ソースコードレベルにおけるプログラムのカムフラージュ

神﨑 雄一郎　門田 暁人　中村 匡秀　松本 健一

本論文では，ソースコードレベルで偽装内容を指定できるプログラムのカムフラージュ法を提案する．提案方法の使用者は，保護対象のプログラムの秘密情報が変更・削除されている「見せかけのソースコード」を作成するだけで，攻撃者が秘密情報を得るためのコストを増大させることができる．保護されたプログラムを静的解析したときに攻撃者が得られるコードには，見せかけのソースコードの内容と自己書換えを行うコードしか含まれていないが，実際には元来のソースコードの内容が実行される．プログラムに含まれる特定の命令やデータを静的解析から容易かつ確実に隠ぺいしたい場合や，プログラムの一部が抽出・再利用されることを防ぎたい場合において特に有効である．

This paper proposes a program camouflage method to protect software from reverse engineering. The user of the proposed method only has to construct a piece of fake source code by modifying a piece of original source code. When an attacker statically analyzes the program that is protected by the method, the program looks like the fake code (with self-modification code fragments). However, when the program is executed, the original code is performed. The proposed method is effective especially in hiding secret instructions/data from static analysis, and preventing the extraction and reuse of secret parts in the program.

1　はじめに

ソフトウェア保護技術，すなわち，ソフトウェアに対する不正な解析行為を防ぐ技術の研究は従来さかんに行われている．ソフトウェア保護の目的は，プログラムに含まれる情報を攻撃者が取得，理解，あるいは改ざんするのに必要なコスト (時間や労力) を可能な限り増大させることである [5]．例えば，ライセンスチェックのプログラムを保護する場合，ライセンスをチェックする比較・分岐命令や，その命令を見つける手がかりとなる命令 (ライセンスコードの入力を促す命令など) を攻撃者が発見・改ざんするためのコストをプログラムの難読化，暗号化，耐タンパ化などの方法を用いて増大させる．

効果の高い保護を行うには，保護方法の使用者 (以下，ユーザと呼ぶ) は保護方法のアルゴリズムを理解した上で，プログラム中のどの部分の解析を困難にすべきかを十分考慮して適用する必要がある．ただし，保護方法のアルゴリズムを理解するには低級言語や計算機構造に関する詳しい知識が求められることが多く，有効な保護処理を行うためのユーザの負担は小さくない．

そこで本研究では，保護適用の容易さに焦点を合わせ，ソースコードレベル[†1]で偽装内容を指定できるプログラムのカムフラージュ法を提案する．提案方法を用いれば，秘密情報 (攻撃者に知られたくない命令や定数) が変更・削除されている「見せかけの」(偽

Program Camouflage at the Source Code Level.

Yuichiro Kanzaki, 熊本高等専門学校人間情報システム工学科, Dept. of Human-Oriented Information Systems Engineering, Kumamoto National College of Technology.

Akito Monden, Ken-ichi Matsumoto, 奈良先端科学技術大学院大学情報科学研究科, Graduate School of Information Science, Nara Institute of Science and Technology.

Masahide Nakamura, 神戸大学大学院システム情報学研究科, Graduate School of System Informatics, Kobe University.

コンピュータソフトウェア, Vol.28, No.1 (2011), pp.300–305.
[研究論文 (レター)] 2010 年 6 月 15 日受付.

†1　本論文において単にソースコードと記した場合，C 言語等の高級言語で記述されたソースコードを指す．

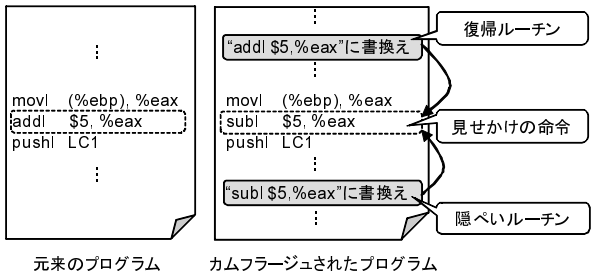

図 1 命令のカムフラージュ法

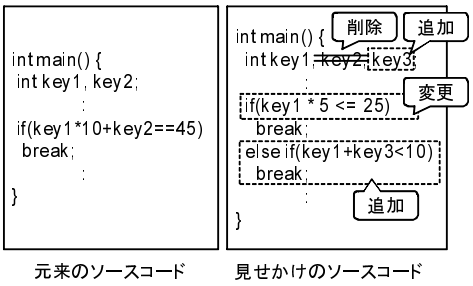

図 2 見せかけのソースコードの作成例

の) ソースコードを作成するだけで，攻撃者が秘密情報を得るためのコストを増大させることができる．例えば，ある暗号アルゴリズムの存在を攻撃者から隠ぺいするために，それを偽のアルゴリズムに置き換えたソースコードを作成した場合を考える．保護されたコードを攻撃者が静的解析したときに得られるコードには，元来のアルゴリズムの代わりに偽のアルゴリズムが存在するため，攻撃者に偽のアルゴリズムを用いているかのように見せかけることができる．攻撃者が元来のアルゴリズムを取得するには，コード内に分散する自己書換えを行うルーチンを探し出して書き換え内容を解析する必要があり，大きなコストがかかる．

提案方法は，アルゴリズム，API 呼出し，DRM システムの復号鍵などの特定の命令やデータを静的解析から隠ぺいする目的や，プログラムの一部が抽出・再利用されることを防ぐ目的に対して特に有効である．また，動的解析を防ぐ方法の保護の効果を高めるのにも役立つ．例えば，アンチデバッグ機構 (例えば文献 [3] で紹介されているもの) が施されている場合，攻撃者は動的解析を行うためにまず静的解析によってアンチデバッグ機構のコードを発見・除去する必要がある．そのコードをカムフラージュすることで，コードの発見・除去に必要なコストを増大させることができ，結果として動的解析・静的解析両方に対して耐性の高い保護を行うことができる．

2 基本アイデア

まず，提案方法の基礎となる命令のカムフラージュ法 [8] について説明する．命令のカムフラージュ(偽装) とは，プログラム中の特定の命令が攻撃者の静的解析によって発見されるのを困難にする方法である．プログラム中のある命令 I_o を見せかけの (偽の) 命令 I_f で上書きし，実行時のある期間のみ元来の命令 I_o に自己書換え機構を用いて復元することで，I_o の存在を攻撃者から隠す．図 1 は，命令のカムフラージュ法によってプログラム中の 1 つの命令をカムフラージュした例である†2．この例においては，`addl $5,%eax` という命令が `subl $5,%eax` という見せかけの命令で上書きされており，その前後に存在する復帰ルーチンおよび隠ぺいルーチンが，自己書換え機構を用いてそれぞれ命令の復帰および再隠ぺいを行う．このようなアセンブリの 1 命令単位の保護処理を繰り返し適用し，プログラムの解析を困難にする．

命令のカムフラージュの適用対象命令の決定は，無作為あるいはユーザの指定によって行われる．無作為に行われる場合は，すべての秘密情報を保護できる確実性がない．また，多数の効果の低いカムフラージュによって実行効率が大幅に低下することもある．一方，ユーザの指定で行われる場合には，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い．

提案方法では，ソースコードレベルでカムフラージュの内容を指定するため，秘密情報を容易かつ確実に隠ぺいできる．ユーザは，保護対象となるソースコードの一部を変更し，**見せかけのソースコード**を作成する．図 2 は，見せかけのソースコードを作成する例を示したものである．この例では，条件判定式 (if 文) の比較演算子などが変更される，新たな条件判定式や新しい変数 (`key3`) が追加される，1 つの変数 (`key2`) が削除される等の処理が行われている．提案

†2 本論文では，説明のための例として Intel x86 系 CPU を想定し，アセンブリ表現は AT&T 文法で示す．

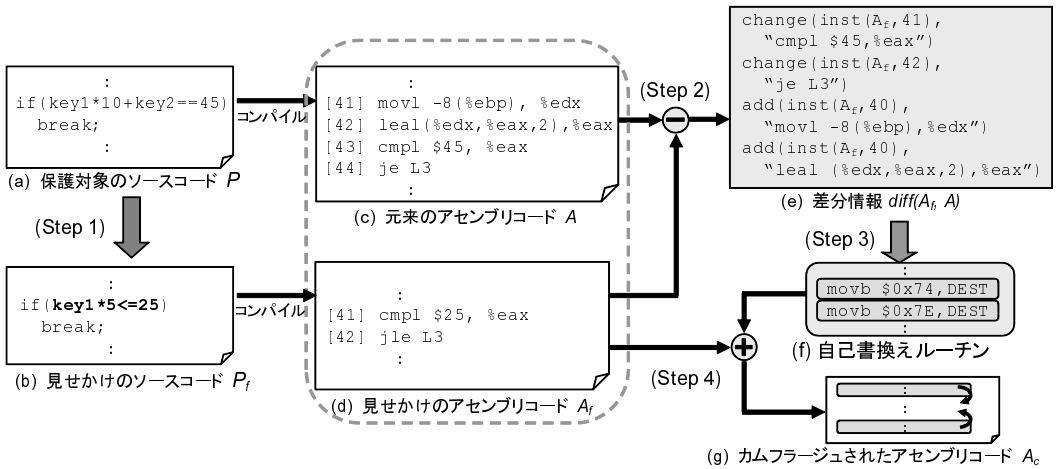

図 3　提案方法によるカムフラージュの流れ

方法は，見せかけのソースコードの制御の依存関係および元来のソースコードとの差分を分析し，実行時には元来のソースコードの内容が実行されるように自己書換え命令を追加する．保護されたプログラムを静的解析したときに攻撃者が得られるコードは，見せかけのソースコードの内容と自己書換えコードしか含まれていないが，実際には元来のソースコードの内容が実行される．このように，ソースコードレベルで秘密情報を別の内容に変更したり，単に削除するといった直観的な作業を行うだけで攻撃者が秘密情報を取得するコストを増大させることができる．

3　カムフラージュの適用方法

提案方法によるカムフラージュの流れを図 3 に示す．まず，ユーザが保護対象となるソースコード P の内容を変更し，見せかけのソースコード P_f を作成する (Step 1)．次に，P および P_f をそれぞれコンパイルして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_f について，**差分情報**を求める (Step 2)．$diff(A_f, A)$ で表される差分情報は，「A_f 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，アセンブリ 1 命令単位での変更，追加あるいは削除の処理を示す**差分要素**の集合として表される．続いて，得られた差分情報をもとに**自己書き換えルーチン**，すなわち，元来の内容に書き換える**復帰ルーチン**と再び偽の内容に書き換える**隠ぺいルーチン**を生成する (Step 3)．最後に自己書換えルーチンを A_f に追加することで，カムフラージュされたアセンブリコード A_c を得る (Step 4)．

以下に，各ステップの詳細を述べる．

(Step 1) 見せかけのソースコードの作成

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，P_f を作成する．P および P_f は，同一のコンパイラでコンパイル可能である必要がある．図 3(a) および (b) に示した例においては，P に含まれる "`if(key1*10+key2==45) break;`" という命令文が，P_f では "`if(key1*5<=25) break;`" という命令文に変更されている．命令文を変更・追加・削除する処理の数には制約はなく，処理を行った箇所が多くなればなるほど，差分要素の数が多くなる．

(Step 2) アセンブリレベルでの差分解析

P および P_f を各々コンパイルして得られるアセンブリプログラム A および A_f を比較し，差分情報 $diff(A_f, A)$ を得る．具体的には，Myers による 2 つの文書間の差分を求めるアルゴリズム [7] に基づき，アセンブリ 1 命令を単位とした差分要素の集合を求める．差分要素は次の 3 つのいずれかとなる．

- $change(i_f, i)$：A_f 内の命令 i_f を，A に含まれる命令 i に変更する処理を示す．
- $add(i_f, i)$：A_f 内の命令 i_f の直後に，A に含まれる命令 i を追加する処理を示す．
- $delete(i_f)$：A_f 内の命令 i_f を削除する処理を示す．

$diff(A_f, A)$ に属するすべての差分要素の処理を A_f 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．

図 3(a) および (b) に示した P および P_f の例に対応するアセンブリコードを，図 3(c) および (d) にそれぞれ示す．ここで各行頭の数値は，各プログラムにおける命令番号 (何番目の命令であるか) を表している．P では “key1*10+key2” であった部分が，P_f 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_f 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 $diff(A_f, A)$ は図 3(e) に示すような 4 つの差分要素 ($change$2 つ，add2 つ) を持つ．なお，図 3(e) 中における $inst(A_f, j)$ は，A_f における j 番目の命令を示す．

(Step 3) 自己書換えルーチンの生成

$diff(A_f, A)$ に属するすべての差分要素 $d_1, d_2, \ldots, d_n$ について，実行時に差分要素に従った処理を行う自己書き換えルーチンを生成する．以下，k 番目の差分要素 d_k についての復帰ルーチン RR_k および隠ぺいルーチン HR_k の生成手順を示す．ここで，RR_k および HR_k の書換え先のアドレス (プログラム上の位置) を $dest$ とする．また，src_R(または src_H) は，RR_k(または HR_k) によって $dest$ に上書きされる命令と定義する．RR_k および HR_k は，次の手順で生成される．

[手順 1] まず，src_R，src_H および $dest$ を，d_k の処理の種類に応じて次のように決定する．

[手順 1-(a)] d_k が $change(i_f, i)$ の場合，src_R は i と同一の内容，src_H は i_f と同一の内容となり，$dest$ は i_f のアドレスとなる．

[手順 1-(b)] d_k が $add(i_f, i)$ の場合，まず A_f 内の i_f の直後に i と同一のバイト長を持つ命令を挿入し，i_a とする．src_R は i と同一の内容，src_H は i_a と同一の内容となり，$dest$ は i_a のアドレスとなる．

[手順 1-(c)] d_k が $delete(i_f)$ の場合，まず i_f のアドレスで実行しても以降のプログラムの実行に影響を与えない命令 (汎用レジスタやフラグの値が変化しない命令など) を作成し，i_n とする．src_R は i_n と同一の内容，src_H は i_f と同一の内容となり，$dest$ は i_f のアドレスとなる．

[手順 2] src_R と src_H を機械語のレベルで比較し，$dest$ に存在すると仮定した src_H を src_R の内容に書き換えるための (一連の) 命令を作る．これを RR_k とする．

[手順 3] 同様に，$dest$ に存在すると仮定した src_R を src_H の内容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．

上に述べた手順に従い，図 3(e) に示した差分要素のひとつ，$change$ ($inst(A_f, 42)$, “je L3”) について，以下に自己書換えルーチンの生成例を示す．

[手順 1] $d_k = change(inst(A_f, 42),$“je L3”$)$ より，src_R は “je L3”，src_H は A_f の 42 番目の命令，すなわち，“jle L3” となり，$dest$ は，A_f の 42 番目の命令を指すアドレスとなる．

[手順 2] src_R(“je L3”) および src_H(“jle L3”) の機械語表現が，それぞれ “74 11” および “7E 11” で表されるとすると，$dest$ に存在する src_H を src_R に変更するには，$dest$ 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．従って，RR_k は，例えば次のようなルーチンになる．

```
movb $0x74,DEST
```

ここで DEST は，$dest$ のアドレスを指すラベルを示す．このルーチンは,「DEST が指す内容を，即値 74(16 進) で上書きせよ」という意味を持つ．

[手順 3] [手順 2] と同様の方法で，HR_k を生成する．$dest$ に存在する src_R を src_H に変更するには，$dest$ に存在する命令の 1 バイト目を “74” から “7E” に書き換えればよい．従って，HR_k は，例えば次のようなルーチンになる．

```
movb $0x7E,DEST
```

(Step 4) 自己書換えルーチンの挿入

最後に，生成した各自己書換えルーチンをプログラムの制御の流れを考慮して A_f に挿入し，A_c を得る．命令のカムフラージュ法における自己書換えルーチンの位置決定のアルゴリズム [8] に従うことで，アセンブリ言語のレベルで自動的に挿入位置を決定することができる．紙面の都合上，詳細は省略する．

```
int main() {
  int key1, key2;

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

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

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

(a) 保護対象のソースコード

```
int main() {
  int key1, key2;

  scanf("%d", &key1);
  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
LC1:.ascii "%d¥0"
LC3:.ascii "Invalid Password.¥0"
LC4:.ascii "Password OK.¥0"
前処理
_main:
      movb  $0xF8, DEST2+2      … RR2
[01]  pushl %ebp
[02]  movl  %esp, %ebp
[03]  pushl %edx
[04]  pushl %edx
[05]  call  ___main
      subb  $0x07, DEST5+1      … RR5
L2:
      movb  $0x6B, DEST1        } RR1
      movw  $0x0AFC, DEST1+2    }
key1およびkey2を標準入力から取得
      movw  $0x9090, DEST7      } RR7
      movb  $0x90, DEST7+2      }
[06]  leal  -4(%ebp), %eax
[07]  pushl  %eax
[08]  pushl  %eax
      movw  $0x9090, DEST8      … RR8
[09]  pushl $LC1
[10]  call  _scanf
[11]  leal  -8(%ebp), %eax
[12]  pushl %eax
[13]  pushl %eax
      movb  $0x74, DEST4        … RR4
[14]  pushl $LC1
[15]  call  _scanf
      movb  $0x2D, DEST3+2      … RR3
if(key1+key2 <= 70)..に見せかける部分
DEST1:  # change to
        # "imull $10,-4(%ebp),%eax"
[16]  movl  -8(%ebp), %eax
      ret   # added (padding)
[17]  addl  $24, %esp
      movb  $0x8B, DEST1        } HR1
      movw  $0xF845, DEST1+2    }
      movb  $0x90, DEST9        … RR9
DEST2:  # change to
        # "addl -8(%ebp),%eax"
[18]  addl  -4(%ebp), %eax
DEST3:  # change to "cmpl $45,%eax"
[19]  cmpl  $70, %eax
DEST4:  # change to "je _L2"
[20]  jg    _L2
[21]  pushl $LC3
[22]  call  _puts
[23]  popl  %edx
      movl  $0x90909090, DEST6  … RR6
DEST5:  # change to "jmp L2"
[24]  jmp  L2+7
if(key1*5 <= 25)..に見せかける部分
_L2:
      movb  $0x45, DEST3+2      … HR3
DEST6:  # delete
[25]  imull $5, -4(%ebp), %eax
DEST7:  # delete
[26]  cmpl  $25, %eax
DEST8:  # delete
[27]  jg    _L3
[28]  pushl  $LC4
      movw  $0x7F27, DEST8      … HR8
[29]  call  _puts
DEST9:  # delete
[30]  popl  %eax
_L3:
      movw  $0x83F8, DEST7      } HR7
      movb  $0x19, DEST7+2      }
      movl  $0x6B45Fc05, DEST6  … HR6
      addb  $0x07, DEST5+1      … HR5
      movb  $0x7F, DEST4        … HR4
後処理
[31]  leave
[32]  xorl  %eax, %eax
      movb  $0xFC, DEST2+2      … HR2
      movb  $0x58, DEST9        … HR9
[33]  ret
```

(c) カムフラージュされたアセンブリコード

図 4　プログラムのカムフラージュ例

4　ケーススタディ

3 章で述べた手順に従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 4(a) に示すような C 言語のプログラムから，図 4(b) に示すような見せかけのソースコードを作成したとする．見せかけのソースコードでは，“`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` 文が削除されるなど，制御構造も変更されている．図 4(c) は，提案方法によってカムフラージュされたアセンブリコードの一例である．行頭に番号が記されている命令は，見せかけのアセンブリコード (図 4(b) のアセンブリコード) に含まれる命令で，番号は同コードにおける命令番号を表している．実行時に書き換えられる命令は，“`DEST`” で

始まる名前のラベルが付けられている．また，それらを書き換える復帰ルーチンおよび隠ぺいルーチンは，RR_1，...，RR_9 および HR_1，...，HR_9 と示す．

カムフラージュされたアセンブリコードは，見せかけのアセンブリコード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，見せかけの内容である “`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．なお，この例では単純な自己書換えルーチンを用いているが，コード領域の書換えがないように見せかける [8]，機械語レベルでの難読化を行うなどの変形を適用することで，静的解析による自己書換えルーチンの特定を困難にでき，解析に要するコストを増大させることができる．

また，CPPM/CPRM で用いられている C2 暗号によって暗号化された 8 バイトのデータを復号するプログラムを対象に，実行時間 (クロック数) のオーバーヘッドを測定した．測定結果から，見せかけの命令 (カムフラージュされた命令) が多くなるに従って，実行時間のオーバーヘッドが増加することがわかった．例えば，元来のコードの実行クロック数が 2.91×10^9 クロックであった場合に，復号処理のサブルーチン全命令の 20%をカムフラージュすると 2.94×10^9 クロックに，50%をカムフラージュすると 2.99×10^9 クロックとなった．

5 関連研究

ソフトウェアに含まれる秘密情報を不正な解析から守るソフトウェア保護技術として，これまで数多くのプログラム暗号化法，難読化法をはじめとして様々な方法が提案されている [4] [5]．提案方法と関連が深い自己書換え機構を用いた方法としては，XOR 演算を用いて実行時にコードを復帰・隠ぺいする Aucsmith らの方法 [1]，書換え内容が記述されたスクリプトに従って実行時にコードを変形する Madou らの方法 [6]，一部のコードに依存して動的にコードの暗号化・復号を行う Cappaert らの方法 [2] がある．提案方法は，攻撃者が静的解析によって取得できるコードの内容をソースコードレベルで自由に記述できる点が新しい．他の保護方法と組み合わせることで，プログラムの解析をより困難にすることも可能である．

6 まとめ

本論文では，ソースコードレベルにおいて偽装内容を指定できるプログラムカムフラージュ法を提案した．ユーザは，高級言語のソースコードの段階で，価値のあるアルゴリズムや分岐命令文などの秘密情報を別の内容に変更したり，単に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できる．そのため，攻撃者に知られたくないコードを容易かつ確実に保護することができる．

謝辞 本研究の一部は，科学研究費補助金 若手研究 (B)，課題番号 20700034 の助成を受けたものである．

参考文献

[1] Aucsmith, D. W.: *Tamper Resistant Software:An Implementation*, Lecture Notes in Computer Science, Vol. 1174, Springer-Verlag, 1996, pp. 317–333.
[2] Cappaert, J., Kisserli, N., Schellekens, D. and Preneel, B.: Self-encrypting Code to Protect Against Analysis and Tampering, in *Benelux Workshop on Information and System Security*, 2006.
[3] Cervan, P.: *Crackproof Your Software*, No Starch Press, 2002.
[4] Collberg, C. and Thomborson, C.: Watermarking, tamper-proofing, and obfuscation – Tools for software protection, *IEEE Trans. Softw. Eng.*, Vol. 28, No. 8(2002), pp. 735–746.
[5] Collberg, C. and Nagra, J.: *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Program Protection*, Addison-Wesley Professional, 2009.
[6] Madou, M., Anckaert, B., Moseley, P., Debray, S., Sutter, B. D. and Bosschere, K. D.: *Software Protection through Dynamic Code Mutation*, Lecture Notes in Computer Science, Vol. 3786, Springer-Verlag, 2006, pp. 194–206.
[7] Myers, E. W.: An O(ND) difference algorithm and its variations, *Algorithmica*, Vol. 1, No. 2(1986), pp. 251–266.
[8] 神﨑雄一郎, 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(2004), pp. 755–767.