Java Obfuscation with a Theoretical Basis for Building Secure Mobile Agents

Yusuke Sakabe[1*], Masakazu Soshi[1], and Atsuko Miyaji[1]

School of Information Science, Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi, Nomi, Ishikawa 923-1292, JAPAN
{y-sakabe,soshi,miyaji}@jaist.ac.jp

Abstract. In this paper we propose novel techniques to obfuscate Java programs for developing secure mobile agent systems. Our obfuscation techniques take advantage of polymorphism and exception mechanism of object-oriented languages and can drastically reduce the precision of points-to analysis of the programs. We show that determining precise points-to analysis in obfuscated programs is NP-hard and the fact provides a theoretical basis for our obfuscation techniques. Furthermore, in this paper we present some empirical experiments, whereby we demonstrate the effectiveness of our approaches.
Keywords: mobile agents, security, obfuscation, static analysis, computational complexity

1 Introduction

In recent years mobile agent systems have been paid much attention to as a new computing paradigm. Here *a mobile agent* is a program that migrates through several hosts and performs a specific job on behalf of users. They can not only move to other hosts, but can also offer advanced computing services such as information retrieval and cooperative computation with other agents. In addition because a mobile agent is self-contained, i.e., it is accompanied by resources necessary for execution, it can run on a host offline. Consequently, mobile agent systems can realize a promising computing environment for next generations, in particular an infrastructure suitable for electronic commerce. In such an application area, security is of critical importance since the mobile agents process sensitive data of users on behalf of them.

In order to attain security for mobile agent systems, we must provide protection against the two kinds of attacks: (i) attacks by malicious agents and (ii) attacks by malicious hosts. The problem of the former attacks has been well understood and thus various countermeasures are available, e.g., sandbox security technologies, cryptographic techniques such as encryption, digital signatures, or authentication protocols. On the other hand, few attempts have been made so far on solutions to the latter problem. Hence we can hardly find any established

[*] The author is currently with Sony Corporation, Network Application & Content Service Sector.

techniques to solve the problem, except for those with some dedicated hardware. For instance, encryption cannot be used to solve the problem since encrypted mobile agents must be eventually decrypted into executable forms and then become vulnerable against the attacks mounted by malicious hosts[1].

In order to solve the problem, *obfuscation* of agent programs is very promising [2, 1]. Therefore in this paper we propose novel methods to protect mobile agents via software obfuscation. The proposed methods are to obfuscate Java since Java is one of the most excellent object-oriented languages for developing mobile agents. We believe that our obfuscation techniques are easily applicable to other object-oriented languages such as C++.

Software obfuscation has been vigorously studied so far [3, 2, 1, 4–6]. Unfortunately previous software obfuscation techniques share a major drawback that they do not have a theoretical basis and thus it is unclear how effective they are.

In order to mitigate such a situation, Wang et al. proposed a software obfuscation technique based on the fact that aliases in a program drastically reduce the precision of static analysis of the program [6]. However, their approach is limited to the intraprocedural analysis. Since a program consists of many procedures in general, we must conduct interprocedural analysis whether or not it is obfuscated. Hence Ogiso et al. proposed obfuscation techniques with the use of function pointers and arrays, which greatly hinder interprocedural analysis [5]. Their work is very promising since it succeeds in providing theoretical basises for the effect of obfuscation techniques. However, unfortunately, the techniques in [5, 6] cannot straightforwardly apply to object-oriented languages, especially, to Java.

Our obfuscation techniques take advantage of polymorphism and exception handling mechanism of Java, which can drastically reduce the precision of static analysis of the obfuscated programs and thus make them significantly harder to understand and to modify. Hence in mobile agent systems developed in Java, the techniques can provide protection against attacks by malicious hosts.

More technically, our obfuscation techniques are based on the difficulty of points-to analysis[2], which can be proved to be NP-hard [8]. Therefore, we can also provide a theoretical basis to the techniques. This is one of the great advantages of our approaches over other related work.

We plan to build secure mobile agent systems with our proposed methods and are now conducting various experiments and implementations. In this paper we present some of empirical evaluation, whereby we demonstrate the effectiveness of our approaches.

The rest of the paper is structured as follows. In Sect. 2, we introduce polymorphism and exception in Java to describe our approaches. Next in Sect. 3, we discuss related work to ours and point out their drawbacks. In order to solve them, we propose new obfuscation techniques in Sect. 4, and we theoretically

[1] In this paper, due to space limitation, we cannot go into the details of the problem of attacks by malicious hosts. Refer to [1] for further details.

[2] See also [7] for the difficulty of conducting static analysis in Java in cases other than ours.

evaluate our techniques in Sect. 5. In Sect. 6 we show empirical experiments of the techniques. Finally we conclude this paper in Sect. 7.

2 Java

Our obfuscation techniques take advantage of functions of Java as an object-oriented language such as polymorphism, exceptions, and so on. Therefore, before going into details of the techniques, in this section we explain about the functions that we use.

2.1 Object-Oriented Languages

Object-orientation is the framework to describe a program with objects and messages. Object-oriented language have advantages over traditional languages such as C from the viewpoint of the cost for reuse or maintenance of programs.

Object-oriented languages mainly consist of the following three foundations:

1. **encapsulation**: integrates data and routines,
2. **inheritance**: defines a hierarchical relationships among objects, and
3. **polymorphism**: handles different functions by a unique name.

While these functions often make it easier to implement programs for large scale or advanced application, the behavior of the program is likely to be more complex. As a result, the analysis of object-oriented programs often becomes more difficult. Our proposed obfuscation techniques exploit this fact.

In the rest of this section, we describe about *Polymorphism on Java* and *Exception with subtyping*, which are ingeniously used in our proposed obfuscation techniques.

2.2 Polymorphism on Java

Polymorphism is one of the fundamental mechanisms of object-oriented languages, which can "handle different functions by a unique name." Especially in Java, we can implement polymorphism by the following features;

1. *method override with class subtyping*,
2. *interface*, or
3. *method overload*.

We explain how to implement polymorphism with interface and method overload, which are used for our obfuscation techniques.

```
interface F {
    public void m();
}
class A implements F {
    public void m() { System.out.println("I'm A"); }
}
class B implements F {
    public void m() { System.out.println("I'm B"); }
}
{
    F obj;
    obj = new A();
    c1: obj.m();
    obj = new B();
    c2: obj.m();
}
```

Fig. 1. Example of Interface

Interface Fig. 1 is an example of the use of *interface*. The variable `obj` is defined as the type of *interface* `F`, therefore `obj` can be an instance of a class that implements *interface* `F`. When this code is executed, the string "I'm A" is printed at the program site **c1**, because `obj` is an instance of class `A`. And at the site **c2**, the program prints "I'm B" because `obj` is an instance of class `B` there.

Here notice that the code `obj.m()` at **c1** is identical to the one at **c2**, although, different methods are called according to the class types of `obj`. That behavior is not decided at the time that the program is compiled, but is dynamically decided when it is executed.

```
static void m(int arg) { System.out.println("int"); }
static void m(char arg) { System.out.println("char"); }

{   int i=0; char c=0;
    c1: m(i);
    c2: m(c);
}
```

Fig. 2. Example of Overloading

Method Overloading Next in Fig. 2, we show a Java code that performs method overloading. At the site **c1** and **c2**, methods of the same name `m` are called. The difference between them is the type of the arguments, which is `int` at site **c1** and `char` at **c2**. Consequently the string printed on the terminal is

"int" and "char", respectively. If there are some methods with the same name, the type or the number of the arguments determine which method is called.

2.3 Exception

```
class E1 extends Exception {}
class E2 extends Exception {}

{
    int d, u;
    1: d=1;
    try {
        2: method();
        // may throw an exception E1 or E2
    }
    catch (E1 e1) { 3: d=4; }
    catch (E2 e2) { 4: d=3; }
    5: u = d;
}
```

Fig. 3. Example of Exception

Java uses exceptions to provide error-handling capabilities. An exception is an event that occurs during program execution, which disrupts the normal flow of instructions. In Java programs, *throw* and *catch* (and *finally*) statements are used to handle exceptions. In order to utilize exceptions, at first we define a class for each error type, then throw/catch the instance of the class.

Fig. 3 is a simple example of how to use exceptions in Java codes. If the `method()` called at the site **2** may throw an exception E1 or E2, the value of `d` substituted for u at the site *5* changes dependently on which exception was thrown, or whether the exception was thrown or not. Thus, operation of the program containing exception exhibits non-deterministic property.

3 Related Work

In this section, we discuss some of existing approaches to solve the problem of attacks by malicious hosts.

Sander and Tshudin proposed mobile cryptography for the problem [9]. In mobile cryptography, we can develop programs that perform operation on encrypted data. It has an advantage that the security is provable, although, it cannot be applicable to general agent programs but only to those of a rather specific form. Hence it is of very limited use for practical situations. For other cryptographic approaches, see also [10].

Now protection of mobile agents with software obfuscation is being paid much attention to. Therefore below we present obfuscation techniques proposed so far, some of which are not limited to security for mobile agent systems.

Hohl proposed the concept of 'time-limited blackbox security', which provides tamper-resistance for mobile agents by software obfuscation techniques until a prescribed time limit [1]. For other obfuscation approaches, Mambo proposed obfuscation techniques in which frequency distributions of instructions in obfuscated programs are made as uniformly as possible by limiting available instructions for obfuscation [4].

Unfortunately previous software obfuscation techniques share a major drawback that they do not have a theoretical basis and thus it is unclear how effective they are.

In order to mitigate such a situation, Wang et al. proposed a software obfuscation technique based on the fact that aliases in a program drastically reduce the precision of static analysis of the program [6]. However, their approach is limited to the intraprocedural analysis. Since a program consists of many procedures in general, whether or not it is obfuscated, we must conduct interprocedural analysis. Hence Ogiso et al. proposed obfuscation techniques with the use of function pointers and arrays, which greatly hinder interprocedural analysis [5]. Their work is very promising since they are successful in providing the theoretical basis for the effect of obfuscation techniques.

Unfortunately the techniques in [5, 6] cannot straightforwardly apply to object-oriented languages, especially, to Java, because they require the use of *pointers* or *goto statements*, which are not supported in Java[3]. Therefore we need new obfuscation techniques that can be applicable to Java.

4 Proposed Obfuscation Techniques

From the discussions so far, in this section we shall propose new software obfuscation techniques using object-oriented features of Java.

4.1 Use of Polymorphism

As described in Sect. 2.2, polymorphism is one of the fundamental mechanisms of object-oriented languages, which can handle different functions by a unique name. While polymorpihsm makes it easy to implement complicated algorithms, it also makes the behavior of the created programs more complicated. Consequently it is difficult to analyze those program. We apply such a feature to our obfuscation techniques.

Our obfuscation procedures with respect to polymorphism on Java are given below. They consist of three phases: (1) *Introduction of method overloading*, (2)

[3] Collberg et al. proposed some obfuscation techniques using object-oriented features [2], however their techniques are limited to rather simple ones, e.g., disturbance of class hierarchies. Furthermore they do not provide any theoretical basis about how effective their techniques are.

Introduction of interfaces and dummy classes, and (3) *Change types and* `new` *sentences*. Below, the procedures are concisely described because of space limitation. Also notice that although the example programs below that result from obfuscation are intentionally not so obfuscated for the purpose of explanation, it is not difficult to transform a program into any more obfuscated form.

(1) Introduction of method overloading At first, we introduce new classes `Ag` and `Rt` as preparation. The instance of `Ag` is used to preserve method[4] arguments, and `Rt` is to preserve return values. Then we pick some classes randomly, and we create new classes `Ag1`$\sim$`Agn` derived from `Ag`, where n is the maximum number of methods contained in the picked classes.

Next, we change the name of every method contained in the classes into the same name. Then we change the type of *return* value for every method into type `Rt`, and change the type of the arguments into a type of subclasses of `Ag`. In this step, the numbers of the arguments of the methods are made to be the same. Class `Rt` manages information of return types and values, and class `Ag` and its subclasses manage argument types and values. Moreover, when two or more classes are chosen, some number of dummy methods are added so that the number of the methods of each class becomes the same.

Fig. 4 is an example of definition changes of methods described above. In that case class `A` and `B` are chosen, then the name of every method in two classes changes into '`m`', and a dummy method is added to class `B`. Finally all return types are changed into `Rt`, and the arguments into `Ag1`$\sim$`Ag3`. Here, notice the method `calc` of class `A`. Although the method originally require two arguments of type `int`, it changes into '`m(Ag2)`', which requires one argument of type `Ag2`. Moreover the return type of `calc` is changed from `int` into `Rt`. Therefore, to maintain the semantics of the original program, we need to modify the method call and the method itself. This process is illustrated in Fig. 5. The constructor of `Ag2` requires two `int` arguments, and `getRetValue(int z)` returns `int` value. They correspond to the arguments and the return value of original method `calc`, respectively. We apply this transformation for each method call.

(2) Introduction of interfaces and dummy classes In this step we newly introduce interface, and dummy classes if needed. The interface defines methods transformed in step (1), and we make targeted classes to '**implements**' this interface. Moreover, we newly create classes that play no role (i.e., dummy). These dummy classes also need to implement the interface defined immediately before. If dummy classes are not needed for some reasons (for example, due to performance the program requires), we can cancel to introduce dummy classes.

As continuation of the example given by (1), we show an example in Fig. 6. The interface `I` defines three methods that have the same name '`m`' and return type `Rt`, and the arguments of each method are `Ag1`, `Ag2`, and `Ag3` respectively.

[4] In Java there are two types of method, *instance method* and *static(class) method*, and our procedure does not count static methods. Hereinafter a 'method' means instance method.

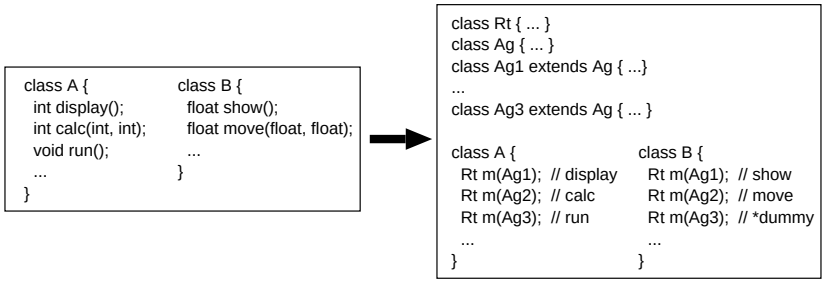

Fig. 4. Change definitions of methods

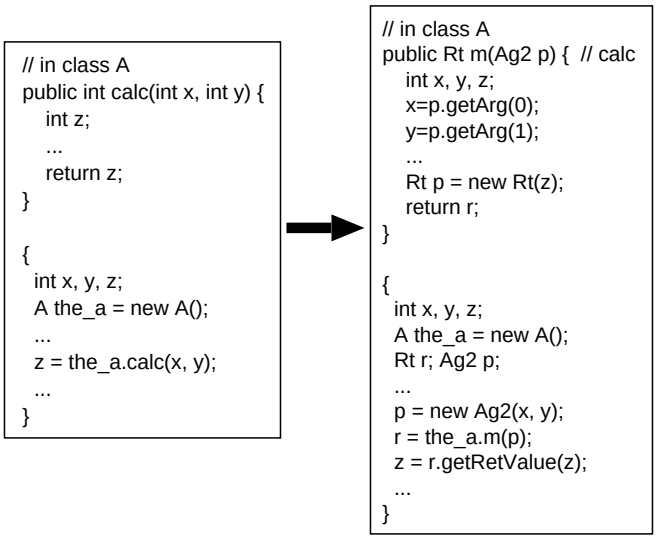

Fig. 5. Modify method and method call

Furthermore, we add the declaration of implementation to class `A` and `B`. Class `C` has the same method as class `A` and `B`.

(3) Change types and `new` *sentences* Finally, we change types of instance variables of targeted classes into the type of interface introduced in the step (2). And for every `new` sentence which creates the reference of the targeted class, we put the `new` sentence into if-sentence with another `new` sentences.

Fig. 7 is an example of that conversion. *EXP_TRUE* is the condition expressions that is always *true* such as `x*(x+1)%2==0` or `y*(y+1)*(y+2)%6==0`. Hence the semantics of the original program is maintained. However, generally speaking, in static analysis it is very difficult to evaluate such expressions and this results in difficulty in determining the execution paths in the presence of

```
interface I { Rt m(Ag1); Rt m(Ag2); Rt  m(Ag3); }

class A implements I {          class B implements I {
  Rt m(Ag1);                      Rt m(Ag1);
  Rt m(Ag2);                      Rt m(Ag2);
  Rt m(Ag3);                      Rt m(Ag3);
  ...                             ...
}                               }
// *dummy
class C implements I {
  Rt m(Ag1);
  Rt m(Ag2);
  Rt m(Ag3);
  ...
}
```

Fig. 6. Definition of new Interfaces and Classes

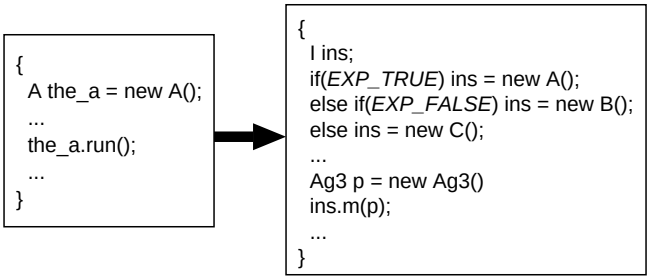

Fig. 7. Change Types and new sentences

if-statements[5] Needless to say, such condition expressions can be made arbitrarily complicated as long as the original semantics is retained. Therefore the if-statements make it difficult to determine the reference variable `ins` points to.

4.2 Use of Exception

In this section, we propose another obfuscation technique using *exception*, which is independent of the technique in Sect. 4.1. Although exceptions are to provide error-handling as explained in Sect. 2.3, of course, they can be inserted in any site of a program. Our technique converts if-sentences to try/catch-sentences as instances.

[5] Here static analysis of a program is conducted under the assumption that all execution paths within procedures, without regard to interprocedural paths, are executable. This assumption is commonly found in the literature and is often called 'meet over all paths' [11].

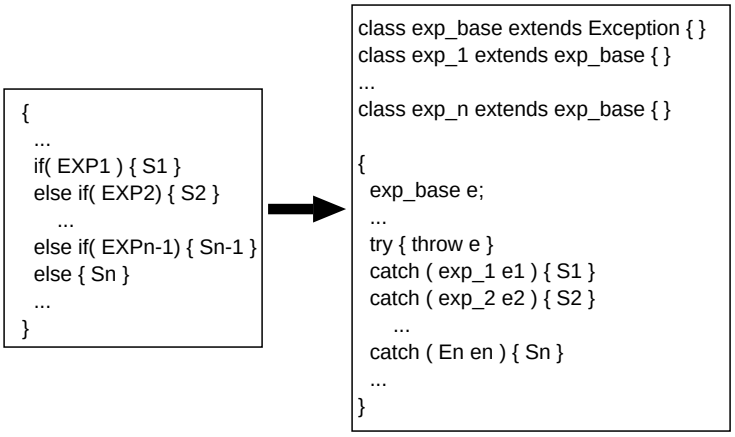

Fig. 8. Use of Exceptions

Here, we consider an if-sentence in Fig. 8, where EXP_1, EXP_2, ..., EXP_{n-1} are appropriate condition expressions, and S_1, S_2, ..., S_{n-1} are sequences of sentences. Now we introduce exception classes `exp_base`, `exp_1`, `exp_2`, ..., `exp_n`, where `exp_1`∼`exp_n` are subclasses of `exp_base`. Then we convert the if-sentence to the try/catch-sentence as shown in the right side in Fig. 8. The variable `e` should be an instance of an appropriate exception class so that obfuscated sentences execute equivalently to the original if-sentence.

4.3 Example of Obfuscation

At the end of Sect. 4, for completeness of the description of this section, we show in Fig. 9 an example of obfuscation to which all obfuscation techniques apply.

5 Complexity Evaluation

Our obfuscation techniques described in Sect. 4 substantially impede precise points-to analysis. In this section, we support this claim by presenting a proof in which we show that statically determining precise points-to is NP-hard.

Theorem 1: In the presence of classes which implement interfaces, method calls by the instances of the classes, and at the same time in the presence of method-overloadings, the problem of precisely determining if there exists an execution path in a program on which a given instance points to a given method at a point of the program is NP-hard [6].

Proof: The proof of Theorem 1 is by reduction from the 3-SAT problem [8]

[6] For further backgrounds behind the way of this proof, see [11].

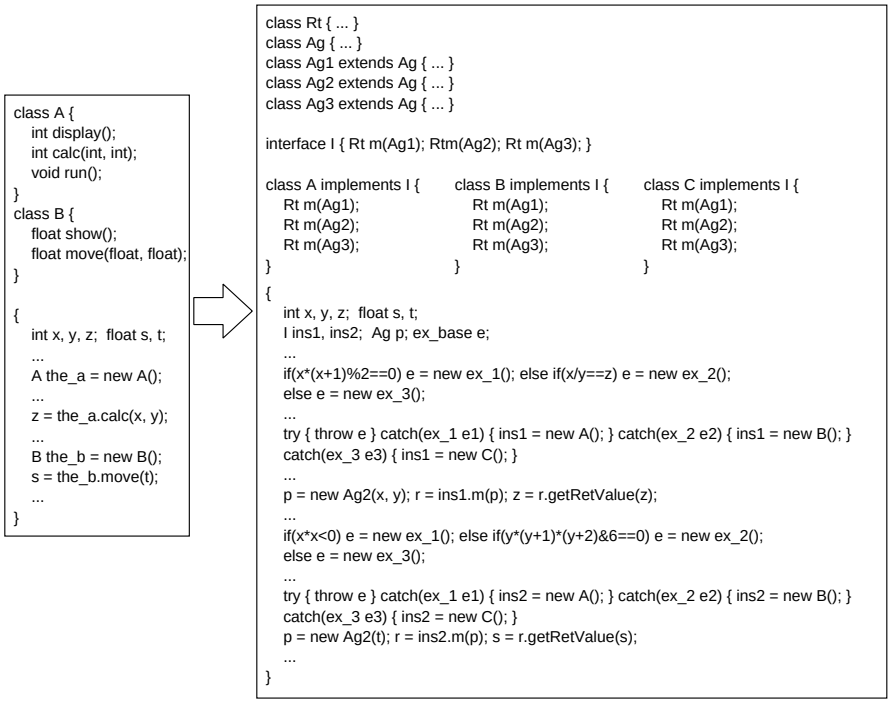

Fig. 9. Example of Obfuscation

for $\wedge_{i=1}^{n}(l_{i,1} \vee l_{i,2} \vee l_{i,3})$ with propositional variables $\{v_1, v_2, ..., v_m\}$, where l_{ij} is a literal and is either v_k or $\overline{v_k}$ for some k $(1 \leq k \leq m)$. The reduction is specified by the program in Fig. 10, which is polynomial in the size of the 3-SAT problem. The conditionals are not specified in the program since we assume that all paths are executable. As will be seen later, paths through the code between `L1` and `L2` represent truth assignments for the propositional variables. The truth assignment on a particular path is encoded in the points-to relationship of certain variables in the program. Paths between `L2` and `L3` then encode in the points-to relationship whether or not the truth assignment resultant from the path to `L2` satisfies $\wedge_{i=1}^{n}(l_{i,1} \vee l_{i,2} \vee l_{i,3})$.

If we interpret v_i pointing to `b_true` as the propositional variable v_i being true, then any path from `L1` to `L2` uniquely determines one truth assignment. Furthermore, the converse is also true, namely, every truth assignment corresponds to exactly one execution path as just mentioned.

Now consider the path from `L2` to `L3`. If the truth assignment for a path from `L1` to `L2` satisfies the formula then every clause has at least one literal which is true. Pick the path from `L2` on which each clause assigns `b_ture` to `c`. Then each assignment corresponds to `c = b_true.and(b_true)` and `c` must point to `b_true` at `L3`. However if the formula is unsatisfiable then every truth assignment

has a clause, say $(l_{i,1} \vee l_{i,2} \vee l_{i,3})$, where all these three literals are false. This implies $l_{i,1}$, $l_{i,2}$, and $l_{i,3}$ all point to `b_false`. Because every path from `L2` to `L3` must go through the statement

`if(-)` c `=` c`.and(`$l_{i,1}$`) else if(-)` c `=` c`.and(`$l_{i,2}$`) else` c `=` c`.and(`$l_{i,3}$`);`

`c` must point to `b_false` on all paths to `L3` and thus `c` never points to `b_true`. Therefore 3-SAT is polynomial reducible to the problem of Theorem 1 and this completes the proof.

6 Empirical Evaluation

In this section we present application of our obfuscation procedures to four programs, RC6, compress, MD5 and FFT. Table 1 shows the difference between the hierarchy/call graphs of the original programs and those of the obfuscated programs. In the rows for a hierarchy graph, '#nodes' represents the sum of the classes and the interfaces, and '#edges' represents the sum of subclassing and implements edges. Furthermore, in the rows of a call graph, '#nodes' represents the number of call sites, and '#edges' represents the number of 'to method nodes'. Here, we can readily see from the table that the increase of the numbers of edges is greater than those of nodes, while the number of edges usually increases in proportion to the number of nodes. In the original call graphs, the numbers of nodes and edges are almost the same, which means that all the call sites and target methods correspond one to one. On the other hand, in the call graphs of the obfuscated programs, the number of edges is 3.4 times the number of nodes on average. Therefore, some call sites have two or more candidates of methods, and these result give a good evidence that precision of analysis is much reduced by our obfuscation techniques.

We have evaluated performance degradation due to the obfuscation, as indicated in Table 2. The experiments were conducted on a Sun Ultra 5(UltraSPARC-II 400Mhz) with Solaris 8 (SunOS 5.8). Programs were compiled and executed by Java version 1.3.1. Each execution time was the average of 1000 times execution. The average rate of execution times of obfuscated programs over the original programs is 1.3, which is not so great as the rise in source codes or class files. Therefore, our obfuscation techniques do not degrade performance so much.

Note that these are results of applying our obfuscation procedures just once. If needed we can apply the procedures two or more times, then it will provide further obfuscated programs.

7 Conclusion

In recent years mobile agent systems have been paid much attention to as a new computing paradigm. However, for advanced application such as electronic commerce, the agent systems are of little value unless their security is ensured. Especially it is significantly important to provide protection for mobile agents against attacks by malicious hosts.

Table 1. Change of hierarchy and call graphs

program			Before Obfuscation	After Obfuscation	ratio
compress	Hierarchy Graph	#nodes	3	21	7.0
		#edges	2	30	15.0
	Call Graph	#nodes	30	74	2.5
		#edges	30	274	9.1
RC6	Hierarchy Graph	#nodes	5	23	4.6
		#edges	4	33	8.3
	Call Graph	#nodes	18	77	4.3
		#edges	18	297	16.5
MD5	Hierarchy Graph	#nodes	10	33	3.3
		#edges	9	42	4.7
	Call Graph	#nodes	194	667	3.4
		#edges	202	1775	8.8
FFT	Hierarchy Graph	#nodes	7	25	3.6
		#edges	6	41	6.8
	Call Graph	#nodes	86	318	3.7
		#edges	86	1057	12.3

In order to solve the problem, obfuscation of agent programs is very promising. Unfortunately previous software obfuscation techniques share a major drawback that they do not have a theoretical basis and thus it is unclear how effective they are. Therefore, in this paper we propose novel obfuscation techniques for Java. We believe it is fairly easy to apply the techniques to other object-oriented languages such as C++.

Our obfuscation techniques take advantage of polymorphism and exception mechanism and can drastically reduce the precision of points-to analysis of the programs. We have shown that determining precise points-to analysis in obfuscated programs is NP-hard and the fact provides a theoretical basis for our obfuscation techniques. Furthermore, in this paper we have presented some empirical experiments. The results show the effectiveness of our obfuscation approaches.

References

1. Hohl, F.: Time limited blackbox security: Protecting mobile agents from malicious hosts. In Vigna, G., ed.: Mobile Agents Security. Volume 1419 of Lecture Notes in Computer Science. Springer-Verlag (1998) 92–113
2. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, the University of Auckland, Auckland, New Zealand (1997)
3. Aucsmith, D.: Tamper resistant software: An implementation. In Anderson, R.J., ed.: Information Hiding: First International Workshop. Volume 1174 of Lecture Notes in Computer Science., Springer-Verlag (1996) 317–333
4. Mambo, M., Murayama, T., Okamoto, E.: A tentative approach to constructing tamper-resistant software. In: New Security Paradigm Workshop. (1997) 23–33

Table 2. Change of program size and execution time

program			Before Obfuscation	After Obfuscation	ratio
compress	program size	source[#lines]	126	332	2.6
		class file[byte]	2906	12056	4.2
	execution time[sec]		0.57	0.69	1.2
RC6	program size	source[#lines]	561	853	1.5
		class file[byte]	6039	18414	2.0
	execution time[sec]		0.70	0.78	1.1
MD5	program size	source[#lines]	762	2142	2.8
		class file[byte]	11257	43462	3.9
	execution time[sec]		0.61	0.92	1.5
FFT	program size	source[#lines]	874	2185	2.5
		class file[byte]	11260	37158	3.3
	execution time[sec]		0.66	0.99	1.5

5. Ogiso, T., Sakabe, Y., Soshi, M., Miyaji, A.: Software obfuscation on a theoretical basis and its implementation. IEICE Transactions on Fundamentals **E86-A** (2003) 176–186
6. Wang, C., Hill, J., Knight, J., Davidson, J.: Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, Department of Computer Science, University of Virginia (2000)
7. Chatterjee, R., Ryder, B.G., Landi, W.: Complexity of points-to analysis of Java in the presence of exceptions. IEEE Transactions on Software Engineering **27** (2001) 481–512
8. Garey, M.R., Johnson, D.S.: Computers and Intractability – A Guide to the Theory of NP-completeness. W. H. Freeman and Co. (1979)
9. Sander, T., Tschudin, C.F.: Protecting mobile agents against malicious hosts. In Vign, G., ed.: Mobile Agents and Security. Volume 1419 of Lecture Notes in Computer Science. Springer-Verlag (1998) 44–60
10. Kotzanikolaou, P., Burmester, M., Chrissikopoulos, V.: Secure transactions with mobile agents in hostile environments. In: Information Security and Privacy: Fifth Australasian Conference on Information Security and Privacy, ACISP 2000. Volume 1841 of Lecture Notes in Computer Science., Springer-Verlag (2000) 289–297
11. Myers, E.W.: A precise inter-procedural data flow algorithm. In: Conference record of the 8th ACM Symposium on Principles of Programming Languages (POPL). (1981) 219–230

```
interface Bool {
    public Bool and(Bool arg);
}
class True implements Bool {
    public Bool and(Bool arg) { return arg; }
}
class False implements Bool {
    public Bool and(Bool arg) { return this; }
}

class theorem {
    Bool b_true, b_false;
    Bool c;

    void var(Bool v1, Bool ¬v1) {
        if(-) var(v1, ¬v1, b_true, b_false);
        else var(v1, ¬v1, b_false, b_true);
    }
    void var(Bool v1, Bool ¬v1, Bool v2, Bool ¬v2) {
        if(-) var(v1, ¬v1, v2, ¬v2, b_true, b_false);
        else var(v1, ¬v1, v2, ¬v2, b_false, b_true);
    }
        . . .
    void var(Bool v1, Bool ¬v1, Bool v2, Bool ¬v2, ... Bool vm,
                Bool ¬vm) {
L2:
        if(-) c = l1,1 else
                        if(-) c = l1,2 else c = l1,3;
        if(-) c = c.and(l2,1) else
                        if(-) c = c.and(l2,2) else c = c.and(l2,3);
            . . .
        if(-) c = c.and(ln,1) else
                        if(-) c = c.and(ln,2) else c = c.and(ln,3);
L3: }

    public theorem() {
        b_true = new True();
        b_false = new False();

L1:     if(-) var(b_true, b_false); else var(b_false, b_true);
    }
    public static void main(String[] args) {
        new theorem();
    }
}
```

Fig. 10. 3-SAT solution iff $\langle c,$ b_true$\rangle$ in Interprocedural Points-to Analysis