社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2006－DPS－126（41）
2006－CSEC－32（41）
2006／3／17

動的解析に対し耐タンパ性を持つ難読化手法の提案

服部 太郎，双紙 正和，宮地充子
北陸先端科学技術大学院大学 情報科学研究科
〒923-1292 石川県能美市旭台 1-1
E-mail:{t-haxtsu,soshi,miyaji}@jaist.ac.jp

概要　ソフトウェアの知的財産を保護するために，難読化は有効な手段である．そのため，多数の難読化手法が提案されてきたが，静的解析に対する耐タンパ性のみに言及した手法が多く，動的解析に対する耐タンパ性に言及した難読化手法はあまり提案されていない．そこで，本研究では，動的解析に対して耐タンパ性を持つ難読化手法の提案を行う．特に，実行経路が外部入力に依存するプログラムについて，その動的解析の困難さが NP-Hard であることを証明し，安全性に一定の理論的根拠が存在する手法の提案を行う．

An Obfuscation Technique with Tampar-resistance against Dynamic Analysis

Taro Hattori, Masakazu Soshi, Atsuko Miyaji
School of Information Science,
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa 923-1292, JAPAN.
E-mail:{t-haxtsu,soshi,miyaji}@jaist.ac.jp

Abstract　Obfuscation is an effective means to protection of software and many obfuscation techniques have so far been proposed. However most of the previous obfuscation techniques paid attention to tamper-resistance against static analysis only. Therefore in this paper we propose an obfuscation techuique which is tamper-resistant to dynamic analysis. We show that the difficulty of conducting dynamic analysis on programs whose execution paths depend on external inputs is in NP-Hard. This fact gives our obfuscation techuniqes a theoretical foundation.

1　はじめに

近年，Java，JavaScript などソースコード形式，もしくは，それに近い形式でのソフトウェア配布が増えている．そのような状況では，ソフトウェアの機密データ，重要なアルゴリズムが盗まれる危険性がある．そのため，知的財産の保護，著作権保護の為には耐タンパソフトウェアが必要となる．

耐タンパソフトウェアとはソフトウェアに対する不正なリバースエンジニアリング（解析，改変）を困難としたソフトウェアである．難読化は耐タンパソフトウェアを実現する方法の一つであり，暗号化が実行時に復号を必要としているのに対して，難読化は復号を必要とせず，そのままの形で実行はできるが，プログラムの解析は困難なものとする技術を言い，ソフトウェアを保護する有力な方法として着目を浴びている [5][6].

しかし，既存研究の多くは静的解析に対する耐タンパ性のみに言及した手法であり，動的解析に対する耐タンパ性に言及した難読化手法はあまり提案されていない．そこで，本研究では，動的解析に対して耐タンパ性を持つ難読化手法の提案を行う．特に，

実行経路が外部入力に依存するプログラムについて，その動的解析の困難さが NP-Hard であることを証明し，安全性に一定の理論的根拠が存在する手法の提案を行う．本研究では，難読化に並列プログラミング (マルチスレッド) を利用する．

2　既存研究

従来の研究において，小木曾ら [1] により，静的解析に置けるポインタ解析の困難さに基づいた難読化手法が提案され，静的解析の困難さに一定の理論的根拠が与えられている．また，刑部ら [2] はオブジェクト指向型言語の特性を利用した難読化手法を提案している．しかし，これらの手法は静的解析に対する耐タンパ性のみを言及しており，動的解析に対する耐タンパ性は言及していない．豊福ら [3] により，乱数を用いてプログラムの制御構造を複雑化すると共に，メソッド実行順序の擬似的な入れ替えを行うことによる，動的解析に対する耐タンパ性についても言及した難読化手法が提案されている．しかしながら，[3] においては理論的な考察が加えられていなかった．そこで本研究では，動的解析の困難さに一定の理論的根拠を与えられるような難読化手法を提案する．

3　動的解析と静的解析

プログラムの解析手法は，プログラムを実行しないで行われる静的解析と，プログラムを実行して行われる動的解析の 2 種類に大きく分けられる．静的解析においては，if 文などの条件分岐が存在した場合，meet over all paths 仮定 [7] に基づき，全ての文が実行される可能性があるという観点から解析を行う．さらに，静的解析においては，関数呼出し等があるとき，実行可能経路 (executable path) を決定することが困難である [1]．しかしながら，静的解析においては，理論的な解析手法がほぼ確立されているという利点がある．一方で，動的解析はプログラムにテストケースとしてある値を入力し，実際に実行された文のみを解析の対象とする．その為に，動的解析はテストケースに対する解析しか行えず，プログラム全体の動きを想定した解析には適していない．一方，実際の実行履歴に基づいて解析を行うため，meet over all paths 仮定は必要なく，また，実行可能経路の決定などの困難さは生じない．これらの違いから静的解析に対して耐タンパ性を持つプログラムが必ずしも動的解析に対して耐タンパ性を持つとは限らず，またその逆も言える．

4　提案手法

本節では，動的解析に強い難読化手法を提案し，その理論的根拠について議論する．

4.1　理論的根拠

定理 1　プログラムが if 文，外部からの入力を持つとき，ある (ブール) 変数が真となるような実行経路が存在するかどうかを判定する問題は NP 困難である．

略証　ある 3-SAT のインスタンスが与えられたとき，それに対応する図 1 のプログラムを考えることができる．ここで，input() は外部からの入力を返す関数とする．このとき，外部入力 i の値に 1 対 1 に対応する実行経路がある．そして，f の値が真となるような実行経路が存在するとき，そのときに限り，元の 3-SAT のインスタンスを満たす真理値割り当てが存在する．(証明終わり)

文献 [1] における静的解析の場合の証明と異なり，図 1 のプログラムには if 文における条件式が明示されている点に注意せよ．そこで，図 1 のプログラムは実際に実行可能であり，ある入力について実行したとき，実行された経路が唯一対応する．

言いかえれば，定理 1 の証明により，図 1 のような外部入力によって実行経路が変わるプログラムにおいて，ある変数の値が真となるかどうかを動的解析によって決定することは困難ということが証明されたことになる．言うまでもなく，may-alias 問題等，様々な問題の動的解析の困難さが同様にして示される．

4.2　難読化手法の提案

本研究では定理 1 を理論的根拠とし，Java をサンプルとして難読化手法の提案を行う．難読化を行うにあたり，問題となるのはオリジナルプログラムの挙動を如何に維持するかと言う点である．そこで

```
int i;
boolean f;
boolean v1, v̄1, ···, vm, v̄m;
main()
{
L1: i = input();

L2: if (i%2 == 0) { v1 = true; v̄1 = false; }
    else { v1 = false; v̄1 = true; }
    i = i/2;
    ···
    if (i%2 == 0) { vm = true; v̄m = false; }
    else { vm = false; v̄m = true; }
    i = i/2;

L3: if (i%3 == 0) f = l1,1;
    else if (i%3 == 1) f = l1,2;
    else f = l1,3;
    i = i/3;
    if (i%3 == 0) f = f && l2,1;
    else if (i%3 == 1) f = f && l2,2;
    else f = f && l2,3;
    i = i/3;
    ···
    if (i%3 == 0) f = f && ln,1;
    else if (i%3 == 1) f = f && ln,2;
    else f = f && ln,3;

L4:
}
```

図 1: 3-SAT からの帰着による証明

以下のような方法を考える:

1. プログラムからセグメント (難読化する対象) の選定を行う.
2. それぞれのセグメントが同じテストケースに対しても複数の異なる実行系列を持つプログラムに変換する. ここで, どの実行系列を実行しても, 同じ結果が得られるようにする.

そして, 実行系列の複数化は下記の三点の方法によって試みる.

1. 同一実行経路の複数化
2. ダミーの付加による実行順序の複雑化
3. 1,2 の併用

4.2.1 外部入力の代替

提案手法の基本的なアイディアは実行経路が外部入力に依存するプログラムを難読化に利用することである. しかし, 難読化の為だけにプログラムに外部入力を持たせることは難しい. そこで, 外部入力の代わりに並列プログラミング (マルチスレッド) を利用することを考える.

実行経路の複数化 まず, マルチスレッドにより実行経路の複数化を行う方法を考える. オリジナルプログラムに存在する (オリジナル) セグメントと同じ動きをするコピー (セグメント) を複数作成し, マルチスレッドの動きにより複数の同じ挙動をするセグメントの中から一つを選び実行する.

```
class orig {
   public static void main(String[] args) {
      A.a();
   }
}
```

図 2: オリジナル

図 3 に概要を示す. 図 2 をオリジナルのプログラムとし, cpA.cpa() を A.a() と同じ挙動をするコピーとする. 新たに実行するセグメントを決定するクラス scheduler, メソッド dispatcher を作成し, scheduler は Thread を継承する. scheduler 内の run メソッドから dispatcher は呼び出され, 複数のスレッドにより実行される形とするが, 排他制御を行い同時に複数のスレッドにより実行されないようにする. そして, 一つ一つのスレッドに固有の ID を付け, どのスレッドが dispatcher を実行しているか識別でき

```
class obf {
   public static void main(String[] args) {
      scheduler.control=0;
      scheduler th1 = new scheduler(1);
      scheduler th2 = new scheduler(2);
      th1.start();th2.start();
   }
}

class scheduler extends Thread {
   static int control;
   int thID;
   scheduler(int thID) {
      this.thID = thID;
   }
   public void run() {
      scheduler.dispatcher(thID);
   }
   synchronized static void dispatcher(int thID) {
      If(control==0) { //(1)
         If(thID==1) { //(2)
            A.a();
            control=10; //(3)
         } else {
            cpA.cpa();
            control=10; //(3)
         }
      }
   }
}
```

図 3: 実行経路の複数化代替

る様にする．

さらに，全スレッドで変数 control をクラス変数として共有し，dispatcher 内において control の値と条件式を見比べ，(図 3(1)) 結果が真であれば，スレッドの持つ ID により実行するセグメントを決定し (図 3(2))，セグメントを実行する．この際，セグメント実行後に，control の値を変化させ (図 3(3))，control の値を条件式が真にならない数値にする．これにより，一つのスレッドがセグメントを実行した後，他のスレッドが dispatcher を実行してもセグメントが実行されることはなく，プログラムの挙動を維持したまま，実行経路を複数化できる．

実行順序の複雑化　次にマルチスレッドにより実行順序の複雑化を行う方法を考える．

概要は図 4 に示す．複雑化にはプログラムの挙動に影響を及ぼさないダミー (セグメント)(図 4(3)) を使用するが，乱数は用いず，スレッドの動きにより実行するセグメントを決定する．そして，dispatcher を繰り返し構造内から呼び出し (図 4(1))，ダミーが実行された際には繰り返し構造の条件式の値を変化させず，繰り返しを抜けないが，ダミーでないセグメントが実行された際は繰り返し構造の条件式の値が変化し (図 4(2))，繰り返しを抜ける．

```
class scheduler extends Thread {
  …
  public void run() {
    While(control<=5) { //(1)
      scheduler.dispatcher(thID);
    }
  }
  synchronized static void dispatcher(int thID) {
    If(control==0) {
      If(thID==1) {
        A.a();
        control=10; //(2)
      } else {
        D.dummy(); //(3)
      }
    }
  }
}
```

図 4: sample:実行順序の複雑化代替

4.2.2　アルゴリズムの能率化

一つ一つのセグメント毎に scheduler を用意し，マルチスレッド化してインスタンス化していては，プログラムサイズの増大化を招く恐れがある．そのため，プログラム中の scheduler は統合し，連続したセグメントにおけるインスタンス化は一度だけ行い，dispatcher の呼び出しを繰り返し文を利用して複数回行う．そして繰り返し文により一度 dispatcher が実行される度にセグメントを一つ実行し，実行後 control の値を次に実行するセグメントの条件式が真となる値に変更する．

4.2.3　if 文を含むプログラムの難読化

プログラムに if 文が含まれるケースは多々考えられる．その為，ここでは if を含むプログラムの難読化方法を考える．

```
if(x<=10) {
  A.a();
}
```

```
class scheduler extends Thread {
  …
  public void run() {
    while(control<=15) {
      scheduler.dispatcher(thID);
    }
  }
  synchronized static
    void dispatcher(int thID) {
    if(control==0) {
      if(eg.x<=10) {
        control=10;
      } else {
        control=20;
      }
    } else if(control==10) {
      If(thID==1) {
        A.a();
        control=20;
      } else if(thID==2) {
        cpA.cpa();
        control=20;
      }
    }
  }
}
```

図 5: sample:if 文の難読化

図 5 に概要を示す．if 文の条件式その物を一つのセグメントとし，条件式の値が真となる際，偽となる際でそれぞれ異なった値を変数 control に代入する．

4.2.4　繰り返しを含むプログラムの難読化

プログラムに繰り返しが含まれるケースもまた，多々考えられる．その為，繰り返しを含むプログラムの難読化方法も考える．

図 6 に概要を示す．

```
while(i<=10) {
   A.a();
   i++;
}
```

```
class scheduler extends Thread {
   ...
   public void run() {
      while(control<=50) {
         scheduler.dispatcher(thID);
      }
   }
   synchronized static
      void dispatcher(int thID) {
      if(eg.i<=10) {
         If(thID==1) {
            A.a();
            eg.i++;
         } else if(thID==2) {
            cpA.cpa();
            eg.i++;
         }
      } else {
         control=100;
      }
   }
}
```

図 6: sample:繰り返しの難読化

4.2.5 変数 control の難読化

変数 control は実行の流れを制御する重要な変数であり，この部分の解析を困難にすることが重要である．そこで、変数 control の難読化を行う．まず、

```
if(array[idx]==10) {
   A.a();
} else if(array[idx]==20) {
   cpA.cpa();
} else if(array[idx]==30) {
   D.dummy();
}
```

図 7:

static な変数，state，idx と配列 array[] を作成し，これを control に代わり全スレッドでクラス変数として共有する．セグメントの実行は変数 idx 番目の array[idx] の位置の値と条件式が真となるセグメントが存在すれば 1 つのセグメントが実行され，条件式が真となるセグメントが存在しない場合はセグメントは実行されない．(図 7)

スレッドにより dispatcher が一度実行される度に，state，idx，array[idx] の値を変えると言った一連の動作を一度づつ行う．この動作をスレッドの動きにより複雑化して行うことにより実行系列を複数化する．

実行経路の複数化　始めに実行経路の複数化に置いて control を難読化する．dispatcher を実行したスレッドの持つ ID により同じ state，同じ array[] に対しても，違う値を array[] に書き込み，state，idx は同じ変化をさせる．

図 9，図 8 に概要を示す．state，idx 共に初期値は 0，array[] も全てが 0 とする．スレッドの動きにより array[] に異なる値を書き込んだ後，その場所を idx が指す示すことにより，実行経路を複数化する．

```
class scheduler extends Thread {
   ...
   public void run() {
      while(state<=50) {
         scheduler.dispatcher(thID);
      }
   }
   synchronized static void dispatcher(int thID) {
      図 7
      図 9 or 図 10
   }
}
```

図 8: 全体象

```
if(thID==1) {
   if(state==0) {
      if(array[idx]==0) {
         state=10;
         array[idx]=10;
         idx++;
      }
   } else if(state==10) {
      if(array[idx]==0) {
         idx−−;
      }
   }
}
```

```
if(thID==2) {
   if(state==0) {
      if(array[idx]==0) {
         state=10;
         array[idx]=20;
         idx++;
      }
   } else if(state==10) {
      if(array[idx]==0) {
         idx−−;
      }
   }
}
```

図 9: sample:実行経路の複数化

実行順序の複雑化　次に，実行順序の複雑化に置いて control の難読化を行う．

図 10 に概要を示す．ダミーが実行される値が書き込まれる，又はダミーでないセグメントが実行される値が書き込まれる，と言った 2 種類の動作がまず存在し，ダミーでないセグメントが実行される値が書き込まれた後はさらにダミーを実行する値を書き込むか，値を書き込むのを止めて書かれている値を読むと言う 2 種類の動作を行う．

```
if(thID==1) {
  if(state==0) {
    if(array[idx]==0) {
      state=10;
      array[idx]=10;
      idx++;
    }
  } else if(state==10) {
    if(array[idx]==0) {
      state=40;
      array[idx]=30;
      idx++;
    }
  } else if(state==20) {
    if(array[idx]==0) {
      state=0;
      array[idx]=30;
      idx++;
    }
  } else if(state==30) {
    if(array[idx]==30
            ||20||10) {
      idx--;
    }
  } else if(state==40) {
    if(array[idx]==0) {
      state=30;
      idx--;
    }
  }
}
```

```
if(thID==2) {
  if(state==0) {
    if(array[idx]==0) {
      state=20;
      array[idx]=30;
      idx++;
    }
  } else if(state==10) {
    if(array[idx]==0) {
      state=30;
      idx--;
    }
  } else if(state==20) {
    if(array[idx]==0) {
      state=40;
      array[idx]=20;
      idx++;
    }
  } else if(state==30) {
    if(array[idx]==30
            ||20||10) {
      idx--;
    }
  } else if(state==40) {
    if(array[idx]==0) {
      state=10;
      array[idx]=30;
      idx++;
    }
  }
}
```

図 10: sample:実行順序の複雑化

5 評価

提案手法を RC6，FFT に適応し，その効果を確かめる．評価については以下の観点により行う．

表 1: 動的解析に対する耐タンパ性

RC6		Before	After	ratio
Call Graph	nodes	6	14	2.2
	edges	6000	96020	16

FFT		Before	After	ratio
Call Graph	nodes	10	17	1.7
	edges	19313	431413	22

表 1 はプログラムの動的なコールグラフの変化を表している．グラフの作成は，同じ実行経路が出現した際には履歴を保存せず，破棄する形で行っている．また，表 2 にはプログラムサイズの変化を表している．表 2 より，プログラムサイズは倍近い大きさになっていることが分かるが，コールグラフはそれ以上に大きくなっており，提案手法によりプログラムを難読化する価値はあると言える．

表 2: プログラムサイズ

RC6		Before	After	ratio
program size	source[lines]	626	1199	1.9
	class file[byte]	7.4	14.3	1.9
総実行経路数		1	16	16

FFT		Before	After	ratio
program size	source[lines]	332	561	1.7
	class file[byte]	7.7	9.33	1.2
総実行経路数		1	16	16

6 まとめ

外部入力の存在するプログラムに対する動的解析の困難さが NP-Hard であることを証明し，動的解析の困難さに一定の理論的根拠を確立することができた．また，上記を根拠としながらも，外部入力の代わりにマルチスレッドを利用することにより実用的な難読化手法の提案が行えた．

参考文献

[1] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, “Software obfuscation on a theoretical basis and its implementation,” IEICE Trans, Fundamentals, vol.E86-A, No.1 176-186, Jan,2003.

[2] A. Miyaji and Y. Sakabe and M. Soshi,“Java Obfuscation – Approaches to Construct Tamper-Resistant Object-Oriented Programs,” IPSJ Trans, vol.46, No.8 2107-2119, Aug,2005.

[3] 豊福，田端，櫻井，“メソッド実行順序の擬似的入れ替えによる難読化手法の提案，” 2006 年暗号と情報セキュリティシンポジウム (SCIS2006)，(1, 2006).

[4] A. Monden, A. Monsifrot, C. Thomborson, “Tamper-Resistant Software System Based on a Finite State Machine,” IEICE Trans, Fundamentals, Vol.E88-A, No.1 112-122, Jan,2005.

[5] C. Collberg, C. Thomborson, D. Low, “A Taxonomy of Obfuscating Transformations,” Technical Report148, Department of Computer Science, the University of Auckland, New Zealand, 1997

[6] D. Low, “Java control flow obfuscation,” Master of Science Thesis, Department of Computer Science, the University of Auckland, New Zealand, 1998

[7] Thomas J. Marlowe and Barbara G. Ryder. Properties of data flow frameworks: A unified model. Acta Informatica, Vol.28, No.2, pp.121-163, 1990.