Proceedings of the IASTED International Conference
SOFTWARE ENGINEERING
February 12-14, 2008, Innsbruck, Austria
ISBN Hardcopy: 978-0-88986-715-4 / CD: 978-0-88986-716-1

INTRODUCING DYNAMIC NAME RESOLUTION MECHANISM FOR OBFUSCATING SYSTEM-DEFINED NAMES IN PROGRAMS

Haruaki Tamada† Masahide Nakamura‡ Akito Monden† Ken-ichi Matsumoto†

†Graduate School of Information and Science,
Nara Institute of Science and Technology,
Ikoma-shi, Nara, Japan,
email: {harua-t, akito-m, matumoto}@is.naist.jp

‡Graduate School of Engineering,
Kobe University,
Kobe-shi, Hyogo, Japan,
email: masa-n@cs.kobe-u.ac.jp

ABSTRACT

Name obfuscation is a software protection technique, which renames identifiers in a given program, to protect the program from illegal cracking. The conventional methods replace names appearing in the declaration part with the meaningless ones. Therefore, the methods cannot be used to obfuscate names declared in system libraries, since changing such system-defined names significantly deteriorates the program portability. This paper presents a new name obfuscation method, which can hide appearance of the system-defined names. In the proposed method, the system-defined names are statically encrypted, and the original names are resolved during run time using the reflection. An experimental evaluation on the Java platform showed that the run-time overhead for the obfuscated program was 1.74 times larger than the one for the original.

KEY WORDS

copyright issue, obfuscation, reflection, program transformation

1 Introduction

A lot of incidents of *software cracking* have been reported in the whole world. From the viewpoint of protecting intellectual properties, the software cracking gives serious damages to the software industries.

The *program obfuscation* [1, 3, 12] is a technique to protect the program from the cracking. Intuitively, the obfuscation is to convert a given program p to a functionally equivalent one p' that is quite difficult to be analyzed. Many obfuscation methods have been proposed so far to obfuscate different aspects of the program, e.g., control flows, data flows, program layouts, names, etc. In this paper, we especially focus on the *name obfuscation*, which *hides* meaningful names in the program.

The *names* in a program (i.e., *identifiers*) are typical clues for the cracking, since a name usually characterizes a feature of the program [2]. For example, a program module that authenticates users may have a method named `authenticate`. An adversary who wants to nullify the authentication would first search the program by the string `authenticate`. Then, he or she would start detailed analysis around the matched portion. The goal of the name obfuscation is to hinder the adversary from analyzing and understanding the program based on the names.

The previous name obfuscation methods [1, 12] *statically* modify names in the program. That is, the methods replace every name appearing in the declaration part with a meaningless one, which can hide any *user-defined names*. However, the previous methods cannot obfuscate the use of *system-defined names* (e.g., `System.out.println()` of Java), which are usually declared in system or third party's libraries. This is because changing such system-defined names significantly deteriorates the program portability.

In this paper, we propose a new method that can hide the use of any system-defined names in an object-oriented program. Our key idea is to introduce a mechanism called *dynamic name resolution* (DNR, for short). Specifically, we first encrypt system-defined names in a program. Then, during runtime, for every reference of the encrypted name, the DNR first decrypts it to the original name, then resolves the name and execute an appropriate action using the reflection mechanism. We propose the DNR that can resolve system-defined names appearing as; (a) references of classes, (b) invocation of methods, (c) references and assignments of the field variables.

We have implemented the proposed method for the Java platform, and conducted a case study. It was shown that the run-time overhead for the obfuscated program was 1.74 times larger than the one for the original.

2 Preliminary

2.1 Program Obfuscation

This section formulates the notion of program obfuscation. We start with the definition of the program understanding, since the obfuscation prevents crackers from understanding the program.

Definition 1 (Program understanding) Let p be a given program, and X be a given a set of information included in p. When a user can extract X from p by a certain method, then we define that the user has understood p about X. For this, we denote a cost of the understanding as $cost(p, X)$ in an abstract manner.

```
 1: import javax.swing.*;
 2: import javax.swing.event.*;
 3: public class PlainWebBrowser extends JFrame
 4:                              implements HyperlinkListener{
 5:   private JEditorPane renderArea = new JEditorPane();
 6:   public PlainWebBrowser(java.net.URL location){
 7:     setLayout(new java.awt.BorderLayout());
 8:     add(new JScrollPane(renderArea), java.awt.BorderLayout.CENTER);
 9:     renderArea.addHyperlinkListener(this);
10:     renderArea.setEditable(false);
11:     setDefaultCloseOperation(DISPOSE_ON_CLOSE);
12:     setSize(400, 600);
13:     setVisible(true);
14:     setPage(location);
15:   }
16:   public void hyperlinkUpdate(HyperlinkEvent event){
17:     if(event.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
18:       setPage(event.getURL());
19:   }
20:   private void setPage(java.net.URL newLocation){
21:     try{ renderArea.setPage(newLocation); }
22:     catch(Exception exception){
23:       JOptionPane.showMessageDialog(this, exception.getMessage());
24:     }
25:   }
26:   public static void main(String[] args) throws Exception{
27:     new PlainWebBrowser(new java.net.URL(args[0]));
28:   }
29: }
```

Figure 1. Sample program (a simple web browser)

```
 1: import javax.swing.*;
 2: import javax.swing.event.*;
 3: public class a extends JFrame
 4:                implements HyperlinkListener{
 5:   private JEditorPane b = new JEditorPane();
 6:   public a(java.net.URL c){
 7:     setLayout(new java.awt.BorderLayout());
 8:     add(new JScrollPane(b), java.awt.BorderLayout.CENTER);
 9:     renderArea.addHyperlinkListener(this);
10:     renderArea.setEditable(false);
11:     setDefaultCloseOperation(DISPOSE_ON_CLOSE);
12:     setSize(400, 600);
13:     setVisible(true);
14:     c(d);
15:   }
16:   public void hyperlinkUpdate(HyperlinkEvent e){
17:     if(e.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
18:       c(e.getURL());
19:   }
20:   private void c(java.net.URL f){
21:     try{ b.setPage(f); }
22:     catch(Exception g){
23:       JOptionPane.showMessageDialog(this, g.getMessage());
24:     }
25:   }
26:   public static void main(String[] h) throws Exception{
27:     new a(new java.net.URL(h[0]));
28:   }
29: }
```

Figure 2. Obfuscated program by the conventional method (Fig.1)

The cost would be characterized by, for example, the time taken for the analysis, efforts, the necessary knowledge, devices, etc. Then, we give a general definition of the program obfuscation.

Definition 2 (Program obfuscation) Let p be a given program, X be a given a set of information of p, and $IO_p : I \rightarrow O$ be a input/output mapping of p. Let I be the all of input set, and O be the all of output set. Then, the *obfuscation of p with respect to X* is to translate p into p' with a certain method T (i.e., $p' = T(p)$), such that

Condition 1 $IO_p = IO_{p'}$

Condition 2 $cost(p, X) < cost(p', X)$

Condition 1 means to keep input/output mapping before and after the obfuscation. This means that the obfuscation must preserve the external specification of target program. Condition 2 means that understanding p' is significantly more difficult than understanding p.

2.2 Name Obfuscation

A name obfuscation of a program replaces each name (i.e., identifier) in the program with another, to hide any information reasoned from the name. Note that changing names provides no effect for the program execution, since names in a program are just identifiers for the computer.

Definition 3 (Name obfuscation) Let p be a given program, U_p be a set of all names appeared in p, and $N_p (\subset U_p)$ be a set of names, which are target of the obfuscation. A name obfuscation of p is to change each name $n \in N_p$ in p to other name $n' (= T(n))$ and to obtain a obfuscated program p', where T is one-to-one mapping ($T : N_p \rightarrow N_{p'} (N_{p'} \subset U_{p'})$)

If p is an object-oriented program, a name appears as a class name, a method name, a field name, or a local variable name. Also, every name appears in its *definition part* (declaration) and its *use part* (reference).

2.3 Previous Name Obfuscation

The previous name obfuscation methods [1, 3] statically change names that are declared in the definition part into other strings.

Previous Name Obfuscation Procedure

Input: Program p, and a set of names N_p

Output: Obfuscated program p'

Procedure: For each name $n \in N_p$, operating following steps. Let p' be a resultant program.

- **Step 1:** For each name $n \in N_p$, replace n in definition part of p to other name n'.
- **Step 2:** Replace n in use part of p to n' obtained at Step 1.

As for the input N_p, the previous method accepts any names which are defined by the user (developer of p). For instance, let us obfuscate a program shown in in Fig. 1 with the conventional method. The obfuscated example is shown in Fig. 2. In this example, we replaced the names as follows:

```
T(PlainWebBrowswer) = a;
T(renderArea) = b;
T(setPage) = c;
T(location) = d;
T(renderArea) = b;
T(event) = e;
T(newLocation) = f;
T(exception) = g;
T(args) = h;
```

The previous method is easy to be implemented, and it has little performance degradation. Thus, it is supported by many obfuscation tools, including Dash-O[6] and

ZKM[13] for the Java language, and Dotfuscator[7] for .Net framework.

Note, however, that we could not obfuscate names such as `setLayout`, `setSize`, `javax.swing.event.HyperlinkListener`, `javax.swing.JFrame`, `java.awt.BorderLayout`, since those names are defined in Java SE[9] and should not be changed to maintain the portability. If we replace those names to other meaningless names, the program will fail to be compiled in another environment. Thus, the previous method cannot hide such system-defined names.

3 The Proposed Method

3.1 Key Idea

In order to obfuscate the use of system-defined names, we propose to introduce a specific mechanism, called *dynamic name resolution* (*DNR*, for short), in the program. Specifically, we first encrypt every system-defined name beforehand, while the DNR decrypts and resolves the name during runtime.

To achieve this, the proposed method extensively uses the *reflection* of the object-oriented language. In most programming languages, all names must be statically resolved in the compilation time. However, using the reflection, we can create and operate an object dynamically from a given *string literal* during runtime. Typically, the reflection is used in meta-programming, such as implementation of plug-in architecture and acquisition of runtime information of the program itself. The proposed method uses reflection for implementing the DNR.

3.2 Dynamic Name Resolution (DNR)

The dynamic name resolution (DNR) is to resolve a name from a given string literal at runtime. For each system-defined name in a program, we encrypt the name into a string literal beforehand. During runtime, for each appearance of the encrypted string literal, the DNR decrypts the string, and restores the original operation from the string using the reflection.

Note that we obfuscate the system-defined names appearing in the *use part* only, since the names in the declaration part must not be changed for the portability. Thus, each name appears as (a) a class name in the object instantiation, (b) a method name (with a class name) in the method invocation, or (c) a field name (with a class name) in the field reference/assignment.

Due to the limited space, we describe the procedure of the DNR for resolving the above (b) method name only. Let n_m be a system-defined method name, and n_c be a system-defined class name in which n_m is defined. Then, let n'_c and n'_m be encrypted names of n_c and n_m using a certain encryption method E (i.e., $n'_c = E(n_c)$, $n_m = E(n_m)$). Then, the procedure for resolving $n'_c.n'_m()$ is defined as the following routine *resolveMethod(n'_c, n'_m)*.

```
 1: import javax.lang.reflect.*;
 2: import javax.swing.*;
 3: import javax.swing.event.*;
 4: public class PlainWebBrowser extends JFrame
 5:                              implements HyperlinkListener{
 6:   private JEditorPane renderArea;
 7:   public PlainWebBrowser(java.net.URL location){
 8:     // resolveClass("javax.swing.JEditorPane");
 9:     Class c1 = Class.forName(decrypt("kbwb/txjoh/KFeupsQbof"));
10:     Object o1 = c1.newInstance();
11:
12:     // resolveField("PlainWebBrowser", "renderArea");
13:     Class c2 = Class.forName(decrypt("QmbjoXfcCspxtfs"));
14:     Field f2 = c2.getField(decrypt("sfoefsBsfb"));
15:     f2.set(this, o1);
16:
17:     // resolveClass("java.awt.BorderLayout")
18:     Class c3 = Class.forName(decrypt("kbwb/bxu/CpsefsMbzpvu"));
19:     Object o3 = c1.newInstance();
20:
21:     // resolveMethod("javax.swing.JFrame", "setLayout")
22:     Class c4 = Class.forName(decrypt("kbwb/txjoh/KGsbnf"));
23:     Method m4 = c4.getMethod(decrypt("tfuMbzpvu"));
24:     m4.invoke(this, new Object[]  o3, );
                 :
```

Figure 3. Proposed name obfuscation with dynamic name resolution

DNR resolveMethod(n'_c, n'_m)

Step 1: Decrypt n'_c and obtain the original name n_c.

Step 2: Using the reflection, obtain a class c whose name is n_c.

Step 3: Obtain a set M_c of methods defined in c with the reflection.

Step 4: Decrypt n'_m and obtain the original name n_m.

Step 5: From M_c, obtain a method m whose name is n_m.

Step 6: Invoke the method m.

Similarly, we can define the resolution procedure *resolveClass(n'_c)* for the (a) class name n_c, and *resolveField(n'_c, n'_f)* for the (c) field name.

3.3 A Name Obfuscation Method with DNR

The proposed obfuscation method is as follows. We assume that the target program p is completely tested and has no bugs. Also, we assume that N_p can contain the system-defined names.

Procedure of Proposed Name Obfuscation Method

Input: a program p, and a set of names N_p.

Output: an obfuscated program p'.

Procedure: Apply following steps to program p and let the resultant program be p'.

Step 1: Encrypt each name $n \in N_p$ with an some encryption method E, and let $n' = E(n)$.

Step 2: For each $n \in N_p$, modify the use part u_n of n in p, as follows:

(a) if n appears as a class name n_c, then replace u_n with *resolveClass(n'_c)*.

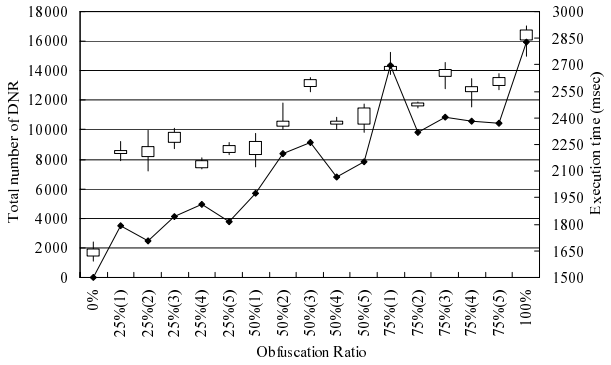

Figure 5. Execution time and the # of DNR

(b) if n appears as a method name n_m of class n_c, then replace u_n with *resolveMethod*(n'_m, n'_c).

(c) if n appears as a field name n_f of class n_c, then replace u_n with *resolveField*(n'_f, n'_c).

Figure 3 shows an example of the proposed obfuscation method in the Java language. In this figure, object instantiations (`javax.swing.JEditorPane`, `java.awt.BorderLayout`), field assignments (`renderArea`), and a method call (`setLayout`) in Fig. 1 are obfuscated. Just for simplicity, we chose the Caesar's cipher with key 1 as the encryption method E. The `decrypt` method in Fig. 3 represents a decrypt routine. We can see that all the system-define names (`javax.swing.JFrame`, `javax.swing.event.HyperlinkListener`, `renderArea`, `java.net.URL`, and `location`) appearing in the use part are successfully encrypted.

Let us look at the details in the last block in Fig.3, which implements a DNR routine *resolveMethod* for `setLayout` method in `javax.swing.JFrame` class. In the Java language, `java.lang.Class` reflects information of a class. A certain class can be obtained by giving class name string to `forName` method defined in `java.lang.Class`. Also, `java.lang.reflect` package have classes which reflect information of methods and fields. By using those classes, we can obtain the information of methods, fields and classes. In this example, encrypted string "kbwb/txjoh/KGsbnf" is decrypted to "java.swing.JFrame" and an instance of `java.lang.Class` is obtained by decrypted string. Then, "tfuMbzpvu" is also decrypted and "setLayout" method object is obtained. Finally, the `setLayout` method is invoked at line 24 by DNR.

4 Case Study

4.1 Tool and Obfuscation Sample

We have developed a tool of the proposed method on Java 5 platform[11]. The tool has been implemented with ASM 2.2.3, Java bytecode manipulation framework[5].

Table 1. Time taken for execution (msec)

		Average (msec)	Minimum (msec)	Maximum (msec)	File size (byte)
0%		1640.9	1589	1700	259,533
25%	1	2208.7	2160	2268	270,617
	2	2207.8	2103	2329	275,579
	3	2288.8	2228	2343	278,329
	4	2138.3	2111	2176	290,565
	5	2224.2	2191	2259	279,167
50%	1	2223.9	2123	2313	326,644
	2	2378.7	2336	2483	346,557
	3	2592.8	2548	2628	333,789
	4	2373.4	2336	2406	304,264
	5	2415.1	2316	2477	353,821
75%	1	2688	2647	2769	364,170
	2	2477.3	2457	2491	366,415
	3	2653.2	2567	2716	359,315
	4	2557.3	2459	2624	381,813
	5	2607.4	2558	2653	360,464
100%		2859.9	2748	2917	412,347

The main features of the tools are; (a) obfuscating directly Java class files (without source code), (b) support several encryption methods, and (c) introducing a helper class, `DynamicCaller`, for unifying *resolveClass*, *resolveMethod*, and *resolveField* DNR routines (its methods `newInstance()`, `invoke()`, `invokeStatic()`, `setField()`, `getField()`, `setStatic()`, and `getStatic()` corresponds the routines respectively).

Figure 4 shows a program obtained by applying the proposed method to the program in Fig. 1. In this example, the 56 bit DES [4] with the key "0xb097f88f0bbc73b5" is chosen. In the figure, we can see that all of method names and field names defined in Java SE are encrypted.

4.2 Obfuscation Overhead

We have conducted an experiment to evaluate the overhead of the proposed method. We applied an encryption method 56 bit DES. The experimental platform is Windows Vista Ultimate, Intel Core 2 1.86GHz, 2046M RAM, and Java 5 (jdk1.5.0_12). We evaluate a overhead of programs by the ratio of the number of obfuscated classes.

As for the target program, we chose SwingSet2 [10] of jdk1.5.0_12. SwingSet2 is a demo program of Java swing package and contains 138 classes. In the experiment, we randomly chose some classes of the 0%, 25%, 50%, 75% and 100% among all classes in `SwingSet2.jar`, and obfuscated them. Then, 5 sets of obfuscated classes were created in each percentage category (excluding 0% and 100%). For each set, we measured the execution time 10 times taken for each class to complete the loading.

The results are shown in Table 1 and Table 2. Table 1 shows the execution time of the average, the maximum, and the minimum of the obfuscated program, and the sum of class file size. Table 2 shows how many times the DNR is called by each routine. Also, the relation between execution overhead and the number of DNR calls is shown in Fig. 5. In Fig. 5, the execution overhead is represented

```
 1:public class PlainWebBrowser extends javax.swing.JFrame implements javax.swing.event.HyperlinkListener{
 2:  public Object renderArea;
 3:  public PlainWebBrowser(java.net.URL url){
 4:    Object o1 = DynamicCaller.newInstance(new Object[0], "a66297a550249aaf06684d0828c6f38b3510f3937c8973f6");
 5:    DynamicCaller.setField(this, o1, "2f1f9f0d17a287c1768c86f425205076", "cb4ae31a9d00d7f33f7fa28a1da39c7b");
 6:    Object o2 = DynamicCaller.newInstance(new Object[0], "322b5e6093539e2def5e1d91ed5673efed3193d71365bf79");
 7:    DynamicCaller.invoke(this, new Object[] { o2, }, "2f1f9f0d17a287c1768c86f425205076", "b090a73d53b053465c0b836b0de958bc");
 8:    Object o3 DynamicCaller.getField(this, "2f1f9f0d17a287c1768c86f425205076", "cb4ae31a9d00d7f33f7fa28a1da39c7b");
 9:    Object o4 = DynamicCaller.newInstance(new Object[] { o3, }, "a66297a550249aaffd45c2d99e870a280d8725555a756cfe";);
10:    DynamicCaller.invoke(this, new Object[] { o4, "Center", }, "2f1f9f0d17a287c1768c86f425205076", "edb49cf44850031c";);
11:    Object o5 = DynamicCaller.getField(this, "2f1f9f0d17a287c1768c86f425205076", "cb4ae31a9d00d7f33f7fa28a1da39c7b");
12:    DynamicCaller.invoke(o5, new Object[] { this, }, "a66297a550249aaf06684d0828c6f38b3510f3937c8973f6", "6e4e8cf0610b90c2d8608819a83b9e2ab1f3a611ef002d56");
13:    Object o6 = DynamicCaller.getField(this, "2f1f9f0d17a287c1768c86f425205076", "cb4ae31a9d00d7f33f7fa28a1da39c7b");
14:    DynamicCaller.invoke(o6, new Object[] { new Boolean(false), }, "a66297a550249aaf06684d0828c6f38b3510f3937c8973f6", "1896096f3a5b52ef56f561980741c259");
15:    DynamicCaller.invoke(this, new Object[] { new Integer(2), }, "2f1f9f0d17a287c1768c86f425205076",
16:                         "2b4e5099a9f92836c1390945ce922b208a5d15ef80bee749de209bef9d1612f6");
17:    DynamicCaller.invoke(this, new Object[] { new Integer(400), new Integer(600), }, "2f1f9f0d17a287c1768c86f425205076", "24209303b0ba19ac");
18:    DynamicCaller.invoke(this, new Object[] { new Boolean(true), }, "2f1f9f0d17a287c1768c86f425205076", "77d76f1b6ec32f97b697d240ae1c4ba2");
19:    DynamicCaller.invoke(this, new Object[] { url, }, "2f1f9f0d17a287c1768c86f425205076", "3c53afe08c4b5898");
20:  }
21:  public void hyperlinkUpdate(HyperlinkEvent e){
22:    Object o1 = DynamicCaller.invoke(e, new Object[0], "a66297a550249aaf00cfa6321b539a892fa0d06307277e054ba570d29ef8709ede209bef9d1612f6",
23:                                     "07fe9f80d9b52479a16a66be70119fd5");
24:    Object o2 = DynamicCaller.getStatic("a66297a550249aaf00cfa6321b539a892fa0d06307277e054ba570d29ef8709e8539bf260b43520811a22d97f25b3750",
25:                                        "0bade759e91db795888d06705c8563c4");
26:    if(o1 != o2){
27:      Object o1 = DynamicCaller.invoke(e, new Object[0], "a66297a550249aaf00cfa6321b539a892fa0d06307277e054ba570d29ef8709ede209bef9d1612f6", "68bd6e25d9fa5a04");
28:      DynamicCaller.invoke(this, new Object[] { o1, }, "2f1f9f0d17a287c1768c86f425205076", "3c53afe08c4b5898");
29:    }
30:  }
31:  public void setPage(java.net.URL location){
32:    try{
33:      Object o1 = DynamicCaller.getField(this, "2f1f9f0d17a287c1768c86f425205076", "cb4ae31a9d00d7f33f7fa28a1da39c7b");
34:      DynamicCaller.invoke(o1, new Object[] { location, }, "a66297a550249aaf06684d0828c6f38b3510f3937c8973f6", "3c53afe08c4b5898");
35:    } catch(Exception e){
36:      Object o2 = DynamicCaller.invoke(e, new Object[0], "a12f27ec29a113c9313d6ae12f76eddeb49254169990f045", "017186a6191bc657c585cbec0730d75b");
37:      DynamicCaller.invokeStatic(new Object[] { this, o1, }, "a66297a550249aafc52a232880c6bd2dd492a1e3e0e62d65",
38:                                 "dbf81f4d9244576c6d5a19eef1358ad970999eb9b50092c2");
39:    }
40:  }
41:  public static void main(String args[]) throws Exception{
42:    Object o1 = DynamicCaller.invokeStatic(new Object[] { args, new Integer(0), }, "a12f27ec29a113c941330416fa2e06ffdf75e527e791863a", "b647e427ff37e89e");
43:    Object o2 = DynamicCaller.newInstance(new Object[] { o1, }, "8ec26e55cf7e06cd3f56aa79fefb5c5a";);
44:    DynamicCaller.newInstance(new Object[] { o2, }, "2f1f9f0d17a287c1768c86f425205076");
45:  }
46:}
```

Figure 4. Obfuscated program by the proposed method (Fig. 1)

with the box plot in right side axis. The number of DNR is represented a line graph plotted in the left side axis.

In the ratio of 100% obfuscation, the sum of class file size increased 1.59 times and the average of execution time increased 1.74 times 0% obfuscation ratio compared with 0% obfuscation ratio. Thus, as the number of DNR increases, the execution time and the file size are increased.

Therefore, if the target program is sensitive about the execution time or the runtime platform has not enough disk space, we would resolve the priority of names from the viewpoint of security to refine the set of target names N_p. Also, we can combine use of the proposed obfuscation method for system-defined names, and the conventional name obfuscation method for user-defined names.

5 Related Works

Since the conventional name obfuscation method is easy to be implemented, most of the obfuscation tools support the method. Typical Java obfuscation tool is Dash-O provided by PreEmptive solutions[6]. Dash-O is also supports the name obfuscation method called overload induction[12]. The overload induction induces the method overloading maximally. For example, when `Foo#foo()` and `Foo#bar(int v)` is applied overload induction, then resultant methods are overloaded and become `a#a()` and `a#a(int a)`.

Other tool Allatori supports a unique name obfuscation method, which change all the local variable names in the Java bytecode into the same name[8]. Even though all of the local variable have the same name, the program is executable because it is referenced by the index, not the name. Additionally, the source code is obtained by decompilation, it will fail to be re-compile.

Unfortunately, those methods can deal with the user-defined name only. Since system provided names cannot be changed, class files still have plain names of system provided names.

Meanwhile, the previous name obfuscation methods and the proposed obfuscation method can be used complementarily. We can apply both obfuscation methods with sequence of previous obfuscation method, proposed obfuscation method.

6 Conclusion

In this paper, we proposed a new name obfuscation method to hide system-defined names using dynamic name resolution. Adopting an approach to resolving encrypted names at runtime with the reflection, the system-defined names can be obfuscated successfully. We also implemented a tool for obfuscating Java class files by proposed method. Then, we evaluated obfuscation overhead to SwingSet2 with the implemented tool.

Finally our future work is summarized as follows.

- Reduce the obfuscation overhead,
- Apply the anti-tampering for the DNR to prevent the dynamic analysis.

Table 2. # of dynamic resolution of obfuscated programs

		Field reference	Field assignment	Static field reference	Static field assignment	Object instantiation	Static method call	Method call	Total # of DNR
0%		0	0	0	0	0	0	0	0
25%	1	40	2	20	0	645	0	2826	3533
	2	311	6	34	0	422	78	1654	2505
	3	1626	51	47	14	251	61	2041	4091
	4	854	26	68	5	699	222	3074	4948
	5	16	4	100	1	700	45	2896	3762
50%	1	1456	68	101	6	698	226	3162	5717
	2	879	60	154	6	1354	548	5399	8400
	3	2002	96	91	14	1133	397	5388	9121
	4	1486	55	77	15	855	38	4296	6822
	5	1513	92	187	6	1023	603	4398	7822
75%	1	2933	132	165	20	1934	296	8893	14373
	2	2380	112	144	20	1206	540	5416	9818
	3	2159	85	135	15	1409	424	6606	10833
	4	1255	76	201	6	1646	581	6811	10576
	5	2297	103	151	15	1252	439	6218	10475
100%		3320	156	201	20	2073	628	9529	15927

Acknowledgement

The work was partially supported by the Comprehensive Development of e-Society Foundation Software program of the Ministry of Education, Culture, Sports, Science and Technology, and by the Ministry of Education, Culture, Sports, Science and Technology for Scientific Research (C), 19500056, 2007.

References

[1] Jien-Tsai Chan and Wuu Yang. Advanced obfuscation techniques for java bytecode. *Systems and Software*, 71, Issue 1-2:1–10, April 2004.

[2] B. D. Chaudhary and H. V. Sahasrabuddhe. Meaningfulness as a factor of program complexity. In *Proc. of the ACM 1980 annual conference*, pages 457–466, 1980.

[3] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *Proc. 1998 International Conference on Computer Languages*, pages 28–38, Washington, DC, USA, October 1998. IEEE Computer Society.

[4] NBS (National Bureau of Standards). Data encryption standard (des). Technical Report FIPS-Pub.46, National Bureau of Standards, U.S. Department of Commerce, Washington D.C., January 1977.

[5] ObjectWeb Consortium. ASM. http://asm.objectweb.org/.

[6] PreEmptive Solutions. DashO - the premier Java obfuscator and efficiency enhancing tool. http://www.agtech.co.jp/products/preemptive/dasho/index.html.

[7] PreEmptive Solutions. Dotfuscator —.net obfuscator, code protector, and pruner. http://www.preemptive.com/products/dotfuscator/index.html.

[8] Smardec. Allatori java obfuscator. http://www.allatori.com/.

[9] Sun Microsystems., Inc. Java standard edition. http://java.sun.com/se/.

[10] Sun Microsystems, Inc. SwingSet2 demo. http://www.JavaDesktop.org/.

[11] Haruaki Tamada. DonQuixote: Java obfuscation framework. http://donquixote.cafebabe.jp/.

[12] Paul M. Tyma. Method for renaming identifiers of a computer program. United States Patent 6,102,966, August 2000. Filed: Mar.20, 1998.

[13] Zelix Pty Ltd. Zelix Klass Master, 1997. http://www.zelix.com/klassmaster/index.html.