ソフトウェアに対する動的電子透かしの検討
– トレース情報のグラフ表現手法 –

2003MT096 志村敏彦　　2003MT101 棚瀬真臣
指導教員　真野 芳久

1　はじめに

現在ソフトウェアの盗用が行われ、著作権侵害が大きな問題となっている。ソフトウェアの盗用を防ぐことを目的としたソフトウェアプロテクション技術の一つとして、電子透かしと呼ばれる技術が存在する。これは、ソフトウェアに対してあらかじめ著作権情報を埋め込んでおき、もし盗用が発生した時にも著作者は、その権利を主張することが可能となる。

ここでは、透かし攻撃への耐性の強い電子透かしの実現を目指し、動的グラフ透かしと制御の流れを組み合わせた手法を提案する。そしてその性能を実験によって確認する。

2　電子透かしとは

電子透かしは、大きく静的電子透かしと動的電子透かしに分けることができる。

2.1　静的電子透かし

静的電子透かしとは、プログラムを実行すること無く電子透かしを取り出す手法である。難読化攻撃や最適化攻撃などの semantics-preserving 攻撃に対して 弱いという欠点がある。静的電子透かしは、データに対する電子透かしとコードに対する電子透かしの 2 種類に分けることが可能である。

2.2　動的電子透かし

動的電子透かしとは、プログラムを実行して、その動作の挙動などを観察することでそのプログラムの中に電子透かしが入っていることを確認する手法である。動的電子透かしには以下の 3 種類を挙げることができる。

■Easter Egg **電子透かし**　プログラムに特定の入力を与えた時に、著作権情報を表示する透かしの手法である。この手法の利点は、手軽であるためオーバーヘッドなく透かしを埋め込むことができるということである。

一方欠点は、著作権情報をそのまま出力してしまうので、一度その存在を知られてしまうと透かし情報を取り外すような攻撃をされる恐れがある。

■Dynamic Data Structure **電子透かし**　特定の入力が与えられた時に、プログラムが動作する中の一時点において構築されるデータ構造に対して電子透かしを埋め込む手法。

この手法の利点は、著作権情報をそのま出力しないので、Easter Egg 電子透かしよりも攻撃をされにくい。

■Dynamic Execution Trace **電子透かし**　特定の入力が与えられた時に、プログラムの実行の流れの中に電子透かしを隠すという手法である。

利点としては、Dynamic Data Structure 電子透かしと同様に、明らかな出力を出さない。さらに、Dynamic Data Structure 電子透かしのような変数の値やデータ構造といった一時点における直接的な透かし情報を持たないので、攻撃者にとってより分かり難い手法だと言える。

2.3　攻撃の種類

■subtractive **攻撃**　透かしを外そうとする攻撃であり、攻撃者にとって透かしの存在とその場所を理解していることと、透かしを取り外した後のオブジェクトが攻撃者にとってまだ有用である必要がある。

■additive **攻撃**　既に透かしが埋め込まれているプログラムを対象に、攻撃者が透かしを上書きする攻撃である。

■distortive **攻撃**　オブジェクト全体に一様に難読化や最適化などの semantics-preserving 変形を行い透かしの品質を低下させ ようとする攻撃である。透かしの場所を見つけられない時に用いられる。 品質が低下したことに気付かないことと、品質低下後のオブジェクトがまだ有用である必要がある。

3　Dynamic Graph 透かし

Dynamic Graph 透かしは Dynamic Data Structure 電子透かしの一種であり、特定の入力での実行に応じて構築されるグラフ構造の位相へ透かしを埋め込む動的手法である。利点としては、以下に示すものがある。

- 動的なグラフ構造は解析を行うことが難しい。その結果、難読化攻撃や最適化攻撃に強いという性質を持つ。
- オブジェクト指向言語では、ポインタの操作が自然な操作なので、透かしの判別が難しい。
- グラフを分割して扱うことで、プログラムの様々なグラフ構造の中に透かし情報を埋め込むことにより大きな秘密情報を埋めこめる。

以下に Radix-k encoding[1]、Enumeration encoding[1] の 2 つの手法を述べる。

3.1　Radix-k encoding

この方法は、埋め込みたい数 n をある進数になおし、それを循環リストへ変換する。m をリストの長さとした時、0 から $(m+1)^m - 1$ の範囲の数を表現することができる。基数が 6 の場合の Radix-6 encoding の例を図 1 に示す。

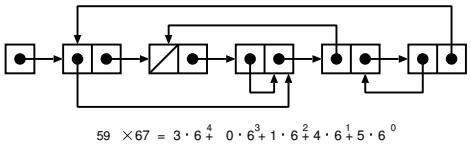

図 1　Radix-6 encoding:基数 6 でのエンコード

- 右のフィールドのポインタは、次のフィールドを指す。
- 左のフィールドのポインタはある桁の値を示しており、以下の規則に従う。
  - 0 の場合は、null ポインタとする
  - 1 の場合は、自分自身のセルを指すポインタ
  - 2 以上の場合、その数だけ先のポインタ

3.2　Enumeration encoding

この手法は列挙的に数を割り振る方法で、例えば図 2 のように木の形と数を対応付けて考える。

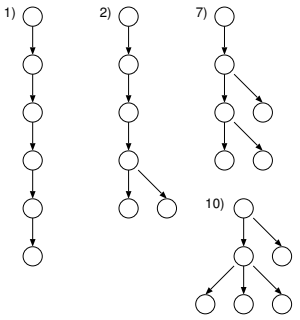

図 2　Enumeration encoding:ノード数が 6 の場合

4　Dynamic Execution Trace Graph 電子透かし

3 節で述べたように、Dynamic Graph 透かしを利用するとより強い攻撃耐性が得られる。しかし、グラフ構造を対象とした攻撃として考えられるダミーポインタフィールドの追加、フィールド名や順序の変更、等に弱く、全ての攻撃に対して耐性が得られるわけではない。よってここでは、動的グラフ構造を対象とした攻撃に強い動的電子透かしの手法として、Dynamic Execution Trace Graph 電子透かし (以下では DETG 電子透かしと略す) を提案する。この手法では、本来データ構造として表現されるグラフ構造をプログラムの動的な実行の流れを使い表す。このようにグラフを表現することによって、メモリ内にはグラフ構造が存在しなくなり、データ構造を操作する攻撃に耐性ができる。その一つの例として我々は、プログラム実行時の制御の流れを使い Radix-k encoding を表す手法を述べる。

4.1　単純な例

DETG 電子透かしとして、我々は switch 分岐を使う方法と配列を使う手法の 2 種類を考えた。以下にそれらの手法を述べる。

4.1.1　switch 分岐を使う Radix-k encoding

この手法は、case の並びを Radix-k encoding でのリストとみたて、case の実行の流れがそれぞれ Radix-k encoding での左側フィールドポインタの示す先を表す。この例では、図 1 と同じ構造を表しており 59 × 67 が透かし情報として埋め込まれている。

```
public class Switch1 {
   public static void cr(int a, int b){
       switch(a-b){
       case 1:break;
       case 2:break;
       case 3:break;
       case 4:break;
       case 5:break;
       default :break;}}

   public static void main(String args[]){
       cr(5,2);cr(7,1);cr(6,3);cr(-2,-4);cr(6,2);
   }}
```

4.1.2　配列を使う Radix-k encoding

この手法は、配列の並びを Radix-k encoding でのリストとみたて、配列の値への操作、例えば値への代入命令の流れが、それぞれ Radix-k encoding での左側フィールドポインタの示す先を表す。

4.2　より攻撃耐性のある手法

4.1 節で述べた手法の利点として、メモリ内にはグラフ構造が存在しなくなり、グラフ構造を対象とした攻撃に耐性ができた。しかし、switch 分岐を使う手法を例として述べていくと、switch 分岐の静的な case の並びをリストとみたてているので、プログラムに影響をあたえないで switch 分岐構造を変える簡単な攻撃 (例えば、switch 分岐並び替え等) で透かし情報が壊れてしまう可能性がある。そのため本節で述べる手法は、switch 分岐の構造を変える攻撃を想定し、それらの攻撃に対して耐性を高めることを目標としている。具体的には、実行の流れで Radix-k encoding の全てのポインタを表すことによって、switch 分岐の静的な case の並びに依存しない手法とする。

4.2.1　2 回実行での手法

Radix-k encoding の全てのポインタを実行の流れで表すので、透かし部分はリストの並び (右フィールドのポインタ) を決める実行と、各桁の重み (左フィールドのポインタ) を決める実行の 2 つに分ける必要がある。それら 2 つの実行を相対的に見ることで、Radix-k encoding を表現する。2 つの実行を相対的に見て透かしを表すこ

とによって、semantics-preserving 攻撃に強くなるという性質がある [3]。

以下に図 1 と同じ構造を表した例を示す。

```
public class Switch1 {
  public static void cr(int a, int b){
     switch(a-b){
     case 1:break;case 2:break;
     case 3:break;case 4:break;
     case 5:break;default :break;}}

  public static void main(String args[]){
     cr(3,2);cr(3,1);cr(6,3);cr(1,-3);cr(10,5);
     cr(5,2);cr(7,1);cr(6,3);cr(-2,-4);cr(6,2);
  }}
```

この例では、switch 分岐を 10 回実行し、10 回の実行のうち最初の 5 回が case(右フィールドのポインタ) の並びを決める実行、後の 5 回がそれぞれの重みポインタ (左フィールドのポインタ) を示している。それらを相対的に見ることで Radix-k encoding を表している。

Radix-k encoding のポインタを全て実行の流れで表した結果、switch 分岐の静的な構造には依存しなくなり switch 分岐の構造を変える攻撃に対して耐性が強くなったと考えられる。しかし、制御の流れを変えられると透かし情報が壊れてしまう。この例では switch 分岐の制御が簡単な手法になっており、攻撃者に分かりやすくなっている。これらの理由で、このままでは switch 分岐の制御部分が攻撃に対して弱いと考えられる。そのため、制御部分の攻撃に対する耐性を上げるため、制御部分が攻撃者にとって透かし情報だと分かり難くする必要があり、その一つの手法として、2 分探索を使う手法を考えた。

4.3 2 分探索と Radix-k encoding の合成手法

この手法は 2 分探索と Radix-k encoding の手法を合成することで Radix-k encoding の耐性をさらに高めることを目標としている。具体的には、4.2 節で述べた提案手法における 1 回目の実行と 2 回目の実行を、それぞれを 2 分探索の探索結果で表すことで、さらに攻撃者にとって分かり難く透かし情報を挿入する。

4.3.1 2 分探索によって得られる情報

この手法は配列に対する 2 分探索の動きが表す情報を使い Radix-k encoding を表す。結果、透かし部分が 2 分探索という一般的なアルゴリズムとなり、攻撃者にとって透かし情報が分かり難くなると予想できる。ビット列を埋め込む手法の具体例として、検索キー値がチェック値よりも大きければ 1、そして、チェック値よりも小さければ 0 を与えることにする。

この手法では配列の長さを n とした時に、1 回の 2 分探索で $\log_2 n$ ビットの情報を得ることが可能になる。

4.3.2 Radix-k encoding の表現方法

2 分探索から抽出したビット列で表現される値を使い Radix-k encoding を表現する。この手法をとる利点として、2 分探索を使うことで透かしが埋め込まれているかどうかの判別が困難となり、その結果攻撃に対して耐性に強くすることが可能になったと考えられる。しかし 2 分探索から抽出したビット列は、2 分探索対象の配列が少しでも変えられると変わる可能性がある。さらに、2 分探索で得られるビット列は配列の静的な並びと探索キーの列に依存しており静的にビット列が解析される可能性がある。

4.4 2 分探索を用いた Radix-k encoding の耐性強化

4.3.2 節で述べたように、2 分探索を普通に用いて Radix-k encoding を表現する方法では、静的に透かしの情報が並ぶだけであり、耐性が弱い。そこで、2 分探索を強化する方法として探索の偶数奇数に応じて探索キーと配列の要素の値を変更する方法を提案する。

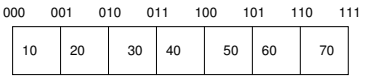

図 3　改良前の 2 分探索の透かし情報

図 3 の例では、例えば探索キーを 35 とした時、30 と 40 の間の 011 というビット列が得られることは一目瞭然であり、これでは耐性が弱い。

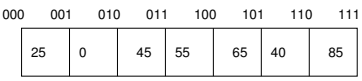

図 4　改良後の 2 分探索の透かし情報

図 4 の例では、探索の回数に合わせて配列内の要素の値を変えておき、探索キーを奇数回目の探索の時は、15 増加させて、偶数回目の探索の時は、20 減少させる。こうすることで、透かし情報が静的に並ばなくなる。

4.5 他の方法への応用

ここでは、DETG 電子透かしの一つの例として、Radix-k encoding を使う。しかし、実行の流れをデータ構造として表現する手法は、他のグラフ構造にも応用できると考えられる。そのため、攻撃に対して強いグラフ構造を使うことで、より強い DETG 電子透かしの現実が可能となる。

5 実験評価

DETG 電子透かしの手法である switch 分岐を使う Radix-k encoding、配列を使う Radix-k encoding、2 分探索を使う Radix-k encoding に対して、透かしの持つべき特性である埋め込み効率、実行効率、攻撃への耐性の 3 種類を実験によって評価する。

表 1　サイズの変化

	switch	配列	2 分探索
埋め込み前 (byte)	10818		
埋め込み後 (byte)	11018	10954	11006
増加量 (byte)	200	136	188
増加割合 (%)	1.8	1.3	1.7
埋め込み効率 (増加量/bit)	17	11	16

表 2　実行時間の変化

	switch	配列	2 分探索
埋め込み前 (m 秒)	6006		
埋め込み後 (m 秒)	6015	6009	6014
増加量 (m 秒)	9	3	8

表 3　難読化攻撃の効果

	switch	配列	2 分探索
透かし情報が壊れない	37	34	37
透かし情報が壊れる	0	1	0

■実験準備　サンプルプログラムとして、行数約 660 行、クラス数 14、メソッド数 50 の規模のものを用いて、透かし情報として「3953」(12bit) を 2 回実行での Radix-6 encoding で表したものを、プログラム中で 1 回実行される位置に埋め込む。サンプルプログラムをまとめた jar ファイルを対象とし、実験を行った。

5.1　埋め込み効率

透かし情報を埋め込む前後での jar ファイルのサイズ変化を表 1 に示す。全ての手法において、増加割合 1〜2% となるので、両手法ともに、サイズ増加量は良いと言える。

透かしコードに対する埋め込み効率は、11〜17byte に対して 1bit の情報を埋め込むことができる。

5.2　実行効率

透かし情報を埋め込む前後での実行時間の変化を表 2 に示す。実行時間の測定は、それぞれ 10 回ずつおこなった結果の平均実行時間を示す。全ての手法において、実行時間増加量が 10m 秒未満であることから、透かし埋め込みによる実行時間の増加は、誤差の範囲だと言える。よって、透かし埋め込み後の実行効率は良いと言える。

5.3　難読化攻撃への耐性

攻撃方法として、SandMark[5] を用いての難読化攻撃を行う。

SandMark に含まれる 37 種類の難読化手法を透かし情報埋め込み後のサンプルプログラムに対しておこなった結果を表 3 に示す。なお、配列を使う手法では 2 つの難読化手法に失敗している。

全ての手法において、難読化に対しては強い耐性を持つという結果となった。特に switch 分岐を使う手法と 2 分探索を使う手法が、配列を使う手法より難読化攻撃に対する耐性が強いという結果となった。

配列を使う手法において、透かし情報を壊す、あるいは壊すかもしれない難読化手法は、どれも配列を 2 つに分けるという手法であり、配列の種類によって壊れるか否かが決まる。また、今回使用した SandMark には switch 分岐の構造を変化させるような攻撃がないため耐性が強いという結果になったが、どちらも基本となる構造を 2 つ以上に分ける種類の難読化攻撃に弱いということが判明した。

6　おわりに

本研究では、実行時に得られるトレース情報からグラフを表現し透かし情報を埋め込む手法を提案した。

SandMark を用いての実験では、switch 分岐を使う手法、配列を使う手法の両手法ともに、埋め込み効率、実行時間、難読化攻撃に対する耐性の全ての面において良い結果となった。

しかし、現状で分かっている問題点も多くある。例えば、透かしコード制御部分の脆弱性、配列を分割する攻撃への脆弱性がある。そのため、他のグラフ構造への応用、トレース情報からグラフを表現する手法などを工夫することが今後の課題として挙げられる。

参考文献

[1] C.Collberg, C.Thomborson：“Software Watermarking: Models and Dynamic Embedding” POPL'99, pp.311-324 (Jan. 1999).

[2] D.Currans, N.J.Hurley, M.OCinneide：“Securing Java through Software Watermarking” In Proc. of the 2nd International Conf. on Principles and Practice of Programming in Java (2003).

[3] C.Collberg, E.Carter, S.Debray, A.Huntwork, J.Kececioglu, C.Linn, M.Stepp：“Dynamic Path-Based Software Watermarking” PLDI 04(June 2004).

[4] C.Collberg, C.Thomborson：“ Error-Correcting Graphs for Software Watermarking ”Workshop on Graph Theoretic Concepts in Computer Seience (July 2003).

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