情報処理学会論文誌 Vol.55 No.2 1100–1109 (Feb. 2014)

推薦論文

可変な中間コードとして振る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[1,a)] 吉岡 克成[1] 松本 勉[1]

受付日 2013年6月7日, 採録日 2013年11月1日

概要：耐タンパーソフトウェアの構成法として自己書き換えを用いた方法が知られている．自己書き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き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き換えができず，機械語プログラムを書き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．本論文では，データメモリに可変な中間コードとして振る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，命令メモリを書き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き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．

キーワード：耐タンパーソフトウェア，自己書き換え，中間コード，インタプリタ，組込みシステム

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

Naoki Yoshida[1,a)] Katsunari Yoshioka[1] Tsutomu Matsumoto[1]

Received: June 7, 2013, Accepted: November 1, 2013

Abstract: Self-modification is one of the techniques used to enhance tamper-resistance of software. A self-modifying code changes itself during execution in order to obfuscate its contents. However most existing self-modification methods are not directly applicable to a typical embedded microcontroller which adopts the Harvard memory architecture with separate instruction and data memories as it prohibits a code from changing itself on the instruction memory. This paper proposes a method to apply self-modification in such an environment. The proposed method converts a given code into a self-modifying code, which consists of an interpreter to be located in the instruction memory and an intermediate code to be located in the data memory. It is not only applicable to a code that runs on the Harvard memory architecture but also on von Neumann memory architecture. Moreover, on von Neumann memory architecture, we show that the speed overhead of our self-modifying code is smaller than that of existing self-modification methods.

Keywords: tamper-resistant software, self-modifying, intermediate code, interpreter, embedded system

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技術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不正に読み出されたり暗号技術の実装が攻撃者にとって都合の良いように改変されたりすることなどは当然避けなければならない．攻

[1] 横浜国立大学大学院環境情報研究院
Graduate School of Environment and Information Sciences, Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
[a)] yoshida-naoki-dk@ynu.jp

本論文の内容は 2012 年 10 月のコンピュータセキュリティシンポジウム 2012 にて報告され，同プログラム委員長により情報処理学会論文誌ジャーナルへの掲載が推薦された論文である．

撃者が機器内に存在するデータやアルゴリズムを読み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．

耐タンパー性を向上させる手法として，難読化があげられる．プログラムの命令を結果は同じだが複雑なものに変換したり，エイリアスを利用してコントロールフローを複雑にしたりといった静的な難読化手法が存在する [14]．ただし，このように難読化したプログラムは，時間をかければ部分的な動作を解読し理解することが可能である．そこで自己書き換えを用いた動的な難読化も求められている．

プログラム実行中に自身の「プログラムコードの書き換え」を行うことを自己書き換えと呼ぶ．この自己書き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き換えを用いた命令のカムフラージュ手法 [1] などがすでに知られている．しかし，機械語プログラムの自己書き換えには，プログラムメモリを書き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，自己書き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．

本論文では，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み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．そのうえで，自己書き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，中間コードの書き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．

本論文の構成は次のとおりである．まず 2 章で命令データを書き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. 自己書換え型タンパー技術

自己書き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，文字どおり機械語で表現されている．たとえば 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．この文字列を書き換えれば命令の書き換えができたことになる．この命令の書き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．

表 1 intel x86 の加算と減算の機械語

Table 1 Machine language of the addition and the subtraction in intel x86 architecture.

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

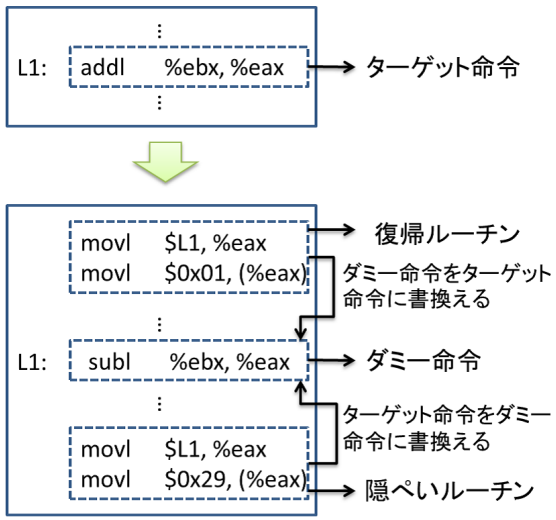

図 1 命令のカムフラージュの例

Fig. 1 Example of instruction camouflage.

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，ダミー命令に置き換える．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き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き換えルーチンという．自己書き換えルーチンは，機械語プログラムを書き換えることで命令を書き換えている．

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わた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 動的自己書き換えによる自己破壊的タンパー応答

「動的自己書き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] を組み合わ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，自己書き換えルーチンと組み合わせる．自己書き換えルーチンが命令を書き換える際に，書き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とるベース値（base）とその自己書き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，書き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き換えが行われるが，改変があった場合には正しい自己書き換えが行われず，本来書き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

自己書き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き換える自己書き換えは，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のシェアを示す．マイコンはプログラムコードを格納する命令メモリと，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き換えを想定した構成になっていない場合が多い．

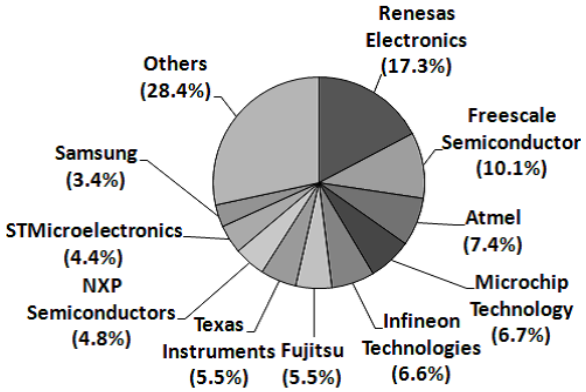

図 2 2011 年のマイコンベンダのシェア [4]
Fig. 2 Microprocessor vender's share in 2011.

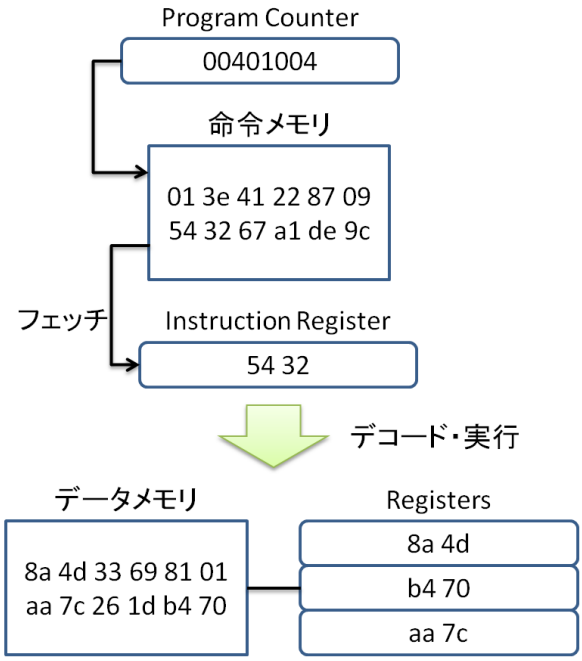

図 3 AVR の CPU の仕組み
Fig. 3 Structure of the CPU of AVR.

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き換えることはできない．書き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き換えを行う 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．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き換えは一般に可能であることが多い．しかし，命令メモリを書き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，自己書き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き換えており，命令メモリ内の書き換えを行うことがネックであった．自己書き換えを用いた手法として，ほかにもメタモフィックやポリモフィックといった手法が存在する．ソフトウェアを保護するための難読化の目的 [11] や，マルウェアがアンチウイルスソフトのパターンマッチングを防ぐ目的 [12], [13] で用いられている．文献 [11] では KMNM 法について，メタモフィックを用いた難読化であると論じられていた．このように，自己書き換えを用いた手法は実際にも用いられており，またさかんに研究されている．しかし，文献 [11], [12], [13] で示されているメタモフィックやポリモフィックの手法は，フォンノイマン・アーキテクチャのような自己書き換えが可能であるアーキテクチャを対象とした手法になっており，自己書き換えが不可能なアーキテクチャでは実行不可能である．そこで，本論文では，プログラムの構成を変更することで自己書き換え不可能なアーキテクチャでもメタモフィックやポリモフィックなど動的な難読化を用いられるようにし，様々なプラットフォームでの利用を目指す．

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き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 法を組み合わせる．

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．

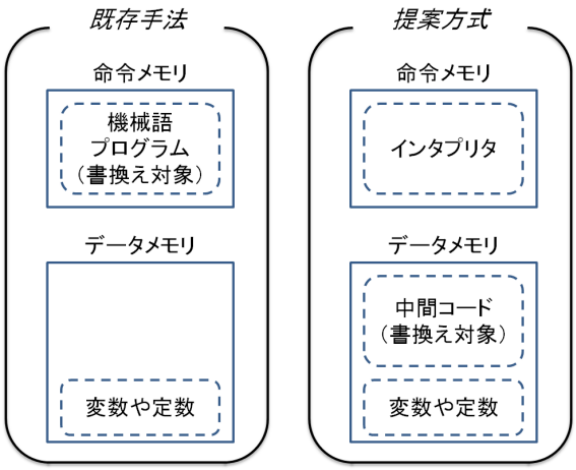

図 4 メモリの配置
Fig. 4 Location of the memory.

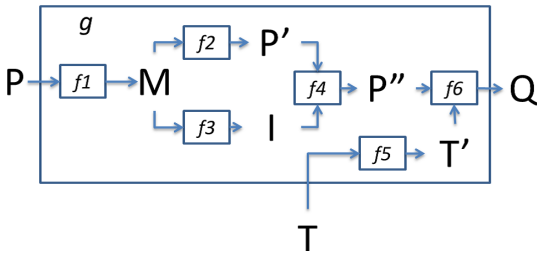

図 5 提案方式の構造
Fig. 5 Structure of suggestion method.

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 など）でも書き換え可能なデータメモリ上の中間コードを書き換え可能とでき，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，命令メモリの書き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．

(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 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．

(f3) 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．

(f4) P' と I を組み合わせたプログラム P" の作成

P' と I を組み合わせて 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 提案方式を適用できるプログラム

提案方式は，二部構成を行う実装上の特徴や自己書き換えを持つ特徴により，すべてのプログラムに適用できるわけではない．たとえば，オブジェクト指向の言語の場合，プログラムすべてを提案方式で耐タンパー化を施そうとすると，クラスやオブジェクトを独自の中間言語を用いて表現し，そのインタプリタを作成することが必要になってくる．クラスやオブジェクトまでを兼ね備えた中間言語やインタプリタを作成するのは容易ではなく，耐タンパー化を施すコストとしては著しいものとなるため提案方式は適さないものであると考えられる．ただし，オブジェクト指向の言語であってもプログラムすべてではなくクラス内のメソッドの処理手順に対してのみ適用を行うのならばこの限りではない．

また，C 言語で記述されていたとしても，DLL を用いたプログラムで実行時の必要なときに必要な DLL のロードを行う明示的リンクを行う場合，プログラムすべてを提案方式で耐タンパー化を施そうとすると，DLL も書き換え対象とすべきであるが，DLL はいつ呼び出されるか分からず呼び出される前はメモリ上に書き換え対象となるコードが存在しないために書き換えアドレスを指定することができない．ただし，明示的リンクで呼び出される DLL 自体が呼び出し元とは独立して提案方式を適用することは可能である．こうすることで呼び出し元と DLL の相互で書き換えを行わないため向上する耐タンパー性は低くなるが，DLL 自体を守ることは可能である．また，DLL を用いてもプログラム実行前にすべてのロードを行う暗黙的リンクであればメモリ上に書き換え対象となるコードが存在するため可能である．

以上のことから，どのプログラムも局所的な部分にならば適用は可能であると考えられるが，提案方式は特に C 言語で記述された静的なプログラムに対してはプログラムすべてに適用が可能であるため有効であると考えられる．

3.4 提案方式を適用できるプログラム

提案方式プログラムを作成するにあたって，3.2 節にならい，以下の手順でプログラムを作成した．なお，すべての工程に対してツールなどは作成しておらず，今回の提案方式プログラムはすべて手動で作成した．

(f1) P に基づく中間言語 M の作成

本実装では既存の中間言語を用いずに，手動で P を解析することによって P 固有の中間言語 M の作成を行う．M の抽象度はできるだけ高めるものとする．中間言語は，以下の (a)～(d) までの手順で作成した．

(a) C のソースファイルを見ていくことでプログラムに使われている命令を解析し，1 命令ずつ抜き出す．今回はソースファイル 1 行を分解能として命令を抜き出した．ただし，変数の宣言のみが行われている行に対しては，抜き出しを行わない．インタプリタを作成する際には switch-case 文を用いる実装する予定であり，変数の宣言も 1 つの命令として扱った場合 case 文の中で変数の宣言を行うことになるが，スコープの関係上コンパイルエラーとなってしまう．これを防ぐために変数は最初にすべて宣言を行う構造とするため，変数の宣言のみは抜き出しの対象としない．しかし，初期化されているならば変数への代入の命令として抜き出す．

(b) 中間コードの解析をより難しくするために，抜き出した命令で類似したものは可能な限り統合し，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式 (1) のように統合する．

$$x+=(operand); \tag{1}$$

また，変数についても型が同じであるのならば配列としてまとめ，変数を使う際にはオペランドで指定することで呼び出すようにし，抽象度を高める．

$int\ a = 0;\ int\ b = 1;$

これは，式 (2) のように統合を行った．

$$int\ variable[2] = \{0, 1\}; \tag{2}$$

式 (1) と (2) を組み合わせることで，int 型の加算の命令は式 (3) のようにオペランド 2 つを用いる 1 つの命令にまとめることができる．

$$variable[(operand1)] += (operand2); \tag{3}$$

また，ループ文と if 文の本質的な処理について考えると，なんらかの条件で分岐を行い，結果に応じてインタプリタが中間コードを読み込む位置を示すプログラムカウンタ変数の値を増減させてジャンプさせることで実装することが可能である．このため，ループ文と if 文は比較を行いその結果に応じてプログラムカウンタ変数の値の増減を行う式 (4) の命令に統合し実装する．

$$if((operand1) < (operand2))\{pc += (operand3);\} \\ else\{pc += (operand4);\} \tag{4}$$

（c）（b）で得られた命令群を用いることで P の処理をすべて行うことができる．しかし，この命令群にはまだ中間コードの自己書き換えを行う命令が含まれていないため，自己書き換えを行えるように式 (5) のような中間コードを格納する配列の書き換えを行う命令を追加する．

$$code[(operand1)] = (operand2); \tag{5}$$

（d）（c）で得られた命令群に 0 から (命令群の数 − 1) の値の中でかぶらないようにランダムに番号を割り振っていく．ランダムに番号を割り振ったのは，（c）で得られた命令群のリストの上から順番に番号を振ると，解析の特徴から，近い番号どうしで処理が行われていることになり，中間コードの解析のヒントとなることを防ぐためである．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 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．ソースコードから 1 行ごとに命令を抜き出したため，ソースコードを 1 行ずつ見てその処理に該当するオペランドとオペコードを探し出し，処理順に中間コードを作成していくだけである．ただし，ループ文と if 文に関しては比較を行いその結果に応じてプログラムカウンタ変数の値の増減を行う命令に統合してしまったため，このとおりにはいかず，先に分岐先の命令を中間コードで用意した後にそのコードサイズを計測し，プログラムカウンタ変数の増減量を定めて処理を組んだ．

（f3）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．（f1）で述べたように，case 文内では変数の宣言が行えないため，必要な変数は事前に宣言する．

（f4）P’ と I を組み合わせたプログラム P” の作成

P’ を unsigned char 型の配列に格納し，main 文に I を入れた新たなプログラム P” を作成した．P’ を格納する配列の型は作成した中間言語のオペコードの数やオペランドで用いられる値によって最適なものを選ぶ必要がある．今回の中間コードではオペコードやオペランドの値が 0 以上かつ 255 よりも小さかったため，プログラムサイズの増加を抑えるために unsigned char 型を選んだ．そして，中間コードを unsigned char 型のグローバル変数として宣言された配列に格納を行った．

（f5）T の改変

今回は自己書換え型耐タンパー化技術の中から KMNM 法を選び，これを改変し，機械語プログラムを書き換え対象としていたものを，中間コードを書き換え対象とするように変えた KMNM’ 法を得た．

（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き換えルーチンとダミー命令を挿入し，Q を得る．KMNM 法の耐タンパー化プログラムはプログラムをカムフラージュするシステムである RINRUN を用いて作成しており，RINRUN はカムフラージュ対象となる命令をランダムに選択し，命令のカムフラージュを行う．RINRUN を用いて耐タンパー化されたアセンブラファイルには，どの命令をダミー命令にしたかまた自己書き換えルーチンをどこに挿入したかがコメントアウトで記述されている．アセンブリ言語と P 固有の中間言語は異なるため，厳密に同様の自己書き換えを行うことはできない．そこで，KMNM 法の耐タンパー化プログラムのカムフラージュ対象となる命令と同等の処理を行う中間コード中の命令をカムフラージュ対象となる命令とした．ただし，各自己書き換えルーチンとカムフラージュ対象となる命令の実行順が KMNM 法の耐タンパー化プログラムと異なる場合には，カムフラージュ対象となる命令を，さきほど選んだ命令位置の近くかつ実行順が等しくなる命令とする．また，自己書き換えルーチンを挿入するということは，道中に余分に中間コードが挿入されるということでもあり，プログラムカウンタ変数の値の増減は相対位置で行っていたため，これがずれてしまう．そのため，ループ文や if 文が正しく動作しなくなってしまうので，プログラムカウンタ変数の値の増減を行った処理に関しては再調整を行い，プログラムが正しく動くように設定した．このような手順で再度実装を試みる場合は

プログラムカウンタ変数の増減量を（f2）で求めてしまうと再度計算を行う必要性が出てくるためここでは適当な仮値としたうえで，（f6）で初めて増減量を求めて挿入することが，効率が良いと考えられる．

3.5 提案方式の工数

今回の提案方式の実装では，すべてを手動で行っており，改変工数としてはかなり高くなる．そこで，各工程についてどこまで改変工数を抑えることができるか考察する．

（f1）P に基づく中間言語 M の作成

今回の実装では独自の中間言語を作成した．しかし，プログラムを行ごとに分割するなどアルゴリズムに沿って手動で作成していたため，このアルゴリズムに沿うツールを作成することで自動化は可能であると考えられる．また，独自の中間言語を使わずに，既存の中間言語を用いれば新たに作成する必要はなく，工夫をすれば工数を抑えることは可能であると考えられる．

（f2）M に基づく中間コード P’ の作成

この工程はコンパイラが行っていることと同じであるから，中間言語があれば，それをもとにソースコードを参照し，自動で中間コードを作ることは可能である．よって，改変工数を抑えることは可能であると考えられる．

（f3）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

中間言語ができていれば，それをもとに自動でインタプリタを生成することは容易であると考えられる．また，既存の中間言語であればそのインタプリタを用いればよい．よって工数を抑えることは可能であると考えられる．

（f4）P’ と I を組み合わせたプログラム P” の作成

生成された P’ と I を組み合わせるだけであるため，自動で組み合わせることも可能であると考えられ，工数はほとんどかからないと考えられる．

（f5）T の改変

書き換え対象を機械語から中間コードへと改変するだけであり，概念の整理であるため，工数はほとんどかからないと考えられる．

（f6）P” への T’ の適用

中間言語に中間コードを書き換える命令を挿入することは（f1）を行う際に考慮するだけでよく，容易である．また，カムフラージュの対象となる命令や自己書き換えルーチンを自動で挿入するシステムは，すでに RINRUN という前例がある．そのため，この工程においても自動化は可能であり，改変工数を抑えることは可能であると考えられる．

以上より，提案方式を適用する工程の大部分は自動化が可能であると考えられ，自動で提案方式を適用するようなツールを作成することで低い改変工数で適用が可能である．

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，20 kByte のファイルの SHA1 のハッシュ値をと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．よって既存方式プログラムの命令カムフラージュ数 0 回は素の耐タンパー化対象プログラムの結果となる．また，3.4 節で説明したように，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．よって，提案方式プログラムの命令カムフラージュ数 0 回は難読化手法の Simulation を適用し，中間コードとインタプリタの構成にしたプログラムの結果となる．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き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 3.24 GByte，CPU が Intel Core 2 Duo Processor E8400（クロック周波数 3 GHz，1 次データ・キャッシュ 64 kByte，2 次キャッシュ 6 MByte）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 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，書き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き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．De-

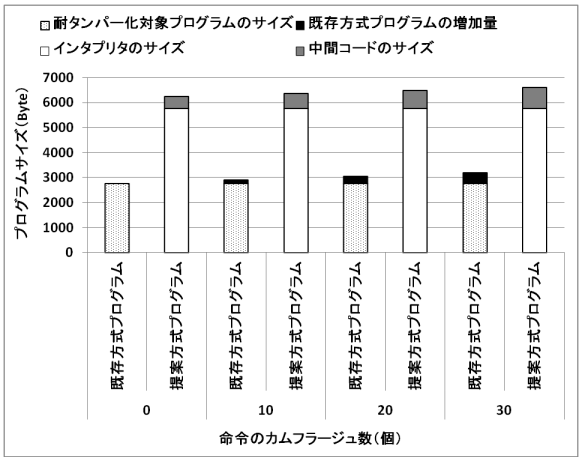

図 6 ファイルサイズの変化
Fig. 6 Difference of the file size.

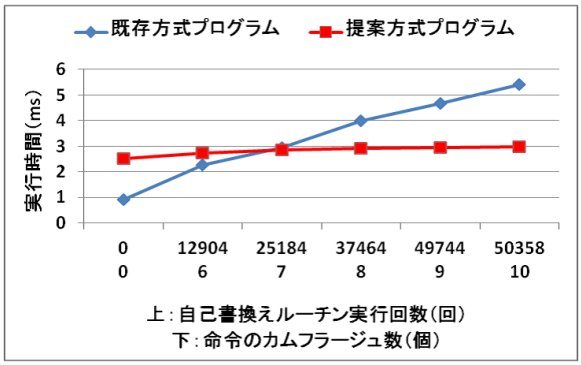

図 7 パソコン上での実行時間の変化
Fig. 7 Difference of the file speed on a PC.

bug モードで実行させ，書き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き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き換えられていた．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．

4.2.1 プログラムサイズ

プログラムサイズは，図 4 で示されたもののうち，既存方式プログラムは機械語プログラムのみを，提案方式プログラムはインタプリタと中間コードの合計値をプログラムサイズとして計測する．変数や定数には手を加えないため，今回は計測対象としなかった．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．

4.2.2 実行時間

フォンノイマン・アーキテクチャであるパソコン上で実行し，実行時間を得た．既存方式プログラムでも自己書き換えが正しく行われ，既存方式プログラムと提案方式プログラムのフローは同一であった．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．実行時間は自己書き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き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），横軸が自己書き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．

4.2.3 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．命令カムフラージュ数が 1 つ増えるとその分自己書き換えルーチン分のプログラムサイズが増える．KMNM 法は 1〜4 byte を書き換えるルーチンで 7〜10 byte，KMNM 法を用いた提案方式では 3.4 節式 (3) で示したように 3〜12 byte 増加する．1 つの自己書き換えルーチンが書き換える byte 数の平均をとると KMNM 法と KMNM 法を用いた提案方式では，命令のカムフラージュ数に対するプログラムサイズの増加量はあまり変わらないため，実験で得られた差は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，提案方式のプログラム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．中間言語の抽象度を変化させることでプログラムサイズが変わると考えられるが，抽象度が高いと中間コードのサイズが大きくなり，抽象度が低いとインタプリタのサイズが大きくなるので，それらの均衡をとり，プログラムサイズを小さくできる中間言語の抽出法を考案することは今後の課題となる．

実行時間は KMNM 法と比べて，差異が得られた．素の耐タンパー化対象プログラムの実行時間は，約 1 ms であり，提案方式を用いると 2.5 倍〜3 倍程度増加しており，既存方式を用いると 2 倍〜5 倍程度の増加となった．自己書き換えルーチン実行回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とき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，自己書き換えルーチン実行回数が多い場合には，KMNM 法は 2.3 節で示したように命令メモリを書き換えたことによる命令キャッシュや命令プリフェッチの無効化が発生するため，実行時間の増加量が大きくなり，対して提案方式の KMNM' 法では命令メモリを書き換えないため実行時間の増加が抑えられ，自己書き換えルーチン実行回数 25,184 回でほぼ同等の実行時間になり，それより多い場合には提案方式の実行時間が少なくなるといった結果になった．既存の自己書換え型耐タンパー技術では自己書き換えによるオーバヘッドが大きいために自己書き換えルーチンを挿入する箇所を吟味しループ文内には挿入しないなど，自己書き換え回数が少

なくなるように選ぶ必要があった．しかし，提案手法では自己書き換え回数によるオーバヘッドが少ないため，自己書き換えルーチンの挿入箇所を吟味する必要がない．これにより求められる挿入アルゴリズムが既存の自己書換え型耐タンパー技術よりも簡単になるため，自動化ツールを作りやすくなるという利点が考えられる．

4.2.4 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．文献 [1] において，静的解析が成功する確率を定式化しており，攻撃者がカムフラージュ数 n 個であるプログラム M（プログラムサイズ L）に対して，長さ m の任意のコードブロックを正しく理解できる確率を

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

と与式している．今回の提案方式では $L = 485$ であり，$m = 100$ とし，n を変化させると**図 8** のようになる．閾値にもよるが，0.2 以下で抑えたい場合は 10 個，0.01 以下で抑えたい場合には 26 個のカムフラージュを行えば十分な耐タンパー性があると考えられる．

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．

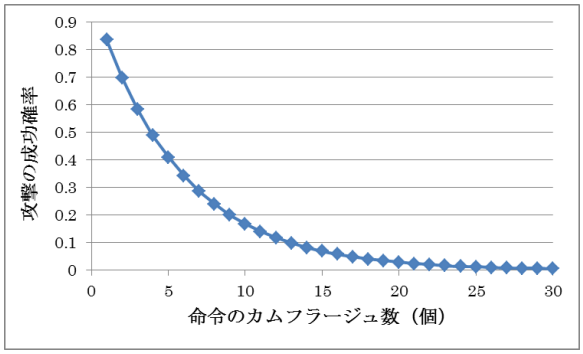

図 8 静的解析が成功する確率（$L = 485$，$m = 100$）
Fig. 8 Probability of success of code analysis.

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．

たと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き換えを余儀なくされる．逆に，抽象度が低く命令が 1 カ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十分には解明されていない．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は今後の課題となる．

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，および，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あげられる．

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 (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 (2011).
[3] Chang, H. and Atallah, M.J.: 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 (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，EE Times（オンライン），入手先 ⟨http://eetimes.jp/ee/articles/1203/23/news032.html⟩（参照 2013-06-05）．
[5] Atmel: ATmega32A Complete, Atmel (online), available from ⟨http://www.atmel.com/Images/doc8155.pdf⟩ (accessed 2013-06-05).
[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，インテル（オンライン），入手先 ⟨http://www.intel.co.jp/content/dam/www/public/

ijkk/jp/ja/documents/developer/IA32_Arh_Dev_Man_Vol3_i.pdf〉（参照 2013-06-05）.
[7] Cohen, F.B.: Operating System Protection through Program Evolution, *Computer Security*, Vol.12, No.6, pp.565–584 (1993).
[8] RFC: US Secure Hash Algorithm 1 (SHA1), RFC (online), available from 〈http://datatracker.ietf.org/doc/rfc3174/〉 (accessed 2013-06-05).
[9] 神崎雄一郎：RINRUN：プログラムカムフラージュ化ツール，奈良先端科学技術大学院大学（オンライン），入手先 〈http://se.naist.jp/rinrun/〉（参照 2013-06-05）.
[10] Atmel: EVK1104, Atmel (online), available from 〈http://www.atmel.com/tools/EVK1104.aspx〉 (accessed 2013-06-05).
[11] Birrer, B.D., Raines, R.A., Baldwin, R.O., Mullins, B.E. and Bennington, R.W.: Program Fragmentation as a Metamorphic Software Protection, *3rd International Symposium on Information Assurance and Security, IAS 2007*, pp.369–374 (2007).
[12] You, I. and Yim, K.: Malware Obfuscation Techniques: A Brief Survey, *International Conference on Broadband, Wireless Computing, Communication and Applications* (*BWCCA 2010*), pp.297–300 (2010).
[13] Li, X., Loh, P.K.K. and Tan, F.: Mechanisms of Polymorphic and Metamorphic Viruses, *European Intelligence and Security Informatics Conference* (*EISIC 2011*), pp.149–154 (2011).
[14] Collberg, C. and Nagra, J. (Eds.): *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, pp.201–299, Addison Wesley (2009).

推薦文

本論文では，プログラム実行時に，データメモリに可変な中間コードとして振る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を作成している．これにより，プログラム実行時に命令の一部を書き換えるという既存の KMNM 法や OM 法を，プログラムを書き換えられない組み込み系の機器においても適用したという点が高く評価できる．また，背景の説明や現状の問題点に関する指摘，解決案の提案と実装による評価など，論文の構成や研究内容が非常に優れているので，推薦したい．

（コンピュータセキュリティシンポジウム 2012
プログラム委員長　高木　剛）

吉田 直樹

1990 年生．2009 年 4 月横浜国立大学工学部電子情報工学科入学．現在，同大学院環境情報学府（情報メディア環境学専攻）博士課程前期在学中．電子情報通信学会学生会員．

吉岡 克成　（正会員）

2005 年 3 月横浜国立大学大学院環境情報学府情報メディア環境学専攻博士課程後期修了，博士（工学）．同年 4 月独立行政法人情報通信研究機構研究員．2007 年 12 月より横浜国立大学学際プロジェクト研究センター特任教員（助教）．2011 年 4 月より横浜国立大学大学院環境情報研究院准教授．マルウェア解析やネットワーク攻撃観測・検知等のネットワークセキュリティの研究に従事．2009 年文部科学大臣表彰・科学技術賞（研究部門）受賞．

松本 勉

1986 年 3 月東京大学大学院工学系研究科電子工学専攻博士課程修了，工学博士．同年 4 月横浜国立大学講師．2001 年 4 月より同大学院環境情報研究院教授．2007 年 4 月～2011 年 3 月同大学教育研究評議員を兼務．2011 年 4 月～2013 年 3 月同大学理工学部副学部長を兼務．日本学術会議連携会員．暗号アルゴリズム・プロトコル，耐タンパー技術，生体認証，人工物メトリクス等の「情報・物理セキュリティ」の研究教育に 1981 年より従事．1982 年にオープンな学術的暗号研究を目指した「明るい暗号研究会」を 4 名で創設．2005～2010 年国際暗号学会 IACR 理事．1994 年第 32 回電子情報通信学会業績賞，2006 年第 5 回ドコモ・モバイル・サイエンス賞，2008 年第 4 回情報セキュリティ文化賞，2010 年文部科学大臣表彰・科学技術賞（研究部門）各受賞．