Novel Obfuscation Algorithms for Software Security

Levent Ertaul
Department of Mathematics & Computer Science
California State University, Hayward,
Hayward, CA, USA.

Suma Venkatesh
Department of Mathematics & Computer Science
California State University, Hayward
Hayward, CA, USA.

Abstract *- Over the years, several software protection techniques have been developed to avoid global software piracy, which has increased over 40% and has caused $11 billion loss. Code Obfuscation is one of these techniques and it is very promising one. Code obfuscation is a form of software protection against unauthorized reverse-engineering. In this paper, we give information about available software obfuscation tool kits in the market, along with JHide and their comparison. We propose three new obfuscation techniques, based on composite functions, which are Array Index Transformation, Method Argument Transformation and Hiding Constants. In addition to that, we also propose a new obfuscation algorithm based on Discrete Logs to Pack the Words and another one, based on Affine Ciphers, to Encode String Literals. Finally, we conclude our paper identifying the need for reviewing the performance of the algorithms as the future scope of our work.*

Keywords: *Software Security, Software Protection Techniques, Code Obfuscation.*

1. Introduction

Fast developments in multimedia and internet technologies have created the need for researching in the areas of securing data. Every company has an intellectual property to protect which often includes algorithms built right into the software that is sold to customers. The secrecy of such software is an edge to beat their competition in the market, so it is not surprising that the approach taken for their protection makes a great deal of difference [1],[2],[3],[4], [5].

Traditionally techniques for securing data resided in the firewalls and gateways of a network or on the operating system of the host. A new idea is to put these defensive mechanisms inside the application software. Vendors of this software distribute them as mobile code in architectural independent formats [1], [2], [4], [5].

Recent statistics [6] show that four out of every ten software programs is pirated worldwide. This is definitely a threat to clean players and thus the global economy. There are two common practices of protecting an intellectual property of a software producer - Legal and Technical methods. Legal methods include getting copyrights on the software and signing legal contracts against creating duplicates. Technical methods include: *Code Authentication, Server side execution, Program Encoding, Code Obfuscation* [1], [4], [7], [8], [9], [10], [11].

Obfuscation is a new area of research in the field of software protection and gaining more attention in recent years [1], [4], [9]. Although the history of first traits of obfuscation techniques dates back to 1990 [12], their impact got higher as Java technologies dominated the software development world. Java is designed to be compiled into a platform independent byte code format, which means decompilation is easier than with traditional native codes. As a result, the Java code can always be reverse-engineered to extract proprietary algorithms from compiled Java programs. Code obfuscation applies transformations to the code to make their analysis very hard and thus safer from being reverse-engineered. They do not change the functionality of the program though. Software protection tools like *Sand Mark* [7], *Dot Obfuscator* [13], *JMangle* [14], *JObfuscator* [15] and *JHide* [16] are all designed based on the principal theories of code obfuscation techniques. Based on our research in the software security field and the capabilities of the existing players we strongly believe in the potential of “code obfuscation“ techniques as a major software protection tool in the near future and hence we created an

obfuscation tool kit *JHide* [16]. In this paper, we propose new obfuscation algorithms based on composite functions, affine ciphers and discrete logs to improve capabilities of *JHide*.

In real world, a change in a quantity is a relative term and can be expressed as a mathematical function. Whenever there is a change in one quantity producing a change in another which, in turn, produces a change in a third quantity then these chain of changes can be represented as a function of functions which are termed as composite functions [17], [18] in mathematics. This phenomenon occurs very frequently in the world of software programs where a change in a constant can cause changes in the way two quantities are compared in a control block. If an attacker manages to change this constant then he can manipulate the results of the control block. To prevent this, a constant can be represented as a complex mathematical function, which is hard and time consuming to be resolved.

In the following sections we discuss software protection techniques in general and present new obfuscation algorithms based on composite functions and other known mathematical techniques [17], [18], [19] such as discrete logs and affine ciphers.

2. Software Protection Techniques

Generally, software code is mobile and distributed across untrusted networks. Their protection must be incorporated into the software and be hardware independent. The main functions of any software protection technique can be listed as detection of pirate attempts to tamper or misuse software, protection against such attempts and alteration of software to ensure that it functionality degrades in an undetectable manner if protection fails. The most common techniques available are *Protection by Server-Side Execution, Hardware based solutions, Protection by Encryption, Protection through Signed Native Code, Tamper Proofing, Software Aging, Watermarking and Code Obfuscation* [1], [4], [7], [8], [9], [10], [11]. Code obfuscation is the idea to hide the code. In this technique, the application is transformed so that it is functionally identical to the original but it is much more difficult to understand. This technique preserves platform independence.

Advantages and disadvantages of the above techniques can be found in [1], [4], [7], [8], [9], [10], [11].

Java is designed to be compiled into a platform independent byte code format, which means that decompilation is easier than with traditional native codes. As a result, the Java code can always be reverse-engineered to extract proprietary algorithms from compiled Java programs. All the drawbacks in other protection techniques make code obfuscation a stronger tool for securing programs written in Java. Although obfuscation attempts to make decompilation a harder task, given enough time and effort, it is possible to retrieve important algorithms and data structures from such an obfuscated code. The aim here is to increase the time and effort required so that it is economically infeasible for a company to reverse-engineer a rival's application [8], [10], [11]. In the next section, we give a brief overview of *JHide* [16] an obfuscation tool kit for Java programs and compare *JHide* with other available tool kits in the market.

3. *JHide* Tool Kit

JHide is an obfuscation tool kit for securing Java programs. It provides a good starting point for beginners to understand various obfuscation algorithms and the issues involved during their implementation. Details of *JHide* and its implementation can be found in [16]. Here we discuss *JHide* in comparison to other available obfuscation tool kits.

3.1. Comparison of *JHide* with other obfuscation tool kits

We have studied the behaviors of obfuscation tool kits which work only on Java source code such as *Sand Mark* [7], *JMangle* [14], *JObfuscator* [15] and *JHide* [16] with respect to parameters like the number of obfuscation algorithms supported by these tool kits, the ease of their use, flexibility of these tools when a new algorithm has to be added, the efficiency of obfuscation on complex Java programs, resources like memory requirement for the tool kits and also the cost incurred to use them Summary of these differences is shown in Table 1 below.

3.2. Obfuscation Algorithms in *JHide*

An obfuscator is a program used to transform program code. The output of an obfuscator is program code that is more difficult to understand but is functionally equivalent to the original. Obfuscation transformations are classified into the following main groups [21]: Layout, Control, Data, Preventive, Splitting, Merging, Reordering, and Miscellaneous like Method Inliner, Method2RMadness, and Name Overloading transformations. JHide primarily supports 30 different obfuscation algorithms. Detailed explanations of these algorithms can be found in [16], [23].

Table 1. Comparison of Obfuscation Tool Kits

	JHide	**Sand Mark**	**JObfuscator**	**JMangle**
Supported Algorithms	30	25	1	1
User Interface	GUI	GUI	C L	GUI and CL
Flexibility	High	High	Low	Low
Complexity	Medium	High	High	High
Resource	Low	High	Medium	Medium
Cost	Free	Free	Trial	Free

3. 3. New Obfuscation Techniques in *JHide*

In *JHide* we have proposed *5* new obfuscation algorithms primarily based on composite functions and known mathematical concepts of discrete logs and affine ciphers.

A function, in mathematics, is a rule, which allows us to work out one set of numbers from another set of numbers [17], [18], [19]. By knowing the fixed cost of renting a telephone for the month, we can calculate the cost per minute to make calls. To do this, we can set up a function to work out the total cost based on the total length of the calls we have made. Combination of two or more such functions will give us a result, which is composite in nature. Such functions are called as composite functions [17], [18], [20], [22].

In general, for any two functions *f* and *g*, the composite function *f o g* is defined by *f o g (x) = f (g(x))* [17], [18], [20], [22]. The domain of *f o g* is the set of all numbers, *x*, in the domain of *g* for which *g(x)* is in the domain of *f*. In the following sections, we will discuss further on the usage of these composites to represent software programs, which would make the code more complex and hard for the hacker to reverse engineer.

3.3.1. Composite functions in code obfuscation.

One of the classifications of obfuscation transformations is data transformations [16]. They affect the way data structures are used by a program and the way a data is stored in the memory. For instance, a local variable in the source program can be changed to be global in the obfuscated program. In this section, we discuss on how we can manipulate the data read and interpreted. For example, position of an array element, method arguments and constants in a program can all be represented as a complex mathematical function and confuse the reverse engineer.

a) Array index transformation using composite functions: Let $I = f(i) = 2 * i + 3$, be a function representing the new value of *I*. Let $J = g(I) = f((I - 3)/2)$ be a function representing the new position of the *i'th* element in the reordered array. Therefore, member variable *i* can be shown as a composite function of *f (g (i))*. Program segment in Table 2 shows the use of composite functions to hide the position indicator element of an array. Results tabulated show that the value of *i* remains same before and after obfuscation.

Table 2. Variations in *i, I = f (i), J = g (I)*

Before

```
int i = 1;
while (i <1000) {
      A[i];
      i++;
}
```

After

```
int I = f (i);
while (I < f (1000)) {
      A [g (I)];
      I = I+2;

}
```

i	*I = f(i) = 2* i + 3*	*J = g(i) = f((I-3)/2)*
1	5	1
2	7	2
3	9	3
4	11	4

Adjustments to the value of the member variable *I* will need to be made based on the functions *f* and *g*. For the example, in Table 2, the value *I* is incremented by *2* in the obfuscated program unlike in the original one.

b) Method Argument transformation using composite functions: The use of composite functions can be extended to hide method

arguments. Let *f(i)=i+1* be a function representing the method argument of a method *B*. Let *g(i)=f (i)+2* be a function representing the return value of the method *B*. The composite of functions *f* and *g* can be used in a program segment as in Table 3. Let *A = f(i) = i+ 1, B = g(i) = f(i) + 2* and *j = B(i)* after obfuscation, *j = B(i) + 1* before obfuscation. The values of *A, B* and *j* is shown in Table 3 for *j = 0, 1, 2, 3, 4*. The grey column represents the values before obfuscation and the white column represents values after obfuscation.

The results in Table 3 show that although the intermediate values using composite functions are different, the final value '*j*' remains the same.

c) Hiding constants using composite functions

Composite functions can also be used to hide a constant value in a program segment. Generally, constants are either strings or integers. An integer constant *y* can be represented as (*Y =a*n + y*) where *n* is a composite of two numbers whose sum is a prime. The operation *y = Y mod n* will de-obfuscate the value *Y*.

Table 3. Variations in *i, f (i), g (i) and j*

Before

```
int j =0;
for (int i = 0; i< 5; i++) {
     j = B (i) + 1;
 }
 int method B (int a) {

    int b = a + 2;
    return (b);
}
```

After

```
int j =0;
for (int i = 0; i< 5; i++){
     j =B (i);
}
int method B (f (i)) {
     return (g (i));
}
```

i	0	0	1	1	2	2	3	3	4	4
A	-	1	-	2	-	3	-	4	-	5
B	2	3	3	4	4	5	5	6	6	7
j	3	3	4	4	5	5	6	6	7	7

Program segment in Table 4 shows this technique in action for hiding integer constant *y = 2* for a generic value of *a = 1, 2, 3...etc*. The iterations shown in Table 4 lists values of *n, Y* and *y* when *y = 2* and *a = 1*. Note that the value of '*a*' is hidden inside '*y*', which is the first argument of the function '*F*' in the code segment shown in Table 4.

Table 4. Iteration for representing constant *y = 2, a =1*

Before

```
public static final
int y = 2;
int x = 2 * y;
System.out.println
   ("Value of x" + x);
```

After

```
public static final
int y = F (41 mod 23 , 2);
int x = 2 * y;
System.out.println
   ("Value of x" + x);

int F(int y, int count){

/* this array can be a dynamic
list of pairs of numbers whose
sum gives a prime */

int[][]  y_factors = new
int[2][count] ;

/* Assign values to y_factors
based on the rule that n1+n2
results in a prime number. */
y _factors = [2,3],[5,6];

for( int i = count ; i  >  0 ; i--)
{
   int y1 = (sum of elements in)
            y_factors[ i -1] ;
    /* Here y1 =
                y_factors[5+6],
                y_factors[2+3]
    */
    y = y mod y1;
}
 return y;
}
```

n_i	*Y*	*y = Y mod n*
2 + 3 = 5	5 + 2 = 7	7 mod 5
5 + 6 = 11	11 + 7 = 18	18 mod 11
11 + 12 = 23	23 + 18 = 41	41 mod 23

Here *p* represents the pair of numbers chosen. The primary rule is to choose two numbers n_1, n_2 such that $n_1 + n_2$ results in a prime number. This rule is needed to form the dynamic 2-dimensional array of factors of '*n*' as in program segment in Table 4.

To recover *y*, the adversary will have to calculate the function, *((41 mod 23) mod 11) mod 5)*. As in Table 4, to deobfuscate '*y*', the additive factors of the last two intermediate modulus are placed in a dynamically generated 2-dimensional array '*y_factors*'. The variable '*count*' represents the depth of the modulus applied to hide *y*. The first parameter in '*y_factors'* represents the pairs p_1 and p_2 (n_1) while the second parameter indicates the p_3 and p_4 (n_2) value. For each iteration of '*count*', modulus *y_factor* is applied

to '*y*' to ultimately get *((41 mod 23) mod 11) mod 5)*. In general, a hacker will not be able to deobfuscate the value of '*y*' without knowing the additive factors of '*n*'. As '*n*' is made larger, knowing its additive factors becomes hard as well. This technique produces larger obfuscated values as '*n*' is made larger by using additive factors whose sum produces a large prime number. Complexity of obfuscating integer constants using this technique can be increased by representing $Y = a*n + y$ as a composite of two functions $Y_1 = a*n_1 + y$ and $Y_2 = a*n_2$. For $Y = 2$, $a = 1$, $n_1 = 2$ and $n_2 = 3$, $Y_1 = 2*1+2$, $Y_2 = 3*1$ and $Y = Y1 + Y2 = 4 + 3 = 7$.

In the next section, we focus on using other mathematical techniques such as affine ciphers and discrete logs to implement obfuscation transformations.

3.3.2. Affine ciphers & discrete logarithms in code obfuscation.

a) Affine cipher technique to encode string literals: String literals in a program can be obfuscated into a cipher text based on affine cipher [22], [24], [25] technique. Here every alphabet in the string will be pushed forward by a definite number (obfuscation parameter) of alphabets, which is chosen at random. Each string is "encrypted" in the obfuscated program. To get the de-obfuscated value of the string literals, on the receiving end any string reference should be replaced by a call to a method that "decrypts" it. Program segment in Table 5 shows the use of affine ciphers to encode string constant.

Consider a string name = "*Caser*". Let the obfuscation parameter chosen at random be *5*. This means each letter in the string literal will be pushed forward by *5* letters. Therefore, encrypted value of string name is "*igykx*". To recover the value of name, the deobfuscation program, will call a function *f'* (where ' means Inverse) which would have the value of the obfuscation parameter *5* used to encrypt the message during obfuscation. Using this, the function *f'* will decrypt name by pushing each alphabet backward by *5* letters.

This technique can easily be compromised if an adversary hacks the obfuscation parameter. If obfuscation parameter can be hidden as composite of some numbers whose sum gives a prime number, then it is hard for the reverse-engineer to compromise the system in a cost effective manner. Hence, it is evident that affine cipher techniques [24], [25] in combination with composite numbers can be used to obfuscate a software program.

Table 5. Affine Cipher Obfuscating String Constants

Before	After

public static final
String name = "Caser";
System.out.println
("Before obfuscation" +
name);
	public static final
	String name =
	f (name, random_number);
	System.out.println
	("Before obfuscation" +
	name);
	String method f (String
	name, int i) {
	//routine to push the
	//string forward by
	//random number of
	//letters
	}

b) Discrete logarithms to pack words: Fundamentals of discrete logarithm [18], [19], [20] can be used to protect variables from dynamic analysis of memory references. Instead of separate words being used to store different variables, multiple variables can be packed into the same word so that the adversary is presented with a storm of events if he sets a break point on all references to a word during dynamic analysis. According to Euler's theorem for every *a* and *n* that are relatively prime, $a^{\varphi(n)} = 1 \bmod n$, an integer *z* can be expressed in the form $z = q + k\,\varphi(n)$, with $0 = q = \varphi(n)$. Where $\varphi(n)$ is called Euler's totient function and is defined as the number of positive integers less than *n* and relatively prime to *n*. Therefore by generalizing the Euler's theorem, an integer can be written as $y = g^x \bmod p$ (Note that unique discrete logarithms *mod m* is exist only to some base *g* if *g* is a primitive root of *m*). This is called discreet logarithms [18], [19], [20], [24]. Based on this theorem the value of a word which is treated as an unsigned integer mod a prime number is the value of a separate variable.

For example if a word that is assigned to *4* can be used to represent two variables, one variable containing $2^5 \pmod 7$ and the other variable containing $5^8 \pmod{19}$. We can express *4* as *32*

*mod 7 = 4*7 + 4* and *390625 mod 19 = 20559*19 + 4*.

Euler's theorem can also be used to obfuscate Boolean operators in a program segment. Boolean variables can be represented by Integer variables and be assigned many possible integer values to true and similarly for false. Example in Table 6 shows how a logical AND – OR can be obfuscated.

Let the obfuscation parameter be assigned a value *4* and the logical operation tied to this parameter is AND. We know that *4* can be represented as *32 mod 7* and *390625 mod 19*. If values of *x* and *y* maps to one of these two then AND condition will be satisfied. Similarly logical OR operation can be tied to an obfuscation parameter where either *x* or *y* should be one of *32 mod 7* or *390625 mod 19*. Table 6 explains this technique of AND-OR operation.

Here obfuscation parameters are some values specific to program segments and are infrequently available to the adversary. They are randomly chosen from a definite set of numbers, which map to a single value. The set of possible values for a variable should be large enough and be uniformly randomly distributed.

Table 6. Obfuscation of *AND–OR* Operations

Before	After

`if ((x AND y)) {`
`  total = total +1;`
`  System.out.println`
`  ("Before`
`  obfuscation" +`
`  name);`
`}`
	`if ((x is a`
	`  F (obfuscation_param)`
	`  AND y is a`
	`  F (obfuscation_param)) {`
	`  total = total + 1;`
	`  }`

Similarly, for logical OR we can use

Before	After

`if ((x OR y)) {`
`  total = total +1;`
`}`
	`if ((x is a F`
	`  (obfuscation_param)`
	`  OR y is a`
	`  F (obfuscation_param))`
	`  {`
	`  total = total + 1;`
	`  }`

This can be achieved by using discrete logs as discussed above. This application of Euler's theorem can be extended to present other operators like =, <, > and XOR operations in a program segments.

4. Conclusions

In this paper, we have presented the scope of using composite functions in combination with other mathematical techniques such as affine ciphers and discrete logs as means of obfuscating a software program especially for reading and interpreting data. We have introduced five new obfuscation techniques which are array index transformations, method argument transformation, hiding constants using composite functions, affine cipher technique to encode string literals and discrete logarithms to pack words. With these methodologies, we can hide constants, Boolean operators and method arguments effectively. We have also presented a comparison of *JHide* based on certain parameters with other Java obfuscation tool kits such as *JObfuscator, Sand Mark and JMangle*. We have compared them based on the number of algorithms supported, complexity of code, user interface and cost incurred to use them. Currently *JHide* provides only a show case of obfuscation algorithms and does not support any interface to measure their efficiency in terms of level of obfuscation achieved (Potency) and the maximum execution time/space that the obfuscated code adds to the application (Cost) [30]. We want to implement such an interface in our future work where the user would select the percentage of obfuscation needed and JHide obfuscation interface would automatically apply all the required algorithms to achieve that percentage level of obfuscation.

5. References

[1] M.R. Stytz, J. A. Whittaker, "Software Protection-Security's Last Stand", *IEEE Security and Privacy,* January/February 2003, pp. 95-98.

[2] G.McGraw, "Software Security", *IEEE Security &Privacy*, March/April 2004.

[3] C. Cowan, "Software Security for open source systems", *IEEE Security & Privacy*, February 2003

[4] M.R Stytz, "Considering Defense in Depth for Software Applications", *IEEE Security & Privacy*, February 2004.

[5] J.Whittaker, "Why Secure Applications are difficult to write", *IEEE Security & Privacy*, April 2003

[6] Business Software Alliance http://global.bsa.org/usa/press/newsreleases/2002-06-10.1129.phtml?CFID=4661&CFTOKEN=73044918

[7] C. Collberg, G. Myles and A.H. Work, "Sand Mark – A Tool for Software Protection Research", *IEEE Security & Privacy* July/August 2003.

[8] C.Collberg, C.Thomborson, "Watermarking, Tamper Proofing and Obfuscation – Tools for Software Protection**",** Technical Report, February 2000-03.

[9] P. Tyma, "Encryption, hashing, & obfuscation", *ZD Net* April 2003

[10] G. Naumovich, N. Memon, "Preventing Privacy, Reverse Engineering & Tampering", *Innovative Technology for Computer Professionals ,* July 2003

[11]D. Low, "Java Control Flow Obfuscation", *Thesis Report University of Auckland*, June 3 1998

[12] G.Wrobliwski, "General Method of Program Code Obfuscation", *PhD Dissertation, Wroclaw University of Technology, Institute of Engineering Cybernetics,* 2002

[13] DashO and Dot Obfuscator http://www.preemptive.com/

[14] JMangle, "The Java Class Mangle" http://www.elegant-software.com/software/jmangle/

[15]Jobfuscator http://download.com.com/3000-2417-10205637.html

[16] L Ertaul, S Venkatesh, "JHide –A tool kit for code obfuscation" , *The 8th IASTED International Conference on Software Engineering and Applications – SEA*,MIT Cambridge, MA – USA, Nov 2004 .

[17] D. Austin, "UBC Calculus Online Lecture Notes" *Department of Mathematics, University Of Columbia*.

[18] W. J. Hane,"Conquering calculus the easy road to understanding mathematics", *Peruses publishing*, 1998

[19]K Ming Teo, "Teaching mathematics and its applications", *International Journal of the IMA*, Volume 21 Issue 4, Dec 2002

[20] W.A.J. Kosmala, "Advanced Calculus a Friendly Approach", *Prentice Hall*, 1998

[21] D. Low, "Protecting Java Code via Code Obfuscation", *ACM Crossroads Student Magazine,* Spring 1998

[22] S. Katzenbeisser, "Information Hiding Techniques for Steganography and Digital Watermarking", *Boston Artech House*, 2000

[23] C. Collberg, C. Thomborson and D. Low, "Taxonomy of Obfuscation Transformations", *Technical Report #148,* July 1997

[24] L. Ertaul, "Cryptography and Data Security Lecture notes", *CSU-Hayward Department of Computer Science, http://www.mcs.csuhayward.edu/~lertaul/CS6520CourseWINTER2005.html*

[25] P. Garrett, "Making, Breaking Codes: An Introduction to Cryptology", *Prentice Hall*, Upper Saddle River, NJ, 2001

[26] L Badger, L Anna ,D Kilpatrick, B Matt, A Reisse and T Vleck , **"**Self-Protecting Mobile Agents obfuscation**",** *NAI Labs Report #01-036*, November 2001

[27] J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth., "Cryptographic security for mobile code", *IEEE Symposium on Security and Privacy*, May 2001, pp. 2-11.

[28] C. Collberg, J. Thomborson, and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", *IEEE International Conference on Computer Languages*, May 1998.

[29] C. Collberg, P. Clark, "Breaking abstractions and structuring data structures", *IEEE Computer Language ICCL'*98

[30] D. E. Bakken, A. A. Franz, T. J. Palmer, R. Paramesvaran, D. M. Blough, "Data Obfuscation: Providing Anonymity and Desensitization of Usable Data Sets", *IEEE Security & Privacy*, Vol:2, No:6, Nov/Dec, 2004.