Computer Security Symposium 2013
21 - 23 October 2013

Java 7 におけるAPI 名隠ぺいのための invokedynamic 命令を用いた難読化の試み

福田 収真† 山本 照明‡ 玉田 春昭‡

†京都産業大学大学院先端情報学研究科
603-8555 京都府京都市北区上賀茂本山
i1358095@cse.kyoto-su.ac.jp

‡京都産業大学コンピュータ理工学部
603-8555 京都府京都市北区上賀茂本山

あらまし JavaはAOPなどを利用したリバースエンジニアリングが容易であるため，攻撃される危険性が高い．そこで，攻撃を防止するため，意図的にプログラムの理解を妨げる難読化手法が多数提案されている．その手法の一つに，リフレクションを用いた動的名前解決難読化がある．しかし，この手法には実行速度が著しく低下するという問題がある．本稿では，Java 7で導入されたinvokedynamic命令を用いて動的名前解決難読化を行う．この命令はJVM上で動くスクリプト言語のために用意された命令であり，呼び出すメソッドを文字列で指定できるようになる．この命令を用いることで，同一メソッドの呼び出し回数が5,000回の場合には従来手法に比べ約3倍，高速化されたことを確認できた．

A Prototype of Obfuscation Technique for Hiding API Names with invokedynamic for Java 7 Platform

Kazumasa Fukuda† Teruaki Yamamoto‡ Haruaki Tamada‡

†Division of Frontier Informatics, Graduate School of Kyoto Sangyo University.
Motoyama, Kamigamo, Kita-ku, Kyoto-shi, Kyoto, Japan, 603-8555.
i1358095@cse.kyoto-su.ac.jp
‡Faculty of Computer Science and Engineering, Kyoto Sangyo University
Motoyama, Kamigamo, Kita-ku, Kyoto-shi, Kyoto, Japan, 603-8555.

Abstract Java programs are easy to analyze, since rigorous specification of the JVM. Therefore, various obfuscation techniques are proposed to prevent illegal analysis. The one of the techniques is to hide API names with reflection mechanism. The obfuscation technique is named dynamic name resolution (DNR). Indeed, the technique can hide API names, however, great delays are also introduced. This paper proposes new DNR mechanism with invokedynamic instruction instead of reflection mechanism. The invokedynamic instruction is introduced for script language running on JVM in Java 7 platform. We found that the proposed method is three times faster than conventional method in the case of calling the method 5,000 times through the experiment.

1　はじめに

ソフトウェアの不正解析により，ソフトウェア産業は大きな被害を受けている．例えば，DVD のプレーヤソフトを不正解析することにより，DVD のコピープロテクトが破られ，違法コピーを可能とするツールが公開された事例がある[1]．このような不正解析による被害を防止するために，様々なソフトウェア保護手法が提案されている．その中の技術の一つに，ソフトウェア難読化手法が提案されている．ソフトウェア難読化とは，ソフトウェアを理解が困難なように変換し，不正解析ならびにクラッキングを妨げる技術である．

本稿では，ソフトウェアへのクラッキングを防止するために，invokedynamic 命令を用いた動的名前解決難読化を提案する．動的名前解決難読化とは，従来隠せなかったシステム定義の名前を隠すための手法である．我々の研究グループでは過去に本手法を，リフレクションを用いた手法として提案している[2]．しかし，リフレクションを用いた手法では，API 名を隠せてはいたものの，プログラムの実行速度が著しく低下するという問題があった．そこで本稿では，リフレクション機構に代わり invokedynamic 命令を利用し，動的名前解決難読化のパフォーマンスの低減を試みる．そして，プロトタイプを作成し，従来手法と提案手法のパフォーマンスについて比較する．

2　ソフトウェア保護技術

2.1　ソフトウェア保護の目的

ソフトウェア保護技術では，不正解析やクラッキングを防止することを目的としている．不正解析の手段として，デコンパイル，メモリダンプなどの多数の方法がある。これらの技術を利用することにより，ソフトウェア内部の秘密情報が取り出される．ソフトウェア内部には秘密鍵など秘密情報が含まれており，これら秘密情報の漏えいは，大きな不利益となる．このようなリバースエンジニアリングを防止する為に，ソフトウェア難読化が用いられる．

2.2　従来のソフトウェア保護技術

ソフトウェア保護技術の一つとして，プログラム暗号化が提案されている[3]．プログラム暗号化では，バイトコードへ変換されたプログラム（class ファイル）の一部または，全てを暗号化しておき，プログラム実行時に復号する方法である．この手法は，確かに静的解析では解析が困難である．しかし，JVM では，暗号化されたプログラムは解釈できないため，必ず実行時に復号される．その時に，バイトコードを読み取ることで，プログラムが解析されるという弱点が指摘されている[4]．

2.3　ソフトウェア難読化

難読化とは，あるプログラムを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アクセスから保護するための技術である．一般的に，難読化は，ソフトウェアの一部の情報をより複雑な形に変換することで理解を妨げようとする．例えば，変数名を意味のない名前に変換して理解しにくくする手法[5]や，コントロールフローを複雑にする手法[6]が挙げられる．また，難読化は次の条件を満たす必要がある[2]．

情報 X を隠蔽する難読化をソフトウェア p に対して適用し，難読化後のソフトウェア p' を作成したとき，以下の条件を満たす．

条件 1　ある入力 I に対する，プログラム p の出力を $\mathrm{result}(p, I)$ としたとき，p' の出力は $\mathrm{result}(p, I)=\mathrm{result}(p', I)$ となる．

条件 2　プログラム p から情報 X を取り出すコストを $\mathrm{cost}(p, X)$ としたとき，$\mathrm{cost}(p, X) < \mathrm{cost}(p', X)$ となる．

条件 1 は難読化前と難読化後のプログラムの同一の入力に対して，出力が変わらない，つまり，プログラムの外部的な振る舞いが変わらないことを保証する．条件 2 は，難読化後のプログラム p' から情報 X を取り出すことがプログラム p から取り出すよりも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応されるとは限らない．一般に保護対象となるソフトウェアはソースコードを公開せず，実行コードのみがリリースされることが多いためである．

3 提案手法

玉田らによって提案されている動的名前解決難読化の性能劣化を低減すべく，Java7 で導入された新たなメソッド呼び出し命令 invokedynamic 命令を導入した動的名前解決難読化を提案する．

3.1 動的名前解決難読化

玉田らは，リフレクション機構を用いた動的名前解決難読化を提案した[1]．この難読化手法では，従来の名前難読化では変更できなかったシステム API やライブラリ関数等，システム定義の名前を隠す．しかし，あらゆる環境で汎用的に用いられる名前を単純に変更することは，そのプログラムの可搬性を著しく下げる．玉田らの手法では，クラスの参照，メソッド呼び出し，フィールドの参照・代入に現れる任意の名前を，予め暗号化しておき，実行時にリフレクションを利用して動的に解決する方法を実現している．また，玉田らの実験では，計 10,580 個のクラス名，メソッド名，フィールド名に対して難読化をおこなった結果，4.11 倍の性能劣化が確認された．

3.2 キーアイデア

第 4.1 節で述べた通り，従来手法では，著しく処理速度が低下するという問題がある．原因となっている処理が，リフレクションを用いた処理である．これに対して，本研究では，リフレクション機構の代わりに，invokedynamic 命令を導入する．この命令は JVM（Java 仮想マシン）上で動くスクリプト言語のために Java 7 で導入された．また，この命令はリフレクション機構と同じく，呼び出すメソッドを文字列で指定できるようになる．リフレクション機構では，`java.lang.reflect.Method` を用いメソッドを呼び出す．この方法では， クラスの定義からメソッドの定義を取り出すという余分なコストが掛かり，実行速度の低下へと繋がる[7]，invokedynamic 命令では，ネイティブな実行速度でのメソッド呼び出しが可能となっており，リフレクション機構より高速に動作することが期待できる．

invokedynamic 命令は，JVM 上で動くスクリプト言語（JVM 言語）のために近年実装された命令であるため，著名な Java のデコンパイルツールである Jad[8]や Java Decompiler[9]はクラッシュし，逆コンパイルできない．invokedynamic 命令の本来の目的は，JVM の動的型付け言語の高速化である．従来の JVM では，動的型付けを行うのが不可能であったために JVM 言語特有の処理系を経由しなければ実現できなかった．しかし，この命令により JVM の機能のみで高速で実行できるようになった．動的型付けを行う invokedynamic 命令を Java 言語で用いることにより，Java のソースコードでは表現できない動的型付けを行うことができる．つまり，動的型付け言語ではない Java では invokedynamic 命令の表現が難しい．このことが，デコンパイルの大きな障壁となると考えられる．

3.3 invokedynamic 命令

invokedynamic 命令は，JVM 上で動作するスクリプト言語の性能向上のため導入されたメソッド呼び出し命令である．

従来の JVM では，メソッド呼び出しは，invokestatic，invokespecial，invokeinterface, invokevirtual の 4 つが用意されている．それぞれ，static メソッドの呼び出し，コンストラクタ，private メソッドの呼び出し，インターフェースのメソッドの呼び出し，その他のメソッド呼び出しに使われている．現在の JVM 言語，例えば JRuby，Jython などでは，一度その言語の処理系を挟み，メソッドの実行を行っている．この処理系を挟むことで実行に時間が掛かっている．その問題を解決するために，Java 以外の言語のメソッドを直接実行する invokedynamic 命令が導入された．

invokedynamic の実行には，`MethodHandle` オブジェクト，`CallSite` オブジェクト，ブートストラップメソッドが用いられ，メソッドの実行を行っている．それぞれの次のような役割を担っている．

`MethodHandle` 関数ポインタとして，任意の

メソッドの情報（メソッド名，メソッドを保持するクラス名，引数それぞれの型，返り値の型）を保持する．

CallSite 実行されるメソッドの MethodHandle を保持する．

ブートストラップメソッド invokedynamic 命令が最初に実行されるときに呼び出される．そのときに，命令と CallSite を関連づける．

MethodHandle，CallSite は Java 7 で導入されたクラスであり，java.lang.invoke パッケージに収められている．MethodHandle オブジェクトは関数ポインタとしてメソッドを扱っており，MethodHandle オブジェクトから直接メソッドを呼び出すことも可能である．CallSite オブジェクトは，どのメソッドを呼び出すのか，すなわち，どの MethodHandle を実行するのかを決める．

ここで，対象メソッドを M としたとき，ivokedynamic 命令の実行順は次のようになっている．また，図 1 に実行手順を表したシーケンス図を示す．

Step 1 invokedynamic に CallSite オブジェクトが登録されていれば，Step 6 を実行する．登録されていなければ Step 2 以降を実行する．

Step 2 invokedynamic に必要な情報を登録するため，ブートストラップメソッドを呼び出す．

Step 3 ブートストラップメソッド内で，CallSite オブジェクトを作成する．

Step 4 M を元に MethodHandle オブジェクトを作成する．

Step 5 CallSite オブジェクトと MethodHandle オブジェクトを CallSite の setTarget メソッドによって関連づける．

Step 6 CallSite オブジェクトを使ってメソッド M を実行する．

上記の手順にも示しているように，invokedynamic 命令はまず登録されている CallSite オブジェクトを見つける．見つかればそのまま CallSite 経由でメソッド M を実行する．見つからない場合，CallSite を登録するため，最初の一度だけブートストラップメソッドを実行する．ブートストラップメソッドはどこに定義されていても良いが，ここでは，メソッド呼び出し元のクラスに定義されているものとする．そして，ブートストラップメソッドで CallSite オブジェクト，MethodHandle オブジェクトを作成し，両者を関連付ける．その後，CallSite オブジェクト経由でメソッド M を実行することで，invokedynamic 命令が実行される．

4 invokedynamic 命令による動的名前解決難読化

invokedynamic 命令を用い，メソッド呼び出しに対する動的名前解決難読化の詳細を述べる．本稿で提案する動的名前解決難読化の隠蔽の対象は，メソッド呼び出しである．従来手法のリフレクション機構を用いたメソッド呼び出しを invokedynamic 命令に置き換える．

難読化対象となるソフトウェア（実行ファイル）を p，難読化対象とするメソッド群を $m_i \in M_p$（$1 \leqq i \leqq k$），メソッド m_i の名前を n_i とし，名前の集合を $n_i \in Np$（$1 \leqq i \leqq k$）とする．さらに難読化後のソフトウェア（実行ファイル）を p' とする．このとき，p に提案手法を適応する手順は以下の通りである．

Step 1 任意の文字列暗号 E を用いて，各名前 $n_i \in N_p$ を暗号化する．暗号化された集合を $n_i' \in N'_p$ とする．

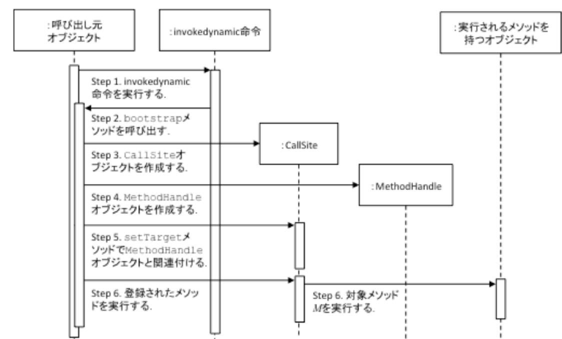

図 1.invokedynamic 実行時の処理

Step 2 各$m_i \in M_p$について，次の動作を行う．

Step 2-1 m_iの実行前に，`setTarget`を呼び出し，メソッドの設定を行う．

Step 2-2 m_iをinvokedynamic命令へ変換する．

Step 3 ブートストラップメソッド内で，$n' \in N'_p$を復号し，実行する．

図 2, 3 に，提案手法による難読化例を Java プログラムのコードとして示す．図 2 の `Integer.bitCount(i)`を提案手法で難読化した結果を図 3 に示す．難読化対象のメソッドは `java.lang.Integer` クラスで定義されており，与えられた数値のビット数を返すメソッドである．説明の簡単化のため，名前の暗号化に鍵 1 のシーザー暗号を用いる．コード中の `decrypt` メソッドは復号を行うメソッドである．また，invokedynamic は JVM の命令であり，コードとして表すことが出来ないため，図 3 では，表現できない箇所を省略している．

難読化後のプログラムには，ブートストラップメソッド，次に実行するメソッド情報を設定する `setTarget` メソッド，次に実行するメソッドの情報を返す `getTarget` メソッド，復号を行う `decrypt` メソッドを追加する．なお，メソッドの情報とは，static の有無，メソッド名，所有クラス名，引数と返り値のそれぞれを文字列とし，それらを配列に代入したものとする．また，`main` メソッド内が書き換えられている．`main` メソッド内で暗号化した文字列用いている．暗号化を行った文字列は，メソッドの情報`{"static", "Integer", "bitCount", "(I)I"}`である．これら文字列を暗号化した結果`{"tubujd", "Joufhfs", "cjuDpvou", ")J*J"}`が図 3 に書かれている．

5 実装

5.1 実装概要

invokedynamic を用いた動的名前解決難読化ツール Shumai を実装する．本研究の要素である invokedynamic は JVM の命令であるため，Java プログラムとして表せない．従って，Shumai の入出力にはともに実行形式のファイル（class ファイル）とした．そして，Shumai では，与えられたクラスファイルを次の 2 つのステップで書き換える．なお，クラスファイルを書き換えるために ASM を利用している[10]．

- メソッド組み込み処理
- `invokedynamic` 命令への変換処理

メソッド組み込み処理では，本手法で必要となるメソッドを変換対象であるクラスファイルに埋め込むための処理である．その後，変換対象の 3 つの命令，invokevirtual, invokestatic, invokeinterface を invokedynamic に変換する．これら以外にもメソッド呼び出しを行う invokespecial 命令がある．invokespecial 命令は，コンストラクタの呼び出し，プライベートメソッドの呼び出しに使われる．これらの呼び出しは区別が困難であるため，今回の提案からは除外している．

5.2 メソッド埋め込み処理

この処理では，対象のメソッド呼び出しが行われている実行形式のファイル(class ファイル)に難読化に必要なメソッドを埋め込む．必要なメソッドは，ブートストラップメソッド，実行するメソッド情報を持つ文字列の配列のフィールドとその setter/getter，そして，復号を行う `decrypt` メソッドである．実行するメソッド情報は，文字列の配列として扱われ，配列の要素に，メソッド呼び出し命令（invokevirtual, invokestatic, invokeinterface），対象メソッド名，対象メソッドの所有クラス名，引数と返り値の型を持つ．

図 3 の例では，図 2 の対象プログラムにシーザー暗号を用いた本手法の例を述べている．図

```
public class Test {
  public static void main(String[] args) {
    int i = 10;
    int j = Integer.bitCount(i);
    System.out.println(j);
  }
}
```

図 2．難読化前のプログラム

3では，図2のプログラムに，対象メソッド情報を格納する文字列配列のフィールドである method，それらの getter/setter，ブートストラップメソッド bms，シーザー暗号の復号メソッドの decrypt を追加している．

5.3 invokedynamic 命令への変換処理

対象メソッドを invokedynamic 命令によって実行するために，3 つのメソッド呼び出し命令 invokevirtual, invokestatic, invokeinterface を invokedynamic 命令へ変換する．なお，invokedynamic に渡す対象メソッドの情報は予め暗号化しておく．具体的な変換手順を以下に述べる．

難読化対象のメソッドを m_i, 暗号化手法を E とする．このとき，暗号化後のメソッド情報を文字列の配列 $E(m_i)$とする．このとき，以下の手順で変換を行う．

```
public class Sample {
  public static String[] method;
  public static CallSite bsm(.....)
 throws Throwable {
    .....
    String[] method = decrypt(getTarget());
       .....
    /*MethodHandle, CallSiteを用いて
       bitCountメソッドを実行する*/
  }
  public void setTarget(String[] method){
    Sample.method = method;.
  }
  public String[] getTarget () {
    return Sample.method;
  }
  public String[] decrypt(String[] args){
    //鍵1のシーザー暗号の復号部
    for(int i = 0; i < args.length; i++){
      char[] str_chars = args[i].toCharArray();
      for(int j = 0; j < args[i].length(); j++){
        str_chars[j] = (char)(str_chars[j]-1);
      }
      args[i] = new String(str_chars);
    }
    return args;
  }
  public static void main(String[] args){
    int i = 10;
    setTarget("tubujd ", "Joufhfs ",
              "cjuDpvou ", ")J*J ");
    int j = invokedynamic();
    System.out.println(j);
    }
  }
```

図 3．難読化後のプログラム

Step 1 対象となるm_iの呼び出し命令が現れるまでメソッドを読み進める．

Step 2 対象メソッド情報m_iを暗号化し，$E(m_i)$を得る．

Step 3 m_iの呼び出し前に，Step 2で得られた$E(m_i)$を引数として，setTarget，メソッドを呼び出す命令を埋め込む．

Step 3 m_iの呼び出し命令をinvokedynamic命令へと変換する．

これらの手順では，invokedynamic命令によって，実行される対象メソッドの暗号化，そして，暗号化された文字列を呼び出すための準備を行う．なお，Step 3は，ブートストラップメソッド内で必要な情報を格納しておくためのものである．

5.4 ブートストラップメソッドでの処理

第 5.3 節の変換処理によって，メソッド情報を暗号化しているため，復号しないと本来のメソッドが呼び出せない．復号処理はブートストラップメソッド内で行う．具体的な処理内容は次の通りである．

Step 1 $E(m_i)$を getTarget メソッドによって取得する．

Step 2 $E(m_i)$をm_iへと復号する．

Step 3 m_iをMethodHandleオブジェクトへ登録する．

Step 4 MethodHandleオブジェクトを CallSiteオブジェクトへ登録する．

```
public class Bubblesort {
public static List<Integer>
        sort(List<Integer> list) {
    for(int i=0; i < list.size()-1; i++){
      for(int j=0; j < list.size()-i-1; j++){
        if(list.get(j) > list.get(j+1)) {
          Collections.swap(list, j, j+1);
        }
      }
    }
    return list;
  }
}
```

図 4．実験対象のプログラム

表 1.評価結果

配列の要素数		1	20	50	60	100	150
swap メソッドの呼び出し回数		1	100	500	1,000	2,500	5,000
実行時間 (ns)	Original	53,000	219,000	726,000	1,253,000	2,897,000	6,127,000
	Reflection DNR	745,000	5,722,000	14,132,000	18,444,000	32,466,000	61,367,000
	Invokedynamic DNR	9,014,000	9,357,000	9,862,000	10,766,000	12,205,000	17,107,000
1 回の呼び出し当たりの実行時間 (ns/回数)	Original	53,000	2,190	1,452	1,253	1,159	1,225
	Reflection DNR	745,000	57,220	28,264	18,444	12,986	12,273
	invokedynamic DNR	9,014,000	93,570	19,724	10,766	4,882	3,421
遅延倍率 (DNR/Original)	Reflection DNR	14.057	26.128	19.466	14.720	11.207	10.016
	invokedynamic DNR	170.075	42.728	13.584	8.592	4.213	2.792

Step 5 `CallSite`オブジェクトによってm_iを実行する.

これらの処理により，対象メソッドの情報を隠ぺいしつつ，呼び出せるようになる．ただし，実行時に復号していることから，対象メソッドの情報は動的解析により暴露する可能性がある.

6 ケーススタディ

6.1 実験

提案手法の有意性を評価するため，従来手法との比較を行う．対象プログラムとして，任意の長さのリストに対してバブルソートを行うプログラム（図 4）を用意する．このプログラムの`Collections`クラスの static メソッド，`swap`を提案手法と従来手法を用いて隠ぺいした．得られた 3 つのプログラム（元のプログラムと両難読化を適用した結果のプログラム）に，ランダムな数値が格納されたリストを渡した．リストの長さは 5, 20, 50, 60, 100, 150 の 6 パターンで実験を行った．それぞれ，`swap` メソッドの実行回数は 1, 100, 500, 1000, 2500, 5000 であった．それぞれ 100 回実行し，計測結果の平均を示す．なお，リストに入れられた数値は乱数であるが，同じシードを指定したため，リストの長さが同じ場合には，同じ値が入るようにした.

結果を表 1 に示す．表 1 では，元のプログラム（図 4）を Original，従来手法を Reflection DNR，提案手法を invokedynamic DNR としている．表 1 の実行時間の行には，それぞれの計測結果を示している．また，それらの結果をそれぞれのメソッド呼び出し回数で正規化することにより，1 メソッド呼び出しの実行時間を示す．また，図 5 に 1 メソッドあたりの実行時間のメソッド呼び出し回数の変化を表したグラフを示す．横軸はメソッド呼び出し回数を表しており，縦軸は実行時間（ナノ秒）を対数で示している.

メソッドの呼び出し回数が 1 回の場合では，提案手法は，オリジナルに比べ約 170 倍の遅延，従来手法とくらべ約 12 倍の遅延が見られる．しかし，メソッドの呼び出し回数が 500 回の場合では，オリジナルに比べ約 13 倍，従来手法と比べわずかに速度が向上していることがわかる．そして，メソッドの呼び出し回数が 5000 回の場合では，従来手法より約 3 倍程度，高速化されている.

6.2 議論

第 6.1 節の評価で得られた結果から提案手法はメソッドの実行回数が増えるほど，すべての手法において，1回のメソッド呼び出し当たりの実行時間が小さくなっていることがわかる．特に invokedynamic DNR は Reflection DNR に比べ，より遅延が小さくなっている．これは，JVM により呼び出しが最適化されているためである．さらに，invokedynamic の場合は，JVM の最適化以外にも，invokedynamic 自体の最適化も行われているため，リフレクションに比べ

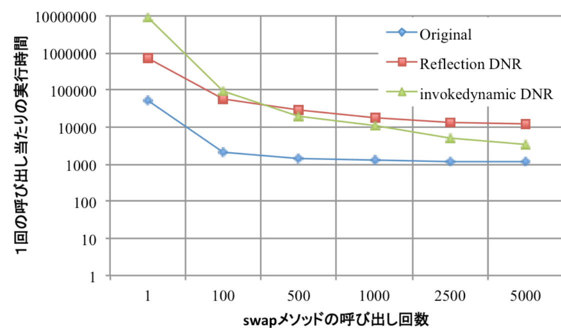

図 5. 1 メソッド当たりの実行時間のメソッド呼び出し回数ごとの推移

て，呼び出し回数が多くなればなるほど，実行時間が短くなっている．実行回数が 1 回のメソッドのみを対象に難読化を加える場合は，170 倍の著しい性能劣化が見られる．しかし，一方で，何回も呼び出されるようなメソッドに対してであれば，本手法を適用することで，性能劣化を抑えられる．また，回数が少ないメソッド呼び出しに対しては，従来手法を適応し，回数が多いメソッド呼び出しに対しては，本手法を適応することで，性能劣化を軽減できる．

6.3 攻撃への耐性

提案手法は，呼び出すメソッドを隠ぺいする方法として，静的解析には非常に強い．呼び出すメソッドを見つけ出すには暗号化された情報を解読しなければならないため，暗号解読と同じ労力が必要なためである．しかし，今回の例では，復号に必要な情報もプログラム内に含めている．そのため，復号に必要な情報を隠すことが，この手法で解決しなければならない点である．

一方， 動的解析に対しては更なる対策が必要である． なぜなら，実行時には暗号化されたメソッド情報が復号される．そのため，特定のタイミングでメモリを探索されると，復号されたメソッド情報が暴露する可能性がある．そのため，他の動的解析に対抗する難読化手法を併用するなど対策が必要である．

7 まとめ

本稿では玉田らによって提案されている動的名前解決難読化を，実行遅延の低減のため，リフレクションの利用から，invokedynamic 命令の利用に置き換えた． invokedynamic 命令を用いることで，対象メソッドの実行回数が 5,000 の場合に，従来手法に比べ，約 3 倍の高速化が確認できた．

今後の課題として，より実践的なプログラムに対して提案手法を適用し，有効性を確認することが挙げられる．また，動的解析に対する攻撃耐性を高めるための議論も必要である．

参考文献

[1] “DeCSS for Linux and DVD - Carnegie Mellon University," http://www.cs.cmu.edu/~dst/DeCSS/ (Last Access: 2013/08/19)

[2] 玉田 春昭，中村 匡秀，門田 暁人，松本 健一，``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," 電子情報通信学会論文誌 D, Vol.J90-D, No.10, pp.2723-2735, October 2007.

[3] Douglas J. Albert and Stephen P. Morse. Combatting software piracy by encryption and key management. Computer, Vol. 17, No. 4, pp. 68–73, April 1984.

[4]門田暁人，Clark Thomborson, “ソフトウェアプロテクションの技術動向(前編),” 情報処理学会学会誌，Vol. 46, pp. 431–437, April 2005.

[5] Jien-Tsai Chan, Wuu Yang: “Advanced Obfuscation Techniques for Java Bytecode”, Journal of Systems and Software, Vol.71, issues 1-2, pp.1-10, April 2004.

[6] 門田，高田，鳥居, "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.

[7] ``JDK 7 の新機能:Java 仮想マシンにおける動的型付け言語のサポート ", http://www.oracle.com/technetwork/jp/articles/java/dyntypelang-427994-ja.html (Last Access: 2013/08/19)

[8] Pavel Kouznetsov, “jad —the fast java decompiler,” February 2004. http://www.kpdus.com/jad.html. (Last Access: 2013/08/19)

[9] “Java Decompiler,” http://java.decompiler.free.fr. (Last Access: 2013/08/19).

[10] Eric Bruneton, “ASM 4.0 A Java bytecode engineering library,” http://download.forge.objectweb.org/asm/asm4-guide.pdf. (Last Access: 2013/08/19).