All rights are reserved and copyright of this manuscript belongs to the authors. This manuscript has been published without reviewing and editing as received from the authors: posting the manuscript to SCIS 2007 does not prevent future submissions to any journals or conferences with proceedings.

SCIS 2007 The 2007 Symposium on Cryptography and Information Security
Sasebo, Japan, Jan. 23-26, 2007
The Institute of Electronics, Information and Communication Engineers

高級言語レベルでの偽装内容の指定が可能なプログラムのカムフラージュ

Software Protection that Specifies Camouflaged Instructions at the High-Level Language Level

神崎 雄一郎 * 門田 暁人 † 中村 匡秀 † 松本 健一 †
Yuichiro Kanzaki Akito Monden Masahide Nakamura Ken-ichi Matsumoto

あらまし 本論文では，プログラムをカムフラージュ(偽装) することにより，悪意を持ったユーザ (攻撃者) によるソフトウェアの解析を困難にする系統的な方法を提案する．提案方法のユーザは，保護の対象となるソフトウェアのソースコードについて，高級言語のレベルで命令の変更，追加あるいは削除を繰り返し行い,「見せかけ」のソースコードを作成する．カムフラージュされたバイナリプログラムは，見せかけのソースコードに対応する命令コード列 (コード) と，それを実行時に命令単位で元来のコードに書き換える自己書換え命令によって構成される．攻撃者が逆アセンブラなどを用いてプログラムの静的解析を試みるとき，プログラム全体に散在する自己書換えルーチンを見つけ出してそのルーチンの動作を理解しない限り，見せかけのコードの内容しか知ることはできない．結果として攻撃者は，広範囲にわたるプログラムの解析を強いられることとなる．

キーワード 著作権保護，ソフトウェア保護，プログラムの難読化，プログラムの暗号化，自己書き換え

1 はじめに

ソフトウェア内部の秘密情報，例えばライセンスをチェックする命令文，商業的価値の高いアルゴリズム [2]，DRM システムの復号鍵 [1,7] などは，悪意を持ったユーザの不正な解析行為 (不正なリバース･エンジニアリング) によって，第三者に知られる危険にさらされている [4,6]．このような秘密情報が第三者に知られると，ソフトウェアベンダやコンテンツプロバイダが多大な損害を被る場合がある．例えば，DVD 再生プログラムの解析結果に基づいて作成された DVD データの暗号解除ツールの流通は DVD の違法コピーを助長し，映像コンテンツのプロバイダに大きな損失を与えた [8]．Web サイトなどを通してソフトウェアの解析を行うためのツールやテクニックがエンドユーザに広く浸透している昨今では，ソフトウェアの不正な解析を防止する技術への要求が高くなっている．

ソフトウェアを不正な解析から守る方法として，これまで数多くのプログラム暗号化法，難読化法，アンチデバッグ技術が提案されてきた [6]．本論文では，本論文と同著者によって提案された命令のカムフラージュ法 [9] について，拡張方式を提案する．

提案方法では，まず提案方法の使用者 (以降，ユーザと呼ぶ) が，保護の対象となるソフトウェアのソースコードについて，高級言語のレベルで命令の変更，追加あるいは削除を繰り返し行い,「見せかけ」のソースコードを作成する．カムフラージュされたバイナリプログラムは，見せかけのソースコードに対応する命令コード列 (以下，単にコードと呼ぶ) と，それを実行時に命令単位で元来のコードに書き換える自己書換え命令によって構成される．攻撃者が逆アセンブラなどを用いてプログラムの静的解析を試みるとき，プログラム全体に散在する自己書換えルーチンを見つけ出してそのルーチンの動作を理解しない限り，見せかけのコードの内容しか知ることはできない．結果として攻撃者は，広範囲にわたるプログラムの解析を強いられることとなる．

プログラム中の命令を異なる命令で上書き (偽装) しておき，自己書換え機構を用いて実行時に元来の命令を実行させるというアイデアは，従来のカムフラージュ法 [9] に基づいている．提案方法が優れている点は，偽装内容の決定 (上書きされる命令と上書き内容の決定) が，高級言語レベルで行える点である．従来のカムフラージュ法における偽装内容の決定は，アセンブリレベルで無作為あるいは手動で行われる．無作為に行われる場合，攻撃者に知られたくないコードの存在を確実に隠すことがで

* 熊本電波工業高等専門学校 情報工学科，〒 861-1102 熊本県合志市須屋 2659-2, Department of Information and Computer Sciences, Kumamoto National College of Technology, 2659-2 Suya, Koshi, Kumamoto, 861-1102 Japan.
† 奈良先端科学技術大学院大学 情報科学研究科，〒 630-0192 奈良県生駒市 8916-5, Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5 Takayama, Ikoma, Nara, 630-0192 Japan.

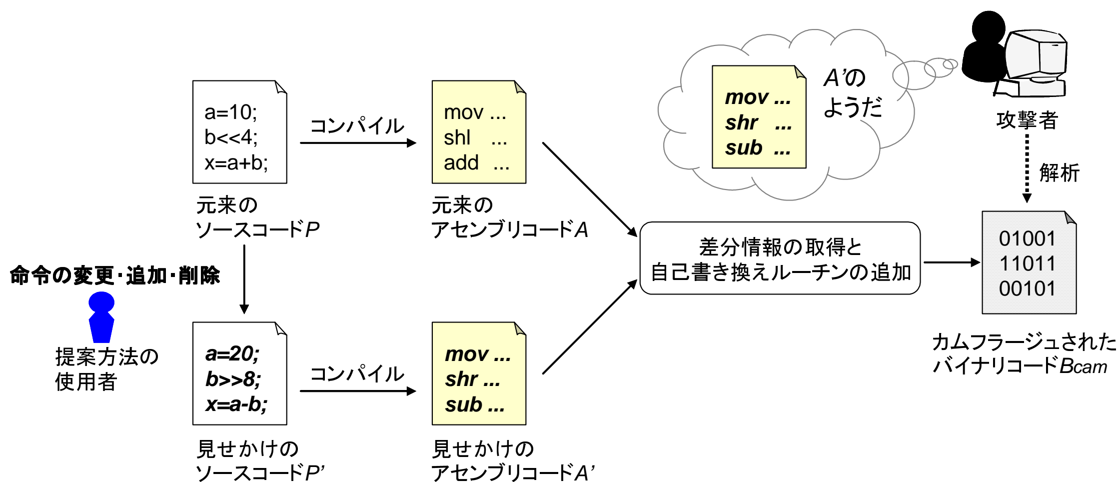

図 1: プログラムのカムフラージュ

きない．また，手動で行われる場合には，ユーザにアセンブリ言語レベルでの複雑な操作が求められることとなり，ユーザへの負担が大きくなる．一方，提案方法では，高級言語のソースコードの段階で，知られたくないアルゴリズムを別の似通ったアルゴリズムに変更したり，知られたくない分岐命令文を削除するといった直観的な作業を行うだけで，偽装内容を決定することができる．このため，攻撃者に知られたくないコードを容易かつ確実に隠すことができる．

以降，2では，提案方法の基本アイデアについて説明する．続く3では，プログラムのカムフラージュの手順について述べた後，カムフラージュされたプログラムの例を示す．最後に4において，結論と今後の課題を述べる．

2　基本アイデア

図1は，プログラムのカムフラージュの流れを示したものである．最初にユーザは，保護対象のソースコード P に含まれる命令を変更，追加あるいは削除して，見せかけのソースコード P' を作成する．次に，P および P' をそれぞれコンパイルし，得られた元来のアセンブリコード A および見せかけのアセンブリコード A' について，1命令単位での**差分情報**を求める．差分情報とは，A' のどの命令を変更，追加，削除すれば，元来の A が得られるか，という情報である．

続いて，得られた差分情報をもとに A' に**自己書き換えルーチン**を追加する．自己書き換えとは，プログラム中の命令の内容を実行時に自ら変化させる動作のことである．自己書き換えルーチンには2つの種類が存在する．ひとつは，カムフラージュする命令を元来の内容に書き換える役割を持つ**復帰ルーチン**で，このルーチンによってプログラムの元来の内容が実行されることを保証する．もうひとつは，復帰ルーチンによって元来の命令となった命令を，再び見せかけの命令に書き換える役割を持つ**隠ぺいルーチン**である．このルーチンは，スナップショットを取得する能力を持つ攻撃者によって，元来の命令を簡単に知られないようにするためのものである．

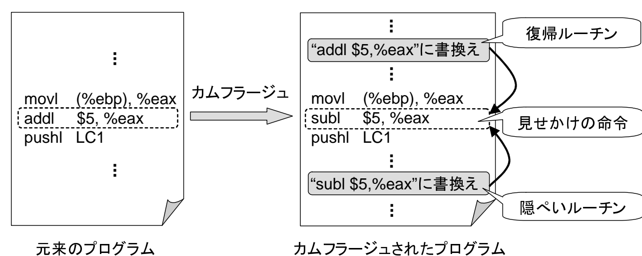

図 2: 見せかけの命令と自己書換えルーチン

図2に，自己書換えルーチンを追加する例を示す．この例においては，元来足し算であった部分を，見せかけのコードでは引き算に変更した命令 (例えば `x=a+5;` を `x=a-5;` に変更した場合) について考える．具体的には，アセンブリレベルで元来 `addl $5,%eax` であった命令を，`subl $5,%eax` に見せかける場合である[1]．この場合，`addl $5,%eax` を，見せかけの命令 `subl $5,%eax` で上書きし，その命令より前に実行される位置に復帰ルーチンを，また後に実行される位置に隠ぺいルーチンを挿入する．復帰ルーチンが実行されたとき，見せかけの命令 `subl $5,%eax` は，元来の命令 `addl $5,%eax` に書き換わる．また，隠ぺいルーチンが実行されたとき，同命令は再び見せかけの命令 `subl $5,%eax` に書き換わる．すなわち，見せかけの命令は，復帰ルーチンが実行されてから，隠ぺいルーチンが実行されるまでの間のみ，元来の命令となる．攻撃者は，見せかけ命令の付近を読んだだけでは，元来の命令が見せかけの命令である `subl $5,%eax` によって上書きされていることに気付くのは難しい．また，隠ぺいルーチンが実行された後の時点におけるプログラムのスナップショットを取得しても，得ら

[1] 本論文では，説明のための例として，Intel x86 系 CPU を想定し，高級言語のコードは C 言語の文法で，アセンブリのコードは AT&T 文法によって示す．

れたスナップショットから元来の命令を知ることはできない．

最後に，自己書換えルーチンが追加されたアセンブリコードをアセンブルし，カムフラージュされたバイナリコード B_{cam} を生成する．攻撃者が静的解析したとき，自己書換えルーチンに気付かない限り B_{cam} には見せかけのアセンブリコード A' の内容しか含まれていないように見えるが，実際には元来のアセンブリコードである A の内容が実行される．

3　カムフラージュの手順とカムフラージュ例

3.1　カムフラージュの手順

プログラムのカムフラージュは，次の (Step 1)〜(Step 3) により行われる．

(Step 1) 見せかけのソースコードの作成

カムフラージュの対象となるソフトウェアの (高級言語で書かれた) ソースコード P に含まれる命令文の 1 つ以上を変更，追加，あるいは削除することによって，見せかけのソースコード P' を作成する．

いま，P の例として，次のような命令文を含むプログラムを考える．

```
        :
if(key1 * 10 + key2 == 45) break;
        :
```

図 3: 元来のソースコード P の例 (一部)

例えば，上の命令文を次のように変更させた命令文を含む P' を作成することができる．

```
        :
if(key1 * 5 <= 25)  break;
        :
```

図 4: 見せかけのソースコード P' の例 (一部)

(Step 2) アセンブリレベルでの差分解析

P および P' をコンパイルし，それぞれのアセンブリプログラム A および A' を得る．

A と A' を命令単位で比較し，差分情報 $D(A', A)$ を得る．差分情報は，アセンブリ 1 命令の変更，追加，削除のいずれかの操作を示す**差分要素**の集合である．差分要素は，次の 3 つのいずれかとなる．

$change(i', i)$: 命令 $i' \in A'$ を，命令 $i \in A$ に変更する操作を示す．

$add(i', i)$: 命令 $i' \in A'$ の直後に，命令 $i \in A$ を追加する操作を示す．

$delete(i')$: 命令 $i' \in A'$ を削除する操作を示す．

$D(A', A)$ に属するすべての差分要素の操作を A' に対して行うと，A と意味的に等価なプログラムが得られる．

いま，A および A' として，(Step 1) の例に挙げた P および P' をコンパイルしたものを考える．図 3 および図 4 に示した部分は，それぞれ図 5 および図 6 のようなアセンブリプログラムとなる．図 5 および図 6 における行頭の数値は，各プログラムにおける命令番号 (何番目の命令であるか) を示す．

```
     :
41:  movl -8(%ebp), %edx
42:  leal (%edx,%eax,2), %eax
43:  cmpl $45, %eax
44:  je   L3
     :
```

図 5: 元来のアセンブリコード A の例 (一部)

```
     :
41:  cmpl $25, %eax
42:  jle  L3
     :
```

図 6: 見せかけのアセンブリコード A' の例 (一部)

P では"key1 * 10 + key2"であった部分が，P' では"key1 * 5"となったことにより，A の 41 番目および 42 番目の命令が A' には含まれていない．同様に，"45"という値が"25"に変更されたことにより，"$45"が"$25"に，比較演算子"=="が"<="に変更されたことにより，"je"が"jle"に変化している．

この例の場合，差分情報 $D(A', A)$ は，次の 4 つの差分要素を持つ．ここで $inst(A', j)$ は，A' における j 番目の命令を示す．

- $change(inst(A', 41),$ "cmpl $45, %eax"$)$
- $change(inst(A', 42),$ "je L3"$)$
- $add(inst(A', 40),$ "movl -8(%ebp), %edx"$)$
- $add(inst(A', 40),$ "leal (%edx,%eax,2), %eax"$)$

(Step 3) 自己書換えルーチンの生成と挿入

$D(A', A)$ に属するすべての差分要素 d_1, d_2, …, d_n について，実行時に差分要素の操作を行う自己書き換えルーチンを生成し，それをプログラムに挿入する．

以下，k 番目の差分要素 d_k についての自己書換えルーチンの生成手順を (Step 3-1) に，また挿入手順を (Step 3-2) に示す．

(Step 3-1) 自己書換えルーチンの生成

まず，書き換え先の命令と，書き換える内容について考える．いま，自己書換えの書き換え先となる命令を $dest_k$，書き換える内容を示す命令を src_k とすると，$dest_k$ および src_k は，d に応じて，次のように決定される．

$d_k = change(i', i)$ **の場合:**

$dest_k$: i' となる．

src_k: i となる．

$d_k = add(i', i)$ **の場合:**

$dest_k$: i' の直後に任意の命令を追加し，それを $dest_k$ とする．

src_k: i となる．

$d_k = delete(i')$ **の場合:**

$dest_k$: i' となる．

src_k: i' の位置で実行されてもプログラムの状態に影響がない (レジスタおよびフラグの値などが変化しない) 命令を作成し，それを src_k とする．

$dest_k$ および src_k が決定したら，次の手順に従って復帰ルーチン RR_k および隠ぺいルーチン HR_k を生成する．RR_k は，実行時に $dest_k$ を src_k に書き換えるルーチン (一連の命令) となり，HR_k は，src_k に書き換えられた命令をを実行時に再び $dest_k$ に書き換えるルーチンとなる．

1. $dest_k$ の直前に，ラベル L_k を挿入する．これにより，L_k を用いて $dest_k$ を間接参照できる．
2. L_k の直後にある命令 ($dest_k$) を src_k の内容に書き換えるための (一連の) 命令を作り，これを RR_k とする．
3. L_k の直後にある命令を，$dest_k$ に書き換えるための (一連の) 命令を作り，これを HR_k とする．

次に，前述した差分要素のひとつ，$change$ ($inst(A', 42)$, "je L3" を用いて，自己書換えルーチン生成の例を示す．この場合，$dest_k$ は A' の 42 番目の命令，すなわち，次の命令となる．

(16 進による機械語表現)　`7E 11`
(アセンブリ表現)　`jle L3`

また，src_k は次の命令となる．

(16 進による機械語表現)　`74 11`
(アセンブリ表現)　`je L3`

まず，$dest_k$ にラベル DEST1 を挿入する．

```
DEST1:  jle L3
```

次に，RR_k を生成する．$dest_k$ を src_k に変更するには，DEST1 に存在する命令の 1 バイト目を「7E」から「74」に書き換えればよい．したがって，RR_k は，例えば次のようなルーチンになる．

```
movb $0x74,DEST1
```

この 1 命令からなる小さなアセンブリのルーチンは，「DEST1 の指すアドレスの内容を，即値 74(16 進) で上書きせよ」という意味を持つ．RR_k が実行されると，DEST1 直後にある命令は，src_k の内容に書き換わる．

同様に，HR_k を生成する．src_k を $dest_k$ に変更するには，DEST1 に存在する命令の 1 バイト目を「7E」に書き換えればよい．したがって，HR_k は，例えば次のようなルーチンになる．

```
movb $0x7E,DEST1
```

RR_k が実行されると，DEST1 直後にある命令は，$dest_k$ の内容に書き換わる．

(Step 3-2) 自己書換えルーチンの挿入

次に，RR_k および HR_k のプログラム上の挿入位置を決定する．以降，RR_k の位置および HR_k の位置をそれぞれ $P(RR_k)$ および $P(HR_k)$ と記す．

アセンブリプログラムの一命令を一つのノードとみなした制御フローグラフにおいて，次の 4 条件を満たすように $P(RR_k)$ および $P(HR_k)$ を決定する．これらの条件は，「$dest_k$ が実行される前に必ず src_k に書き換えられ，プログラム終了前に，必ず元通り $dest_k$ に書き換えられる」ことを保障するためのものである．

条件 1　プログラム開始点 $start$ から $dest_k$ に至る全てのパスに $P(RR_k)$ が存在する．

条件 2　$P(RR_k)$ から $dest_k$ に至る全てのパスに $P(HR_k)$ が存在しない．

条件 3　$P(HR_k)$ から $dest_k$ に至る全てのパスに $P(RR_k)$ が存在する．

条件 4　$dest_k$ からプログラム終了点 end に至る全てのパスに $P(HR_k)$ が存在する．

$P(RR_k)$ および $P(HR_k)$ が決定したら，(Step 3) において生成した RR_k および HR_k をそれぞれ挿入する．

なお，この挿入位置の決定方法は，文献 [9] における，カムフラージュ化プログラムの作成手順の (Step 1) と同じもので，同文献には詳細なアルゴリズムが記されている (本論文では紙面の都合上省略する)．

3.2　カムフラージュ例

3.1 で述べた手順にしたがってカムフラージュされたプログラムの例を示す．いま，図 7(a) に示すような C 言語のプログラムから，図 7(b) に示すような見せかけのプログラムを作成したとする．見せかけのプログラムでは，"key1 * 10 + key2 == 45" という式を含む命令文が削除され，"key1 + key2 <= 70"および"key1 * 5 <= 25"という式を含む命令文が追加されている．また，while 文が削除されることで，制御構造も変更されている．ここでは，差分要素は 9 つ ($change$4 つ，add1 つ，$delete$4 つ) となり，カムフラージュされたプログラ

```c
#include <stdio.h>

int main() {
  int key1, key2;

  while(1) {
    printf("Input key1: ");
    scanf("%d", &key1);
    printf("Input key2: ");
    scanf("%d", &key2);

    if(key1 * 10 + key2 == 45)
      break;

    printf("Invalid Password.\n");
  }
  printf("Password OK.\n");
  return 0;
}
```

(a) 元来のソースコード

```c
#include <stdio.h>

int main() {
  int key1, key2;

  printf("Input key1: ");
  scanf("%d", &key1);
  printf("Input key2: ");
  scanf("%d", &key2);

  if(key1 + key2 <= 70)
    printf("Password OK.\n");

  if(key1 * 5 <= 25) {
    printf("Invalid Password.\n");
  }

  return 0;
}
```

(b) 見せかけのソースコード

定数の宣言

```
LC0:
.ascii "Input key1: \0"
LC1:
.ascii "%d\0"
LC2:
.ascii "Input key2: \0"
LC3:
.ascii "Invalid Password.\0"
LC4:
.ascii "Password OK.\0"
```

前処理

```
_main:
  movb  $0xF8, DEST2+2    #RR2
  pushl %ebp
  movl  %esp, %ebp
  pushl %edx
  pushl %edx
  call  ___main
  subb  $0x07, DEST5+1    #RR5
L2:
  movb  $0x6B, DEST1      #RR1
  movw  $0x0AFC, DEST1+2  #RR1
```

key1 および key2 を標準入力から取得する部分

```
  pushl $LC0
  call  _printf
  movw  $0x9090, DEST7    #RR7
  movb  $0x90, DEST7+2    #RR7
  leal  -4(%ebp), %eax
  pushl  %eax
  movw  $0x9090, DEST8    #RR8
  pushl $LC1
  call  _scanf
  pushl $LC2
  call  _printf
  leal  -8(%ebp), %eax
  pushl %eax
  movb  $0x74, DEST4      #RR4
  pushl $LC1
  call  _scanf
  movb  $0x2D, DEST3+2    #RR3
```

if(key1+key2 <= 70).. に見せかける部分

```
DEST1:  #1 change to "imull $10,-4(%ebp),%eax"
  movl  -8(%ebp), %eax
  ret   # added (padding)
  addl  $24, %esp
  movb  $0x8B, DEST1      #HR1
  movw  $0xF845, DEST1+2  #HR1
  movb  $0x90, DEST9      #RR9
  movb  $0x45, DEST3+2    #HR3
DEST2:  #2 change to "addl -8(%ebp),%eax"
  addl  -4(%ebp), %eax
DEST3:  #3 change to "cmpl $45,%eax"
  cmpl  $70, %eax
DEST4:  #4 change to "je _L2"
  jg    _L2
  pushl $LC3
  call  _puts
  popl  %edx
  movl  $0x90909090, DEST6  #RR6
DEST5:  #5 added and change to "jmp L2"
  jmp  L2+7
```

if(key1*5 <= 25).. に見せかける部分

```
_L2:
DEST6:  # delete
  imull $5, -4(%ebp), %eax
DEST7:  # delete
  cmpl  $25, %eax
DEST8:  # delete
  jg    _L3
  pushl  $LC4
  movw  $0x7F27, DEST8    #HR8
  call  _puts
DEST9:  # delete
  popl  %eax
_L3:
  movw  $0x83F8, DEST7      #HR7
  movb  $0x19, DEST7+2      #HR7
  movl  $0x6B45Fc05, DEST6  #HR6
  addb  $0x07, DEST5+1      #HR5
  movb  $0x7F, DEST4        #HR4
```

終了処理

```
  leave
  xorl  %eax, %eax
  movb  $0xFC, DEST2+2      #HR2
  movb  $0x58, DEST9        #HR9
  ret
```

(c) カムフラージュされたアセンブリコード

図 7: プログラムのカムフラージュ例

ムとして，例えば図 7(c) に示すようなプログラムが得られる．ここでは見やすさを考え，プログラムを 6 つの部分に分けて示している．

図 7(c) において，DEST1:，DEST2:，...，DEST9:は，実行時に書き換えられる命令を指すラベルであり，行内にコメントとして差分要素の操作と書き換えられる内容を記している．また，RR および HR とコメントしてある命令は，それぞれ復帰ルーチンおよび隠ぺいルーチンを示す．

このプログラムは，見せかけのプログラムに含まれる命令と自己書換えルーチンによってのみ構成されており，一部を読むだけでは，正しい内容を理解できない．例えば，ラベル DEST2:の位置にある addl，cmpl，jg という命令が続く部分だけを読むと，このプログラムでは，"key1 + key2 <= 70"という処理が行われるように見える．しかし，その部分は実行時に復帰ルーチンによって書き換えられ，実際には元来のプログラムに含まれる "key1 * 10 + key2 == 45"の動作 (の一部) が実行されることになる．

この例においては単純な自己書換えルーチンを用いているが，機械語命令の難読化 [5] や mutation [3] の従来技術を併用するなどして静的なパターンマッチングによる自己書き換えルーチンの特定を難しくすると，解析をより困難にできる．また，このプログラム例は極めて規模が小さいため，復帰ルーチンがプログラムの前半に，また隠ぺいルーチンがプログラムの後半に集中して存在している．プログラムの規模が大きくなると，自己書換えルーチンが広い範囲に散在することになり，攻撃者の解析により多くの労力が求められることになる．

4 おわりに

本論文では，プログラムをカムフラージュすることで，ソフトウェアの解析を困難にするための系統的な方法を提案した．カムフラージュされた部分について攻撃者が静的解析を試みるとき，復帰ルーチンの存在を探し当ててその内容を理解しない限り，その部分の元来の動作を正しく理解することはできない．結果として攻撃者は，広範囲にわたるプログラムの解析を強いられることとなる．

ユーザは，高級言語のソースコードの段階で，知られたくないアルゴリズムを別の似通ったアルゴリズムに変更したり，知られたくない分岐命令文を削除するといった直観的な作業を行うだけで，偽装内容を決定することができる．このため，ユーザの意図する偽装内容に基づいたカムフラージュを容易に行うことができる．

最後に，今後の課題について述べる．まず，提案方法を適用することによって，プログラムにどの程度の実行時間のオーバーヘッドが生じるかを，実験を通して調べる予定である．さらには，カムフラージュされたプログラムの解析の困難さを評価する方法について検討していくことを考えている．

参考文献

[1] S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot. A white-box DES implementation for DRM applications. In *Proc. 2nd ACM Workshop on Digital Rights Management*, pp. 1–15, Nov. 2002.

[2] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. *IEEE Transactions on Software Engineering*, Vol. 28, No. 8, pp. 735–746, June 2002.

[3] J. Irwin, D. Page, and N.P. Smart. Instruction stream mutation for non-deterministic processors. In *Proc. ASAP2002*, pp. 286–295, July 2002.

[4] Yuichiro Kanzaki. *Protecting Secret Information in Software Processes and Products*. PhD thesis, Nara Institute of Science and Technology, Mar. 2006.

[5] M. Mambo, T. Murayama, and E. Okamoto. A tentative approach to constructing tamper-resistant software. In *Proc. 1997 New Security Paradigm Workshop*, pp. 23–33, Sep. 1997.

[6] 門田暁人, Clark Thomborson. ソフトウェアプロテクションの技術動向 (前編) - ソフトウェア単体での耐タンパー化技術. 情報処理, Vol. 46, No. 4, pp. 431–437, April 2005.

[7] 山田尚志, 河原潤一. デジタルコンテンツ保護の現状と課題. 東芝レビュー, Vol. 58, No. 6, pp. 2–7, June 2003.

[8] 舩本昇竜. プロテクト技術解剖学. すばる舎, 東京, 2002.

[9] 神崎雄一郎, 門田暁人, 中村匡秀, 松本健一. 命令のカムフラージュによるソフトウェア保護方法. 電子情報通信学会論文誌, Vol. J87-A, No. 6, pp. 755–767, June 2004.