類似した命令列の畳込みによるプログラムの耐タンパ性の向上

西岡 隆司 †　山内 寛己 †　門田 暁人 †　中村 匡秀 †　松本 健一 †

† 奈良先端科学技術大学院大学 情報科学研究科
〒 630-0192 奈良県生駒市高山町 8916-5
E-mail: †{ takashi-ni, hiroki-y, akito-m, masa-n, matumoto } @is.naist.jp

要 旨

本稿では，命令の自己書き換えを用いたソフトウェア耐タンパ化方法として “類似した命令列の畳込み” を提案する．提案方法では，プログラムに含まれる類似した複数の命令列を一つの命令列にまとめる．命令列間の差分は，自己書き換えにより，実行時に必要な命令に置き換える．畳込まれた命令列に改ざんを加えると，全ての畳込み元の処理に影響を与えるため，耐タンパ性が確保される．

Improvement of Tamper Resistance of Programs Based on Instruction Folding

Takashi Nishioka †　Hiroki Yamauchi †　Masahide Nakamura †
Akito Monden †　and Ken-ichi Matsumoto

†Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara, 630-0192, Japan
E-mail: †{ takashi-ni, hiroki-y, akito-m, masa-n, matumoto } @is.naist.jp

Abstract

This technical report proposes a software tamper proofing method called instruction folding. In this method, a set of similar (but sligtly different) instruction sequences is detected in a program, and then, they are integrated (folded) into a single instruction sequence. A self-modifying mechanism is applied to the folded sequence to bridge the gap between those slightly different sequences at run-time. The folded sequence is tamper resistant since tampering affects every functionality where original (similar) instruction sequences were present.

1　はじめに

今日，悪意を持ったエンドユーザ (以下, 攻撃者と呼ぶ) によるプログラムの不正な解析・改ざんが問題となっている．例えば，ライセンスチェックルーチンの解析・改ざんによるソフトウェアの不正利用は，ソフトウェア業界の長年の懸案となっている．また, 近年では, デジタルコンテンツを扱うプログラムの解析・改ざんが特に問題となっている．1999 年には，DVD 再生プログラムの解析により，DVD-Video のデジタルコピー防止用の暗号化規格 CSS（Content Scrambling System）の復号鍵が漏洩した [5]．同様に，2007 年には，HD DVD および Blu-ray のプロテクト技術 AACS(Advanced Access Content System) の復号鍵が漏洩した [6]．そのた

め, プログラムの解析や改ざんを防ぐ “プログラムの耐タンパ化技術” の必要性が高まっている.

本論文では, プログラムの耐タンパ化技術の一つである命令のカムフラージュ[2][3] に着目し, その課題の一つを克服した新たな方法を提案する. 命令のカムフラージュは, プログラムに含まれる任意の命令（ターゲット）を異なる命令（ダミー命令）で偽装し, プログラムの自己書き換え機構を用いて, 実行時のある期間においてのみ元来の命令に復元する方法である. この方法の課題の一つは, 自己書き換えのターゲットとなる命令が攻撃者に発見されると, ダミー命令を元来の命令に置き換えることで, カムフラージュを解除する（オリジナルのプログラムへ復元する）ことが比較的容易な点である.

本稿では, 命令のカムフラージュと同様, 自己書き換えを用いた耐タンパ化方法である “類似した命令列の畳込み” を提案する. 提案方法では, プログラムに含まれる類似した命令列の集合を特定し, それら類似命令列の差分（非類似部分）を自己書換えにより吸収することで, 一つの命令列にまとめる（これを, 命令列の畳込みと呼ぶ）. この方法では, 自己書換えのターゲットとなる命令は, 書き換え前, 書き換え後のいずれにおいても,（ダミーではなく）プログラムの動作に必要不可欠なものである. そのため, 攻撃者が自己書き換えのターゲットとなる命令を発見したとしても, オリジナルのプログラムへ復元することは容易でない. また, 畳込まれた命令列に改ざんを加えることは, 全ての畳込み元の処理に影響を与えるため, 畳込みを完全に解かない限り, 畳込み部分を改ざんすることも困難である.

以下, 本稿では, 2 章で提案手法の説明に必要となる用語を定義し, 3 章で提案手法について述べる. 4 章で x86 プラットホームにおける実装例を示し, 5 章で評価実験と考察を行い, 6 章で本稿のまとめと今後の課題について述べる.

2 準備

2.1 耐タンパ化

ソフトウェアの耐タンパ化とは, ソフトウェアまたはその実行環境に何らかの処理を施すことで, 改ざん（タンパ：tamper）に対する耐性を確保することである. ただし, 一般に, ソフトウェアに対して意味のある改ざんを行うためには, 改ざんの対象となるソフトウェアを理解したり, ソフトウェア解析ツールを動作させたりする必要があるため, それらの作業を妨げることも耐タンパ化に含まれる. その要素技術としては, ソフトウェア自身の理解や解析を妨げる暗号化や難読化 [1], ソフトウェア解析ツールの動作を妨げるアンチデバッガやアンチ逆アセンブル, 改ざんを検出してソフトウェアの不正実行を妨げる integrity verification（改ざんチェック）などがある. これらの要素技術の多くは汎用的なものであるが, 近年では, 特定の種類のソフトウェアに特化した white-box cryptography などの耐タンパ化技術も提案されている. 一般には, これらの要素技術を多数併用することで, 耐タンパ化を実現することが望ましい.

2.2 自己書換え

自己書換えは, プログラムの暗号化, 難読化, アンチ逆アセンブルなどの要素技術として広く用いられており [4], 本稿の提案方法においても重要な要素技術の一つである. 自己書換えとは, プログラム P に含まれる命令 X が, P に含まれる他の命令 Y を異なる命令 Y’ に書き換えるプログラム記述形式のことである.

自己書換えを用いる場合の一つの課題は, プログラム実行時に “無効なメモリアクセス” 例外が発生することである. 実行ファイルがロードされたメモリ領域は, 通常, 読込み可, 書込み不可, 実行可であるのにも関わらず, 自己書換えによりプログラム領域の書き換えが発生するためである. 以降では, 無効なメモリアクセス例外を発生させずに, 自己書換えを行うためのプログラムを実行するための手順を示す.

- 実行ファイルは, セクションと呼ばれる領域を複数持っており, プログラムはコードセクションと呼ばれる領域に格納されている. コンパイラによって生成された実行ファイルのコードセクションの属性は, 書込み不可となっている. そこで, コードセクションの属性を書込み可とすることで, オペレーティングシステムは, 書込み可能なメモリ領域にプログラムをロードすることができる.
- 多くのオペレーティングシステムでは, メモリ領域のアクセス許可属性を変更するシステムコー

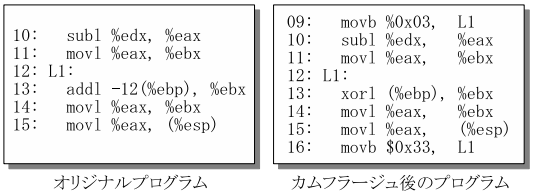

図 1: 命令のカムフラージュ

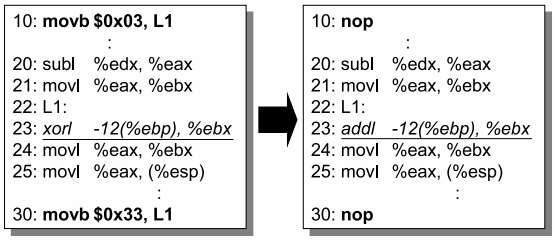

図 2: オリジナルプログラムへの復元

ルを持つ. 例えば, Linux においては, mprotect システムコールが該当する. プログラムの開始時に, 実行ファイルがロードされたメモリ領域の属性をこのシステムコールで書込み可とすることで, 無効なメモリアクセス例外を発生させることなく, 自己書換えを行うプログラムを実行することができる.

2.3 命令のカムフラージュ

本稿では，自己書換えを用いたプログラムの難読化の一手法である命令のカムフラージュ[2][3] に着目する．プログラムの難読化とは, 与えられたプログラムを，その仕様を保存したままで理解や解析が著しく困難なプログラムに変換することである．命令のカムフラージュは，プログラムに含まれる隠したい命令（ターゲット）を異なる命令（ダミー命令）で偽装し，プログラムの自己書き換え機構を用いて，実行時のある期間においてのみ元来の命令に復元することで，難読化を実現する．

図 1 に, オリジナルプログラムと命令のカムフラージュを適用したプログラムを示す. ターゲットとなる隠したい正規命令は, 13 行目の addl である. ダミー命令は, カムフラージュ後のプログラム 13 行目の xorl に対応する. 正規命令・ダミー命令に書換えるルーチンは, それぞれ 9 行目・16 行目に対応する. 命令のカムフラージュ法が適用されたプログラムの実行の流れを次に示す.

1. ダミー命令から正規命令を復元する
   9 行目において, 13 行目のダミー命令 xorl を正規命令 addl に書換える.
2. 正規命令を実行する
   13 行目において, 復元された正規命令 addl を実行する.
3. ダミー命令で正規命令を隠蔽する
   16 行目において, 13 行目の正規命令 addl をダミー命令 xorl に書換える.

以上のように, カムフラージュの対象となった正規命令がメモリ上に正しく表現される期間が 9 行目から 16 行目までとなるため, プログラムの内容理解を困難にすることができる.

2.4 命令のカムフラージュの問題点

命令のカムフラージュが適用されたプログラムにおいて, ダミー命令がプログラム実行に不要な（冗長な）命令であることが問題となる. そのため, ダミー命令と正規命令を切り替える自己書換え命令を発見することで, ダミー命令の期間が発生しないオリジナルプログラムへ復元する攻撃が可能である. 図 2 に, 命令のカムフラージュ適用済みのプログラムから適用前のプログラムを復元する手順を示す.

1. カムフラージュ機構を発見する
   正規命令に書換えるルーチンである 10 行目の mov 命令を発見する. この mov 命令は, L1 アドレスのプログラムに値 0x03 を書込む自己書換え命令である. L1 アドレスへの書込みに着目することで, ダミー命令が L1 アドレスに存在することが分かる. さらに, L1 アドレスへ書込む値 0x03 に着目すると, 0x03 は addl 命令のバイナリ表現であるから, 正規命令は, addl 命令であると分かる. ダミーの命令に書換えるルーチンを発見することでも, 同様に, ダミー命令と正規命令が何であるかが分かる.
2. カムフラージュ機構を無効化する
   カムフラージュを行ったアドレスである 23 行目を正規命令である addl 命令に書換え, 10 行目, 30 行目のカムフラージュルーチンを何の操作も行わない命令 (NOP 命令) で埋める.

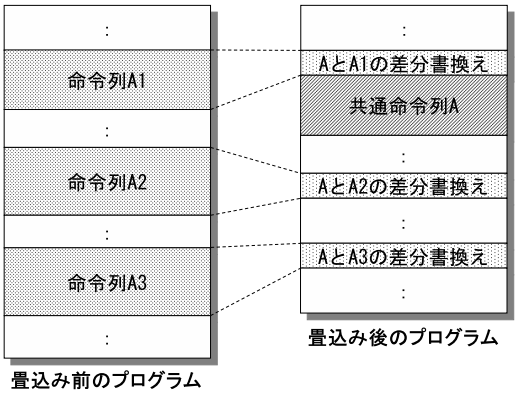

図 3: 命令列の畳込みの概要

3 提案手法

3.1 キーアイデア

命令のカムフラージュ法は, ダミー命令がプログラムの実行に必要とされないため, ダミー命令を除去し, オリジナルプログラムに復元することができることが問題となった. ダミー命令と正規命令の切替えを行うのではなく, プログラムの実行に必要不可欠な命令列 A と命令列 B の切替えを行うようにすることで, 上記の問題を解決できると考えている. 具体的には, 1 つの命令列が複数の命令列の内容を持つことが出来るように, 自己書換えを用いて, 実行時に命令列の内容を切り替えることで, 自己書換え前も実行に不可欠な命令列であり, 自己書換え後も実行に不可欠な命令列となる. 本稿では, 自己書換えを用いて複数の類似する命令列を 1 つの命令列に畳込み, 実行時にそれらの命令列間の差分を書換えることで, 攻撃者がプログラムを改ざんすることを防ぐ手法を提案する.

図 3 は, 命令列の畳込みの概要を示したものである. 畳込み前のオリジナルプログラムは, 類似した命令列 A1, A2, A3 を持つ. このプログラムに対して畳込み処理を行うことで得られたプログラムが畳込み後のプログラムである. これは, A1, A2, A3 で共通した命令を取出して作成した共通命令列 A, A と A1 との差分を書換える命令列, A と A2 との差分を書換える命令列, A と A3 との差分を書換える命令列で構成される. 次に, 畳込み後のプログラムにおいて, 命令列 A2 がどのように実行されるかについて説明する.

1. A と A2 の差分の命令を書換えることにより, 共通命令列 A は命令列 A2 となる.
2. 自己書換えにより命令列 A2 に切替えられた共通命令列領域へ実行の制御を移し, 命令列 A2 を実行する.
3. 命令列 A2 の次に実行する命令が存在するアドレスへ実行の制御を移す.

共通命令列領域は, 命令列 A1, A2, A3 の実行で共通して利用される. 畳込み後のプログラムにおいて, 命令列 A1 のみを改ざんしたいと考えている攻撃者を想定すると, 攻撃者は共通命令列領域を改ざんすることになる. しかし, 共通命令列領域を改ざんすると, 同時に命令列 A2, A3 の実行に影響を与えるため, プログラムの誤動作を引き起こす. したがって, 攻撃者による命令列 A1 の改ざんは失敗である. また, 共通命令列 A と命令列 A1, A2, A3 の差分を書換える自己書換えルーチンは, プログラムの正常実行に必要不可欠なものであるため, 改ざんはプログラムの誤動作を引き起こす. 以上のように, 命令列の畳込みでは, 攻撃者によるプログラムの改ざん耐性を有していることがわかる.

3.2 類似命令列の畳込み

3.2.1 類似命令列

命令 m は 1 個のオペコードと p 個のオペランドを持つとするとき, $m = (opcode, operand_1, ..., operand_p)$ と定義する. また, 命令列 M は q 個の命令を持つとするとき, $M = (m_1, ..., m_q)$ と定義する.

類似命令列を既存の文字列検索アルゴリズムで発見できるようにしたいと考えている. 文字列検索アルゴリズムでは, ユニコード等の文字コードを用いた文字同士の大小比較を用いている. 文字列検索アルゴリズムを類似命令列の検索に適用するためには, 全ての命令において互いに大小関係が定義されている必要がある. 2 つの任意の命令を引数として取り, 2 つの命令間の大小関係を定義する関数 $f_c(x, y)$ が与えられたときの類似命令列の定義を次に示す. 長さ n の命令列 $p = (p_1, p_2, ..., p_n)$, $q = (q_1, q_2, ..., q_n)$ について, $f_c(p_i, q_i) = 0$(ただし, $i \leq n$) を満たす命令の数が $match(p, q) = n$ となるとき, p, q は類似命令列であり, p, q の組み合わせを類似命令列組と呼ぶ.

3.2.2 畳込み命令列

次に, 命令列の畳込みについて述べるために必要となる用語を定義する.

畳込み命令列 畳込み命令列は, プログラムに含まれる畳込み対象の命令列のことである.

畳込み命令列組 n 個の畳込み命令列 $A_1, ..., A_n$ がある 1 つの命令列 $A_i(i \leq n)$ に畳込まれるとすると, 畳込み命令列組 Z は, $Z = (A_1, ..., A_n)$ と定義される.

あるプログラム P に含まれる全ての畳込み命令列を m 個とすると, 畳込み命令列は $A_1, ..., A_m$ と表現できる. これらの畳込み命令列を構成する畳込み命令を, $A_i = A_i^1, ..., A_i^n$(ただし, n は A_iの命令列長) とした時の, 畳込み命令が満たすべき条件を次に示す.

1. 全ての命令 A_i^j(ただし, $i \leq m, j \leq n$) は, P 上における位置を移動させても, P は正しく動作する.
2. プログラムに含まれるある命令 p の位置を $index(p)$ と表現すると, P に含まれる全ての畳込み命令が離れた位置に存在すること. すなわち, すべての i, j(ただし, $i \leq m$ かつ $j \leq n$) において, $index(A_i^j)$ が互いに異なる値であることが必要である.

3.3 類似命令列の畳込みによる耐タンパ化

本節では, キーアイデアに基づいた類似命令列の畳込みの実現方法について述べる. 始めに, 図 4 を用いて, 畳込みの実現に無条件分岐命令と無条件分岐命令の分岐先を書き換える自己書換え命令が必要となることを示す.

あるプログラム P に 2 つの畳込み命令列 $A1, A2$ からなる畳込み命令組が存在するとする. 命令列 $A1$ の前に実行される命令を $A1_p$ とし, 後に実行される命令を $A1_n$ とする. さらに, 一連の実行の流れを $P1$ とする. 同様に, 命令列 $A2$ の前に実行される命令を $A2_p$ とし, 後に実行される命令を $A2_n$ とする. さらに, 一連の実行の流れを $P2$ とする. また, $A1, A2$ の畳込み後に作成された共通部分の命令列を $A1$・$A2$ とする. ここで, 畳込み後のプログラム P' を実行することを考え, $P1$ で実行された場合, $A1$・$A2$ は, $A1$ として実行され, 次に実行する命令が $A1_n$ である必要がある. また, $P2$ で実行された場合, $A1$・$A2$ は, $A2$ として実行され, 次に実行する命令が $A1_n$ である必要がある. そのため, 畳込み後のプログラムでは, 命令列 $A1$・$A2$ の実行後, $A1_n$ または $A2_n$ のどちらから実行を継続しなければならないかが問題となる. 本稿では, 命令列 A の末端に, 無条件分岐命令を追加し, 命令 $A1_p$ の実行後, 無条件分岐命令のジャンプ先を命令 $A1_n$ とし, 命令 $A2_p$ の実行後, 無条件分岐命令のジャンプ先を命令 $A2_n$ とすることで解決している.

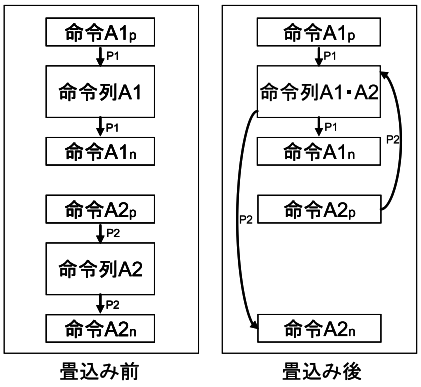

図 4: 命令列組の畳込み

プログラムの耐タンパ化を行う時の構成手順を次に示す.

1. 畳込み命令列群を決定する. 畳込み命令列の組合わせの候補となる命令列として, プログラムに繰り返して現れる類似命令, プログラムから改ざんを防ぎたい命令列とその命令列に類似した命令列から構成される類似命令列が考えられる.
2. 決定した畳込み命令列群を畳込むための自己書換え命令を決定する.
3. 自己書換えルーチンをプログラムに挿入する. 自己書換えルーチンが満たすべき条件として, 自己書換えルーチンを実行後, 畳込まれた命令列に制御を移し, 畳込まれた命令列を実行した後, 元々実行していたルーチンへ戻るがある.

4 実装

プログラムは, 類似命令列を列挙するプログラム, 類似命令列から畳込みの対象とする命令列を選択するプログラム, 選択された命令列を畳込むプログラムの 3 つの要素から構成される.

4.1 類似命令列の検索

本節では, 類似命令列の検索手法について述べる.

step1 プログラムを gcc でコンパイルし, アセンブリリストを取得する. このアセンブリリストを構文解析して, 命令オブジェクトの配列 p を得る.

step2 命令オブジェクトの配列 p の長さを n とすると, $p = (p_1, p_2, ..., p_n)$(ただし, p_iは命令オブジェクト) と表される. p が含む部分命令列の集合を k とすると, k が類似命令列を検索するための検索キー集合である. 開始位置を i とし, 命令列長を j とした時のある検索キーを, $k_{i,j} = (p_i, ..., p_{i+j-1})$ とした時の求める検索キー集合を次に示す.

$\{k \in k_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq (n-i)/2\}$(ただし, i, j は任意の自然数)

step3 3.2.1 項で, 既存の文字列探索アルゴリズムを利用するために, 順序関係を定義する関数が必要であることを示した. 本実装では, オペコード名で順序付け, オペランド種別で順序付け, オペランドが含む文字列で順序付けを行う関数を利用している.

step4 step2 で作成した全ての検索キーに対して, step3 で定義した順序付け関数と既存の文字列検索アルゴリズムを用いて, 一致する命令列を探索する. p からある検索キー k に一致する n 組の命令列 $r_1, ..., r_n$ をプログラムから発見したとすると, 類似命令列組は $r_1, ..., r_n$ となる. この手順を全ての検索キーに対して行うことで, 全ての類似命令列組を検索することができる.

4.2 畳込み命令列の選択

本節では, 4.1 節で得た類似命令列組から, 畳込み命令列組を求める方法を述べる. 3.2.2 項で述べたように, 類似命令列が含む全ての命令について配置位置の移動を行うことができ, かつ互いに離れた位置に存在する命令であることを満たす必要がある.

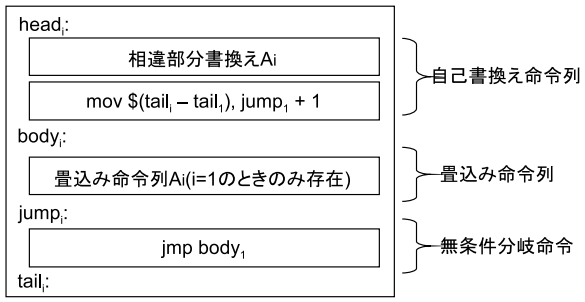

図 5: 畳込みルーチンの構成

- 類似命令列に含まれる全ての命令の組合わせ (p, q) において, プログラム上の位置が一致する ($index(p) = index(q)$ が成立する) 場合, q を含む類似命令列組を取り除く.

- 命令オブジェクトを 1 ノードとみなした制御フローグラフにおいて, 複数の基本ブロックをまたぐ類似命令列 q が存在するとき, q を含む類似命令列組を取り除く.

以上の操作を行い, 残りの類似命令列が畳込み命令列となる.

畳込み命令列は, 一定以上の長さを持つ命令列であることが望ましい. 自己書換え部分が畳込み命令列より大きなサイズとなると, 自己書換え部分を潰し, そこへ畳込み命令列を書き込むことにより, オリジナル命令列に復元する攻撃を行うことができるからである. 畳込み命令列組は, 2 つ命令列からなり, その命令列間で相違する命令数が d 個存在するとする. このとき, 相違部分を書き換えるために d 命令, 無条件分岐の分岐先を書き換えるために 2 命令必要なので, 畳込み命令列は $d + 2$ より長い命令長を持つ必要がある. これらの条件を満たす命令列を畳込むことにより, オリジナルプログラムと比較して, プログラムサイズが縮小する. このとき, 自己書換え部分を潰して, 元の命令列に復元する攻撃を行うことはできない.

4.3 x86 における自己書換えを用いた畳込み

本節では, x86 アーキテクチャにおいて, 命令列を畳込む手法について述べる. 図 5 に, n 個の命令列 $A_1, ..., A_n$ から構成される畳込み命令列組において, 命令列 A_i を命令列 A_1 に畳込む場合に作成する畳込みルーチンの構成を示す. なお, $\mathtt{head}_i, \mathtt{body}_i, \mathtt{jump}_i, \mathtt{tail}_i$ はラベルである.

自己書換え命令列 自己書換え部分は, 相違部分書換え命令列 A_i と jmp 命令の分岐先の書換え命令 $\mathtt{mov}\ \$(\mathtt{tail}_i - \mathtt{tail}_1), \mathtt{jump}_1 + 1$ からなる. 相違部分書換え命令列 A_i は, 畳込まれる命令列 A_i と共有命令列 A_1 の間に異なる命令が含まれる時, 共有命令列の相違部分を書き換える命令列である. 分岐先の書換え命令は, tail_1 に配置されている jmp 命令の分岐先を tail_i に書き換える命令である.

畳込み命令列 畳込み命令列は, 畳込み対象の命令列を表す. $i = 1$ の時, 畳込み命令列は, 畳込み対象の命令列 $A_1, ..., A_n$ の共有命令列となる. $i \neq 1$ の時, 畳込み命令列は, 共有命令列 A_1 に畳込まれるため削除する.

無条件分岐命令 $i = 1$ の時, jmp 命令は, 畳込み命令列の次に実行される命令へ制御を移すように, 自己書換え命令列によって分岐先が書き換えられる. 自己書換えと共に jmp 命令を用いることで, スタックポインタの値を変更することなく, 畳込み命令列へ制御を移した jmp 命令の次に実行される命令に戻ることができる. $i \neq 1$ の時, jmp 命令は, 畳込み命令部へ制御を移すための命令である.

図 6 に, 命令の畳込みの適用例を示す. (a) は, 命令の畳込み適用前のアセンブリリストであり, (b) は, 適用後のアセンブリリストである. 畳込み命令列組は, (a) の A_1, A_2 であり, 畳込み適用後は, (b) の A_1', A_2' となる. 次に, 畳込みルーチン A_1', A_2' の各構成要素の作成手法について述べる.

自己書換え命令列 A_1, A_2 において相違する命令は, `call _check_licence`, `call _check_args` であるから, それぞれ差分書換え命令は,
`movl $(_check_license − modify_tail), modify_head + 1`,

```
_task:
  pushl %ebp
  movl %esp, %ebp
  subl $24, %esp
  movl $0, -4(%ebp)
  movl 8(%ebp), %eax
  movl %eax, (%esp)
A1 call _check_license
  testl %eax, %eax
  je L8
  movl 8(%ebp), %eax
  movl %eax, (%esp)
A2 call _check_args
  testl %eax, %eax
  je L8
  movl 8(%ebp), %eax
  movl %eax, (%esp)
  call _do_task
  movl %eax, -8(%ebp)
  jmp L7
L8:
  movl $0, -8(%ebp)
L7:
  movl -8(%ebp), %eax
  leave
  ret
```

(a)適用前のアセンブリリスト

```
_task:
  pushl %ebp
  movl %esp, %ebp
  subl $24, %esp
  movl $0, -4(%ebp)
head1:
  movl $(_check_license - modify_tail), modify_head + 1
  movb $(tail1 - tail1), jump1 + 1
body1:
  movl 8(%ebp), %eax
  movl %eax, (%esp)
  modify_head:
A1' call _check_license
  modify_tail:
  testl %eax, %eax
  je  L8
jump1:
  jmp body1
tail1:
head2:
  movl $(_check_args - modify_tail), modify_head + 1
  movb $(tail2 - tail1), jump1 + 1
A2' body2:
jump2:
  jmp body1
tail2:
  movl 8(%ebp), %eax
  movl %eax, (%esp)
  call _do_task
  movl %eax, -8(%ebp)
  jmp L7
L8:
  movl $0, -8(%ebp)
L7:
  movl -8(%ebp), %eax
  leave
  ret
```

(b)適用後のアセンブリリスト

図 6: 命令列の畳込み適用例

movl $(_check_args − modify_tail), modify_head + 1

となる． 畳込み命令列 A_1, A2 の次に実行されるアドレスは， $tail_1, tail_2$ であるから， 無条件分岐命令の分岐オフセットは， $tail_1 - tail_1, tail_2 - tail_1$ となる．

畳込み命令列 (b) において, A_2 を A_1 に畳込むとしたので, 畳込み命令列は $A_1^{'}$ にのみ存在する．

無条件分岐命令 $jump_1, jump_2$ 共に， 畳込み先の畳込み命令列の先頭へ分岐できるように jmp $body_1$ とした．

5 実験

5.1 実験概要

提案手法のオーバーヘッドを計測するため, 400 万要素の数値データをマージソートするプログラムに対して, 類似命令列の畳込みを適用し, 実行時間を計測した．実験環境は, Intel Pentium4 2.2GHz, RAM 1024MB, WindowsXP SP2 であり, 最適化オプション O0 として gcc 3.4.4(cygwin) でコンパイルした．ただし, 命令列の畳込みを行った命令列組は 1 組である．畳込みを行った命令列組は, それぞれ 6 個の命令列からなる 2 つの命令列から構成される．また, これらの命令列を含む基本ブロックの実行回数は, プログラムに含まれる全ての基本ブロックの合計実行回数の 8.6%を占める．

5.2 実験結果と考察

畳込む前のプログラムと畳込み後のプログラムに対して実行時間の計測を 3 回行い, その平均値を求めたものを表 1 に示す．畳込み前のプログラムが 3.06 秒, 畳込み後のプログラムが 19.47 秒であった．この結果から命令列の畳込みを行うと約 6.3 倍遅くなることがわかる．したがって, 実行頻度の高い命令列への提案手法の適用は, 実行速度の大幅な劣化を引き起こすため, 注意深く畳込む命令列を選択する必要があると考えられる．

表 1: オーバーヘッド計測結果

畳込み前のプログラム	畳込み後のプログラム
3.06 s	19.47 s

6 おわりに

本稿では, 類似命令列の畳込みに基づく耐タンパ化手法を提案した．提案する耐タンパ化手法をプログラムに適用することにより, プログラムの改ざんが非常に困難となる．今後の課題として, 提案手法を適用することでどのような改ざんに対してどの程度困難であるかに対するより定量的な評価, 効果的な畳込み命令列の選択手法についての研究がある．

謝辞

本研究の一部は，文部科学省科学研究費補助金 (若手研究 (B), 課題番号:16700033) の補助を受けた．

参考文献

[1] C. Collberg and C. Thomborson, “Watermarking, tamper-proofing, and obfuscation - Tools for software protection,” IEEE Trans. on Software Engineering, Vol. 28, No. 8, pp. 735-746, 2002.

[2] 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 A, Vol.J87-A, No.6, pp.755–767, June 2004.

[3] 神崎 雄一郎, 門田 暁人, 中村 匡秀, 松本 健一, “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,” 2007 年暗号と情報セキュリティシンポジウム (SCIS2007) 予稿集 CD-ROM(講演番号 4D1-3), January 2007.

[4] C. Linn and S. Debray, “Obfuscation of executable code to improve resistance to static disassembly,” In Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS), pages 290–299, October 2003.

[5] A. Patrizio, “Why the DVD Hack Was a Cinch,” Wired News, Nov. 1999, http://www.wired.com/news/technology/0,1282,32263,00.html

[6] Zonk, “HD-DVD and Blu-Ray Protections Fully Broken,” Slashdot, Jan. 2007, http://slashdot.org/article.pl?sid=07/02/13/1724238