制御構造難読化の一手法に対する検討
-いくつかの改善とその実験的評価-

2004MT055 牧野 英雄　　2004MT115 八木 春樹
指導教員　真野 芳久

1　はじめに

悪意ある使用者が、他人の開発したプログラムを開発者の許可なく利用するプログラムの無断盗用によりプログラムの正規の開発者が利益が得られないという問題が起こっている。この問題をソフトウェアプロテクション技術で解決しようとしている。

ソフトウェアプロテクション技術の一つにプログラムの実行結果を保ったまま、プログラムの解析を困難にする難読化が存在する。難読化は、一般的に、時間やメモリのコストを増大させるなどの欠点もいくつかもっている。プログラムの正規の開発者の利益を守るために、新たなソフトウェアプロテクション技術の開発、既存のソフトウェアプロテクション技術の発展が求められている。

2　難読化

[1] と [2] を参考に、本研究での難読化の定義を定める。

定義：ある言語で書かれたプログラム P と P に関する命題 Q が与えられたとする。そのときに、同一の言語で書かれたプログラム P' を次の 2 条件を同時に満たすように導くことを、Q に関して P を難読化すると言う。

- 仕様の保存：任意の入力について、P' は P と同一の出力を返す。
- 解析の困難さの増加：P' は Q に関する解析に P よりも時間がかかる。

難読化の諸手法には、レイアウト難読化、データ構造難読化、制御構造難読化の 3 つが挙げられる。制御構造難読化は、 ループ文や switch 文などの制御構造を変更することで処理の見た目を隠したり、複雑にする手法である。

攻撃者のプログラムへの攻撃 (解析) に対する強度を耐性と呼ぶ。無制限に時間を費せば解析できないとは限らないので、 現実的に、不可能に近い程度に解析を困難にすることを目標とする [**1**]。

3　制御構造難読化諸手法に関する関連研究 (switch 文を用いた Control Flow Scramble)

制御構造難読化の代表的な論文である [**3**] の概要を述べる。プログラムの CFG を解析されないように、高級な構造 (high-level control) を 2 ステップで変更する。

1. 高級な構造を仕様が保存されるように、if-then-goto 構造に変更する (if-then-goto 構造とは、if 文、goto 文のみ使用する文構造)。ブロックの分割の仕方は、分岐が出発するところと終了するところで分割する。
2. 行き先を動的に決定するために、goto 文の代わりに、switch 文と switch 変数を使って、次のブロックを決定するために、実行する。

このような変換に対して、攻撃者は switch 変数の値を解析することで、プログラムの制御構造を解析しようとする。そこで、攻撃者に switch 変数の値を解析されないための手法を以下に示す。

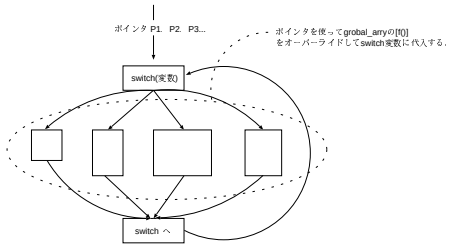

図 1　switch 文を用いた制御構造難読化を攻撃者から解析されないようにする手法 [**3**]

攻撃者は、制御が移る前の場所から調べることで、switch 変数に代入される値を知ることができる。そこで、switch 変数に代入される値を、あらかじめ用意してある配列 globalarray から参照する。このとき、配列の添字値を攻撃者に知られないように、配列の添字値は関数を使って求めるものとする。さらに、ポインタをエイリアスとして使って、globalarray の中身を上書きしたり、参照し終わった配列の中身を攻撃者に解析されないように配列を上書きしている。

このような手法を行うことで、攻撃者に switch 変数の値を解析されることを、防ぐことができる。

4　2 プロセス方式による Control Flow Scramble

制御構造難読化の中で、別のプロセス内に制御情報を隠す方式 [**4**] の概要を示す。私達は [**4**] をもとに研究を行う。

4.1　難読化手法

2 プロセス方式による Control Flow Scramble では、プログラムを P-Process と M-Process と呼ばれる 2 つのプロセスに分ける。P-Process がプログラムの主な実行内容のまとまりで、M-Process が P-Process の実行順序を制御する。この 2 つのプロセスでプロセス間通信を行う。P-Process は M-Process に制御移動先のアドレスを要求して、M-Process は必要なアドレスを

P-Process に返す。この手法は P-Process の制御のためのアドレス情報を M-Process 内に隠すことで難読化を行っている。

4.2 P-Process

P-Process は難読化前のプログラムの大部分を占める。このプロセスでは Hot Node と呼ばれる概念を用いている。P-process ではプログラム全体をいくつかのまとまったブロックに分割するが、その分割したブロック 1 つ 1 つのことを Hot Node と呼ぶ。Hot Node は 1 つ以上の隣り合った基本ブロックのまとまりで、Hot Node 単位で並び替えを行い、静的なレイアウトを難読化する。実行は Hot Node 単位で行われるが、このままでは実行順序まで分からなくなるため、M-Process にアクセスすることで、正しい順序でプログラムを実行する。図 2 に P-process 内の制御の流れを示す。

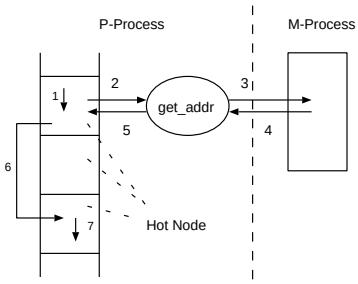

図 2　P-Process の制御の流れ

図 2 中の番号 2、3、4、5 は get_addr 関数を通して、M-Process にアドレスを要求し、取得していることを示している。そして、番号 6、7 で得られたアドレスの Hot Node に移動し、実行を続行する。

4.3 M-Process

M-Process では P-process の実行順序を制御する。P-Process と比べると小さいプログラムである。M-Process は Cell と呼ばれる、いくつかのまとまったブロックに分割する。Cell は 1 つ以上の基本ブロックのまとまりで、実行は Cell 単位で行われる。本手法では P-Process のアドレス情報を隠すことが重要となっているため、全ての Cell に暗号化を施す。この暗号化の方法については 4.4 節で説明する。

図 3 に M-Process の実行の様子について示す。plain-text は復号されている Cell で、Encrypted は暗号化されている Cell であることを示している。xorAll は M-Process にある関数で、4.4 節で説明する暗号化方法を実現するために用意したものである。図 3 中の番号 2 で xorAll を呼び出すと、Cell 全体に暗号鍵を適用する。番号 4 で、右は適用後を示している。Cell$i$ は暗号化、Cell$j$ が復号されていることが分かる。その後、番号 5 で Cell$j$ に制御が移る。

4.4 M-Process コードの暗号化

M-Process は耐性を高めるために Cell 単位で暗号化をする。暗号鍵は全ての Cell 間で生成される。始めに

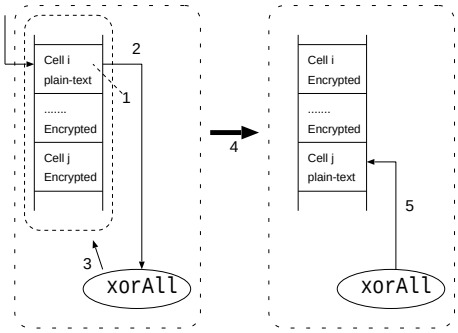

図 3　M-Process の実行の様子

実行される Cell を Cell0 とすると、Cell0 と他の全ての Cell との組み合わせの暗号鍵は乱数で生成される。そして、Cell0 を除いた Cell の組み合わせの暗号鍵は計算式 (1) に当てはめて生成する [4]。

$$k_{ab} = k_a \bigoplus k_b \tag{1}$$

k_{ab} : $Cell a$と $Cell b$ の間用の暗号鍵。
　　k_a、k_bの XOR（排他的論理和）で生成される。
k_a, k_b : それぞれ $Cell0$ と $Cell a$、$Cell0$ と $Cell b$ の間用の暗号鍵。ランダムに生成される。

全ての Cell に特定の暗号鍵を適用することで、次に実行する Cell のみを復号できる。暗号鍵を適用したときの変化は、次に実行する Cell、現在実行中の Cell、その他の Cell の 3 通りに分類される。この変化が正しく行われることを示す。

それぞれの Cell を順に Cell$t$、Cell$s$、Cell$i$ とすると、適用される暗号鍵は Cell$s$ と Cell$t$ の間で生成されたものとなる。Cell$t$ は現在、Cell$s$ と Cell$t$ の間で生成された暗号鍵によって暗号化されている (帰納法の仮定) ので、暗号鍵を適用すると復号される。Cell$s$ は復号された状態なので、暗号鍵を適用することで暗号化される。Cell$i$ は現在、Cell$i$ と Cell$s$ の間で生成された暗号鍵によって暗号化されている。(1) 式に当てはめると、Cell$i$ に適用されている暗号鍵は、Cell$i$ と Cell$t$ の間で生成された暗号鍵に変わる。この変化の繰り返しにより、実行中の Cell のみを復号した状態にすることができる。

5　2 プロセス方式における問題点

5.1 プロセス間通信による実行時間のオーバーヘッド

この難読化は P-Process と M-Process の 2 つでプロセス間通信を行う。プロセス間通信は 1 つのプロセス内での通信に比べて、データの受け渡しに時間がかかる。表 1 は難読化前と難読化後の実行時間の変化をまとめたものである。

難読化前と難読化後の実行時間を比べてみると、難読化後の実行時間が長くなっていることが分かる。これは問題点に挙げられる。

表 1　難読化による実行時間のオーバーヘッド [4]

ファイル名	ソースプログラム			難読化後		
	real	user	sys	real	user	sys
tsort	4.90	0.29	0.05	9.85	1.60	1.05
compress42	2.05	0.42	0.52	10.31	2.31	2.77

5.2　Hot Node の呼び出しに関する問題点

P-process は M-Process にアドレスを要求して実行順序を制御するが、プログラムによっては 1 つの Hot Node に複数回、制御が移ることがあるかもしれない。同じ Hot Node が何度も実行されると、P プロセスから M プロセスへの要求に対して、同じパラメータを渡すので、解析されやすくなる。

5.3　暗号化と復号による実行時間のオーバーヘッド

M-Process では暗号化と復号を行うために、暗号鍵を使用して操作を行うが、ある Cell から別の Cell に制御が移るたびに、全ての Cell に対して暗号鍵を適用する。この操作により攻撃に対する耐性を高めているが、全ての Cell に対して適用されるため、実行時間に影響が出る。表 2 は暗号化の有無による実行時間の変化をまとめたものである。

表 2　暗号化による実行時間のオーバーヘッド [4]

ファイル名	暗号化なし			暗号化あり		
	real	user	sys	real	user	sys
tsort	9.85	1.60	1.05	18.45	1.61	1.12
compress42	10.31	2.31	2.77	20.22	3.30	3.10

暗号化ありの場合は暗号化なしに比べて実行時間が長くなっているのが分かる。これは問題点に挙げられる。

5.4　fork 関数使用によるプロセス作成時の問題点

プロセス作成の方法の一つに fork 関数がある。[4] では fork 関数を使用して 2 プロセスを実現しているが、この関数を使用すると子プロセス作成のための時間とメモリが必要となる。現在のプロセスとは別に、新しいプロセスを作成するため、メモリ消費量が多くなる。fork によって作られたプログラムのほとんどは使われないので、メモリを無駄に使用することになる。

6　問題点改善のための検討

6.1　プロセス間通信による実行時間のオーバーヘッドに対する検討

2 プロセス方式の適用において重要なことは、いかに耐性を高めて、実行効率を上げるかということである。プロセス間通信の回数を減らせば、実行時間のオーバーヘッドを緩和することができる。プロセス間通信は実行中の Hot Node から別の Hot Node に制御を移す時に行われる。Hot Node の数を減らせば、プロセス間通信の回数は減る。しかし、1 つ 1 つの Hot Node のサイズが大きくなり、解析されやすくなるという問題点がある。

このほかに制御方法を変更する案として、Hot Node の数はそのままで、制御を移すポイントを減らすことで、プロセス間通信を減らすというものがある。この方法は制御の流れを簡略化することになるので、制御関係が分かりやすくなるという問題点がある。

また、プロセス間通信を用いずにプログラムを構成するという案もある。具体的な手法としてはマルチスレッド、コルーチンを使用する。マルチスレッドやコルーチンを使用したプログラムでもマルチプロセスとほぼ同じ働きをさせることができる。マルチプロセスに比べて、マルチスレッドやコルーチンは動作が軽快なため、実行時間のオーバーヘッドを緩和することができる。しかし、マルチスレッドやコルーチンはマルチプロセスと異なる部分もある。マルチプロセスはそれぞれでプロセス空間が独立しているので、プロセス間でグローバル変数などは干渉しない。一方、マルチスレッドは 1 つのプロセス空間内で動作するため、グローバル変数は共通している。そのため、グローバル変数の使用には排他制御と同期制御などを適用する必要である。

6.2　Hot Node の呼び出しに対する検討

複数回実行される Hot Node があれば、同じ内容の Hot Node を複数用意して、それぞれに制御を分ける。そうすることで、同じ Hot Node を複数回実行させるという制御の流れとは異なった印象を攻撃者に与えることができる。このような処理を行えば、もとのプログラムよりも、攻撃者が解析を行うことを困難にすることができる。しかし、本来は一つで十分な Hot Node を複数個用意するので、プログラムのサイズが大きくなってしまう。

これの問題を解決するために、以下の手法をサイズが大きくなったプログラムに施し、プログラムのサイズの問題を抑制する。

プログラムのサイズが大きくなるのを抑制する手法を説明する。まず、複数回実行される Hot Node を発見し、複数回実行される Hot Node をいくつかに分割する (図 4 では 3 つに分割)。次に、分割したものと同じものを、複数用意する (図 5 では 1 と 4、2 と 5、3 と 6 のように、それぞれ 2 つずつ用意する)。最後に、分割し複数になったものを、もとの実行の順番を保つ条件のもとで順列を作り (図 5 では、図 4 のブロックの色の順番を保つという条件の下で、実行の順番を決定する)、その順序でを実行させることで、複数回実行される Hot Node と同じ処理を行う。

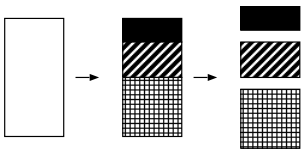

図 4　Hot Node を分割する図

同じ内容の Hot Node を複数用意する場合と比べれ

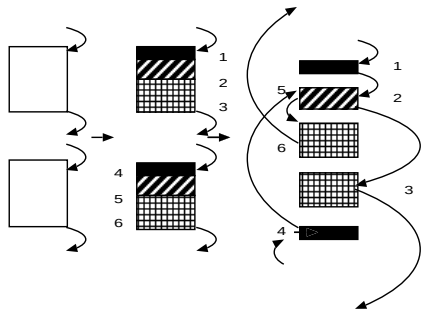

図 5　分割した Hot Node で作った順列をもとの Hot Node に置き換える図

ば、プログラムのサイズは小さくなる。しかし、サイズがもとのプログラムよりは大きくなることが問題点となる。また、別の検討案として、P プロセスから M プロセスの要求に対して、渡されるパラメータを攻撃者から隠したり、発見しにくくするという案も考えられるが、具体的な手法は、検討中である。

6.3　暗号化と復号による実行時間のオーバーヘッドに対する検討

暗号化と復号を行う際、暗号鍵はすべての Cell に対して適用される。この方法は耐性は高いが、暗号鍵を適用するための計算量が多くなり、実行時間は長くなる。暗号鍵の適用による実行時間のオーバーヘッドを緩和するために、実行中の Cell と次に実行する Cell にのみに暗号鍵を適用するという案がある。しかし、暗号鍵の適用方法を変更すると、暗号鍵の生成方法から考え直す必要がある。理想は耐性を維持しつつ、実行時間のオーバーヘッドを緩和することである。

6.4　fork 関数使用によるプロセス作成時に対する検討

この問題点は fork 関数を使用することによって起こるので、fork 関数を使わないで実現する手法を考えれば、改善することができる。fork 関数はマルチプロセスを実現する 1 つの手法だが、6.1 節でも述べたように、マルチスレッドで代用することができる。マルチスレッドはマルチプロセスに比べて、メモリ使用量が少ないため、問題点であるメモリ消費を緩和することができる。問題点は、6.1 節でも述べたが、マルチスレッドの実装方法はマルチプロセスと違いがあるため、注意が必要である。

7　実装・評価

我々は検討案の 1 つとして挙げたマルチスレッドを用いて、プログラムの実装を行った。プログラム作成には C 言語のマルチスレッドライブラリである pthread（POSIX スレッド）を使用した。

7.1　実装

pthread を用いたプログラムは大きく分けて P-thread、M-thread に分けられる。それぞれ 2 プロセス方式における P-Process、M-Process に相当する。P-thread と M-thread 間のデータの受け渡しは共有メモリによって行う。マルチスレッドは同じプロセス内で実行されるので、メモリを共有している。しかし、P-thread と M-thread はそれぞれで実行を行うので、共有メモリにデータを書き込む時に、P-thread と M-thread を同期させる必要がある。Pthread ライブラリにはスレッド間の同期、排他制御を行うための機能があるため、それを利用する。

7.2　評価

評価は実装したプログラムを用いて行う。実装したプログラムにいくつかのサンプルプログラムを組み込み、実行結果の変化を調べる。評価項目は次のように設定する。

- プログラムサイズ
- 実行時間

評価項目のそれぞれについて、Hot Node の数を 4 個、10 個の 2 通りで構成したときの実行結果の変化をまとめる。また、6 節の検討案をもとに改良したプログラムの変化をまとめ、比較する。表 3 に実行時間の一部を示す。

表 3　プログラムの実行時間

ファイル名	ソースプログラム (ms)	Hot Node 4 個 (ms)	Hot Node 10 個(ms)
sample1	0.227	0.322	0.440
sample2	0.064	0.192	0.342
sample3	0.109	0.217	0.308

8　おわりに

実装では pthread を用いてプログラムを作成し、難読化の有無による変化を調べることができた。しかし、暗号化の実装は行わなかったため、暗号化の有無による変化を調べることができなかった。また、暗号化に関する検討が不十分に終わった。今後は、暗号化による実行時間のオーバーヘッドを緩和するための具体的な手法について考察し、改善することが課題となる。

参考文献

[1] 門田暁人ら: “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 (1997.7).

[2] Christian Collberg etal: “A Taxonomy of Obfuscating Transformations”, TR148, Department of Computer Science, University of Auckland (July 1997).

[3] Chenxi Wang etal: “Protection of Software-based Survivability Mechanisms”, DSN’01 (July 2001).

[4] Jun Ge etal: “Control Flow Based Obfuscation”, DRM’05, pp.83-92 (Nov. 2005).