論　文

# API ライブラリ名隠ぺいのための動的名前解決を用いた名前難読化

玉田　春昭[†a)]　　中村　匡秀[††]　　門田　暁人[†]　　松本　健一[†]

## Obfuscating API Library Names Using Dynamic Name Resolution

Haruaki TAMADA[†a)], Masahide NAKAMURA[††], Akito MONDEN[†], and Ken-ichi MATSUMOTO[†]

**あらまし**　名前難読化とは，プログラム中の名前（識別子）を別の名前に付け替えることで，プログラムを理解しづらいものにするソフトウェア保護手法である．従来の名前難読化手法は，各名前を静的に別の文字列で置換するものであり，プログラム中に現れる任意のユーザ定義の名前を隠ぺいできる．しかしながら，従来手法を用いてシステム定義の名前（標準ライブラリや API の呼出し等）を難読化することは，プログラムの移植性を著しく低下させるため，現実的に不可能である．そこで本論文では，オブジェクト指向言語を対象に，システム定義の名前をも隠ぺい可能な新たな名前難読化手法を提案する．具体的には，プログラム中の名前使用部分をあらかじめ暗号化しておき，実行時に名前を復号して当該処理を実行する，動的名前解決の仕組みを導入する．提案手法では，オブジェクト指向言語のリフレクション機構を用いて，クラスの参照，メソッド呼出し，フィールドの参照・代入に現れる任意の名前を動的解決する方法を実現している．また，提案手法を Java プログラム用に実装し評価実験を行った．ある実用プログラムへの適用では，4.11 倍の性能劣化でプログラム中のすべてのクラス名，メソッド名，フィールド名（計 10,580 回出現）を難読化できることが分かった．

**キーワード**　ソフトウェア保護，著作権保護，難読化，オブジェクト指向，リフレクション

## 1. まえがき

インターネット等の情報流通基盤の発展により，ソフトウェアは入手しやすくなっている反面，その不正利用も年々深刻化している．ソフトウェアプロテクションとは，こうした不正利用からソフトウェア（含データ）を保護するための技術の総称であり，その重要性が高まってきている．

しかし，今日，ソフトウェアに適用されたプロテクトのほとんどが，プログラムの不正な解析行為（クラックと呼ばれる）によって無効化されている．例えば，ある DVD 再生ソフトウェアがクラックされ，元来ライセンスを受けた開発ベンダしか知り得ない，DVD コンテンツ復号鍵と復号アルゴリズムが漏えいしたことは記憶に新しい [1]．その結果，DVD の暗号解除ツール，及び，復号済みの DVD コンテンツがインターネット上に流出している．最近では，Apple 社の iTunes もクラックの対象となり，音楽コンテンツのコピープロテクトを無効化するパッチが公開されている [2]．

ソフトウェアをこうしたクラックから保護するため，ソフトウェア難読化が研究されている．難読化とは，与えられたプログラムを，その振舞いを変えないように，非常に読みにくいプログラムへと等価変換する技術であり，これまでにも様々な手法が提案されてきている（例：データ難読化 [3]，制御フロー難読化 [4]，演算難読化 [5], [6]）．本論文では特に，オブジェクト指向ソフトウェアを対象に，名前難読化 [7] に注目して議論を行う．

ソフトウェアをクラックする際の典型的な手法の一つとして，攻撃者がプログラム中の名前（フィールド，メソッド，クラス，型等の識別子）を探し，それらを頼りにプログラムの理解を行う方法がある．名前難読化は，これらの名前を意味のない分かりにくい名前へと変換し，元来の意味を隠ぺいすることで，攻撃者のクラックに必要なプログラム理解コストを増大させることをねらう．名前難読化は，実装が比較的容易なこ

† 奈良先端科学技術大学院大学情報科学研究科，生駒市
Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan
†† 神戸大学大学院工学研究科情報知能学専攻，神戸市
Graduate School of Engineering, Kobe University, Kobe-shi, 657–8501 Japan
a) E-mail: harua-t@is.naist.jp

とから，現在ほとんどの難読化ツールに実装されている（例：DashO [8]，CodeShield [9]，ZKM [10]）．

従来の名前難読化は，プログラム中の名前を別の名前に静的に置換するものである．したがって，自前で作成した（ユーザ定義の）メソッドやフィールド変数，クラス名などを任意の文字列に置換することは可能である．しかしながら，プログラムが利用するシステム API やライブラリ関数（Java で例を挙げれば，`System.out.println()`）等，システム定義の名前を静的に置換することは現実的に不可能である．あらゆる環境で汎用的に用いられる名前を変更することは，そのプログラムの移植性を著しく下げるためである．一般にこれらの名前は，プログラムの実行コード中にもそのまま現れ，攻撃者の格好の手掛りとなる [11]．

そこで本論文では，オブジェクト指向ソフトウェアを対象に，動的名前解決を行うことで任意の名前を隠ぺいする，新たな名前難読化手法を提案する．具体的には，プログラム中の名前使用部分をあらかじめ暗号化しておき，実行時に名前の参照があるたびに名前を復号して，元来の処理を実行するよう，プログラムを変換する．一般的なプログラミング言語では，名前の参照は静的（コンパイル時）に決定している必要がある．しかしながら，オブジェクト指向言語では，実行時にクラスやメソッドの情報を取得し，操作することのできるリフレクション機構が備わっている．提案手法では，リフレクションを用いて，クラスの参照，メソッド呼出し，フィールドの参照・代入に現れる任意の名前を，その参照時に動的解決する方法を実現する．結果として，プログラムの静的なクラックを著しく困難にすることが可能となる．

また，提案手法を Java プログラム用に実装し，性能及びセキュリティの観点において評価した．性能評価では，ある実用規模のプログラム（Jakarta Commons Digester 1.7）に対して難読化を行い，2,359 のフィールド参照，673 のフィールドへの代入，197 のオブジェクト生成，7,351 のメソッド呼出しに用いるすべての名前を隠ぺいすることができた．そして，難読化前後の実行速度を比較した結果，約 4 倍程度の性能劣化で難読化が可能となることが分かった．セキュリティ評価では，静的解析と動的解析の 2 種類のクラック方法を想定して議論を行う．更に，攻撃者が提案手法を知っている場合の攻撃に対する耐性についても議論を行う．

## 2. 準　　備

### 2.1 ソフトウェア難読化

ソフトウェア難読化とは，あるプログラムを非常に理解しづらい等価なプログラムに変換することで，プログラムを不正なクラックから保護するものである．難読化の概念をより厳密に定義するために，まず，プログラム理解について考える．人間（攻撃者）がプログラムを理解する際には，様々な認知活動が行われるため，プログラム理解の一般的な定義を与えることは難しい．また，プログラムの何を理解するかで，認知活動は異なる．例えば，あるモジュールの機能を理解する場合と，データ構造を理解する場合，特定の関数の場所を理解する場合とでは，理解のプロセスが異なる．このことに注目し，本論文では，プログラムの理解とは「与えられたプログラムから理解の対象となる情報を取り出すこと」と定義する．

[定義 1]（プログラム理解） $p$ を与えられたプログラム，$X$ を $p$ に含まれる任意の情報（の集合）とする．このとき，ユーザが $p$ から $X$ を何らかの方法で外部に抽出（リバースエンジニアリング）できたとき，「ユーザは $X$ に関して $p$ を理解した」と定義する．このとき，ユーザが理解にかかるコストを $cost(p, X)$ と書くことにする．ここでコストとは，解析にかかる時間，労力，必要な知識，設備などを含む．

[定義 2]（難読化） $p$ を与えられたプログラム，$X$ を与えられた $p$ の情報（の集合）とする．また，$p$ の入出力写像を $\mathcal{IO}_p : I \to O$ と書く．ここで，$I$ はすべての入力の集合，$O$ はすべての出力の集合である．このとき，「$p$ の $X$ に関する難読化」とは，あるプログラム変換 $T$ を $p$ に適用して，以下の条件を満たすプログラム $p'(= T(p))$ を得ることである．

**条件 1**　$\mathcal{IO}_p = \mathcal{IO}_{p'}$

**条件 2**　$cost(p, X) < cost(p', X)$

条件 1 は難読化前と難読化後のプログラムが，同一の入出力写像をもつ，すなわち，プログラムの外部的な振舞いが変わらないことを保証するものである．条件 2 は，難読化後のプログラム $p'$ から情報 $X$ をとり出すことが困難になることを意味する．

難読化は必ずしも $p$ のソースコードに適用されるとは限らない．一般に保護の対象となるプログラムのソースコードは公開されないため，むしろ，実行コード（例：ネイティブコードやバイトコード）やアセンブリコードに直接適用することが多い．

### 2.2 名前難読化

名前難読化は，プログラム中に現れる名前（識別子）を別のものに付け替える難読化である．プログラム言語における名前は，計算機にとっては単なる識別子であるため，名前の付替えがプログラムの挙動に影響を与えることはない．しかし，名前は人間にとってプログラムを理解するための重要な手掛りとなる [12]．したがって，元プログラムの名前を非常に分かりづらい名前に変換することで，難読化を実現する．

[定義 3]（名前難読化） $p$ を与えられたプログラム，$U_p$ を $p$ に現れるすべての名前の集合，$N_p(\subset U_p)$ を難読化すべき任意の名前の集合とする．$p$ の名前難読化とは，$p$ における各名前 $n \in N_p$ を別の名前 $n'(= T(n)$ とする ) に変換し，別のプログラム $p'$ を得る難読化である．ここで，$T$ は 1 対 1 写像 $T : N_p \rightarrow N_{p'}$ $(N_{p'} \subset U_{p'})$ である．

オブジェクト指向言語の場合，名前は主にプログラム $p$ 中に，クラス名，フィールド名，メソッド名，変数名といった形で現れる．更に，同じ名前でも定義部（宣言部）と使用部（呼出し部）の双方に現れる．名前難読化では，$p$ が含むすべての名前から，難読化すべき名前の集合 $N_p$ をいかに選択するか，また，名前の変換方法 $T$ をいかに実装するかがかぎとなる．

### 2.3 従来手法

従来の名前難読化手法は，プログラム中の定義部に現れる名前を静的に別の名前に置き換えるものである．具体的には，以下の手順で行われる．

[従来の名前難読化の手順]

入力： プログラム $p$，名前集合 $N_p$.

出力： 難読化されたプログラム $p'$.

手順： 各 $n \in N_p$ について，以下の操作を行う．結果として得られたプログラムを $p'$ とする．

Step 1: 各 $n \in N_p$ について，$p$ における $n$ の定義部を別の名前 $n'$ に置き換える．

Step 2: $p$ における $n$ の使用部を，Step 1 で置き換えた $n'$ に置き換える．

入力における $N_p$ として，ユーザ（開発者）が $p$ で定義した任意のユーザ定義の名前を与えることができる．図 1 にある Java プログラムを従来の名前難読化手法を用いて難読化した例を図 2 に示す．なお，この例では，ソースコードを用いて難読化を説明しているが，難読化のプロセスは必ずしもソースコードレベルで行われるとは限らない．

この例では，クラス名 `ImageViewer` を `a` に，フィールド名 `icon` を `aa` に，また，コンストラクタの引数の変数名（ローカル変数名）を `aaa`，`myIcon` を `aaaa` にしている．また，`main` メソッドの引数を `aaa` に，ローカル変数 `i` を `aaaa` に，元 `ImageViewer` 型（`a` に変更されている）のローカル変数を `aaaaa` に変更した(注1)．

```
 1: import javax.swing.JFrame;
 2: import javax.swing.JLabel;
 3: import javax.swing.Icon;
 4: import javax.swing.ImageIcon;
 5:
 6: public class ImageViewer extends JFrame{
 7:   private Icon icon;
 8:   public ImageViewer(String file){
 9:     setTitle(file);
10:     Icon myIcon = new ImageIcon(file);
11:     icon = myIcon;
12:     getContentPane().add(new JLabel(icon));
13:     pack();
14:   }
15:   public static void main(String[] args){
16:     for(int i = 0; i < args.length; i++){
17:       ImageViewer viewer =
18:         new ImageViewer(args[i]);
19:       viewer.setVisible(true);
20:     }
21:   }
22: }
```

図 1 サンプルプログラム（画像ビューワ）
Fig. 1 Sample program (Image Viewer).

上で述べた手法は，比較的実装が簡単であり，難読化後のプログラムの性能劣化が少ないため，広く実用化されている．例えば，Java 言語用には，Dash-O [8]，CodeShield [9]，ZKM [10] のほかにも数多くの名前難読化ツールが存在する．

しかしながら，この手法はシステム定義の名前に適用できないという大きな制限がある．システム定義の名前は汎用ライブラリや API クラスなどで定義される名前であり，$p$ においては使用部のみに現れる．システム定義の名前は，様々な環境で汎用的に利用される．したがって，$p$ においてこれらを難読化してしまうと，$p$ のポータビリティを著しく下げてしまうため，実用的ではない．図 2 の例では，`JFrame` や `JLabel`，`setVisible`，`setTitle` といった名前を別名に変更することはできない．これらは，`javax.swing` パッケージで定義される名前，また，`JFrame` で定義されてい

(注1)：Java 言語の場合，クラス名，メソッド名，フィールド名は異なる名前空間を使用しているため，同じ名前を使用しても問題なくコンパイル，実行を行うことも可能であるが，ここでは説明の明確化のため，異なる名前を使用している．

```
 1: import javax.swing.JFrame;
 2: import javax.swing.JLabel;
 3: import javax.swing.Icon;
 4: import javax.swing.ImageIcon;
 5:
 6: public class a extends JFrame{
 7:   private Icon aa;
 8:   public a(String aaa){
 9:     setTitle(aaa);
10:     Icon aaaa = new ImageIcon(aaa);
11:     aa = aaaa;
12:     getContentPane().add(new JLabel(aa));
13:     pack();
14:   }
15:   public static void main(String[] aaa){
16:     for(int aaaa = 0; aaaa<aaa.length; aaaa++){
17:       a aaaaa = new a(aaa[aaaa]);
18:       aaaaa.setVisible(true);
19:     }
20:   }
21: }
```

図 2　従来の名前難読化
Fig. 2　Conventional name obfuscation.

るメソッド名で，変更すると別の環境で動作しなくなるからである．

このように，従来手法では，システム定義の名前を隠ぺいできないため，攻撃者にプログラム理解の手掛りを残してしまうことになる．したがって，従来の名前難読手法は，難読化としてはあまり強力ではない．

## 3. 提案手法

従来手法の問題点を解決すべく，システム定義の名前を難読化可能な新たな手法を提案する．

### 3.1 キーアイデア

**2.3** で述べたとおり，基本的にシステム定義の名前は変更すべきではないため，プログラム中から別名で呼び出す（使用する）ことはできない．よって，プログラム中の名前を静的に置換する従来の名前難読化法を適用することができなかった．これに対し，本研究では動的名前解決という機構を導入する．プログラム中で使用部に現れる名前をあらかじめ別の名前に変換（暗号化）しておき，プログラム実行時，その名前参照があるたびにもとの名前に戻す機構である．

多くのプログラミング言語では，関数（メソッド）の呼出しや変数参照に現れる名前は，コンパイル時に静的に決定している必要がある．一方，オブジェクト指向言語にはリフレクション機構が備わっている．リフレクションの機能の一つとして，実行時に渡される文字列に基づいて，その文字列を名前としてもつクラスを生成したり，メソッドを呼び出したりすることが可能である．元来，リフレクションは，プラグイン機構の実装など，利用するクラスの詳細が決定していない場合や，生成したクラスの詳細情報を実行時に取得するため等に用いられることが多い．本研究では，このリフレクション機構を動的名前解決を実現する手段として用いる．

### 3.2 動的名前解決

動的名前解決は，プログラム中の使用部に現れる名前を文字列としてあらかじめ暗号化しておき，実行時，必要に応じてもとの名前に戻すアプローチである．今，難読化すべき名前の集合を $N_p$ とする．ただし，$N_p$ は使用部に現れる名前である(注2)．また，各 $n \in N_p$ に，あらかじめ暗号 $E$ を適用し，得られた名前を $n'(= E(n))$ とする．

本論文では，$p$ において $n$ は，(a) オブジェクト生成文におけるクラス名，(b) メソッド呼出しにおけるメソッド名，(c) フィールド参照・代入におけるフィールド名のいずれかに現れるものとする．以下それぞれの場合について，どのように名前解決を行うのかを説明する．

#### 3.2.1 オブジェクト生成文におけるクラス名

名前 $n_c$ がオブジェクト生成文（Java 言語では new）のクラス $c$ の名前として現れる場合，以下のように名前解決を行う．ここで，$n'_c = E(n_c)$ とする．

**手続き** $resolveClass(n'_c)$：

Step C-1:　文字列 $n'_c$ を復号し，もとのクラス名 $n_c$ を得る．

Step C-2:　リフレクションを用いて，名前が $n_c$ であるクラス $c$ を取得する．

Step C-3:　$c$ から新たなオブジェクト $o$ を生成する．

#### 3.2.2 メソッド呼出しにおけるメソッド名

名前 $n_m$ がクラス $c$ のメソッド呼出しに現れる場合，以下のように名前解決を行う．ここで，$n'_m = E(n_m)$，クラス $c$ の名前 $n_c$ は暗号化されており，暗号化後の名前を $n'_c$ とする（すなわち $n'_c = E(n_c)$ である）．

**手続き** $resolveMethod(n'_c, n'_m)$：

Step M-1:　文字列 $n'_c$ に対して，$resolveClass(n'_c)$ のステップ C-1，C-2 を実行して，クラス $c$ を取得する．

Step M-2:　リフレクションを用いて，$c$ が所有するメソッドの集合 $M_c$ を取得する．

(注2)：$N_p$ には名前の定義部は含まない．定義部の名前は **5.1** でも述べるが，従来の静的な名前難読化で難読化可能であるため，提案手法では扱わない．

Step M-3: 文字列 $n'_m$ を復号し，もとのメソッド名 $n_m$ を得る．

Step M-4: 名前が $n_m$ であるメソッド $m$ を $M_c$ から取得する．

Step M-5: メソッド $m$ を実行する．

#### 3.2.3 フィールドの参照・代入におけるフィールド名

名前 $n_f$ がクラス $c$ のフィールド参照（代入）のフィールド名として現れる場合，以下のように名前解決を行う．ここで，$n'_f = E(n_f), n'_c = E(n_c)$ とする．

**手続き** $resolveField(n'_c, n'_f)$ **：**

Step F-1: 文字列 $n'_c$ に対して，$resolveClass(n'_c)$ のステップ C-1，C-2 を実行して，クラス $c$ を取得する．

Step F-2: リフレクションを用いて，$c$ が所有するフィールドの集合 $F_c$ を取得する．

Step F-3: 文字列 $n'_f$ を復号し，もとのフィールド名 $n_f$ を得る．

Step F-4: 名前が $n_f$ であるフィールド $f$ を $F_c$ から取得する．

Step F-5: $f$ に対して，参照または代入を行う．

### 3.3 動的名前解決を用いた名前難読化

提案する名前難読化の手順は以下のとおりである．ただし，以下においてプログラム $p$ は十分にテストされ，問題なく実行できるプログラムとし，$N_p$ はシステム定義の名前を含んでもよい．

[提案する名前難読化の手順]

入力： プログラム $p$，名前集合 $N_p$.

出力： 難読化されたプログラム $p'$.

手順： $p$ に以下の手順を適用する．得られたプログラムを $p'$ とする．

Step 1: 任意の文字列暗号 $E$ を用いて，各名前 $n \in N_p$ を暗号化する．得られた集合を $N'_p = \{n' | n' = E(n)\}$ とする．

Step 2: 各 $n \in N_p$ について，$p$ における $n$ の使用部を以下のように置換える．ただし，$n$ の定義部は置換えを行わない．

$n$ がクラス名 $n_c$ の場合：$resolveClass(n'_c)$ を実現するコードに置き換える．

$n$ がクラス $c$ (名前 $n_c$) のメソッド名 $n_m$ の場合：$resolveMethod(n'_c, n'_m)$ を実現するコードに置き換える．

$n$ がクラス $c$ (名前 $n_c$) のフィールド名 $n_f$ の場合：$resolveField(n'_c, n'_f)$ を実現するコードに置き換える．

```
import javax.swing.JFrame;
        :
public class ImageViewer extends JFrame{
  private Icon icon;
  public ImageViewer(String file){
    setTitle(file);
    Icon myIcon = new ImageIcon(file);
        :
```

↓ Obfuscating names `setTitle` and `ImageIcon`

```
import javax.swing.JFrame;
        :
import java.lang.Class;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
public class ImageViewer extends JFrame{
  public javax.swing.Icon icon;
  public ImageViewer(String file){
    // resolveMethod("JnbhfWjfxfs", "tfuUjumf")
    Class c1 =Class.forName(decrypt("JnbhfWjfxfs"));
    Method m = c1.getMethod(decrypt("tfuUjumf"),
        new Class[]{ file.getClass() });
    m.invoke(this, new Object[]{ file });

    // resolveClass("kbwby/txjoh/JnbhfJdpo")
    Class c2 = Class.forName(
        decrypt("kbwby/txjoh/JnbhfJdpo"));
    Constructor c = c2.getConstructor(
        new Class[]{ file.getClass() });
    Object o2 = c.newInstance(new Object[]{ file });
        :
```

図 3 動的名前解決を用いた難読化（図 1 のコード）
Fig. 3 Proposed name obfuscation with dynamic name resolution.

図 3 に，提案手法による名前難読化例を Java コードで示す．このコードは，図 1 におけるメソッド呼出し（`setTitle`）とオブジェクト生成（`ImageIcon`）を動的名前解決によって難読化した例である．上記二つの名前は，汎用クラス `JFrame` で定義されるシステム定義の名前である．簡単のため，名前文字列の暗号化に鍵 1 のシーザー暗号を用いている．コード中の `decrypt()` は復号ルーチンを示す．

最初のブロックでは，`setTitle` のメソッド呼出しを動的名前解決 resolveMethod() により実現している．Java 言語では，`java.lang.Class` クラスがクラス情報を反映するクラスに相当し，`forName` メソッドに文字列を与えることでクラスを取得できる（ステップ C-2）．また，`java.lang.reflect` パッケージにメソッドやフィールドなどを反映するクラスが含まれる．これらのクラスを使うことでリフレクション機能を用いることができ，得られたクラスから，そのクラスがもつメソッドやフィールドの情報を得ることができる（ステップ M-4, F-4）．この例では，“tfuUjumf” を復号

表 1 DynamicCaller のメソッド
Table 1 Methods implemented in DynamicCaller.

| Method Interface | Supporting Procedure |
|---|---|
| `Object newInstance(Object[] arg, String EclassName)` | resolveClass |
| `Object invoke(Object instance, Object[] arg, String EclassName, String EmethodName)`<br>`Object invokeStatic(Object[] arg, String EclassName, String EmethodName)` | resolveMethod |
| `Object getField(Object instance, String EclassName, String EfieldName)`<br>`Object getStatic(String EclassName, String EfieldName)` | resolveField (reference) |
| `void setField(Object instance, Object value, String EclassName, String EfieldName)`<br>`void setStatic(Object value, String EclassName, String EfieldName)` | resolveField (assignment) |

した文字列 “setTitle” からメソッドを取得し，invoke により実行している（ステップ M-5）．次のブロックでは，ImageIcon オブジェクトの生成を，resolveClass() により行っている．java.lang.Class クラスを用いて，文字列 “javax.swing.ImageIcon” からクラスオブジェクトを取得し，newInstance メソッドによってオブジェクトを生成している（ステップ C-3）．難読化後のコードでは，オリジナルコードのシステム定義名が完全に隠ぺいされていることが分かる．

## 4. 実　　装

提案する難読化手法を自動化するため，Java プログラムを対象とする難読化ツールの実装を行った．実装したツールは，任意の Java クラスファイル（バイトコード）を入力とし，クラス内のすべての名前参照を難読化したクラスファイルを出力する．ここでは，実装の詳細について述べる．

### 4.1 動的名前解決支援クラス DynamicCaller の導入

動的名前解決を支援するため，本ツールでは DynamicCaller という新たなクラスを実装した．このクラスは，表 1 に示す七つのメソッドをもち，難読化対象のプログラム $p$ に組み込まれる．

表中の各メソッドは，**3.2** で提案した動的名前解決の手続きをラップする．暗号化されたシステム定義の名前文字列（EclassName，EmethodName，または EfieldName）を引数にとり，該当する動的名前解決を行う．また，オブジェクト生成またはメソッド呼出しの際に渡される引数は，Object 型の配列 arg を通して渡される．Static の接尾語がついているメソッドは，static メンバを扱う場合のメソッドである．この場合，操作対象のオブジェクト（インスタンス）を省略可能である．

また，DynamicCaller は文字列暗号の復号ルーチンを内包する．現在の実装では DES [13], AES [14], DES-EDE [15], MD5 [16] をサポートしている．

### 4.2 クラスファイルの書換え

難読化対象プログラム $p$ の Java クラスファイルが与えられると，本ツールは以下の手順で $p$ の難読化を行う．

#### 4.2.1 準　　備

まず，$p$ に含まれる全クラスのメンバ（フィールド，メソッド）に対して，アクセス権を public に書き換え，外部クラスからアクセス可能にする．これは，動的名前解決を $p$ の外部クラス DynamicCaller を用いて行うためである．

また，$p$ におけるすべてのフィールドとローカル変数（仮引数以外）の型を Object 型に書き換える．Object は Java 言語におけるすべてのクラスの親クラスにあたるため，$p$ の動作を変えることなく，$p$ に現れる型情報を隠ぺいすることができる．この書換えは，動的名前解決の手法そのものには関係なく，より解析を困難にするための付加的な操作である．例えば，図 1 の 7, 10 行目の型 Icon は，Object に変換して差し支えない．変数宣言における型情報は主に静的に名前解決を行うために用いられるため，動的に名前を解決する本手法においては型情報は必要ない．

なお，これらの変更はプログラムの制約を緩い方へ変更するため，プログラムの挙動に変更を与えるものではない．

#### 4.2.2 名前の暗号化

$p$ において難読化すべき名前を，与えられた暗号アルゴリズムと鍵を用いて暗号化する．これは，**3.3** で提案する難読化手順の Step 1 に相当する．このとき用いる暗号は，DynamicCaller における復号ルーチンで用いる暗号と合致させておく必要がある．本ツールは，$p$ のクラスファイル内の Constant Pool（定数データを保存するために確保される領域）を検索し，クラス名，メソッド名，フィールド名を取得する．次に，各名前について暗号化を行った後，得られた名前

| | (a) Object instantiation | (b) Method invocation | (c) Field reference | (d) Field assignment |
|---|---|---|---|---|
| Original bytecode | :<br>new Foo<br>dup<br>*push args to stack*<br>invokespecial Foo#foo<br>: | :<br>*push instance to stack*<br>*push args to stack*<br>invokevirtual<br>Foo#fooMethod<br>: | :<br>*push instance to stack*<br>getfield Foo#fooField<br>: | :<br>*push instance to stack*<br>*push value to stack*<br>putfield Foo#fooField<br>: |
| Obfuscated bytecode | :<br>*push args to stack*<br>*store args to array*<br>ldc "Bar"<br>invokestatic<br>DynamicCaller#newInstance<br>: | :<br>*push instance to stack*<br>*push args to stack*<br>*store args to array*<br>ldc "Bar"<br>ldc "barMethod"<br>invokestatic<br>DynamicCaller#invoke<br>: | :<br>*push instance to stack*<br>ldc "Bar"<br>ldc "barField"<br>invokestatic<br>DynamicCaller#getField<br>: | :<br>*push instance to stack*<br>*push value to stack*<br>ldc "Bar"<br>ldc "barField"<br>invokestatic<br>DynamicCaller#setField<br>: |

図 4　バイトコード書換えルール
Fig. 4　Rules for obfuscating bytecode.

を文字列（String）として新たに Constant Pool に追加する．なお，名前の頻度解析による攻撃が懸念される場合には，名前の各出現ごとに，異なる暗号鍵を用いる対策が考えられる．

#### 4.2.3　DynamicCaller の呼出し部を埋め込む

$p$ における静的な名前参照を，DynamicCaller を用いて動的に名前解決するように，クラスファイルを書き換える．これは，**3.3** で提案する難読化手順の Step 2 に相当する．

クラスファイルの書換えは，図 4 に示す書換えルールに従って行われる．各ルールは，Java の擬似バイトコードを用いて書かれている．図の上段がオリジナルのバイトコード，下段が変換後のバイトコードである．図中，$p$ において難読化するもとの名前を Foo，fooMethod，fooField，これらを暗号化した名前をそれぞれ Bar，barMethod，barField と仮定している．名前が使用される場所によって，(a), (b), (c) または (d) の変換規則を適用する．

図 4 (a) は，Foo がオブジェクト生成時のクラス名として現れる場合である．もとのバイトコードでは，コンストラクタへの引数をスタックに積んだ後，invokeSpecial によって Foo オブジェクトが生成される．変換後のバイトコードでは，まずスタックに詰まれた引数を一度配列に格納して，Foo の暗号化文字列である"Bar"をスタックに積み，DynamicCaller の newInstance メソッド（表 1 参照）を呼び出す．

スタック上の引数を配列に格納する理由は，これらの引数を DynamicCaller に配列として渡す必要があるからである．図 5 に，スタック上に詰まれた二つの引数 arg1, arg2 を配列に格納するための，バイトコード列及びスタックの状態遷移を示す(注3)．

同様に，図 4 の (b), (c), (d) ではそれぞれ，メソッド呼出しの場合，フィールド参照の場合，フィールド代入の場合のバイトコード書換えルールを示している．ページ数の制限のため，static メンバに関する変換ルールは割愛しているが，ほぼ同様の手順で変換を行う．

### 4.3　難読化の例

図 1 のプログラムに対し実装したツールを適用して難読化を行った．結果を図 6 に示す．この例では，文字列暗号に 56 ビット DES を用い，鍵は 0xefb08fa71fdcc29e としている．なお，説明の簡単化のため，図 6 には Java のソースコードを示しているが，ツールではクラスファイルを直接書き換えることに注意されたい．なお，変数 *o1*, *o2* への代入は説明の簡単化のために加えたもので，バイトコード上では代入は行われていない．

図 6 のコードでは，オリジナルコード（図 1）で使用されたすべてのシステム定義の名前が難読化されている．例えば，図 6 の 4, 5 行目は，図 1 の setTitle(file) に相当する．このとき，図 6 のコードのみを静的に解析して，setTitle(file) が実行さ

(注3)：引数がプリミティブ型の場合は，aastore を実行する直前にラッパクラスに置き換えるバイトコードを挿入する必要がある．また，引数が double 型や long 型ならばスタック上で 2 ワードの領域を使うため，dup2_x1 ではなく，dup2_x2 としなければならない．

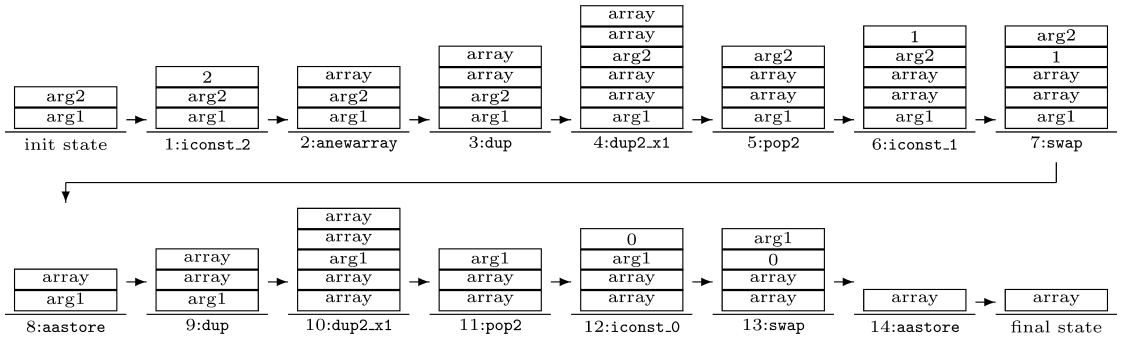



図 5　スタック上の引数を配列に格納する手順
Fig. 5　Storing arguments on operand stack into array.

```
 1: public class ImageViewer extends JFrame{
 2:   public Object icon;
 3:   public ImageViewer(String file){
 4:     DynamicCaller.invoke(this, new Object[] { file },
 5:       "f29f6e89c20c525ca9ad991e2806eec6", "e46a1b932f90efc76d5606a286a3c585");
 6:     Object myIcon = DynamicCaller.newInstance(new Object[] { file },
 7:       "2309ad079c732c3eee007fd16630de66fbe9154c7e632f85");
 8:     DynamicCaller.setField(this, myIcon, "f29f6e89c20c525ca9ad991e2806eec6", "6c25f9e31afd88e4");
 9:     Object o1 = DynamicCaller.invoke(this, new Object[0],
10:       "f29f6e89c20c525ca9ad991e2806eec6", "2ff8414ab86e8975a6439f43d8690008");
11:     Object o2 = DynamicCaller.newInstance(
12:       new Object[] { DynamicCaller.getField(this, "f29f6e89c20c525ca9ad991e2806eec6", "6c25f9e31afd88e4") },
13:       "2309ad079c732c3ea3d8e7d5ada803b78542f3f87b7db5c4");
14:     DynamicCaller.invoke(o1, new Object[] { o2 },
15:       "5459bd4f08f7b753d035fd5b82780cd28aa49371f35be2ea", "878dfc57666c5eb9");
16:     DynamicCaller.invoke(this, new Object[0], "f29f6e89c20c525ca9ad991e2806eec6", "bd627b43d640f316");
17:   }
18:   public static void main(String[] args){
19:     int o1 = ((Integer)DynamicCaller.invokeStatic(new Object[] { args },
20:       "f127001edc194f1a034a6c9b64b22b57326fe908709c0456", "637ff5714bb80cc1cd5952a95803c6ea")).intValue();
21:     for(int i = 0; i < o1; i++){
22:       Object o2 = DynamicCaller.invokeStatic(new Object[] { args, new Integer(i) },
23:         "f127001edc194f1a034a6c9b64b22b57326fe908709c0456", "669658105ab0e482");
24:       Object viewer = DynamicCaller.newInstance(new Object[] { o2 }, "f29f6e89c20c525ca9ad991e2806eec6");
25:       DynamicCaller.invoke(viewer, new Object[] { new Boolean(true) },
26:         "f29f6e89c20c525ca9ad991e2806eec6", "8f98baa6837f9236b117be0cb0a26503");
27:     }
28:   }
29: }
```

図 6　難読化後のサンプルプログラム
Fig. 6　Obfuscated sample program.

れることを理解するのは，ほぼ不可能である．同様に，`getContentPane.add()`，`ImageIcon()`，`Icon` といったシステム定義名も完全にコードから隠ぺいされている．更に動的名前解決は，ユーザ定義のクラス，メソッドの使用部（定義部は不可）に対しても適用できるため，図 1 の `main` メソッドも同様に難読化することができる．

## 5. 評　　価

### 5.1 性 能 評 価

提案手法では，静的な名前参照をすべて動的な呼出しに変換するため，難読化後のプログラムの実行速度の低下が懸念される．そこで，実用規模のプログラムを難読化し，難読化によるオーバヘッドを評価した．

難読化の対象として Jakarta Commons Digester 1.7 [17] を採用した．これは XML で書かれたデータ

ファイルを読み込み，そのデータをメンバとしてもつ Java オブジェクトを生成するための汎用ツールである．比較的簡単に扱えることもあり，多くの Java アプリケーションにおいて設定ファイルの読込み等に使われている．

実験では，難読化したプログラムに Digester チュートリアル [18] の XML ファイルを読み込ませ，オブジェクトが作成されるまでの時間を測定する．測定環境は，Windows XP Professional SP2（Intel Pentium 4 3.00 GHz，1 GB RAM）である．文字列の暗号化には 56 ビット DES を採用した．

実験手順は Digester に含まれる 55 のクラスのうち，ある割合だけランダムに選択して難読化する．こうして得られたクラスファイル群に対し，実行時間を 10 回計測し，その平均を算出することを 10 回繰り返した．難読化するクラスの割合は 0%, 25%, 50%, 75%, 100% とした．難読化に要した時間を表 2 に示す（単位はミリ秒）．

実験結果を表 3，表 4 に示す．表 3 は，難読化後のプログラムにおいて，`DynamicCaller` の各メソッドがそれぞれ平均何回呼ばれたか，またそれらの合計回数を示している．また，表 4 には，難読化後のプログラムの実行時間の平均，中央値，最大，最小（それぞれ単位はミリ秒）を示している．最下段には，難読化後のプログラムのサイズ（すべてのクラスファイルの合計サイズ，単位はバイト）を示す．

すべてのクラスを難読化した場合 (100%) の実行時間の平均は，元プログラム (0%) の 4.11 倍に増加している．`DynamicCaller` のメソッドの呼出し回数が多いほど，実行時間が増加し，かつ，ファイルサイズは 2.20 倍に増加している．

したがって，ハードリアルタイム性が要求されるプログラムや，実行環境に容量制限がある場合には，名前にセキュリティ上の優先順位を付け，難読化を行うべき名前の集合 $N_p$ を絞り込むことが望ましい．また，従来の静的な名前難読化を併用し，ユーザ定義の名前に関しては従来法で難読化するといったアプローチも可能である．

### 5.2 セキュリティ評価

プログラム解析による攻撃を想定し，提案手法のセキュリティ評価を行う．前提として，攻撃者は難読化後のプログラム $p'$ しか入手できないものとする．

#### 5.2.1 静的解析への耐性

まず静的解析に対する難読化の耐性について考察する．静的解析は，プログラムを実行せずにプログラム内の静的な情報のみを解析する攻撃である．提案手法で難読化されたプログラムでは，オブジェクト生成，メソッド呼出し，フィールド参照・代入がすべて動的呼出しに変更されている．更に，呼出しに現れる名前はすべて暗号文字列で置き換えられている．したがって，逆アセンブルや逆コンパイル等のリバースエンジ

表 2　難読化に要した時間
Table 2　Time taken for obfuscation.

| **Obfuscation Rate** | 25% | 50% | 75% | 100% |
|---|---|---|---|---|
| **Time (msec)** | 2501 | 2758 | 3145 | 3539 |

表 3　難読化したプログラムの `DynamicCaller` 呼出し回数
Table 3　# of DynamicCaller calls of obfuscated programs.

| | **0%** | **25%** | **50%** | **75%** | **100%** |
|---|---|---|---|---|---|
| `DynamicCaller.newInstance()` | 0 | 44.6 | 127.7 | 161.7 | 197 |
| `DynamicCaller.invoke()` | 0 | 1814.8 | 3457.5 | 5136.1 | 7114 |
| `DynamicCaller.invokeStatic()` | 0 | 56.5 | 90.6 | 159.3 | 237 |
| `DynamicCaller.getField()` | 0 | 558.3 | 1349.2 | 1889.9 | 2329 |
| `DynamicCaller.getStatic()` | 0 | 6.2 | 9.1 | 15.2 | 30 |
| `DynamicCaller.setField()` | 0 | 135.8 | 377.8 | 537.2 | 671 |
| `DynamicCaller.setStatic()` | 0 | 0.6 | 0.7 | 1.2 | 2 |
| **# of DynamicCaller calls** | 0 | 2616.8 | 5412.6 | 7900.6 | 10580 |

表 4　難読化したプログラムの実行時間
Table 4　Execution time of obfuscated programs.

| | | **0%** | **25%** | **50%** | **75%** | **100%** |
|---|---|---|---|---|---|---|
| **Execution Time (msec)** | **Average** | 348.1 | 985.0 | 1143.7 | 1280.4 | 1429.7 |
| | **Medium** | 347.5 | 982.9 | 1121.2 | 1376.2 | 1414.5 |
| | **Maximum** | 353.0 | 1128.5 | 1365.6 | 1417.6 | 1484.0 |
| | **Minimum** | 346.0 | 822.0 | 878.1 | 965.1 | 1406.0 |
| **File size (byte)** | | 165,831 | 247,533 | 281,221 | 307,815 | 365,314 |

ニアリングを行っても，暗号化されたシンボル名しか抽出されない．これらの暗号文字列を復号をするのは暗号の解読と同じ労力が必要となる．更に，たとえ暗号が解読されたとしても，すべての変数の型が Object 型に置き換えられているため，どのクラスのメンバが呼び出されるのかをプログラムを実行せずに一意に決定することが難しい．このことから，この難読化手法は静的解析に非常に強いことがいえる．

**5.2.2** 動的解析への耐性

次に，デバッガ等を用いてプログラムを実行させ，実行中のプログラム情報を解析する動的解析による攻撃を考える．具体的には，実行時のある時点でプログラムを停止（ブレーク）させ，スタックトレースを出力する攻撃が考えられる．この攻撃により，その時点でスタック上にあるすべてのメソッド名が出力される．これらのメソッドは DynamicCaller から動的に呼び出された後，スタックに積まれるため，オリジナルのメソッド名が露出してしまう．そのため，提案難読化手法は動的解析に対しては何らかの保護が必要である(注4)．

対策として，スタックトレース上に積まれるメソッド呼出しに，ダミーメソッドを混ぜることで改善できる [19], [20]．具体的には (A) DynamicCaller 経由で難読化対象のクラスから副作用のないダミーのメソッドを呼び出す，(B) DynamicCaller 自身からランダムなタイミングで副作用のないメソッドを呼ぶ，の二つの方法が有効である．こうすることで，同じ処理を行ったときのメソッドの実行系列を，手に入れたとしても同一の系列が取得できることはないため，動的解析への耐性を付加することが可能である．

そのほかにも動的解析への耐性を高めるため，アンチデバッガ技術 [21] や，動的解析そのものを防ぐアプローチを併用することが望ましい．

**5.2.3** DynamicCaller に対する攻撃

攻撃者が DynamicCaller を解析することが想定される．DynamicCaller は暗号化文字列の復号ルーチン（及び復号鍵）を含むため，攻撃者がこれらを理解すると難読化を解除する手掛りを与えてしまう．残念ながら，提案手法によって DynamicCaller そのものを難読化することはできない．動的名前解決が再帰的になるからである．したがって，white-box 暗号 [22], [23] 等の別の難読化手法を用いて DynamicCaller を難読化する必要がある．別のアプローチとして，DynamicCaller をハードウェア化することが考えられる [24], [25]．なぜなら DynamicCaller は，与えられたプログラムに依存しない汎用クラスであるためである．ハードウェアとして DynamicCaller を実装することで，DynamicCaller に含まれている復号ルーチン，復号鍵を攻撃者が得ることは困難になる．

このアプローチが有効であるのはセキュアな API の呼出しである．例えば SD カードの読み書きに代表されるセキュアな API の呼出しは，提案手法で難読化し，ハードウェアで実装した DynamicCaller 経由で呼び出すことで，呼出し自体を隠ぺいすることが可能である．セキュアな API の呼出しそのものを隠すことにより，セキュアな API への攻撃を防ぐことができる．

### 5.3 従来の名前難読化手法との比較

図 7 に，従来の名前難読化法（**2.3** 参照）と提案法の適用範囲を示す．従来法では，基本的に名前の定義部を書き換えるため，システム定義のクラス（の使用）を隠ぺいする目的に使えなかった．提案法では，名前の使用部を暗号化し動的呼出しを行うことで，この問題を解決している．なお，従来法と提案法は併用が可能である．その場合は，まず従来法によってユーザ定義の名前を難読化した後，提案法でシステム定義の名前（使用部）を難読化することになる．

また，刑部らはオブジェクト指向言語の多態性（ポリモーフィズム）を用いて，異なるメソッドを同一名にする難読化手法を提案している [26]．この方法も名前難読化の一つに数えられるが，システム定義の名前がオーバライドされている場合に適用できない．

### 5.4 プログラム暗号化との比較

関連研究として，プログラム全体を暗号化してお

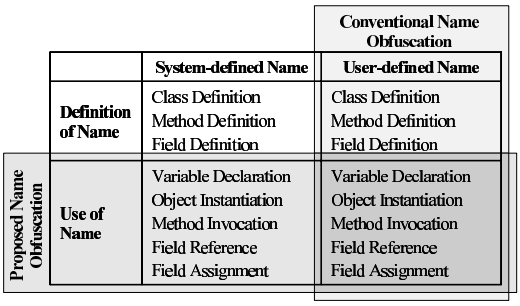



図 7 従来法と提案法の適用範囲
Fig. 7 Coverage of conventional/proposed name obfuscations.

(注4)：なお，この攻撃は自動化が可能であるが，すべての名前を得るためには，何度も入力を変えてプログラムを実行する必要がある．

表 5　提案手法とプログラム暗号化の違い
Table 5　Difference between proposed method and program encryption.

| | **Proposed Method** | **Program Encryption** |
|---|---|---|
| **Attack method** | Peeping stack and giving inputs to appear all names | Peeping memory after decryption |
| **Encryption targets** | Names of methods, fields, classes | whole class file |
| **Class on memory** | Obfuscated | Decrypted (original) |
| **Runtime overhead** | decrypt/invoke a method, refer a field, assign a field, instantiate a object | decrypt/load a class |
| **Additional class** | DynamicCaller | Decrypt ClassLoader |
| **Feasibility** | Research tool | Commercial tool |
| **Applicability** | Any classes | Any classes |

き実行の直前に復号するプログラム暗号化が存在する [27]．提案手法との違いを表 5 に示す．

提案手法とプログラム暗号化の本質的な違いはその攻撃方法である．プログラム暗号化はロード時に復号され，復号された形式のままメモリ上に展開されるため，メモリをのぞき見ることで攻撃することができる．一方，提案手法は必要になったときに復号し，使用後直ちに破棄されるため，スタックを常に監視しておく必要がある．更に一度の実行ではすべての名前を得られないため，通るフローを変えるよう入力を変えて，プログラムを何度も実行する必要がある．このように両者は攻撃方法が異なるため，同時に用いることが可能である．両者を併用することで，より堅牢性を高めることができる．

提案手法とプログラム暗号化は暗号化する対象が異なっている．提案手法では名前単位に暗号化を行うか否かを決定することができる．対してプログラム暗号化はクラスファイル単位でしか暗号化を行うことができない．また，一つの jar ファイル内に暗号化されたクラスとそうでないクラスを混在させることができない．暗号化されたクラスは復号クラスローダで，とそうでないクラスは復号処理を行わないクラスローダでロードする必要があり，クラスローダと jar ファイルは 1：n の関係であるためである．

また，提案手法ではクラスがロードされ，メモリに展開された後においても難読化されたまま（名前が暗号化された状態）であるが，プログラム暗号化手法ではロード時に復号されるため，暗号化前のオリジナルの形式でメモリに展開される．そのため，ロード後のクラスにアクセスすることで，復号鍵を用いずにオリジナルのクラスファイルを得ることが可能である．

実行オーバヘッドはクラスロード時に一度復号すればよいプログラム暗号化に対して，提案手法はメソッド呼出しやフィールド参照など名前を解決する必要があるたびに復号するため，プログラム暗号化の方が実行オーバヘッドが少ない．

そして，提案手法とプログラム暗号化は両者ともにどのようなクラスにも適用可能であり，ツールで自動化することが可能である．また，両者ともにそれぞれの手法を適用したプロダクトを配布するとき，一つのクラスを追加する必要がある．提案手法では `DynamicCaller` であり，プログラム暗号化手法ではロード時に暗号化されたクラスファイルを復号するクラスローダである．

提案手法とプログラム暗号化は暗号化する対象が異なっており，提案手法の方が処理対象をより詳細決定することが可能である．そのため，隠ぺいしたい部分のみを難読化することでオーバヘッドを少なくすることが可能である．また，この二つの技術は相反するものではないため，提案手法で難読化したあと，全体を暗号化することでより理解を難しくすることが可能である．

## 6. むすび

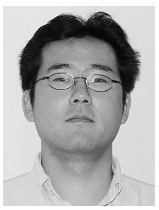
本論文では，オブジェクト指向ソフトウェアを対象として，動的名前解決という機構を用いた新たな名前難読化手法を提案した．プログラムの実行時に暗号化された名前を解決するアプローチをとることにより，従来法によって静的に名前を変更できないシステム定義の名前も難読化することができる．また，提案手法に基づいて，Java クラスファイルを難読化するツールを実装した．更に，実行速度，セキュリティ，他の手法との比較という観点から提案手法を評価し，その特長と限界を考察した．

## 文　献

[1] A. Patrizio, “DVD piracy: It can be done,” Wired

News, 1999. http://www.wired.com/news/technology/0,1282,32249,00.html
[2] A. Orlowski, "DVD jon unlocks iTunes' locked music," The Register, 2003. http://www.theregister.co.uk/2003/11/22/dvd_jon_unlocks_itunes_locked/
[3] C. Collberg, C. Thomborson, and D. Low, "Breaking abstractions and unstructuring data structures," Proc. 1998 International Conference on Computer Languages, pp.28–38, Washington, DC, USA, 1998.
[4] 門田暁人，高田義広，鳥居宏次，"ループを含むプログラムを難読化する方法の提案," 信学論（D-I），vol.J80-D-I, no.7, pp.644–652, July 1997.
[5] 佐藤弘紹，門田暁人，松本健一，"データの符号化と演算子の変換によるプログラムの難読化手法," 信学技報，IT2002-49, 2002.
[6] 福島和英，清水晋作，田中俊昭，"多変数の符号化による難読化方式の提案," コンピュータセキュリティシンポジウム 2004 (CSS 2004) 予稿集，pp.247–252, 2004.
[7] P.M. Tyma, "Method for renaming identifiers of a computer program," United States Patent 6,102,966, 2000. Filed: March 20, 1998.
[8] PreEmptive Solutions, "DashO — The premier Java obfuscator and efficiency enhancing tool," http://www.agtech.co.jp/products/preemptive/dasho/index.html
[9] CodingArt Pty Ltd., "Codeshield java byte code obfuscator," 1999. http://www.codingart.com/codeshield.html
[10] Zelix Pty Ltd., "Zelix Klass Master," 1997. http://www.zelix.com/klassmaster/index.html
[11] Kracker's and BEAMZ, クラッカー・プログラム大全—禁断のシリアルナンバー解析テクニック，データハウス，2003.
[12] B.D. Chaudhary and H.V. Sahasrabuddhe, "Meaningfulness as a factor of program complexity," Proc. ACM 1980 Annual Conference, pp.457–466, 1980.
[13] NBS (National Bureau of Standards), "Data encryption standard (des)," Technical Report FIPS-Pub.46, National Bureau of Standards, U.S. Department of Commerce, Washington D.C., 1977.
[14] NIST (National Institute of Standards and Technology), "Advanced encryption standard (aes)," 2001.
[15] NIST (National Institute of Standards and Technology), "Data encryption standard," 1999. http://csrc.nist.gov/cryptval/des/fr990115.htm
[16] R.L. Rivest, "The md5 message-digest algorithm," Technical Report RFC 1321, MIT LCS and RSA Data Security, Inc., 1992. http://www.rfc-editor.org/rfc/rfc1321.txt
[17] Apache Software Foundation, "Jakarta commons digester," 2005.
[18] P.K. Janert, "Learning and using jakarta digester," 2002. http://www.onjava.com/pub/a/onjava/2002/10/23/digester.html
[19] X. Zhuang, T. Zhang, S. Pande, and H.H.S. Lee, "HIDE: Hardware-support for leakage-immune dynamic execution," Technical Report, GIT-CERCS-03-21, 2003.
[20] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," J. ACM, vol.43, no.3, pp.431–473, 1996.
[21] S. Cesare, "Linux anti-debugging techniques (fooling the debugger)," 1999. http://www.uebi.net/silvio/linux-anti-debugging.txt, accessed 29 Dec. 2003.
[22] S. Chow, P.A. Eisen, H. Johnson, and P. C. van Oorschot, "White-box cryptography and an aes implementation," 9th Annual International Workshop on Selected Areas in Cryptography, SAC 2002, Lecture Notes In Computer Science, vol.2595, pp.250–270, Newfoundland, Canada, 2002.
[23] S. Chow, P.A. Eisen, H. Johnson, and P.C. van Oorschot, "A white-box des implementation for drm applications," 2nd ACM Workshop on Digital Rights Management (DRM2002), Lecture Notes in Computer Science, vol.2696, pp.1–15, Washington, DC, USA, 2003.
[24] 門田暁人，C. Thomborson，"ソフトウェアプロテクションの技術動向（後編）—ハードウェアによるソフトウェア耐タンパー化技術," 情報処理，vol.46, no.5, pp.558–563, 2005.
[25] 小檜山清之，藤山博之，吉武敏幸，"デジタル TV 対応 PC 向け権利保護 LSI," 雑誌 FUJITSU, vol.55, no.6, pp.575–579, 2004.
[26] Y. Sakabe, M. Soshi, and A. Miyaji, "Java obfuscation —Approaches to construct tamper resistant object-oriented programs," J. IPSJ, vol.46, no.8, pp.2107–2119, 2005.
[27] New-Era-Soft, "JEncoder," 2005. http://www.new-era-soft.com/

（平成 19 年 5 月 2 日受付）



**玉田　春昭**　（正員）

平 11 京都産業大・工・情報通信卒．平 13 同大大学院博士前期課程了．同年住商エレクトロニクス（株）入社．平 18 奈良先端大・情報科学・博士後期課程了．同年同大学産学官連携研究員．平 19 同大・情報科学・特任助教．博士（工学）．ソフトウェアセキュリティ，エンタープライズアプリケーション，エンピリカルソフトウェア工学の研究に従事．IEEE，IPSJ 各会員．

**中村　匡秀**　（正員）

平 6 阪大・基礎工・情報卒．平 8 同大大学院博士前期課程了．平 11 同大学院博士後期課程了．同年カナダ・オタワ大学ポスドクフェロー．平 12 阪大・サイバーメディアセンター助手．平 14 奈良先端大・情報科学・助手．平 19 神戸大・情報知能・准教授．博士（工学）．サービス指向アーキテクチャ，Web サービス，サービス競合，ソフトウェアプロテクション等の研究に従事．IEEE, ACM 各会員．

**門田　暁人**　（正員）

平 6 名大・工・電気卒．平 10 奈良先端大・情報・博士課程了．同年同大助手．平 16 同大助教授．平 19 同大准教授．平 15〜16 Auckland 大学客員研究員．博士（工学）．定量的ソフトウェア開発支援，ソフトウェアプロテクション等の研究に従事．IEEE, ACM，IPSJ，JSSST，JSiSE 各会員．

**松本　健一**　（正員）

昭 60 阪大・基礎工・情報卒．平元同大大学院博士課程中退．同年同大・基礎工・情報・助手．平 5 奈良先端科学技術大学院大学・情報科学・助教授．平 13 同大・情報科学・教授．工博．ソフトウェア品質保証，ユーザインタフェース，ソフトウェアプロセス等の研究に従事．IEEE，ACM，IPSJ，JSSST 各会員．