ソフトウエア電子すかしの挿入法、攻撃法、評価法、実装法

一杉 裕志
電子技術総合研究所

概要

ソフトウエアの違法コピーを抑止するための、ソフトウエア電子すかしについて述べる。電子すかしの挿入法として素朴な方法と比較的有望と思われる assert 法を提案し、それらに対する攻撃法と利便性の評価法について述べる。また、電子すかしアルゴリズム等を実装するためのツールとして用いることができる、拡張可能 Java プリプロセッサ EPP を紹介する。さらに、電子すかしに関連したいくつかの研究課題について述べる。

1 はじめに

マルチメディアデータの違法コピーを抑止する方法として、電子すかしが最近実用化されつつある。これは、マルチメディアデータの冗長性を利用し、そのままでは見えない形でユーザＩＤなどの情報を挿入する手法である。万一データの違法コピーが行なわれても、発見された違法コピーデータから挿入された情報を取り出すことによって流出元のユーザを特定できるので、違法コピーの抑止に役立つ。

筆者の知る限り、ソフトウエア（プログラム）に対して電子すかしを挿入する研究は現在のところほとんど行なわれていない。しかし、ソフトウエアもその表現は冗長であり、それを利用した電子すかしの挿入が可能である。例えば、変数名や関数名は変更しても意味は変わらないので、それを利用して情報を埋め込むことができる。ただし、このような素朴な方法は埋め込まれた情報を消去するのも簡単である。攻撃者はすべての変数名をランダムな名前に付け変えることで、情報を消去することができる。

なお、現在もソフトウエアの知的所有権を保護するメカニズムとして、ライセンスサーバなどが実際に使われているが、プログラムを解析してライセンスをチェックする部分を外される危険は常につきまとう。暗号技術も違法コピー防止には無力である。例え配送時にソフトウエアを暗号化したとしても、プロセッサ上で実行する時点では平文にならざるを得ないからである。

Watermark for Software and its insertion, attacking, evaluation and implementation methods. Yuuji ICHISUGI (Electrotechnical Laboratory).

本論文の 2章ではソフトウエア電子すかしの挿入方法をいくつか提案し、3章ではそれぞれに対して考えられる攻撃法を分類する。また、4章では電子すかしアルゴリズムの評価法について述べる。5章ではいくつかの簡単なアルゴリズムに対する主観的な評価についてまとめる。6章ではソフトウエア電子すかしの実装の道具として用いることができる、拡張可能 Java プリプロセッサキット EPP を紹介する。7章では電子すかしに関連する他の研究テーマについて述べる。最後に 8章でまとめを行なう。

2 ソフトウエア電子すかしアルゴリズムの分類

ここでは電子すかしの挿入法として、素朴な方法を含めていくつか列挙し、それぞれの特徴や想定される攻撃方法について述べる。これらの複数の挿入アルゴリズムを複雑に組み合わせることで、攻撃の難しい実用的な電子すかしアルゴリズムができると期待できる。

なお、本論文では、ソースコードからに何らかの操作を施して情報が挿入されたソースコードを挿入するアルゴリズムだけを考えている。ソースコードからバイナリへの変換時、バイナリからバイナリへの変換は本論文では対象としない。また、 CD-ROM での大量配布は想定しておらず、ユーザごとに異なった情報を挿入して配布する状況を想定している。

2.1 データ挿入

これは最も簡単な方法で、ソースコード中に文字列定数などの形で情報を挿入する方法である。このままでは、攻撃者が文字列がある場所を見つけて改変するのは容易である。プログラムに自分自身のチェックサムを計算させて、改変されていないかチェックするともう少し強力になるが、プログラムを解読されてチェックルーチンを外される危険性は残る。

2.2 変数名変換

ソースコード中の変数名や手続き名を、プログラムの意味を変えないように変換して情報を挿入する方法である。この方法は変数名をすべて変換することで簡単に情

報を消去できる。

2.3 プログラム構造変換

言語のセマンティックスを利用した、より高度な方法として、以下のようなものが考えられる。

- 変数の代入の順序を入れ換える。
- if-then-else の変換。(if 文の条件式に not をつけて、 then 部と else 部を入れ換える。)
- 手続き呼びだしの順序を入れ換える。
- 手続きの定義の順番を入れ換える。
- ループの回り方を入れ換える。

ただし変換規則が単純だと、コンパイラによる最適化によって簡単に情報が消える恐れがある。また、変換ルールが公開されていて、なおかつ逆変換が簡単に行なえる場合も、攻撃者によって簡単に情報を除去できてしまう。また、局所的な変換は 3.2で述べる結託攻撃に弱い。

2.4 言語仕様のあいまいさを利用

多くの言語では言語仕様に厳密に決まっていない部分があり、コンパイラの実装にまかされている。この部分を利用してソースコードを変換することで情報を挿入できる。

例えば C 言語では関数呼び出しや演算子の引数の評価順序はコンパイラが決めてよいと定義されている。これを利用し、特定の順序で引数を評価するコードに変換することで情報を挿入することができる。並列言語の場合は、逐次言語より実行順序の自由度の高い構文が通常用意されているため、情報挿入の可能性も高い。

2.5 複数バージョン実装

実際に n 人のユーザに対して、全く独立して開発された n 個のバージョンを配布するという方法である。個々のバージョンは全く異なっているので、違法コピーがあれば流出元は一目瞭然である。これは、考えられる中で最も強力な方法である。

この方法はこのままでは非現実的だが、より現実的なバリエーションは考えられる。

例えば、ソフトウエアの一部のモジュールに対してだけ実装をいくつか用意し、ユーザごとに異なる組み合わせで配布すればよい。

また、もともとソフトウエアは次々にバージョンアップしていくものなので、1つのバージョンにつき1人のユーザにのみ配布するという手段も、ユーザの少ないソフトウエアならば可能である。

2.6 プログラマのみが知り得る情報の利用

プログラムのソースコード中には、配布するバイナリには必ずしも反映されないが、プログラマは知っている情報が含まれており、これを利用することができる。

その1つ例として assert 法を提案する。C/C++ プログラマはデバッグのために、次のような assert マクロをソースコード中に埋め込むことが多い。

```
assert(expression);
```

プログラマは、実行時に expression が必ず真であると仮定してプログラムを書く。もし expression の評価が性能にあまり影響を与えないならばそれを利用してソースコードを次のように変換することができる。

```
assert(P); A;   →   if (P) { A; } else { X; }
```

式 P はソフトウエアにバグがなければ常に真であるが、機械的にソースコードを解析してもそのことを見つけるのは不可能である点が重要である。つまり、この変換は非可逆的である。式 X は、オリジナルのソースコード中に存在しない任意のコードで、この部分は電子すかしアルゴリズムによって非常に利用価値が高い。単にこの部分に情報を挿入できるというだけでなく、静的解析攻撃を混乱させる情報をこの部分に入れることができるからである。

assert 法に対しては、次のような攻撃法が考えられる。まず assert 法による情報挿入が施されたソースコードが手元にあるとする。ソース中の全ての if 文に対し、条件式の評価結果のログを取るコードを埋め込み、実際にソフトウエアを実行してみることで、「一度も偽にならない条件式」を見つけ出すことができる。そして、そのような条件式を持つ if 文は assert 文であったと見なし、

```
if (P) { A; } else { X; }   →   P; A;
```

という変換を施すことができる。もちろん、有限時間でのソフトウエアの実行だけでは、条件式が本当に常に真になるものかどうかは判別できない。しかし、完全ではなくても「ほとんど問題なく動く」違法コピーソフトウエアが作れることになる。

3 攻撃法の分類

本章では、前章で述べた各アルゴリズムで挿入された情報を除去する手法について分類する。これらの攻撃法のいくつかは、マルチメディアデータの電子すかしに対しても適用可能なものがある。

3.1 解読攻撃

なんらかのセキュリティを保証する「コード」がソフトウエアそのものに局所的に埋め込まれている場合、そのコードを解読して改変するという攻撃法がある。

例えば、従来からあるライセンスサーバーの場合、アプリケーションを逆アセンブルして解読して、ライセンスをチェックするルーチンを実行しないようにソフトウエアを改造される危険性が常につきまとう。

解読攻撃の最も極端な場合として、配布されたソフトウエアの仕様を完全に理解し、独自の方法で0から実装し直せば、それは「すかしの除去されたコピー」となる。画像データに例えれば、元絵を見ながら模写することに相当する。これはあらゆる電子すかし挿入アルゴリズムに有効ではあるが、違法コピー業者にとってはもちろん意味のない方法である。

3.2 結託攻撃

異なる電子すかし情報が入ったソフトウエアを複数入手し、それらを元に本来の挿入されていたものと異なる情報が挿入されたソフトウエアを作り出す方法である。結託攻撃はマルチメディアデータに対する電子すかしの多くにも適用することができる。

結託攻撃の例を、画像に対する電子すかしで説明する。画像情報の各 pixel の色を、ユーザに分からない程度に微妙に変化させることで情報が挿入されているとする。攻撃する側は、複数の画像データを手に入れ、同一箇所の pixel ごとに平均値をとったり、それぞれの画像データからランダムに pixel を選択したりすることで、画像としての見ためは変えずに挿入されたデータを破壊することができる。

ソフトウエアの場合、結託攻撃の例として次のようなものがある。各手続きの内部の式の順序を入れ換えることで情報を挿入するアルゴリズムがあったとする。複数のバージョンのソフトウエアを手に入れ、それぞれのバージョンから、ランダムに1つづつ手続きを選び出すことで、挿入された情報を破壊できる。

マルチメディアデータの場合、技術的知識を持たないユーザによる気軽なコピーを抑止できれば十分とするケースも多く、結託攻撃は必ずしも致命的なものとはならない。

一方ソフトウエアの場合、技術的知識を持たないユーザだけを対象としてよいのなら、コード中にユーザＩＤを文字列として入れるなどの従来の簡単な方法で十分である。したがって、ソフトウエア電子すかしの研究者は、もし従来の方法よりも強力な電子すかしであることを主張したいならば、結託攻撃を無視することはできない。

なお、ユーザＩＤをビット列に対応させる符号化方式を工夫することで、結託攻撃に対処する方法が提案されている [7]。この方法を用いると、複数人のユーザが結託し、電子すかし入りデータを持ち寄って混ぜ合わせ新しいデータを作っても、高い確率でその中の１人のユーザを特定することができる。

結託攻撃に対するもう１つの対策として、難読化や難改変化を電子すかしと併用することがあげられる。これについては7章で述べる。

3.3 多重挿入攻撃

電子すかし挿入ツールあるいは正確な挿入アルゴリズムが公開されている場合、攻撃者は、配布された電子すかし入りデータに、異なった情報を繰り返し挿入することで、もともと入っていた情報を破壊することができる。これを多重挿入攻撃と名付ける。

多重挿入攻撃を紙の上の文字に例えると、次のようになる。もともと 12 という情報が書かれた紙に、攻撃者がその上にさらに 34 という情報を書き込むことができる。その結果、もともとの情報が 12, 14, 32, 34 のどれだったのかは、判別できなくなる。攻撃者はさらに数百、数千も異なった情報を挿入することで、もとの情報を事実上読めなくすることができる。

マルチメディアデータに対する電子すかしも含めて、多くの挿入アルゴリズムはこの攻撃に弱いと思われる。対策としては、使用した電子すかし挿入ツールや正確な挿入アルゴリズムを公開しないことしかない。

2.6節で述べた assert 法は、多重挿入攻撃が不可能な挿入法の例である。一度挿入を施すと assert 文がなくなるので、それ以上情報を挿入することができない。

また、情報の多重挿入が急激な「質の悪化」を招くようなアルゴリズムは、多重挿入攻撃に強いと考えられる。ソフトウエアの場合、例えば１つ情報を挿入するたびにサイズや実行速度が２倍になるようなアルゴリズムであれば、例え数回であっても情報を多重挿入するともはやそのソフトウエアは使いものにならない。多重挿入の回数が少なければ、もとの情報をある程度推察することが可能なので、電子すかしとしての役割を果たしうる。

3.4 最適化攻撃

電子すかし挿入アルゴリズムが単純な場合は、すかしの挿入されたソースコードに対し、最適化コンパイルをかけることでプログラムの冗長性さが消滅し、挿入した情報が消される可能性がある。この攻撃方法は、挿入アルゴリズムが既知でなくても、また電子すかしの原理に関する知識が全くなくても実行することができる。

しかし、コンパイラの最適化が必ずしも「最適」なコードにならないのと同様に、情報が消えないで残る可

能性も大きい。一般的なコンパイラがどの程度の最適化をするかを理解した上で挿入ツールを作成することで、最適化攻撃には十分対処できる。

3.5 静的解析攻撃

挿入アルゴリズムがソースコードの静的な性質のみに基づいており、なおかつ挿入アルゴリズムが攻撃者に知られている場合、ソースコードから機械的に情報を除去するツールが作成されてしまう場合がある。

3.6 動的解析攻撃

プログラムの動的な性質に基づいている挿入アルゴリズムでは、デバッガとともにソフトウエアを実行し、実行状態を調べることで電子すかしを除去することが可能なものがある。

2.6節で述べた assert 法に対する攻撃法も動的解析の一種と言える。

4 電子すかしの評価法

電子すかしアルゴリズムはもちろん様々な攻撃に対して安全であることが重要であるが、それ以外にも利便性が良いことが必要である。この章では、利便性を評価するいくつかの視点を列挙する。

4.1 情報挿入密度

一定量のソースコード中に、何 bit の情報を埋め込めるか、という尺度である。アルゴリズムによっては無限なものもある。情報挿入密度が小さいと、ソースコードが短く配布先が多い場合に、全員に異なった情報を挿入することができない。

[7] の結託攻撃に強い符号化法では、ユーザの数 n に比例した bit 数が必要である。これは、通常の 2 進数による符号化では log n であるのに比べると、非常に大きい値である。また、攻撃者を正しく特定する確率は、符号の bit 数を増やすほど指数関数的に上がる。従って [7] を利用するためにはできるだけ情報挿入密度が高い方が望ましい。

4.2 原理の明解さ

実際に違法コピーが行なわれ、裁判によって争われることになった時、陪審員あるいはソフトウエア専門家に対して「本当に流出元がここである」と説得できる必要がある。

つまり、解読さえされなければ目的が達成される暗号とは異なり、いかに強力であっても、アルゴリズムや原理が複雑すぎると役に立たない場合もある。

4.3 挿入ツール・アルゴリズムを公開できるか

一般に挿入ツール・アルゴリズムが公開されていると、多重挿入攻撃や静的解析攻撃による情報の破壊の危険性が生じる。挿入アルゴリズムを公開しない場合、サーバを信頼できるホストで稼働させてそこに電子すかしの挿入を依頼するという運用形態に制限されてしまう。このような形態ではアルゴリズム自体の信頼性を得られくい。また、挿入ツールのフリーソフトでの公開、あるいは商品としての販売ができないので、電子すかし自体の普及のさまたげになる。

できれば、挿入アルゴリズム／ツールを公開しても安全なアルゴリズムであることが望ましい。

4.4 配布側の負荷

ソースコードへ情報を挿入するために必要な時間は小さい方が良い。もしこれが十分小さければ、ネットワークを使ったソフトウエア配布の際に、要求があるたびにその場で情報を挿入して配布する、という使い方ができる。

4.5 実行効率低下

情報を挿入することでプログラムのサイズおよびプログラムの実行時間が増加することがありうる。理想的にはオリジナルのソースコードとほとんど変わらないことが望ましい。しかし、ソフトウエアの用途によっては、サイズや実行時間が数倍になったとしても、電子すかしの安全性がより強力に保証されている方が望ましい場合もある。

4.6 プログラマの協力が必要かどうか

任意のソースコードに対して電子すかしが挿入できる方が望ましいが、プログラマに冗長性に関するヒントなどを書いてもらうことで、より強力な挿入アルゴリズムが実現できる場合がある。

5 いくつかの挿入アルゴリズムの評価

これまでで述べた挿入アルゴリズムからいくつか選び、（主観的に）評価して表にしたものが表 1である。この中では、動的解析の危険や実行効率低下はあるものの、 assert 法が比較的有望と考えられる。

また、いずれの方法も局所的な変換に基づいているので、そのままでは結託攻撃に弱い。従って、ユーザＩＤの符号化の工夫や難読化との組み合わせなどが必要である。

挿入法	1	2	3	4
耐解読攻撃	×	○	○	○
耐結託攻撃	×	×	×	×
耐多重挿入攻撃	×	×	×	○
耐最適化攻撃	○	×	×	○
耐静的解析攻撃	×	×	×	○
耐動的解析攻撃	○	○	○	×
情報挿入密度	○	○	×	○
原理の明解さ	○	○	○	○
挿入ツール公開	×	×	×	○
配布側の負荷	○	○	○	○
実行効率低下	○	○	○	×
プログラマの協力	○	○	○	△

1. 単純な文字列定数挿入
2. 変数名変換
3. if-then-else の変換
4. assert 法

表 1: 挿入アルゴリズムの評価

6 電子すかしの実装法

6.1 ソースコード操作ツールの必要性

ソフトウエア電子すかし挿入ツールは、プログラムのソースコードに変換を施す source to source トランスレータである。同様なツールとして、ソースレベルでの最適化ツールなどがある。このようなトランスレータを作るためのフレームワークを、以後ソースコード操作ツールと呼ぶ。

電子すかしアルゴリズムの発展に関しては、暗号アルゴリズムがそうであったように、実際にそのアルゴリズムを施した結果を公開して、攻撃方法を広く公募し、結果をフィードバックするというサイクルが不可欠であろう。また、対象とする言語の言語仕様のバージョンアップへの追従やさまざまな方言への対処も容易でなければならない。そのためにも、新しい電子すかし挿入アルゴリズムの実装のフレームワークとなるソースコード操作ツールが重要である。

筆者はプログラミング言語に関する研究の道具として用いるために拡張可能 Java プリプロセッサ EPP[10] を設計・実装した[1]。EPP は、電子すかしを実装するためのソースコード操作ツールキットとしても利用することができる。以下の節では、EPP について述べる。

[1]EPP は、拡張可能コンパイラキット Lods [1] のアーキテクチャをシンプルにして、プリプロセッサに特化させたシステムである。また、文法の広範囲な拡張を可能にしている。

```
#epp load "swap"
public class test {
  public static void main(String[] argv){
    int a = 1, b = 2;
    swap(int, a, b);
  }
}
```

図 1: EPP plugin を使ったプログラムの例

6.2 拡張可能 Java プリプロセッサキット EPP

EPP は次の 3 通りの使い方ができるシステムである。

1. Java プログラムのソースコード操作ツール
2. Java 言語のためのマクロプリプロセッサ
3. 拡張 Java 言語実装のためのフレームワーク

EPP は、それだけでは Java のソースコードを入力として受けとり、意味を変えずに Java のソースコードを出力するプリプロセッサである。しかし、極めて高い拡張性を持っている。EPP に入力するソースコードの先頭に、追加したい EPP plugin を指定することにより、様々な拡張機能が利用できる。図 1は、 swap マクロを定義する plugin を指定した Java プログラムの例である。また、複数の plugin を同時に指定することもできる。個々の plugin があるマナーに従って書かれていれば、plugin どうしが衝突して不具合を起こすことはない。EPP は Java のソースコードを構文解析して、構文解析木にする。そして、EPP plugin はこの構文解析木に対して様々な操作を施すことができる。

EPP が持つ Java パーザは、基本的に再帰下降のスタイルで書かれている。また、拡張のための “hook” が多く埋め込んであり、「継承」によってパーザの動作を拡張することによって、新しい文法ルールを Java 言語に追加することができる。

EPP は、 system mixin と呼ぶソフトウエア部品の再利用性を高める特殊な継承機構を持ったオブジェクト指向言語、 Ld-2 によって書かれている。Ld-2 は現在 Common Lisp の上に実装されている。しかし、現在 system mixin の機能を Java に追加し、それを用いて EPP を全て Java に書き直す作業を行なっている。

6.2.1 モジュール化のメリット

ソースコード操作ツールとして見た場合の EPP のメリットは、文法ルールの定義がモジュール化されている

ことと、ソースコードへの操作もモジュールとして定義できるという点である。

そのため、将来 Java の文法が多少拡張されても、多くの「電子すかし挿入 plugin」は全く変更せずに使えると予想される。また、文法の拡張された Java がすでに多く存在するが、それらの多くに対しても plugin を全く変更せずに適用することができる。

また、構文解析木に対する操作を容易にするため、パターンマッチングの機能、 Lisp の backquote macro に相当する機能などを今後提供する予定である。

6.2.2 電子すかしの実装例

EPP を使った電子すかしの例として、 if-then-else の変換による電子すかしを実装した。これは Common Lisp で 55 行のプログラムである。実装の概略は以下のとおり。

1. EPP に電子すかし挿入パスを追加する。
2. 挿入すべき bit 列をキューにして、大域変数に入れておく。
3. 電子すかし挿入パスでは、ソースコードの抽象構文木を再帰的にたぐって、 if 文がないかどうかをさがす。
4. if 文があれば、 bit 列のキューから 1bit とりだし、 1 ならば条件式に ! をつけて、 then 部と else 部を入れ換える。この if 文の subtree はこれ以上変換しない。

6.3 その他のソースコード操作ツール

6.3.1 lisp

tree に対する操作を行なうための最も強力な言語として lisp がある。つまり、 lisp はそれ自体がソースコード操作ツールである。Common Lisp および scheme 上のフリーの LALR の parser generator も存在している。

6.3.2 Sage++

C++ および FORTRAN のソースコード操作ライブラリとして sage++[6] というシステムがある。これは、ソースコードをパーズし、 tree にして操作しやすくしたあと、再びソースコードに変換する。ソースコードレベルの最適化や、並列 C++ などが Sage++ を使って実装されている。

6.3.3 MPC++, OpenC++

C++ のためのコンパイル時 MOP を備えた言語として MPC++[9], OpenC++[8] がある。これらは最適化、言語拡張のためのツールであり、ソースコードに対する操作をそのソースコード自身の中で記述する。

7 電子すかしに関連する他の研究課題

ソフトウエア電子すかしに密接に関連する研究研究課題として以下のものがある。いずれもセキュリティに深く関連しておりかつ暗号技術が利用が不可能であり、プログラミング言語研究者による研究が望まれる。

7.1 ソフトウエア難読化

ソフトウエア難読化とは、プログラムのソースコードあるいはバイナリコードを、その意味を変えずに変換することにより、他人によるプログラムの意味の解析をできるだけ困難にすることを言う。難読化については最近いくつかの研究がある [2, 3, 4, 5]。

難読化は以下の点で電子すかしと密接に関連する。

- **電子すかしの安全性を増すための難読化**
  難読化により解読攻撃の危険性を減らすことができる。

  また、難読化は単純な結託攻撃に対する対策ともなる。結託攻撃は複数のユーザがソースコードを持ちよってそれを相互に見比べることで挿入された情報を消す方法である。個々のソースコードに異なった難読化を施し、プログラムの表面的な構造を全く異なったものにすることで、結託攻撃を難しくできる。
- **電子すかしに対する攻撃法としての難読化**
  挿入アルゴリズムが単純な場合、電子すかし入りソフトウエアに対して難読化を施すことによって、挿入情報が完全に完全に破壊されてしまう。情報が破壊されていない場合でも、配布ソフトウエアから挿入された情報を機械的に取り出すことは一般に難しくなる。
- **アルゴリズム・実装上の共通点**
  電子すかしと難読化はいずれも「プログラムの表現の冗長性」を利用しているため、難読化アルゴリズムがそのまま電子すかし挿入アルゴリズムになる場合が多い。

  例えば難読化の最も素朴な方法として、変数名を意味のない記号に付け変える方法が考えられるが、これはそのまま電子すかしのアルゴリズムとしても利用できる。

7.2 ソフトウエア難改変化

電子すかしに対する攻撃はいずれもソフトウエアに対する改変を必然的に伴う。もし、任意のソフトウエアを

```
#define BUFSIZE 1024
#define STRLENGTH 1024

main(){
 ... BUFSIZE ...;
 ... STRLENGTH ...;
 ... BUFSIZE ...;
 ... STRLENGTH ...;
}
```

図 2: 改変しやすいプログラム

```
#define CONST1024 1024
#define CONST512 512

main(){
 ... CONST1024 ...;
 ... CONST1024 ...;
 ... CONST512 * 2 ...;
 ... CONST512 * 2 ...;
}
```

図 3: 改変しにくいプログラム

極めて改変しにくいソフトウエアに自動変換する手段が存在するならば、それは電子すかしの安全性を増す強力な武器となる。

従来、プログラミング言語の研究者は、保守性・拡張性の高いプログラミングスタイルおよび言語機能を研究してきた。定数定義、サブルーチン、構造化プログラミング、モジュール化、オブジェクト指向とその手段は発展してきている。これらの手段は、そのまま裏返すことで難改変ソフトウエアの技術として用いることができると考えられる。

図 2、図 3は、定数定義の変換による難改変化の例である。もともと同じ意味の定数を指すべきところを異なる定数に変え、違う意味の定数を同じ定数で表すことにより、定数の改変を難しくしている。

同様の手法をサブルーチンに適用することで、プログラムの改変を難しくすることができるであろう。プログラムを複数の手続きに分割して記述すれば保守性が増すが、その逆、すなわち手続き呼び出しの inline 展開は難改変化に有効である。また、「本来全く別の役割を持つ似たコード」を無理矢理 unify して共有コードにすれば、その部分の変更がやりにくくなる。

7.3 にせモジュール生成

最適化、電子すかし、難読化はいずれもプログラムの意味を変えない変換だが、全く逆に、一見似ているが意味が全く違う（あるいは微妙に違う）プログラムを自動生成する「にせモジュール化」も可能である。にせモジュールを配布するソースコードに混ぜることにより、解読作業を混乱させることができる。また、にせモジュール生成は、 assert 法を実装する際に、追加する else 部のコードを決めるためにも必要である。

7.4 ソフトウエア筆跡鑑定

プログラマの癖を抽出し、作者不明のコードから作者を推定することも、おそらくある程度可能である。具体的には、使用するライブラリの種類、構文の使い方、変数名の付け方などを利用することが考えられる。もし他人の「筆跡」を完全にまねることが不可能であることが証明されれば、これもソフトウエアの著作権保護に役立つ技術となる。

8 まとめ

ソフトウエアの違法コピーを抑止するための、ソフトウエア電子すかしについて述べた。電子すかしの挿入法としていくつか素朴な方法と比較的有望と思われる assert 法を提案し、それらに対する攻撃法と利便性の評価法について述べた。また、電子すかしアルゴリズム等を実装するためのツールとして用いることができる、拡張可能プリプロセッサ EPP について述べた。

現在のところ実用的な挿入ツールの実装にまでは至っていないが、 assert 法を発展させることで、攻撃がしにくくかつ挿入アルゴリズムの公開可能なツールが実装できるものと考えている。

謝辞

本研究に関して議論していただいた奈良先端大の門田さんに感謝します。

参考文献

[1] 一杉裕志, 平野聡, 田沼均, 須崎有康, 塚本 享治: ”コンパイラキット Lods における交換可能なＧＣモジュールの構築法”, プログラミングシンポジウム, Jan, 1997.

[2] 満保雅浩、岡本栄司：”ソフトウエアの難読化における課題”、1997 年暗号と情報セキュリティシンポジウム講演論文集,SCIS97-10A, Jan, 1997.

[3] 村山隆徳、満保雅浩、岡本栄司：”プログラムコードの難読化について”、1996 年暗号と情報セキュリティシンポジウム講演論文集,SCIS96-8D, Jan, 1996.

[4] 門田暁人、高田義広、鳥居宏次：”プログラムの難読化法の提案”, 情報処理学会第５１回全国大会講演論文集、5G-7, pages 4–263, 1995.

[5] 門田暁人、高田義広、鳥居宏次：”プログラムの難読化法の実験的評価”、ソフトウエア工学, Vol.108,No.5, Mar, 1996.

[6] Bodin, F. et al: ”Sage++: A Class Library for Building Fortran and C++ Restructuring Tools”, In Proc. of Object-Oriented Numerics Confs., Oregon, Apr.1994.

[7] Boneh, D. and Shaw, J.: ”Collusion-Secure Fingerprinting for Digital Data”, Advances in Cryptology Proceedings of CRYPTO’95, LNCS 963, pp.452–465, 1995.

[8] Chiba, S.: ”A Metaobject Protocol for C++”, In Proc. of OOSPLA’95, pp.285–299, 1995.

[9] Ishikawa, Y.: ”Meta-level Architecture for Extendable C++ Draft Document”, Technical Report TR-94024, RWCP, 1994.

[10] Ichisugi, Y.: EPP and Lods home page. http://www.etl.go.jp/etl/bunsan/~ichisugi