# メソッド分散によるJava言語の難読化手法の提案

福島 和英† 櫻井 幸一‡

〒812-0061 福岡市東区箱崎6丁目10番1号

† 九州大学大学院システム情報科学府 ‡ 九州大学大学院システム情報科学研究院

† fukusima@tcslab.csce.kyushu-u.ac.jp ‡ sakurai@csce.kyushu-u.ac.jp

**概要**

難読化とはプログラムの機能を保ったまま，ソースコードを解析されにくいものへと変換を行う手法である．この手法は，ソフトウェアの知的財産の保護に応用することができる．本研究はJava言語のソースコードに対して，メソッドをクラス間に分散させることにより，カプセル構造を破壊する難読化を行なう．また，インターフェース, メソッドオーバーロードを導入した難読化手法 [刑部ら,ISEC2002-6(2002-5)] を重ねて適用することによって，プログラムの解析がNP困難となり，難読化されたプログラムの静的解析に対する耐性に理論的な根拠を与える．

**キーワード**: 耐タンパーソフトウェア, ソフトウェアセキュリティー, point-to 解析, 難読化, Java,

# A Proposal of a Obufuscatione Technique for Java language by Distributing Methods

Kazuhide FUKUSHIMA† Kouichi SAKURAI‡

Department of Computer Science and Communication Engineering, Kyushu University, 6-10-1 Hakozaki, Higashi-ku, Fukuoka, Fukuoka, 812-8581, Japan

† fukusima@tcslab.csce.kyushu-u.ac.jp ‡ sakurai@csce.kyushu-u.ac.jp

**Abstract**

Obfuscating transformation is a technique for protecting the intellectual property rights of software. This transformation makes the program more difficult for a reverse-engineer to understand but do not affect the functionality of the program. In our research we separate fields and methods in order to destroy these encapsulated structures. By this mean we successfully obfuscate the source-code of Java language. Moreover by applying another obfuscation technique[ISEC2002-6(2002-5)] of Osabe et al using interface and method-overload later, we can make point-to-analysis of the program NP-hard. As a result, a quantitive reason can be given to the difficulty of the static analysis of the obfuscated program.

**Keywords**: Tamper-resistant software, Software Security, Obfuscation , Java, Point-to Analysis

## 1 はじめに

### 1.1 背景

Java言語のコードはマシンやオペレーティングシステムに依存しないクラスファイルへとコンパイルされ，Java仮想マシン (JVM) 又はJavaインタープリタなどのソフトウェアを通じて実行される．そのため，他のプラットフォームで実行を行うときにも移植やリコンパイルなどの作業は一切不要である．

しかし，この性質を実現するためにクラスファイル中には，ソースコード中のほとんどのデータが残ってしまう．そのためクラスファイルの可読性は高く，現在までに逆コンパイラをはじめとするクラスファイルの解析ツールが数多く開発されてきた．結果として，ソフトウェアは悪意ある第三者のリバースエンジニアリング (逆行解析) によりプログラム中のデータ構造やアルゴリズムを盗用される危険にさらされてしまう．この問題への対策として，ソフトウェアの耐タンパー化が必要となる．

### 1.2 耐タンパーソフトウェア

耐タンパーソフトウェアとは論理的，物理的な手段によって，外部からの観測や不当な改ざんを行うことが困難なソフトウェアのことである．耐タンパー性を実現するため，現在迄にさまざまな手法が考案されている [4]．

1 つの方法として，ソフトウェアの暗号化が考えられる．しかし，実行するためには復号化を行わなければならず，結局はユーザーはソースコードの情報を得てしまうことになる．別の方法としてはサーバサイド方式がある．プログラムのうち，保護する必要が無い部分はユーザに配布し，重要なアルゴリズム，データが含まれている部分はサーバに残しておく．サーバ側のプログラムはサービスは行うが，プログラムがユーザーに渡ることはないのでプログラムは完全に保護される．しかし，プログラムの実行時に通信が必要になることは避けられない．現在のところ，最も有効だと考えられているものの 1 つに難読化という手法がある．難読化とはプログラムの機能を保ったまま，ソースコードを解析されにくいものに変換することである．この手法によって，プログラムを完全に保護できるという保証はない [4]．しかし，プログラムの解析を行うためのコストをそのプログラムの価値と比べて十分に高くできれば，実質的にプログラムが解析されることはないと考えられる．

### 1.3 従来の難読化に対する研究の問題点

現在，逆コンパイラに代表される Java のクラスファイルの解析ツールは数多く存在している．このため，すでに Java 言語のソースコードに対する難読化手法は数多く提案されている [2][4],[5]．しかし，これらの手法はオブジェクト指向言語の特性を十分に利用していない．

そこで，刑部ら [1] はメソッドオーバロード，インターフェースを用いて，静的解析に対する理論的な安全性を持つ手法を提案した．

一方，クラス間の関係を解析するツールはほとんど存在しないが，そのことに着目した手法はほとんど提案されていない．本研究では，メソッドをクラス間に分散させることによりクラス間の関連を分かりにくくする．さらに，[1] の手法を重ねて適用することにより，理論的な安全性を持たせる．

## 2 メソッド分散による手法

### 2.1 難読化手法

Java 言語では，データ構造 (フィールド) とそれに対する操作 (メソッド) をひとまとめにして定義する (カプセル化) ことにより，クラスの外部からは内部のメソッドが行っている処理を考慮する必要がなくなる．このことを Java 言語における抽象性と呼ぶ．提案手法ではメソッドを分散させ，抽象性を破壊することにより，クラス間の関連を分かりにくくすることができる．

以下で示す手順 1 から手順 3 までの操作によって, メソッドをクラス間に分散させるための準備を行う．さらにメソッドを分散した後には，手順 4 によって呼び出し側の修整を行う必要がある．

**手順 1　カプセル化の破壊**

1. フィールドの公開
   すべてのクラスのすべてのフィールドの修飾子を **public** に変更する．このことにより，すべてのプログラムのサイトからすべてのフィールドにアクセスできるようになる．

2. メソッドの公開
   すべてのクラスのすべてのメソッドの修飾子を **public static** に変更する．このことにより，すべてのメソッドはクラスメソッドとしてすべてのクラスのサイトから (クラス名).(メソッド名) の形で呼び出されるようになる．

**手順 2　メソッドの変更**

手順 1 のカプセル化の破壊の際に，もともとはインスタンスメソッドであったものが，クラスメソッドに変更されてしまう場合がある．そのため，メソッドの引数としてインスタンスを新たに追加する必要がある．以下の図 1，図 2 に例を示す．

```
class A{
    private int a;
    public void set(int x){
        a = x;
    }
}
```

図 1　変更前

```
class A{
    public int a;
    public static void set(int x, A Obj){
       Obj.a = x;
    }
}
```

図 2　変更後

変更後にはメソッド set はクラス A のインスタンスである Obj を新たに引数としてとるようにする必要がある．更に，フィールド a への代入命令もインスタンス Obj のフィールド a への代入であることを明示的に示さなければならなくなる．

**手順 3　クラス情報の除去**
手順 2 のメソッド変更により，もともとインスタンスメソッドであったものは，そのクラスのインスタンスを引数としてもつクラスメソッドに変換された．このままではメソッドをクラス間に分散させても，その引数を調べることによってもともと属していたクラスが直ちに分かってしまう．そこで以下の手順により，引数としてとるインスタンスの型を統一することができる．

1. インスタンスを引数とするメソッドを持つクラスの中から他のクラスとの継承関係を持たない $C_1, \cdots, C_n$ を選び出す．
2. 新たなクラス $C_0$ を宣言する．
3. クラス $C_0$ のフィールドは，クラス $C_1, \cdots C_n$ に含まれるすべてのフィールドを集めたものとする．
4. クラス $C_0$ のメソッドは必要はないが，後の操作で分散したメソッドを置くことは可能である．
5. クラス $C_1, \cdots, C_n$ をクラス $C_0$ のサブクラスにする．すなわち，クラス $C_1, \cdots, C_n$ の宣言の後に **extends** $C_0$ を付け加える．
6. クラス $C_1, \cdots, C_n$ の中のメソッドの引数のインスタンスをすべて $C_0$ に変更する．

手順 1 から手順 3 までの操作によってすべてのメソッドはどこからでも呼び出すことが可能なクラスメソッドに変更される．これにより，全てのメソッドは任意のクラスに配置可能である．

**手順 4　呼び出し側の修整**
手順 1 から手順 3 までのメソッドの変更および，メソッドをクラス間に分散したことにより，メソッドの呼び出し側でも修整を行う必要がある．
例えば，クラス A のインスタンスメソッド setA(int) はクラス A のインスタンス obj_A を介して図 3 のように呼び出されていたはずである．

```
obj_A.setA(3);
```

図 3　難読化前のメソッド setA の呼び出し

しかし，難読化を適用することによって，引数としてクラスのインスタンスが加わり，他のクラス (例えばクラス B) のクラスメソッドに変更される．ゆえに，以下の図 4 のように setA の呼び出しを修整する必要がある．

```
B.set(3,Obj_A);
```

図 4　難読化後のメソッド setA の呼び出し

## 2.2　提案手法に対するコストの評価

手順 3 の操作によって使用するメモリーの量はクラス $C_1, \cdots C_n$ に含まれるフィールドの数に比例して増大するが，クラス数が少なければ数倍に収まると考えられる．また，メソッドの数やメソッド内部で実行される命令は変わらないので，実行時間はほとんど変わらないといえる．

# 3　静的解析に対する理論的安全性

前章で提案した手法では，安全性に対する明確な根拠が示されていない．この節では，メソッドの分散による難読化と，メソッドオーバーロードとインターフェースによる難読化 [1] を重ねることにより，プログラムへの静的解析に対する計算量的な安全性を持つ新たな難読化手法を提案する．

## 3.1　安全性の根拠

メソッドオーバーロードとインターフェースを用いた手法 [1](刑部ら) の安全性は以下の定理に基づいている．

**定理**
Java 言語のソースコードにメソッドオーバーロードが存在し，かつインターフェースを実装したクラスが存在するとき，手続き間の正確な point-to 解析は NP-hard である．

## 3.2　point-to 解析

point-to [3] は $< \mathrm{var}, \mathrm{obj} >$ という形式で記述される．その意味は var が参照型の変数であり，obj は (1) 参照型の static 変数 (2) 参照型のローカル変数 (3) プログラムのあるサイト m で作成されたオブジェクトの配列 $\mathrm{object_m}$ (4) プログラムのあるサイト n で作成されたオブジェクト $\mathrm{object_n}$.f の中のいずれかであり，変数 var が obj を参照しているということである．point-to

解析とは，プログラムのあるサイトにおいて成り立つ point-to を求める問題である．

### 3.3 難読化手法の適用順序

この章では，2 つの難読化を重ねて適用するが，2 通りの方法が考えられる．

(1) インターフェース，メソッドオーバーロードを用いた難読化変換 [1] を先に行い，その後，我々が提案したメソッド分散による難読化変換を行う．

(2) 我々が提案したメソッド分散による難読化変換を先に行い，その後，インターフェース，メソッドオーバーロードによる難読化変換を行う．

ここで (1) の方法に対して考察を行う．メソッドオーバーロードとは，1 つのクラス内に名前が同一で引数が異なる複数のメソッドを配置することである．(1) の方法は，同一クラス内に名前が同一のメソッド配置しても，その後に，メソッドの分散を行うのでメソッドオーバーロードを導入した意味がなくなってしまう．インターフェースの導入に対しても同様の議論が成り立つ．よって，(2) の方法を用いることにする．以下の節で説明するメソッドオーバーロードの導入とインターフェースの導入はメソッド分散による難読化を行った後のソースコードを対象としている．

### 3.4 メソッドオーバーロードの導入

以下の手順によりメソッドオーバーロードを導入することができる．

1. 任意のメソッドを 1 つ選ぶ．そのメソッドが存在するクラスを $C_i$，メソッド名を M とする．
2. クラス $C_i$ 内に同じ名前のダミーのメソッド M を作成する．ただし，引数は元のメソッド M と異なるようにする．
3. $C_i$.M(); と呼び出していたメソッド M を図 5 の様に呼び出すようにする．
   元のメソッド M の引数はなし，ダミーのメソッド M の引数は int 型とする．

```
if (EXP_TRUE) Ci.M();
else Ci.M(3)
```

図 5　メソッドオーバーロードの導入後

EXP_TRUE は評価して true になる論理式である．論理式の部分が常に true になることにより，Ci.M() だけが実行されることが保証され，この変換が実行結果に影響を与えないことが示される．

### 3.5 インターフェースの導入

以下の手順によりインターフェースを導入することができる．

1. 任意のメソッドを 1 つ選ぶ．そのメソッドが存在するクラスを $C_i$，メソッド名を M とする．
2. メソッド M の修飾子を **publc static** から **public** に変更する．
3. 任意のクラス $C_j$ を選ぶ．(ただし $i \neq j$)
4. クラス $C_j$ にダミーのメソッド M を追加する．修飾子は **public** とする．ただし，引数は元のメソッド M と全く同じものとする．
5. インターフェース I を定義し，その中でメソッド M を宣言する．
6. クラス $C_i, C_j$ でインターフェースを実装する．すなわちクラス $C_i, C_j$ の宣言の後に **implements** I を付け加える．
7. $C_i$.M() と呼び出していたメソッド M を図 6 の様に呼び出すようにする．
   メソッドの引数はなしとする．

```
I Obj;
if (EXP_TRUE) I = new Ci();
else I = new Cj();
I.M();
```

図 6　インターフェース導入後

EXP_TRUE は評価して true になる論理式である．論理式の部分が常に true になることにより，I = new Ci(); だけが実行されることが保証され，この変換が実行結果に影響を与えないことが示される．

以上により，メソッドオーバーロードが存在し，なおかつインターフェースを実装したクラスが存在することになるので，4.1 節の定理により，プログラムに対する point-to 解析が NP 完全となることが示される．

# 4　難読化の適用例

クラス数が 2 つの簡単なプログラムに対して 3 章，4 章で提案した手法を適用する．この章では，クラス構造を図示することによって変換の様子を説明する．実際のソースコードについては論文の最後に付録として示す．

まず，難読化を行う前のクラス構造を示す．

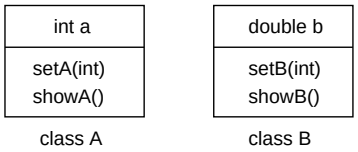


図 7　難読化前のクラス構造

このプログラムは図 7 のように 2 つのクラス A,B を持つ．クラス A は int 型のフィールド a を持ち, フィールドの値を設定するためのメソッド setA とフィールドの値を表示するためのメソッド showA を持つ. クラス B は double 型のフィールド b を持ち, フィールドの値を設定するためのメソッド setB とフィールドの値を表示するためのメソッド showB を持つ.

3 章の難読化手法 (メソッド分散) を適用すると，図 8 のようなクラス構造になる．

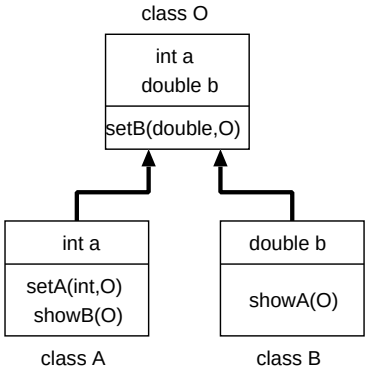


図 8　3 章の難読化適用後のクラス構造

ここでは，setA と showB をクラス A に，showA をクラス B に，setB をクラス O に設置した．各メソッドの引数には新たにクラス O のインスタンスが加えられる．

4 章の難読化手法 (メソッド分散+メソッドオーバーロードの導入) を適用すると，図 9 のようなクラス構造になる．

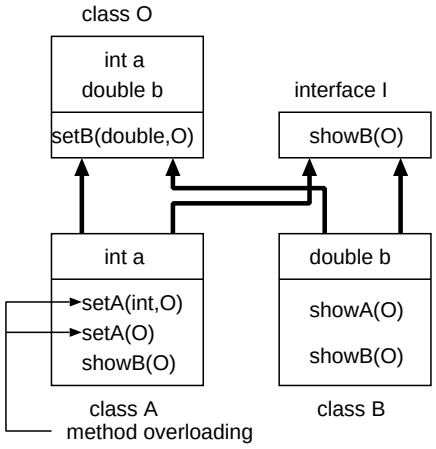


図 9　4 章の難読化適用後のクラス構造

クラス A の showB と全く同じ引数をとるダミーのメソッド showA をクラス B に置き，インタフェース I をクラス A，クラス B で実装した．さらに，setA と名前が等しく引数の型が異なるダミーのメソッド setA() を同じクラス A 内に配置した．以上のように，インターフェースとメソッドオーバーロードを導入し，このプログラムに対する point-to 解析を NP-hard にすることができた．

# 5　まとめ

本研究ではオブジェクト指向である Java 言語の大きな特徴であるカプセル化構造に着目した変換を行うことによって，抽象性の破壊を行うことができた．さらに我々が提案した手法適用した後にも，メソッドオーバーロード，インターフェースを利用した他の難読化手法が簡単に適用でき，静的解析に対する理論的な安全性を持つことができる．本論文で紹介した手法の比較を以下の表にまとめる．

| | 実行時間 | メモリ量 | コードサイズ | 理論的根拠 | 抽象性の破壊 |
|---|---|---|---|---|---|
| 提案手法 | ほぼ不変 | 数倍 | ほぼ不変 | なし | 可能 |
| 手法[1] | ほぼ不変 | ほぼ不変 | 数倍 | あり | 不可能 |
| 合成手法 | ほぼ不変 | 数倍 | 数倍 | あり | 可能 |

今後の課題としては，提案した難読化手法の実装，さらに別の難読化手法との組み合わせの評価などが考えられる．

# 参考文献

[1] 刑部, 双紙, 宮地, “オブジェクト指向言語の難読化の提案”, IEICE Japan Tech. Rep. , ISEC2002-6(2002-05),

33-38.

[2] 桑原, 松本, “Java クラスファイルに対する対タンパー化方式の提案”, SCIS2000.

[3] R.Chatterjee, B.G.Ryder, and W.A.Landi, “complexity ob Point-to Analysis ob Java in the Presence of Exceptions”, IEEE Transactions on Software Engineering, Vol.27, 481-512, 2001.

[4] Douglas Low, “Java control flow obfuscation”, Thesis (MSc–Computer Science)–University of Auckland, 1998.

[5] Christian Collberg, Clark Thomborson, Douglas Low, “A taxonomy of obfuscating transformations”, Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

# 付録

5章の難読化の適用に関するプログラムを以下に記す.

```
class A{
    private int a;
    public void setA(int x){
        a = x;
    }
    public void showA(){
        System.out.println("a = " + a);
    }
}
class B{
    private double b;
    public void setB(double x){
        b = x;
    }
    public void showB(){
        System.out.println("b = " + b);
    }
}
class before{
    public static void main(String[] args){
        A obj_a = new A();
        obj_a.setA(1);
        obj_a.showA();
        B obj_b = new B();
        obj_b.setB(1.5);
        obj_b.showB();
    }
}
```

図 8　難読化前のプログラム

```
class O{
    public int a;
    public double b ;
    public static void setB(double x, O obj){
        obj.b = x;
    }
}
class A extends O{
    public int a;
    public static void setA(int x, O obj){
        obj.a = x;
    }
    public static void showB(O obj){
        System.out.println("a = " + obj.b);
    }
}
class B extends O{
    public double b;
    public static void showA(O obj){
        System.out.println("b = " + obj.a);
    }
}
class after1{
    public static void main(String[] args){
        A obj_a = new A();
        A.setA(1,obj_a);
        B.showA(obj_a);
        B obj_b = new B();
        O.setB(1.5,obj_b);
        A.showB(obj_b);
    }
}
```

図 9　3章の難読化手法を適用した後

```
class O{
    public int a;
    public double b ;
    public static void setB(double x, O obj){
        obj.b = x;
    }
}
interface I{
    public void showB(O obj);
}
class A extends O implements I{
    public int a;
    public static void setA(int x, O obj){
        obj.a = x;
    }
    public static void setA(O obj){
        obj.a = 0;//dummey
    }
    public void showB(O obj){
        System.out.println("a = " + obj.b);
    }
}
class B extends O implements I{
    public double b;
    public static void showA(O obj){
        System.out.println("b = " + obj.a);
    }
    public void showB(O obj){
        System.out.println("a = " + O);
        //dummey
    }
}
class after2
{
    public static void main(String[] args){
A obj_a = new A();
        //メソッドオーバーロードを用いた難読化
        if (x*(x+1)%2==0) A.setA(1,obj_a);
        else A.setA(obj_a);
        B.showA(obj_a);
        B obj_b = new B();
        O.setB(1.5,obj_b);
        //インターフェースを用いた難読化
        I obj_i;
        if (x*(x+1)*(x+2)%6==0) obj_i = new A();
        else obj_i = new B();
        obj_i.showB(obj_b);
    }
}
```

図 10　4章の難読化手法を適用した後