命令のカムフラージュによるソフトウェア保護方法

神﨑雄一郎† 門田　暁人† 中村　匡秀† 松本　健一†

A Software Protection Method Based on Instruction Camouflage

Yuichiro KANZAKI†, Akito MONDEN†, Masahide NAKAMURA†, and Ken'ichi MATSUMOTO†

あらまし　本論文では，プログラムに含まれる多数の命令をカムフラージュ（偽装）することにより，悪意をもったユーザ（攻撃者）によるプログラムの解析を困難にする方法を提案する．提案方法では，プログラム中の任意の命令（ターゲット）を異なる命令で偽装し，プログラムの自己書換え機構を用いて，実行時のある期間においてのみ元来の命令に復元する．攻撃者がカムフラージュされた命令を含む範囲の解析を試みたとしても，ターゲットの書換えを行うルーチン（書換えルーチン）の存在に気づかない限り，プログラムの元来の動作を正しく理解することは不可能である．解析を成功させるためには，書換えルーチンを含む範囲についても解析する必要があり，結果として，攻撃者はより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．提案方法は自動化が容易であり，要求される保護の強さ，及び，許容される実行効率の低下の度合に応じて，ターゲットの個数を任意に決定できる．

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，自己書換え

1. まえがき

ネットワークの普及によってプログラムやディジタルコンテンツの流通形態が著しく進歩する中，エンドユーザによるプログラム内部の解析，及び，改ざんを防止する技術への要求が高まっている．例えば，ディジタルデータの著作権管理 (Digital Rights Management, DRM) を行うプログラムは，内部に含まれる復号鍵の漏えいを防止することが求められる [3], [24]．また，携帯電話やセットトップボックス等のハードウェアに含まれる組込プログラムも，ユーザによる解析・改ざんを防ぐ必要に迫られている [23]．解析による問題が起きた例として，DVD データの暗号解除ツールが流通した事件が挙げられる [6], [22]．このツールは，DVD 再生プログラムの解析結果に基づいて作成され，DVD の違法コピーを助長する大きな問題となった．

本論文における解析とは，プログラム中の秘匿情報（暗号鍵やアルゴリズム等）を得ようとする行為のことを指し，典型的に次のような手順によって行われると想定する．まず，攻撃者は，プログラムを逆アセンブルし，得られたアセンブリプログラムの理解を試みる [17]．ただし，大規模プログラムの全体を理解することは多大な労力・時間を要すため，現実的とはいえない．そこで，理解すべき範囲（秘匿情報に関係すると思われる範囲）を絞り込んでから，その範囲に限定して理解を試みる [1], [2]．このような範囲の絞込みと理解は，目的とする秘匿情報が得られるまで繰り返し行われる．

本論文では，このような範囲の絞込みを伴うアセンブリプログラムの解析を困難にすることを目的として，プログラムに含まれる多数の命令をカムフラージュ（偽装）する方法を提案する．提案方法では，プログラム中の任意の命令（ターゲット）を異なる命令で偽装し，プログラムの自己書換え機構を用いて，実行時のある期間においてのみ元来の命令に復元する [13], [15]．攻撃者がカムフラージュされた命令を含む範囲の解析を試みたとしても，ターゲットの書換えを行うルーチン（書換えルーチン）の存在に気づかない限り，プログラムの元来の動作を正しく理解することは不可能である．解析を成功させるためには，書換えルーチンを含む範囲についても解析する必要があり，結果として，攻撃者はより広範囲にわたるプログラムの解析を強いられ

† 奈良先端科学技術大学院大学情報科学研究科，生駒市
Graduate School of Information Science, Nara Institute of Science and Technology, 8916–5 Takayama, Ikoma-shi, 630–0192 Japan

ることとなる．提案方法は自動化が容易であり，ターゲットの個数を任意に決定できる．多数のターゲット，及び，多数の書換えルーチンをプログラム中に分散させることで，範囲を絞った解析を著しく困難にできると期待される．

以降，**2.** では，自己書換えを用いてプログラム中の多数の命令をカムフラージュする系統的な方法を提案する．**3.** では，提案方法に対する攻撃，及び，その困難さと防御について考察する．**4.** では，提案方法を用いたケーススタディについて報告する．**5.** では，関連研究について述べる．最後に **6.** において，結論と今後の課題を述べる．

2. 命令のカムフラージュを用いたソフトウェア保護の方法

2.1 攻撃者モデル

本論文では，攻撃者のモデルを次のように仮定している．

- 攻撃者は，逆アセンブラを所有し，それを用いて範囲の絞込みを伴う静的な解析を行う能力をもつ．
- 攻撃者は，ブレークポイントの機能をもったデバッガを所有し，プログラムの任意の個所に（手動で）ブレークポイントを設定することで，任意の実行の時点におけるスナップショット（メモリにロードされている解析対象のプログラムの内容）を取得する能力をもつ．ただし，スナップショット収集の自動化，及び，収集したスナップショットの履歴を用いた動的解析の自動化を行えるツールを所有しない．また，そのようなツールを作成する能力をもたない．

なお，この攻撃者は，Monden らの能力別の攻撃者のモデル [18] における「レベル 2 の攻撃者」に該当する．

上記の攻撃者モデルを前提とした場合，プログラム保護のメカニズムは，次の性質を満たすことが要求される．

- 逆アセンブラを用いた静的解析により，保護のメカニズムが容易に無効化されない．
- （少数の）スナップショットを用いた攻撃により，保護のメカニズムが容易に無効化されない．

続く節において，上記の性質を満たす保護方法を提案する．

2.2 キーアイデア

提案方法は，プログラム中の命令をカムフラージュすることによって，攻撃者によるプログラムの理解を困難にする．カムフラージュとは，元来の命令を，内容の異なった偽（にせ）の命令で上書きすることで，元来の命令の存在を攻撃者から隠すことである．

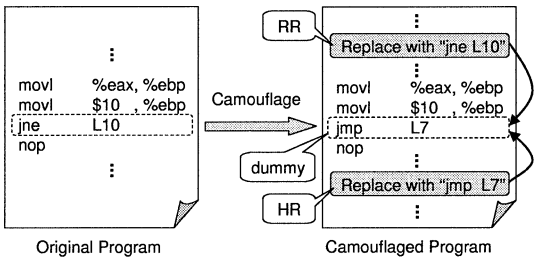


図 1 カムフラージュの例
Fig. 1 Example of camouflage.

図 1 は，カムフラージュの例を示したものである(注1)．保護の対象となるアセンブリプログラム中の `jne L10` という命令がカムフラージュされるとき，まず，`jne L10` のダミー命令，すなわち，`jne L10` と異なった内容をもつ命令が作成される．ダミー命令として，`jmp L7` が作成されたとすると，`jmp L7` が `jne L10` の存在する位置に上書きされる．

続いて，自己書換えを行うルーチンが追加される．自己書換えとは，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．自己書換えルーチンには二つの種類が存在する．一つは，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もつルーチン（図 1 中 RR）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．図 1 の例の場合，カムフラージュする命令を `jne L10` へ自己書換えするルーチンとなる．もう一つは，RR によって元来の命令となった命令を，再びダミー命令に書き換える役割をもつルーチン（図 1 中 HR）である．このルーチンは，スナップショットを取得できる能力をもつ攻撃者によって，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，図 1 の例の場合，カムフラージュする命令を `jmp L7` へ自己書換えするルーチンとなる．ダミー命令は，RR が実行されてから，HR が実行されるまでの間のみ，元来の命令となる．攻撃者は，カムフラージュされた命令の付近を読んだだけでは，元来の命令がダミー命令である `jmp L7` によって上書きされていることに気づくのは難しい．また，HR が実行された後の時点におけるプログラム

(注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．

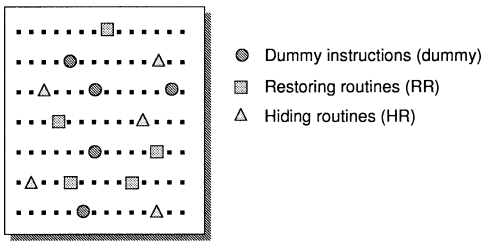


図 2　カムフラージュされたプログラムの概念図
Fig. 2　Image of a camouflaged program.

のスナップショットを取得しても，得ら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．

保護の対象となるアセンブリプログラムに対して，以上に述べたような命令のカムフラージュを複数繰り返すことで，プログラムの理解を困難にする．図 2 は，命令のカムフラージュを多数繰り返したプログラムの概念図である．プログラム中の多数の命令が，実行前にダミー命令で上書きされている（図 2 中の●）．また，各々のダミー命令について，実行時に元来の（ダミー命令で上書きされる前の）命令に自己書換えするルーチン（図 2 中の■）と，そのルーチンによって元来の命令に書き換わった命令を，実行時に再びダミー命令に自己書換えするルーチン（図 2 中の▲）が存在する．

攻撃者が解析を試みる部分にダミー命令が含まれていると，その部分を読むだけでは，プログラムの元来の動作を正しく理解することができない．正しく理解するためには，自己書換えが行われていることに気づき，理解したい部分に含まれる各ダミー命令について，それぞれの上書きされる前の内容を知る必要がある．しかし，それらを知るためには，プログラム全体から元来の命令に書き換えるルーチンを探し出す必要があり，非常に大きなコストがかかる．

2.3 諸 定 義

提案方法に関する用語の定義を行う．

オリジナルプログラム O は，カムフラージュの対象となるプログラムで，カムフラージュされていない状態のものを指す．ターゲット命令は，O の中に存在する，カムフラージュのターゲットとなる命令である．ダミー命令は，ターゲット命令をカムフラージュするために，ターゲット命令に上書きされる命令である．ユーザが複数のターゲット命令を設けるとき，i 番目のターゲット命令を $target_i$ とし，各々の $target_i$ をカムフラージュするための命令を，$dummy_i$ とする．

復帰ルーチンは，ダミー命令によってカムフラージュされた命令を，(元来の）ターゲット命令に自己書換えするためのルーチン（一連の命令）である．$dummy_i$ を $target_i$ に書き換える復帰ルーチンを，RR_i とする．一方，隠ぺいルーチンは，ターゲット命令をダミー命令に自己書換えするためのルーチンである．$target_i$ を $dummy_i$ に書き換える隠ぺいルーチンを，HR_i とする．復帰ルーチン及び隠ぺいルーチンを総称して，自己書換えルーチンと呼ぶ．

カムフラージュされた命令とは，実行時に内容が（$target_i$ か $dummy_i$ に）変化する命令である．カムフラージュ化プログラム M は，カムフラージュされた命令を含むアセンブリプログラムである．

続く節では，オリジナルプログラム O から，カムフラージュ化プログラム M を得るための系統的な方法を述べる．

2.4 カムフラージュ化プログラム M の作成手順

次の (Step 1)〜(Step 6) により M を作成する．

(Step 1) ターゲット命令と自己書換えルーチンの位置の決定

$target_i$，及び，RR_i，HR_i のプログラム上の位置を決定する．以降，RR_i の位置及び HR_i の位置をそれぞれ $P(RR_i)$ 及び $P(HR_i)$ と記す．

まず，M を構成する命令から $target_i$ をランダムに決定する，若しくは，プログラム開発者が直接指定する．次に，アセンブリプログラムの 1 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，次の 4 条件を満たすように $P(RR_i)$ 及び $P(HR_i)$ を決定する．これらの条件は，「$dummy_i$ が実行される前に必ず $target_i$ に書き換えられ，プログラム終了前に，必ず元どおり $dummy_i$ 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 $target_i$ に至るすべてのパスに $P(RR_i)$ が存在する．

[条件 2]　$P(RR_i)$ から $target_i$ に至るすべてのパスに $P(HR_i)$ が存在しない．

[条件 3]　$P(HR_i)$ から $target_i$ に至るすべてのパスに $P(RR_i)$ が存在する．

[条件 4]　$target_i$ からプログラム終了点 end に至るすべてのパスに $P(HR_i)$ が存在する．

図 3 に，条件 1〜4 を満たした $P(RR_i)$ 及び $P(HR_i)$ の例を示す．

次に，条件 1〜4 を満たす $P(RR_i)$，$P(HR_i)$ を決定する手順を示す．

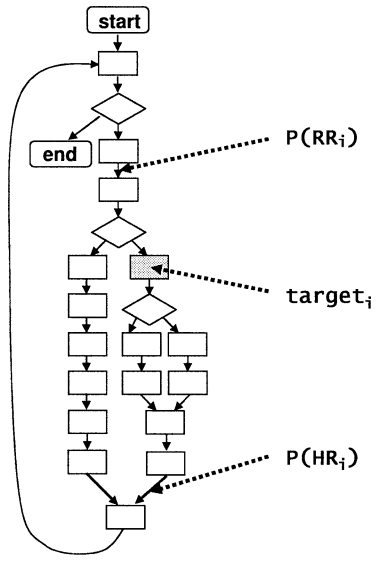


図 3　四つの条件を満たす $target_i$，$P(RR_i)$ 及び $P(HR_i)$ の例
Fig. 3　Example of $target_i$, $P(RR_i)$ and $P(HR_i)$ that satisfy four conditions.

（1）　$start$ から $target_i$ 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）で特定したすべてのパス $t \in T_u$ に共通に含まれるノードのうち，入次数と出次数がともに 1 であるものの集合 N_u を求める．ただし，$target_i \notin N_u$ とする．$N_u = \emptyset$ のとき，$target_i$ を選び直し，（1）に戻る．

（3）　ノード $n_u \in N_u$ をランダムに選択し，n_u への入力辺若しくは出力辺のどちらかを $P(RR_i)$ とする．同様に，

（4）　$target_i$ 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）で特定したすべてのパス $t \in T_l$ に共通に含まれるノードのうち，入次数と出次数がともに 1 であるものの集合 N_l を求める．ただし，$target_i \notin N_l$ とする．$N_l = \emptyset$ のとき，$target_i$ を選び直し，（1）に戻る．

（6）　ノード $n_l \in N_l$ をランダムに選択し，n_l への入力辺若しくは出力辺のどちらかを $P(HR_i)$ とする．

(Step 2) ダミー命令の決定

$target_i$ と同一の命令長をもつ任意の命令を選択し，ダミー命令 $dummy_i$ とする．ここでは，$target_i$ を構成するオペコード，若しくは，オペランドのうちの 1 バイトを変更したものを $dummy_i$ として選択する例を示す．次の $target_i$ について考える．

（16 進による機械語表現）　`03 5D F4`
（アセンブリ表現）　`addl -12(%ebp),%ebx`

この $target_i$ におけるオペコード `03` を `33` に変更することで，次の $dummy_i$ ができる．

（16 進による機械語表現）　`33 5D F4`
（アセンブリ表現）　`xorl -12(%ebp),%ebx`

また，$target_i$ のオペランド `F4` を `FA` に変更することで，次の $dummy_i$ ができる．

（16 進による機械語表現）　`03 5D FA`
（アセンブリ表現）　`addl -6(%ebp),%ebx`

(Step 3) 自己書換えルーチンの生成

次の手順に従い，自己書換えルーチン RR_i 及び HR_i を生成する．

（1）　$target_i$ の直前に，ラベル L_i(注2)を挿入する．これにより，L_i を用いて $target_i$ を間接参照できる．

（2）　L_i を用いて，$dummy_i$ を $target_i$ に書き換えるための（一連の）命令を作り，これを RR_i とする．

（3）　L_i を用いて，$target_i$ を $dummy_i$ に書き換えるための（一連の）命令を作り，これを HR_i とする．

次に例を示す．$target_i$ として，`addl -12(%ebp),%ebx` を想定し，$dummy_i$ として `xorl -12(%ebp),%ebx` を想定する．まず，前述の $target_i$ にラベル `L1` を挿入する．

```
L1:  addl -12(%ebp),%ebx
```

次に，RR_i を生成する．RR_i は，`L1` に存在する命令の 1 バイト目を「`33`」から「`03`」へ変更する機能をもつ．

```
movb $0x03,L1
```

この 1 命令からなる小さなアセンブリのルーチンは，「`L1` の指すアドレスの内容を，即値 `03`（16 進）で上書きせよ」という意味をもつ．

RR_i が実行されると，$dummy_i$ は $target_i$ に書き換わる．

同様に，HR_i を生成する．HR_i は，`L1` に存在する命令の 1 バイト目を「`03`」から「`33`」に変更する機能をもつ．

(注2)：ラベルとは，アセンブリ言語において，プログラム内の命令の位置（メモリ番地）を指し示す名前のことを指す．

```
movb $0x33,L1
```

HR_i が実行されると，$target_i$ は $dummy_i$ に書き換わる．

(Step 4) ダミー命令の書込みと自己書換えルーチンの挿入

(Step 2) で生成したダミー命令 $dummy_i$ を，(Step 1) で決定した $target_i$ に上書きする．これにより，プログラム実行前の時点において，$target_i$ が $dummy_i$ により偽装された状態となる．次に，(Step 3) で生成した自己書換えルーチン RR_i 及び HR_i を，(Step 1) で決定した $P(RR_i)$ 及び $P(HR_i)$ にそれぞれ挿入する．

(Step 5) 自己書換えルーチンの変形

自己書換えルーチンは，ラベル（即値アドレス）によってプログラム領域内の $target_i$ のアドレスを指定し書き換えるという特徴をもつため，攻撃者による（静的な）解析によってその位置を知られ，その結果，$target_i$ の位置が特定される可能性がある．例えば，プログラム内に存在する `movb` 命令が，第 2 オペランドとしてプログラム領域内を指し示す即値アドレスを保持する場合に，その `movb` 命令は自己書換えルーチンであると推測される可能性がある．

そこで，静的解析を困難にするために，自己書換えルーチンの変形を行う．ラベルに対して演算を行うことでプログラム領域への書込みをしていないように見せかけたり，機械語命令の難読化 [20] や mutation [11] の従来技術を併用して，静的なパターンマッチングによる自己書換えルーチンの特定を困難にする．`movb $0x03,L1` の変形の例を次に示す．

```
movl $L1 + 1250, %eax
subl $1250, %eax
movb $0x03,(%eax)
```

このアセンブリルーチンをアセンブルして得られた機械語プログラム中には，`L1` は現れない（`L1` に 1250 を足した値が現れる）．そのため，静的解析によってアドレス `L1`（すなわち $target_i$ の存在位置）を特定することはより困難となっている．なお，`L1` に 1250 を足したアドレスは，プログラム領域を指すとは限らない．また，`movb` 命令の第 2 オペランドは即値アドレスではなくレジスタ`%eax` の指すアドレスとなっており，静的解析によってその値を知ることはより困難となっている．このような変形に加えて，難読化や mutation を併用することで，パターンマッチングやアドレス解析による攻撃を更に困難にできる．

(Step 6) 以上のステップの繰返し

(Step 1) から (Step 5) を繰り返す．1 回繰り返すごとに，カムフラージュされた命令が 1 個ずつ増える．**4.** で述べるように，カムフラージュ命令を増やすことと，実行効率の低下はトレードオフの関係にある．したがって，要求される保護の強さ，及び，許容される実行効率の低下の度合に応じて，繰返しの回数を決めることが望ましい．

2.5 カムフラージュ化プログラム M の作成例

カムフラージュ化プログラムの例を，図 4 に示す．(a) はオリジナルのプログラムを示し，(b) はカムフラージュ化プログラムを示す．

(a) から (b) を得るための手順の例を次に述べる．1 回目のカムフラージュにおいて，図 4 (a) に点線枠で示される命令 (`addl -12(%ebp),%ebx`) が $target_1$ として選択され，図 4 (b) で示されるように，$dummy_1$(`xorl -12(%ebp),%ebx`) で上書きされる．そして，$target_1$ を自己書換えするルーチン RR_1 及び HR_1 が生成・挿入される．2 回目のカムフラージュにおいて，RR_1 を構成する命令の一つ（1 回目のカムフラージュが終わった時点では（`movb $0x03,(%eax)`）が $target_2$ として選択され，図 4 (b) で示されるように，$dummy_2$(`movb $0x4a,(%eax)`) で上書きされる．そして，$target_2$ を自己書換えするルーチン RR_2 及び HR_2 が生成・挿入される．

この例の場合，RR_1 の一部が $dummy_2$ によって書き換えられているため，$dummy_1$ の元来の命令を知るためには，RR_1 だけでなく RR_2 も発見する必要がある．

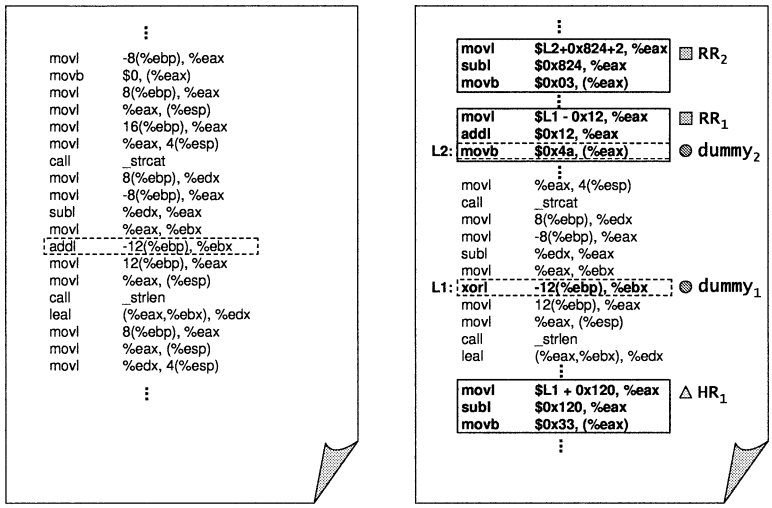
なお，付録として，条件分岐を一つ含む簡単なプログラムと，そのカムフラージュ化プログラムのリストを巻末に示す．

3. 解析の困難さに関する議論

3.1 想定する解析方法

2.1 で述べた攻撃者が M の秘匿部分 $C(M)$ の解析を行う際，想定される解析方法について考える．

まず，解析のゴールを「攻撃者が $C(M)$ を正しく理解すること」とする．$C(M)$ を正しく理解するためには，$C(M)$ に含まれるダミー命令それぞれについて，対応する元来の命令を知る必要がある．そのためには，$C(M)$ に含まれる各々のダミー命令に対応する



(a) Original Program (b) Camouflaged Program

図 4 カムフラージュ化プログラムの例
Fig. 4 Example of a camouflaged program.

復帰ルーチンを，プログラム全体から発見する必要がある．

攻撃者が利用し得る解析方法として，静的解析と動的解析の二つが考えられる．静的解析は，解析対象のプログラムを実行させずに解析する方法である．典型的な方法は，キーワード検索やパターンマッチングなどを M に適用して $C(M)$ の範囲を絞り込み，$C(M)$ を理解していく．プログラム M の全体を考慮せず，$C(M)$ に集中して解析を行うため，後に述べる動的解析よりも解析コストが低く，一般的によく用いられる．提案手法は，この静的解析を困難にすることを第一目的としている．

一方，動的解析は，解析対象のプログラムを実行させながら解析する方法である．攻撃者は，デバッガなどのツールを用いて M を実行し，ツールの出力情報を頼りに $C(M)$ の特定及び理解を試みる．動的解析により，攻撃者は M の実行を完全に追跡できるが，解析が入力に依存することやプログラム M 全体を実行させなければならないことから，M が大規模になると解析コストが非常に高くなる．また，商用のプログラムでは一般的にデバッグ情報が削除されていたり，意図しない実行を禁止するような工夫がなされていることがあるため，すべてのプログラムに対して動的解析が適用できるとは限らない．ただし，実行されたプログラムの任意の時点におけるスナップショット，すなわち，実行の任意の時点においてメモリ上にロードされている対象プログラムの内容を保存し，その結果を用いて静的解析の成功を容易にすることは，比較的低いコストで実現可能である．そのため，いくつかのスナップショットを取得された場合でも，保護が容易に無効化されないようにする必要がある．

続く節において，それぞれの解析方法に対する M の安全性について述べる．

3.2 静的解析に対する安全性

静的解析に対するプログラム M の安全性を示すために，攻撃者が秘匿部分 $C(M)$ を正しく理解できる確率を定式化してみる．

まず，M にダミー命令 $dummy_i$ が一つだけ含まれるとき，攻撃者が M における長さ m の任意のコードブロック $D(M)$ を正しく理解するには，以下の事象 E_i が成立しなければならない．

> E_i : $dummy_i$ が $D(M)$ に存在しない，または，$dummy_i$ が $D(M)$ に存在し，かつ RR_i が $D(M)$ に存在する．

$D(M)$ において $dummy_i$ が存在しない（つまりカムフラージュが全くされていない）場合は，解析者は $D(M)$ をそのまま追跡できるため，$D(M)$ の元来の動作が容易に理解されてしまう．また，$D(M)$ に $dummy_i$ が存在しカムフラージュされているにもかかわらず，その復帰ルーチン RR_i も $D(M)$ 中に存在した場合，RR_i の解析により $target_i$ が同定され，

$D(M)$ の元来の動作が露呈してしまうことになる．

今，M の命令数を L とし，M において $dummy_i$, RR_i をランダムに決定した場合，E_i が成り立つ確率 $P(E_i)$ は次のように表される．

$$P(E_i) = \frac{L-m}{L} + \frac{m}{L} \times \frac{m}{L} = \frac{(L-m)^2 + Lm}{L^2}$$

次に n 個のダミー命令 $(dummy_1, \ldots, dummy_n)$ が M に含まれる場合，攻撃者が $D(M)$ の解析に成功するためには，E_i がすべての i $(1 \leq i \leq n)$ について成立しなければならない．したがって，$D(M)$ の解析に成功する確率 $P(Success, D)$ を概算すると，次のようになる．

$$P(Success, D) = \left(\frac{(L-m)^2 + Lm}{L^2}\right)^n$$

図 5 は，$P(Success, D)$ と n の関係を示すグラフである．横軸は，M 中のカムフラージュされた命令数 n を表し，縦軸は，解析が成功する確率 $P(Success, D)$ を表す．$D(M)$ の命令数 m は 100 に設定した．M の命令数 L は 1000，2000，3000 と変化させ，各々の場合についての結果を示している．図 5 より，ダミー命令数 n が増加すると（つまり，カムフラージュの度合を増加させると），$D(M)$ の解析成功確率が 0 に近づくことが分かる．

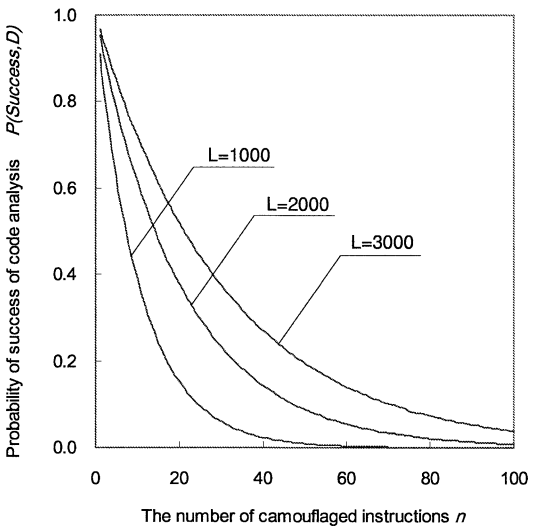


図 5　長さ m の任意のコードブロックの解析が成功する確率 $(m = 100)$

Fig. 5　Probability of success of code analysis $(m = 100)$.

秘匿部分 $C(M)$ が，攻撃者が任意に選んだコードブロック $D(M)$ に一致する（または含まれる）場合，M に対する静的解析が成功したことになる．$C(M)$ の特定（推定）は，攻撃者のスキルに依存するため，確率論では定式化することが困難であるが，仮に $C(M)$ が $D(M)$ に含まれる確率を X とすると，解析が成功する確率 $P(Success)$ は以下のようになる．

$$P(Success) = X \times P(Success, D) = \left(\frac{(L-m)^2 + Lm}{L^2}\right)^n X$$

以上の定式化により，攻撃者が静的解析の成功確率を上げるには，$C(M)$ をうまく推定して X を上げるか，解析する部分 $D(M)$ のサイズ m を広げる必要があることが分かる．一方，提案法の利用者はカムフラージュ命令数 n を増加させることで，$P(Success)$ を容易に制御できることも分かる．

なお，上述の議論においては，L が増加すると $P(Success)$ も増加してしまう．これは，事象 E_i の定式化において，$dummy_i$ をランダムに決定しているため，$C(M)$ の中に $dummy_i$ がうまく挿入されない場合があるからである．しかし，利用者が $C(M)$ の位置をあらかじめ知っている場合には，$dummy_i$ を $C(M)$ 内に挿入したり，RR_i と $dummy_i$ との距離を想定される m より大きくとることで，$P(E_i)$ を下げることができ，結果として解析の成功確率を下げることができる．

一方，利用者が $C(M)$ の位置を正確に把握していない場合や，攻撃者が $C(M)$ を推定する確率 X を下げたい場合には，M を L/m 個のブロックに分け，それぞれのブロックに定数個のカムフラージュ命令を挿入しておけばよい．こうすれば，攻撃者がどの任意のブロック $D(M)$ を解析しようとしても，一様にカムフラージュ命令が存在するため，解析が困難になる．

3.3　動的解析に対する安全性

デバッガを用いて，実行時のある位置で M を停止させたとき，$C(M)$ に存在するダミー命令のいくつかが，元来の命令に書き換わった状態になり得る．このとき，攻撃者がスナップショットを取得し，メモリ上にロードされたプログラムの $C(M)$ に相当する部分を観察すると，いくつかの元来の命令を知ることができる．このことは，$C(M)$ を理解されたくない者にとって脅威である．ただし，$C(M)$ に存在するダミー

命令の元来の内容をすべて知るのは困難であるといえる．なぜなら，$C(M)$ 中のダミー命令を書き換える復帰ルーチンはプログラム全体にわたって散在しているため，それらすべてを実行させるためには，プログラム中の様々な位置を実行させなければならないからである．プログラム全体が理解できていない限り，その作業は大きなコストを要する．また，隠ぺいルーチンが実行された時点で，元来の内容に戻された命令は再びダミー命令となるため，攻撃者はプログラムの終了の直前においても，多くの命令が元来の命令に戻されたスナップショットを得ることはできない．

しかし，特に $C(M)$ に含まれるダミー命令が少ない場合，動的解析は効率の良い攻撃となり得るため，割込み命令などを用いてデバッガの動作を妨げることで動的解析を困難にする技術 [1] を併用することが望ましい．これにより，動的解析に対する M の安全性を高めることができる．

4. ケーススタディ

4.1 概　　要

この章では，提案方法が適用されたソフトウェアに対して次の三つの項目を測定し，その結果について報告する．

（1） ターゲット命令と復帰ルーチン間の距離
（2） ファイルサイズの変化
（3） 実行時間の変化

提案方法を適用するソフトウェアとして，ファイルを暗号化・復号化するためのツール `ccrypt` を選んだ．このソフトウェアは，GPL ライセンスに基づくフリーソフトウェアとして公開されているものである(注3)．

我々は，提案方法に基づいて，プログラムをカムフラージュするシステムを試作し [14]，そのシステムを用いて，対象プログラムに下記の手順で提案方法を適用した．

（1） C 言語のソースファイル $s_1, s_2, \ldots, s_n$ をコンパイルして，オリジナルのアセンブリファイル $a_1, a_2, \ldots, a_n$ を得る．

（2） $a_1, a_2, \ldots, a_n$ に対してそれぞれカムフラージュを施し，カムフラージュされたアセンブリファイル $a'_1, a'_2, \ldots, a'_n$ を得る．

（3） $a'_1, a'_2, \ldots, a'_n$ をアセンブルして，実行モジュール $o_1, o_2, \ldots, o_n$ を得る．

（4） $o_1, o_2, \ldots, o_n$ をリンクして，実行可能ファイル p を得る．

いずれの実験においても，実行可能ファイル p が正しく動作することを確認した．

なお，Windows 上で動作する実行可能ファイル（Microsoft Portable Executable 形式など）は，ファイル中のセクションヘッダ内のフラグによって，コード領域への書込みの許可/不許可が制御されている [16]．提案方法を適用する際には，あらかじめこのフラグを立てておくことで，コード領域を実行時に書換え可能とする必要がある．

実験で用いた計算機は，OS が Windows XP，メインメモリのサイズが 512 MByte，CPU が Pentium 4（クロック周波数 1.5 GHz，1 次トレース・キャッシュ 12 kμOps，1 次データ・キャッシュ8 kByte，2 次キャッシュ256 kByte）である．

4.2 ターゲット命令と復帰ルーチン間の距離

図 6 は，一つのカムフラージュされたアセンブリファイルについて，ダミー命令と復帰ルーチンの分布の様子を示したものである．このファイルは，カムフラージュ前の状態において，行数が 1490，命令数が 947 であり，そのうち 130 の命令をカムフラージュした．縦軸は行番号を示し，横軸は行番号の 30 の剰余を示す．縦軸で示される値に，横軸で示される値を加算したものが，命令あるいは復帰ルーチンの存在する行

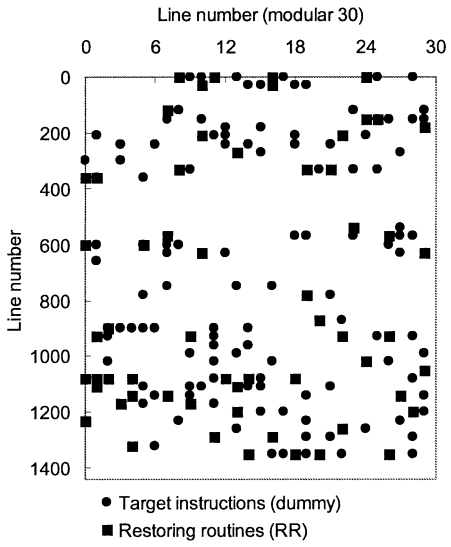


図 6　ターゲット命令と復帰ルーチンの分布
Fig. 6　Distribution of target instructions and restoring routines.

(注3)：http://quasar.mathstat.uottawa.ca/~selinger/

表 1　ターゲット命令と復帰ルーチンの距離
Table 1　Distance between target instructions and restoring routines.

	平均値	最大値	最小値	標準偏差
距離 [命令]	151	611	1	192

番号となる．図 6 より，ターゲット命令及び復帰ルーチンは，プログラム全体に散在していることが分かる．

表 1 に，ターゲット命令と復帰ルーチン間の距離の平均値，最大値，最小値及び標準偏差を示す．表 1 より，このプログラム中のある命令がカムフラージュされた命令かどうかを知るために，平均で 151 命令，最大で 611 命令離れた位置にある復帰ルーチンを探し出す必要があることが分かる．このプログラムはおよそ 7 命令に 1 命令の割合でカムフラージュされているため，復帰ルーチンを探す過程においても，カムフラージュされている命令が多数現れる．また，**2.5** の例で述べたように，復帰ルーチンを構成する命令がカムフラージュされる場合もある．そのため，復帰ルーチンを見つけるための解析には大きなコストを要すると予想される．

最小値が 1 命令になっていることから，ターゲット命令と復帰ルーチンが隣接して出現する場合があることが分かる．ターゲット命令の位置や復帰ルーチンの挿入位置が，候補の中からランダムに選ばれるため，このような場合が現れる．ターゲット命令と復帰ルーチンが一定の距離以上離れるように，**2.4** (Step 1) で述べた挿入位置を決定するアルゴリズムを改良することは，今後の課題の一つである．

4.3　ファイルサイズの変化

カムフラージュ化プログラムのファイルサイズを調べると，カムフラージュされた命令数に比例して，ファイルサイズが増加していることが分かった．平均すると，カムフラージュされた命令数が 100 増加するごとに，ファイルサイズが約 2.4 kByte 増加する．このようなファイルサイズの増加が発生するのは，カムフラージュされる命令数を増やす量に応じて，挿入される自己書換えルーチンが増加するためである．

ファイルサイズの増加は，2 次記憶装置が大容量化する傾向にあることを考慮に入れると，それほど大きなデメリットにはならないと考えられる．ただし，ファイルサイズに関する制約が厳しい環境においては，ファイルサイズの増加を最小限に抑えなければならない場合も考えられる．そのような場合は，ファイルサイズが制約の範囲内に収まるようにカムフラージュする命令数を調整することで対処できる．

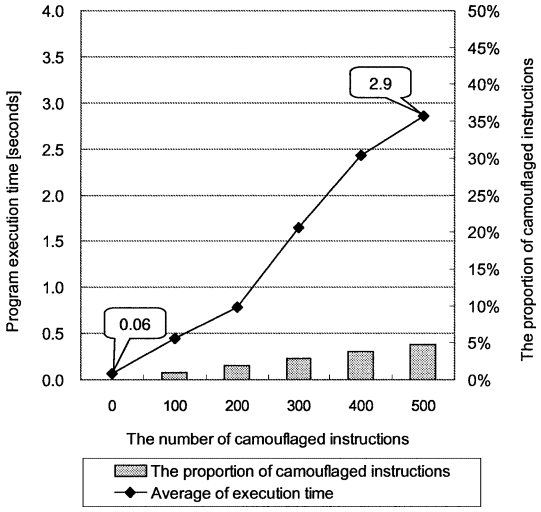


図 7　ccrypt の実行時間の変化
Fig. 7　Impact on program execution time.

4.4　プログラムの実行時間の変化

カムフラージュされた `ccrypt` が 100 kByte のテキストファイルを暗号化するのに要した時間を，カムフラージュされた命令数を変化させるごとに 10 回ずつ測定し，それぞれ平均値を計算した．実行時間の測定は，対象となるカムフラージュ化プログラムを起動させる直前におけるシステム時計の経過時間と，プログラム終了直後におけるそれとの差分を用いて行った．なお，システム時計の経過時間は，C 言語の clock 命令を用いて取得した．

図 7 は，実行時間を測定した結果を示すグラフである．横軸は，カムフラージュされた命令の数を表し，縦軸は，プログラムの実行時間の平均値，及び，カムフラージュ率（棒グラフで示される）を表す．ここで，カムフラージュ率とは，プログラムがカムフラージュされている度合を示すものである．

図 7 より，カムフラージュされた命令数が多くなるに従って，実行時間の平均値が増大していることが分かる．500 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約 2.9 秒となる．これは，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（約 0.06 秒）のおよそ 48 倍である．このような実行時間の増加の原因として，次の三つが考えられる．

（1）　自己書換えルーチンが挿入されることにより，

実行される命令数が増える．

（2） CPU にキャッシュされているコードに対して自己書換えルーチンが書込みを行うごとに，対応するキャッシュラインが無効化される [10]．

（3） 自己書換えが行われることによって，CPU の条件分岐予測の失敗が増加する．

実行時間の増大に関しては，デメリットにならない場合と，そうでない場合がある．例えば，将棋やチェスといったゲームの手を考えるアルゴリズムや，音声のストリーミング再生ルーチンといったリアルタイム性を求められる個所に過度のカムフラージュは推奨されない．

一方，パスワード認証により制限しているようなプログラムに対して，パスワードのチェックルーチンの解析を困難にするために，提案方法を適用したい場合を考える．このような場合，パスワードチェックの部分にのみ適用すれば，パスワードのチェック時に時間がかかるようになるだけで，本来の機能の使い勝手が悪くなることはない．そのため，実行時間のオーバヘッドはほとんどデメリットにはならないといえるだろう．したがって，カムフラージュする個所とカムフラージュの度合は，適用対象となるプログラムやモジュールの性質・用途に応じて決定することが望ましい．

5. 関 連 研 究

自己書換えの仕組みそのものは，従来より用いられている．その目的の一つは，プログラムサイズや実行時のメモリ使用量を節約することである [8]．もう一つは，本論文と同様，プログラムを保護することが目的であり，自己書換えによってプログラムの暗号化，及び，復号を行う [4], [7], [12]．後者の方法では，プログラム中の特定の範囲をあらかじめ暗号化しておき，自己書換えによって実行時に復号し，必要に応じて再度暗号化を行う．実行時にプログラムの書換えを行うという点で提案方法と類似するが，次の点が異なる．

- カムフラージュされた命令は，他の命令との区別がつかないため，静的解析によってその位置を特定することは容易でない．一方，プログラム中の暗号化された部分は，他の部分と異なる特徴をもつ（命令の系列が見られない，逆アセンブルできない等）ため，静的解析によってその範囲を容易に特定されるおそれがある．
- カムフラージュを解くための復帰ルーチンは，メインメモリ上の 1 バイトないし数バイトの書換えを行うごくありふれた短いルーチンであり，かつ，プログラム中に分散して存在するため，静的解析によってその位置を特定することは容易でない．一方，復号のためのルーチンは，より大がかりとなるため，その特定のための手掛りをなくすことは必ずしも容易でない．特に，プログラム全体が暗号化されている場合には，プログラムの実行開始直後に復号が行われるため，復号ルーチンの特定は容易である．

「あらかじめ別の内容で上書きした命令を，実行時に元来の命令で置き換える」という動作をソフトウェアの保護に利用する考えは従来より存在するが，アセンブラに関する十分な知識，技術，及び，リソースをもった作業者が，手動で保護の処理を行う必要があった．本論文では，自己書換えを用いてソフトウェアを保護するための系統的な（定式化された）手法を提案し，評価を行った．提案手法を用いることで，保護の処理を自動化でき，保護の強さとオーバヘッドとの間のトレードオフのバランス（カムフラージュの度合）を自由に調整することも可能となる．

プログラムの解析を困難にするための他の方法として，プログラムを難読化する方法が数多く提案されている [5], [9], [19]〜[21]．難読化とは，与えられたプログラムを，その仕様を変えずに，より解析の困難な（複雑な）プログラムへと変換する技術である．ただし， 難読化されたプログラムは，たとえ部分的であっても，時間をかければその部分の動作を正しく理解することが可能である．一方，提案方法によって保護されたプログラムを部分的に理解しようと試みても，そこに元来のものと内容が異なる命令が含まれていれば，（復帰ルーチンを特定しない限り）時間をかけても正しく理解することは困難である．この点が，難読化と提案方法の相違点である．

なお，提案方法は，暗号化や難読化と対立する技術ではないため，併用することで，プログラムの解析をより困難にできる．例えば，難読化を併用すると，（1）カムフラージュされていない状態を得る，（2）難読化されたプログラムを理解する，という二つの段階を経なければ解析に成功できないプログラムを得ることができる．ただし，併用によってプログラムサイズや実行時間が更に増大する場合があるため，注意が必要である．

6. む す び

本論文では，命令のカムフラージュを用いてプログ

ラム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命令が含まれたプログラム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．

カムフラージュされたプログラムの解析の困難さについて，攻撃者がプログラムの秘匿部分を正しく理解できる確率を定式化した．導かれた式から，カムフラージュされたプログラムを正しく理解するには，解析する範囲を広げる必要があるという結論を得た．

ケーススタディにおいては，あるプログラム (ccrypt) にカムフラージュを施し，ターゲット命令と復帰ルーチン間の距離，及び，ファイルサイズや実行時間のオーバヘッドを測定した．947 の命令のうち 130 個所をカムフラージュした場合，ターゲット命令と復帰ルーチン間の距離の平均は 151 命令であった．ターゲット命令と復帰ルーチン間にもカムフラージュされた命令が多数現れるため，復帰ルーチンの発見には大きな解析コストを要すると予想される．また，オーバヘッドに関しては，カムフラージュされる命令を増やすほど，ファイルサイズや実行時間のオーバヘッドが高くなることが分かった．カムフラージュする個所，及び，カムフラージュの度合は，適用対象となるプログラムやモジュールの性質・用途に応じて決定することが望ましい．

最後に，今後の課題について述べる．まず，ターゲット命令と復帰ルーチンが一定の距離以上離れるように，自己書換えルーチンの挿入位置を決定するアルゴリズムを改良することを考えている．また，実行時間のオーバヘッドを軽減するために，CPU のパイプライン機能や分岐予測機能を考慮した自己書換えを行えるよう，システムを改良する予定である．

文　献

[1] P. Cerven, Crackproof Your Software, No Starch Press, San Francisco, 2002.

[2] H. Chang and M. Atallah, “Protecting software codes by guards,” Proc. Workshop on Security and Privacy in Digital Rights Management 2001, LNCS, vol.2320, pp.160–175, Springer-Verlag, 2001.

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

[4] F.B. Cohen, “Operating system protection through program evolution,” Comput. Secur., vol.12, no.6, pp.565–584, 1993.

[5] C. Collberg and C. Thomborson, “Watermarking, tamper-proofing, and obfuscation — tools for software protection,” IEEE Trans. Softw. Eng., vol.28, no.8, pp.735–746, June 2002.

[6] 舩本昇竜，プロテクト技術解剖学，すばる舎，東京，2002.

[7] D. Grover (Ed.), The Protection of Computer Software: Its Technology and Applications, Cambridge University Press, 1989.

[8] 日高　徹，Z80 マシン語秘伝の書，啓学出版，東京，1989.

[9] F. Hohl, “Time limited blackbox security: Protecting mobile agents from malicious hosts,” in Mobile Agents Security, ed. G. Vigna, LNCS, vol.1419, pp.92–113, Springer-Verlag, 1998.

[10] インテル株式会社，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・マニュアル下巻：システム・プログラミング・ガイド，第 9 章 p.18, http://www.intel.co.jp/

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

[12] 石間宏之，齋藤和雄，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.

[13] 神崎雄一郎，門田暁人，中村匡秀，松本健一，“命令コードの実行時置き換えを用いたプログラムの解析防止,” 信学技報，ISEC2002-98, Dec. 2002.

[14] 神崎雄一郎，プログラムカムフラージュ化ツール，http://se.aist-nara.ac.jp/rinrun/

[15] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto, “Exploiting self-modification mechanism for program protection,” Proc. 27th IEEE Computer Software and Applications Conference, pp.170–179, Dallas, USA, Nov. 2003.

[16] J.R. Levine, 榊原一矢（監訳），ポジティブエッジ（訳），Linkers & Loaders, pp.75–83, オーム社，東京，2001.

[17] C. Linn and S. Debray, “Obfuscation of executable code to improve resistance to static disassembly,” Proc. 10th ACM Conference on Computer and Communications Security, pp.290–299, Oct. 2003.

[18] A. Monden, A. Monsifrot, and C. Thomborson, “Obfuscated instructions for software protection,” Information Science Technical Report, NAIST-IS-TR2003013, Graduate School of Information Science, Nara Institute of Science and Technology, Nov. 2003.

[19] 門田暁人，高田義広，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,” 信学論 (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,” 信学技報，ISEC95-25, Nov. 1995.

[21] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, “Software obfuscation on a theoretical basis and its implementation,” IEICE Trans. Fundamentals, vol.E86-A, no.1, pp.176–186, Jan. 2003.

[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] The United Kingdom Parliament, “The mobile tele-

phones (re-programming) bill," House of Commons Library Research Paper no.02/47, July 2002.
[24] 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.

付　　録

条件分岐を一つ含む簡単なプログラムと，そのカムフラージュ化プログラムのリストを示す．

1. オリジナルプログラム（C 言語）

```
#include <stdio.h>
#define PASSNUM 13

int main() {
    int n;
    scanf("%d," &n);
    if(n!=PASSNUM) {
  printf("INVALID\n");
  return -1;
    }
    printf("OK\n");
    return 0;
}
```

2. オリジナルプログラム（アセンブリ）

```
LC0:
  .ascii "%d\0"
LC1:
  .ascii "INVALID\12\0"
LC2:
  .ascii "OK\12\0"
  .align 2
.globl _main
_main:
  pushl  %ebp
  movl  %esp, %ebp
  subl  $24, %esp
  andl  $-16, %esp
  movl  $0, %eax
  movl  %eax, -12(%ebp)
  movl  -12(%ebp), %eax
  call  __alloca
  call  ___main
  movl  $LC0, (%esp)
  leal  -4(%ebp), %eax
  movl  %eax, 4(%esp)
  call  _scanf
  cmpl  $13, -4(%ebp)
  je  L10
  movl  $LC1, (%esp)
  call  _printf
  movl  $-1, -8(%ebp)
  jmp  L9
L10:
  movl  $LC2, (%esp)
  call  _printf
  movl  $0, -8(%ebp)
L9:
  movl  -8(%ebp), %eax
  leave
  ret
```

3. カムフラージュ化プログラム

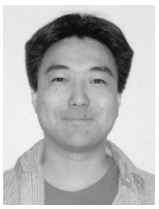
```
LC0:
  .ascii "%d\0"
LC1:
  .ascii "INVALID\12\0"
LC2:
  .ascii "OK\12\0"
  .align 2
.globl _main
_main:
  movl  $T2 + 0x824, %eax      # RR2
  subl  $0x824, %eax           # RR2
  movb  $0xeb, (%eax)          # RR2
  pushl  %ebp
  subb  $0x3d, T3 + 2          # RR3
  movl  %esp, %ebp
  subl  $24, %esp
  andl  $-16, %esp
  movl  $T1 - 20 + 3, %eax     # RR1
  addl  $20, %eax              # RR1
T3:
  movb  $0x4a, (%eax)          # RR1 target3
  movl  $0, %eax
  movl  %eax, -12(%ebp)
  movl  -12(%ebp), %eax
  call  __alloca
  call  ___main
  movl  $LC0, (%esp)
  leal  -4(%ebp), %eax
```

```
  movl  %eax, 4(%esp)
  movl  $T3 - 0x08 + 2, %eax    # HR3
  addl  $0x08, %eax             # HR3
  movb  $0x4a, (%eax)           # HR3
  call  _scanf
T1:
  cmpl  $7, -4(%ebp)            # target1
  je  L10
  movl  $LC1, (%esp)
  call  _printf
  movl  $-1, -8(%ebp)
T2:
  je  L9                        # target2
L10:
  movl  $LC2, (%esp)
  call  _printf
  movl  $0, -8(%ebp)
  movb  $0x74, T2               # HR2
L9:
  movl  -8(%ebp), %eax
  movl  $T1 + 0x120 + 3, %eax   # HR1
  subl  $0x120, %eax            # HR1
  movb  $0x07, (%eax)           # HR1
  leave
  ret
```

（平成 15 年 9 月 22 日受付，16 年 1 月 18 日再受付，
2 月 23 日最終原稿受付）

神﨑雄一郎　（学生員）

平 13 神戸大・工・情報知能卒．平 15 奈良先端科学技術大学院大学博士前期課程了．現在，同大博士後期課程に在学中．ソフトウェアプロテクションの研究に従事．IEEE 学生会員．

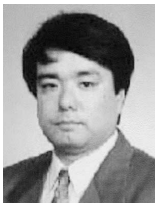


門田　暁人　（正員）

平 6 名大・工・電気卒．平 10 奈良先端科学技術大学院大学博士後期課程了．同年同大・情報科学・助手．平 15〜16 ニュージーランド・オークランド大学客員研究員．博士（工学）．ソフトウェアプロテクション，ソフトウェアメトリックス，ヒューマンインタフェース等の研究に従事．情報処理学会，日本ソフトウェア科学会，IEEE，ACM 各会員．

中村　匡秀　（正員）

平 6 阪大・基礎工・情報卒．平 11 同大大学院博士後期課程了．平 12 阪大・サイバーメディアセンター・助手．平 14 奈良先端科学技術大学院大学・情報科学・助手．博士（工学）．通信ソフトウェア，サービス競合，ソフトウェアプロテクション等の研究に従事．IEEE 会員．



松本　健一　（正員）

昭 60 阪大・基礎工・情報卒．平元同大大学院博士課程中退．同年同大・基礎工・情報・助手．平 5 奈良先端科学技術大学院大学・情報科学・助教授．平 13 同大・情報科学・教授．工博．ソフトウェア品質保証，ユーザインタフェース，ソフトウェアプロセス等の研究に従事．情報処理学会，IEEE，ACM 各会員．