INFORMATION
SCIENCE
TECHNICAL
REPORT

NAIST-IS-TR2007015
ISSN 0919-9527

高級言語によって
偽装内容を指定できる
拡張プログラムカムフラージュ法

神崎雄一郎, 門田暁人, 中村匡秀, 松本健一

December 2007

NAIST
〒 630-0192
奈良県生駒市高山町 8916-5
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
8916-5 Takayama, Ikoma, Nara 630-0192, Japan

高級言語によって偽装内容を指定できる拡張プログラムカムフラージュ法

神崎 雄一郎[†]　門田 暁人[††]　中村 匡秀[†††]　松本 健一[††]

[†]熊本電波工業高等専門学校
情報工学科
kanzaki@cs.knct.ac.jp

[††]奈良先端科学技術大学院大学
情報科学研究科
{akito-m,matumoto}@is.naist.jp

[†††]神戸大学大学院
工学研究科
masa-n@cs.kobe-u.ac.jp

概要

ソフトウェア保護方式の 1 つであるプログラムカムフラージュ法の拡張方式を提案する．提案方式のユーザは，攻撃者に知られたくない命令の内容変更や削除を高級言語のレベルで行い，見せかけのソースコード，すなわち，偽(にせ)の内容が記述されたソースコードを作成する．提案方式によって得られる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バイナリコード列と，それを実行時の一定の期間のみ元来のバイナリコード列に書き換える自己書換えルーチンによって構成される．攻撃者が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動作を理解しない限り，見せかけのソースコードに対応する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，多くのコストを費やすこととなる．

1　はじめに

近年，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おり，ソフトウェアに含まれる秘密情報 (例えば，ライセンスをチェックする命令文や商業的価値の高いアルゴリズムなど) は，悪意を持ったユーザ (攻撃者) の解析行為によって第三者に漏えいする危険にさらされている．ソフトウェア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，アンチデバッグ法が提案されてきた [2,8]．本論文では，難読化の一手法であるプログラムカムフラージュ法 [9] について，拡張方式を提案する．

プログラムのカムフラージュ (偽装) とは，プログラム中のある命令 I_o について，見せかけの (ダミーの) 命令 I_f で上書きし，プログラムの自己書き換えにより実行時のある期間のみ元来の命令 I_o に復元することで，命令 I_o の存在を静的解析から隠す方式である [9]．図 1 は，従来方式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 (アセンブリ言語で示し

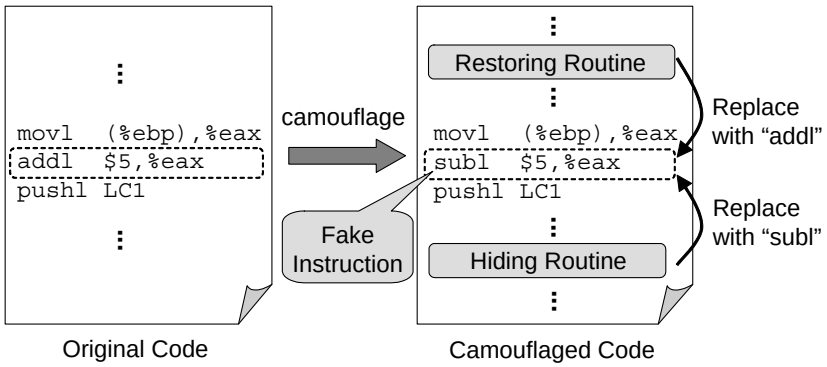


図 1　従来のカムフラージュ法

ている)[*1]．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 (fake instruction) で上書きされており，その命令より前に実行される位置に**復帰ルーチン** (restoring routine) が，また，後に実行される位置に**隠ぺいルーチン** (hiding routine) が挿入されている．復帰ルーチンは，自己書換え，すなわち，コード領域の内容を実行時に書き換える機構によって，見せかけの命令 `subl $5,%eax` を元来の命令 `addl $5,%eax` に書き換える．一方，隠ぺいルーチンは，同命令を再び見せかけの命令の内容 `subl $5,%eax` に書き換える．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．復帰ルーチンが見せかけの命令から離れた位置 (アドレス) に存在するとき，見せかけの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攻撃者が気付くのは難しい．

プログラムカムフラージュ法は，プログラムに含まれる特定の命令群を静的解析から隠ぺいする目的や，プログラムの一部 (サブルーチン) 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．一方，命令の書換えが実行時に行われるため，動的解析に対して弱点を持つ方式であるといえる．ただし，動的解析を防ぐためのアンチデバッグコードと併用し，さらに，アンチデバッグコード自身にカムフラージュを施すことで，保護の強度を高めることができる．動的解析を行おうとする攻撃者は，まず，プログラム中のアンチデバッグコードを発見・除去するために静的解析を行う必要がある．そこで，アンチデバッグコードにカムフラージュを施しておくことで，保護のレベルを高めることができる．

従来のプログラムカムフラージュ法における実用上の大きな課題は，偽装内容の決定 (I_o の選択と I_f の決定) をアセンブリ言語のレベルで行わねばならないことである．そのため，特定の命令や定数を隠ぺいしたい場合には，アセンブリ言語に熟達した人間でないと対応が困難であった．そこで本論文では，従来方式を拡張し，高級言語レベルで偽装内容を指定できる方式を提案する．提案方式のユーザは，高級言語のソースコードの段階で，秘密情報であるアルゴリズムや分岐命令文を別のものに変更したり，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

[*1] 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 文法によって示す．

る．そのため，攻撃者に知られたくない命令コード列を容易かつ確実に隠すことができる．

以降，2 章では，高級言語によって偽装内容を指定できるプログラムカムフラージュ法の詳細手順を述べる．3 章では，カムフラージュされたプログラムの具体例を挙げて考察する．4 章では，カムフラージュされたプログラムの実行時間に関するオーバーヘッドを測定した実験結果について報告する．最後に 5 章において，結論を述べる．

2 提案方式

2.1 カムフラージュの流れ

提案方式によるカムフラージュの流れを図 2 に示す．まず，提案方式の使用者 (以降，ユーザと呼ぶ) が保護対象となるソースコード[*2] P(original source code) の内容を変更し，**見せかけのソースコード** P_f(fake source code) 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(original assembly code) および見せかけのアセンブリコード A_f(fake assembly code) について，**差分情報** (differences) を求める (Step 2)．差分情報とは，「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** (difference) 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 (self-modification routines)，すなわち，復帰ルーチンと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(camouflaged assembly code) を得る．

続く 2.2 において，各ステップの詳細を述べる．

2.2 詳細手順

(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 2(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いが，処理を行った箇所が多くなればなるほど，多くの自己書換え処理が必要となり，実行パフォーマンスの低下の原因となる．

自己書換え処理と実行パフォーマンスの関係については，4 章で議論している．

[*2] 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意味する．

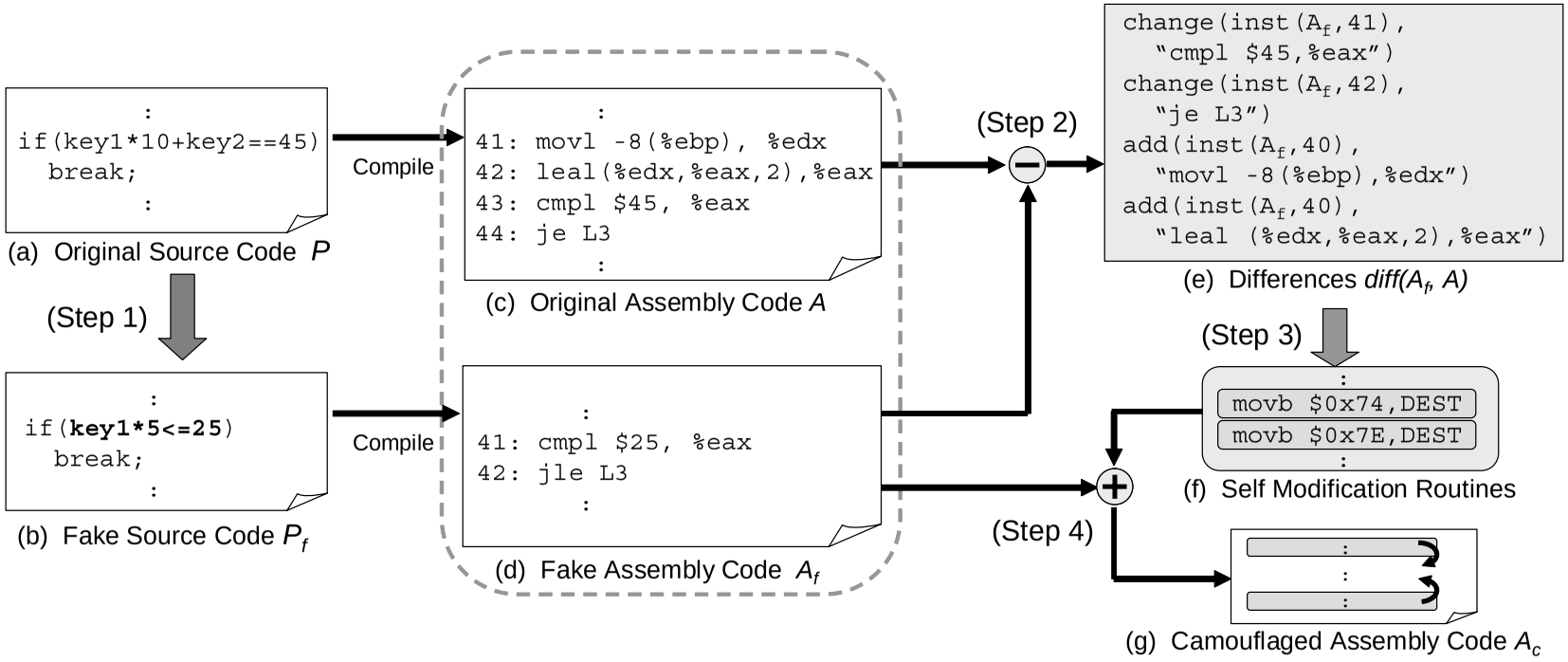


図２　提案方式によるカムフラージュの流れ

(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 つの文字列間の差分を探索するアルゴリズム [5] を応用して，差分要素の集合，すなわち，アセンブリ言語の 1 命令単位での差分の集合を求める．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 2(a) および (b) で示した P および P_f の例に対応するアセンブリコードを，図 2(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 2(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 2(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, ..., d_n について，実行時に差分要素の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プログラムの状態に影響

がない (すなわち，各レジスタの値が変化しない) という条件のもと，命令の内容をランダムに決定し，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 の先頭アドレスとなる．

[手順 2] src_R と src_H を機械語のレベルで比較し，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，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．

上に述べた手順にしたがい，図 2(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．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,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` 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．

[**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．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,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 A_f に挿入し，A_c を得る．従来のカムフラージュ法のアルゴリズム [9] の (Step 1) にしたがうことで，アセンブリのレベルにおいて自動的に挿入位置を決定することができる．この方法では，A_f の一命令を一つのノードとみなした制御フローグラフ (有向グラフ) において，「見せかけの命令が実行される前に必ず元来の命令に書き換えられ，プログラム終了前に，必ず元通り見せかけの命令に書き換えられることが保障される範囲」を抽出し，その範囲の中からランダムに挿入位置を決定する．

以下に，$dest$ のアドレスに存在する命令を書き換える復帰ルーチン RR_k の挿入位置 $P(RR_k)$ および隠ぺいルーチン HR_k の挿入位置 $P(HR_k)$ を決定する手順を示す．

1. $start$ から $dest$ に至るパス (ノードの重複を許さないルート) の集合 T_u を特定する．
2. (1) で特定した全てのパス $t \in T_u$ に共通に含まれるノードのうち，入次数と出次数が共に 1

であるものの集合 N_u を求める．ただし，$dest \notin N_u$ とする．$N_u = \emptyset$ のとき，$dest$ を選びなおし，(1) に戻る．

3. ノード $n_u \in N_u$ をランダムに選択し，n_u への入力辺もしくは出力辺のどちらかを $P(RR_k)$ とする．同様に，
4. $dest$ から end に至るパス (ノードの重複を許さないルート) の集合 T_l を特定する．
5. (4) で特定した全てのパス $t \in T_l$ に共通に含まれるノードのうち，入次数と出次数が共に 1 であるものの集合 N_l を求める．ただし，$dest \notin N_l$ とする．$N_l = \emptyset$ のとき，$dest$ を選びなおし，(1) に戻る．
6. ノード $n_l \in N_l$ をランダムに選択し，n_l への入力辺もしくは出力辺のどちらかを $P(HR_k)$ とする．

3 カムフラージュ例

2.2 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 3(a) に示すような C 言語のプログラムから，図 3(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` 文が削除されることで，制御構造も変更されている．

カムフラージュされたアセンブリコードとして，例えば図 3(c) に示すようなものが得られる．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．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 3(b) をアセンブリコードにコンパイルしたもの) 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．ここでの差分要素は次の 9 つとなっている．

- $d_1 = change(inst(A_f, 16),$“`imull $10,-4(%ebp),%eax`”$)$
- $d_2 = change(inst(A_f, 18),$“`addl -8(%ebp),%eax`”$)$
- $d_3 = change(inst(A_f, 19),$“`cmpl $45,%eax`”$)$
- $d_4 = change(inst(A_f, 20),$“`je L2`”$)$
- $d_5 = change(inst(A_f, 24),$“`jmp L2`”$)$
- $d_6 = delete(inst(A_f, 25))$
- $d_7 = delete(inst(A_f, 26))$
- $d_8 = delete(inst(A_f, 27))$
- $d_9 = delete(inst(A_f, 30))$

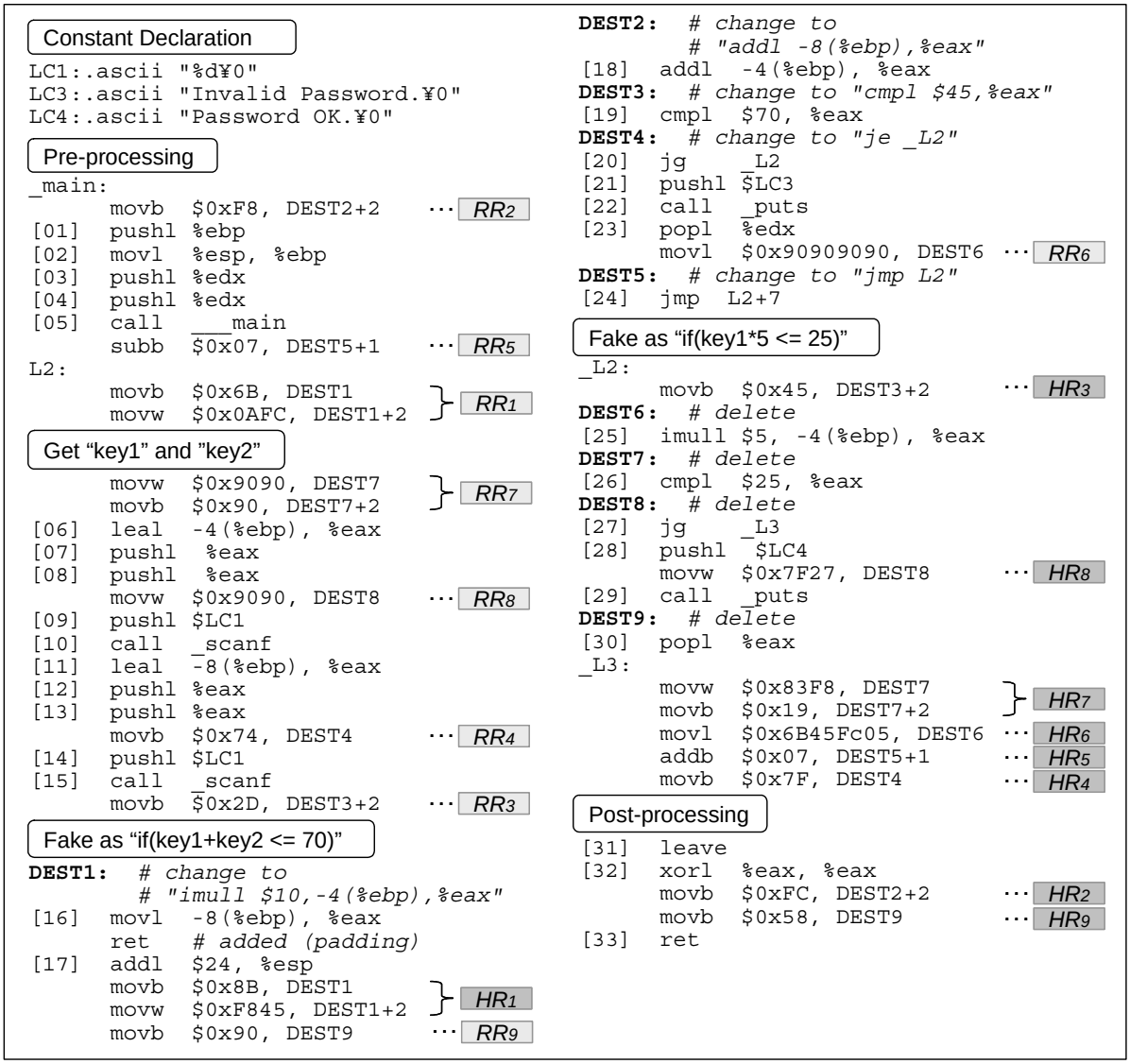
図 3(c) において，実行時に書き換えられる命令は，上の差分要素に対応して `DEST1:`，`DEST2:`，...，`DEST9:`というラベルが付けられている．また，RR_1，RR_2，...，RR_9 および HR_1，HR_2，...，HR_9 と示してある命令列は，それぞれ `DEST1:`，`DEST2:`，...，`DEST9:`に存在する命令を書き

```
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) Original Source Code

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

  scanf("%d", &key1);
  scanf("%d", &key2);

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

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

  return 0;
}
```

(b) Fake Source Code



(c) Camouflaged Assembly Code

図 3　プログラムのカムフラージュ例

換える復帰ルーチンおよび隠ぺいルーチンを示す.

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  -4(%ebp), %eax
cmpl  $70, %eax
jg    _L2
```

という部分を読むと，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 [4] 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，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コストが求められることになる．

4 実行時間のオーバーヘッド

この章では，提案方式を適用することによって生じるプログラムの実行時間の変化 (オーバーヘッド) を測定した結果について報告する．

測定の対象は，内部に含まれた 8 バイトの暗号化されたデータを，入力された 7 バイトの鍵データの値に応じて復号し, その結果を表示するプログラムである．暗号アルゴリズムは, ディジタルコンテンツ保護規格の 1 つ CPPM(Content Protection for Prerecorded Media) /CPRM(Content Protection for Recordable Media) で用いられている C2 暗号 [1] の ECB (Electronic Code Book) モードを使用した．

ここでは，復号処理のサブルーチンを構成する全命令のうち，一定の割合 (具体的な値は後述) の命令が見せかけの命令となるように変更したプログラムを用意し，そのバイナリコードを 100 回連続で実行するのに要した時間を測定した (10 度測定し，平均値を求めた)．このとき，どの命令を見せかけの命令にするか，また，どのような内容で見せかけるかは，ランダムに決定した．また，カムフラージュの度合 (復号処理のサブルーチン全体のアセンブリ命令のうち，何 % の命令が見せかけの命令であるか) は，10% から 50% まで 10% 刻みで変化させた．なお，実験で用いた計算機の CPU は，IA-32(Intel Architecture 32) [6] に基づいており，実行時間の測定は，RDTSC 命令 [7] によって得られるプロセッサのタイムスタンプ・カウンタの差分を用いて行った．

図 4 は，実行時間を測定した結果を示すグラフである．横軸は，カムフラージュの度合を表し，縦軸は，プログラムの実行時間の平均値を表す．全体的な傾向として，カムフラージュの度合が高

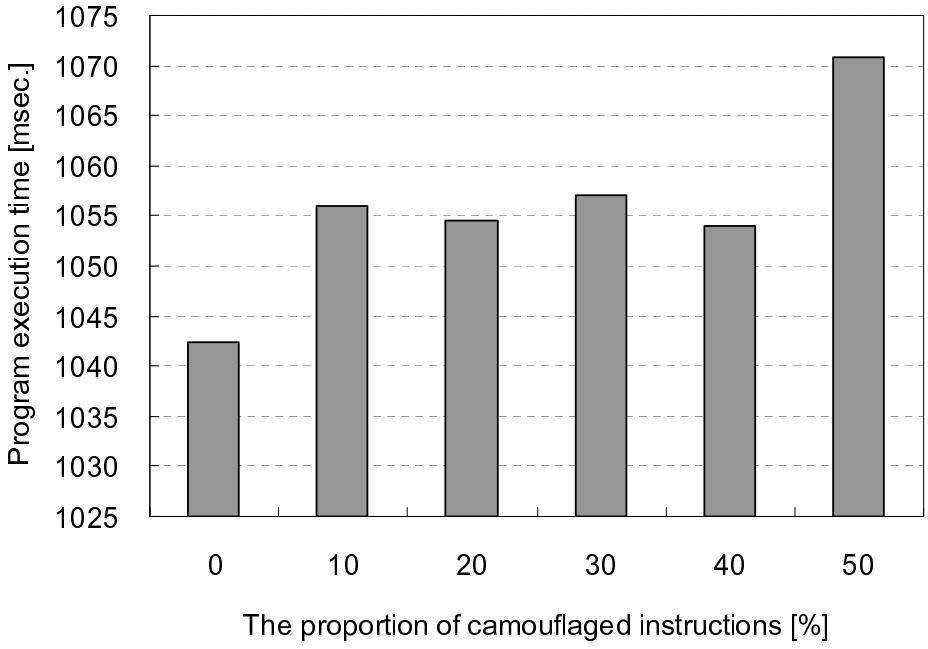


図 4　カムフラージュの度合に応じた実行時間の変化

くなるにしたがって，実行時間の平均値が増大していることがわかる．50% の命令がカムフラージュされたとき，実行時間の平均値は 1.071[秒] となり，どの命令もカムフラージュされていないときの実行時間 (1.042 秒) のおよそ 1.03 倍となっている．このような実行時間の増加の原因は，自己書換えを行うルーチンが，CPU のキャッシュ機能や条件分岐予測機能に影響しているためと考えられる．また，カムフラージュの度合が 20% および 40% のときの実行時間が少し低くなっているのは，自己書換えルーチンが挿入された基本ブロックの実行頻度が低く，自己書換え処理の回数が少なくなったためと考えられる．

5　おわりに

本論文では，高級言語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，任意の命令文を別のものに変更したり，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命令を容易に隠すことができる．

例えば，“`(a+b)*5==10`” という命令を，“`a*8>=10`” という命令に見せかけたい場合，従来方式では，アセンブリコードを読み，制御の流れや変数 `a` の `b` などのデータの流れ，スタックの操作などを理解する必要があった．一方，提案方式では，ソースコードにおいて “`(a+b)*5==10`” を “`a*8>=10`” に変更するだけでよい．また，ユーザがどのような命令に見せかければよいか分から

ない場合は，攻撃者に知られたくない命令をソースコードから削除するだけで，その存在をランダムに決められた偽装内容によって隠すことができる．また，偽装内容の決定が容易であるため，規模の大きい (多くの命令から構成される) 単位での偽装も簡単になる．例えば，プログラム中で用いている暗号アルゴリズム E_1 を，別の暗号アルゴリズム E_2 に置き換えるだけで，E_1 ではなく E_2 を用いているように攻撃者に見せかけることができる．

謝辞

本研究の一部は，日本学術振興会 科学研究費補助金若手研究 (スタートアップ)，課題番号 18800079 による補助のもとで行われた．

参考文献

[1] 4C-Entity. *Policy statement on use of content protection for recordable media, (CPRM) in certain applications*, 2001. (Available online).

[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] 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.

[5] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, Vol. 1, No. 2, pp. 251–266, 1986.

[6] IA-32 インテル・アーキテクチャ・ソフトウェア・デベロパーズ・マニュアル上巻: 基本アーキテクチャー. インテル株式会社. http://www.intel.co.jp/ (Available online).

[7] IA-32 インテル・アーキテクチャ・ソフトウェア・デベロパーズ・マニュアル中巻:命令セット・リファレンス. インテル株式会社. http://www.intel.co.jp/ (Available online).

[8] 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.

[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.