Changing Data Type Method of Data Obfuscation on Java Software

Hsiang-Yang Chen[1,3] Ting-Wei Hou[1]
Department of Engineering Science, National Cheng Kung University, Tainan, R.O.C.[1]
Department of Information Management, Hsing Kuo University of Management, Tainan, R.O.C.[3]
i14248@mail.hku.edu.tw hou@nc.es.ncku.edu.tw

***Abstract**- A defense against reverse engineering is obfuscation, a process that renders software unintelligible but still functional. Our goal is to change data type of variables for hiding original meaning to prevent attacked, especial for integer variables. We designed "Changing Data Type Obfuscation" method by changing the data type of variables, from long-term to short-term or short-term to long-term, to protect important variables of program code. We illustrate the concept and techniques of "Changing Data Type" method.*

***KEYWORD*：** reverse engineering, data obfuscation, Java

1. Introduction

Since the Java decompiler appeared [1], the threat of reverse engineering becomes worth-noticing. The Java language was designed to compile into a platform independent bytecode format. Much of the information contained in the source code remains in the bytecode, which means that decompilation is easier than with traditional native code. Today, it is not a secret that Java programs can be easily decompiled and reverse engineered from Java bytecode to Java source code [2].

A defense against reverse engineering is obfuscation. Obfuscation is a process that it keeps the program's semantic but make the program difficult to decompile. The design of obfuscation is to prevent from the theft of intellectual property by making it unable to derive usable source code from bytecode. Obfuscating transformations can be applied automatically to a program by a tool called an obfuscator. Using obfuscator to protect intellectual property for Java commercial software is very important. Obfuscation has become a critical to commercial software licensing.

In figure 1,the types of obfuscation techniques are as follows [3] [4]:

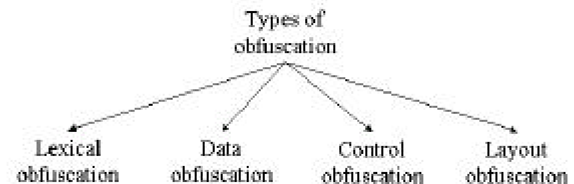

Figure 1. Four types of obfuscations.

Lexical obfuscation modifies the lexical structure of the program. Typically, they do nothing more then split identifiers. All meaningful symbolic information of a Java program, such as classes, fields, and method names are replaced with meaningless information, such as Crema[5] Java obfuscator.

Data obfuscation modifies the program data fields. For example, it is possible to replace an integer variable in a program with two integers. **Data aggregation** obfuscations alter how data is grouped together. For example, a two-dimensional array can be converted into a one-dimensional array and vice-versa. **Data ordering** obfuscations change how data is ordered. For example, an array used to store a list of integers usually has the ith element in the list at position i in the array. Instead, we could use a function $f(i)$ to determine the position of the ith element in the list.[6]

Control obfuscation makes thieves difficult to understand the control flow in individual program functions [7][8]. One example, opaque predicates, uses conditional- instructions whose predicates always evaluate true or false. The branch of such a condition that is always taken will contain meaning code, while the other branch will contain arbitrary code.

Layout obfuscation involves obscuring the logic inherent in splitting a program into procedures. One approach is to perform in-line expansion of a procedure in all places where the procedure is called.

Our study is focus on data obfuscation technique. Section 2 describes the method of *Changing Data Type Obfuscation*. Section 3 describes the design approach. Section 4 describes the techniques of spitting data types. Section 5 describes the discussion. Section 6 is the

conclusion of this paper. Section 7 is the future work of this study.

2. Design Methods

Figure 2 displays the concept: both the original program and the policy feed into a transformation procedure that generates the obfuscated program. After some period of time and expended effort, an attacker can gain understanding of OP. It has been postulated that the program can run safely for a limited time [9].

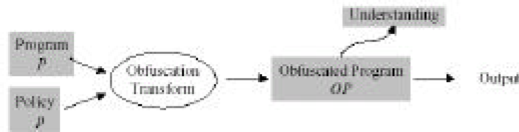

Figure 2. Obfuscation Transformation

We designed changing the data type of variables from long-term to short-term, as figure 3. The data length of long variable is 8 bytes. It can be splitting into 2 integer variables. It was a straightforward thinking that long-term variable could be splitting short-term variables more and more, such as 4 short variables, 8 byte variables. The techniques of splitting techniques introduces in section 3. We want to make the data obfuscation complicated. So we thing can replace the variable into an array, such "byte b[8];". Because human usually see an array store some data of same type together, it does not store a whole data value. Even, variable can be obfuscated into different type of numeric, such as String.

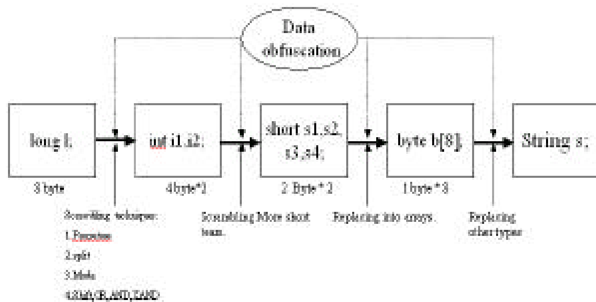

Figure 3. Long data-items obfuscate short data-items

Figure 4 is the reverse method of figure 3. We can split long-term data type variables into short-term data type variables. And we can reverse the method to extend the short-term data type variables into long-term data type variables directly. But this extending method will make the memory of obfuscated program larger then an unobfuscated program. This is the main defect of this method.

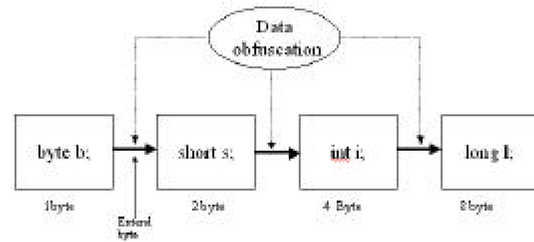

Figure 4. Short data items obfuscate long data items

3. Design Approach.

Figure 5 shows the flow of changing data type obfuscator. The approach is following:

1. Parse program: Parse the unobfuscated program to find all tokens of the program.
2. Search all variables of the program: Search and keep all variables of tokens in the program.
3. Choice variables: User choice which variables are important to obfuscate.
4. Choice splitting or extending: User can choice using splitting or extending method to obfuscate variables, these two method that we describe in section 2.
5. Choice splitting techniques: If user choice splitting method in step 4, user must choice splitting techniques.
6. Obfuscate variables: Final step is to obfuscate variables in the program.
7. End: The process is end.

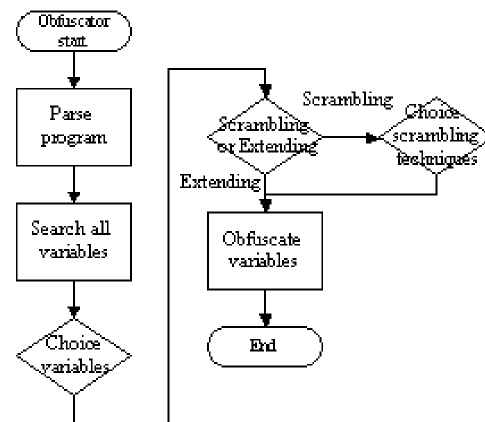

Figure 5. The flow of Changing Data Type Obfuscation

4. Splitting Techniques.

We design five kinds of splitting techniques for obfuscation [10].

1. **Parse tree.** A long-term variable store as parse

tree, which using short–term variables. As Figure 6.

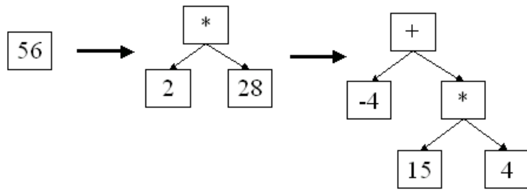

Figure 6. The splitting technique is parse tree.

2. **Permutations ordered lists.** We can obfuscate integers into permutations. The obfuscation parameters can be used to control the size of the set of elements, a permutation/mapping that corresponds to incrementing, as Figure 7.

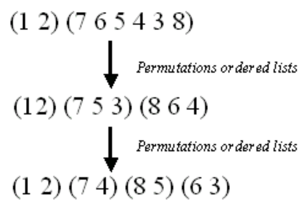

Figure 7. Permutaions ordered lists.

3. **Using module method.** We can use module method to splitting variable, as Figure 8. The prime is a prime number in Figure 8.

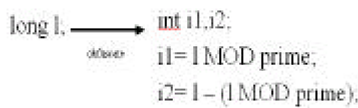

Figure 8. Using module method.

4. **Using Boolean operator.** We can use Boolean operator to splitting variable, such as NOT, OR, AND, XOR etc. For example:

$11110000_2 = 11110000_2$ AND 11110000_2;

5. **Restructure array.** We can split an array into several sub-arrays, merge two or more arrays into one array, fold an array (increasing the number of dimensions), or flatten an array (decreasing the number of dimensions) [11].

```
(1) int A[10];
(2) A[i] = ···;
      ···
      ···
(3) int B[10],C[20];
(4) B[i] = ···;
(5) C[i] = ···;
      ···
(6) int D[10];
(7) for(i=0;i<=8;i++)
      D[i]=2*D[i+1];
      ···
      ···
      ···
(8) int E[3,3];
(9) for(i=0;i<=2;i++)
      for(j=0;i<=2;i++)
        swap(E[i,j], E[j,i]);
```

```
(1') int A1[5],A2[5];
(2') if ((i%2)==0) A1[i/2]=···
     else A2[i/2]=···;
      ···
(3') int BC[30];
(4') BC[3*i] = ···;
(5') BC[i/2*3+1+i%2] = ···;
      ···
(6') int D1[2,5];
(7') for(j=0;j<=1;j++)
       for(k=0;k<=4;k++)
         if (k==4)
           D1[j,k]=2*D1[j+1,0];
         else
           D1[j,k]=2*D1[j,k+1];
      ···
(8') int E1[9];
(9') for(i=0;i<=8;i++)
       swap(E[i], E[3*(i%3)+i/3]);
```

Figure 9. Restructure array [11].

5. Discussion

The *Changing Data Type Obfuscation* can use to protect very important data, such as pay-money. Figure 9 show that bob tamper the pay money form 0.05$ to 0.01$. It can use our designed obfuscation to protect the money variable. The variable of money is splitting into an array of the short-term variables. Bob tamper an array of the short-term variable is hardly, he is very easy to makes mistake. So the obfuscation method can use in Tamper-proofing techniques, too.

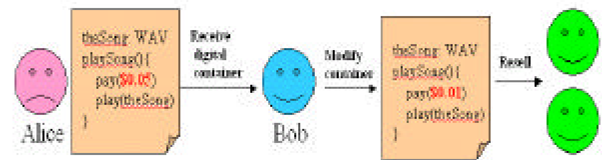

Figure 9. Bob tamper the value of pay money.

Lexical obfuscation makes meaningful symbolic information of a Java program, such as classes, fields, and method names are replaced with meaningless information. For example, a field name PayMoney may simply be replaced with p1. If our designed obfuscation combined lexical obfuscation, it will protect important data field better, as Figure 10.

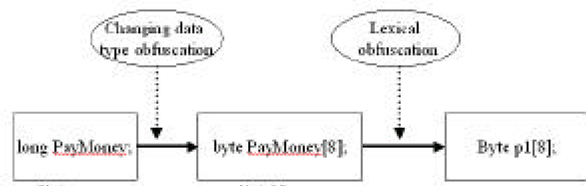

Figure 10. Changing Data Type Obfuscation combined Lexical obfuscation.

6. Conclusion

The task of making reverse engineering difficult is not easy. We designed "Changing Data Type Obfuscation" method by changing the data type of variables, from long-term to short-term or short-term to long-term, to protect important variables of program

code. We illustrate the concept and techniques of "Changing Data Type" method. Our designed obfuscation is a very simple method of obfuscator. But it can let attacker understand hardly from the obfuscated program.

Our designed obfuscation can combine other obfuscation techniques, such as Lexical obfuscation, other Data obfuscations, Control obfuscations, etc. It will protect software very better.

5. Future Work

We will increase the splitting techniques in this study. We want to change numeric data type to other data type, as figure 3, such as "long variables transfer into String variables".

Beside, for protecting obfuscated programs from being de-obfuscated to original source code, we must improve to combine other obfuscations, such as other Data obfuscations, Control obfuscations, etc.

Acknowledgements

The project is supported by NSC under project NSC (93-2213-E-006-105-).

Reference

[1] Hans Peter van Vliet. "Mocha - The Java decompiler", http://wkweb4.cableinet.co.uk/jinja/mocha.html, January 1996.

[2] WingSoft Company. "JavaDis - The Java Decompiler", http://www.wingsoft.com/wingdis.ml, March 1997.

[3] Christian Collberg, Clark Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation -Tools for Software Protection", IEEE Transactions on Software Engineering vol.28, no.8, August 2002, pp.735-746

[4] Gleb Naumovicb Nasir Memom, "Preventing Piracy, Reverse Engineering, and Tampering", IEEE Computer Society, 2003, pp.64-71

[5] Hanpeter van Vliet, "Crema: the Java obfuscator", http://www.brouhaha.com/~eric/computers/mocha.html, 1996

[6] Douglas Low, Protecting Java Code Via Code Obfuscation , ACM Crossroads, Spring 1998

[7] Christian Collberg, Clark Thomborson, Douglas Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs", In Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, San Diego, California, United States, 1998, pp.184-196

[8] Douglas Low, "Java Control Flow Obfuscation", Master's Thesis, Department of Computer Science, University of Auckland, New Zealand, June 1998

[9] F. Hohl. "Time limited blackbox security: Protecting mobile agents from malicious hosts", In Mobile Agents and Security, 1419 in LNCS. Springer-Verlag, 1998, pp. 92-113.

[10] Lee Badger,Larry D'Anna,Doug Kilpatrick,Brian Matt,Andrew Reisse,Tom Van Vleck, "Self-Protecting Mobile Agents Obfuscation Techniques Evaluation Report", www.networkassociates.com, November 30, 20011

[11] Collberg, Thomborson, Low, "Breaking Abstractions and Unstructuring Data Structures", International Conference on Computer Languages, 1998