難読化の冗長性を利用した電子透かしの実現

2002MT007 伊達 一貴　　2002MT027 神戸 隆志　　2002MT037 河合 秀哉
指導教員　真野 芳久

1　はじめに

ソフトウェアに対する攻撃としては、リバースエンジニアリングを用いた解析行為、ソフトウェアの違法コピー、ソフトウェアの改変を行うタンパリング等が挙げられる。

その防御策として、解析を困難にする難読化や、コピーであることを証明する電子透かしがある。ソフトウェアプロテクションという点からすると、ソフトウェアの解析と違法コピーの双方に対応できるように、難読化と電子透かしの挿入は併用して行うことが望ましい。ここでは、難読化と透かしの挿入を併用する方法の一つとして、難読化時の冗長性を利用して透かしを挿入するいくつかの手法を提案し実現する。

2　関連研究

2.1　難読化

攻撃者にプログラムを解読されないように、プログラムを意図的に読みにくくする技術のことである。これにより、プログラム中のアルゴリズムやデータ構造が知られるのを防ぐことができる。大規模プログラムになるほどプログラム全体を読みにくくする余地があり、攻撃者がプログラムを解析するコストをプログラムの価値と比べ高くすることで事実上解読を不可能にする。

2.2　電子透かし

画像や動画、音声などのディジタルデータに、画質や音質にはほとんど影響を与えずに秘匿情報や著作権情報を埋め込む技術のことである。透かしデータが埋め込まれた画像などのデータは、一見すると元のデータと変わりないように見え、透かしの埋め込まれたプログラムは元のプログラムと同じ処理を行う。

2.3　opaque 述語

プログラム記述者はあらかじめ真理値を判定できるが、第三者 (攻撃者) から見た場合、一見どのような真理値をとるのかわからない述語である。簡単な例としては、j*j*(j+1)*(j+1)%4==0 は必ず真になる。

3　研究概要

ソフトウェア電子透かしはプログラムの冗長性に透かしを埋め込む。透かしを埋め込むために、プログラムに元からある冗長性を利用する、あるいは冗長性を設けてそこに埋め込む等が考えられる。そこで、本研究では難読化の際に生じる冗長性に着目し、その冗長性を利用して電子透かしの挿入を行う。

本手法は、透かし挿入のためにあらかじめ冗長性を設ける必要がないので、透かしの挿入によってプログラムの実行効率やサイズ等が増加しない利点がある。

表 1　透かしの埋め込みを検討した難読化一覧

構造	難読化名	難読化の説明
Control	Contrl Flow Scramble	プログラムを複数のブロックに分割し、制御変数を用いて各ブロックの実行順序を難読化する
	Interleave Methods	複数のメソッドを 1 つのメソッドにまとめる
	Extend Loop Condition	ループ条件に opaque 述語を追加で記述する
	Add Redundant Operands	opaque 変数を導入して数式を複雑化する
	Clone Methods	あるメソッドを同じ仕様のまま複数のメソッドに変換する
Data	Split Variable	bolean 変数を複数の変数で表現する
	Change Encoding	データの符号化とそれに伴う演算の変換を行う
	Merge Scalar Variables	2 つ以上の変数を 1 つの拡張した変数にまとめる
	Restructure Array	配列の分割、併合、高次化、低次化などを行う
	Reorder Arrays	配列の index を $f(i)$ に置き換え、それによって配列の添字を難読化する
Class	Modify Inheritance Relations	クラスの分割、ダミークラスの挿入、クラスの false refactoring 等の継承関係の修正を行う
	Class Coalescing	2 クラスのフィールドとメソッドを結合させることで、1 つのクラスにする
	Type Hiding	クラス中のメソッドを取り出してインタフェースとして宣言する

Collberg らは、レイアウト、制御構造、データ構造の 3 つに難読化を分類している [1]。レイアウト難読化は、変数名等の識別子を無意味なものに変換する、コメント文を削除するといったものである。これらは、容易に実現でき、プログラムの性能低下が小さく、かつそれなりに高い難読化効果が得られる。しかしながら、それほど多くの手法があるわけではなく、変数名やコメントに透かしを埋め込んでも消去されやすい。そこで我々は、制御構造、データ構造にクラス構造を加えた 3 つの観点で種々の難読化手法を取り上げ、その冗長性を利用した透かしの挿入を検討した。**表 1** は、これまでに透かし挿入法を検討した難読化手法の一覧である。

次節以降ではそれぞれから 1 つの手法を例として挙げ、難読化の方法、難読化による冗長性、透かし挿入法、透かし抽出法を示す。

4　制御構造難読化を利用した透かし挿入法

制御構造難読化の代表的な手法としては、制御の流れをより複雑にする方式や、文の実行順序を変数で制御する方式が挙げられる。制御の流れを複雑にする方式では、主に opaque 述語を用いる。

4.1　Control Flow Scramble

4.1.1　難読化手法

[2] で提案されている難読化方法は、プログラムを複数のブロックに分割し、ループの中で switch 文を用いてどのブロックを実行するかを決定することにより、各ブロックの実行順序を難読化する。これにより、制御の流れが横一列に並んでいるように見える (**図 1**)。

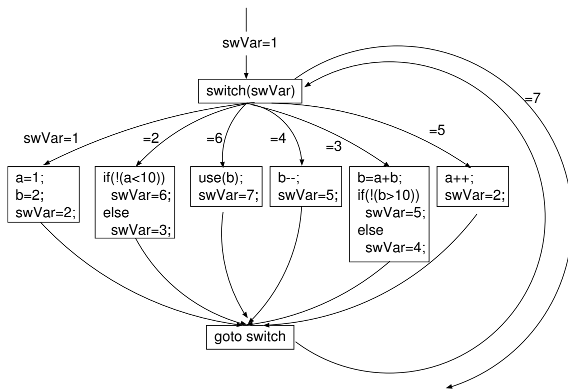

図 1　switch 文による制御フローの難読化 [2]

4.1.2　難読化の冗長性

処理の順序が正しくあるために、switch 文で用いられる制御変数の値の更新の順番はあらかじめ決まっているが、そのときの制御変数の値の取り方に制限はない。

4.1.3　透かし挿入法

制御変数の値は任意に設定できるので、switch 文で用いられる case の値に透かしを埋め込む。

簡単な埋め込み例を以下に示す。

```
switch(swVar){
    case A:{処理 X; swVar = C; break;}
    case B:{処理 Y; goto END; break;}
    case C:{処理 Z; swVar = B; break;}
}
```

上の例に対して、A=0XXX0, B=0XXX1, C=1XXX0 とする。このとき、XXX が透かしを埋め込む bit 列であり、それ以外は識別 bit である。識別 bit は case の値を一意にし、透かし情報を順番に並べることができるようにするために用いる。ここで、010101110 の情報を埋め込む場合、A=00100,B=01011,C=11100 であり、下のようになる。

```
switch(swVar){
    case  4:{処理 X; swVar = 28; break;}
    case 11:{処理 Y; goto END; break;}
    case 28:{処理 Z; swVar = 11; break;}
}
```

4.1.4　透かし抽出法

case の値を全て調べ、それらを識別 bit の値の順に並べることで透かしを取り出す。識別 bit を用いることにより透かしの取り出し順序が決定できるので、case の並べかえが行われても透かしを取り出すことができる。

5　データ構造難読化を利用した透かし挿入法

データ構造難読化は、あるデータを複雑にすることで、それに伴う演算の変換をする方法が多い。実際に使用されるデータや演算に対し透かしを挿入しているので、安易に透かしの挿入されているデータや演算の改ざんを行うと処理自体が正しく行われないものが多い。

5.1　Split Variable

5.1.1　難読化手法

boolean 変数や、その他範囲の制限された変数を複数の変数で表現する手法である。boolean 変数は通常 0 か 1 の値によって True と False を表すが、[1] では 2 変数の値の組み合わせによって True と False を表し、真理値がどちらであるのかをわかりにくくしている (**図 2**)。

図 2 では、2 変数の値が共に等しいなら False を、異なるなら True を表現している。また、プログラム中の元の boolean 変数が使われている箇所を、2 変数を使用したものに置き換えて、正しい動作を行うようにする必要がある。そのときに、**図 3** の規則を用いている。

p q	V	2p+q
0 0	FALSE	0
0 1	TRUE	1
1 0	TRUE	2
1 1	FALSE	3

図 2　2 変数 p, q の値と真理値の対応

VAL[p,q]	p=0	p=1
q=0	0	1
q=1	1	0

AND[A,B]	A=0	A=1	A=2	A=3
B=0	3	0	0	0
B=1	3	1	2	3
B=2	0	2	1	3
B=3	3	0	0	3

OR[A,B]	A=0	A=1	A=2	A=3
B=0	3	1	2	3
B=1	1	1	2	2
B=2	2	2	1	1
B=3	0	1	2	0

図 3　式の置き換えに用いる規則

A,B,C は難読化前の boolean 変数として、これらを難読化すると式の置き換えは例えば以下のようになる。ここで $a_1, a_2, b_1, b_2, c_1, c_2$ は難読化後の 2 変数である。

```
if(A) → x=2*a1+a2; if((x==1)||(x==2))
C=A&B → x=AND[2*a1+a2,2*b1+b2];c1=x/2;c2=x%2;
C=A|B → x=OR[2*a1+a2,2*b1+b2];c1=x/2;c2=x%2;
```

5.1.2 難読化の冗長性

True と False の値の取り方がそれぞれ 2 通りあり、どちらを用いても良い。また、式の置き換え方法も、複数個用意しておくことによりどれを選択するかの冗長性ができる。

5.1.3 透かし挿入法

True と False の値の取り方が 2 通りであり、boolean 変数に値を代入するたび、1 bit の埋め込みが可能。式の置き換えは、下に 4 通りの例を示す。この場合、置き換えのたびに 2 bit 埋め込めむことができる。

if(A) の置き換え

```
00:x=2*a1+a2; if((x==1)||(x==2))
01:if(a1^a2)
10:if(VAL[a1,a2])
11:if((a1-a2)!=0)
```

5.1.4 透かし抽出法

まず、分割された 2 変数を特定する。難読化前のコードと比較を行えば簡単に特定が可能であるが、条件判定文で用いられている変数を調べる等することにより難読化前のコードと比較を行わなくても特定することができる。

その後、プログラムコードを先頭から見て、その 2 変数に対する値の代入や 2 変数を用いた条件判定が現れるたび、対応する bit 列を取り出す。

6 クラス構造難読化を利用した透かし挿入法

近年ソフトウェアの開発言語としてオブジェクト指向型の Java 言語が多く用いられるようになった。Java はコンパイル後も元の情報を多く含んでいるため、攻撃者のリバースエンジニアリングによってソフトウェアの機能を分析され、情報を盗用される危険性が高い。そのため、Java 言語特有の性質を利用した難読化が多く提案されている。

6.1 Type Hiding

6.1.1 難読化手法

元のクラスにあるメソッドをインタフェースとして宣言する。宣言されたインタフェースを元のクラスで実装させることでプログラムを難読化する [3]。以下の例では、クラス C にあるメソッド $m_x ... m_y$ をインタフェース I として宣言し、クラス C でインタフェース I を実装している。

```
class C                 interface I
 {m1,m2,..mx,      ->    {mx,..my}
   ...my,..mn}
                        class C implement I
* m:Method               {m1,m2,..mx,..my,..mn}
```

6.1.2 難読化の冗長性

難読化によって宣言されたインタフェースは、元々メソッドがあったクラスで実装するだけでなく、他のどのクラスでも実装できる。インタフェースを実装したクラスの中で、実際にそのメソッドが使用されないクラスでは、そのメソッドの内容を自由に記述できる。以下の図では、クラス C_1 中のメソッド M_1 をインタフェース I として宣言している。クラス C_1 とは独立なクラス C_2 ではメソッド M_1 は実際に使用されないため、どのように記述してもプログラムに全く影響はない。

```
class C1        interface I
 {m1,m2}        {m1}
          ->
class C2        class C1 implement I
 {m3,m4}        {m1,m2}
                class C2 implement I
                {m3,m4,m1}
```

6.1.3 透かし挿入法

門田らの提案する Java クラスファイルに挿入する方法 [4] を用い、次に示す書き変え可能な特定の数値オペランドとオペコードを透かし挿入箇所とする。

1. オペコードには、その数値オペランドの値を書き変えることができるものがあり、その数値オペランド部分には 8 bit 埋め込むことができる。
2. オペコードの中には、相互に置換できるものがある。この性質を利用して、例えば *iadd* は可換な 8 個のオペコードのいずれかに置き換えることで、3 bit 埋め込むことができる。このようなオペコード群は 16 個存在する。

6.1.4 透かし抽出法

各メソッドの先頭から透かし挿入箇所を検索することで、透かしを取り出すことができる。また、挿入箇所以外のメソッドからの取り出し結果は無意味な文字列となり、プログラム解析等により挿入箇所を特定する必要はない。

7 実装

後述の評価実験のために実装を試み、Control Flow Scramble の手法については実際に実装した。

作成したプログラムは、Java のソースプログラムを入力として受け取り、難読化及び透かしの挿入が行われた Java プログラムを出力する。以下、プログラムの内容について説明していく。

7.1 作成したプログラムの行う処理

プログラムは、次の処理を順に行うことにより、提案手法を実現する。

- Java プログラムの構文解析
- 構文木の生成
- 構文木の操作

まず、JavaCC というツールが生成する構文解析ルーチンを使用して構文解析を行う。その過程で構文木を作成し、構文解析の終了時点で構文木を操作、変更することによって、難読化、透かしの挿入を行う。

7.2　構文木を操作するメソッドの例

構文木を操作するメソッドの例としては、以下のものがある。

- searchNode: 指定したタイプ (METHOD_DEC, BLOCK 等) のノードを検索する。
- searchCallMethod: メソッド呼び出しをしている箇所を探す。
- compareVarName: 引数として与えられた文字列が、プログラム中の識別子として用いられているかを調べる。
- insertNode: 新しい子を作成し、初期化を行う。

これらのメソッドを用いて提案手法を実装する。1 個の手法の実装しかできていないが、このような各手法共通の操作をメソッドとして充実させていくことで、他の手法の実装も容易になると考えられる。

8　評価

ここでは、各提案手法についての評価を行う。

8.1　評価項目

透かしでは、埋め込み量と攻撃への耐性が重要となる。最低 32 bit 程度の情報が埋め込めれば、透かしとして機能するとここでは考えているが、Java のような言語の場合、プログラム部品であるクラス毎の盗用の可能性もあるため、クラス単位で埋め込み量を確保できることが望ましい。

透かしに対する攻撃としては、人間の手作業による除去と、プログラムに難読化や最適化等のプログラム変換を施すことによる除去がある。特に後者は、プログラム等の知識がない者も行うことができ、実行に手間がかからないので、こちらの方が透かしへの攻撃手段としては一般的であると言える。

よって、ここでは、以下の 2 点を提案手法に対する評価項目とする。

- 埋め込み量
- プログラム変換に対する耐性

8.2　評価方法

あるプログラムに対して、提案する各手法を用いた難読化と透かしの挿入を行い、クラス単位で透かしの挿入を行えるかどうかを目安に埋め込み量を調べた。但し、Control Flow Scramble しか実装ができていないので、難読化及び透かしの挿入は手作業で行っている。

その後、そうしてできたそれぞれのプログラムに SandMark[5] を用いて 21 種類の難読化、最適化を適用し、適用後も透かしが取り出せるかどうかを調べた。

サンプルプログラムは、行数 800 程度、クラス数 11、メソッド数 63 の規模のものを用い、48 bit の情報の埋め込みを試みた。なお、手法の適用と透かしの抽出はソースレベルで行っている。

8.3　評価

いずれかのクラスに対してそのクラス内で透かしを挿入できた手法は、Control Flow Scramble, Interleave Methods を含め 5 つの手法があり、逆に、Extend Loop Condition, Restructure Array, Class Coalesing は透かしを挿入できるだけの埋め込み量が確保できなかった。

プログラム変換への耐性に関しては、調べた範囲で透かしが破壊されなかった手法は Control Flow Scramble や Change Encoding 等の 4 手法があり、他の手法は数個の変換に対して透かしが破壊された。

9　おわりに

難読化と電子透かしの挿入を併用する方法の一つとして、難読化の際に生じる冗長性を利用して透かしを挿入する手法を提案した。そして、実際に提案手法による難読化及び透かしの挿入を行い、それらに対して難読化、最適化を行うことで、埋め込み量と耐性を評価した。

結果、透かしの埋め込みが行えた手法もあれば、十分に埋め込むことができなかった手法もあった。耐性に関しては、全てのプログラム変換に対して耐性がある手法は少なかったが、いずれの手法も多くの変換では透かしが破壊されないので、実用性はあると言える。透かしの埋め込みができなかった手法については、難読化時の冗長性が皆無ではないので、今後は、別の透かし挿入法を検討していく必要がある。

難読化の冗長性を利用した透かしの挿入は、透かしの挿入によるプログラムの性能の低下が生じない。よって、様々な難読化手法に対して、その冗長性を利用した透かし挿入法を用意することで、難読化を行う際には常に、透かしを挿入することが期待できる。

参考文献

[1] Christian Collberg, Clark Thomborson, and Douglas Low: A Taxonomy of Obfuscation Transformation, TR 148, Department of Computer Science, University of Auckland, July 1997.

[2] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight: Protection of Software-based Survivability Mechanisms, International Conference of Dependable Systems and Networks, July 2001.

[3] Mikhail Sosonkin, Gleb Naumovich, and Nasir Memon: Obfuscation of Design Intent in Object-Oriented Applications, Proc. ACM Workshop on Digiatl Rights Management, October 2003.

[4] 門田暁人, 松本健一, 飯田元, 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, November.2000

[5] Christian Collberg, Ginger Myles, and Andrew Huntwork: Sandmark - a tool for software protection research, IEEE Security and Privacy, Vol. 1 No. 11, July/August 2003