programs. Although such software is protected by software licenses, it may be analyzed or reverse-engineered to employ in developing other software without permission. To cope with such problems, many researches are in progress in various areas, such as software theft detection, software watermark, and software birthmark.

Code obfuscation is an approach to making original code harder to analyze or understand by transforming its original code into different form with preserving semantics or execution results. So, after obfuscating a program, its obfuscated version of program must have same output as its original version.

The goal of code obfuscation is to prevent from being illegally reused by making an original program more difficult to analyze or decompile. So, this approach is helpful to protect against reverse engineering of software. Obfuscation can be classified as several categories according to the kind of information it targets [1]-[3]. In this paper, we introduce several approaches to obfuscate program code to make software harder to analyze or understand. We introduce evaluation measures for code obfuscation and compare the performance of each code obfuscation methods according to the measures.

The remainder of this paper is organized as follows. In Section II, we introduce several approaches to obfuscating program code to make program difficult to analyze. In Section III, we introduce measures for evaluating code obfuscation methods, and show the comparison results of each method. Finally, in Section IV, we conclude this paper.

Manuscript received May 30, 2015; revised November 10, 2015.

Hyun-Il Lim is with the Dept. of Computer Engineering, Kyungnam University, Republic of Korea (e-mail: hilim@ kyungnam.ac.kr).

II. Code Obfuscation

A. Layout Obfuscation

Layout obfuscation is a method to modify the layout structure of a program, such as identifier names, comments, and debugging information. Layout structure has many information, but it is unnecessary information in executing a program. So, modification of such information does not affect the semantics of a program. However, layout structure may provide much information to reverse engineers. So, modification of layout structure makes a program more difficult to analyze.

This type of obfuscation is effective in protecting source code of software, which has plenty of identifiers and comments. For example, informative identifier names may be renamed to meaningless ones, or debugging information may be removed. Layout obfuscation is unlikely to prevent a program from being reverse engineered, but it can make code analysis or reverse engineering more time-consuming.

B. Data Obfuscation

Data obfuscation is a method to modify data structures used in a program to make difficult to analyze. Data obfuscation is classified into three groups according to the method of modification [2]-[4].

1) Storage and encoding obfuscation

Storage obfuscation modifies the representation of storages used in a program. This obfuscation can be applied in several ways as following examples:

1) A local variable can be converted into a global one.
2) A variable can be split into more than one variable.
3) A scalar variable can be integrated into a more complex object.

Encoding obfuscation is a way to change the access method of variables in a program. For example, static data can be converted to a procedure, which returns the data after

DOI: 10.7763/IJCTE.2017.V9.1106

performing several operations to protect the software. For example, a constant value 1 can be converted to a formula, $f(a, b)=(b+1-a)/\cos(a+\pi-b)$ with assumption that $a = b$. logic values true and false can be encoded into ($x \parallel 1$) and (y && 0), respectively [3].

TABLE I: AN EXAMPLE OF ENCODING OBFUSCATION

int *i* = 1;
while (*i* < 100) {
... A[*i*] ... ;
i++;
}
	int *i* = 7;
	while (*i* < 207) {
	... *A*[(*i*-5)/2] ... ;
	i += 2;
	}

After analyzing data access patterns, an original program can be obfuscated by modifying the access patterns of the data. Table I Shows an example of encoding obfuscation that changes access pattern of data in a program. In the left program code, the array A[*i*] is accessed with the index ranges between 1 and 99. By modifying the access pattern of the array, the integer variable *i* is replaced by $2 \times i + 5$ in the right program [4].

2) Aggregation obfuscation

Aggregation obfuscation modifies grouping methods of data, such as merging independent data or splitting dependent data. This obfuscation can be applied in several ways as following examples:

1) A two-dimensional array can be converted into one or two one-dimensional arrays.
2) Several dependent scalar variables can be merged in a single array.
3) Several data structures can be modified by adding some redundant objects [3], [4].

3) Ordering obfuscation

Ordering obfuscation is a method for modifying order of data by restructuring the layout of the data. The examples are as follows [3], [4].

1) Variables or methods in a program can be reordered.
2) Array elements can be reordered in a different order. The reordered position of the *i*-th array element can be retrieved via an index function *f*(*i*), which returns the *i*-th element of the original array.

C. Control Flow Obfuscation

Control flow obfuscation [5]-[7] changes control flows of a program to have different control structures. Control flow obfuscation can be applied in several ways according to the method of modifying the control flows of programs.

1) Aggregation obfuscation

Aggregation obfuscation modifies grouping methods of program statements by splitting or merging fragments of codes. The examples are as follows [3].

1) Method inlining can replace every method calls in a program with the statements in the called method after obfuscating the statements.
2) Outlining statements can replace a sequence of statements with a call to an intentionally created method that performs the sequence of statements.
3) Loop unrolling can change statements in loop body or replace loop with a sequence of repeated statements in the loop body.

2) Ordering obfuscation

Ordering obfuscation is a method of modifying the execution order of statements in a program with maintaining their dependence relations. The examples are as follows [3], [4].

1) Basic block reordering can modify the order of basic blocks by using branch instructions.
2) Loop reordering can change the order of loops in a nested loop.
3) Expression reordering can change the order of statements by introducing some additional statements.
4) Loop evaluation order can be reversed by iterating backwards instead of forwards.

3) Computation obfuscation

Computation obfuscation modifies the main structures of a program to hide its real control flows. The examples are as follows [4].

1) Smoke and mirrors obfuscation hides real control flow by introducing bogus instructions. For example, dead code can be inserted between real control flows.
2) High-level language breaking obfuscation (reducible to nonreducible flow graphs) introduces features that are supported by object code level but not supported by source code level. For example, Java language has no goto statement, but Java bytecode supports goto instruction. So, inserting a jump into the middle of loop makes the program code awkward, and the program cannot be transformed back to the original loop.
3) Control flow abstraction modifies statements of a program into more abstract ones. Conceptually, it is the inverse of transformation that can be done by compiler optimizations. Table II shows an example of control flow abstraction. In the right program, redundant condition is inserted in the while loop to the original one without changing the results.

TABLE II: AN EXAMPLE OF CONTROL FLOW ABSTRACTION

int *i* = 1;
while (*i* < 1000) {
...
i++;
}
	int *i* = 1;
	while ((*i* < 1000)
	|| (*i*%1000 == 0)) {
	...
	i++;
	}

4) Obfuscation via Opaque Predicates

An opaque predicate [8] means a predicate that is always evaluated to either true or false regardless of its run-time environments. A predicate P is opaque at program point t if its outcome is known at obfuscation time. We write an opaque predicate as P^T or P^F if the opaque predicate P always evaluates to true or false at program point t, respectively. The property is that although the outcome of a predicate is known at obfuscation time, the evaluation of the predicate is not easy without executing the program. From this property several bogus instructions or methods can be inserted via opaque predicates to make a program difficult to analyze.

Table III shows an example of control obfuscation via insertions of opaque predicates. A sequence of instructions are split into two parts by inserting conditional branch with an opaque predicate P^F. In the right program, the opaque predicate is evaluated to be false. So, the else body is executed, but the control flow of the right program is more complex than that of the left program. Because an opaque predicate is not evaluated easily by static analysis, it will confuse reverse engineers in analyzing modified programs.

TABLE III: AN EXAMPLE OF OBFUSCATION VIA OPAQUE PREDICATE

```
                        int i = 1;
                        int j = f(i);
                        if( P^F ) {
int i = 1;                  i++;
int j = f(i);               return i;
j++;                    }
return j;               else {
                            j++;
                            return j;
                        }
```

III. PERFORMANCE COMPARISON

A. Evaluation Measures

The performance of code obfuscation methods can be evaluated via several criteria according to specific measures. The criteria presented in [8] is effective in measuring the performances of obfuscators of software. The performance evaluation measures for code obfuscation are as follows:

1) Potency: How much obscurity it adds to the program [8]. Because the goal of code obfuscation is to make an original program difficult to analyze, a good obfuscation method can make an original programs more complex.
2) Resilience: How difficult it is to break an automatic deobfuscator [8]. If an obfuscated program can be easily reverse-engineered by automatic static analysis, the method cannot be used in practice.
3) Stealth: How well the obfuscated code blends in with the rest of a program [8]. The obfuscated code should not be noticed by reverse engineers to protect software safely.
4) Cost: How much computational overhead it adds to the obfuscated application [8]. If the cost of execution time or space is too high, the method cannot be applied in practical application.

The four evaluation criteria are proper combination for measuring performance of obfuscation methods in various aspects of quality of code obfuscation.

B. Comparison Results

In this section, we will compare the performance of several code obfuscation methods according to criteria presented in the previous section.

Table IV shows the results of performance comparison according to the evaluation criteria. In the evaluation of potency measure, obfuscation methods make original program complex to add much obscurity. The complexity of software is measured in several software complexity metrics, such as McCabe and Harrison metrics [8]. The complexity measures are closely related to this question. "Is it difficult to understand the control structures or data structures of software as compared to its original version?" So, aggregation obfuscation and computation obfuscation have high potency, because they heavily modify the structures of data or control flows of programs.

TABLE IV: THE ARRANGEMENT OF CHANNELS

Obfuscation Methods	Evaluation Criteria			
	Potency	Resilience	Stealth	Cost
Layout obf.	Med	Low	Low	Low
Storage & encoding obf.	Med	Med	Med	High
Agreggation obf.	High	Med	High	Med
Ordering obf.	Med	Med	High	Med
Computation obf.	High	High	High	High
Opaque predicate	Med	High	High	Low

In the evaluation of resilience measure, computation obfuscation and opaque predicate obfuscation have highly ranked. Because computation obfuscation hides the original control flows of programs, it is difficult to understand the original control flow by static analysis.

In the evaluation of stealth measure, it should be difficult to find the fact that software is whether obfuscated or not. In other hand, layout obfuscation and storage obfuscation may be noticed the fact of obfuscation because of abnormal format or instruction patterns in obfuscated programs.

In the evaluation of cost measure, storage obfuscation and computation obfuscation have high cost. The two methods must change wide ranges of a program to transform structures of storage or control flows of programs. Such modifications of wide ranges of program make the obfuscation transformation expensive in cost.

The evaluation results may be different according to development environments or individual obfuscation algorithms. In this paper, the comparison results consider general performance characteristics of each obfuscation method.

IV. CONCLUSION

In the recent computing environments, software is widely used in various application. An enormous number of software is developed in many companies. Software is intellectual property of its author, so it is protected by software license or copyright law. Although software is protected by software licenses, software can be analyzed or reverse-engineered to employ in developing other software without permission. To cope with such problems, many researches are in progress to protect software products.

Code obfuscation is an approach to making original code harder to analyze or understand by transforming its original code into different form with preserving semantics or execution results. The goal of code obfuscation is to prevent from being illegally reused by making an original program more difficult to analyze or decompile. So, this approach is helpful to protect against reverse engineering of software.

In this paper, we introduced approaches to obfuscating program code to make software harder to analyze or understand. We introduced measures for evaluating code

obfuscation, and showed comparison results of code obfuscation methods.

In the future, software will play an increasing role in the most of computing environments. Code obfuscation method is expected to be applied in various applications to protect software from being reused or reverse-engineered illegally.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (NRF-2010-0024658).

REFERENCES

[1] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
[2] D. Low, "Java control flow obfuscation," MS Thesis, Department of Computer Science, University of Auckland, 1998.
[3] G. Wroblewski. (2002). General method of program code obfuscation. [Online]. Available: http://www.mysz.org/papers/147sp.pdf
[4] D. Low, "Protecting java code via code obfuscation," *Crossroads*, vol. 4, no. 3, pp. 21-23, Apr. 1998.
[5] T. W. Hou, H. Y. Chen, and M. H. Tsai, "Three control flow obfuscation methods for java software," *IEE Proceedings-Software*, vol. 153, no. 2, pp. 80-86, April 2006.
[6] J.-T. Chan and W. Yang, "Advanced obfuscation techniques for java bytecode," *Journal of Systems and Software*, vol. 71, no. 1-2, pp. 1-10, Apr 2004.
[7] A. Majumdar, C. Thomborson, and S. Drape, "A survey of control-flow obfuscations," in *Proc. International Conference on Information Systems Security*, LNCS 4332, 2006, pp. 353-356.
[8] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proc. Symposium Principles of Programming Languages (POPL)*, San Diego, CA, January 1998, pp. 184-196.

Hyun-Il Lim received his BS, MS and PhD degrees in computer science from KAIST, Republic of Korea, in 1995, 1997, 2009, respectively. He is currently an assistant professor in the Department of Computer Science and Engineering, Kyungnam University, Republic of Korea. His current research interests include software security, software protection, watermarking, and program analysis.