B-009

実行時間差に着目したコードの隠ぺい方法

A Method for Hiding Program Code Focused on the Execution Time Difference

神崎 雄一郎 †　　門田 暁人 ‡
Yuichiro Kanzaki　　Akito Monden

1. はじめに

ソフトウェアには，商業的価値の高いアルゴリズムやライセンスチェックに用いられる条件判定式など，ユーザに知られたくない秘密情報が含まれている場合がある．秘密情報がユーザに漏えいすることでソフトウェアベンダが大きな損失を受けた例は数多く報告されており [6]，ソフトウェア保護方法，すなわち，ソフトウェア内部の情報を攻撃者 (悪意のあるユーザ) の解析行為から保護するための方法が求められている．

ソフトウェア保護方法は，プログラムの難読化やプログラムの暗号化を始めとして多数提案されている [1, 2]. 従来提案されている方法の多くは静的解析 (解析対象を実行させずに行う解析) から保護することを目的としたものであるが，デバッガなどの動的解析 (解析対象を実行させながら行う解析) のためのツールをユーザが容易に入手できる昨今では，静的解析と動的解析を組み合わせた攻撃が主流となっている [5] ため，動的解析に対して有効な方法も強く求められている．

そこで本論文では，動的解析に対して有効なソフトウェア保護方法として，実行時間差を用いたコードの隠ぺい方法を提案する．ここで実行時間差とは，動的解析を行った時に生じるプログラムの実行時間のオーバーヘッド (通常実行時の実行時間との差) のことである．また，コードを隠ぺいするとは，攻撃者がコード (命令列やデータ列) の内容を取得するのを極めて困難にするという意味である．

隠ぺいされたコードは偽 (にせ) の内容で上書きされており，プログラムが実行されるとき，プログラムの一部の実行時間 (実行クロック数) に応じて自己書換えされる．デバッガのブレイクポイント機能を用いた場合など，一時停止を伴う動的解析が行われることでその実行時間が通常実行よりも長くなった場合 (あるいは短くなった場合) は，隠ぺいされたコードは元来のものと異なる内容で書き換えられることとなり，元来のコードが攻撃者に露呈しない．

以降，2.では，提案方法の基本アイデアについて説明する．3.では，提案方法によってコードを隠ぺいする手順について述べる．4.では，提案方法によって保護されたプログラムの例を示す．5.では，提案方法によって保護されたプログラムに対する攻撃およびその困難さに関する考察を行う．最後に 6.において，結論と今後の課題を述べる．

† 熊本電波工業高等専門学校 情報工学科，Dept. of Information and Computer Sciences, Kumamoto National College of Technology
‡ 奈良先端科学技術大学院大学 情報科学研究科，Graduate School of Information Science，Nara Institute of Science and Technology

2. 基本アイデア

図 1 に従い，提案方法の基本アイデアを説明する．図 1 (a) および (b) は，元来のプログラム P および保護された状態のプログラム P_p の概念図をそれぞれアセンブリレベルで示している．ここでは説明のための例として Intel x86 系 CPU を想定し，AT&T 文法によって表している．

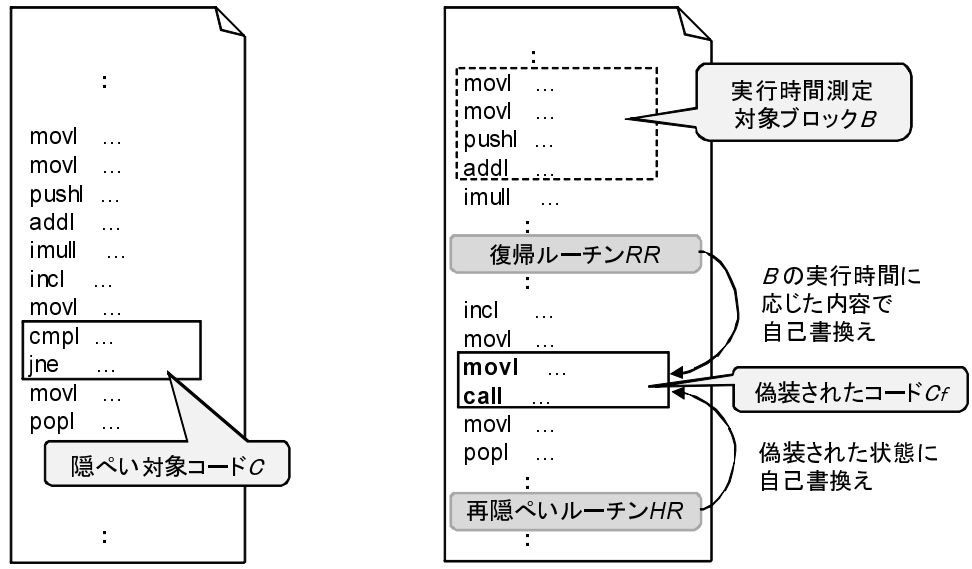
元来のプログラムの一部は，隠ぺい対象のコードとして選択され，別のコードで偽装 (上書き) される．図 1 の例では，元来のプログラムに含まれる `cmpl` および `jne` という命令が，保護されたプログラムでは `movl` および `call` という命令に偽装されている．偽装されたコード C_f は，復帰ルーチン *RR* および再隠ぺいルーチン *HR* により自己書換えされる，すなわち，実行時に書き換えられる．*RR* は，保護されたプログラムに含まれる実行時間測定対象ブロック B の実行時間が一定の範囲内である場合は，元来のコードで書き換え，範囲外である場合は，元来とは異なるコードで書き換える．一方 *HR* は，*RR* によって書き換えられたコードを再び偽装されたコードに書き換える．C_f は，*RR* が実行されてから *HR* が実行される間の期間のみ，C となる．ただし，B の実行時間が不自然に長い場合あるいは短い場合は，C_f は C とは異なるコードに書き換わる (C のコードはメモリ上に現われない)．攻撃者が動的解析によって C のコードを知るには，B を一時停止させずに通常実行時と同程度の速度で通過し，*RR* と *HR* に挟まれたコード内に実行の制御を移す必要がある．このような仕組みを連鎖的に何重にも適用することにより，攻撃者が C を知るためのコストを高くする．

3. コードの隠ぺい手順

保護対象のアセンブリプログラム P に対して，以下の Step 1 から Step 6 を繰り返し適用することによって，保護されたアセンブリプログラム P_p が得られる．

(Step 1) 隠ぺい対象コードの決定

まず，隠ぺい対象のコード C を決定する．提案方法のユーザが，攻撃者に知られたくない任意のコードを隠ぺい対象として選定する．



(a) 元来のプログラムP　　(b) 保護されたプログラムP_p

図 1　基本アイデア

隠ぺい対象とすべきコードは，P の性質によって異なるが，攻撃者が特徴のある命令を手がかりに解析範囲を絞り込もうとする場合があることを考慮すると，DRM システムに含まれる秘密鍵などの特定の定数，パスワードの入力結果等に基づいて処理を分岐する命令といった秘密情報そのものに加えて，正規の製品かどうかを確認するために CD/DVD ドライブにアクセスする (割込み) 命令や，パスワードの入力を促すため等に用いられるダイアログボックスを呼び出す命令など，攻撃者が秘密情報を見つける手がかりとなるコードも隠ぺい対象とすることが望まれる．

また，2 回目以降の適用において，復帰ルーチン，再隠ぺいルーチン，実行時間を測定する命令などを隠ぺい対象にすると，保護機構の存在を隠ぺいできるため，最初に隠ぺいしたコードを攻撃者が知るためのコストを高くできる．

(Step 2) 偽装コードの生成と上書き

C を偽装するコード C_f を生成し，C に C_f を上書きする．C_f はランダムに生成することもできるが，コードの文脈や出現頻度を参考にして，偽装されたコードの発見が困難になるように考慮することが望ましい．

(Step 3) 実行時間測定対象ブロックと復帰ルーチン・再隠ぺいルーチンの挿入位置の決定

実行時間測定対象ブロック B，および，復帰ルーチン RR と再隠ぺいルーチン HR の挿入位置を決定する．以降，RR の挿入位置および HR の挿入位置をそれぞれ $P(RR)$ および $P(HR)$ と記す．

命令のカムフラージュ法 [4] の適用手順における Step 1 で示されているものと同様の方法で，P の制御の流れを考慮して B，RR，(RR によって自己書換えされた)C_f，HR の順に実行されることが保障されるように各位置を決定する．具体的には，P の一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，以下の条件をすべて満たすように，B，$P(RR)$ および $P(HR)$ を決定する．

1. B は基本ブロックであり，B から制御の流れが分岐することはない．
2. プログラム開始点 *start* から $P(RR)$ に至るすべてのパスに B が存在する．
3. B から C_f に至るすべてのパスに $P(RR)$ が存在する．
4. $P(RR)$ から C_f に至るすべてのパスに $P(HR)$ が存在しない．
5. $P(HR)$ から C_f に至るすべてのパスに $P(RR)$ が存在する．
6. C_f からプログラム終了点 *end* に至るすべてのパスに $P(HR)$ が存在する．

(Step 4) 実行時間測定命令の挿入

B の実行時間を測定するための命令を挿入する．具体的な方法としては，プロセッサのクロック数を測定する命令 (Intel IA-32 アーキテクチャにおける RDTSC 命令 [8] など) を用いて，B に要したクロック数を測定する方法が挙げられる．

(Step 5) 判定基準となる実行時間の決定

B の構成内容に基づき，動的解析が行われているかどうかを判断するための実行時間の基準を定める．実行時

```
          :
01:  rdtsc
02:  movl  %edx, CLOCK
          :
  （実行時間測定対象ブロック B）
          :
03:  rdtsc
04:  subl  (CLOCK), %edx
          :
05:  addl  $0x38, %edx
06:  movb  %dl, TARGET1              # RR
07:  movl  -8(%ebp), %eax
08:  imull  -8(%ebp), %eax
09:  sall  $2, %eax
   TARGET1:
10:  movl  -4(%ebp), %eax            # Cf
   （元来のコード C は cmpl -4(%ebp),%eax）
11:  jne  L3
12:  call  _success
          :
13:  movb  $0x8B, TARGET1            # HR
          :
```

図 2　保護されたプログラムの例 (一部)

間は実行に消費したクロック数 (実行クロック数) で表す．具体的には，最小実行クロック数 T_{min} 以上，かつ，最大実行クロック数 T_{max} 未満であれば通常の実行であると判定し，そうでなければ動的解析 (あるいは B の改ざん等) が行われていると判断する．

T_{min} および T_{max} は B を構成する命令列から概算するか，ソフトウェアが実行される対象となる環境の上で実際に B の実行クロック数を測定し，その測定値を参考に決定する．

(Step 6) 復帰ルーチン・再隠ぺいルーチンの生成と挿入

次の手順に従い，復帰ルーチン *RR* および再隠ぺいルーチン *HR* を生成する．ここで，B の実行クロック数を T_B とする．

1. $T_{min} \leq T_B < T_{max}$ であれば，C_f を C に，そうでなければ C_f を C とは異なる命令に書き換えるための (一連の) 命令を作り，それを *RR* とする．
2. C を C_f に書き換えるための (一連の) 命令を作り，それを *HR* とする．

自己書き換えを行うルーチンは，mov 命令等の出現頻度の高い命令で記述できる [4]．生成された *RR* および *HR* は，Step 3 で決定された位置にそれぞれ挿入される．

4. 保護されたプログラムの例

図 2 に，提案方法によって保護されたアセンブリプログラムの例 (一部) を示す．このプログラムには，簡単なパスワードのチェックルーチンが含まれている．各行の左端に記してある番号は説明のための行番号である．

元来のプログラムでは 10 行目の位置に存在した，パスワードのチェックに必要な比較命令である cmpl 命令 (オペコードのバイナリ表現は 3B) が隠ぺい対象 C として選択され，C_f として生成された movl 命令 (オペコードのバイナリ表現は 8B) で偽装されている．6 行目の復帰ルーチン *RR*(movb 命令 1 つで構成されている) は，1〜4 行目において測定された B(プログラムの一部) の実行時間に応じて，10 行目の命令を書き換える．また，13 行目の再隠ぺいルーチン *HR* は，10 行目の命令が実行された後に，その命令を再び C_f である movl 命令に書き換える．

実行時間の測定には RDTSC 命令 [8] が用いられている．B の直前 (1 行目) と B の直後 (3 行目) において各時点における実行クロック数を RDTSC 命令で測定しておき，その差分を求めることによって B に要した実行時間 (実行クロック数) を取得している．

movl 命令 (C_f) を元来の cmpl 命令 (C) にするためには，movl 命令のオペコードの部分 (1 バイト目) を 3B に書き換えればよい．5 行目の addl 命令において，書き換えるコードの値を B の実行時間に基づいて計算しているが，B の実行時間が長い場合は書き換える値が大きくなり，cmpl 命令とは異なる命令に書き換わる．また，B の実行クロック数が短くなるように *RR* や B を改ざんしても，その実行クロック数が判定基準の範囲外である限り，C_f は元来の命令に書き換わることはない．

5. 解析の困難さに関する考察

ここでは，提案方法によって保護されたプログラムに対する攻撃およびその困難さに関する考察を行う．

まず，攻撃者が保護されたプログラムに対して静的解析を試みた場合について考察する．静的解析とは，プログラムを実行させずに行う解析のことである．一般的には逆アセンブラなどのツールを用いてアセンブリレベルのプログラムを取得し，プログラムの流れを解析する．

保護されたプログラムにおいて，隠ぺいされたコードは別の命令で偽装されているため，元来のコードの値はプログラム内に存在しない．そのため，逆アセンブルされたプログラムに対して単純に特定の種類の命令を検索しても，攻撃者はそれらの命令を見つけることができない．例えば，パスワードのチェックルーチンを見つけるために比較命令や分岐命令を攻撃者が検索しても，それらが隠ぺいされている場合，見つけることができない．また，ドライブへのアクセス命令やダイアログボックスを表示する命令といった，秘密情報を見つけるための手がかりとなるような命令が隠ぺいされている場合，効率の良い解析を行うために解析の範囲を絞り込むことも困難になる．

攻撃者が隠ぺいされたコードの元来の内容を静的解析によって知るには，mov 命令等のありふれた (一般的にプログラム中に多く存在する) 命令で構成された復帰ルーチンをプログラム全体から探し出し，書き換え先のアドレスと書き換え内容を計算する必要がある．書き換

え内容の計算には，実行時間測定対象ブロックの実行時間を推測する必要があり，非常に高いコストを要する．

次に，攻撃者が動的解析を試みた場合について考察する．動的解析とは，プログラムを実行させながら行う解析である．デバッガなどのツールを用いてメモリ，レジスタの値や出力の変化をプログラムを実行させながら観測する．多くのデバッガは指定された位置でプログラムを実行を一時停止させるブレイクポイント機能や，実行を一命令ずつ停止させながら実行させるステップ実行機能を備えている．カーネルモードで解析を行うデバッガも存在するが，操作が複雑で扱いが難しいため，OllyDbg や IDA Pro などの通常のデバッガによる解析 [3, 7] が未だよく行われている．

攻撃者がデバッガ等により，保護されたプログラムの B において一時停止を伴う解析を行った場合，例えば，デバッガのブレイクポイント機能を用いて B 中の命令を実行中に一時停止させたり，ステップ実行機能により B を一命令ずつ実行させたりした場合，停止させた時間に応じて T_B が増加する．T_B が T_{max} を上回った場合，後に実行される RR において C_f は元来のコードである C とは異なるコードに書き換わり，攻撃者は C の内容を知ることができない．また，RR や B を改ざんして，T_B が短くなるように見せかけても，T_B が T_{min} 未満である場合は，正しく書き換わらない．

攻撃者が動的解析によって C の内容を知るには，B を一時停止させずに通過し，RR と HR に挟まれたコード内に実行の制御を移す必要がある．RR と HR に挟まれたコードが，別の隠ぺい対象についての実行時間測定対象ブロックとなっているなど，連鎖的にコードの隠ぺいが適用されている場合，攻撃者が C を知るためのコストはより高くなる．

6. おわりに

本論文では，動的解析に対して有効なソフトウェア保護方法として，実行時間差，すなわち，動的解析を行った時に生じるプログラムの実行時間のオーバーヘッドを利用したコードの隠ぺい方法を提案した．攻撃者が隠ぺいされたコードの元来の内容を静的解析によって知るには，プログラムに散在する復帰ルーチンの書き換え先のアドレスと書き換え内容を計算する必要がある．その計算にはプログラムの一部の実行時間を推測する必要があり，大きなコストを要する．また，動的解析によって元来のコードを知るには，限られた実行の期間に一時停止させずに到達する必要がある．連鎖的にコードの隠ぺいが適用されている場合は特に，大きなコストを要する．

今後の課題として，動的解析を伴った実行かどうかを判断する基準値の計算方法や，文脈を考慮したコードの偽装方法について詳細に検討することが挙げられる．

謝辞

本研究は，日本学術振興会 科学研究費補助金 若手研究 (B)，課題番号 20700034 の助成を受けたものである．

参考文献

[1] C. Collberg, and C. Thomborson, “Watermarking, tamper-proofing, and obfuscation – Tools for software protection,” IEEE Transactions on Software Engineering, vol.28, no.8, pp.735-746, June 2002.

[2] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” Technical Report 148, Technical Report of Dept. of Computer Science, U. of Auckland, New Zealand, 1997.

[3] C. Eagle, The IDA Pro Book: The Unofficial Guide to the World’s Most Popular Disassembler, No Starch Press, 2008.

[4] 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，vol.J87-A，no.6，pp.755-767，June 2004.

[5] M. Madou, B. Anckaert, B.D. Sutter, and K.D. Bosschere, “Hybrid static-dynamic attacks against software protection mechanisms,” Proc. The Fifth ACM Workshop on Digital Rights Management(DRM2005), Nov. 2005.

[6] 門田暁人，C. Thomborson，“ソフトウェアプロテクションの技術動向 (前編) - ソフトウェア単体での 耐タンパー化技術,” 情報処理，vol.46，no.4，pp.431-437，April 2005.

[7] V. Pirogov, Disassembling Code: IDA Pro And SoftICE, A-List Publishing, 2005.

[8] IA-32 Intel Architecture software developer’s manual vol.2 : Instruction Set Reference, Intel Co.http://www.intel.co.jp/ (Available online).