Java クラスファイルに対する電子透かしの実現

2002MT013 早川 覚太郎　　2002MT091 山川 高明
指導教員　真野 芳久

1　はじめに

Java 言語はオブジェクト指向言語であり、プログラム部品であるクラスファイルは再利用性が高い。これによる利便性は非常に高いが、その反面クラスファイルの盗用が容易になる原因にもなっている。また、Java の特性からクラスファイルの逆コンパイルが容易であるため、盗用の危険性はさらに高まる。本研究では、Java プログラムの盗用を抑止することを目的としてクラスファイルに対する電子透かしの埋め込み手法を提案する。

実装には Java クラスファイルを扱うためのライブラリである BCEL[1] を用いる。

2　電子透かし

電子透かしとは、デジタルデータに対して予め密かに著作権情報などを埋め込んでおき、必要な場合に透かし情報を取り出して盗用の証明をする技術である。電子透かしをプログラムに適用する場合、以下の要件を満たすことが望まれる。

- 透かし挿入後も、プログラムの時間・空間的な効率が大きく低下しない
- 難読化などの変換を適用しても、透かしが壊れることなく取り出すことができる
- 透かしの上書き攻撃に対して耐性がある

3　関連研究

電子透かしの対象メディアとしては、画像データ、音声データ、テキスト文書などが主流である。しかし近年、プログラムに対する電子透かしの研究も盛んに行なわれている。

門田ら [4] は、Java クラスファイルの数値オペランド、オペコード部分に透かし情報を埋め込む手法を提案している。この手法ではソースコードに対してダミーメソッドを追加することによって透かし情報を埋め込むための領域を確保する。コンパイル後のクラスファイル中の数値オペランド、オペコード部分に透かし情報を埋め込む。この手法ではクラスファイル部品単体が盗用された場合でも透かし情報挿入部分を調べることにより簡単に透かし情報を復元できる。この電子透かし埋め込み手法は静的透かしである。また、この手法に基づいた埋め込み、取り出しツールも公開されている。

門田らの手法では、攻撃への耐性に関してはある程度考慮されている。しかし、福島ら [5] はダミーメソッドの特定による攻撃に弱いということを指摘している。

4　電子透かしの埋め込み手法

4.1　同値命令列の置き換えによる埋め込み

Java 仮想マシンの命令は 1 バイトのオペコードと命令ごとに異なる長さのオペランドから成っている。命令は 200 種類程だが、実行効率重視のために使用頻度の高いプッシュ、ロード、ストア命令などにはオペランドを省略した専用命令が用意されている。このため、専用命令を汎用命令に置き換えることで、透かし情報を埋め込むことができる。

表 1 に同値命令の例を示す

表 1　同値命令の例

	専用命令	汎用命令	種類数
例 1	iconst_<i>	bipush i	7
例 2	iload_<n>	iload n	4
例 3	lload_<n>	lload n	4
例 4	fload_<n>	fload n	4
例 5	dload_<n>	dload n	4

iconst_<i>命令は -1〜5 までの整数をオペランドスタックにプッシュする命令であるが、bipush 命令で代用することもできる。この置き換えにより、1 箇所当たり $\lfloor \log 2 \rfloor = 1_{(bit)}$ の透かし情報を埋め込むことができる。

また、バイトコード中の命令列を同値な命令列と置き換えることでも透かし情報の埋め込みが可能である。

表 2 に同値命令列の例を示す。

表 2　同値命令列の例

	基本形	同値命令列 1	同値命令列 2
例 1	istore i iload i	dup istore i	
例 2	lconst l lload n ladd	lload n lconst l ladd	
例 3	iadd	ineg isub	iconst_m1 imul isub

表 2 の例 1 で使用している istore、iload 命令はそれぞれ int 型の値をローカル変数にストア、ロードする命令である。また、dup 命令はオペランドスタック上の値を複製する命令である。値をストア後にロードした場合と、複製後にストアした場合はどちらもオペランドスタックの状態が同じになるため、この 2 つの命令列は同値であると言える。この置き換えにより、1 箇所当たり $\lfloor \log 2 \rfloor = 1_{(bit)}$ の透かし情報を埋め込むことができる。

4.2 同一命令列のサブルーチン化による埋め込み

本研究では透かし情報をクラスファイルに埋め込むことを検討しているが、埋め込み後のクラスファイルを逆コンパイルした場合、透かし情報が消失する可能性は高い。このため、逆コンパイルそのものを防ぐことを検討した。

Java 仮想マシンの命令は 200 種類程であるため、バイトコード中にはオペコード、オペランド共に同一の命令、命令列が複数存在する。これらの命令列をサブルーチン化し、サブルーチンを並び替えることによって透かし情報を埋め込むことができる。サブルーチンの数が n 個の時、$n!$ 通りのパターンを表現できるため、埋め込むことのできる情報量は $\lfloor \log n! \rfloor \simeq \lfloor n \log n \rfloor_{(bit)}$ となる。

サブルーチン化を行なう場合に留意すべき制約として、「ある命令がいくつかの異なった実行経路によって実行される場合、通ってきた経路に関わらず該当命令の実行に先だってオペランドスタックの深さ及び値の型は同じものとなっていなければならない。」という Java 仮想マシンコードにおける構造上の制約がある。この制約は実行時にベリファイヤによって検証される。このため、上記の制約を満たすようにサブルーチン化を行なう必要がある。

図 1 に透かし情報埋め込みの流れを示す。

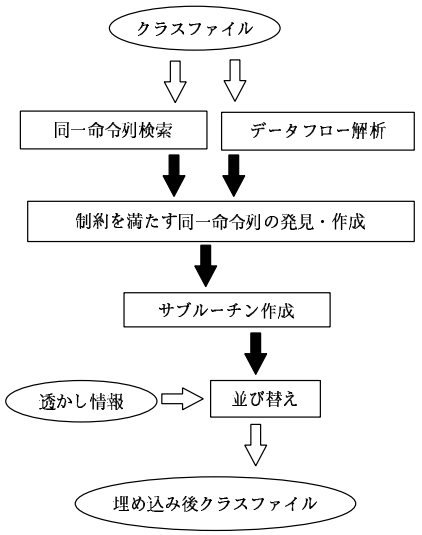

図 1　透かし情報埋め込みの流れ

まず、クラスファイル中の各メソッドごとに同一命令列を検索する。また、各メソッドのデータフロー解析を行なって、オペランドスタック状態表を作成する。次に、制約を満たす同一命令列の発見・作成を行なう。最後に、同一命令列のサブルーチンを作成し、並び替えることで透かし情報を埋め込む。

データフロー解析ではベリファイヤと同様の解析を行なって、オペランドスタック状態表を作成する。また、入力クラスファイルは正規のベリファイヤで検証済みであると仮定する。このオペランドスタック状態表を用いて制約を満たす同一命令列か否かを判断し、満たさない場合は修正を試みる。

表 3 にオペランドスタック状態表の例を示す。

表 3　オペランドスタック状態表の例

コード	スタックの状態	深さ
0: iconst_0		0
1: istore_1	int	1
2: iconst_0		0
3: iconst_0	int	1
4: istore_1	int int	2
5: istore_1	int	1

表 3 の例では、1: istore_1 と 4: istore_1 は同一命令であるが、オペランドスタックの状態が異なるため修正を加える必要がある。この例では、1: istore_1 に int が 1 つ足りないため、0: iconst_0 の前で int をプッシュ、1: istore_1 の後ろでポップすることでオペランドスタックの状態が等しくなる。

スタックの状態で一方の全体が他方の上部と一致する場合、オペランドスタックの修正が可能である。**図 2** に修正可能な例及び修正不能な例を示す。

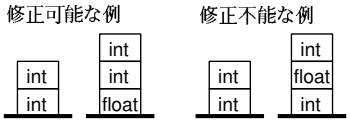

図 2　オペランドスタック修正可否の例

オペランドスタックの修正は深さが小さい方の同一命令列に対して以下の手順で行なう。

1. 同一命令列の先頭の命令 (h とする) 以前でオペランドスタックの深さが 0 となる命令 (b とする) を探し、位置を保持する。
2. 同一命令列の終端の命令 (t とする) 以降でオペランドスタックの深さが 0 となる命令 (a とする) を探し、位置を保持する。
3. h-b 間及び t-a 間に特殊な命令が存在する場合は修正に失敗する。
4. b の前及び a の後ろにそれぞれ対応する型のプッシュ、ポップ命令を追加する。

特殊な命令の例を以下に示す。

- 他の同一命令列として使用する命令
- 分岐先になっている命令
- 分岐命令
- メソッド呼び出し命令

サブルーチン化には特殊な命令を用いるが、通常のコンパイラはこの命令を finally 節の実装にのみ用いる。このため、本手法を用いて埋め込みを行なう場合、多くの逆コンパイラが正常に動作しないという利点と共に、埋め込み後のバイトコードが不自然になるという欠点も生ずる。この点に関しては対策を検討する必要がある。

4.3 ダミーメソッド実行による耐性の強化

前述の門田らによる手法の欠点を解消するため、ダミーメソッドを実行させることでダミーメソッドの特定を困難にする。

以下に、ダミーメソッドの挿入手順を示す。

1. クラスファイルへのダミーメソッドの追加
2. クラスファイル中のあるメソッドの繰り返し文中にダミーメソッドの呼び出しを追加 (クラスファイル全体で繰り返し文が無い場合は適当な位置に追加)

この手法では、攻撃者によるダミーメソッドの発見をより困難にさせるためにダミーメソッドの呼び出しを繰り返し文中に追加する。しかし、そのままでは繰り返し文の中でダミーメソッドを実行してしまい、実行速度を著しく低下させてしまう。したがって、条件文などを使って、1 回目は必ず実行させ、あとは繰り返し回数に応じて数回だけ実行させる。条件文の例を**図 3** に示す。

```
if(j == k){ method(); k = 2 * k;}
j++;
```

図 3　挿入する呼び出しの例

図 3 の例では繰り返し回数を n 回として、ダミーメソッド (method) を $\lfloor \log n \rfloor$ 回実行する。`k=2*k` の部分を変更することにより、ダミーメソッドの実行回数を変化させることができる。

次にダミーメソッドの呼び出しの挿入について述べる。Java の繰り返し文には、for 文、while 文、do-while 文がある。**図 4** に for 文と do-while 文のバイトコードの例を示す。図の上部がソースコード、下部はバイトコードを表わす。

```
for(int i=0;i<10;i++)
      ;

0  : iconst_0
1  : istore_1
2  : iload_1
3  : bipush 10
5  : if_icmpge 14
8  : iinc 1,1
11 : goto 2
14 : return
```

```
int i=0;
do{i++;
 }while(i<10);

0  : iconst_0
1  : istore_1
2  : iinc 1,1
5  : iload_1
6  : bipush 20
8  : if_icmplt 2
11 : return
```

図 4　for 文と do-while 文のバイトコードの例

図 4 を見ると、for 文では `11 : goto 2`、do-while 文では、`8 : if_icmplt 2` が繰り返し文の折り返し部分になっている。そして、それぞれのオペランドが繰り返しの開始の位置を示している。この折り返し部分の直前に呼び出しを挿入する。

しかし、この判別方法は 1 つの例であり、全てのコンパイラが繰り返し文に対してこのようなバイトコードを出力するとは言い切れない。したがって、この繰り返し文の判定に関しては、更なる検討が必要である。

5　評価実験

各手法をクラスファイルに適用した場合の時間・空間的効率の変化、攻撃に対する耐性に関して評価実験を行った。

本章の表中では表幅の都合上、各手法をそれぞれ同値、サブ、ダミーと表現する。

実験準備

実験に使用するクラスファイルとして、j2sdk1.4.2_08 に付属しているサンプルのクラスファイルから無作為に 10 個のクラスファイルを選択した。これらのクラスファイルに透かし情報として「hayama」を埋め込み、実験を行なった。同値命令列による手法では、**表 1** の 23 種類の同値命令を置き換え対象として実験を行なった。また、ダミーメソッドによる手法ではダミーメソッドとして、バイトコードのサイズが 100 バイト程度のメソッドを使用した。

クラスファイルサイズの変化

透かし情報を埋め込む前後でのクラスファイルサイズの変化を**表 4** に示す。各数値は 10 クラスファイルの平均である。

表 4　クラスファイルのサイズ変化

埋め込み手法	同値	サブ	ダミー
埋め込み前 (byte)	4377		
埋め込み後 (byte)	4504	4659	4654
増加量 (byte)	125	282	276
増加割合 (%)	3.0	6.4	6.3

各手法とも増加量の割合が 3〜6% となっているため、透かし情報埋め込みのコストとしては許容範囲内であると考えられる。

実行時間の変化

透かし情報を埋め込む前後での、実行時間の変化を**表 5** に示す。実行時間の測定はそれぞれ 5 回ずつ行ない、平均実行時間を記録した。表の各行は上から順に、実験対象のクラスファイル、元クラスファイルの実行時間、同値命令列による手法適用後の実行時間、サブルーチン化による手法適用後の実行時間、ダミーメソッドによる手法 (実行回数抑制あり) 適用後の実行時間、ダミーメソッドによる手法 (実行回数抑制なし) 適用後の実行時間を表わす。

表 5　実行時間の変化

クラスファイル	A	B	C	D
元実行時間 (秒)	31.0	28.3	20.5	48.9
同値 (秒)	33.0	28.7	20.5	48.9
サブ (秒)	45.6	31.7	37.1	48.9
ダミー 1(秒)	33.3	28.2	22.1	48.9
ダミー 2(秒)	34.3	28.3	22.4	48.8

同値命令列による手法及びダミーメソッドによる手法では元クラスファイルの実行時間との差がすべて 1 割未

満となっているため、透かし情報埋め込みのコストとしては許容範囲内であると考えられる。サブルーチン化による手法では、クラスファイル A、C において高い数値となっている。この理由としては、通常、サブルーチンは特殊な場合においてのみ使用され、使用頻度が低いため Java 仮想マシンで実行の高速化がされていないということが考えられる。

難読化攻撃に対する耐性

クラスファイルの難読化には SandMark[3] を使用した。まず、SandMark を用いて 37 種類の難読化を透かし情報埋め込み後の各クラスファイルに適用した。その後、難読化適用後のクラスファイルから透かし情報を取り出せるかどうか実験を行なった。SandMark には、門田らの手法を元にした透かし情報埋め込みアルゴリズムも実装されているため、この手法についても実験を行なった。

実験結果を**表 6** に示す。すべてのクラスファイルから透かし情報が完全に取り出せた場合を○、一部が変化して取り出せた場合を△、半分以上が変化、あるいは取り出しに失敗した場合を×で表現している。

表 6　難読化攻撃に対する耐性

	×	△	○
同値	5	6	26
サブ	1	0	36
ダミー	3	5	29
門田	4	4	29

サブルーチン化による手法では失敗数が 1 と他の手法に比べてかなり高い結果となり、強い耐性を持つことが確認された。

また、他の手法において取り出しに失敗した難読化の大半は変数の追加や変換、新しい命令の追加などを行なう。このため、透かし情報埋め込み箇所の誤認や、透かし情報を埋め込んだ命令の変更などにより、取り出し時の透かし情報が変化する結果となった。

再コンパイル攻撃に対する耐性

逆コンパイルには広く用いられている逆コンパイラの 1 つである Jad[2] を使用した。まず、透かし情報埋め込み後の各クラスファイルを逆コンパイルしてソースファイルに復元した。次に、復元したソースファイルを再コンパイルし、再コンパイル後のクラスファイルから透かし情報を取り出せるかどうかの実験を行なった。

実験結果は、同値命令列による手法がすべて失敗、ダミーメソッドによる手法はすべて成功と両極端な結果となった。また、サブルーチン化による手法では、各クラスファイルの逆コンパイル中にエラーが出て逆コンパイルに失敗するか、逆コンパイルが終了した場合でも正しいソースファイルが得られない結果となった。

図 5 にサブルーチン化による手法で埋め込みを行なった前後のクラスファイルをそれぞれ逆コンパイルした例を示す。左が埋め込み前、右が埋め込み後のソースコードを表わす。

```
for(int k = 0; k < 10; k++) {
    k = 1 + k;
    k = 1 + k;
}
```

```
    0;
_L7:
    j;
    JVM INSTR icmpge 96;
     goto _L3 _L4
_L3:
    break MISSING_BLOCK_LABEL_82;
_L4:
    break; /* Loop/switch isn't completed */
    true;
     goto _L5
    if(true) goto _L7; else goto _L6
_L6:
```

図 5　埋め込み前後の逆コンパイル例

図 5 右のソースコードを修正して再コンパイルするのは非常に困難であるため、逆コンパイルに対して強い耐性を持つことが確認された。

6　おわりに

本研究では、Java クラスファイルの盗用を抑止するための手法として、クラスファイルへの電子透かしの適用を提案した。

同値命令列の置き換えによる埋め込み手法は、耐性に関して更なる検討が必要であるが、時間・空間的な効率変化の面においては実用的な埋め込み手法になったと言える。

同一命令列のサブルーチン化による埋め込み手法は、評価実験の結果から、強い耐性を持つことが確認された。埋め込み方法を工夫することでさらに良い手法になると考えている。

門田らによる手法の改善では、ダミーメソッドを実行させてダミーであることを分かりにくくするという当初の目的が達成できた。また、ダミーメソッドの実行回数を抑制することで時間的効率の低下はほぼ無視できるものとなった。

参考文献

[1] Apache Software Foundation: BCEL, http://jakarta.apache.org/bcel/.

[2] P. Kouznetsov: Jad - the fast java decompiler, http://kpdus.com/jad.html.

[3] C. Collberg, G. Myles, A. Huntwork: “SandMark - A Tool for Software Protection Research”, *IEEE Security and Privacy*, Vol.1 No.4, pp.40-49 (Jul./Aug. 2003).

[4] 門田 暁人, 松本 健一, 飯田 一, 井上 克郎, 鳥居 宏次: “Java クラスファイルに対する電子透かし法”, 情報処理学会論文誌, Vol.41 No.11, pp.3001-3009 (Nov. 2000).

[5] 福島 和英, 櫻井 幸一: “ソフトウェア透かしにおける個人識別情報埋め込み位置の難読化”, 暗号と情報セキュリティシンポジウム (SCIS2003), pp.1053-1058 (Jan. 2003).

[6] T. Lindholm and F. Yellin 共著, 村上 雅章 訳: “Java 仮想マシン仕様 第 2 版”, ピアソン・エデュケーション (May 2001).