Symbolic Execution に対する難読化の評価

ステュワート ギャヴィン† 宮地充子†‡§ 布田裕一†

†北陸先端科学技術大学院大学
923-1211 石川県能美市旭台 1-1
{gstewart, miyaji, futa}@jaist.ac.jp

‡大阪大学工学部 565-0871 大阪府吹田市山田丘 2-1
§独立行政法人科学技術振興機構 CREST
332-0012 埼玉県川口市本町川口中央ビル 4-1-8

あらまし Linear Obfuscation は，マルウェアの Trigger-based コードを動的解析手法 Symbolic Execution から隠す難読化方式として提案された．Trigger-based コードの検知による難読化位置の特定は共通鍵暗号にとっても脅威と考えられるが，提案された難読化手法の共通鍵暗号への適用については検討されていない．そこで本研究では，Linear Obfuscation を共通鍵暗号 AES へ適用し，評価を行う．AES の鍵を使う処理を White-Box Cryptography により暗号化し，分岐を加え，そこに Linear Obfuscation を適用する．評価は試行錯誤的な実験的評価と，メトリックスを用いた客観的評価，実行時間や計算コストに基づく性能評価を行う．

Evaluation of Obfuscation Against Symbolic Execution

Gavin L. STEWART† Atsuko MIYAJI†‡§ Yuichi FUTA†

†Japan Advanced Institution of Science and Technology
1-1 Asahidai, Nomi, Ishikawa 923-1211, Japan
{gstewart, miyaji, futa}@jaist.ac.jp

‡Graduate School of Engineering, Osaka University
2-1 Yamadaoka, Suita, Oosaka 565-0871, Japan
§JST CREST
Kawaguchi Center Building 4–1–8, Honcho, Kawaguchi-shi, Saitama, 332–0012, Japan

Abstract Linear Obfuscation is obfuscation technique against Symbolic Execution which is effective for trigger-based behavior detection and analysis. Currently, trigger-based behavior analysis and its protection is often performed in malware. In this paper, we apply Linear Obfuscation to common key cryptosystem AES, and evaluate its effectiveness and performance.

1 はじめに

DRM システム (Digital Right Management System) などの著作権保護システムでは，データの暗号化/復号に AES をはじめとする共通鍵暗号が使用されている．このシステムは，ユーザが秘密鍵を手に入れることで，ソフトウェアを不正に使用することが可能となる．そのため，通常のシステムは難読化により秘密鍵を防護している．難読化とは，防護したいプログラムを，解析に耐性のある等価なプログラムに置き換えることで，ソフトウェアタンパー攻撃を防ぐ技術である．難読化は，動的解析により無効化す

ることが可能といわれている．そのため，動的解析に強い共通鍵暗号の難読化方式が必要とされている．

動的解析の一種，Symbolic Execution に耐性を持った，Linear Obfuscation と呼ばれる難読化手法が Zhi らによって提案された [7]．この論文はマルウェアの難読化についての議論はなされているが，暗号方式に対する難読化の議論はなされていない．そこで本研究では，Linear Obfuscation を共通鍵暗号の難読化に応用し，性能評価や，攻撃耐性による評価を行う．これらの評価を通して，Linear Obfuscation の共通鍵暗号に適した方式を検討する．

2 既存研究

2.1 Symbolic Execution.

Symbolic Execution は 76 年に King [3] によって提案され，以降ソフトウェア解析手法の一つとして広く利用されている．主に，マルウェア解析，テストデータの自動生成，入力フィルタの自動生成などの応用例があり [6]，特に Trigger-based code と呼ばれる，ある入力を与えられたら動作する悪意のあるコードの検出に有効な解析手法として知られている [7, 5]．Brumley らは Symbolic Execution を用いて自動的に Trigger による動作を検知する方式を提案した [1]．この Trigger の検知により，共通鍵暗号の難読化位置が特定されやすくなり，攻撃に応用される可能性がある．

2.2 Linear Obfuscation.

Linear Obfuscation [7] は Zhi らが提案した Symbolic Execution に耐性を持った難読化手法である．この手法は，コラッツの予想 [4] を用いたループを挿入することで実現されている．そのため非常にシンプルであり，元のプログラムに 100 bytes 程度コードを追加するだけで実現することができるという利点がある．また，プログラムの動作そのものを隠したり，可読性を下げたりすることが目的ではなく，Symbolic Execution を用いた Trigger-based コードの発見を防ぐことのみを目的としている．そのため，共通鍵暗号に適用する際は，他の難読化手法と組み合わせることが前提となる．

コラッツの予想. コラッツの予想 (Collatz Conjecture) は数論の未解決問題のひとつであり，任意の正の整数 n について，以下の式で表される操作を有限回繰り返したとき $a_i = 1$ を満たす正の整数 i が必ず存在する，という定理である．

$$a_i = \begin{cases} n & \text{for } i = 0 \\ f(a_{i-1}) & \text{for } i > 0 \end{cases}$$

$$\text{where } f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2 \end{cases}$$

この定理は証明がなされていないが，経験則的に正しいことがわかっている．

方式の概要 Linear Obfuscation は 偽の入力とコラッツの予想を基にしたループを挿入することで実現している．図 2.2，図 1 は難読化の適用例である．

```
if(x==30){
  do_m();
}
```

(a) トリガーを元に動作するmaliciousなコード

```
while(y>1){
  if(y%2==1){
    y=3*y+1;
  }else{
    y=y/2;
  }
}
```

(b) Collatz Conjecture

(a) が難読化の対象となる Trigger-based code の例で，(b) はコラッツの予想をコードに書き下したもの，(c) は (b) を用いて (a) を難読化したものである．ここで，x はトリガーとなる入力を格納した変数であり，y は偽の変数である．この偽の変数 y がループを経るごとに変化するためパスが増加するため，正しい x を導くために非常に多くの時間を要することになる．また，y の値が 1 に収束することは経験則的にわかっているため，難読化によりプログラムの結果が変化することはない．

```
// x:user input
// y:spurious input

y=x+1000;
while(y>1){
  if(y%2==1){
    y=3*y+1;
  }else{
    y=y/2;
  }
  if((x-y>28)&&(x+y<32)){
    do_m();
    break;
  }
}
```

(c) 難読化されたコード

図 1: Linear Obfuscation の適用例

2.3 White-Box Cryptography

暗号化/復号にプログラムに埋め込まれた秘密鍵を使う方式の共通鍵暗号では，プログラムコードとメモリ上に現れるデータから鍵データを推測する，という攻撃が考えられる．そのため，共通鍵暗号では秘密鍵をユーザから隠すための特殊な処理が必要となる．White-Box Cryptography は共通鍵暗号の難読化方式の一つである．鍵を使う処理を等価なテーブルによる処理に置き換え，プログラムコードから秘密鍵を推測させないことを目的としている．また，エンコードテーブルを使ってデータを処理をし，出力やテーブルそのものから鍵を推測されることを防ぐ．このとき，鍵を使う処理とテーブルを使う処理は等価であるため，難読化により最終的な出力が変化することはない．

3 提案方式

本研究では，実験に AES を用いる．AES では，暗号化/復号の際に秘密鍵から生成されたラウンド鍵と平文の排他的論理和をとる *AddRoundKey* と呼ばれる処理を行う．この処理を White-Box cryptography を適用して分岐を加えたものを， Liner Obfuscation を用いて難読化する．AES の 1 ラウンド分の処理の流れを図に表したものが図 2である．*AES* ではこの処理を 10 ラウンド繰り返す．今回は，128bits の平文 P を， 128bits の秘密鍵 sk から生成したラウンド鍵 rk を用いて暗号化することを考える．ここで，R はラウンド数を表す．

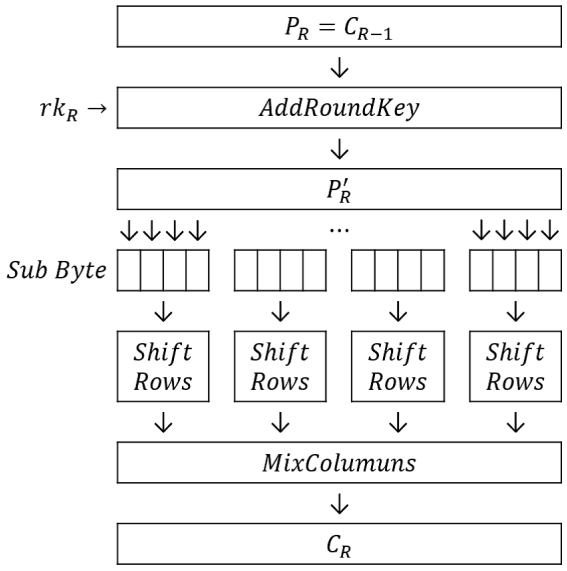

図 2: AES の処理の流れ

3.1 White-Box Cryptography を適用した AddRoundKey

AddRoundKey ではラウンド鍵と平文の排他的論理和をとり，その結果を SubByte に渡す，という処理を行う．式であらわすと，以下のようになる．

$$P'_R = P_R \oplus rk_R$$

P_R は R ラウンド目の入力 ($R = 0$ のときは平文 P)，rk_R は R ラウンド目のラウンド鍵，P'_R は出力を表す．今回はこの処理に簡易な難読化を施し，分岐を加え Linear Obfuscation を適用する．

White-Box Cryptography　ここでは，AddRoundKey の入力（平文）$P = \{\langle p_1, \ldots, p_{31}\rangle \mid p_i \in \{0,1\}^4,\ i \in [1,31]\}$ と R ラウンド目のラウ

ンド鍵 $rk_R = \{\langle rk_{R,1},\ \ldots,\ rk_{R,31}\rangle \mid rk_{R,i} \in \{0,1\}^4,\ i \in [1,31]\}$ の排他的論理和の結果をあらかじめテーブルに保持しておき，平文の値を元に出力を決定するアルゴリズムを作成する．以下に流れを示す．

1. 各 $j \in [0,15]$ に対してテーブル S^j を作成する．

   (1-1) 関数群 F, G から j に対応する関数 $f_j \in F,\ g_j \in G$ を選ぶ．

   (1-2) 各 $i \in [1,31]$ に対して，テーブル S^j の $(f_j(R),\ g_j(i))$ 要素に $rk_{R,i} \oplus j$ を代入する．

$$S^j_{f_j(R),g_j(i)} \leftarrow rk_{R,i} \oplus j$$

2. 各ラウンド $R \in [0,9]$ で以下の操作を行う．(図 3)

   各 $i \in [1,31]$ に対して

   (2-1) j に P の i 番目の要素 p_i を代入する．

   (2-2) j に対応する $f_j,\ g_j$ を選択する．

   (2-3) p'_i に $S^j_{f_j(R),g_j(i)}$ を代入する．

$$p'_i \leftarrow S^j_{f_j(R),g_j(i)}$$

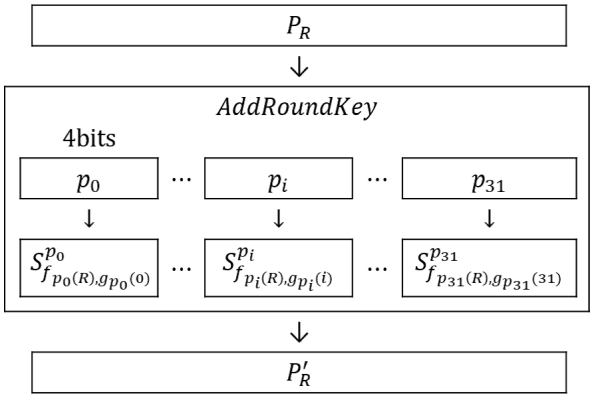

図 3: 難読化した AddRoundKey の処理の流れ

3. $P' = \langle p'_i \rangle$ を SubByte に渡す．

この処理の流れを擬似コードに書き下したのが **Algorithm 1** である．なお，アルゴリズム中に Chose で書かれている処理は switch-case などの分岐処理を用いて実現される．

Algorithm 1 White-Box cryptography を用いた AddRoundKey アルゴリズム

Require: P, rk, F, G
Ensure: P'

```
/* Create S box S^j */
for j = 0 to 15 do
    Chose function f_j ∈ F, g_j ∈ G
    for R = 0 to 9 do
        for i = 0 to 31 do
            S^j_{f_j(R),g_j(i)} ← rk_{R,i} ⊕ j
        end for
    end for
end for

/* do AddRoundKey for each round R */
for i=0 to 31 do
    j ← p_i
    Chose f_j ∈ F, g_j ∈ G
    p'_i ← S^j_{f_j(R),g_j(i)}
end for
```

3.2 分岐の追加

難読化を施した AddRoundKey の制御フローの例を図 4 に示した．ここでは，この制御フローへの分岐を以下の観点で考える．第一に正しい動作とダミー動作の差がない方法，第二に動作には違いがあるがパフォーマンス低下が押さえられる方法である．以下の 3 つのパターンについて検証する．

1. ダミーのテーブル S^d を用いる
2. ダミーの関数群 F_d, G_d を用いる
3. ダミーの関数 f_d, g_d を用いる

1,2 が正しい動作とダミー動作の差がない方法，3 が動作には違いがあるがパフォーマンス低下が押さえられる方法である．これらについて，追加する分岐の差がパフォーマンスや攻撃耐性に及ぼす影響を検証し，比較することで適切な適応手法を検討する．

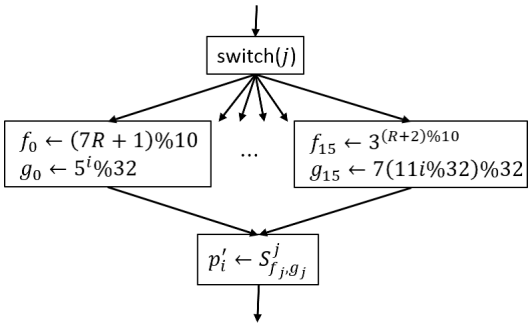

図 4: White-Box Cryptography を用いて構成した AddRoundKey の制御フローの例

パターン 1　ダミーのテーブルを用いて分岐を構成した場合のアルゴリズムを **Algorithm 2** に示す．ダミーテーブルをあらかじめプログラム中に埋め込む必要があるため，その分データサイズが増加する．追加したダミーテーブルの数を n としたとき，増加するデータサイズは $160n$bytes となる．分岐の数は追加したダミーテーブルの数に依存して増加するため，パターン 2,3 に比べてシンプルな構造となる．

Algorithm 2 ダミーのテーブルを用いて分岐を構成した場合のアルゴリズム

for i=0 to 31 **do**
  $j \leftarrow p_i$
  Chose $f_j \in F$, $g_j \in G$
  if Check **then**
    $p'_i \leftarrow S^j_{f_j(R),g_j(i)}$
  else
    $p'_i \leftarrow S^d_{f_j(R),g_j(i)}$
  end if
end for

パターン 2　ダミーの関数群を用意した場合のアルゴリズムを **Algorithm 3** に示す．この場合，関数 f, g を選択する分岐処理の数が 2 倍になり，その分プログラムの計算コストやプログラムの複雑度が増加することが予想される．データサイズの増加量は，関数群 F, G のサイズ，関数 f, g のサイズに依存する．

Algorithm 3 ダミーの関数群を用いて分岐を構成した場合のアルゴリズム

for i=0 to 31 **do**
  $j \leftarrow p_i$
  if Check **then**
    Chose $f_j \in F$, $g_j \in G$
  else
    Chose $f_j \in F_d$, $g_j \in G_d$
  end if
  $p'_i \leftarrow S^j_{f_j(R),g_j(i)}$
end for

パターン 3　ダミーの関数を用いて分岐を構成した場合のアルゴリズムを **Algorithm 4** に示す．関数 f, g を選択する際の分岐処理の数の増加数は，追加するダミー関数の数とサイズに依存する．そのため，パターン 2 と比較してコストや複雑度の増加が少なくなることが予想される．また，check が真だった場合と偽だった場合で実行時間やフローに有意な差が現れることが予想される．

Algorithm 4 ダミーの関数を用いて分岐を構成した場合のアルゴリズム

for i=0 to 31 **do**
  $j \leftarrow p_i$
  if Check **then**
  else
    $j \leftarrow d$
  end if
  Chose $f_j \in F$, $g_j \in G$
  $p'_i \leftarrow S^{p_i}_{f_j(R),g_j(i)}$
end for

3.3　Linear Obfuscation の適用

追加した分岐を利用して Linear Obfuscation を AES に適用する．ユーザ入力 x を偽の変数 y を使って検証し，真の場合は正しい処理，偽の場合はダミー処理を行うようにループを挿入する．以下のようにコードを記述する．

```
// x : userinput
// y : spuriousvariable
y = x + 1000;
while (y>1){
   if(y%2==1) y=3*y+1;
   else y=y/2;

   if(check){
      //正しい処理
      break;
   }else{
      //ダミー処理
      break;
   }
}
```

4 評価方法

Linear Obfuscation は Symbolic Execution に耐性を持つ難読化手法として提案されている．この手法の有効性を評価するためには，Linear Obfuscation を施したプログラムに対して Symbolic Execution による解析を実際に行い，耐性を持っているかを検証する必要がある．加えて本研究では，客観的な評価の指標として，Collberg [2] によりまとめられた，メトリックス (Metrics) に基づいた難読化手法の評価を行う．また，実行時間などのパフォーマンスも評価の対象とする．

メトリックス　ソフトウェアメトリックスはプログラムの複雑度をあらわす指標であり，主にソフトウェア開発プロジェクトにおいてプログラムの構造やデザインの洗練度を評価するために利用される．また，オリジナルのプログラムと難読化後のプログラムのメトリックスを比較することで，難読化手法の評価に用いることが可能である．例として，以下のような評価項目があげられる．

- オペレータ，オペランドの数 (プログラムの長さ，LOC)
- 実行パスの数
- サイクロマチック複雑度
- ネストの深さ
- データフローの複雑さ (ベーシックブロックをまたいで参照される変数の数)
- オブジェクト指向複雑度 (クラスやモジュールの結合度，メソッドの数，継承木の深さ)

本研究で用いる Linear Obfuscation は，実行パスの数を増やすことで Symbolic Execution に耐性を持たせることを目的としている．そのため，有効性の評価には実行パスの数が重要となる．また，パフォーマンス維持の観点から他の指標に関しては増加しないことが望ましいといえる．

5 Conclusion

本紙では，マルウェアの難読化手法として提案された Linear Obfuscation を共通鍵暗号に適用するための方式と，その評価方法をまとめた．今回は，分岐の追加の仕方を 3 パターン用意し，パフォーマンスやプログラムの複雑さの観点から最も有効な Linear Obfuscation の適用方法を検討する． 今後の方針として，提案方式を実装し，Symbolic Execution およびメトリックスに基づいた評価を進めていく．

謝辞

本研究の一部は科学研究費·基盤 (C) (15K00183) 及び (15K00189) の助成を受けています．

参考文献

[1] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pp. 65–88. Springer, 2008.

[2] Christian Collberg and Jasvir Nagra. Surreptitious software: Obfuscation, watermarking, and tamperproofing for software protection. 2009.

[3] James C King. Symbolic execution and program testing. *Communications of the ACM*, Vol. 19, No. 7, pp. 385–394, 1976.

[4] Jeffrey C Lagarias. The 3x+ 1 problem and its generalizations. *American Mathematical Monthly*, pp. 3–23, 1985.

[5] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pp. 231–245. IEEE, 2007.

[6] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 317–331. IEEE, 2010.

[7] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear obfuscation to combat symbolic execution. In *Computer Security–ESORICS 2011*, pp. 210–226. Springer, 2011.