Regular Paper

Java Obfuscation – Approaches to Construct Tamper-Resistant Object-Oriented Programs

Yusuke Sakabe,[†,☆] Masakazu Soshi[†] and Atsuko Miyaji[†]

In Java programs, it is difficult to protect intellectual property rights and secret information in untrusted environments, since they are easy to decompile and reverse engineer. Consequently realization of software obfuscation becomes increasingly important. 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 we shall propose new software obfuscation techniques for Java in this paper. Our obfuscation techniques take advantage of features of object-oriented languages, and they 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.

1. Introduction

With the wide spread of computer networks such as the Internet, a huge number of computers of different architectures are now interconnected through the networks. As a result, the classical style of software distribution in binary code form is rapidly being replaced by the one in source code form. Notable examples of such a way of software distribution are via Perl scripts, JavaScripts, and Java applications. Here, *Java*[3] is one of the most promising object-oriented languages used in the large range, for example, in developing network servers or tiny applications on cellular phones. Java applications are usually distributed over the Internet as Java *class files*, i.e., hardware-independent binary codes that contain virtually all the information of the original Java sources. Hence, these class files are easy to decompile.

In such situations, malicious users and hosts can analyze software distributed over the Internet and can extract secret information and/or proprietary algorithms from it. Unfortunately encryption is hardly competent to solve the problems since encrypted programs must be eventually decrypted into executable forms and then adversaries can intercept them in hostile environments.

Consequently realization of software with *tamper-resistance*, which means the difficulty to read and modify the software in an unauthorized manner, becomes increasingly important. Although tamper-resistant software can be realized with the help of hardware, much attention is now being focused on *software obfuscation*, which transforms a program into a tamper resistant form. Thus software obfuscation has been vigorously studied so far[2),4),6),7),9),11)~14),16),18),19),23)]. 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[23)]. 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[18),19)]. Their works are also successful in providing theoretical bases for the effect of obfuscation techniques.

However the techniques in 18), 19), 23) cannot straightforwardly apply to object-oriented languages, especially, to Java. Furthermore, some theoretical basis should be provided for the obfuscation techniques. Unfortunately, most of previous obfuscation techniques fail to meet all of these requirements at once.

† School of Information Science, Japan Advanced Institute of Science and Technology
☆ Presently with Sony Corporation

Here we observe that Java is an object-oriented language, which has some advanced features such as:

- addition/deletion of a part of program without changing existing objects,
- establishment of relationship between one object and another,
- ease of maintenance, and so on.

However in some cases such features make it significantly difficult to perform precise analysis of programs. For example, in the presence of *polymorphism* (see Sect. 2), precise *points-to analysis* is known to be P-space complete[5),10)], which means that there is very little possibility of finding precise solution in such a situation.

From the observation above, we shall propose novel software obfuscation techniques for Java, which take advantage of the features of polymorphisms. Hence the techniques are easily applicable to other object-oriented languages such as C++. In addition, technically our obfuscation techniques are based on the difficulty of points-to analysis, which can be proved to be NP-hard. Therefore, we can also provide a theoretical basis to the techniques.

Finally, in this paper we present some empirical experiments, whereby we demonstrate the effectiveness of our approaches.

The rest of the paper is structured as follows. In Sect. 2 and Sect. 3, we explain about Java and software analysis in general. Next in Sect. 4, we discuss related work and point out the drawbacks. In order to solve such problems, we propose new obfuscation techniques in Sect. 5, and we evaluate our techniques in Sect. 6 and 7. Then we discuss several aspects of our work in Sect. 8 and finally we conclude this paper in Sect. 9.

2. Java

Our obfuscation techniques take advantage of functions of Java as an object-oriented language such as polymorphism. 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 languages have advantages over traditional languages such as C from the viewpoint of cost for reuse or maintenance of programs.

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

```
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

(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 present *Polymorphism on Java*, which is ingeniously used in our proposed obfuscation techniques.

2.2 Polymorphism in Java

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

Fig. 1 is an example of the use of *interface*. The variable `obj` is defined to have the type of *interface* `F`, therefore `obj` can be an instance of a class that *implements* interface `F`. When this

```
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

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`. The behavior cannot be decided when the program is compiled, but is dynamically decided when it is executed.

Method overloading

Next in **Fig. 2**, we show a Java code that performs method overloading. At the sites **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 types or the number of the arguments determine which method should be called.

3. Software analysis and points-to analysis

In this section we present what is software analysis (especially points-to analysis) and describe a threat model we suppose in this paper, i.e., what attackers can or cannot do.

If an adversary is trying to gather or tamper secret information in a program, first he must analyze it by some means. The major and important approach of software analysis is *static analysis*[1),15),17)]. The objective of static analysis is to extract useful information from a program without running it. Generally speaking, static analysis first builds the (control) flow graph of the program and then examines the control flow or data flow of the program through the graph. Static analysis will be discussed in a more formal manner in Sect. 6.

For object-oriented languages such as Java, *points-to analysis*, which is a kind of static analysis, is important to investigate the behavior of a program. At each program point, points-to analysis determines to which object a reference may refer during execution. When a reference is used in an object oriented program, it can potentially refer to many objects of the same type as the reference and thus static (points-to) analysis is much deterred. Furthermore, notice that in almost all cases, a program has multiple if-statements. In such a case, generally speaking, it is well-known that it is undecidable to determine which path is executable[1),20)☆]. Hence the existence of if-statements hinders precise points-to analysis. Putting it altogether, we can conclude that it is remarkably hard to perform points-to analysis[5),20)]. Actually many points-to problems are known to be not easier than NP-complete[17)].

Now it should be easy to see that as the numbers of references and variables of the same type as the references increase, static analysis becomes more difficult. Moreover, as the number of execution paths due to possible combination of if-branches increases, the analysis also becomes more difficult. Our proposed obfuscation techniques exploit the features of polymorphism and succeed in making points-to analysis significantly harder by increasing the number of execution paths that we should examine in the analysis. This will be discussed in Sect. 8.

Finally, in this paper we suppose that what attackers can do is to perform static analysis on obfuscated programs. Therefore, we do not assume that they can perform analysis of any other types, for instance, analysis with a debugger with breakpoint functionalities, which is called *dynamic analysis*[☆☆].

4. Related work

In this section, we discuss some of existing software tamper-resistance approaches.

In 1997, Mambo proposed new software obfuscation techniques in which frequency distributions of instructions in obfuscated programs are made as uniformly as possible by limiting

☆ Therefore we usually conduct static analysis on the assumption 'meet over all paths'. This will be further discussed in Sect. 6.

☆☆ However, we are doing research on software obfuscation techniques resistant to dynamic analysis. We hope that we can publish it in a near future.

available instructions for obfuscation[14].

Keeping in mind application to mobile agent systems, Hohl proposed the concept of 'time-limited blackbox security', which provides tamper-resistance until a prescribed time limit in order to protect mobile agents against attacks mounted by malicious hosts[12].

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[23]. 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[18),19]. Their works are also successful in providing the theoretical basis for the effect of obfuscation techniques.

Chow et al. proposed an obfuscation technique, which also has a theoretical basis[6]. Their approach embeds an intractable problem with respect to computational complexity theory into a program to be obfuscated. Consequently, to analyze the obfuscated program is equivalent to solve the intractable problem. Their approach is very interesting and seems promising. However, we can point out two significant differences between their approach and ours. First, our approach does not embed an intractable problem as a whole into the obfuscated program. From the viewpoint of computational complexity theory, such transformation as in 6) is not always necessary since all we need is polynomial reducibility between the intractable problem and some analysis problem (see Sect. 6). Second, their approach does not suppose object oriented languages and hence it would be difficult to apply it to object oriented programs.

Most of the techniques in 6), 18), 19), 23) 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. Therefore we need now obfuscation techniques that can be applicable to Java.

Collberg et al. proposed some obfuscation techniques using object-oriented features[7], 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.

Tyma proposed "overload induction" techniques[22], although they do not take obfuscation into consideration. They transform the identifiers of multiple methods into an identifier of a smaller length. So at first glance, the proposed technique in Sect. 5.1 would seem similar to them. However, one of the biggest differences between them is that our approach can transform arguments and return values of methods into the same identifiers.

Sosonkin et al. propose obfuscations of object-oriented programs[21]. In their paper they propose three obfuscation techniques, i.e., class coalescing, class splitting, and type hiding. The techniques are interesting, although, they also are not based on theories. However, they seem to be independent of our proposed techniques, and could be combined with ours to strengthen obfuscation.

5. Proposed obfuscation techniques

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

5.1 Use of Polymorphism

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

5.1.1 Introduction of method overloading

At first, we randomly pick some classes to be obfuscated and add some number of dummy methods☆ in the classes so that the number of the methods of each class becomes the same.

☆ 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.

Let s be the number of the methods (including dummy methods). Then we introduce new classes `Ag` and `Rt` as preparation☆. Class `Rt` manages information of return types and values, and class `Ag` and its subclasses manage argument types and values. Furthermore, we create new classes `Ag1`∼`Ag`s derived from `Ag`.

Next, we change the name of every method contained in the classes into the same name. Then we change the type of a return value of every method into type `Rt`, and change the type of the arguments into a type of subclasses of `Ag`.

An example process of the definition changes of methods just described above are depicted in **Fig. 3** and **Fig. 4**. Let us suppose that class `A` and `B` in Fig. 3 are chosen to be obfuscated. Then the name of every method in two classes changes into '`m`', and a dummy method is added to class `A`. Finally all the return types are changed into `Rt`, and the arguments into `Ag1`∼`Ag3`.

Now let us examine the method `move` of class `A`. Although `move` originally requires two arguments of type `float`, it now changes into '`m(Ag2 x)`', which requires one argument `x` of type `Ag2`. Moreover the return type of `move` is changed from `float` into `Rt`. Here class `Ag2` is defined to accommodate argument values of two methods, i.e., `move` of class `A` and `power` of class `B`. Furthermore, in `Ag2`, private fields `i` and `f` are arrays of `int` and `float`, respectively. They are used to store actual arguments of `move` and `power`. Example usages of classes `Rt` and `Ag2` and of methods `getRetValue` and `getArg` are given below.

In order to maintain the semantics of the original program, we modify the method calls and the way of arguments and return values passing. The example of this is illustrated in Fig. 4. Note that method `move` is now transformed into `m`. Then in order to execute `move` with arguments `x` and `y` in line 13 of the figure at the right hand side of Fig. 4, first we newly create an instance `x` of class `Ag2` with initial arguments `f1` and `f2` in line 12 (notice that variables `x`, `y`, and `z` now correspond to `f1`, `f2`, and `f3`, respectively). These two arguments are stored in `x.f[0]` and `x.f[1]` by calling the constructor `Ag2(f1, f2)` in the line 12 (see also the definition of this constructor in Fig. 3). These private fields values `f1` and `f2` can be accessed in the method `m` (i.e., `move`) via calls to `getArg(f1, 0)` (in line 3) and `getArg(f2, 1)` (in line 4), respectively (see the definition of `getArg` in Fig. 3). Here '`0`' and '`1`' mean the first and the second arguments, respectively. Furthermore, note that the type of the first argument of `getArg` is used to obtain the private field of the desirable type (i.e., `float`), so the actual values of the arguments `f1` and `f2` are not significant when we invoke `getArg` in lines 3 and 4.

Finally, a return value of `m` is stored in an instance `r` of class `Rt` in line 13 (see also the statement in line 5). The execution of the statement in line 5 sets `f3` to `r.f`. The actual return value can be obtained by calling `r.getRetValue(f3)` in line 14, where the type of `f3` is used to distinguish the desirable private fields for the return value from others as in the case of `getArg`. So at the entry to the call of `getRetValue(f3)`, the actual value of `f3` is not significant (see the definitions of `getRetValue` in Fig. 3).

Needless to say, constructors `Rt()`s and `Ag2()`s, methods `getRetValue`, `getArg`, and `m` have the form of method overloading.

We apply the transformation described above for each method call.

5.1.2 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. 5**. 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. Furthermore, we add the declaration of implementation to class `A` and `B`. Class `C` has the same method as class `A` and `B`.

5.1.3 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

☆ Construction of class `Ag` and `Rt` is a bit complicated and somewhat tedious. So we describe the actual procedure in Sect. A.1.

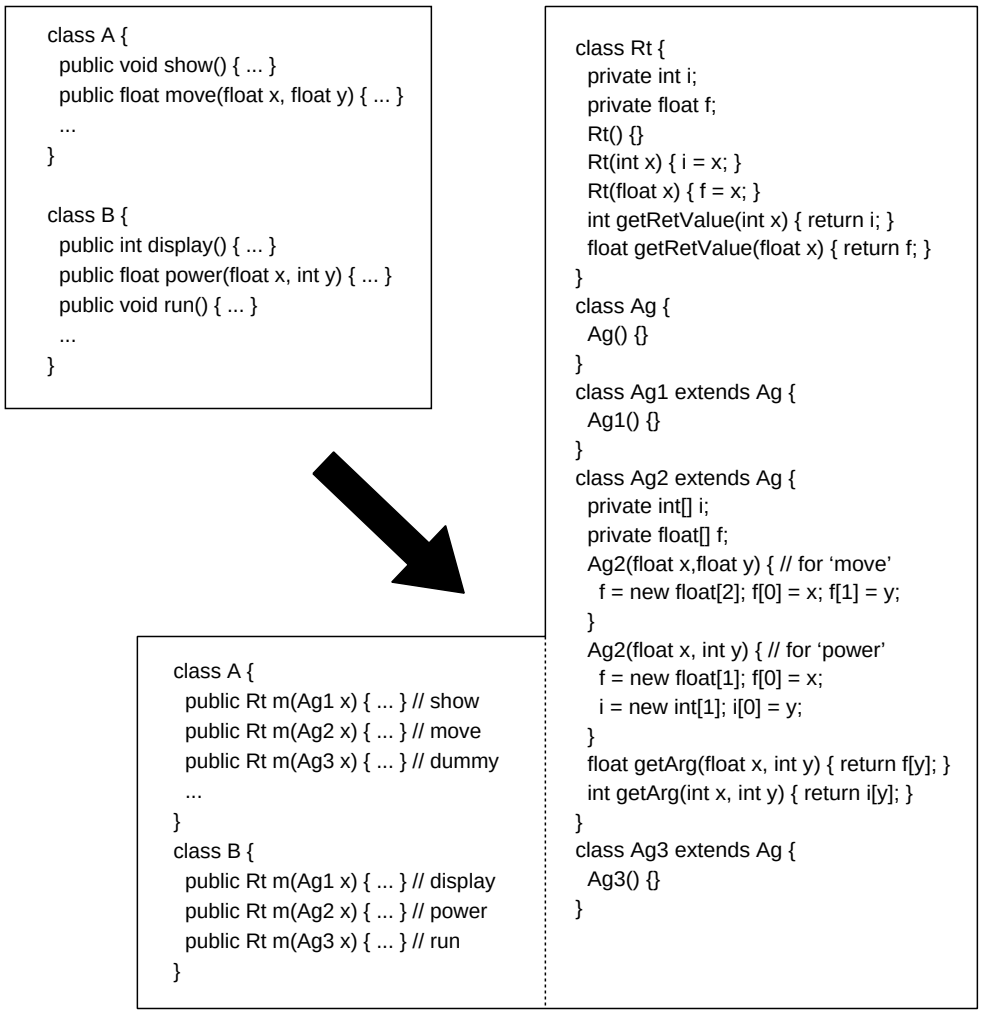


Fig. 3 Change definitions of methods

with another `new` sentences.

Fig. 6 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 if-statements☆. 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.

☆ This will be discussed in Sect. 6 in more details.

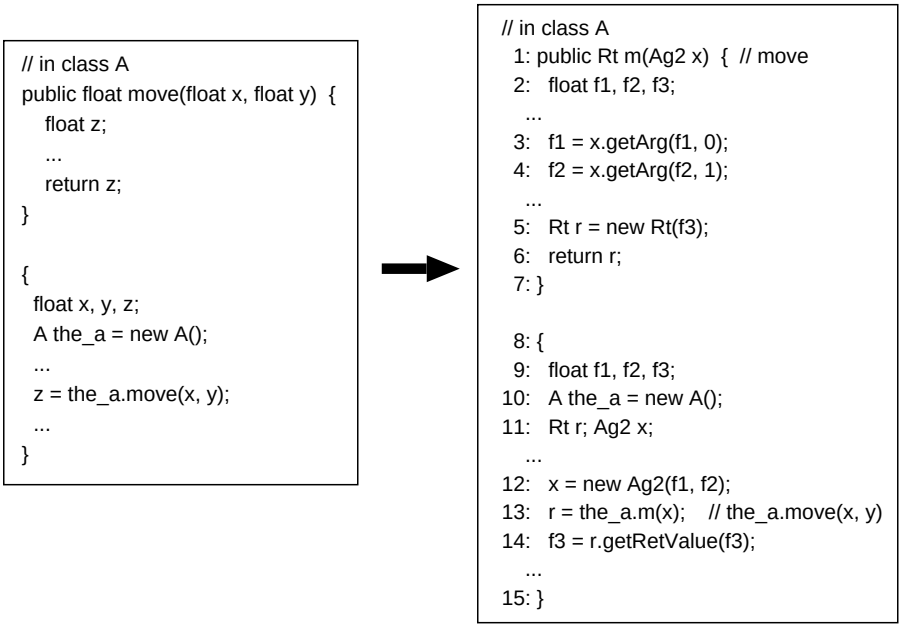
5.2 Example of obfuscation

At the end of Sect. 5, for completeness of the description of this section, we show in **Fig. 7** an example of obfuscation to which all obfuscation techniques apply.

6. Complexity Evaluation

Our obfuscation techniques described in Sect. 5 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 an interface, method call by the instance of the classes, and the presence of method-overloadings (i.e., polymorphism), the



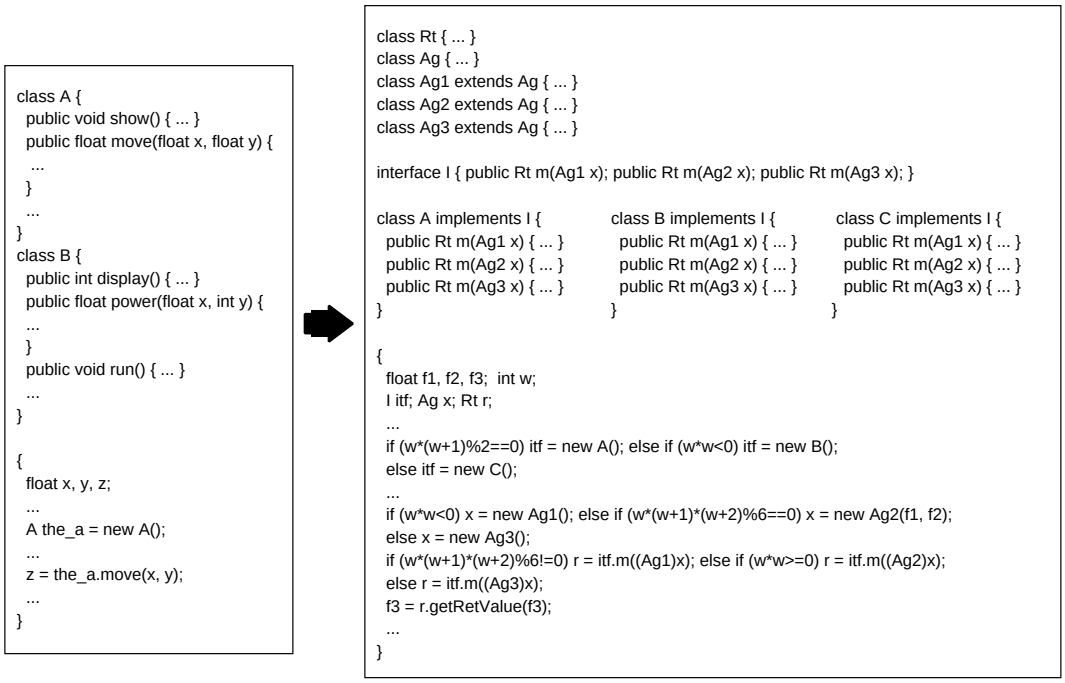
```
// in class A
public float move(float x, float y) {
   float z;
   ...
   return z;
}

{
  float x, y, z;
  A the_a = new A();
  ...
  z = the_a.move(x, y);
  ...
}
```

```
// in class A
 1: public Rt m(Ag2 x) { // move
 2:   float f1, f2, f3;
   ...
 3:   f1 = x.getArg(f1, 0);
 4:   f2 = x.getArg(f2, 1);
   ...
 5:   Rt r = new Rt(f3);
 6:   return r;
 7: }

 8: {
 9:   float f1, f2, f3;
10:   A the_a = new A();
11:   Rt r; Ag2 x;
   ...
12:   x = new Ag2(f1, f2);
13:   r = the_a.m(x);   // the_a.move(x, y)
14:   f3 = r.getRetValue(f3);
   ...
15: }
```

Fig. 4 Modify methods and method calls



```
class A {
  public void show() { ... }
  public float move(float x, float y) {
   ...
  }
  ...
}
class B {
  public int display() { ... }
  public float power(float x, int y) {
  ...
  }
  public void run() { ... }
  ...
}

{
  float x, y, z;
  ...
  A the_a = new A();
  ...
  z = the_a.move(x, y);
  ...
}
```

```
class Rt { ... }
class Ag { ... }
class Ag1 extends Ag { ... }
class Ag2 extends Ag { ... }
class Ag3 extends Ag { ... }

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

class A implements I {
  public Rt m(Ag1 x) { ... }
  public Rt m(Ag2 x) { ... }
  public Rt m(Ag3 x) { ... }
}
class B implements I {
  public Rt m(Ag1 x) { ... }
  public Rt m(Ag2 x) { ... }
  public Rt m(Ag3 x) { ... }
}
class C implements I {
  public Rt m(Ag1 x) { ... }
  public Rt m(Ag2 x) { ... }
  public Rt m(Ag3 x) { ... }
}

{
  float f1, f2, f3;  int w;
  I itf; Ag x; Rt r;
  ...
  if (w*(w+1)%2==0) itf = new A(); else if (w*w<0) itf = new B();
  else itf = new C();
  ...
  if (w*w<0) x = new Ag1(); else if (w*(w+1)*(w+2)%6==0) x = new Ag2(f1, f2);
  else x = new Ag3();
  if (w*(w+1)*(w+2)%6!=0) r = itf.m((Ag1)x); else if (w*w>=0) r = itf.m((Ag2)x);
  else r = itf.m((Ag3)x);
  f3 = r.getRetValue(f3);
  ...
}
```

Fig. 7 Example of Obfuscation

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☆.

☆ Here static analysis of a program is conducted under the assumption that all execution paths are executable. This assumption is commonly found in the literature and is often called 'meet over all paths'[15]. For further backgrounds behind the way of this proof, see 17), for example.

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

class A implements I {
  public Rt m(Ag1 x) { ... }
  public Rt m(Ag2 x) { ... }
  public Rt m(Ag3 x) { ... }
  ...
}

class B implements I {
  public Rt m(Ag1 x) { ... }
  public Rt m(Ag2 x) { ... }
  public Rt m(Ag3 x) { ... }
  ...
}

// *dummy
class C implements I {
  public Rt m(Ag1 x) { ... }
  public Rt m(Ag2 x) { ... }
  public Rt m(Ag3 x) { ... }
  ...
}
```

Fig. 5 Definition of new Interfaces and Classes

```
{
  float x, y, z;
  A the_a = new A();
  ...
  z = the_a.move(x, y);
  ...
}
```

→

```
{
  float f1, f2, f3; I itf; Rt r;
  if (EXP_TRUE) itf = new A();
  else if (EXP_FALSE) itf = new B();
  else itf = new C();
  ...
  Ag2 x = new Ag2(f1, f2);
  r = itf.m(x);
  ...
}
```

Fig. 6 Change Types and new sentences

Proof: The proof of Theorem 1 is by reduction from the 3-SAT problem[10] 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. 8**, 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_true` 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. □

7. Empirical Evaluation

In this section we present application of our obfuscation procedures to five programs, that is, zip (compression), decompress (lzw_j decompression), FFT (Fast Fourier Transform), RC6 (a symmetric encryption), and MD5 (a hash function).

Obfuscation was done in manners described in Sect. 5. The numbers of classes and methods in the obfuscation processes are as follows:

- Only one class in each program was obfuscated.
- The number of methods (including dummy methods) in each obfuscated class was prepared to be 20 at most and then method overloading was applied to them in a manner presented in Sect. 5.1.
- Each if-statement was made to have 20 branches at most.

Table 7 shows the differences between the hierarchy/call graphs of the original programs and those of the obfuscated programs. In hierarchy graph, 'nodes' represents the sum of classes and interfaces, and 'edges' represents the sum of subclassing and implements edges. Furthermore, in the call graph, 'nodes' represents the number of call sites, and 'edges' represents the number of 'to method nodes'.

In the hierarchy graphs of the obfuscated programs, in the average, the number of nodes and edges are 10.7 and 20.0 times greater than the originals, respectively. On the other hand, in

```
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:
        /* The code below will create a ⟨c, b_true⟩ points-to
           iff the truth assignment from above makes the formula true */

        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. 8 3-SAT solution iff $\langle c, \texttt{b\_true} \rangle$ in Interprocedural Points-to Analysis

the call graphs of the obfuscated programs, the number of nodes and edges are 4.4 and 18.5 times greater than the originals, respectively. Furthermore, we can see that in the average one call site has more than 4.2 candidates of methods. These results 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 7**. The experiments were conducted on a Windows XP machine with Pentium 4 1.80GHz and 768MB memory. Programs were compiled and executed by Java version 1.4.2_06. Each execution time was the average of 1000 times execution. The average rate of execution times of obfuscated programs over the original programs is 1.43, which is not so great as the rise in source codes (5.32) or class files (10.3). Therefore, our obfuscation techniques do not decrease performance so much.

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

8. Discussion

So far we have presented our proposed obfuscation techniques in Sect. 5 and theoretical analysis in Sect. 6. Furthermore, we have demonstrated the effectiveness and usefulness of our approaches by experimental evaluation

Table 1 Change of hierarchy and call graphs

program			Before Obfuscation	After Obfuscation	ratio
zip	Hierarchy Graph	nodes	3	46	15.3
		edges	2	65	32.5
	Call Graph	nodes (Call site)	37	163	4.4
		edges	37	183	4.96
decompress	Hierarchy Graph	nodes	4	48	12.0
		edges	3	67	22.3
	Call Graph	nodes (Call site)	41	160	3.9
		edges	41	1160	28.3
FFT	Hierarchy Graph	nodes	5	48	9.6
		edges	4	67	16.8
	Call Graph	nodes (Call site)	47	174	3.7
		edges	49	216	4.41
RC6	Hierarchy Graph	nodes	5	48	9.6
		edges	4	68	17.0
	Call Graph	nodes (Call site)	26	125	4.81
		edges	26	860	33.1
MD5	Hierarchy Graph	nodes	7	50	7.14
		edges	6	69	11.5
	Call Graph	nodes (Call site)	208	1044	5.02
		edges	212	4648	21.9

Table 2 Change of program size and execution time

program			Before Obfuscation	After Obfuscation	ratio
zip	program size	source [lines]	102	887	8.7
		class file [byte]	2727	44154	16.19
	execution time [sec]		0.00558	0.00566	1.01
decompress	program size	source [lines]	143	965	6.75
		class file [byte]	3641	41276	11.3
	execution time [sec]		0.0087	0.0204	2.34
FFT	program size	source [lines]	250	1032	4.13
		class file [byte]	4638	44900	9.68
	execution time [sec]		0.0017	0.0018	1.06
RC6	program size	source [lines]	578	1465	2.53
		class file [byte]	7149	51018	7.14
	execution time [sec]		0.0071	0.0118	1.66
MD5	program size	source [lines]	774	3505	4.53
		class file [byte]	11567	84801	7.33
	execution time [sec]		0.191	0.211	1.10

in Sect. 7.

In this section, for a summarization purpose, we show by examples why our obfuscation techniques obstruct points-to analysis considerably.

First of all, remember that as the numbers of references and variables of the same type as the references increase, static analysis becomes more difficult, as mentioned in Sect. 3. Moreover, as the number of execution paths due to possible combination of if-branches increases, the analysis also becomes more difficult.

Our obfuscation techniques force methods to be called in a polymorphic way. For example, let us consider Fig. 3. In the figure, the method `m` now potentially points to `show`, `move`, `display`, `power`, or `dummy`. Furthermore, by introduction of interfaces and dummy classes, many classes '**implements**' an interface. The interface `I` in Fig. 5, for example, is possibly implemented by classes `A`, `B`, or `C`.

Now let us look at Fig. 6. The interface `itf` can be implemented by `A`, `B`, or `C`. Furthermore, statement '`r = itf.m(x);`' is a polymorphic method call. At this stage our obfuscation techniques newly introduce if-statements with many branches. Notice that it is significantly difficult to determine realizable execution paths in the presence of if-statements, as discussed in Sect. 3. Therefore, in Fig. 6, it becomes much difficult for points-to analysis to determine which method, i.e., one of six methods just presented, is actually called in `itf.m(x);`. This is the intuitive reason why our obfuscation techniques hinders points-to analysis remarkably.

We have succeeded in providing a theoretical basis for the difficulty of static analysis of the

obfuscated programs. However, from the practical viewpoint, we would have to consider some kind of attacks. For example, as in previous obfuscation techniques[7], our proposed techniques also depend on 'always-true' conditions such as $i * (i + 1)\%2 == 0$, which are well-known as 'opaque' conditions[7),8]. Since the opaques look somewhat peculiar, if they are found by an attacker, then they might facilitate the analysis of the attacker. However, Collberg and others extensively have studied various techniques to make it harder to analyze the opaques[8] and so we can take advantage of such techniques to obstruct the attacks mounted on our opaque conditions.

9. Conclusion

In Java programs it is difficult to protect intellectual property rights and secret information in untrusted environments, since they are easy to decompile and reverse engineer. Consequently realization of software obfuscation becomes increasingly important. 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 we have proposed new software obfuscation techniques for Java in this paper. Our obfuscation techniques take advantage of features of object-oriented languages, and they 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, it is fairly easy to apply our obfuscation techniques to other object-oriented languages such as C++. The empirical results show that the precision of analysis is reduced and the structure of graphs of obfuscated programs are made more complicated than original ones. They imply the effectiveness of our obfuscation approaches.

Acknowledgments This research is conducted as a program for the "Fostering Talent in Emergent Research Fields" in Special Coordination Funds for Promoting Science and Technology by Ministry of Education, Culture, Sports, Science and Technology, and is partially supported by Grant-in-Aid for Exploratory Research, 16650011, 2004.

References

1) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers: Principles, Techniques and Tools*, Addison-Wesley (1986).
2) Akai, K. and Matsumoto, T.: An Evaluation Model of Tamper Resistant Software, *Computer Security Symposium (CSS 2004)*, pp. 253–258 (2004).
3) Arnold, K., Gosling, J. and Holmes, D.: *The Java Programming Language*, Addison-Wesley, 3rd edition (2000).
4) Aucsmith, D.: Tamper resistant software: An implementation, *Information Hiding: First International Workshop* (Anderson, R. J.(ed.)), Lecture Notes in Computer Science, Vol. 1174, Springer-Verlag, pp. 317–333 (1996).
5) Chatterjee, R., Ryder, B. G. and Landi, W.: Complexity of Points-To Analysis of Java in the Presence of Exceptions, *IEEE Transactions on Software Engineering*, Vol. 27, No. 6, pp. 481–512 (2001).
6) Chow, S., Gu, Y., Johnson, H. and Zakharov, V. A.: An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs, *4th International Conference on Information Security (ISC 2001)* (Davida, G. I. and Frankel, Y.(eds.)), Lecture Notes in Computer Science, Vol. 2200, Springer-Verlag, pp. 144–155 (2001).
7) Collberg, C., Thomborson, C. and Low, D.: A Taxonomy of Obfuscating Transformations, Technical Report148, Department of Computer Science, the University of Auckland, Auckland, New Zealand (1997).
8) Collberg, C., Thomborson, C. and Low, D.: Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs, *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 184–196 (1998).
9) Fukushima, K., Kiyomoto, S. and Tanaka, T.: Program Obfuscation Scheme by Encoding Variables, *Computer Security Symposium (CSS 2004)*, pp. 247–252 (2004).
10) Garey, M. R. and Johnson, D. S.: *Computers and Intractability – A Guide to the Theory of NP-completeness*, W. H. Freeman and Co. (1979).
11) Goto, H., Mambo, M., Shizuya, H. and Watanabe, Y.: Evaluation of Tamper-Resistant Software Deviating from Structured Programming Rules, *Information Security and Privacy: Sixth Australasian Conference on Information Security and Privacy, ACISP 2001*, Lecture Notes in Computer Science, Vol.2119, Springer-Verlag, pp. 145–158 (2001).

12) Hohl, F.: Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts, *Mobile Agents Security* (Vigna, G.(ed.)), Lecture Notes in Computer Science, Vol. 1419, Springer-Verlag, pp. 92–113 (1998).
13) Kuwahara, J. and Matsumoto, T.: A Method of Constructing Tamper-Resistant Java Classfiles, *Symposium on Cryptography and Information Security (SCIS 2000)*, p. D10 (2000).
14) Mambo, M., Murayama, T. and Okamoto, E.: A Tentative Approach to Constructing Tamper-Resistant Software, *New Security Paradigm Workshop*, pp. 23–33 (1997).
15) Marlowe, T. J. and Ryder, B. G.: Properties of Data Flow Frameworks: A Unified Model, *Acta Informatica*, Vol. 28, No. 2, pp. 121–163 (1990).
16) Monden, A., Takada, Y. and Torii, H.: Methods for Scrambling Programs Containing Loops, *IEICE Transactions on Fundamentals*, Vol. J80-D-I, No. 7, pp. 664–652 (1997).
17) Myers, E.W.: A Precise Inter-Procedural Data Flow Algorithm, *Conference record of the 8th ACM Symposium on Principles of Programming Languages (POPL)*, pp. 219–230 (1981).
18) Ogiso, T., Sakabe, Y., Soshi, M. and Miyaji, A.: Software Tamper Resistance Based on the Difficulty of Interprocedural Analysis, *The Third International Workshop on Information Security Applications (WISA 2002)*, pp. 437–452 (2002).
19) Ogiso, T., Sakabe, Y., Soshi, M. and Miyaji, A.: Software Obfuscation on a Theoretical Basis and its Implementation, *IEICE Transactions on Fundamentals*, Vol. E86-A, No. 1, pp. 176–186 (2003).
20) Ramalingam, G.: The undecidability of aliasing, *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 5, pp. 1467–1471 (1994).
21) Sosonkin, M., Naumovich, G. and Memon, N.: Obfuscation of Design Intent in Object-Oriented Applications, *Proceedings of the ACM Workshop on Digital Rights Management*, pp. 142–153 (2003).
22) Tyma, P. M.: Method for renaming identifiers of a computer program, United States Patent, No. 6,102,966, Assignee: PreEmptive Solutions, Inc. (2000).
23) Wang, C., Hill, J., Knight, J. and Davidson, J.: Software Tamper Resistance: Obstructing Static Analysis of Programs, Technical Report CS-2000-12, Department of Computer Science, University of Virginia (2000).

Appendix

A.1 Construction of argument class `Ag` and return value class `Rt`

In this appendix, we describe a procedure to construct argument class `Ag` and return value class `Rt`. The procedure presented here is actually used in experimental obfuscation programs in Sect. 7.

(1) First of all, choose multiple classes to be obfuscated. The chosen classes are denoted by A_1, ..., A_n. Furthermore, OA_i denotes the class obtained by obfuscating A_i $(1 \leq i \leq n)$.

(2) Let m_i be the number of methods defined in A_i $(1 \leq i \leq n)$. Furthermore, let us suppose that $m = \max\{m_1, m_2, ..., m_n\}$. Now we arbitrarily determine s $(\geq m)$, which is the number of methods to be obfuscated. $s - m_i$ means the number of dummy methods to be introduced in OA_i $(1 \leq i \leq n)$.

(3) Compose dummy methods. We can define arbitrary dummy methods if they do not interfere with any methods in the program. As a result, each class has s methods including dummy methods.

(4) In order to express the types of return values of methods of OA_i $(1 \leq i \leq n)$ in one type, we introduce an `Rt` class. An `Rt` class is structured in the following way:

- Each field variable of `Rt` corresponds to a distinct type of return values of the methods and stores a return value of the type. For example, let us look at `Rt` in Fig. 3. Private field variable `i` corresponds to the return values of `display`. Similarly, `f` corresponds to `move` or `power`. The types of return values of `show` and `run` are `void`, so that they do not have corresponding fields in `Rt`.
- Each constructor of `Rt` is introduced to set a return value of a distinct type. In Fig. 3, constructor `Rt(float f)` corresponds to return values of `move` and `power`.
- `getRetValue` is used to retrieve a return value of a distinct type. The type of an argument of `getRetValue` determines a return value of a desirable type. For example, in Fig. 3, if you want to get a return value of type `float` for `move` or `power`, call

`getRetValue` with an arbitrary argument variable of type `float`.

(5) Define an Ag class. An Ag class is actually an empty class as follows:

```
class Ag {
   Ag( ){ }
}
```

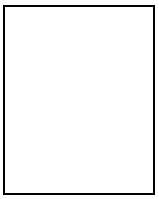
(6) Choose one method from every A_i ($1 \leq i \leq n$). Now we have chosen n methods. In order to express arguments types of the chosen n methods in one type, introduce an class Ag_1. Repeat this procedure s times and finally we have Ag_1, Ag_2, ..., and Ag_s. Each Ag_i inherits from Ag, i.e., Ag_i is a child of Ag in the Java class hierarchy ($1 \leq i \leq n$).
A_i has a similar structure to that of `Rt` ($1 \leq i \leq n$), except for the private fields and `getArg` methods. Each private field of A_i is an array of the same type of arguments of the original methods. For example, in Fig. 3, the field `i`, which is an array of `int`, accommodates the second argument of `power`. In a similar manner, the field `f`, which is an array of `float`, accommodates the arguments `x` and `y` of `move`, or `x` of `power`. Finally, each `getArg` is used to obtain an argument value of a distinct type. The type of the first argument of `getArg` is used to obtain the private field of the desirable type, as discussed in Sect. 5.1.1.

(7) Define an interface I. The interface I has the following form:

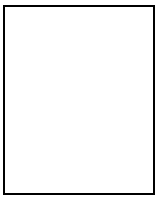
```
interface I {
  public Rt m(Ag1 x);
       ...
  public Rt m(Ags x);
}
```

(8) For $1 \leq i \leq n$, let OA_i '**implements**' the interface I (see also Sect. 5).

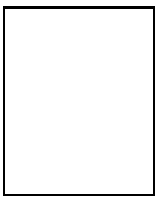
(Received November 30, 2004)
(Accepted March 6, 2005)



Yusuke Sakabe received his B.E. degree from the Nagoya Institute of Technology in 2000, and his M.E. degree from Japan Advanced Institute of Science and Technology in 2003. Since joining Sony Corporation in 2003, he has been working as a computer engineer for network services.



Masakazu Soshi received his B.E. and M.S. degrees from the University of Tokyo in 1991 and in 1993, respectively, and his Ph.D. degree from the University of Electro-Communications in 1999. He worked as an associate for the University of Electro-Communications from 1997 to 1998 and for the Japan Advanced Institute of Science and Technology (JAIST) from 1999 to January 2003. Since February 2003, he has been a research associate professor of JAIST. His research interests include access matrix models, mobile agent security, anonymous communication, software obfuscation, quantum cryptography, and IP traceback.



Atsuko Miyaji received the B. Sc., the M. Sc., and Dr. Sci. degrees in mathematics from Osaka University, Osaka, Japan in 1988, 1990, and 1997 respectively. She joined Matsushita Electric Industrial Co., LTD from 1990 to 1998 and engaged in research and development for secure communication. She has been an associate professor at JAIST (Japan Advanced Institute of Science and Technology) since 1998. She has joined the computer science department of University of California, Davis since 2002. Her research interests include the application of projective varieties theory into cryptography and information security. She received IPSJ Sakai Special Researcher Award in 2002 and the Standardization Contribution Award in 2003. She is a member of the International Association for Cryptologic Research, the Institute of Electronics, Information and Communication Engineers and the Information Processing Society of Japan.