NAIST-IS-MT0151036

修士論文

自己書き換えを用いた プログラムの解析防止方法の提案

神崎 雄一郎

2003 年 2 月 7 日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
修士 (工学) 授与の要件として提出した修士論文である。

神崎 雄一郎

審査委員：　松本 健一 教授
渡邉 勝正 教授
飯田 元 助教授

自己書き換えを用いた プログラムの解析防止方法の提案*

神崎 雄一郎

内容梗概

本論文では，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，実行時に内容が変化する命令コードを多数含んでいる．これらの命令コードは，実行時のある期間だけ，オリジナルの命令コードとなるが，それ以外の期間においては，オリジナルのものと異なった，ダミーの命令を装っている．不正行為者がこのようなプログラムの解析を試みるとき，ダミーの命令を装っている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．以上のようなアイデアに基づく提案方式を用いると，特別なハードウェアを必要とせずに，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

情報セキュリティ, 知的財産権, ソフトウェア保護, プログラムの難読化, 自己書き換え

* 奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 修士論文, NAIST-IS-MT0151036, 2003年 2月 7日.

Protecting Software by Replacing Instructions at Run-time*

Yuichiro Kanzaki

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.

Keywords:

information security, intellectual property,software protection, program obfuscation, self-modifying code

* Master's Thesis, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT0151036, February 7, 2003.

目 次

図 目 次

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]．このシステムは，サーバから送られてきた暗号化されたコンテンツのデータを，パソコン上のアプリケーションで秘密鍵を用いて復号するようになっているが，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．

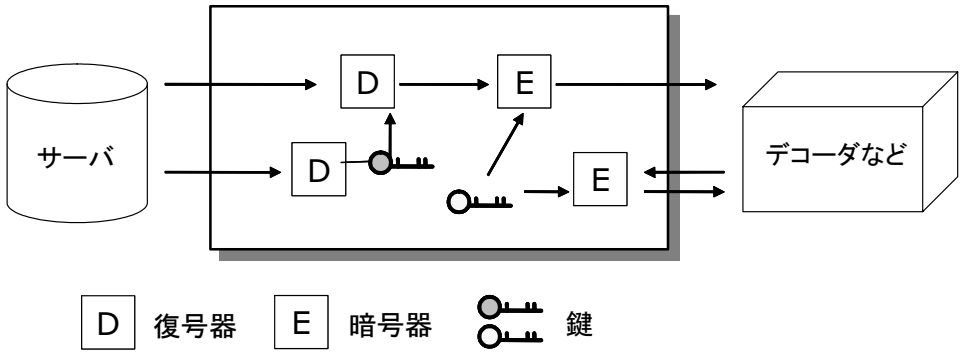

図 1　SDMI 準拠のコンテンツ流通システムの一部

プログラムの解析を困難にする技術は，2 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．そこで，本論文では，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．

本論文では，まず，2 章において，従来技術とその問題点について述べる．その後，3 章において，提案方式である自己書き換えを用いたプログラムの解析防止方法の詳細について述べ，続く 4 章で提案方式を適用する具体例を述べる．また，5 章では，保護対象となるプログラムが，プログラムサイズやプログラムの実行時間に関してどの程度の影響を受けるかということについての実験を行い，得られた結果から，保護のトレードオフとして受けるプログラムの実行時間のロスを軽減するための対策や，提案方式の改善に向けての具体的な課題について考察する．最後に 6 章において，まとめと今後の課題について述べる．

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 (program obfuscation) 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 [11] [19] [20],「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] [21]，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 [6] [7]，データ構造を変形する方式 [8]，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 [22] [25] などがある．

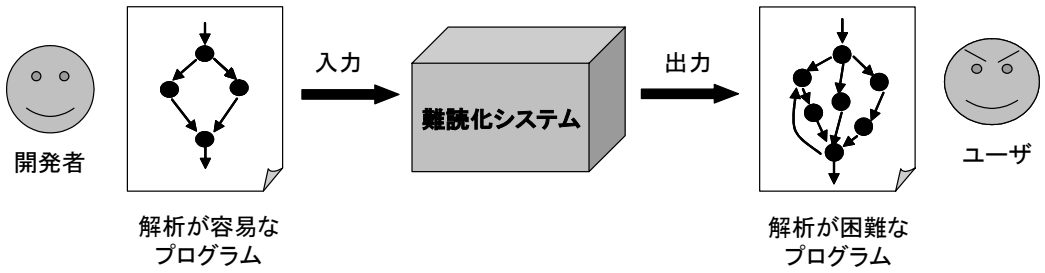

図 2 プログラムの難読化

プログラムの暗号化 (program encryption) 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．したがって，暗号化された命令コードを復号するための機構 (図 3 では「復号化モジュール」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] [9] [12] [24] などにおいて提案されている．

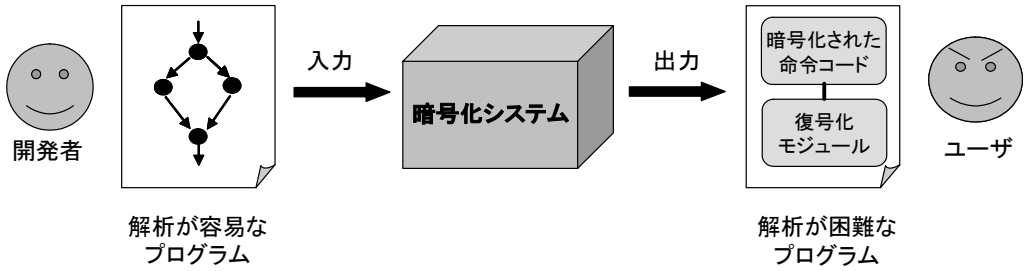

図 3 プログラムの暗号化

プログラムの断片化 (Program Fragmentation) 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] [13] [15] [16] などにおいて提案されている．

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

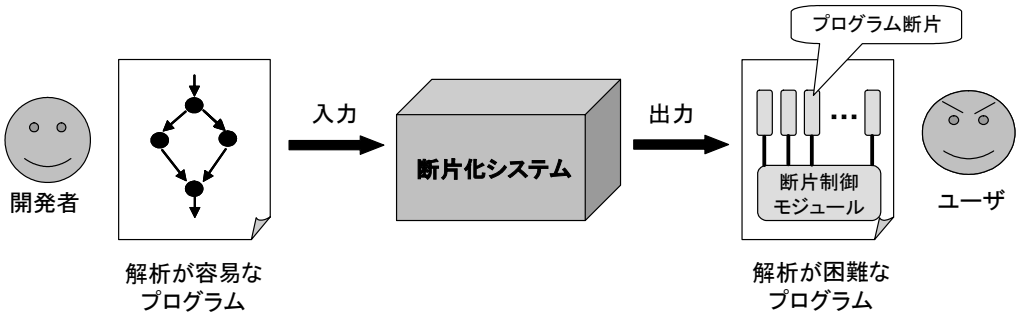

図 4 プログラムの断片化

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. 自己書き換えを用いたプログラムの解析防止方法

3.1 キーアイデア

提案方式のキーアイデアは，自己書き換えの機能をプログラムに追加することで，解析を困難にすることである．ここでいう自己書き換えとは,「実行時に，プログラム中の一命令が，同じプログラムに含まれるある命令を異なった命令に書き換える」という動作のことを指す.

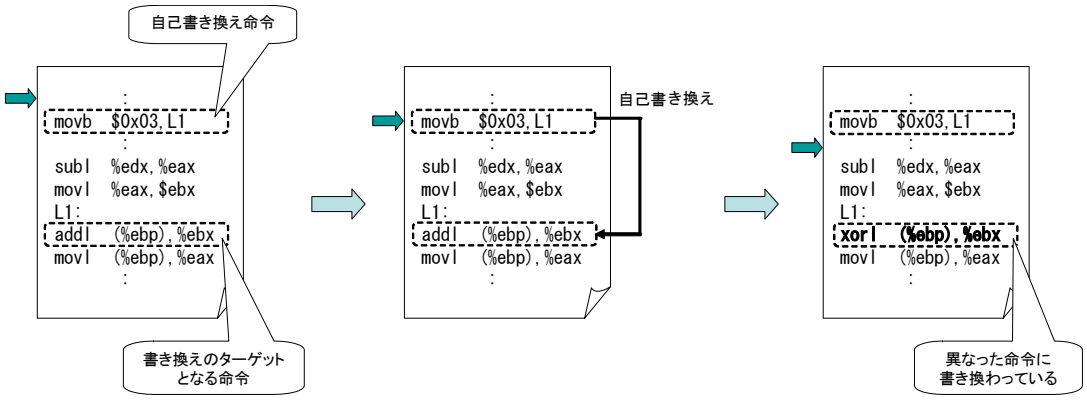

図 5 自己書き換えプログラムの実行の様子

図 5 は，自己書き換えプログラムの実行の様子を示したものである (ここでは，実行される機械語プログラムを，プログラムの中身が分かりやすいようにアセンブリ命令であらわしている)．この例では，自己書き換え命令 (“movb $0x03,L1”) に実行の制御が達する前の段階においては “addl ($ebp),%ebx” となっている命令コードが，自己書き換え命令の実行によって，異なった命令 “xorl ($ebp),%ebx” に書き換えられている．このような，命令コードを実行中に動的に変化させる動作を用いて，プログラムの解析を困難にすることを考える．具体的には，プログラムの多数の箇所をダミー (偽(にせ)) の命令でカムフラージュし，それらが実行時のある期間だけ正しい命令に書き換わるような自己書き換えプログラムを構成することを考える.

図 6 は，提案する方式によって保護の処理が行われた後のプログラムの模式図である．プログラムの多数の命令コードが，カムフラージュのターゲットの命令

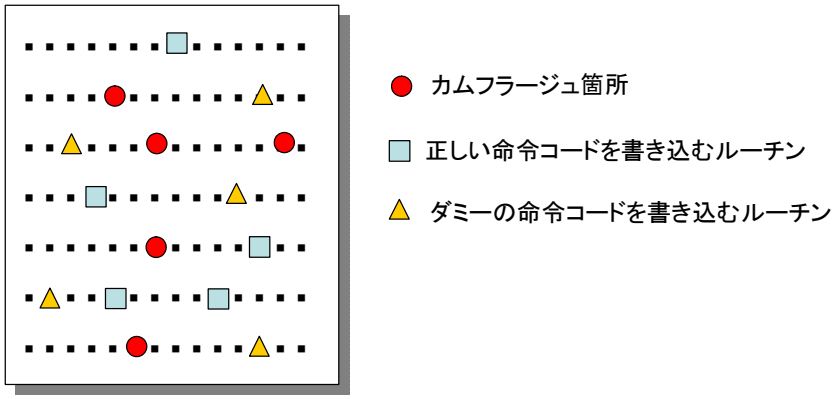

図 6 提案方式により保護されたプログラムの模式図

として選ばれ，実行前からあらかじめダミーの命令コードで上書きされている．また，カムフラージュされている命令と同じ数だけ,「カムフラージュ箇所に正しい (オリジナルの) 命令コードを書き込む処理を行うルーチン」と，実行時に再びカムフラージュするための,「ダミーの命令コードを書き込む処理を行うルーチン」が挿入されている．各々のカムフラージュ箇所は，正しい命令コードを書き込むルーチンが実行されてから，ダミーの命令コードを書き込むルーチンが実行されるまでの間のみ，正しい命令となり，その期間に実行される．ソフトウェアを，このような自己書き換えプログラムに変換して配布することで，不正な解析を困難にすることが可能である．

3.2 提案方式に基づくシステムについて

図 7 は，提案方式に基づくシステムを利用してソフトウェアを保護する流れを示したものである．システムの利用者は，まず，保護の対象となるアセンブリプログラムを用意する．アセンブリプログラムは通常，C 言語などの高水準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得られる．バイナリプログラムを逆アセンブルすることによってアセンブリプログラムを得る方法も考えられるが，それを再びアセンブルしてバイナリプログラムに戻すことは，現在普及している計算機環境では難しいことが多い．

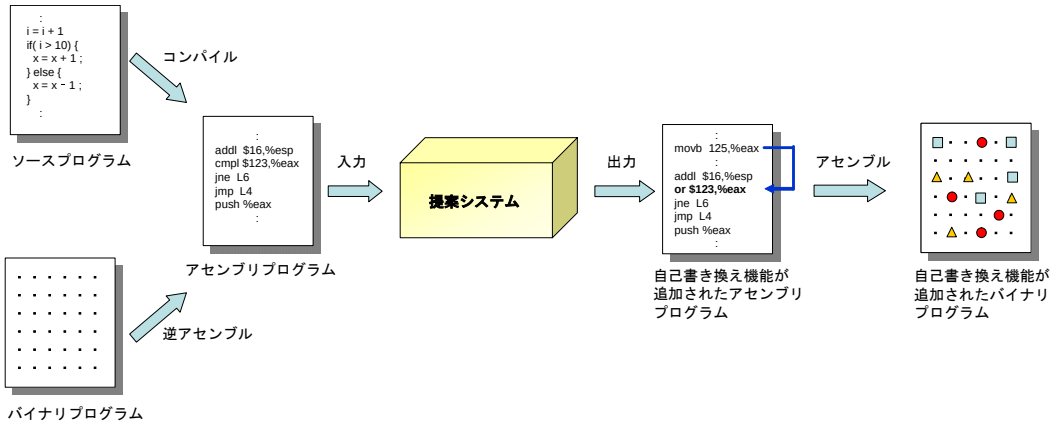

図 7 提案システムを利用してソフトウェアを保護する流れ

保護の対象となるアセンブリプログラムを提案システムに入力すると，自己書き換え機能が追加されたアセンブリプログラムが出力される．自己書き換え機能が追加されたアセンブリプログラムをアセンブルすることにより，自己書き換え機能が追加された，解析が困難なバイナリプログラムを得ることができる．この後，他のバイナリプログラムとリンクするなどの過程を経て，実行可能ファイルを作成する．なお，自己書き換えを行うために，実行時におけるコード領域へのデータの書き込みを許可するように実行可能ファイルの属性を変更する必要がある．

3.3 自己書き換えプログラムの構成手順

ここでは，自己書き換え機能プログラムを構成するために提案システムが行う処理の内容について述べる．提案システムでは，次に示す (ステップ 1) から (ステップ 6) を順に実施することによって，入力されたアセンブリプログラムに自己書き換え機能を追加する．

(ステップ 1) カムフラージュのターゲットとなる命令の選択と自己書き換えルーチンの挿入位置の決定

まず，対象となるアセンブリプログラムに対して，I_t，R_u，R_c のプログラム上の位置を決定する．ここで，I_t，R_u，R_c はそれぞれ次のような意味を持つ．

I_t: カムフラージュのターゲットとなる命令コード

R_u: 実行時にオリジナルの (正しい) 命令を書き込むルーチン

R_c: 実行時にダミーの命令を書き込むルーチン

また，I_t，R_u，R_c のプログラム上の位置を，それぞれ $P(I_t)$，$P(R_u)$，$P(R_c)$ と定義する．このとき，次に示す 3 条件をすべて満たすように，$P(I_t)$，$P(R_u)$，$P(R_c)$ を決定する．

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

(条件 2) $P(R_u)$ から $P(I_t)$ へ至る全ての制御フロー上に $P(R_c)$ が存在しない．

(条件 3) $P(R_c)$ から $P(I_t)$ へ至る全ての制御フロー上に $P(R_u)$ が存在する．

これらの条件は，ダミーの命令でカムフラージュされている命令コードがカムフラージュされたままの状態で実行されることによって，プログラムの誤動作が起きることを防ぐためのものである．これらの条件が満たされるように $P(I_t)$，$P(R_u)$，$P(R_c)$ を決定する様子を，図 8 に例示する．多くの場合，上の条件を満たす候補は，複数存在する．その場合，ランダムに 1 つの候補を選択する．

(ステップ 2) ダミーの命令コードの生成

オリジナルの命令コード，すなわち，適用前のプログラムの $P(I_t)$ に存在する命令コードの一部分を別の内容に変えることによって，ダミーの命令コードを生成する．図 9 は，ダミーの命令コードを生成する例を示したものである．この例では，

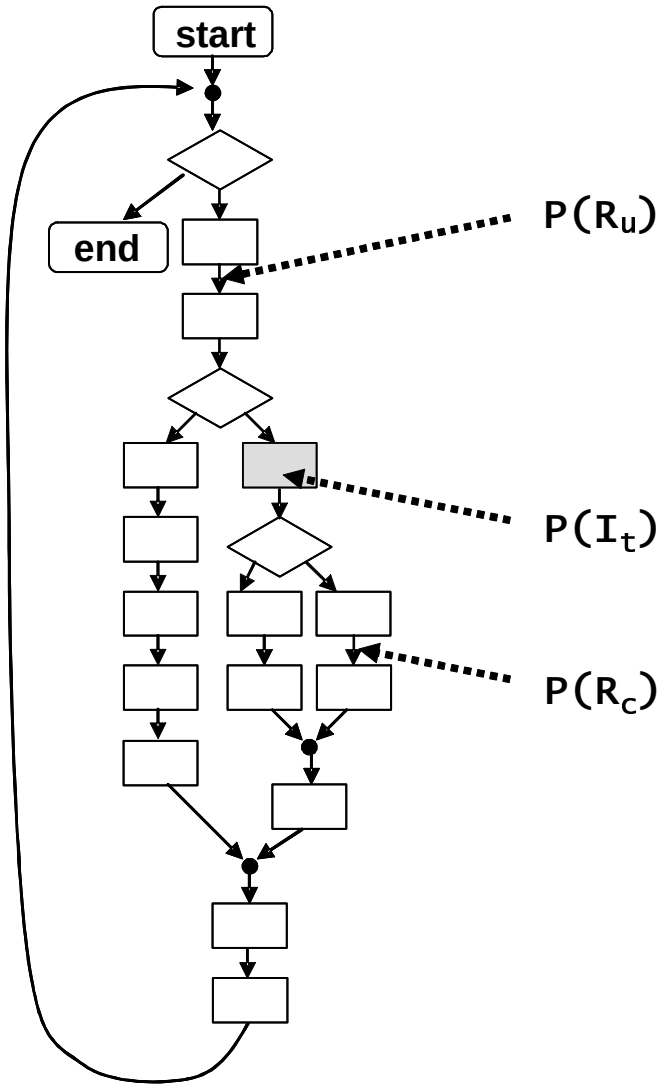

図 8　カムフラージュのターゲットとなる命令の選択と自己書き換えルーチンの挿入位置の決定の例

アセンブリ表現で “addl -12(%ebp), %ebx”，機械語表現 (16 進数) で “03 5D F4” として表されるオリジナルの命令コードの 1 バイト目を “03” から “33” に変更することで，“xorl -12(%ebp), %ebx” というアセンブリ表現で表される命令コードを生成し，その命令をダミーの命令コードとしている．

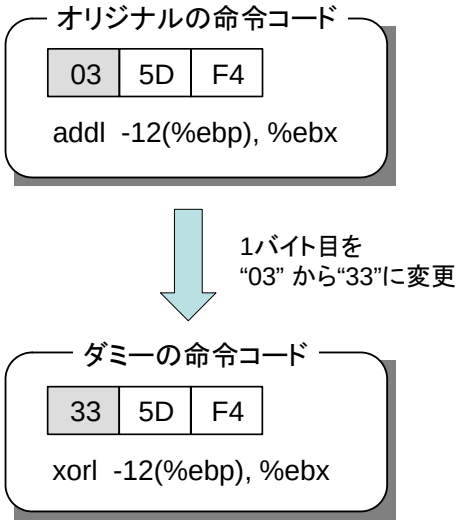

図 9 ダミーの命令コードの生成の例

ダミーの命令コード生成するために,「オリジナル命令のを一部分を変える」という方法をとっているのは，実行時にオリジナル命令コードをダミーの命令に書き換えるルーチンおよびその逆を行うルーチンの記述が単純となり，結果，それらのサイズを小さくとどめることが可能であるからである．これらのルーチンのサイズを小さくすることができると，解析者にとってルーチンの存在が特定しにくくなり，防止効果の向上につながる．また，プログラムサイズの増加を軽減できるという点でも有利となる．

(ステップ 3) 自己書き換えルーチンの生成

オリジナルの命令コードを実行時に $P(I_t)$ へ書き込む処理を行うルーチン R_u と，ダミーの命令コードを実行時に $P(I_t)$ へ書き込む処理を行うルーチン R_c を生成する．オリジナルの命令コードおよびダミーの命令コードが，(ステップ 2) で挙げ

た例と同じであるとすると，R_u は，“xorl -12(%ebp), %ebx” を “addl -12(%ebp), %ebx” に変換するルーチン，すなわち，命令の 1 バイト目を “33” から “03” に変更するルーチンとなる (図 10 参照)．一方，R_c は，“addl -12(%ebp), %ebx” を “xorl -12(%ebp), %ebx” に変換するルーチン，すなわち，命令の 1 バイト目を “03” から “33” に変更するルーチンとなる．

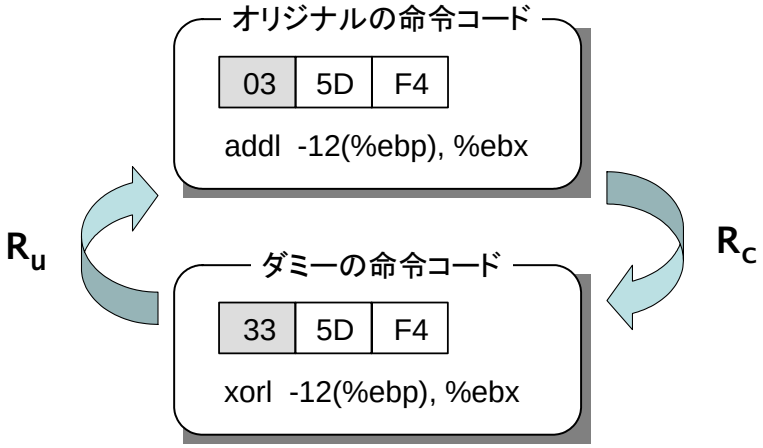

図 10 自己書き換えルーチンの生成の例

(ステップ 4) ダミーの命令コードの書き込みと自己書き換えルーチンの挿入

(ステップ 1) で決定した $P(I_t)$ に (ステップ 2) で生成したダミーの命令コードを書き込み，また，(ステップ 1) で決定した $P(R_u)$, $P(R_c)$ のそれぞれに対して，(ステップ 3) で生成した R_u，R_c を挿入する．

(ステップ 5) 自己書き換えルーチンの複雑化

R_u および R_c が，人間にとって読みにくい (複雑な) 表現を持つと，保護機構の無効化がより困難になる．そのために，ここで R_u および R_c を複雑化することが望ましい．

(ステップ 6) 前記ステップの繰り返し

(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 I_t，R_u，R_c，$P(I_t)$，$P(R_u)$，$P(R_c)$ は，それまでの繰り返しにおけるものと異なっていてよい．繰り返す回数については，提案方式の利用者が決定できる．

3.4 提案方式の特長

提案方式は，以下に示すような特長を持つ．

- プログラムの解析を著しく困難にすることができる．

  解析者は，ダミーの命令でカムフラージュされた命令コードを含む部分を読む限り誤った理解をすることになるため，解析に成功しない．プログラム中に，誤った理解を誘うダミーの命令コードを数多く設けることで，プログラムの解析を著しく困難にすることができる．

- 攻撃者による保護機構の無効化が容易でない．

  提案方式によって保護の処理が加えられたプログラムは，サイズの小さな自己書き換えルーチンが，多数，プログラム中に分散されている．それらをすべて探し出して無効化するのは，非常に長い時間と大きな労力が必要となる．

- 難読化，暗号化，断片化などの既存技術と併用できる．

  提案方式は，2 で述べた難読化，暗号化，断片化などの既存技術と対立するものではないため，ひとつのプログラムに対して，これらの手法をいくつか組み合わせたものを適用することが可能で

ある．既存技術を併用することで，不正な解析や改ざんの防止効果をより高めることができると考えられる．

- 特別なハードウェアを必要とせず，システム化が容易に行える．

  多くの計算機環境において，特別なハードウェアを追加せずに実施が可能なため，システム化が容易に行える．

4. 提案方式の適用例

この章では，提案方式の具体的な適用例を示す．適用の対象となるアセンブリプログラムとして，図 11 に示すようなプログラムを考える．[1]

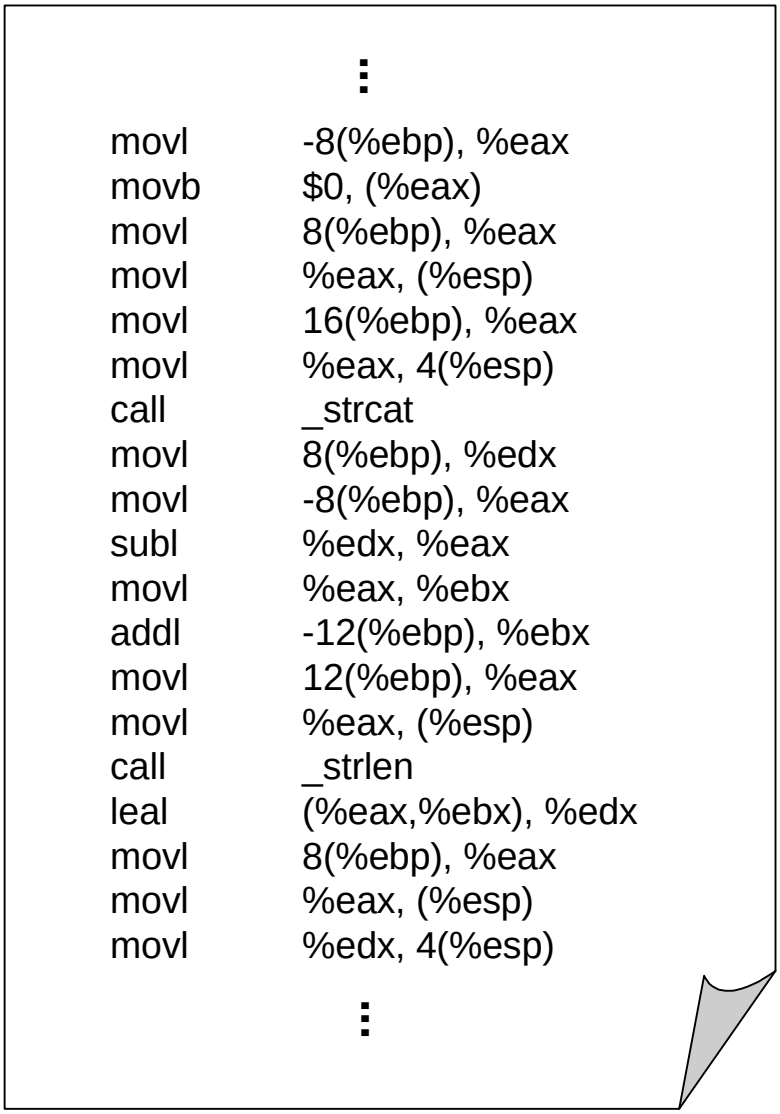

図 11　適用例におけるオリジナルのアセンブリプログラム

[1] 本論文では，すべてのアセンブリプログラムを i386 系のプロセッサのアセンブラコードを用いて表す．

このプログラムに対して自己書き換え機能を追加する例を，3.3 節において述べた提案システムの処理手順に沿って述べる．

(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，$P(I_t)$，$P(R_u)$ および $P(R_c)$，すなわち，カムフラージュのターゲットとなる命令コードの位置，オリジナルの命令コードを $P(I_t)$ に書き込む処理を行うルーチン R_u の挿入位置およびダミーの命令コードを $P(I_t)$ に書き込む処理を行うルーチン R_c の挿入位置を決定する．

これらを決定した例を，図 12 に示す．この例では，プログラム開始点から $P(I_t)$ へ至るまでに必ず通る制御フロー上に $P(R_u)$ が存在し，$P(R_u)$ と $P(I_t)$ を結ぶ唯一の制御フロー上に $P(R_c)$ が存在しない．また，$P(R_c)$ から $P(I_t)$ へ至るまでに必ず通る制御フロー上に $P(R_u)$ が存在する．したがって，挿入された $P(R_u)$，$P(I_t)$，$P(R_c)$ は，3.3 節の (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) ダミーの命令コードを決定する．ここでのオリジナルの命令は “addl -12(%ebp), %ebx” であり，その機械語表現 (16 進数) は “03 5D F4” である．この命令の一部を変化させてできる命令として，1 バイト目の “03” を “33” に変更した命令，すなわち，“33 5D F4” という機械語表現を持つ “xorl 12(%ebp), %eax” をダミーの命令コードに決定した．

(ステップ 3) 正しい命令コードを $P(I_t)$ に書き込む処理を行うルーチン R_u と，ダミーの命令コードを $P(I_t)$ に書き込む処理を行うルーチン R_c を生成する．ここでは，R_u は “movb $0x03,L1”，$R_c$ は “movb $0x33,L1” という 1 命令からなる短いプログラムがそれぞれのルーチンとして生成された．

(ステップ 4) $P(R_u)$ に R_u を挿入し，$P(R_c)$ に R_c を挿入し，$P(I_t)$ にダミーの命令コードを書き込む．図 11 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 13 に示す．

(ステップ 5) 挿入された R_u および R_c を複雑化する．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

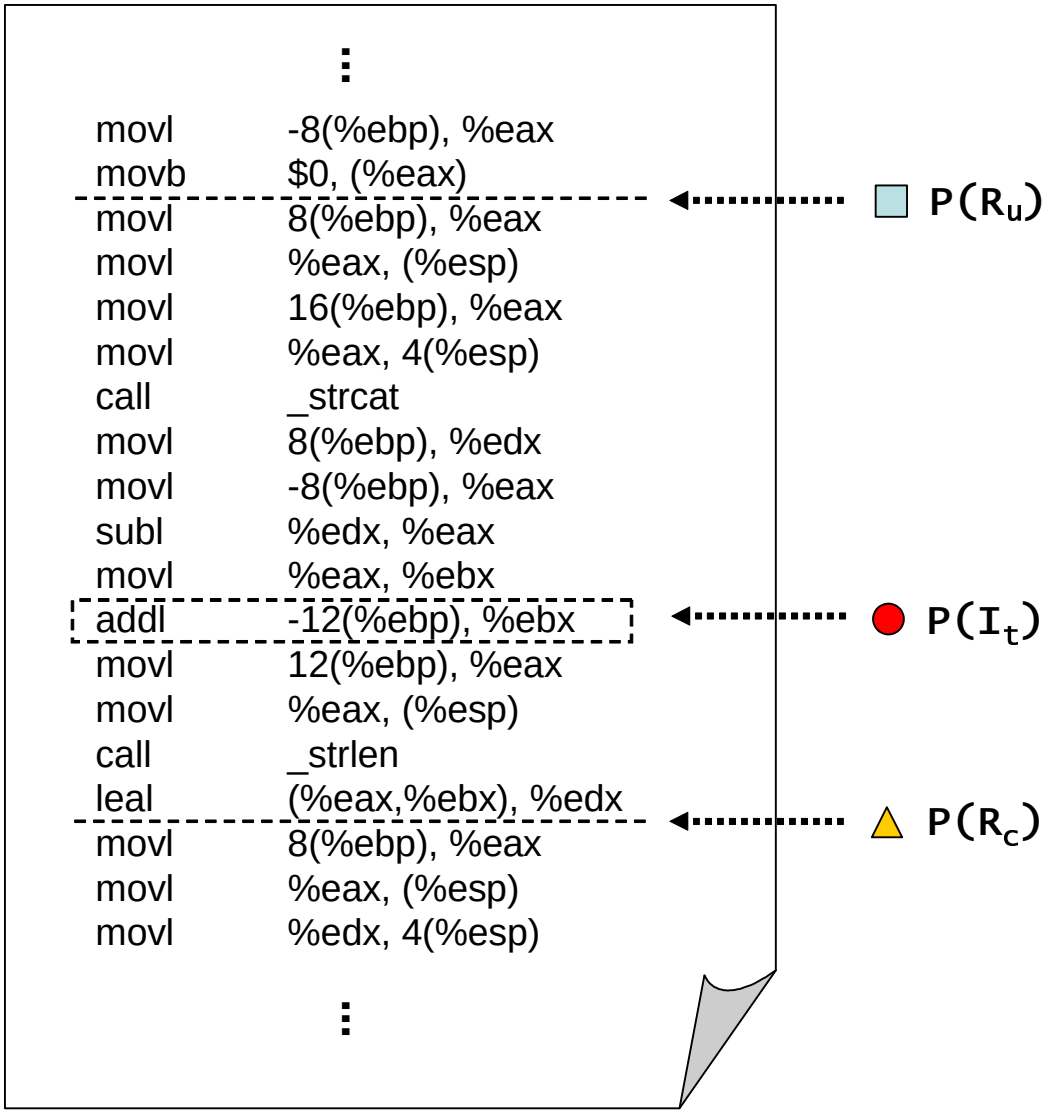

図 12　アセンブリプログラムに対する $P(I_t)$, $P(R_u)$, $P(R_c)$ の決定

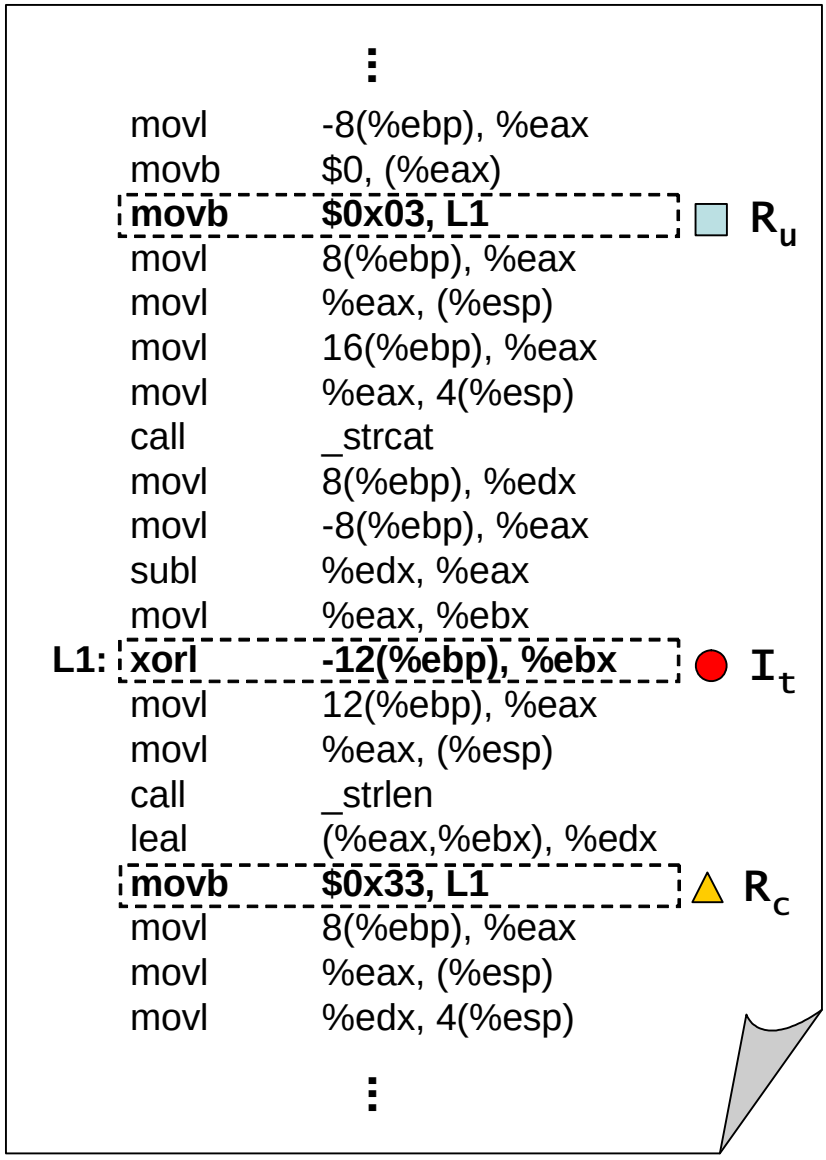

図 13　自己書き換え処理が追加されたアセンブリプログラム

$P(I_t)$ を R_u，R_c 両方のルーチンにおいて同じ方法で指定すると，R_u，R_c の位置，あるいは I_t の位置が発見されやすい」ということに注目して，R_u，R_c がそれぞれ別の方法で $P(I_t)$ を指定するように変更している．具体的には，$P(I_t)$ を直接的に示す同一のラベル (図 13 における`L1`) を用いて指定するのをやめ，それぞれ異なったラベル (`A1` あるいは `A2`) を用いて間接的に $P(I_t)$ を指定するように変更している．R_u および R_c に難読化処理を加えた後のアセンブリプログラムを図 14 に示す．

(ステップ 6) (ステップ 1) から (ステップ 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．

以上，提案方式の適用例について述べた．なお，文献 [14] および文献 [18] において，他の例を参照することができる．

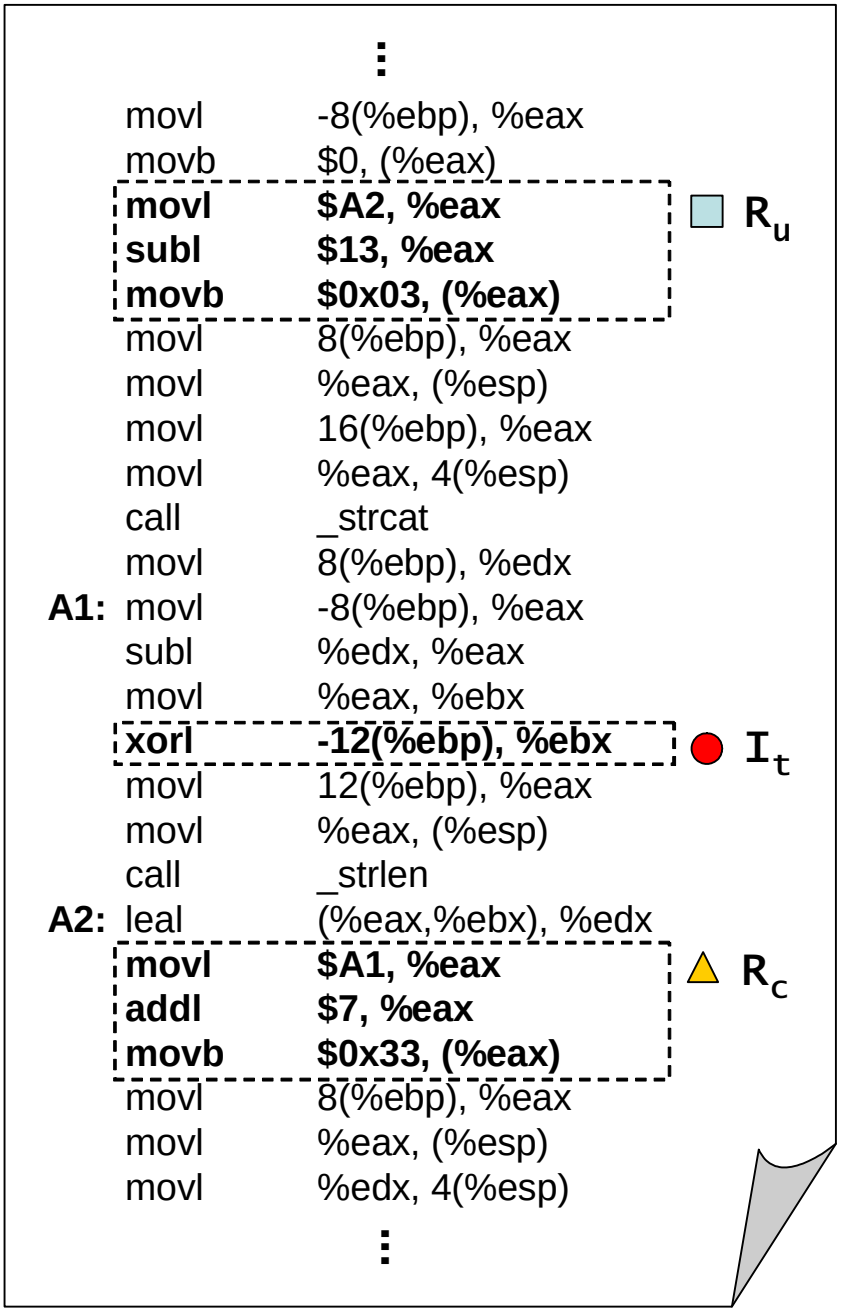

図 14 自己書き換え処理が追加されたアセンブリプログラム (Ru および Rc を複雑化した場合)

5. プログラムサイズおよびプログラムの実行時間への影響を評価するための実験

5.1 実験概要

この章では，提案方式を適用することによって，保護対象となるプログラムがプログラムサイズおよびプログラムの実行時間についてどの程度の影響を受けるかということについて検討する．具体的には，カムフラージュされた命令数に応じて，プログラムのサイズおよびプログラムの実行時間がどの程度変化するかを，実験を通して調べる．得られた結果から，保護のトレードオフとして受けるプログラムの実行時間のロスを軽減するための対策や，提案方式の改善に向けての具体的な課題について考察する．

実験の対象となるプログラムとして，ccrypt とgzip の 2 つを選んだ．ccrypt は任意のファイルを暗号化・復号化するためのツール[2] で，gzip はファイルの圧縮・解凍を行うためのツール[3] である．いずれのプログラムも，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．

なお，この実験は，CPU がIntel Pentium 4 1500MHz，OS がMicrosoft Windows XP であるマシン上で行った．

5.2 実験手順

以下に示す手順にしたがって実験を行った．

1. C 言語で書かれたソースプログラムをアセンブリプログラムに変換する．

2. 提案方式に基づいて実装したシステムを用いて，対象となるプログラム中の一定数の命令をカムフラージュし，それを実行時に書き換えるためのルーチンを挿入する．カムフラージュする命令の個数 (すなわち，追加する自己書き換えルーチンのセット数) は，200, 400, 600, 800, 1000 の 5 通りとする．

[2] http://quasar.mathstat.uottawa.ca/ selinger/

[3] http://www.gzip.org/

3. それぞれのプログラムに対してアセンブル・リンクなどを行い，(5つの) 実行可能ファイルを得る．

4. それぞれの実行可能ファイルのサイズと，一定のデータを処理するのに要する実行時間を測定する[4]．

5. 実行時間に関しては，システムによる自己書き換え機能の追加のされ方によって測定値が大きく変化することが考えられるので，10回測定を繰り返し，その平均値，最大値，最小値を求める．

5.3 実験結果と考察

5.3.1 プログラムサイズの変化について

図15は，ccryptのプログラムサイズの変化を，また，図16は，gzipのプログラムサイズの変化を示すグラフである．両グラフとも，横軸は，カムフラージュされた命令の数を表し，縦軸は，プログラムサイズ (折れ線グラフによって示される) およびカムフラージュされた命令数の比率 (棒グラフによって示される) を表す．ここで，カムフラージュされた命令数の比率というのは，カムフラージュされている命令数が，保護される前のプログラムの全体の命令数の何%に相当するかを示すものである．

どちらのグラフを見ても，カムフラージュされた命令数に比例して，プログラムサイズが増加していることがわかる．1000の命令がカムフラージュされたとき，プログラムの全体の約9%の命令がカムフラージュされている状態となるが，この場合のプログラムサイズの増加は約42KBとなっている．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，プログラムサイズが約4.2KB増加している．このような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，それを自己書き換えするための部分的なプログラムが挿入されるためである．近年，個人用のPCにおいてさえ二次記憶装置が大容量化する傾

[4] 具体的には，ccryptの場合，100KBのテキストファイルを暗号化するのに要した時間を，gzipの場合，1MBのテキストファイルを圧縮するのに要した時間を測定した．

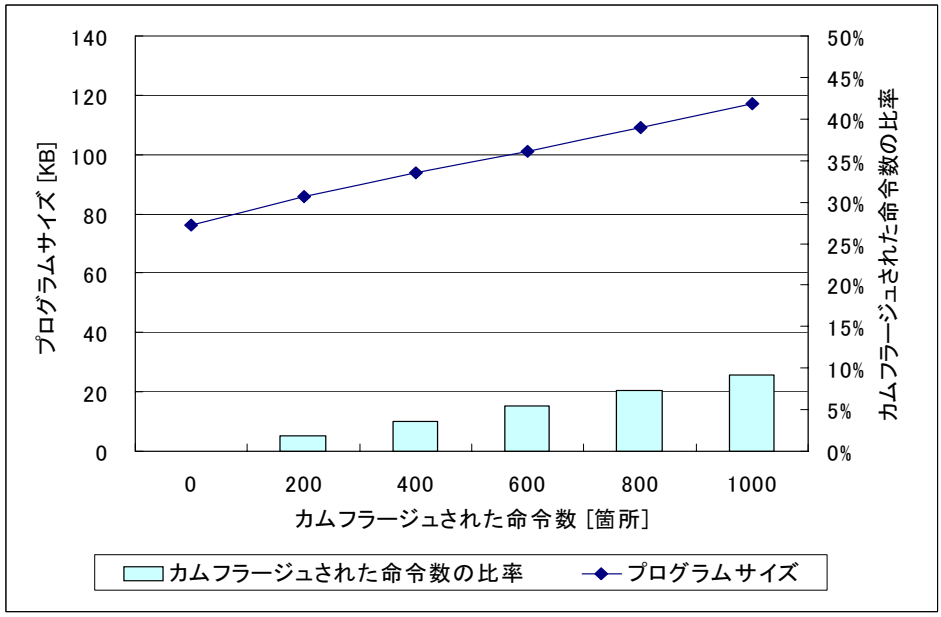

図 15　ccrypt のプログラムサイズの変化

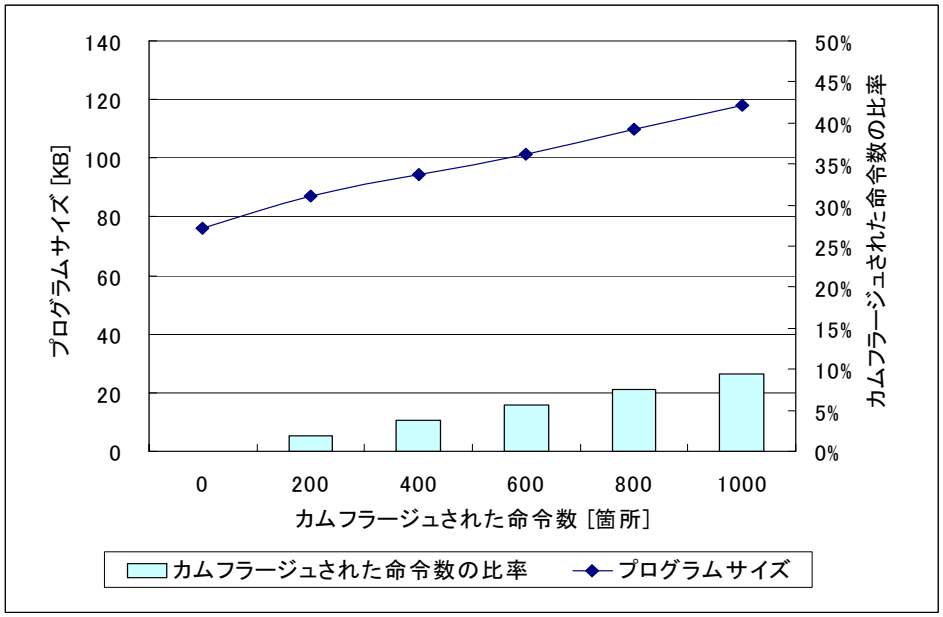

図 16　gzip のプログラムサイズの変化

向にあることを考慮に入れると，プログラムサイズが増加することは，特に大きな問題とはならないと考えられる．ただ，組み込みソフトウェアなど，プログラムサイズに関する制約が厳しい環境においては，プログラムサイズの増加を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，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．

5.3.2 プログラムの実行時間の変化について

図 17 は，ccrypt の実行時間の変化を，また，図 18 は，gzip の実行時間の変化を示すグラフである．両グラフとも，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 (平均値は実線で，最小値，最大値は点線で表される) およびカムフラージュされた命令数の比率を表す．

まず，両グラフの平均値を示す線に注目する．どちらのグラフを見ても，カムフラージュされた命令数が多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．1000 の命令がカムフラージュされたときの実行時間と，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を比較したとき，1000 の命令がカムフラージュされたときの方が，ccrypt に関しては 134 倍 (9.4/0.07 倍) 程度，gzip に関しては 16 倍 (4.1/0.25 倍) 程度の実行時間を要するという結果になっている．このオーバーヘッドは，挿入された自己書き換えルーチンによるものと考えられる．実行時間の増加は，プログラムが保護されることとのトレードオフとして考えると，ある程度までは許容されるべきではあるが，実行時間が大きく上昇することを避けたい場合が多いという推測は否定できない．したがって，より実用的なシステムにするためには，実行時間のロスを軽減する必要があると考えられる．自己書き換えルーチンの挿入の量を減らすためには，提案システムの利用者が持つ，保護の対象となるプログラムについての知識を利用するという方法が考えられる．具体的には，保護したいという部分がプログラムのどこに記述されているかという知識を利用して，その部分のみ (たとえば保護したい部分が記述されているファイルに対してのみ) 提案方式を適用するようにすれば，自己書き換えルーチンの挿入の数を抑えることができ，実行時間のロスをある程度軽減することが可能であると考えられる．

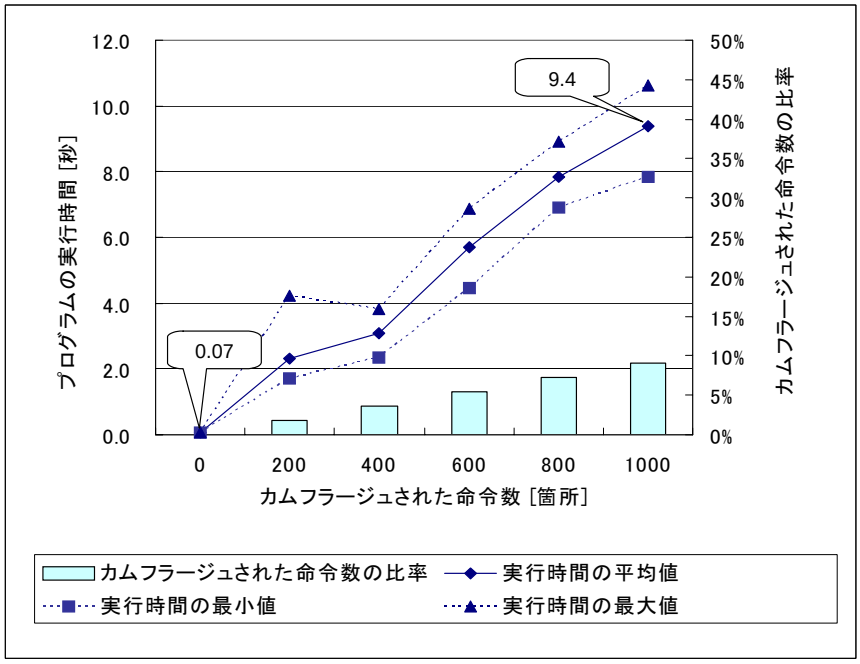

図 17　ccrypt の実行時間の変化

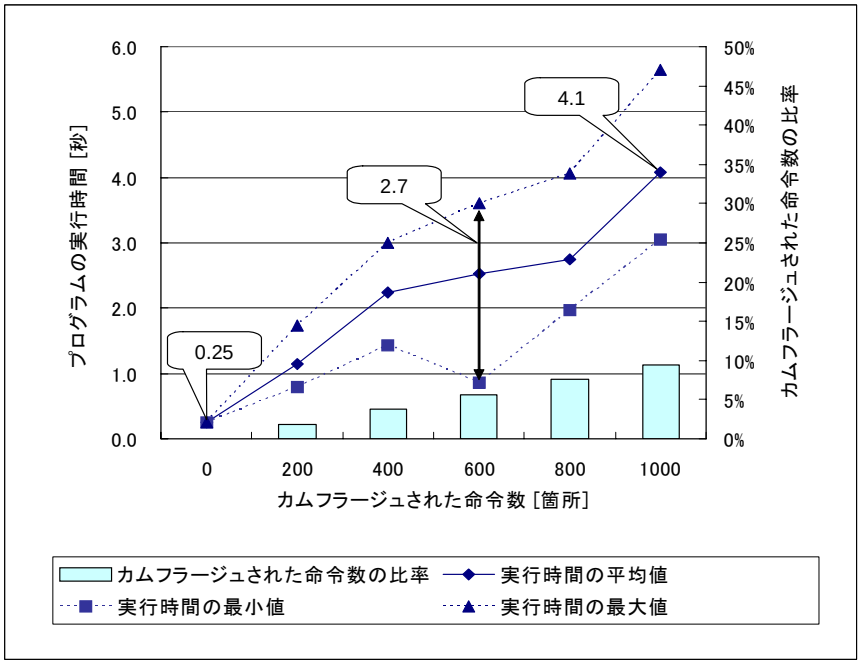

図 18　gzip の実行時間の変化

最小値を示す線および最大値を示す線に注目すると，実行時間が大きく変動していることがわかる．たとえば，gzip(図 18) の 600 の自己書き換えルーチンが挿入された場合には，最大値と最小値の間に約 2.7 秒という大きな差が認められる．このことは，自己書き換えルーチンが挿入される位置によって，プログラムの実行時間が大きく変化することを示している．このことから,「条件が満たされる位置にランダムに挿入する」という自己書き換えルーチンの挿入方法を改良することによっても，実行時間の増大を軽減することが期待できることがわかる．例えば，ルーチンの挿入が可能である各位置に対して，その位置の実行頻度を解析し，その結果を参照して，実行頻度の少ない位置にルーチンを挿入するように改善するということが考えられる．このような改良案の具体化および実装は，今後の課題となる．

5.3.3 プログラムサイズの増分とプログラムの実行時間の増分の関係について

図 19 は，プログラムサイズの増分とプログラムの実行時間の増分の関係を示すグラフである．横軸は，プログラムサイズの増分を表し，縦軸は，プログラムの実行時間の (平均値の) 増分を表す．

このグラフから，プログラムサイズの増分が大きくなるのに比例して，プログラムの実行時間の増分が大きくなることがわかる．ただし，ccrypt の場合と，gzip の場合では傾きが大きく異なっている．この傾きの大きさは，増加するサイズの量，すなわち，追加された自己書き換えルーチン数の量に応じて，どれだけ実行時間のロスが生じるかということを示している．この傾きがプログラムによって大きく異なるのは，各々のプログラムに追加された自己書き換えルーチンの実行回数に大きな差があるためであると考えられる．

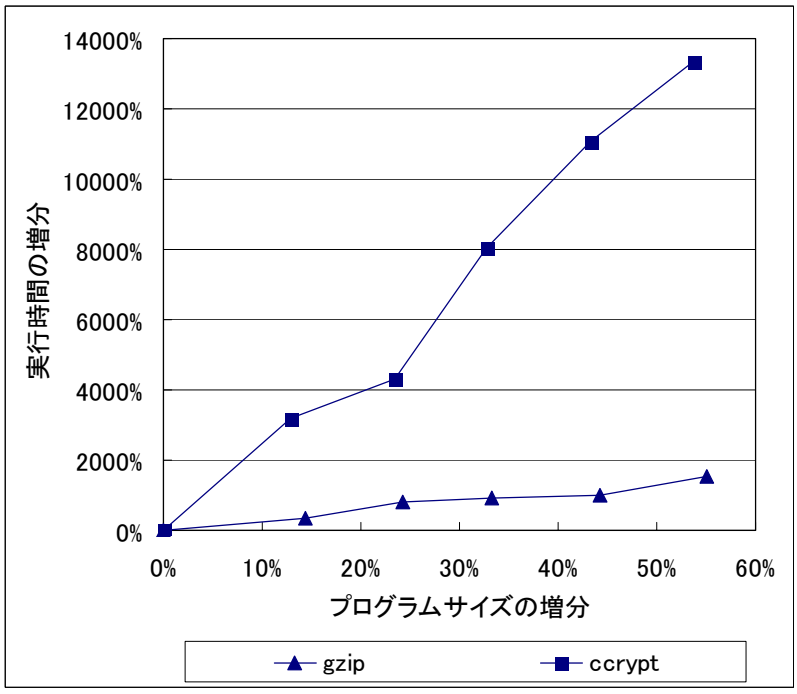

図 19　プログラムサイズの増分と実行時間の増分の関係

6. おわりに

本論文では，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．提案方式を用いれば，実行時のある期間だけオリジナルのものに書き換わる命令コードが多数含まれたプログラムを構成することができる．これらの命令コードは，実行時のある期間を除いてダミーの命令コードでカムフラージュされているため，解析者の誤った理解を誘う．したがって，このような命令コードが多数含まれたプログラムは，解析に著しく大きなコストを要することになる．また，自己書き換えを行うために追加されるルーチンは非常にサイズが小さく，プログラム中に散在しているため，保護機構を無効化することも困難である．さらに，多くの計算機環境において, 特別なハードウェアを追加せずに実施が可能であるという特長を持つ．

5 章で述べた，プログラムサイズおよびプログラムの実行時間への影響を評価する実験を通しては，プログラムサイズへの影響よりも，プログラムの実行時間への影響が大きいことがわかった．したがって，提案方式は，高い実行パフォーマンスが要求されるプログラムには適用されるべきではないといえる．一方，実行パフォーマンスを犠牲にしてもよい場合には，強い防止効果を持つ保護が行えると考えている．提案方式を適用するユーザは，適用の対象となるプログラムの目的や，求められる保護の強さを十分考慮して，適用の度合い (具体的には，カムフラージュする命令の数の多さ) を調整する必要がある．

最後に，今後の課題について述べる．まず，実行時間のロスが少なくなるように，提案方式を改良することが挙げられる．具体的には，自己書き換えルーチンの位置を決定するための方法を改良することで，ロスを軽減できるのではないかと考えている．これに加えて，ハードウェアの面から実行時間のロスの原因を考察することで，さらなる改良につなげることが期待できる．

また，より解析が困難なプログラムを構成することを目的として，オペコード (命令コードのうち，命令の種類を示す部分) だけでなく，オペランド (命令コードのうち，命令の処理の対象を示す部分) もカムフラージュできるように提案方式を改良することも考えられる．たとえば，`jmp L1` という命令 (ラベル `L1` にプ

ログラムの実行位置を移す命令) をカムフラージュの対象にするとき，オペコードの jmp だけでなく，ジャンプ先を示す，オペランドの L1 についてもカムフラージュすることができれば，さらに解析者の混乱を招くことができ，より解析が困難なプログラムを構成することができると考える．

謝辞

丁寧なご指導と多大なご助力をいただきました松本 健一教授に感謝いたします.

折にふれて貴重なご意見をいただきました渡邉 勝正教授，飯田 元助教授に感謝いたします.

研究生活に関する多くの貴重な助言をいただきました島 和之助手に感謝いたします.

研究を進めるに当たり，有益なご助言，ご指導をいただきました門田 暁人助手に感謝いたします.

貴重な助言や励ましの言葉をいただきました中村 匡秀助手に感謝いたします.

また，ソフトウェアプロテクション班の一員としてともに研究活動をしてきた佐藤 弘紹氏をはじめ，同輩としてともに支えあってきた大杉 直樹氏，岡井 洋樹氏，岡久 浩章氏，粕淵 健郎氏，亀井 俊之氏，後藤 徹平氏に感謝いたします.

最後に，公私ともに私を支え，成長させてくださいましたソフトウェア工学講座の皆様に深く感謝いたします.

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] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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, pp.13-19, Dec. 2002.

[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, 神崎雄一郎, "自己書き換え処理追加プログラム，自己書き換え処理追加装置及び自己書き換え処理追加方法," 特願 2002-355881, Dec. 2002.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[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.