社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.
ISEC2002-98 (2002-12)

命令コードの実行時置き換えによるプログラムの解析防止

神崎 雄一郎　門田 暁人　中村 匡秀　松本 健一

奈良先端科学技術大学院大学 情報科学研究科 〒630-0101 奈良県生駒市高山町 8916-5

E-mail: {yuichi-k, akito-m, masa-n, matumoto}@is.aist-nara.ac.jp

あらまし

本稿では，ソフトウェアを不正な解析行為から保護するための一手法を提案する．キーアイデアは，命令コードを自己書き換えする仕組みをプログラムに追加することで，プログラムの解析を困難にすることである．提案方式によって得られる機械語プログラムは，多数の箇所がダミーの命令コードでカムフラージュされており，各々が実行時のある期間だけ正しい命令に置き換えられる．解析者がダミーの命令コードを含む部分を読むと誤った理解をすることになり，解析に失敗する．提案する方式は，特別なハードウェアを必要とせず，低コストで著しく解析が困難なソフトウェアが実現可能である．

キーワード　ソフトウェア保護，プログラムの難読化，自己書き換え

Protecting Software Programs by Replacing Instructions at Run-time

Yuichiro KANZAKI　Akito MONDEN　Masahide NAKAMURA　Ken-ichi MATSUMOTO

Graduate School of Information Science, Nara Institute of Science and Technology

8916-5 Takayama, Ikoma, Nara, 630-0101 Japan

E-mail: {yuichi-k, akito-m, masa-n, matumoto}@is.aist-nara.ac.jp

Abstract

In this paper, we present a new method to protect software against illegal acts of hacking. The key idea is to add a mechanism of self-modifying codes to the original binary program, so that the original program becomes hard to be analyzed. In the binary program obtained by the proposed method, the original code fragments we want to protect are camouflaged by dummy instructions. Then, the binary program autonomously retrieves the original code fragments within a certain period of execution, by replacing the dummy instructions with the original ones. Since the dummy instructions are completely different from the original ones, code hacking fails if the dummy instructions are read as they are. Moreover, the dummy instructions are scattered over the program, therefore, they are hard to be identified. As a result, the proposed method helps to construct highly invulnerable software without special hardware.

Keyword　Software Protection,　Program Obfuscation,　Self-modifying Code

1. はじめに

従来より，不正行為を目的としたソフトウェアの解析･改ざんが問題となっており,不正行為者の存在は，ソフトウェア業界における脅威となっている.例えば，コピープロテクトのチェックルーチンを解析し，そのルーチンが無効になるように改ざんする不正コピーの問題は後をたたず，開発者側に大きな不利益を生じさせている.

近年普及しているコンテンツ流通のシステムに関しても，不正行為者に攻撃を受け，被害が生じる危険性について案じられている[2]．図1は，SDMI(Secure Digital Music Initiative)準拠のコンテンツ流通システムのうち，ユーザのパソコン上で実行される部分を示したものである[2].不正行為者が解析によって復号アルゴリズムや秘密鍵を知ることに成功した場合，不正にコンテンツを入手される可能性がある．このようなコンテンツ流通システムに対する攻撃によって生じる問題が拡大しないためにも，不正行為を目的とした内部解析や改ざんを防止することのできる技術が必要不可欠である.

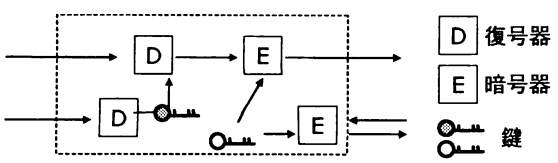

図1　SDMI準拠のコンテンツ流通システムの一部

プログラムの解析を困難にする技術は，次章に詳しく述べるように，従来から盛んに論じられ，プログラムの難読化や暗号化など，いくつかの手法が提案されている．しかし，いずれの方式も保護の脆弱さや実装面の難しさに関する問題を抱えており，新しいアプローチが求められている.

本稿では，ソフトウェアを不正な解析行為から保護することを目的として，与えられた任意のアセンブリプログラムから，正しく解析することの困難な自己書き換えプログラムを作成する系統的な方式を提案する.提案方式によって得られる機械語プログラムは，多数の箇所がダミーの命令コードでカムフラージュされており，各々が実行時のある期間だけ正しい命令に置き換わる仕組みを有している.

2. 従来技術とその問題点

解析が困難なプログラムの作成技術は，従来よりいくつか提案されており，それらの技術は「プログラムの難読化」,「プログラムの暗号化」,「プログラムの断片化」の3つに大別できる．いずれの方法も，プログラムの解析に要するコスト（労力，時間）を増大させる効果がある.

プログラムの難読化は，与えられたプログラムを読みにくい（複雑な）プログラムに変換することで，解析に要するコストを増大させる技術である（図2）．難読化したプログラムは，その表現や計算手順が複雑化しており，人間にとって解析が困難となっているが，難読化していないプログラムと同様，計算機上で実行が可能である．開発したプログラムを難読化してからプログラムの使用者(ユーザ)へ配布することで,ユーザや第三者によってプログラムが解析される危険性を減らすことができる．プログラムの難読化の具体的な方式としては，プログラムの制御構造を複雑にする方式[12][18][19],「複雑な処理を行う命令コード」を，複数の「単純な処理を行う命令コード」の組み合わせに置き換える方式[17][20]，プログラムの実行結果に影響を与えない（無意味な）プログラムコードを挿入する方式[7][8]，データ構造を変形する方式[9]，プログラム中の手続きの名前（メソッド名）を変換する方式[23]，配列やポインタの参照，代入を利用してプログラムを複雑化する方式[21][25]などがある.

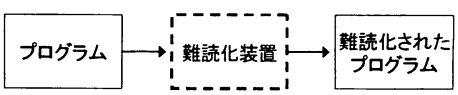

図2　プログラムの難読化

プログラムの暗号化は，プログラムの全体または一部を暗号化することによって，解析を困難にする技術である（図3）．暗号化された部分のプログラム（図3では「暗号化された命令コード」として示されている部分）は，人間が読んでその内容を理解することはできない．ただし，暗号化された部分のプログラムはそのままでは計算機によって実行できないため，プログラムの実行前，もしくは，実行中に必ず復号（暗号の逆変換）されることになる．従って，復号を行うための機構（図3では「暗号化された命令コードを復号するモジュール」として示されている部分）をあらかじめプログラムに追加しておく，もしくは，復号を行うハードウェアをあらかじめ計算機に追加しておく必要がある．プログラムの暗号化の具体的な方式は，文献[1][5][10][11][13][24]などにおいて提案されている.

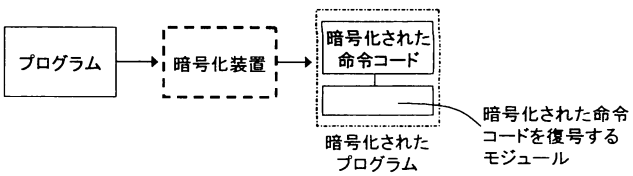

図3　プログラムの暗号化

プログラムの断片化は，与えられたプログラムを多数のプログラム断片に分割し，それらの実行順序を制御する技術である（図4）．プログラムの難読化，および，暗号化の技術と併用される場合もある．個々のプログラム断片を人間が読んでも，プログラム全体の動作が理解できないため，プログラムの解析は困難である．断片化の具体的な方式は，文献[3][4][14][15][16]などにおいて提案されている．

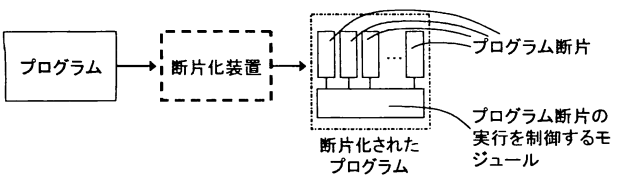

図 4　プログラムの断片化

上記した従来技術は，下記の問題点を有している．

難読化されたプログラムは時間をかければ解析される危険性がある．また，難読化の方式によっては必ずしも自動化（システム化）が容易でない．

暗号化されたプログラムは人間が読んでも理解できないため解析が困難であるが，プログラムの実行前，もしくは，実行中に暗号化された命令コードが必ず復号されるため，復号後の命令コードが解析される危険性がある．また，「暗号化された命令コードを復号するモジュール」が解析，改ざんされ，暗号化によるプログラムの保護が容易に無効化されてしまう危険性がある．ハードウェアにより復号処理を行う場合は解析の危険性が減るが，ソフトウェアのみを用いる場合と比べて製品のコストが高くなるという問題がある．

断片化されたプログラムは，「プログラム断片の実行を制御するモジュール」が解析され，断片化によるプログラムの保護が容易に無効化されてしまう危険性がある．ハードウェアでプログラム断片の実行順序を制御する場合は解析の危険性が減るが，ソフトウェアのみを用いる場合と比べて製品のコストが高くなるという問題がある．また，難読化や暗号化と比べると自動化が難しい場合があり，商品化が必ずしも容易でない．

以上のように，難読化，暗号化，断片化のいずれの技術についても問題点を有しており，プログラムの不正な解析を防止するには不十分である．

3. 提案する方式

3.1. 概要

提案する方式の基本構成を図5に示す．自己書き換え処理追加装置は，入力されたアセンブリプログラムをもとに，自己書き換えを行うプログラムを出力する．得られる自己書き換えプログラムは，一部分がダミーの命令コードYでカムフラージュされており，実行時のある期間だけ正しい命令yに書き換えられる．解析者が自己書き換えプログラムの解析を試みたとき，ダミーの命令コードYを含む部分を読むと誤った理解をすることになり，解析に失敗する．カムフラージュする箇所を数多く設けることで，プログラムの解析および改ざんが著しく困難になる．

自己書き換えプログラムは，カムフラージュする箇所に正しい命令コードyを書き込むための処理ルーチン（図5において「正しい命令コードの書き込み処理ルーチン」で示される），および，カムフラージュする箇所に正しい命令yを書き込んだ後に再びダミーの命令コードYを書き込むためのルーチン（図5において「ダミーの命令コードの書き込み処理ルーチン」で示される）を含むが，それらはサイズが小さく，かつ，プログラム中に分散されているために，解析者がそれらを発見することは困難である．そのため，解析者がプログラム中の全てのダミーの命令コードを特定しカムフラージュを無効化することは著しく困難である．

提案方式は自動化が可能であり，しかも，多くの計算機環境において新たなハードウェアを追加せずに実施が可能である．また，難読化や暗号化などの従来技術と対立するものではないため，提案方式と従来技術を併用することで不正な解析および改ざんの防止効果を高めることができると考えられる．

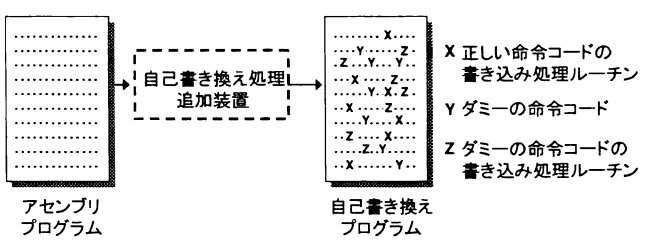

図 5　基本構成

3.2. 実施手順

提案方式における自己書き換え処理追加装置では，次に示す(ステップ 1)～(ステップ 6)を順に実施することによって，自己書き換えプログラムを生成する．ここで，Xは正しい命令コードの書き込み処理ルーチン，Yはダミーの命令コード，Zはダミーの命令コードの書き込み処理ルーチンを表す．また，P(X), P(Y), P(Z)はそれぞれX, Y, Zのプログラム上の位置と定義する．

(ステップ 1)

図6に例示するように，アセンブリプログラムに対して，次の3つの条件を満たすようにP(X), P(Y), P(Z)

を決定する.

(条件 1) プログラム開始点から P(Y)へ至る全ての制御フロー上に P(X)が存在する.

(条件 2) P(X)から P(Y)へ至る全ての制御フロー上に P(Z)が存在しない.

(条件 3) P(Z)から P(Y)へ至る全ての制御フロー上に P(X)が存在する.

ここで, X は正しい命令コード y を P(Y)に書き込む処理を行うルーチンであり, P(X)は X を挿入する位置を示す. また, Y は正しい命令コード y をカムフラージュするためのダミーの命令コードであり, P(Y)は Y によるカムフラージュを行う位置を示す. また, Z はダミーの命令コード Y を P(Y)に書き込む処理を行うルーチンであり, P(Z)は Z を挿入する位置を示す.

(ステップ 2)

正しい命令コード y と同じ命令長を持つ, y をカムフラージュするためのダミーの命令コード Y を決定する. ここで, y はステップ(2)の実行前に P(Y)に存在する命令コードを指す. ここでいう命令長とは, 命令コードの機械語表現のバイト数のことである.

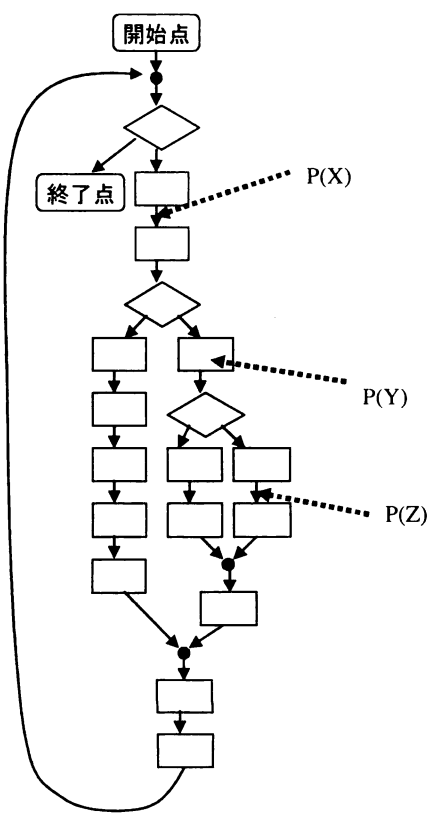

図 6　制御フロー上の P(X), P(Y), P(Z)

(ステップ 3)

正しい命令コードを P(Y)に書き込む処理を行うルーチン X と, ダミーの命令コード Y を P(Y)に書き込む処理を行うルーチン Z を生成する.

(ステップ 4)

P(X)に X を挿入し, P(Z)に Z を挿入し, P(Y)に Y を書き込む.

(ステップ 5)

X および Z が, 人間にとって読みにくい (複雑な) 表現を持つと, プロテクトの無効化がより困難になる. そのために, ここで X および Z を難読化することが望ましい.

(ステップ 6)

(ステップ 1)～(ステップ 5)の実行の結果得られた自己書き換えプログラムを前記アセンブリプログラムであると見なし, (ステップ 1)～(ステップ 5)を再帰的に 0 回以上繰り返す. ただし, 各繰り返しにおいては, 同一の箇所をカムフラージュするとは限らない. つまり, 各繰り返しにおける X, Y, Z, P(X), P(Y), P(Z), y は, それまでの繰り返しにおける X, Y, Z, P(X), P(Y), P(Z), y と異なっていてよい.

前記 X および Z は, 解析者に発見されにくくするために, できるだけサイズを小さくすることが望ましい. そのためには, y および Y を機械語表現に変換して比較した場合に 1 バイトだけが異なるという条件を満たすように Y を決定し, X および Z における書き込み処理を, その異なる部分だけに限定することが望ましい.

前記の(ステップ 1)～(ステップ 5)によって得られた自己書き換えプログラムは, そのままでは実行できないため, アセンブラに入力して機械語プログラムを生成し, 次に, その機械語プログラムをリンカに入力して実行可能プログラムを生成し, さらに, その実行可能プログラムに対して実行時書き換え可能(writable)の属性を付けるステップが必要となる.

また, (ステップ 1)への入力となるアセンブリプログラムは, 高級言語で記述されたソースプログラムをコンパイルするステップ, または, 機械語プログラムを逆アセンブルするステップを, (ステップ 1)に先だって実施することで生成できる.

プログラム終了時に P(Y)が必ずカムフラージュされた状態になることを保証したい場合は, 前記 P(X), P(Y), P(Z)を決定する(ステップ 2)において, 「P(Y)からプログラム終了点へ至る全ての制御フロー上に P(Z)が存在する」という条件をさらに追加する.

また, P(X)および P(Y)の候補となる位置を増やしたい場合には, 前記 P(X), P(Y), P(Z)を決定するステップ(2)において P(Z)の決定を省き, 前記 P(X)への X の挿入と, P(Z)

への Z の挿入と，P(Y)への Y の書き込みとを行う(ステップ 3)において，P(Z)への Z の挿入を省く．

4. ケーススタディ

提案方式の対象となるのは，アセンブリプログラムであるが，多くの場合，アセンブリプログラムは高級言語で記述されたソースプログラムをコンパイルすることによって生成される．ここでは，典型的な実施例として，C 言語で記述されたソースプログラムを元に，解析および改ざんが困難な自己書き換えプログラムを作成する実施手順の例を示す．

まず，ユーザは，コンパイラシステムを用いてソースプログラムをアセンブリプログラムに変換する．いま，一例として図 7 のような C 言語で記述されたソースプログラムを考える．このプログラムは，プログラムユーザにシリアル番号の入力を求め，それが正しければ（この例では入力が 123 であれば），先に続く処理に移ることができるが，正しくなければ，再びシリアル番号の入力を求める処理に戻るという単純なプログラムである．このソースコードをコンパイラ（gcc など）によってアセンブリプログラムに変換する．例として，図 8 に i386 系のプロセッサのアセンブラコードを示す（一部のみ抜粋）．自己書き換え処理プログラムは，アセンブリプログラムから，次のようなステップで自己書き換えプログラムを生成する．

```
int main(void) {
    long n;

    while(1) {
        printf("Enter Your Serial Number: ");
        scanf("%ld", &n);

        if(n == 123) {
            break;
        } else {
            printf("Wrong!¥n");
        }
    }
    printf("Correct¥n");
        :
    return 0;
}
```

図 7　実施例のソースプログラム

```
        :
    pushl %ebp
    movl %esp,%ebp
    subl $24,%esp
    call ___main
L5:                                  P(X)
    addl $-12,%esp
    pushl $LC0
    call _printf
    addl $16,%esp
    addl $-8,%esp
    leal -4(%ebp),%eax
    pushl %eax
    pushl $LC1
    call _scanf
    addl $16,%esp                    P(Y)
    cmpl $123,-4(%ebp)
    jne L6                           P(Z)
    jmp L4
        :
```

図 8　実施例のアセンブリプログラム

(ステップ 1)

アセンブリプログラムの制御の流れを解析し，前記の P(Y), P(X)および P (Z), すなわち，ダミーの命令コード Y によって偽装を行う位置，正しい命令コード y を P(Y)に書き込む処理を行うルーチン X の挿入位置およびダミーの命令コード Y を P(Y)に書き込む処理を行うルーチン Z の挿入位置を決定する（図 8 参照）．

この例では，プログラム開始点から P(Y)へ至るまでに必ず通る制御フロー上に P(X)が存在し，P(X)と P(Y)を結ぶ唯一の制御フロー上に P(Z)が存在しない．また，P(Z)から P(Y)へ至るまでに必ず通る制御フロー上に P(X)が存在する．したがって，挿入された P(X), P(Y), P(Z)は，実施手順の(ステップ 1)で述べた条件をすべて満たしている．

(ステップ 2)

正しい命令コード y の機械語表現と同一のサイズの機械語表現を有する命令コード Y を決定する．ここでの y は「cmpl $123, -4(%ebp)」であり，その機械語表現(16 進数表現)は「83 7D FC 7B」となり 4 バイトのサイズを有する．従って，命令コード Y は 4 バイトの機械語表現を持つ必要がある．一例として，y および Y を機械語表現に変換して比較した場合に 1 バイトだけが異なるという条件を満たすように Y を決定する．ここでは，「83 4D FC 7B」という機械語表現を持つ「or $123, -4(%ebp)」を Y に決定した．

(ステップ 3)

正しい命令コードを P(Y)に書き込む処理を行う

ルーチンXと，ダミーの命令コードYをP(Y)に書き込む処理を行うルーチンZを生成する．

(ステップ 4)

P(X)にXを挿入し，P(Z)にZを挿入し，P(Y)にYを書き込む．図8のアセンブリプログラムにこれらの処理を加えた例を図9に示す．XおよびZの先頭に「pushl %eax」が入っており，最後に「popl %eax」が入っているが，これはXおよびZの挿入によってプログラムが誤動作しないように配慮したためである．

(ステップ 5)

挿入されたXおよびZを難読化する．これは，挿入されたルーチンを解析者に発見されにくくするための処置である．この例では，「書き込み先P(Y)をX，Z両方のルーチンにおいて同じ方法で指定すると，X，Zの位置，あるいはYの位置が発見されやすい」ということに注目して，X，Zがそれぞれ別の方法でP(Y)を指定するように変更している．具体的には，P(Y)を指定するときに，P(Y)の位置を示すラベル（図9におけるCFLOC_1）を用いるのではなく，他のラベルを2箇所に挿入し，XおよびZがそれぞれ異なったラベルを用いて間接的にP(Y)を指定するように変更している．XおよびZに難読化処理を加えた後のアセンブリプログラムを図10に示す．

(ステップ 6)

前記ステップ(1)～(5)の処理をプログラムの多くの箇所に対して繰り返し行う．

自己書き換え処理追加装置の出力として得られたアセンブリプログラムをコンパイラシステムによってアセンブルし，機械語プログラムを得る．必要であればリンクの実行，コード領域への書き込みを許可するフラグを立てるなどの処理を行い，実行時に自分自身のコード領域を書き込むことが可能な実行可能プログラムを得る．これによって得られた実行可能プログラムは，解析および改ざんが困難な自己書き換えプログラムとなる．

```
        :
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp
        call ___main
L5:
        pushl %eax
        movl $CFLOC_1, %eax      X
        movb $125, 1(%eax)
        popl %eax
        addl $-12,%esp
        pushl $LC0
        call _printf
        addl $16,%esp
        addl $-8,%esp
        leal -4(%ebp),%eax
        pushl %eax
        pushl $LC1
        call _scanf
        addl $16,%esp
CFLOC_1:
        or $123,-4(%ebp)         Y
        pushl %eax
        movl $CFLOC_1, %eax      Z
        movb $77, 1(%eax)
        popl %eax
        jne L6
        jmp L4
        :
```

図 9 自己書き換え処理を追加したアセンブリプログラム

```
        :
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp
        call ___main
L5:
        pushf
        pushl %eax
        movl $A2, %eax           難読化後のX
        subl $20, %eax
        movb $125, 1(%eax)
        popl %eax
        popf
        addl $-12,%esp
        pushl $LC0
        call _printf
A1:
        addl $16,%esp
        addl $-8,%esp
        leal -4(%ebp),%eax
        pushl %eax
        pushl $LC1
        call _scanf
        addl $16,%esp
        or $123,-4(%ebp)         Y
        pushf
        pushl %eax
        movl $A1, %eax           難読化後のZ
        addl $23, %eax
        movb $77, 1(%eax)
        popl %eax
        popf
A2:
        jne L6
        jmp L4
        :
```

図 10 自己書き換え処理が追加されたアセンブリプログラム（XおよびZを難読化した場合）

5. おわりに

本稿では，ソフトウェアを不正な解析行為から保護することを目的として，任意のアセンブリプログラムから解析が難しい自己書き換えプログラムを作成するための系統的な方法を提案した．現在著者らは，提案方式を実施するためのシステムの試作を行っている．

今後の課題としては，システムの実装を完成させること，および，自己書き換え処理を追加することによって機械語プログラムに与えられる影響（実行効率やプログラムサイズの変化など）を評価することなどが挙げられる．

参考文献

[1] D.J. Albert and S.P. Morse, "Combating software piracy by encryption and key management," IEEE Computer, pp.68-73, April 1984.

[2] 穴澤健明,"モバイル音楽配信とそのセキュリティ保護について,"電子情報通信学会研究会資料，オフィスシステム研究ワークショップ，pp.3-12, 2001.

[3] D. W. Aucsmith, "Tamper Resistant Software: An Implementation," In R. J. Anderson ed. Information Hiding Workshop, Lecture Notes in Computer Science, Vol. 1174, pp.317-333, 1996.

[4] D. W. Aucsmith and G. L. Graunke, "Tamper resistant methods and apparatus," United States Patent, No. 5,892,899, Assignee: Intel Corporation, Apr. 1999.

[5] R. M. Best, "Crypto microprocessor for executing enciphered programs," United States Patent, No. 4,278,837, July 1981.

[6] H.Chang and M.Atallah, "Protecting Software Codes By Guards," In Workshop on Security and Privacy in Digital Rights Management 2001, LNCS 2320, Springer-Verlag, pp.160-175, 2001.

[7] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Technical Report of Dept. of Computer Science, U. of Auckland, No.148, New Zealand, 1997.

[8] C. Collberg, C. Thomborson, and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL98), San Diego, California, 1998.

[9] C. Collberg, C. Thomborson, and D. Low, "Breaking Abstractions and Unstructuring Data Structures," IEEE International Conference on Computer Languages(ICCL'98), Chicago, IL, May 1998.

[10] C. N. Drake, "Computer software authentication, protection, and security system," United States Patent, No. 6,006,328, Dec. 1999.

[11] B. E. Hampson, "Digital computer system for executing encrypted programs," United States Patent, No. 4,847,902, Assignee: Prime Computer, Inc., July 1989.

[12] F. Hohl, "Time limited blackbox security: Protecting mobile agents from malicious hosts," In G. Vigna ed. Mobile Agents Security, Lecture Notes in Computer Science, Vol. 1419, pp.92-113, Springer-Verlag, 1998.

[13] 石間宏之，斉藤和夫，亀井光久，申吉浩，"ソフトウェアの耐タンパー化技術," 富士ゼロックステクニカルレポート No.13, pp.20-28, 2000.

[14] 鴨志田昭輝，松本勉，井上信吾，"耐タンパーソフトウェアの構成手法に関する考察," 信学技報，ISEC97-59, pp.69-78, 1997.

[15] J. M. Nardone, R. T. Mangold, J. L. Pfotenhauer, K. L. Shippy, D. W. Aucsmith, R. L. Maliszewski, G. L. Graunke, "Tamper resistant methods and apparatus," United States Patent, No. 6,178,509, Assignee: Intel Corporation, Jan. 2001.

[16] J. M. Nardone, R. P. Mangold, J. L. Pfotenhauer, K. L. Shippy, D. W. Aucsmith, R. L. Maliszewski, and G. L. Graunke, "Tamper resistant methods and apparatus," United States Patent, No. 6,205,550, Assignee: Intel Corporation, Mar. 2001.

[17] M. Mambo, T. Murayama, and E. Okamoto, "A tentative approach to constructing tamper-resistant software," In Proc. New Security Paradigm Workshop, Cumbia, UK, 1997.

[18] 門田暁人，高田義弘，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, pp.4-263-4-264, 1995.

[19] 門田暁人，高田義弘，鳥居宏次，"ループを含むプログラムを難読化する方法の提案," 電子情報通信学会論文誌 D-I, Vol.J80-D-I, No.7, pp.644-652, July 1997.

[20] 村山隆徳，満保雅浩，岡本栄司，植松友彦，"ソフトウェアの難読化について," 電子情報通信学会技術研究報告, ISEC95-25, Nov. 1995.

[21] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, "Software tamper resistance based on the difficulty of interprocedural analysis," In Proc. International Workshop on Information Security Applications (WISA2002), pp. 437-452, August 2002.

[22] 岡村久道，インターネット訴訟 2000，ソフトバンクパブリッシング，東京，2000.

[23] P. M. Tyma, "Method for renaming identifiers of a computer program," United States Patent, No. 6,102,966, Assignee: PreEmptive Solutions, Inc., Aug. 2000.

[24] W. Paulini, and D. Wessel, "Process for securing and for checking the integrity of the secured programs," United States Patent, No. 5,224,160, Assignee: Siemens Nixdorf Informations systeme AG, June 1993.

[25] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: Obfuscating static analysis of programs," Technical Report SC-2000-12, Department of Computer Science, University of Virginia, Dec. 2000.