International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:12, 2007

Array Data Transformation for Source Code Obfuscation

S. Praveen, and P. Sojan Lal

***Abstract*—**Obfuscation is a low cost software protection methodology to avoid reverse engineering and re engineering of applications. Source code obfuscation aims in obscuring the source code to hide the functionality of the codes. This paper proposes an Array data transformation in order to obfuscate the source code which uses arrays. The applications using the proposed data structures force the programmer to obscure the logic manually. It makes the developed obscured codes hard to reverse engineer and also protects the functionality of the codes.

***Keywords*—**Reverse Engineering, Source Code Obfuscation.

I. Introduction

THE first part of the translation is from Java source to Java Virtual Machine (JVM) machine code. Followed by, translation to real machine instruction in the browser on the user's machine. Since byte code retains almost all information of the source file, there are possibilities for reverse engineering [10] and reengineering. Reverse Engineering or Reengineering is a form of intellectual property theft which is illegal. The code obfuscation was a novel move for software protection and the intention is to hide the functionality of the codes, to limit the possibilities of reverse engineering or reengineering. The possibility of the execution of the obfuscated object code, has led to the popularity of obfuscation. The popular code obfuscation transformation techniques are (i) Layout transformation (making the code unreadable) (ii) Data transformation (obscuring data and data structures) (iii) Control transformation (obscuring the flow of execution) [2] [7] [8]. Source code obfuscation is achieved through source code transformations, Java bytecode obfuscation through bytecode transformations and binary obfuscation through binary rewriting [6]. Our focus is on source code obfuscation, aiming at obscuring the source code manually. It consists of techniques to target at making source code less comprehensible and automatically transform the programmer's source code into more complex, functionally equivalent source code.

A. The Format of a Class File

The Java class file stores all necessary data regarding the class. The main components of a class file are the Magic number, Version number, Constant pool, Access flags, Super class, Interfaces, Fields, Methods and Attributes[1][4]. JVM is a stack based machine. Each thread has JVM stack which stores frames. A frame is created each time a method is invoked and consists of an operand stack, an array of local variables, and a reference to the runtime constant pool of the current method. The constant pool is a table of structures representing various constants such as string constants, class and interface names and field names.

B. The Obfuscation Procedure

The class files are the input to the obfuscator [5] and the obfuscator transforms the byte code files into a complex format which is almost difficult to perceive. The obfuscator imposes a one level security by obfuscating the class file.

One of the proposals in this paper is applying the proposed data structures in the source code. It forces the developer to manually obfuscate the source code. The details of the data structures are maintained within the developing organization. The source code can again be obfuscated, by passing the compiled class file through the obfuscator. Hence, this methodology implements a second level security to the source code.

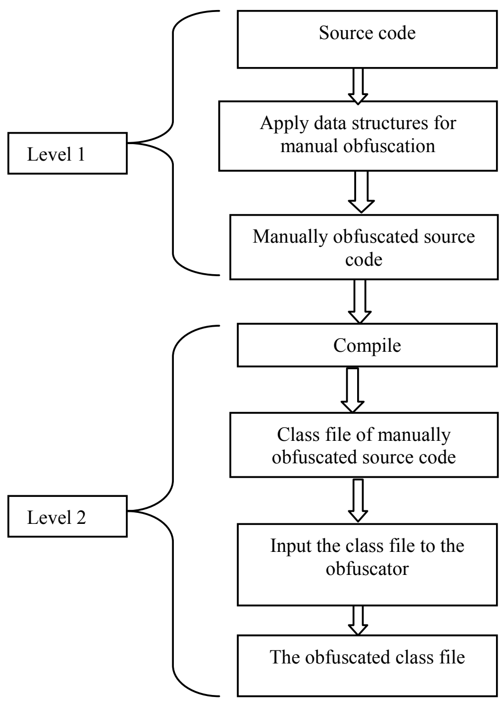

Fig. 1 Two levels obfuscation process

S. Praveen is with Federal Institute of Science and Technology, Angamaly, Kerala, India (e-mail: praveen_sivadas@yahoo.com).
P. Sojan Lal is with School of Computer Sciences, Mahatma Gandhi University, Kottayam, Kerala, India (e-mail: sojanlal@gmail.com).

The main threat to obfuscation is deobfuscation[3]. Deobfuscation is performed on the obfuscated class files and it could result in byte code of manually obfuscated source code which is hard to understand and reverse engineer.

II. The Existing Array Index Transformation Data Structure

In [9], Array index transformation using composite functions has been proposed.

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.

TABLE I
Array Index Transformation Table

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

The starting index of the array is stored within the program. The observation of the first two row values of I, clearly reveals that the difference of numbers is 2. It helps to predict the value of I for any i, without the help of the function f(i). The purpose of f(i) is not served fully. This is a drawback of the algorithm. Other drawbacks are for all the transformed arrays, the starting index is 5(for index i=1) with respect to the transformation function f(i). The elements are stored at places 5,7,9,11,… and the indices 0,1,2,3,4,6,8,10,… remain unassigned with respect to Table I, which leads to improper utilization of arrays.

The proposal is suggesting a data structure which performs the array index transformation, mainly considering proper storage of elements.

III. Array Index Transformation Data Structure - Proposal

Before proposing the Data Structure, some improvements have to be suggested for the Data Structure proposed in [9].

The Table I, proposed in [9], shows that the index 'i' with start value as 1. It can be enhanced with start value as 0.

TABLE II
Enhanced Array Index Transformation Table

i	I=f(i)=2*i+3	J=g(I)=f((I-3)/2)
0	3	0
1	5	1
2	7	2
3	9	3

Instead of f(i)=2* i + 3, the transformation function can be considered as f(i)=2 * i + 1,which possibly makes to utilize the unfilled array space during the second trace without much effort.

TABLE III
Enhanced Array Transformation Function

Trace 1			Trace 2	
i	I=f(i)=2*i+1	J=g(I)=f((I-1)/2)	j	I=2*j
0	1	0	0	0
1	3	1	1	2
2	5	2	2	4
3	7	3	3	6

For the first trace, the reordered array starts with indices 1,3,5,7,9… and the second trace with indices 0,2,4,6,8….to utilize the free spaces .

Let 'count' be the number of elements to be stored in the array.

The pseudo code for array storage as in Table III is as follows:

```
i=0,j=0,p=0;
while (p<count)
{  if(f(i)<array.length)
       {      a[f(i)] = args[p] ;
       i=i+1;
       }
       else
       {     a[m(j)]=args[p];
       j=j+1;
       }
       p=p+1;
 }

 int f(i)        //transformation function for first trace
{
          return (2 * i + 1);
 }
int m(j)      //transformation function for second trace
{
     return (2 * j);
 }
```

The array is filled in two traces. In trace1, the array positions with odd indices are filled, followed by positions with even indices in trace 2. Now, let us propose a data structure based on this procedure, to obscure the array elements. In this approach variable splitting is also applied to array elements.

The declaration procedure for a Decimal array of size 10 is using the statement Double [] a=new Double [10]. Instead of this one statement, our proposal is to use 3 separate member arrays of the same size. array1 is the obscured integer part storage array (Table IV), array2 is the obfuscator-shuffling array(Table V), array3 is the obscured fractional part storage array (Table VI). The obfuscator-shuffling array, array2 is applied for obscuring and relocating elements of array1 and array3. array1 and array3 are of Data type double and array2 of type Integer. The elements of array2 are shuffled during run-time. array1 stores the integer part and array3 stores the fractional part of the element after transformation.

The data transformation of array1 are performed using the expressions,

array1[f(i)]=array1[f(i)]+array2[f(i)]
array1[array2[i]]=array1[array2[i]]+array2[array2[i]].

The data transformation of array3 are performed using expressions,
array3[f(i)]=array3[f(i)]+array2[f(i)]
array3[array2[i]]=array3[array2[i]]+array2[array2[i]].

Let 'Ri' be the ith real number. Ii be the integer part of Ri

TABLE IV
ARRAY1 – OBSCURED INTEGER PART STORAGE ARRAY DATA STRUCTURE

I0	I1	I2	I3	I4	I5	I6	……	……	In
0	1	2	3	4	5	6	…..	…..	n

In Table IV, the assigning of integer part of elements start with indices 1, 3, 5,7,…. and later from indices 0,2,4,6,… so-on. The integer part of the real number is obscured and stored in this table.

TABLE IV
ARRAY2 – OBFUSCATOR SHUFFLING ARRAY DATA STRUCTURE

0	1	2	3	…..	…..	…..	….	….	n
0	1	2	3	4	5	6	…..	…	n

Table V is used for obscuring and shuffling elements of array1 and array3. Initially, the values of the array are from 0 to maximum index of array1. The elements of the array are shuffled. For shuffling, the details given are the start element, the length and the number of times of shuffling. The elements at the position of indices 'start' and 'start+len' will be swapped.

TABLE VI
ARRAY3 – OBSCURED FRACTIONAL STORAGE ARRAY DATA STRUCTURE

F0	F1	F2	F4	F5	F6	F7	….	….	Fn
0	1	2	3	4	5	6	…..	…	n

In Table VI, the assigning of fractional part of elements start with indices 1, 3, 5, 7,… and later from indices 0,2,4,6,.. so-on. The fractional part of the real number is obscured and stored in this table.

IV. ALGORITHM FOR ARRAY DATA TRANSFORMATION

a) Start

b) Shuffle the obfuscator shuffling array,array2

c) Read the real number say,Real_num .

d) Split the integer and fractional part

e) Store the integer part of Real_num, say int_Real_num in array1, with first element at position specified by array1[f(i)].

   Store the fractional part of the Real_num,say fract_Real_num in array3 ,with first element at position specified by array3[f(i)].

f) Transform the elements in array1 by,
   array1[f(i)]=array1[f(i)]+array2[f(i)].
   Transform the elements in array3 by,
   array3[f(i)]=array3[f(i)]+array2[f(i)].

g) Obscure the elements of array1 using,
   array1[array2[i]]=array1[array2[i]]+array2[array2[i]];
   Obscure the elements of array3 using,
   array3[array2[i]]=array3[array2[i]]+array2[array2[i]];

h) Relocating elements of array1
   array5[array2[i]]=array1[i]
   array1[i]=array5[i]

i) Relocating elements of array3
   array5[array2[i]]=array3[i]
   array3[i]=array5[i]

j) Obfuscating fractional part
   array3[i]=array3[i]+array3[i]

k) Stop

Let the execution be on arrays of say size 10.

Step 1 Start

Step2 Shuffling of obfuscator shuffling array

Enter start
0
Enter length
2
Enter Times
3
Shuffling.....
Shuffling.....
Shuffling.....
2
5
8
1
0
9
4
3
6
7

Step 3 Let the elements read to arrays are
2.3, -3.45, 4.6, 7.5, 3.6789, 5, 9.2

Step4
The integer part and fractional part of the numbers are split and stored in array1 and array3 at positions specified by array1[f(i)] and array3[f(i)] respectively

array1

0	5
1	2
2	9
3	-3
4	0
5	4
6	0
7	7
8	0
9	3

array2

0	2
1	5
2	8
3	1
4	0
5	9
6	4
7	3
8	6
9	7

array3

0	0
1	0.3
2	0.2
3	0.45
4	0
5	0.6
6	0
7	0.5
8	0
9	0.6789

Step 5 Transforming elements of array1 and array3
array1[f(i)]=array1[f(i)]+array2[f(i)].
array3[f(i)]=array3[f(i)]+array2[f(i)]

array1

0	7
1	7
2	17
3	-2
4	0
5	13
6	4
7	10
8	6
9	10

array2

0	2
1	5
2	8
3	1
4	0
5	9
6	4
7	3
8	6
9	7

array3

0	2
1	5.3
2	8.2
3	1.45
4	0
5	9.6
6	4
7	3.5
8	6
9	7.6789

Step 6 Obscuring elements of array1 and array3
array1[array2[i]]=array1[array2[i]]+array2[array2[i]]
array3[array2[i]]=array3[array2[i]]+array2[array2[i]]

array1

0	9
1	12
2	25
3	-1
4	0
5	22
6	8
7	13
8	12
9	17

array2

0	2
1	5
2	8
3	1
4	0
5	9
6	4
7	3
8	6
9	7

array3

0	4
1	10.3
2	16.2
3	2.45
4	0
5	18.6
6	8
7	6.5
8	12
9	14.6789

Step 7
Relocating elements of array1
array5[array2[i]]=array1[i]
array1[i]=array5[i]

array1

0	0
1	-1
2	9
3	13
4	8
5	12
6	12
7	17
8	25
9	22

array2

0	2
1	5
2	8
3	1
4	0
5	9
6	4
7	3
8	6
9	7

array5

0	0
1	-1
2	9
3	13
4	8
5	12
6	12
7	17
8	25
9	22

Step 8
Relocating elements of array3
array5[array2[i]]=array3[i]
array3[i]=array5[i]

array3

0	0
1	2.45
2	4
3	6.5
4	8
5	10.3
6	12
7	14.6789
8	16.2
9	18.6

array2

0	2
1	5
2	8
3	1
4	0
5	9
6	4
7	3
8	6
9	7

array5

0	0
1	2.45
2	4
3	6.5
4	8
5	10.3
6	12
7	14.6789
8	16.2
9	18.6

Step 9
Obfuscating fractional part
array3[i]=array3[i]+array3[i]

array3

0	0
1	4.9
2	8
3	13
4	16
5	20.6
6	24
7	29.3578
8	32.4
9	37.2

Step 10

The obscured integer and fractional parts for the given input data are as follows,

array1

0	0
1	-1
2	9
3	13
4	8
5	12
6	12
7	17
8	25
9	22

array3

0	0
1	4.9
2	8
3	13
4	16
5	20.6
6	24
7	29.3578
8	32.4
9	37.2

The input elements are hidden and obscured and make it difficult to identify them. Deobfuscation is the process of retrieving the input elements from the obscured elements.

Suppose the program is to find the biggest of numbers 2.3, -3.45, 4.6, 7.5, 3.6789, 5, 9.2. The integer part of the elements will be transformed, hidden and stored in array1 (Step 10). The fractional part of the elements are transformed, hidden and stored in array3 (Step 10). Hence, to write the logic of the biggest on the given numbers using the proposed data structures, the deobfuscation process has to be carried out on numbers, to retrieve the original values. The deobfuscation code has to be included along with the obscured codes to confuse the reverse engineer.

V. Data Structure Forcing to Obscure the Logic

Let the algorithm be to find the biggest of numbers 2.3, -3.45, 4.6, 7.5, 3.6789, 5, 9.2 using the proposed data strucure.

Obscured algorithm

1. Read numbers and store obscured numbers in array1 and array3 as in Step 10
2. Deobfuscate to get values of array1 and array3 as in Step 4 (Code becomes obscured)
3. Find the biggest element in array1.
4. If the biggest element does not repeat, say at index ‘j’
   a. Find the corresponding fractional part from array3 for index ‘j’
   b. Find the sum of elements of array1 and array3 for index ‘j’. Let the sum be ‘biggest’

   else
   c. Consider all the fractional parts in array3 for the biggest repeated elements. From array3, choose the biggest fraction say at index ‘k’.
   d. Find the sum of biggest element of array1 and element of array3 for index ‘k’. Let the sum be ‘biggest’
5. Return ‘biggest’

VI. Conclusion

The proposed data structure performs data transformation through variable splitting on array elements. The shuffling process tries to complicate the data storage process and makes reverse engineering hard. The execution speed can be compromised considering the cost to avoid malicious thefts. The obfuscation process in most of the cases results in lengthy programs and possibility of optimization can be investigated on the codes.

References

[1] Markus Dahm, ‘Byte Code Engineering with the BCEL API’ Technical Report B-17-98, April 3, 2001.
[2] Arjan de Roo, Leon van den Oord, ‘Stealthy obfuscation techniques: misleading the pirates’, Department of Computer Science, University of Twente Enschede, The Netherlands.
[3] Sharath.K.Udupa.Saumya K.Debray,Matias Madou, ‘Deobfuscation-Reverse Engineering obfuscated Code’, Proceedings of the 12th Working Conference on Reverse Engineering (WCRE’05).
[4] Christian Collberg, Ginger Myles, Michael Stepp, ‘An Empirical Study of Java Bytecode Programs’,Department of Computer Science, University of Arizona.
[5] Christian Collberg Clark Thomborson Douglas Low, ‘Breaking Abstractions and Unstructuring Data Structures’,Department of Computer Science,The University of Auckland.
[6] Madou, M.; Anckaert, B.; De Bus,De Bosschere, K.; Cappaert, J.; Preneel, B.;‘On the Effectiveness of Source Code Transformations for Binary Obfuscation’, Proc. of the International Conference on Software Engineering Research and Practice (SERP06), June. 2006.
[7] Christian Collberg, Clark Thomborson, and Douglas Low, ‘A Taxonomy of obfuscating Transformations’, Report 148, Department of Computer Science, University of Auckland, July 1997.
[8] C. Collberg and C. Thomborson, ‘Watermarking,Tamper-proofing, and obfuscation – tools for software protection’, IEEE Transactions on Software Engineering, Vol. 28,pp. 735-746, August 2002.
[9] L. Ertaul, S. Venkatesh, ‘Novel Obfuscation Algorithms for Software Security’, Proceedings of the 2005 International Conference on Software Engineering Research and Practice, SERP’05, June, Las Vegas.
[10] Ira D. Baxter,Michael Mehlich, ‘Reverse Engineering is Reverse Forward Engineering’, Proceedings of Fourth Working Conference on Reverse Engineering, 1997.