可変な中間コードとして振舞うデータ部とそれを実行するインタプリタ部からなる2部構成の耐タンパーソフトウェア作成法

吉田 直樹　　吉岡 克成　　松本 勉

横浜国立大学 大学院 環境情報学府・環境情報研究院
240-8501 神奈川県横浜市保土ヶ谷区常盤台 79-7
{yoshida-naoki-dk, yoshioka, tsutomu}@ynu.ac.jp

あらまし 耐タンパーソフトウェアの構成法として自己書換えを用いた方法が知られている. 自己書換えはプログラム自身のコードをそのプログラムの実行中に書換える技術であるが, 組込みシステム向けの一部のマイコンでは命令メモリの書換えができず, 機械語プログラムを書換える方法が適用できない場合があり, そのような場合においても有効となる方法にニーズがあると考えられる. 本稿では, データメモリに可変な中間コードとして振舞うデータ部を格納し, 命令メモリにそれを実行するインタプリタ部を格納する2部構成の耐タンパーソフトウェアの作成方法を提案する. この方法は, 命令メモリを書換えられないマイコンに適用できるだけでなく, 命令メモリ内の機械語プログラムを自己書換えする方法に比べて一般的に高速な実行が可能であるという特徴を有する.

Writing Two-Part Tamper Resistant Software with Data Part as Modifiable Intermediate Code and Interpreter Program Part

Naoki Yoshida　Katsunari Yoshioka　Tsutomu Matsumoto
Yokohama National University
Graduate School and Research Institute of Environment and Information Sciences
79-7 Tokiwadai, Hodogaya, Yokohama, Kanagawa 240-8501, JAPAN
{yoshida-naoki-dk, yoshioka, tsutomu}@ynu.ac.jp

Abstract Certain program self-modification techniques are useful to enhance tamper-resistance of software. A self-modifying code changes its appearance when it is executed. However a typical embedded microcontroller, which adopts the Harvard memory architecture with separate instruction and data memories, prohibits a code from changing the contents of the instruction memory during execution. To apply the self-modification based tamper-resistance enhancement methods for codes to be executed on such an embedded microcontroller this paper develops a "simulation" technique, which converts a given code into another output code. The output code consists of an "interpreter" part to be located in the instruction memory and an "intermediate code" part to be located in the data memory. In addition, for a microcontroller with the von Neumann memory architecture the speed overhead of the tamper-resistance enhanced code against its original version becomes small compared to the conventional self-modifying code located wholly in the instruction memory.

1　はじめに

パソコンや携帯端末などの各種の機器内にはセキュリティ機能実現のために暗号のアルゴリズムや鍵のデータなどが実装されることが多い．そのような暗号の鍵が攻撃者の手によって不正に読取られること，もしくは都合のよいように暗号のアルゴリズムを改変されることなどは当然避けなければならない．

攻撃者が機器内に存在するデータやアルゴリズムを読取ることが困難であることを指す秘密情報守秘性と，それらを改変することが困難であることを指す機能改変困難性とを総称して耐タンパー性という．この耐タンパー性を向上させる技術は社会の重要な基盤技術となってきている．耐タンパー性を保つソフトウェアは耐タンパー性ソフトウェアと呼ばれ，多方面から研究が進められている．

プログラム実行中に自身の「プログラムコードの書換え」を行うことを自己書換えと呼ぶ．この自己書換えを用いた耐タンパー技術には，機械語プログラムの自己書換えを用いた命令のカムフラージュ手法[1]などが既に知られている．しかし，機械語プログラムの自己書換えには，プログラムメモリを書換えられない組込み機器向けマイコン（マイクロコントローラ）に適用しにくいことや，自己書換えを行わないオリジナルのプログラムに比べて実行時間の増加が著しいことなどの点で課題があった．

本稿では，上記課題に挑戦している．まず，既存の中間言語，あるいは保護対象のプログラムに対して定義した中間言語を導入し，それらの中間言語で記述した中間コードである可変なデータ部と，その中間コードを読込んで処理を行うインタプリタ部の 2 つの部分でソフトウェアを構成することとする．その上で，自己書換え手法の適用対象を，直接機械語プログラムではなく，中間コード部分とすることを考える．すなわち，中間コードの書換えを活用する耐タンパーソフトウェアの作成方法を提案する．

本稿の構成は次の通りである．まず 2 章で命令データを書換える耐タンパーソフトウェアの関連研究について述べる．続く 3 章で提案方式の考え方と具体的方法と実現できる耐タンパー性に関する考察を示した後に，4 章で提案方式の性能評価実験とその結果を述べ，5 章でまとめを行う．

2　自己書換え型耐タンパー技術

自己書換えを用いてソフトウェアの耐タンパー性を向上させる技術を自己書換え型耐タンパー技術と呼ぶ．関連研究で示されている自己書換え型耐タンパーソフトウェア技術を概観し，自己書換え型耐タンパーソフトウェア技術の持つ課題について述べる．

2.1　命令のカムフラージュによるソフトウェア保護

機械語プログラムの命令は，文字通り機械語で表現されている．例えば intel x86 アーキテクチャでは加算（addl）と減算（subl）は表 1 のように文字列で記述されている．この文字列を書換えれば命令の書換えができたことになる．

この命令の書換えを応用した方法に,「命令のカムフラージュによるソフトウェア保護方法」（以下，KMNM 法という）[1] がある．その簡単な具体例を図 1 に示す．

まず，カムフラージュを行うターゲット命令を定める．そのターゲット命令をカムフラージュするため，ダミー命令に置換える．次に，ダミー命令が実行される前のパスにダミー命令をターゲット命令に復元する復帰ルーチンを挿入し，ダミー命令が実行された後のパスにターゲット命令をダミー命令に書戻す隠ぺいルーチンを挿入する．復帰ルーチンと隠ぺいルーチンの総称として自己書換えルーチンという．自己書換えルーチンは，機械語プログラムを書換えることで命令を書換えている．

表1　intel x86 の加算と減算の機械語

機械語	命令（アセンブラ）
0x01 d8	addl %eax, %ebx
0x29 d8	subl %ebx, %eax

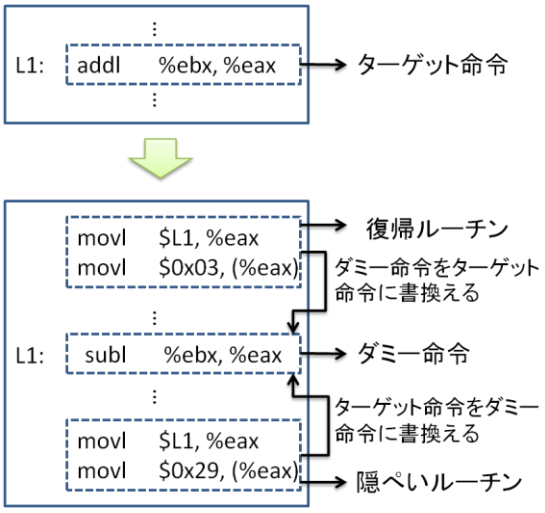

図1　命令のカムフラージュの例

ターゲット命令はプログラム実行中の復帰ルーチンの実行後から隠ぺいルーチンの実行後までの間で出現することになる．そのため，プログラムを逆アセンブルし，どのようなアルゴリズムであるのかを静的解析をしても，ダミー命令が紛れているため，処理を正しく把握することを難しくする効果がある．本来の命令であるターゲット命令を見つけるためには，復帰ルーチンがどこにあるかを見つける必要があり，攻撃者はプログラムを広い範囲に渡って見なければならず，攻撃者がプログラムの正しい処理を把握することが困難となる．よって秘密情報守秘性の向上を狙える．

なお，KMNM 法をプログラムに適用する自動化ツール RINRUN [9] が，KMNM 法の提案者によって公開されている．

2.2　動的自己書換えによる自己破壊的タンパー応答

「動的自己書換えによる自己破壊的耐タンパー応答」（以下，OM 法という）[2]は，KMNM 法と自己インテグリティ検証[3]を組合せた耐タンパー手法である．自己インテグリティ検証とはプログラム自身がプログラム実行時に自身が改ざんされていないか確認する手法である．機械語プログラムのある一部を保護領域とし，事前にその領域のハッシュをとり得られたハッシュ値を参照値とし，プログラム内部に持つ．プログラム実行時に同じ保護領域のハッシュをとり，参照値と比較することで改ざんされているかどうかを判断することができる．

この自己インテグリティ検証を，自己書換えルーチンと組合せる．自己書換えルーチンが命令を書換える際に，書換えを行うアドレスを smaddress とし，smaddress はプログラムの命令が存在するメモリアドレスに近い値を取るベース値（base）とその自己書換えルーチンが保護する保護領域にハッシュを取り得られた値（hv）と補正を行うマスク値（mask）を用いて

$$smaddress = base + hv - mask$$

の式で求める．base と mask を調整することで，hv が参照値と一致した場合には，書換える対象とする命令のアドレスと smaddress が一致するように実装する．

保護領域に改変がなければ正しい機械語プログラムの自己書換えが行われるが，改変があった場合には正しい自己書換えが行われず，本来書換えるはずの命令と，別の命令の少なくとも 2 つが本来処理される命令とは異なる．改変があれば正しくプログラムは実行されないため，機能改変困難性の向上が狙える．

2.3　自己書換え型耐タンパー技術の課題

自己書換えを用いることでソフトウェアの耐タンパー性の向上を狙える．しかし，機械語プログラムを書換える自己書換えは，組込みシステムでは適用できない場合がある．組込みシステムの機能はマイコンによって，実現されている．

マイコンには多様なものがあり，CPUの構成や開発環境などが異なっている．図2 に2011年におけるマイコンベンダーのシェアを示す．

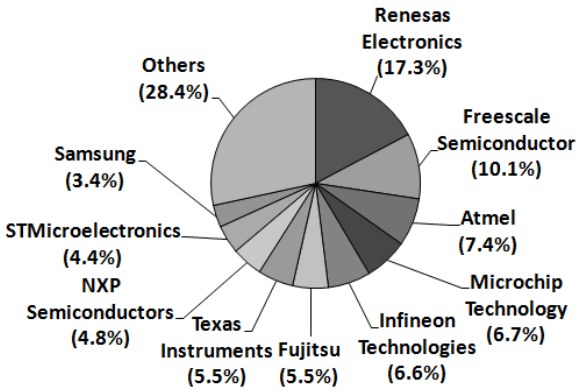

図2　2011 年のマイコンベンダーのシェア[4]

マイコンはプログラムコードが格納されている命令メモリと，変数や定数などのプログラムで用いるデータを格納するデータメモリとが別々のメモリに配置されている構成のハーバード・アーキテクチャが採用されているものがあり，ハーバード・アーキテクチャでは機械語プログラムの自己書換えを想定した構成になっていない場合が多い．

たとえば，2011年のマイコンのシェア第3位であるAtmel社が発売しているAVRのCPUは図3のような仕組みになっている[5]．機械語プログラムはFlashメモリに格納されており，プログラムカウンタに対応するアドレスの機械語命令をInstruction Registerにフェッチし，そのレジスタを参照して命令デコードが行われる．Flashメモリの内容を一部だけ書換えることはできない．書換えのできる機械語命令はInstruction Register に格納された命令だけであり，機械語プログラムの自己書換えを行うKMNM法やOM法は適用できない．

ところで一般にパソコンはハーバード・アーキテクチャとは異なり，命令メモリとデータメモリが1つのメモリに配置されている構成のフォンノイマン・アーキテクチャである．また，組込みシステムにおいてもフォンノイマン・アーキテクチャを採用したマイコンが用いられる場合もある．

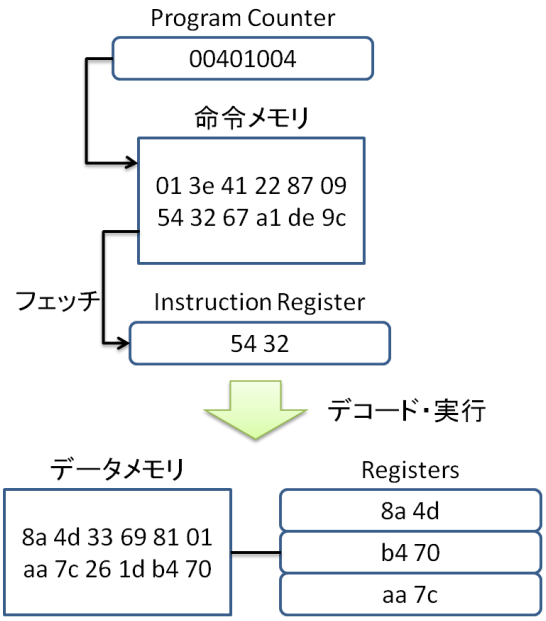

図3　AVR の CPU の仕組み

フォンノイマン・アーキテクチャでは，機械語プログラムの自己書換えは一般に可能とできることが多い．しかし，命令メモリを書換えると命令キャッシュや命令プリフェッチといったプログラムの高速化手法が有効に働かない場合が多い[6]．このため，自己書換えを行うと実行時間が著しく増加する傾向がある．

既存研究の方法を実用機器に対して適用するためには，上記のような課題がある．

3　提案方式

本章では，最初に自己書換え型耐タンパーソフトウェアの課題を解決する提案方式のアイディアについて述べる．そして，提案方式の構造について述べ，提案方式の耐タンパー性について考察する．

3.1　提案方式のアイディア

KMNM 法や OM 法では，機械語プログラムを書換えており，命令メモリ内の書換えを行うことがネックであった．これを解消するために，データメモリにプログラムコードを格納し，そのプログラムコードの書換えを行うというアイディアを得た．その実現のために，Simulation [7] と KMNM 法や OM 法を組合せる．

Simulation は難読化手法の 1 つであり，ある中間言語を用いて，プログラムの中間コードを作成しデータメモリに格納し，データメモリに格納された中間コードを解釈・実行するインタプリタを機械語プログラムで作成し，命令メモリに格納する手法である．

KMNM 法や OM 法で機械語プログラムを対象としていたものを，中間コードを対象とするように変えた手法を KMNM’法，OM’法とする．

提案方式では，まず Simulation を適用しプログラムの構成を中間コードとインタプリタの 2 部構成にした後に，KMNM’法や OM’法を適用する．既存手法のプログラムのメモリ配置と提案方式のプログラムのメモリ配置を図 4 に示す．

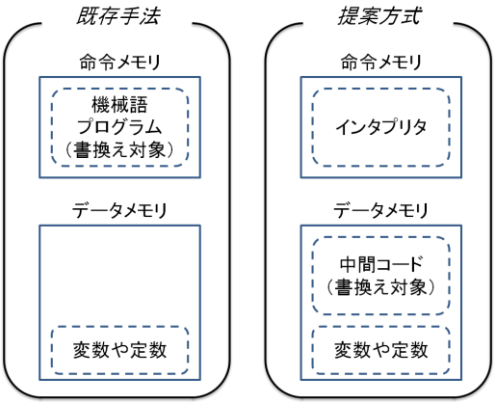

図4　メモリの配置

提案方式のようにプログラムを構成することで，ハーバード・アーキテクチャのマイコン（AVR 等）でも書換え可能なデータメモリ上の中間コードを書換え可能とでき，KMNM’法や OM’法を適用することが可能となる．実際に我々は，提案方式により作成した耐タンパーソフトウェアを AVR で動かすことに成功した．

また，フォンノイマン・アーキテクチャにおいても，命令メモリの書換えが起こらないために，キャッシュを用いた高速化手法が無効にならない．KMNM 法や OM 法を適用した場合に比べて実行時間の増加を抑えることが可能である．

3.2　提案方式の構造

耐タンパー化したいプログラム P と適用したい自己書換え型耐タンパー技術 T とを入力し，耐タンパー化されたプログラム Q を作成する手順を g として提案方式を記述する．g は以下の (f1), …, (f6) のステップからなり，図 5 のような構造をしている．

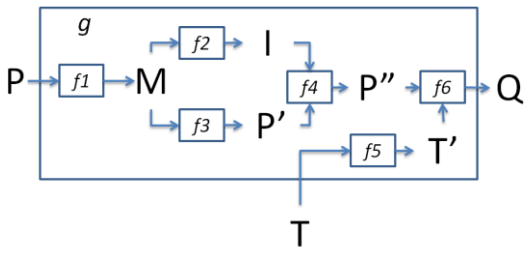

図5　提案方式(g)の構造

(f1)　P に基づく中間言語 M の作成

P と同じ処理が実現できるような中間言語 M を用意する．M は P を解析することによって作られた P 固有の中間言語であってもよいし，同じ処理が実現できるのであれば，Java のバイトコードなど既存の中間言語を利用してもよい．

(f2) M を解釈実行するインタプリタのソースコード I の作成

P’を読み出し，解釈・実行を行う M のインタプリタのソースコード I を，作成する．

(f3) M に基づく中間コード P’の作成

P と同じ処理を行うよう，M に基づいて中間コード P’を作成する．

(f4) P’と I を組合せたプログラム P”の作成

P’と I を組合せて，Simulation を適用したプログラム P”を作成する．

(f5) T の改変

入力された自己書換え型耐タンパー手法 T で機械語プログラムを対象としていたものを，中間コードを対象とするように変えた T’を作成する．

(f6)　P”への T’の適用

自己書換え型耐タンパー手法 T’を選び，P”に適用することで，Q を得る．

Q に変化を与えるパラメータを考える．Q は P”と T’から成り立つが，P”は I と P’から成り立ち，そのどちらも M に基づいて作成される．M は中間言語を作成する f1 によって変化するため，f1 は Q に変化を与える．また，T’は T によって変化する．つまり，T も Q に変化を与える．よって Q を変化させるには，f1 と T を変化させればよい．

3.3　提案方式による耐タンパー化の効果

提案方式で作成した耐タンパーソフトウェアについて，秘密情報守秘性と機能改変困難性の両者の観点から考察する．

1.　秘密情報守秘性

中間コードとインタプリタの 2 部構成により，アルゴリズムや秘密データを把握しようとする攻撃者は，インタプリタを解析し，ソフトウェアに用いられている中間言語の構造を把握

する．その後，中間コードを見てアルゴリズムや秘密データを把握するといったシナリオを採用すると考えられる．そうであるならば，インタプリタの解析の分だけ攻撃者の解析コストが高くなると考えられる．また，中間コード部分の解析においては，KMNM’法やOM’法の効果により中間コードの一部を見ただけでは処理が把握できず，全体を見渡さなければならない．

よって提案方式においても，秘密情報守秘性の向上が狙えると考えられる．

2. 機能改変困難性

攻撃者が改変を行う場合，中間コードの改変とインタプリタの改変の 2 つの攻撃が考えられる．

中間コードを改変する攻撃には，OM’法が適用されていれば，攻撃者はインテグリティ検証のつじつまが合うように中間コードの改変を行う必要がある．このため，中間コードに対する機能改変困難性はOM法と同程度の向上が狙えると考えられる．

インタプリタを改変する攻撃は，用いる中間言語 M によってその機能改変困難性が異なると考えられる．

例えばオペランドを多用することにより，命令の抽象度を高めて様々な場所から呼び出されているとする．その命令が用いられている特定の箇所を改変したい場合，インタプリタを改変すると改変をする必要のないところまで改変される．そのため攻撃者は中間コードの書換えを余儀なくされる．

逆に，抽象度が低く命令が 1 箇所でしか使われていない場合には，他に影響を与えることなくインタプリタを改変すればよい．

このように M によって機能改変困難性が保たれるかどうか変わるといえるが，どのように M を構成することが合理的であるかはまだ解っていない．インタプリタに対する機能改変困難性を向上が狙えるような M の構成を考えることが今後の課題となる．

4 提案方式の性能評価実験

本章では，KMNM 法と，KMNM 法を用いた提案方式の性能面での比較実験を行い，評価を行う．ハーバード・アーキテクチャのマイコンにおける実行の可否を確認した後，フォンノイマン・アーキテクチャのパソコン上でプログラムサイズと実行時間について実験し，考察する．

4.1 実験内容

RFC 3174 [8] で公開されている SHA-1 のソースコードを 1 つの C ソースファイルにし，20kByte のファイルの SHA1 のハッシュ値を取り参照値と比較するプログラムを作成した．これを耐タンパー化対象プログラムとする．耐タンパー化対象プログラムを gcc でアセンブラファイルにし，RINRUN で耐タンパー化されたアセンブラファイルを得る．これを gcc で実行ファイルにコンパイルして得られたものを KMNM 法の耐タンパー化プログラムとし，以下これを既存方式プログラムと呼ぶ．また，4.2 節で説明するように，KMNM 法を利用した提案方式を，耐タンパー化対象プログラムに適用して得られたプログラムを提案方式プログラムと呼ぶ．この 2 つのプログラムをパソコン上で実行し，命令のカムフラージュ数や自己書換えの回数を変化させてファイルサイズと実行時間を測定する．

実験環境は，実験計算機が OS は Windows XP Professional Version 2002 Service Pack 3 であり，メインメモリのサイズは 3.24GByte，CPU が Intel Core 2 Duo Processor E8400（クロック周波数 3GHz，1 次データ・キャッシュ 64kByte，2 次キャッシュ 6MByte）である．この計算機上で Cygwin をインストールし，実験プログラムを Cygwin 上でコンパイルと実行を行った．実行には Atmel の組込みマイコンの 32-bit AVR UC3 シリーズである AT32UC3A3256 の評価用ボードの EVK1104 [10] も使用した．

4.2 耐タンパー化手順

提案方式プログラムを作成するに至って，3.2 節に習い，以下の手順でプログラムを作成した．

（f1） P に基づく中間言語 M の作成

P 固有の中間言語の作成を行い，以下の(1)～(4)までの手順で作成した．

(1) プログラムに使われている命令を解析し，一命令ずつ抜き出す．ただし，変数の定義は除き，初期化されているならば抜き出す．

(2) 抜き出した命令で類似したものは統合し，オペランドにより処理を変えられるように抽象度を上げる．たとえば，以下のような 2 つの命令があったとする．

x += 1;　x += 2;

これは，どちらも加算であるので以下のように統合する．

x += (operand);

また，変数もオペランドで指定できるようにし，抽象度を高める．

a += (operand);　b += (operand);

(operand1) += (operand2);

(3) (2)で得られた命令群に，自己書換えを行えるように中間コードを格納する配列の書換えを行う命令を追加する．

(4) (3)で得られた命令群に 0 から番号を割り振っていく．この番号をその命令の M におけるオペコードとした．

（f2）M を解釈実行するインタプリタのソースコード I の作成

P’を読み出し，解釈・実行を行う M のインタプリタのソースコードを switch-case 文を利用して作成する．条件式には中間コードを格納する配列の中身を指定し条件分岐では（f1）で得たオペコードを用いる．case 文内では変数の宣言ができないため，必要な変数は事前に宣言する．

（f3）M に基づく中間コード P’の作成

P と同じ処理を行うように，M に基づいて中間コード P’を作成する．

（f4）P’と S を組合せたプログラム P”の作成

P’を unsigned char 型の配列に格納し，main 文に S を入れた新たなプログラム P”を作成する．

（f5）T の改変

KMNM 法を改変し，機械語プログラムを対象としていたものを，中間コードを対象とするように変えた KMNM’法を得る．

（f6）P”への T’の適用

P”に，KMNM’法を適用した．その際，KMNM 法の耐タンパー化プログラムとほぼ同じ処理になるように自己書換えルーチンとダミー命令を挿入し，Q を得る．

4.3 ハーバード・アーキテクチャのマイコン上での実行

EVK1104 に既存方式プログラムと提案方式プログラムを書き込み，実行させた．

既存方式プログラムは，動作はしたが，正しい処理とはならなかった．Debug モードで実行させ，書換え対象となるアドレスのメモリの挙動を見ながら自己書換えルーチンを実行させたところ，メモリの内容は変わっていなかった．すなわち，プログラムではダミー命令がそのまま実行され，したがって正しい処理は行われなかったと考えられる．

これに対し提案方式プログラムは正しく動作した．Debug モードで実行させ，書換え対象となるアドレスのメモリの挙動を見ながら自己書換えルーチンを実行させたところ，メモリの内容が確かに書換えられていた．ダミー命令はターゲット命令に復号され，正しい処理が行われたと考えられる．

4.4 プログラムサイズ

プログラムサイズは，図 4 で示された変数や定数を除いたもののそれぞれの合計値を計測する．ファイルサイズは命令のカムフラージュ数によって増加するので，命令のカムフラージュ数を変化させて計測を行う．その結果が図 6 であり，縦軸がファイルサイズ（Byte），横軸が命令のカムフラージュ数（個）を表す．

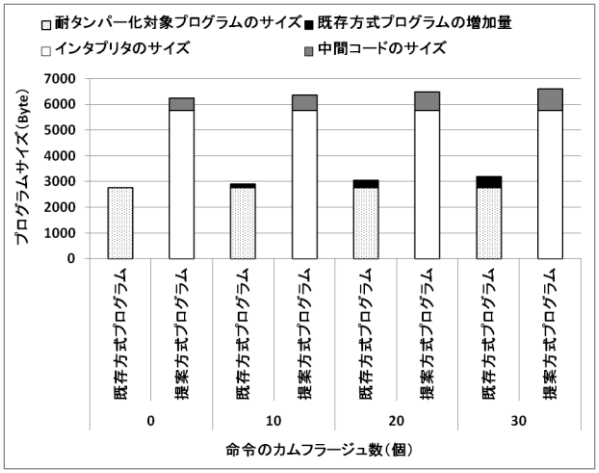

図6 ファイルサイズの変化

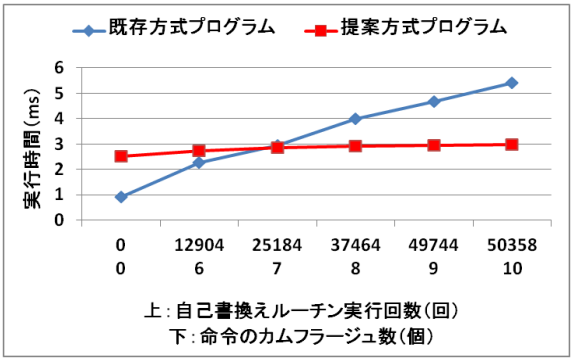

図7　実行時間の変化

4.5　実行時間

実行時間は，プログラムを 1,000 回繰り返して実行した際に要した時間を 1,000 で割ることで 1 回あたりの平均実行時間として求めた．時間計測には gettimeofday 関数を用いた．

実行時間は自己書換えルーチンの実行回数によって変化するので，命令のカムフラージュ数を変化させることで自己書換えルーチンの実行回数を変化させて計測を行った．その結果が図 7 であり，縦軸が平均実行時間（ms），横軸が自己書換えルーチンの実行回数（回）と命令のカムフラージュ数（個）を表す．

4.6　考察

KMNM 法を用いた提案方式を適用した場合，既存手法と比べてファイルサイズは約 2 倍増加していた．KMNM 法と KMNM 法を用いた提案方式で傾きにそれほど差はないため，プログラムをインタプリタと中間コードで構成した影響が大きいと考えられる．よって，提案方式のファイルサイズを小さくするには中間コードとインタプリタを小さくすればよく，それらの基となる中間言語の作り方を工夫する必要がある．

実行時間は KMNM 法と比べて，差異が得られた．カムフラージュ命令数が少ない場合には，インタプリタと中間コードを構成した時のコストが大きいために提案方式の方が実行時間はかかっている．しかし，カムフラージュ命令数が多い場合には，提案方式の傾きがかなり小さいために実行時間の増加量は小さく，対して KMNM 法は大きいので，実行時間に大きな差が出た．

5　まとめ

中間コードとインタプリタの 2 部構成の耐タンパーソフトウェアの作成方法を提案した．提案方式では，マイコンでも自己書換え型耐タンパー技術を適用することができた．また，実行時間の増加を抑えることもできた．

今後の課題には，機能改変困難性やファイルサイズや実行時間に変化を与える中間言語 M について調査し，最適な M を明らかにすること，提案方式を自動的にプログラムに適用するシステムを作成することが挙げられる．

参考文献

[1] 神崎雄一郎，門田暁人 中村匡秀松本健一，“命令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.

[2] 大石和臣，松本 勉，“自己破壊的タンパー応答を発生する耐タンパーソフトウェア，” 電子情報通信学会論文誌 Vol. J94-A, No.3, pp. 192-205, March 2011.

[3] H. Chang and M. J, Atallah, “Protecting software codes by guards,” Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management, LNCS Vol. 2320, pp.160-175, Springer-Verlag, 2002.

[4] EE Times, 2011 年のマイコン市場，震災乗り越えルネサスが首位を維持，http://eetimes.jp/ee/articles/1203/23/news032.html アクセス日：2012 年 8 月 20 日．

[5] Atmel, ATmega32A Complete, http://www.atmel.com/Images/doc8155.pdf アクセス日：2012 年 8 月 20 日．

[6] インテル, IA-32 インテル® アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル下巻: システム・プログラミング・ガイド, http://download.intel.com/jp/developer/jpdoc/IA32_Arh_Dev_Man_Vol3_i.pdf アクセス日：2012 年 8 月 20 日．

[7] Frederick B. Cohen, “Operating system protection through program evolution,” Computer Security Vol. 12, No. 6, pp. 565-584, 1993.

[8] RFC, US Secure Hash Algorithm 1 (SHA1), IETF http://datatracker.ietf.org/doc/rfc3174/ アクセス日 2012 年 8 月 20 日．

[9] 神崎雄一郎, RINRUN: プログラムカムフラージュ化ツール, http://se.naist.jp/rinrun/ アクセス日：2012 年 8 月 20 日

[10] Atmel, EVK1104, http://www.atmel.com/tools/EVK1104.aspx アクセス日：2012 年 8 月 20 日