平成 25 年度 修士論文

# 仮想メソッド呼出しを用いることによる プログラム難読化のオーバヘッド削減

電気通信大学 大学院情報システム学研究科
情報システム基盤学専攻
1253002　石田 峰文

指導教員　小宮 常康　准教授
多田 好克　教授
古賀 久志　准教授

提出日　平成 26 年 1 月 27 日

# 目次

# 図目次

# 表目次

# プログラムコード一覧

# 第 1 章

# 導入

プログラムに含まれるアルゴリズムなどの知的財産が盗まれることを困難にするため、プログラムに対して制御フローの難読化が行われることがある。プログラムの制御フローの難読化として、実行時の取りうる値が決まっているが静的な解析では取り除けない条件分岐を用いて実際には実行されないコードや崩れた制御構造を作り出す手法が存在する。しかし、難読化によって加えられる条件分岐によるオーバヘッドを伴うという問題点がある。

そこで、難読化で加える条件分岐の動作を仮想メソッド呼出しで実現する手法を提案する。仮想メソッド呼出しはオブジェクトの実行時の型によって呼出すメソッドが決まる、多態性が存在する。この仮想メソッド呼出しにおける多態性を実現するメソッドディスパッチが、条件分岐と同等の役割を果たす。そのため、条件式の値と仮想メソッド呼出しのオブジェクトの型を対応させることで、条件分岐と同等の役割を果たす仮想メソッド呼出しを行うことができる。しかし、この仮想メソッド呼出しにおけるオブジェクトの実行時の型を一意にすることで、仮想メソッド呼出しの実行時の振舞いは多態的ではなくなるため、実行時コンパイラが行う動的最適化によるインライン展開によってメソッドディスパッチの除去が可能である。それにより、難読化によってプログラムに加わるオーバヘッドの削減を期待できる。

本論文では、従来は条件分岐と goto 命令を用いて実現していた、実際には実行されないコードや崩れた制御構造を作り出す難読化手法に対し、提案手法である仮想メソッド呼出しを用いて同等の難読化を実現するための変換の流れを示す。また、難読化対象となるプログラムの文と、Java 言語で書かれたプログラムに対して提案手法を適用するための方法についても示す。

提案手法による仮想メソッド呼出しを用いた難読化を実際に行い、既存手法による条件分岐を用いた難読化とのプログラムの実行時間に関する評価を、Java 言語で作られたプログラムを対象として行った。結果、プログラム全体の実行時間が長いならば、難読化によって加わる実行時間に関するオーバヘッドを最大で 478% から 235% に削減することができることが分かった。また、提案手法による難読化手法を Java プログラムコードに対して行う変換器の実装も行った。

本論文の構成を以下に示す。第 2 章では、本研究の背景となるプログラムの難読化手法と動的最適化について述べる。特に、本研究の提案手法で活用する制御フローの難読化と仮想メソッド呼出しのインライン展開について詳しく述べる。第 3 章では、本研究の提案手法である仮想メソッド呼出しを用いた難読化の基本アイデアと実例について述べる。第 4 章では、提案手法に基づき実装した Java 言語で書かれたプログラムコードを対象とする変換器について述べる。第 5 章では、提案手法に基づいて難読化されたプログラムの実行時間のオーバヘッドに関する評価について述べる。第 6 章では、研究のまとめと今後の課題について述べる。

# 第 2 章

# 背景

本研究では、プログラムに対して難読化を行うことにより加わるオーバヘッドを削減するため、実行時コンパイラが実行時に行う動的最適化処理を活用する。そこで、まず 2.1 節 (p.3) ではプログラムの難読化に関する背景を述べ、次に 2.2 節 (p.10) ではプログラムの動的最適化に関する背景を述べる。

## 2.1 プログラムの難読化

プログラムに含まれるアルゴリズムやデータ構造などの知的財産を守るため、プログラムの改ざんや流用を防止することでプログラムの機密性を保つことは重要である。そのため、プログラムが持つ本来の機能を保ちつつ、他者による不正な解析を難しくさせる手法として、**プログラムの難読化**が使われることがある。プログラムの暗号化と比べ、プログラムの難読化ではプログラムの知的財産を完全に守ることはできないが、現実的な時間ではプログラムの解析を行えなくすることができる [1][2]。

プログラムの難読化は、大きく分けて以下の 3 種類に分類することができる。

**レイアウトの難読化**
: プログラムに含まれた変数や関数などの識別子の名前やフォーマット、コメントなどの字句情報を対象とした難読化手法

**データ構造の難読化**
: 変数や配列、型などのデータ構造を対象とした難読化手法

**制御フローの難読化**
: プログラムコードの位置やメソッドが持つコード情報などのプログラムの流れ (アルゴリズム) である制御フロー情報を対象とした難読化手法

Collberg ら [2] は、様々な難読化手法に対し、難読化によりどれだけ解析し難くなるかの有益性、解析しやすい形へ復元できるかの復元力、難読化によりプログラムに加わるコスト、の 3 つの観点から評価を行った。彼らは、レイアウトの難読化やデータ構造の難読化は、情報の復元が難しくプログラムに加わるコストも低いことが多いが有益性は高くな

いことが多く、制御フローの難読化は有効性が高く復元が難しいほどコストが高くなる傾向があるとまとめている。

本研究では、最後の**制御フローの難読化**に着目する。

プログラムが実行したときに通る可能性のある経路を**制御フロー**と呼ぶ。プログラム $P$ の基本ブロック (分岐を含まないコード列) の集合を $V$、制御フローの辺集合を $E$ としたときのグラフ $G =< V, E >$ を、プログラム $P$ の**制御フローグラフ**と呼ぶ。例として、図 2.1 のループを含んだプログラムコードの制御フローグラフを図 2.2 に示す。

制御フローグラフはプログラムのアルゴリズムそのものであるため、制御フローグラフを難読化することで、サブルーチンやアルゴリズムの隠蔽に役立てることができる [1]。

制御フローの難読化の実例として [1]、図 2.3 のように制御フローグラフを平滑化することで、基本ブロックがどのような順序で実行されるかを分からなくする手法 [3][4]、条件分岐を例外処理で置換えることで制御フローの解析を困難にする手法 [5] などが存在するが、本研究では条件分岐などを用いて制御フローグラフを複雑にする手法に着目する。

```
1 n = 0;
2
3 do {
4       n++;
5 }while(n < 10);
6
7 x = n;
```

図 2.1　プログラムコード例

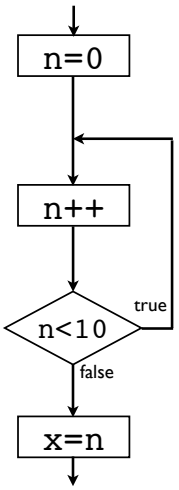



図 2.2　図 2.1 の制御フローグラフ

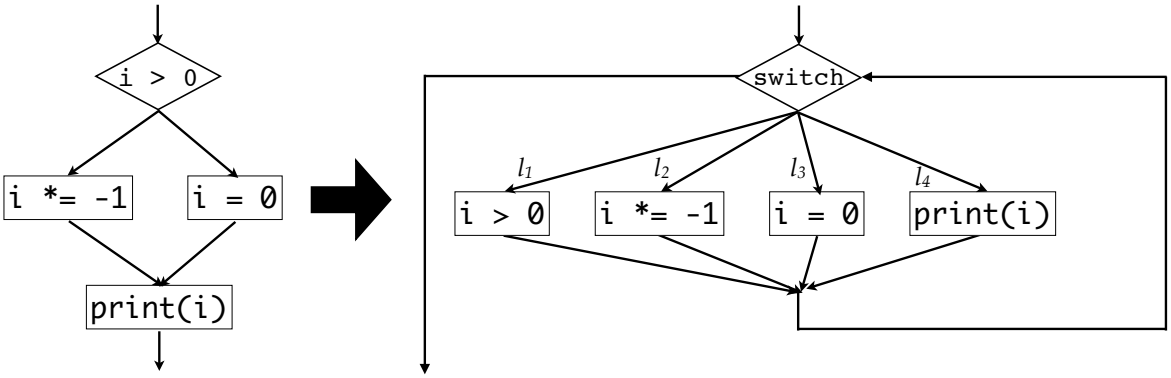



図 2.3　制御フローグラフの平滑化の例

### 2.1.1 Opaque な条件式を利用したプログラムの制御フローの難読化

Collberg ら [6] は、実行時の取る値が一定である変数や式を用いた難読化手法について述べている。このような性質を **opaque** と呼ぶ。Opaque な変数や条件式の例を図 2.4 に示す。2 行目の変数 v の値は必ず 11 であり、3 行目の条件式 “b > 5” は必ず真、4 行目の条件式 “random(1,5) < 0” は必ず偽である。また、5 行目で a と b が不変ならば、6 行目の条件式 “b < 7” は必ず真であり、7 行目の変数 v の値は必ず 36 である。

図 2.4 で示した opaque な変数や述語は単純であるため、制御フローの難読化のために使うのは有用ではない。条件式が必ず真 (または偽) であることを解析者にとって分からなくするため、$x^2(x+1)^2 \mod 4 \equiv 0 (x \in \mathbb{Z})$ や $7x^2 - 1 \neq y^2 (x, y \in \mathbb{Z})$[1] のような数学的性質などを利用した opaque な条件式を利用することができる。これら opaque な条件式を利用することで、解析者にとっては必ず真 (または偽) であることが分からない条件分岐をプログラムに含ませることができる。

Collberg らは opaque な条件式を用いた難読化手法として、デッドコード挿入、ループ条件拡張、非可約なフローグラフへの変換、の 3 手法について述べている。以下、これら難読化手法について実例を用いて説明を行う。

#### デッドコード挿入による難読化

図 2.5 に opaque な条件式を用いたデッドコード挿入による難読化を行った制御フローグラフを示す。$\mathsf{S_j}$ が難読化の対象となる文、$\mathsf{P^T}$ が opaque である恒真の条件式、$\mathsf{S'_j}$ がデッドコードとなる文である。実行時は $\mathsf{P^T}$ が必ず真であるため $\mathsf{S_j}$ を必ず実行する。しかし、解析者にとって $\mathsf{P^T}$ が必ず真であることが分からないならば、$\mathsf{S_j}$ と $\mathsf{S'_j}$ のどちらを実行するかを確定できない。よってプログラムに含まれる制御フローの複雑度が増すため、制御フローの難読化となる。

```
1    int v, a=5, b=6;
2    v=11 = a+b;                    //v は必ず 11
3    if (b > 5)^T ···               //条件式は必ず真
4    if (random(1,5) < 0)^F ···     //条件式は必ず偽
5        ⋮ (a と b は不変)
6    if (b < 7)^T a++;              //条件式は必ず真
7    v=36 = (a > 5) ? b*b : b;      //v は必ず 36
```

図 2.4 Opaque な変数や述語の例

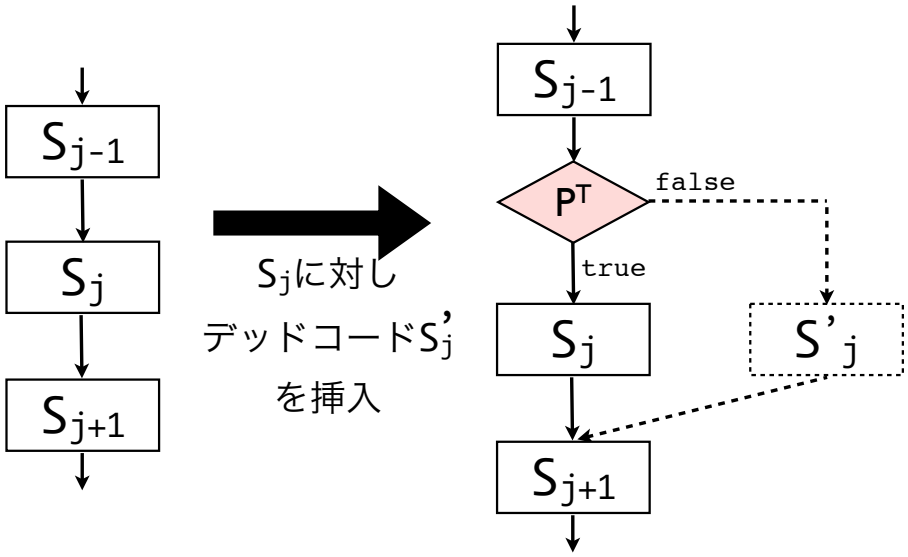



図 2.5　Opaque な条件式を用いたデッドコード挿入による難読化

### ループ条件の拡張

図 2.6 に opaque な条件式を用いたループ条件拡張による難読化を行った制御フローグラフを示す。この難読化の基本アイデアは、ループ条件をループの実行回数に影響しない述語 $P^T$ 又は $P^F$ により拡張することである。

図 2.7 に、$x^2(x+1)^2 \mod 4 \equiv 0$ という数学的性質を用いた while ループの条件拡張による難読化のコード例を示す。変換後の 3 行目のプログラムコードが、難読化によって加えられた恒真となる条件分岐である。

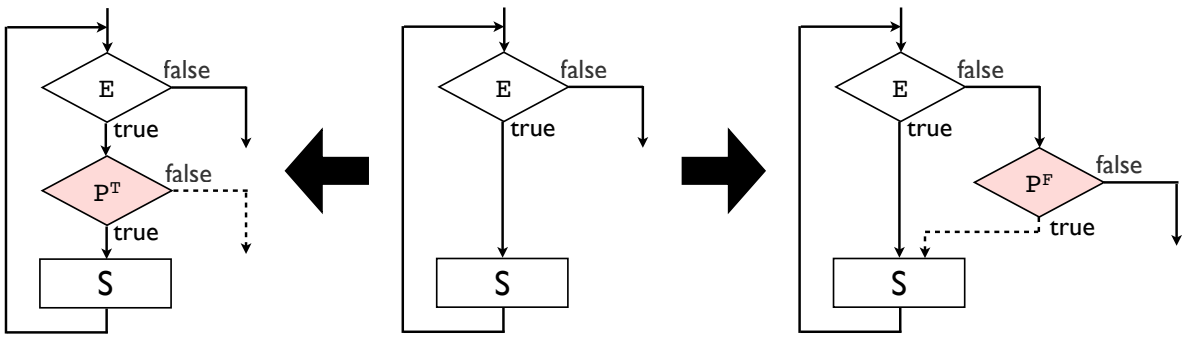



図 2.6　Opaque な条件式を用いたループ条件拡張による難読化

```
1 int i=1;
2 while(i < 100) {
3       ⋮
4       i++;
5 }
```

⇒

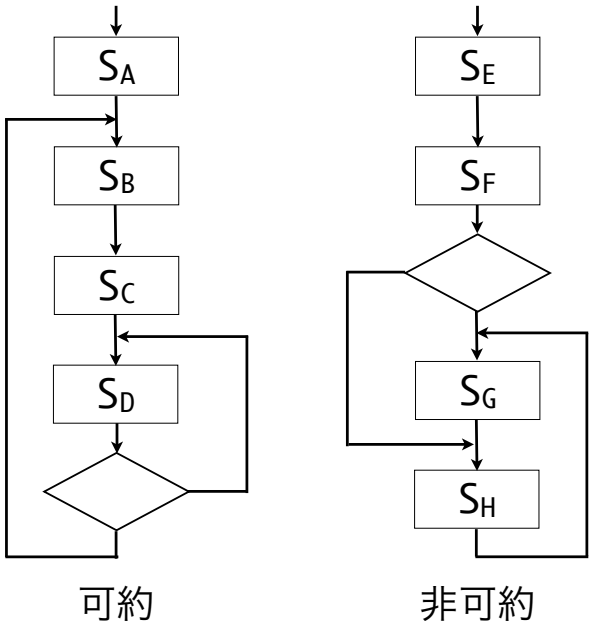
```
1 int i=1, j=100;
2 while( (i<100) &&
3       (((j*j*(j+1)*(j+1))%4) == 0)^T){
4       ⋮
5       i++;
6       j=j*i+3;
7 }
```

図 2.7　Opaque な条件式を用いたループ条件拡張による難読化のコード例

### 非可約なフローグラフへの変換

制御フローグラフにおいて、繰返しループの入口が 1 つであるループを可約ループ、入口が複数あるループを非可約ループと呼ぶ [7]。図 2.8 に、非可約なループを含まない可約なフローと、非可約なループを含む非可約なフローの例、それらのプログラムコードを示す。図 2.8 の非可約なフローでは、$S_G$ と $S_H$ が 1 つのループに対する 2 つの入り口となっている。

goto 文を含まないプログラムのフローグラフは全て可約であるため [8]、Java 言語のようなプログラムコードで goto 文を使えない言語では、非可約なフローとなるプログラムを書くことができない。しかし、一般にプログラムは元の言語よりも制御フローの自由度




```
SA
while(true) {
      SB
      SC
      do {
            SD
      } while(...);
};
```

```
SE
SF
if (...)
      goto label_H;
while(true) {
      SG
label_H:
      SH
}
```

図 2.8　可約なフローと非可約なフローの例とそのプログラムコード

が高いネイティブコードや仮想マシンコードにコンパイルされる。Java 言語においても、コンパイルされ得られる Java 仮想マシンコードでは任意のフローグラフを表現できるため、非可約なフローであるプログラムを作ることができる。

opaque な条件式を利用してプログラムのフローグラフにおけるループを可約から非可約にすることで、while や for などの制御構造では表現できない形に変換することができる。これは、while や for などの制御構造を opaque な条件式によって壊す難読化と言える。

図 2.9 に、while ループに対する opaque な条件式を用いた非可約なフローグラフへの変換による難読化を行った制御フローグラフを示す。図左の制御フローでは、ループの前からループ中にジャンプする偽のフローが追加されている。このフローは、図 2.10 左に示すように Java 言語には存在しない goto 命令を利用しないと表現できない。図右の制御フローでは、ループ中からループ前にジャンプする偽のフローが追加されている。これは、E を条件とするループと $Q^F$ を条件とするループが交差した制御フローグラフとなる。このフローも図 2.10 右に示すように Java 言語には存在しない goto 命令を利用しないと表現できない。

### 条件分岐以外の opaque な要素を利用する難読化

条件分岐以外の opaque な要素を利用する難読化として、オブジェクト指向言語の特性である、異なる機能を同一の名前で扱う多態性を利用した難読化手法も存在する。刑部ら [9] は、インタフェースとメソッドオーバロードを利用した難読化手法を提案し、Java に

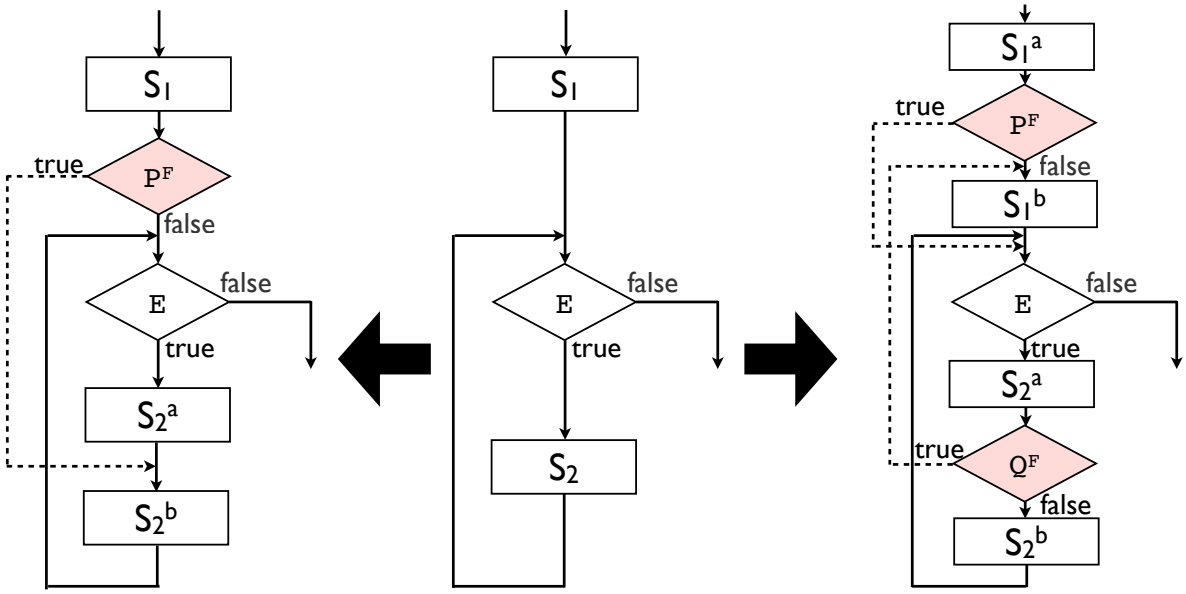



図 2.9　opaque な条件式を用いたフローグラフの非可約化による難読化

```
S1
if (P^F)
    goto l_{S2^b};
while (E) {
    S2^a
l_{S2^b}: S2^b
}
```

```
S1
while (E) {
    S2
}
```

```
S1^a
if (P^F)
    ;
else {
l_{S1^b}: S1^b
}
while (E) {
    S2^a
    if (Q^F) {
        goto l_{S1^b};
    }
    S2^b
}
```

図 2.10　図 2.9 のプログラムコード

おける points-to 解析の困難さに基づき、難読化の効果に一定の理論的評価を与えた。

### 2.1.2　制御フローの難読化の問題点

制御フローの難読化の問題点として、制御フローを複雑にすることによって難読化されたプログラムにオーバヘッドが加わることが考えられる。例として、先に述べた opaque な条件式を用いた難読化は、opaque な条件式の計算処理が難読化によってプログラムに加わるオーバヘッドとなる。あまり複雑ではない opaque な条件式ならば条件式の計算処理によるオーバヘッドも少なくなるが、そのような条件式では解析者にとって opaque であることが簡単に分かってしまう恐れがある。そのため、opaque な条件式を利用した難読化における有効性とオーバヘッドは、トレードオフの関係になることが考えられる。

プログラム中の頻繁には実行されないコードを難読化した場合は、難読化によって加わるオーバヘッドがプログラムの実行時間に与える影響は少ない。しかし、頻繁に実行されるコードに対して制御フローの難読化が行われるならば、難読化で加わる処理も頻繁に実行されるため、プログラムの実行時間に悪影響を与える可能性が考えられる。

そこで、本研究では先に述べた opaque な条件式を利用した難読化手法によって加わるオーバヘッドを削減する手法を考案する。条件分岐と等価な動作を行いつつも、条件分岐よりもオーバヘッドが少ない技法を利用することで、難読化の有益性をそのままにしつつ、プログラムに加わるオーバヘッドが削減できると考えられる。

## 2.2 プログラムの動的最適化

条件分岐と等価な動作を行いつつも、条件分岐よりもオーバヘッドが少ない技法は、一般的に考えれば存在しないように思われる。そこで本研究では、プログラムの実行時に得られる情報を利用して実行時に行う最適化である**動的最適化**が行われることで、結果的に条件分岐よりもオーバヘッドが少なくなる技法について考える。本節では、実行時に動的最適化を行いネイティブなコードを作り出す実行時コンパイルと、動的最適化の一つである仮想メソッド呼出しの最適化について述べる。

### 2.2.1 実行時コンパイル

プログラムを高速に実行するため、実行時にプログラムに対し最適化を行い、より高速なプログラムコードへ変換する手法が使われることがある。これを**実行時コンパイル** (**JIT コンパイル**) と呼び、JIT コンパイルを行う処理系を**実行時コンパイラ** (**JIT コンパイラ**) と呼ぶ。

JIT コンパイラは、Java 言語のような中間コードを仮想マシンで実行する言語処理系や、JavaScript 言語のようなスクリプトをインタプリットして実行する言語処理系で使われることが多い。Java 言語における JIT コンパイラを搭載した実行処理系として、Oracle (旧 Sun Microsystems) の「HotSpot」[10][11]、IBM の「IBM Developer Kits, Java Technology Edition」[12][13][14] などがある。JavaScript 言語における JIT コンパイラを搭載した実行処理系として、Google の「Google V8 JavaScript Engine」[15]、Mozila の「IonMonkey」[16] などがある。

IBM の JIT コンパイラでは、メソッドのインライン展開、例外チェックの除去、共通部分式の除去、ループのバージョニングなどの動的最適化を行っている [13]。

Oracle の JIT コンパイラは、定数の畳込み、基本ブロック内の共通部分式の除去、メソッド呼出しのインライン展開、不要な null 検査の除去、共通した分岐コードパターンの除去、基本ブロックを超えた共通部分式の除去、などの動的な最適化を行っている [11]。また、エスケープ解析を用いたフィールドのローカル変数化やオブジェクトのスタック割当て、正当な範囲である配列インデックスの範囲検査の除去などの最適化についての研究を行っている。

次節では、動的最適化の 1 つである仮想メソッド呼出しのインライン展開について述べる。

### 2.2.2 仮想メソッド呼出しの最適化

関数やメソッドの呼出しを呼出し先で実行するコードで置換える処理を、**インライン展開**と呼ぶ。メソッド呼出しがインライン展開されることで、プログラムカウンタの更新やリターンアドレスをスタックに積むなどの、呼出し先コードへのジャンプ処理や呼出し元へのリターン処理を行わずに済む。

プログラム中のあるメソッド呼出し式において呼出されるメソッドの候補が 1 つのみと特定できる場合は、そのメソッド呼出し式のインライン展開は簡単に行うことができる。しかし、Java 言語のようなオブジェクト指向言語では、メソッドの呼出し先が特定できない**仮想メソッド呼出し**がある。仮想メソッド呼出しは、静的に決まる表記上のオブジェクトの型 (クラス) から呼出すメソッドを決めるのではなく、実行時におけるオブジェクトの実際の型 (クラス) から呼出すメソッドが決まる。これを仮想メソッド呼出しの**多態性**と呼ぶ。図 2.11 に多態性のある仮想メソッド呼出しのコード例を示す。23 行目の仮想メソッド呼出し o.m() について考える。インスタンス変数 o の型は 22 行目の仮パラメータ宣言からクラス O である。そのため、呼出すメソッドは 2 行目のクラス O のメソッド m となり、3 行目の $S_1$ を実行することになるように見える。しかし、変数 o にはクラス O のサブクラスであるクラス A またはクラス B のインスタンスを渡すことができる。よって仮想メソッド呼出し o.m() が呼出すメソッドは、9 行目のクラス A のメソッド m や、16 行目のクラス B のメソッド m、更には他所で定義された A や B 以外の O のサブクラスのメソッド m を呼出す可能性がある。このため、仮想メソッド呼出し o.m() を $S_1$ や $S_2$ や $S_3$ のどのコードを用いてインライン展開すべきかが決まらない。

メソッド呼出しのオーバヘッドは Java のパフォーマンスを低下させる主な理由の 1 つ

```
1  class O {
2      public void m(){
3          S1
4      }
5  }
6
7  class A extends O{
8      @Override
9      public void m(){
10         S2
11     }
12 }
13
14 class B extends O{
15     @Override
16     public void m(){
17         S3
18     }
19 }
20
21 class M {
22     public void test(O o){
23         o.m();
24     }
25 }
```

図 2.11 多態性のある仮想メソッド呼出しのコード例

であるため [13]、仮想メソッド呼出しをインライン展開することの効果は大きいが、仮想メソッド呼出しの多態性によって静的には最適化できない。しかし、実際にプログラムで使われる仮想メソッド呼出しの全てに多態性があることは少なく、呼出されるメソッドの候補は 1 つだけであることが多い。そこで、実行時のオブジェクトの型情報の頻度を用いた、動的情報による仮想メソッド呼出しのインライン展開が JIT コンパイラによって行われることがある。このとき、インライン展開されたメソッドの型と実際に実行すべきオブジェクトの型が異なる場合が考えられる。そのため、展開されたコードが実行すべきコードかどうかを調べる検査が行われる。これを**ガード**と呼ぶ。検査に問題無いならばインライン展開されたコードを実行し、そうでなければ通常の仮想メソッド呼出しを行う。ガードの種類として、インライン展開したメソッドを定義するクラスのアドレスと実行時オブジェクトのクラスを比較するクラステストと、インライン展開の対象となったメソッドのアドレスと実行時オブジェクトのクラスが定義するメソッドのアドレスを比較するメソッドテストがある [17]。

図 2.12 に、図 2.11 の 23 行目の仮想メソッド呼出し o.m() において、o の型が大抵クラス A であった場合のインライン展開とクラステストによるガード処理を表す疑似コードを示す。

仮想メソッド呼出しをインライン展開することによる効果は大きい。Kotzmann ら [11] は、Java HotSpot VM においてインライン展開を行わなかった場合、ベンチマークプログラムの実行速度が最大で 45% ほど低下していることを示している。

仮想メソッド呼出しのオーバヘッドをより削減するため、先のテスト無しにインライン展開をするための手法も研究されている。石崎 [14] は、メソッドのオーバライドが発生しテストが必要になった場合に、コードを書換え通常の仮想メソッド呼出しを行う手法を考案した。この手法により、ベンチマークプログラムにおいて相乗平均で 10.8%〜16.9% の性能向上が得られたことが示されている。

```
    /* oの型は大抵がクラスAとする */
23  if (o.class == A.class) {
        S2
    } else {
        o.m();
    }
```

図 2.12　仮想メソッド呼出しのインライン展開に対するガード処理の疑似コード

# 第 3 章

# 提案手法

2.1.2 節 (p.9) で述べた通り、本研究では opaque な条件式を利用した難読化手法に対し、難読化の有益性を保ちつつも、難読化されたプログラムに加わるオーバヘッドを削減する手法を提案する。

3.1 節 (p.13) では、提案手法の基本アイデアについて説明する。3.2 節 (p.15) と 3.3 節 (p.19) では、2.1.1 節 (p.5) で述べた opaque な条件式を用いた制御フローの難読化に対して提案手法を用いた場合の例と変換の流れについて説明する。本論文では if 文を用いた制御フローの難読化を対象として説明するが、提案手法は if 文以外の条件分岐を用いた制御フローの難読化にも適用することができる。3.4 節 (p.26) では、これら難読化を適用する文の制約について述べる。3.5 節 (p.30) では、Java 言語で書かれたプログラムコードに対して提案手法を適用するための方法について述べる。

## 3.1 基本アイデア

本研究では、opaque な要素として if 文による条件分岐の代わりに、仮想メソッド呼出しを用いる手法を提案する。

2.2.2 節 (p.11) で述べたように、仮想メソッド呼出しはオブジェクトの実行時の型によって呼出すメソッドが決まる、多態性が存在する。仮想メソッド呼出しにおける多態性を実現するメソッドディスパッチを、オブジェクトの型を条件式として呼出すメソッドを決める条件分岐と見なすことができる。つまり、条件式の値と仮想メソッド呼出しのオブジェクトの型を対応させることで、条件分岐と同等の役割を果たす仮想メソッド呼出しを行うことができる。

2.1.1 節の opaque な条件分岐を利用する難読化では、恒真の条件式 $\mathsf{P}^{\mathsf{T}}$ または恒偽の条件式 $\mathsf{P}^{\mathsf{F}}$ の結果を用いて分岐を行う。そこで、仮想メソッド呼出しのオブジェクトの型をこの恒真または恒偽の条件式で決め、仮想メソッド呼出しによって呼出されるメソッドにおいて元の条件分岐の分岐先で行う処理を行うことで、既存の opaque な条件式を利用した難読化に対し仮想メソッド呼出しを利用することができる。

図 3.1 に、恒真条件式 $\mathsf{P}^{\mathsf{T}}$ を用いた opaque な条件分岐に対し、提案手法に基づき $\mathsf{P}^{\mathsf{T}}$

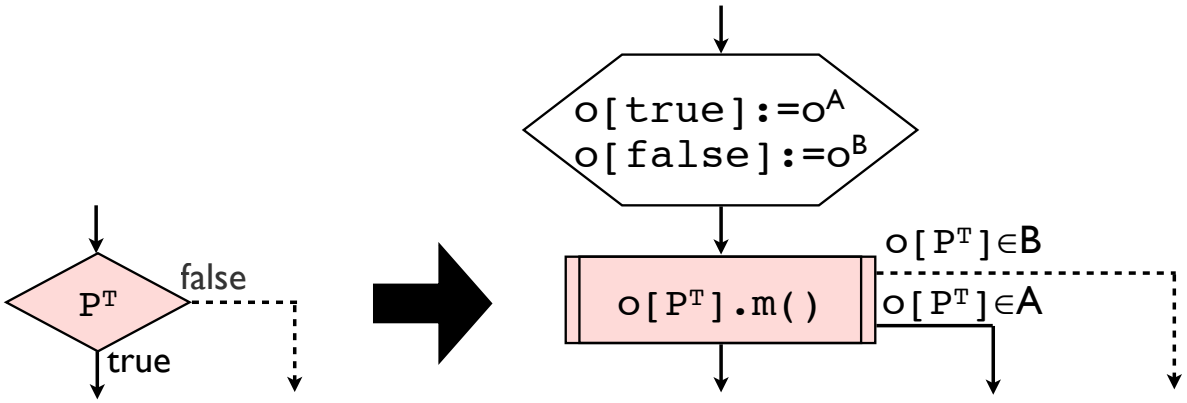



図 3.1　$P^T$ に対する提案手法による仮想メソッド呼出し

とオブジェクトの型を対応させた仮想メソッド呼出しの制御フローグラフを示す。実線のフローが実際に実行される可能性のあるフロー、点線のフローが実際に実行されることは無いダミーのフロー、赤の背景色のあるノードが難読化により加わったオーバヘッドとなるノードである。図左が条件分岐を用いた既存手法、図右が仮想メソッド呼出しを用いた提案手法である。ここで、o はブール値 `true` または `false` を添字にとる連想配列、$o^A$ はクラス A を型とするオブジェクト、$o^B$ はクラス B を型とするオブジェクト、$o[P^T] \in A$ のフローは $o[P^T]$ の型がクラス A の場合のメソッド呼出しフロー、$o[P^T] \in B$ のフローは型がクラス B の場合のメソッド呼出しフローを表す。$o[\mathsf{true}] := o^A$、$o[\mathsf{false}] := o^B$ のノードは o の初期化に関する事前処理とする。

図 3.1 右のフローでは、条件式 $P^T$ は必ず真であるため、仮想メソッド呼出し $o[P^T].m()$ は必ずクラス A のメソッド m を呼出す。しかし、解析者が条件式 $P^T$ は必ず真であることを見抜けない場合、クラス A のメソッド m を呼出すのか、クラス B のメソッド m を呼出すのか分からない。よって、仮想メソッド呼出しを用いた本手法は既存の opaque な条件分岐を用いた難読化と同じ有益性を保っている。

仮想メソッド呼出しはオーバヘッドの大きい処理であるため、このままでは実行速度は遅くなってしまう。しかし 2.2.2 節 (p.11) で述べたように、仮想メソッド呼出しは実行時情報を用いた動的最適化によってインライン展開される可能性が高い。特に、提案手法で用いる仮想メソッド呼出しの呼出し先が実質 1 つのみであるため、インライン展開の効果が最大限に活かされる。

図 3.2 に、図 3.1 右の仮想メソッド呼出し $o[P^T].m()$ に対するインライン展開前後の制御フローグラフを示す。仮想メソッド呼出し $o[P^T].m()$ がインライン展開されることで、$o[P^T]$ の実行時の型が実際にインライン展開されたメソッドの型 (この例ではクラス A) であるかを検査するガードが行われる。そして $o[P^T]$ の実行時の型は必ずクラス A であるため、もはや仮想メソッド呼出し $o[P^T].m()$ は実行されない。

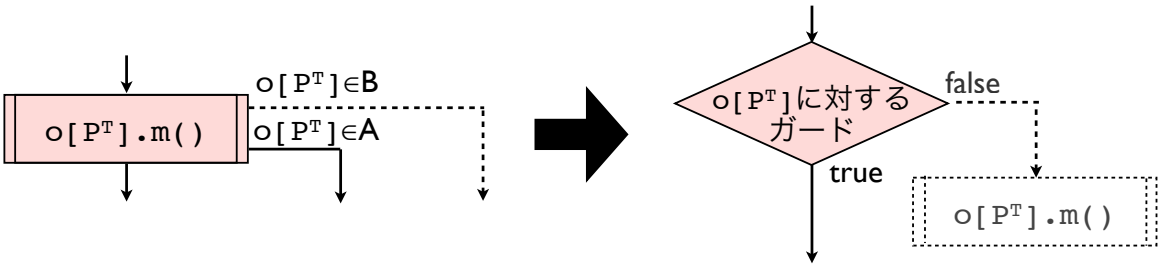



図 3.2　図 3.1 の仮想メソッド呼出しに対するインライン展開前後の制御フローグラフ

図 3.1 左の既存手法による制御フローグラフと、図 3.2 右の提案手法によるインライン展開後の制御フローグラフはほぼ等しい。よって、opaque な条件分岐による難読化で用いる恒真恒偽の条件式を仮想メソッド呼出しで代用しても、JIT コンパイラが行う動的最適化によるインライン展開された後のオーバヘッドはほぼ等しくなると考えられる。更に JIT コンパイラが行う最適化ではガードが多くの場合成功する (値が true となる) ことを想定して最適化を行うため、既存手法と比べオーバヘッドが削減される可能性が見込まれる。

## 3.2　デッドコード挿入による難読化への適用

本節では、2.1.1 節 (p.5) で述べたデッドコード挿入による難読化に対して、仮想メソッド呼出しを用いる提案手法を適用したときの適用例と、実際に難読化を行うための変換の流れについて述べる。

### 3.2.1　提案手法の適用例

図 3.3 に、既存手法である opaque な条件分岐を用いたデッドコード挿入による難読化の制御フローグラフを示す。これは、図 2.5(p. 6) の再掲である。図 3.4 に図 3.3 のプログラムコードを示す。

図 3.5 に、提案手法である opaqe な要素として仮想メソッド呼出しを用いたデッドコード挿入による難読化の制御フローグラフを示す。図 3.6 に図 3.5 のプログラムコードを示す。

図 3.3 の既存手法では、解析者は条件式 $\mathsf{P}^{\mathsf{T}}$ が恒真であることを分からないならば、ノード $\mathsf{S}'_{\mathsf{j}}$ が決して実行されないことが分からない。図 3.5 の提案手法では、解析者は条件式 $\mathsf{P}^{\mathsf{T}}$ が恒真であることを分からないならば、$\mathsf{o}[\mathsf{P}^{\mathsf{T}}]$ の型が必ず B ではないことが分からず、ノード $\mathsf{S}'_{\mathsf{j}}$ が決して実行されないことが分からない。よって、提案手法を適用した難読化は既存手法と同じ有効性を持つことが言える。

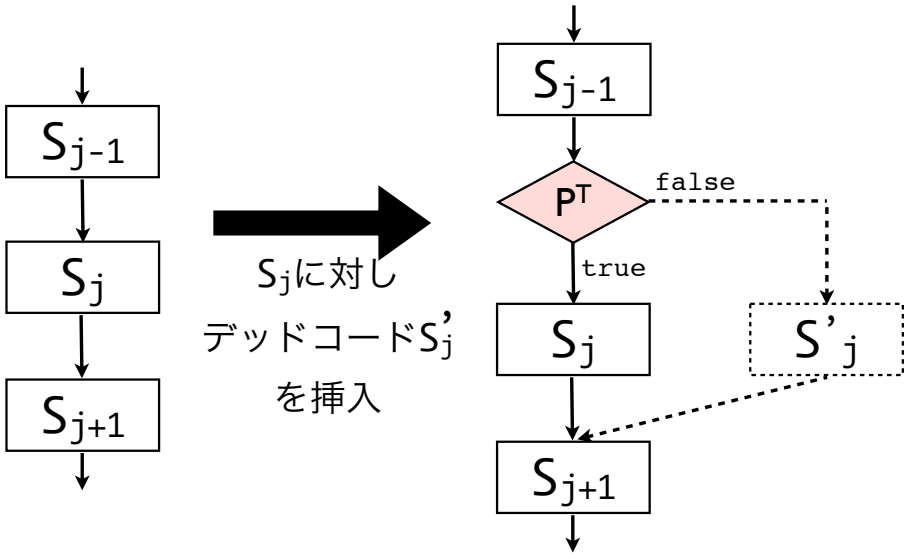



図 3.3　既存手法である条件分岐を用いたデッドコード挿入による難読化

$S_{j-1}$
$S_j$
$S_{j+1}$

$\Rightarrow$

$S_{j-1}$
**if** ($P^T$) {
　　$S_j$
} **else** {
　　$S'_j$
}
$S_{j+1}$

図 3.4　図 3.3 のプログラムコード

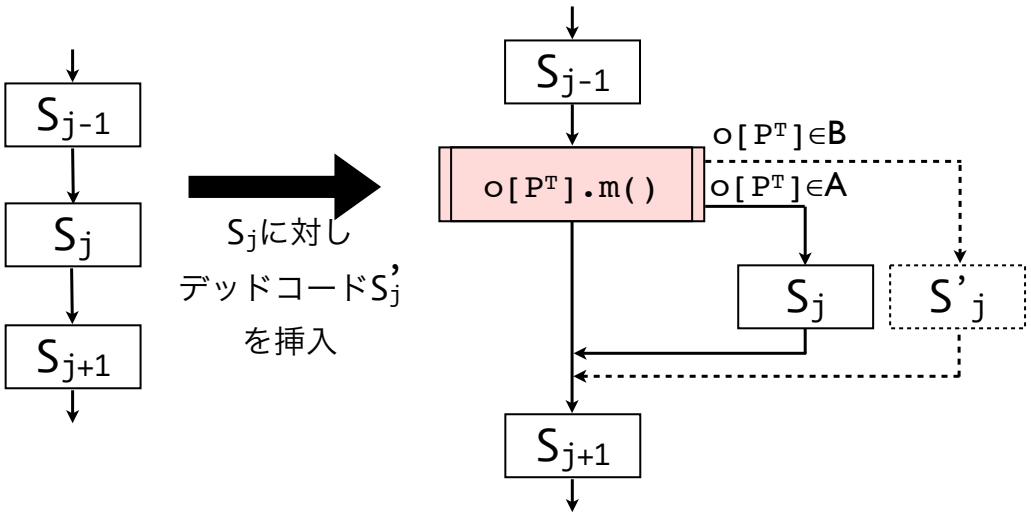



図 3.5　提案手法である仮想メソッド呼出しを用いたデッドコード挿入による難読化

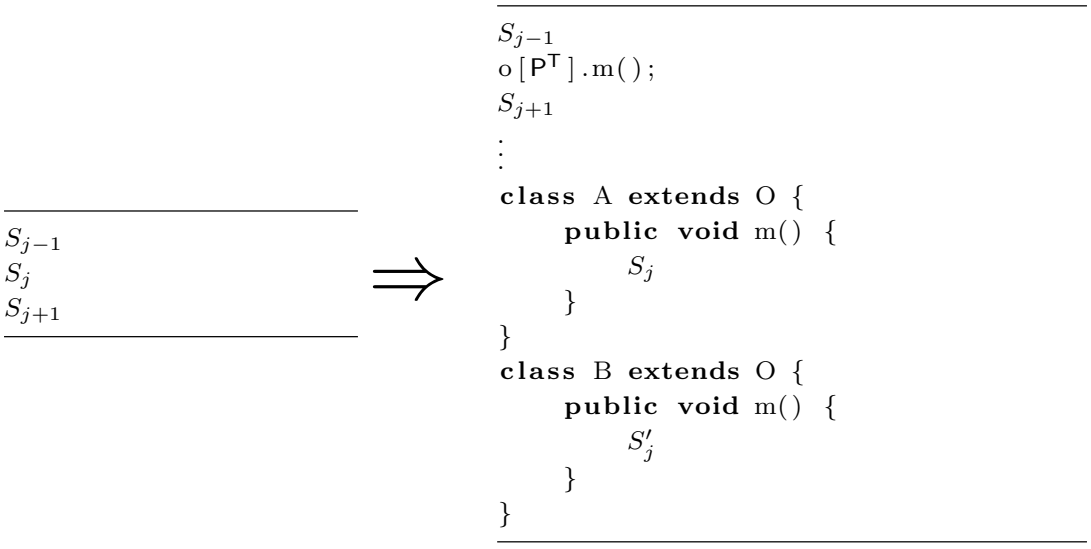



図 3.6　図 3.5 のプログラムコード

図 3.7 に、図 3.5 の難読化後の仮想メソッド呼出しがインライン展開されたときの制御フローグラフを示す。インライン展開されることでクラステストなどのガード処理が行われる。このガード処理は必ず成功するため、インライン展開の最適化がされた後は $o[P^T].m()$ の仮想メソッド呼出しを行うことはなく、展開された $S_j$ の文が実行される。

### 3.2.2　提案手法に基づいた変換の流れ

提案手法に基づいた、仮想メソッド呼出しを用いたデッドコード挿入による難読化の変換の流れを図 3.8 に示す。

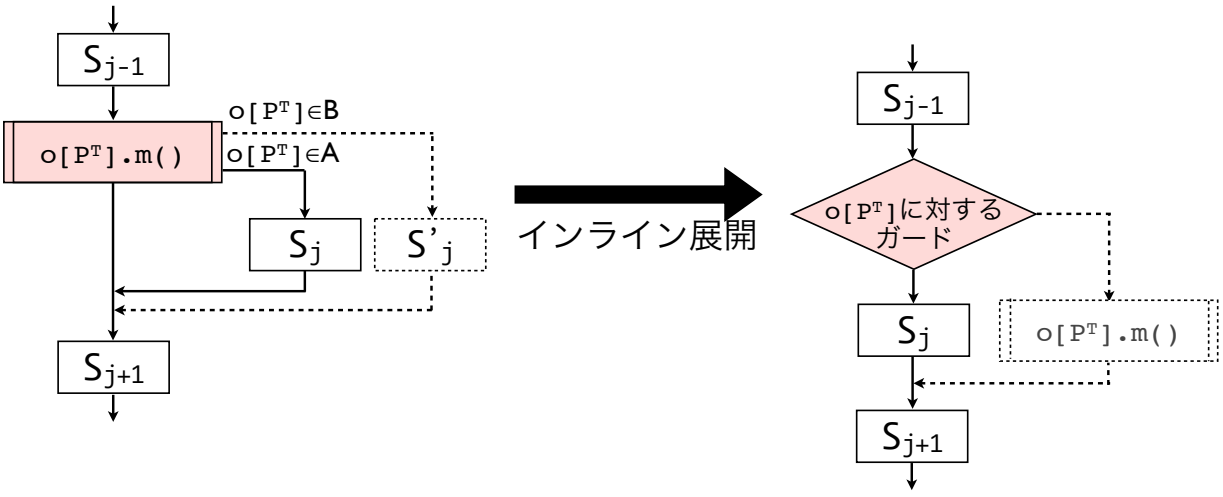



図 3.7　図 3.5 のインライン展開後の制御フローグラフ

$s$ : 難読化対象となる制御遷移の無い 1 つ以上の文の列
$v$ : $s$ が参照する変数全て
$P^T$ : 恒真となる条件式
$C_{body}$ : $s$ を含むクラス本体

1. $C_{body}$ に以下のように定義される内部クラス $O$, $S^T$, $S^F$、フィールド $o$ を追加

```
abstract static class O {
    abstract type m(vを参照するための仮パラメータ);
}

static class S^T extends O{
    type m(vを参照するための仮パラメータ) {
        sのコード
    }
}

static class S^F extends O{
    type m(vを参照するための仮パラメータ) {
        ダミーとなるコード
    }
}

static final O[] o = new O[Boolean];
```

2. $o[true]$ をクラス $S^T$ のインスタンス、$o[false]$ をクラス $S^F$ のインスタンスで初期化
3. メソッド中の $s$ を $o[P^T]$.m($v$ を渡すための実パラメータ) で置換

図 3.8 仮想メソッド呼出しを用いたデッドコード挿入難読化の変換の流れ

$s$ は、難読化の対象となる 1 つ以上の文の列で、$s$ 中の文から別の文への制御遷移 (break や return など) は含まないこととする。

$v$ は、文の列 $s$ が参照している変数の全てである。なお、Java 言語のようにフィールドが内部クラスからも参照できるならば除外できる。なお、$s$ が副作用を持ち、メソッド m の引数が $v$ を参照渡しではなく値渡しする場合は、メソッド m に移された $s$ の副作用が $v$ に反映されないため、注意する必要がある。

$C_{body}$ は、$s$ を含むメソッドを実装するクラスの本体部分である。提案手法では仮想メソッド呼出しのためのクラスを新たに作る必要があるため、これを難読化を行ったクラスの内部クラスとして追加する。

$O$ は仮想メソッド呼出しを行うオブジェクトの表記上の型となるクラス、$S$ は仮想メ

ソッド呼出しを行うオブジェクトの実行時の型となる $O$ の子クラス、$S$ は仮想メソッド呼出しを行うオブジェクトの型の候補となるが実際には使われない $O$ の子クラスである。

$o_j$ は仮想メソッド呼出しを行うオブジェクトの連想配列である。opaque な条件式として恒真な条件式 $\mathsf{P}^{\mathsf{T}}$ を利用するならば、$true$ を添字とする $o$ の要素を $S^T$ のインスタンスと、$false$ を添字とする $o$ の要素を $S^F$ のインスタンスと初期化することで、opaque な条件分岐と等価な動作をする仮想メソッド呼出しを行うことができる。逆に、opaque な条件式として恒偽な条件式 $\mathsf{P}^{\mathsf{F}}$ を利用するならば、$false$ を添字とする $o$ の要素を $S^T$ のインスタンス、$true$ を添字とする $o$ の要素を $S^T$ のインスタンスと初期化する。

## 3.3 制御構造パターンの破壊による難読化への適用

本節では、2.1.1 節 (p.7) で述べた、非可約なフローグラフへ変換することで制御構造パターンを破壊することによる難読化に対して、仮想メソッド呼出しを用いる提案手法を適用したときの適用例と、実際に難読化を行うときの変換の流れについて述べる。

### 3.3.1 提案手法の適用例

既存手法である opaque な条件分岐を用いた制御構造パターンの破壊による難読化の制御フローグラフを図 3.9 に示す。これは、図 2.9 の左と同じフローである。図 3.10 に goto 文を用いた図 3.9 のプログラムコードを示す。

制御構造パターンの破壊による難読化に対して提案手法を元に仮想メソッド呼出しを用いるためには、ループのフローも仮想メソッド呼出しで実現する手法と、ループのフローはそのままにする手法の 2 通りが考えられる。以下、これらを分けて提案を行う。

#### ループフローの仮想メソッド呼出し化による適用例

図 3.9 の opaque な条件分岐 $\mathsf{P}^{\mathsf{F}}$ を仮想メソッド呼出しで等価な動作をさせるためには、$\mathsf{P}^{\mathsf{F}}$ の分岐先である条件分岐 $\mathsf{E}$ と文 $\mathsf{S}_2^{\mathsf{b}}$ を異なるメソッドに移す必要がある。しかし、式 $\mathsf{E}$ と文 $\mathsf{S}_2^{\mathsf{b}}$ を異なるメソッドにしたことにより、$\mathsf{S}_2^{\mathsf{b}}$ から $\mathsf{E}$ への制御フローは goto ではなくメソッド呼出しで行わなければならない。このメソッド呼出しは、$\mathsf{E}$ が真である限り繰返し起こるため、相互再帰呼出しである。

図 3.11 に、相互再帰となる仮想メソッド呼出しを行うことで、図 3.9 に対し提案手法である opaque な要素としてとして仮想メソッド呼出しを用いた制御構造パターンの破壊による難読化の制御フローグラフを示す。図 3.12 に図 3.11 のプログラムコードを示す。再帰呼出しを含む制御フローグラフは、メソッド呼出しのコールとリターンの関係が複雑である。そのため、メソッド呼出しのコールフローとリターンフローを明示的に表したフ

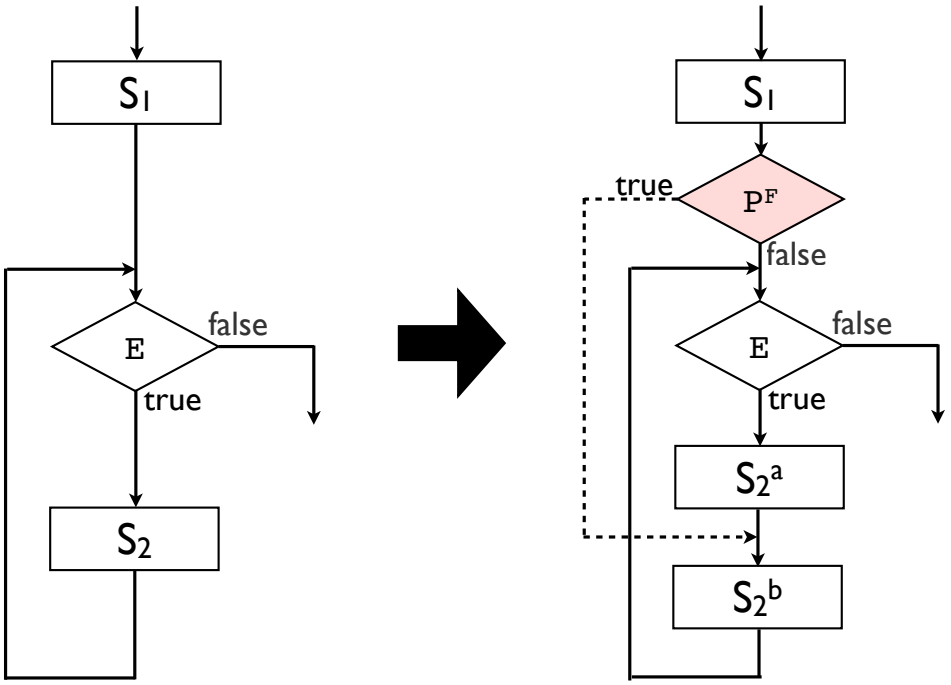



図 3.9　既存手法である条件分岐を用いた制御構造パターンの破壊による難読化

```
S1
while (E) {
    S2
}
```

⇒

```
S1

if (PF) {
    goto label_Sb2;
}

while (E) {
    Sa2
label_Sb2:
    Sb2
}
```

図 3.10　図 3.9 のプログラムコード

ル制御フローグラフ [21] を図 3.11 では用いている。同じ色のフローは、対応したメソッドのコールフローとリターンフローであることを表す。

図 3.11 の赤色のフローは、図 3.9 の $P^F$ から E へのフローに対応する。図 3.11 の緑色のフローは、図 3.9 の $S_2^a$ から $S_2^b$ へのフローに対応する。図 3.11 の黄色のフローは、図 3.9 の $P^F$ から $S_2^b$ へのフローに対応する。図 3.11 の青色のフローは、図 3.9 の $S_2^b$ から E へのフローに対応する。

また図 3.12 から、仮想メソッド呼出しを用いた提案手法では既存手法と違い goto 文を使わずに難読化できることが言える。つまり、本提案手法を用いた難読化は Java ソース

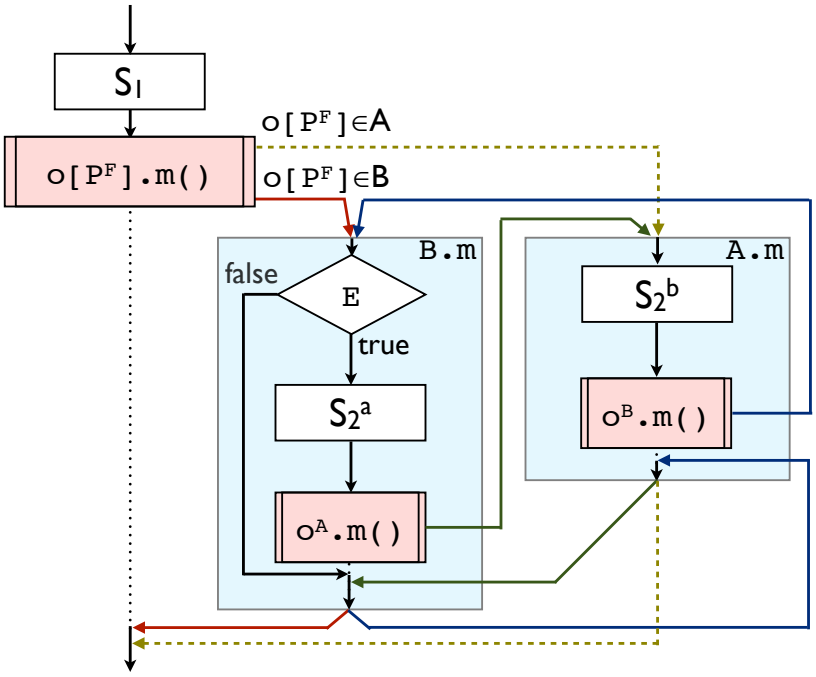



図 3.11　提案手法である仮想メソッド呼出しを用いた制御構造パターンの破壊による難読化 1

```
S1
o[PF].m();
⋮
class A extends O {
    public void m() {
        S2b
        oB.m();
    }
}
class B extends O {
    public void m() {
        if (E) {
            S2a
            oA.m();
        }
    }
}
```

図 3.12　図 3.11 のプログラムコード

プログラムに対して行うことができる。

$o^A.m()$ と $o^B.m()$ は相互再帰になっているため、通常のインライン展開では図 3.13 のようにコードのネストが深くなり、完全にインライン展開することは不可能である。しかし、提案手法で用いる仮想メソッド呼出しは、メソッドの最後に行う処理がメソッドの呼出しである末尾呼出しになる。末尾呼出しは、メソッドを呼出した後に行う処理がない呼出しである。再帰的な末尾呼出しは、スタックを消費しないループに変換できることが一般に知られている。

図 3.11 において、$o^A.m()$ と $o^B.m()$ はどちらとも末尾呼出しである相互再帰であるため、これらは末尾再帰呼出しである。そのため、図 3.14 のように 末尾再帰の最適化を行うことができる。よって、仮想な末尾再帰メソッド呼出しの最適化を行う処理系であれば、図 3.11 のように仮想メソッド呼出しを導入してもそのオーバヘッドを抑えることができる。

### ループフローを残す手法による適用例

仮想な末尾再帰メソッドの呼出しの最適化が行われるならば、先の手法で難読化の効果を保ちつつ、条件分岐の代わりに仮想メソッド呼出しを使うことができる。しかし、そのような最適化が行われるとは限らないため、末尾再帰の最適化に頼らない手法を考案する。

図 3.9 の条件分岐 $P^F$ が真のときの分岐先ブロックである $S_2^b$ を複製することで、ループのフローの形を崩さない形にした制御フローグラフを図 3.15 に示す。図 3.15 右の制御フローグラフに対して、提案手法に基づき仮想メソッド呼出しを用いた場合、ループはそのまま残すことができるため、末尾再帰の最適化に頼る必要がなくなる。

```
S1
if (o[P^F].class == B.class) {
    if (E) {
        S2^a
        if (o^A.class == A.class) {
            S2^b
            if (o^B.class == B.class) {
                if (E) {
                    S2^a
                    if (o^A.class == A.class) {
                        S2^b
            ⋮
```

図 3.13　図 3.11 のインライン展開後の疑似コード (再帰呼出しを直接展開)

```
S1
if (o[P^F].class == B.class) {
L:   if (E) {
         S2^a
         if (o^A.class == A.class) {
             S2^b
             if (o^B.class == B.class) {
                 goto L; //末尾再帰のループへの最適化
             } else {
                 o^B.m();
             }
         } else {
             o^A.m();
         }
     }
} else {
     o[P^F].m();
}
```

図 3.14　図 3.11 のインライン展開後の疑似コード (末尾再帰を最適化により除去)

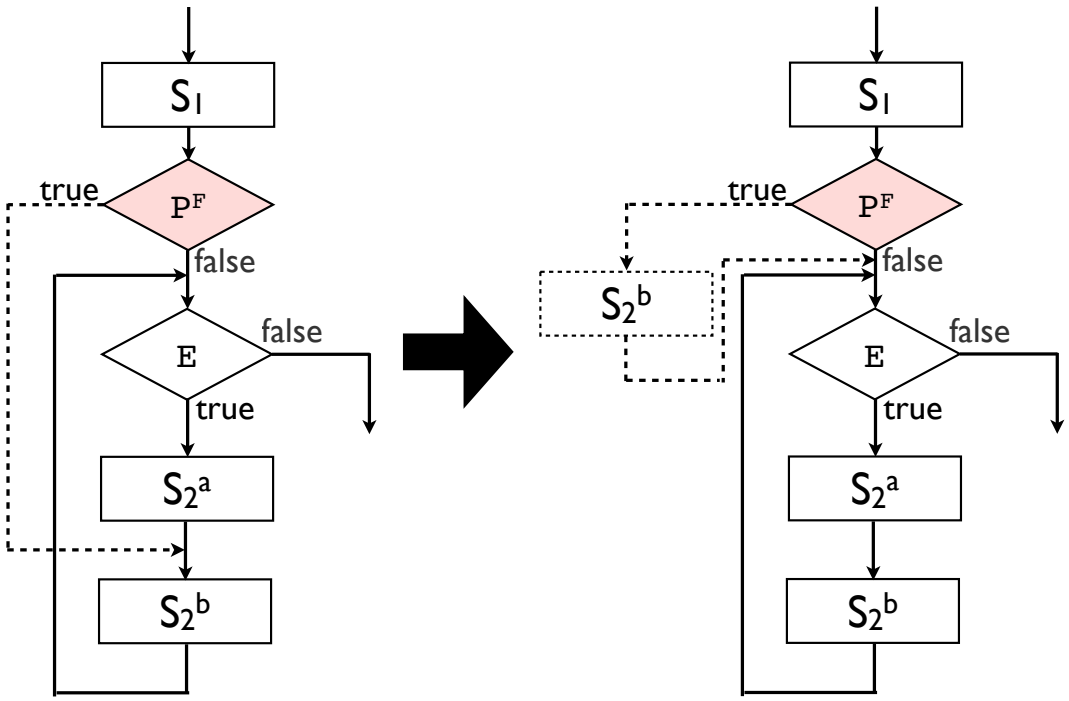



図 3.15　$S_2^b$ を複製したフロー

図 3.16 に、図 3.15 右の制御フローグラフに対し提案手法である opaque な要素として仮想メソッド呼出しを適用した制御構造パターンの破壊による難読化の制御フローグラフを示す。また、図 3.17 に図 3.16 のプログラムコードを示す。 図 3.16 では、複製した式 $S_2^b$ が共通であることを意図させるため、式 $S_2^b$ を実行するためのメソッドを作り出し、そのメソッドを呼出す方法を取っている。図 3.16 の赤色のフローは、図 3.15 の $P^F$ から E へのフローに対応する。図 3.16 の黄色のフローと紫色のフローは、図 3.15 の $P^F$ から $S_2^b$ へのフローに対応する。図 3.16 の緑色のフローは、図 3.15 の $S_2^a$ から $S_2^b$ へのフローに対応する。図 3.16 の青色のフローは、図 3.9 の $S_2^b$ から E へのフローに対応する。

図 3.18 に、図 3.16 がインライン展開されたときの疑似コードを示す。

この手法では、実行時にメソッドの相互再帰は行わないため、仮想な末尾再帰メソッド呼出しの最適化を行わない処理系でもオーバヘッドを抑えた実行ができる。しかし問題点として、この手法を用いた難読化は、非可約なフローへの変換による難読化よりも、デッドコード挿入による難読化に似ていること、変換後のフローは非可約ではないことの 2 点がある。図 3.15 右の制御フローグラフから、ループのフローを複雑にする難読化ではなく、$S_1$ とループの間にデッドコードである $S_2^b$ を挿入した難読化であると言える。よって、この手法は元となった難読化の有効性が変わってしまっていると言える。

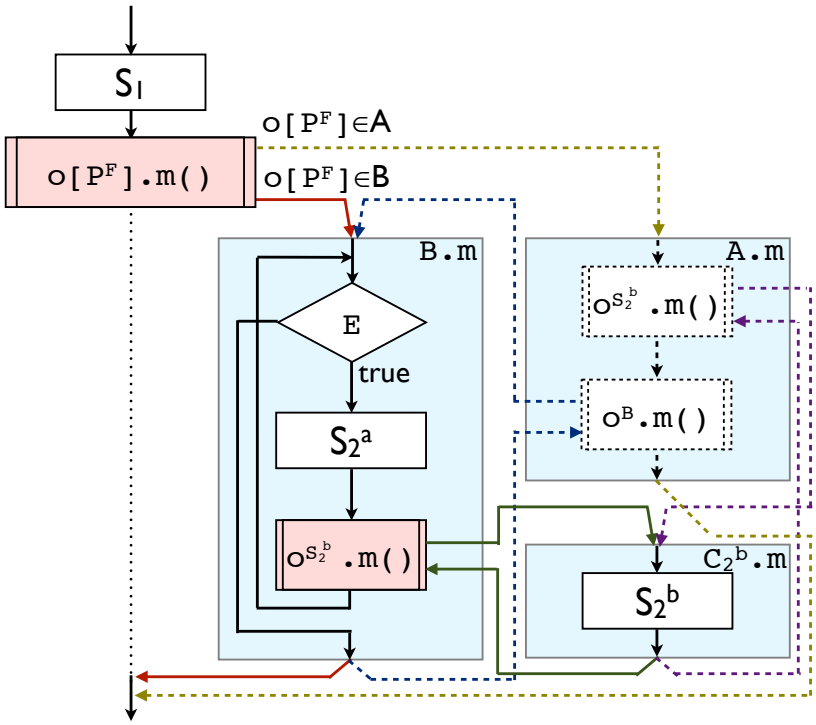



図 3.16　提案手法である仮想メソッド呼出しを用いた制御構造パターンの破壊による難読化 2

```
S1
o[P^F].m();
⋮
class A extends O {
    public void m() {
        o^{C_2^b}.m();
        o^B.m();
    }
}
class B extends O {
    public void m() {
        while (E) {
            S_2^a
            o^{C_2^b}.m();
        }
    }
}
class C_2^b extends O {
    public void m() {
        S_2^b
    }
}
```

図 3.17　図 3.16 のプログラムコード

```
S1
if (o[P^F].class == B.class) {
    while (E) {
        S_2^a
        if (o^{C_2^b}.class == C_2^a.class) {
            S_2^b
        } else {
            o^{C_2^b}.m();
        }
    }
} else {
    o[P^F].m();
}
```

図 3.18　図 3.16 のインライン展開後の疑似コード

### 3.3.2 提案手法に基づいた変換の流れ

本節では、末尾再帰の仮想メソッド呼出しを最適化する処理系があると仮定し、p.19 で述べた提案手法に基づいた制御構造パターンの破壊による難読化の変換の流れについて述べる。

制御構造パターンの破壊による難読化は、$\mathsf{P}^{\mathsf{F}}$ の挿入位置とジャンプ先により図 3.19 のようなパターンとその組合わせであると考えることができる。ここでは、図 3.19 左上のパターンを元に変換の流れを述べる。

提案手法に基づいた、仮想メソッド呼出しを用いた制御構造パターンの破壊による難読化の流れを図 3.20 に示す。

$S^{loop}$ は難読化の対象となるループ本体の文の列で、$s_3$ と $s_4$ から構成される。$s_2, s_3, s_4$ には別の文への制御遷移は含まないこととする。

$O^L$ はループの条件処理と $s_3$ を繰返すための内部クラスである。$O^L$ のメソッドは、条件処理が真ならば $O^T$ のメソッドを呼出し、$O^T$ のメソッドは $O^L$ のメソッドを呼出すため、相互再帰となっている。条件処理が偽ならば相互再帰呼出しのリターン処理を行うため、最終的には $o[P^F].\mathrm{m}(v)$ にリターンされる。

## 3.4 難読化対象となる文について

本提案手法では、難読化を行うため複数の文をメソッド内部に移している。図 3.8 に示すデッドコード挿入による難読化では文 $s$、図 3.20 に示す制御構造パターンの破壊による難読化では文 $\{s_*, S^{loop}\}$ が、本来存在したメソッドとは別の新たに作成したメソッドに移動される。

移動される文において break や return などの別の文への制御が遷移する文が無い場合は、文をそのまま別のメソッドに移動することによる問題は (変数の副作用を除けば) 無い。しかし、移動される文において別の文への制御遷移となる文が存在する場合は、文をそのまま別のメソッドに移すことができなくなる。以下、それら別の文への制御遷移を含む場合について述べる。

#### 文に break や continue を含む場合

デッドコード挿入による難読化の対象となる文に break や continue を含む場合、移動先のメソッドから break や continue の遷移先である移動前のメソッドへ遷移するには、リターンを行う必要がある。しかし、メソッドのリターン先は必ずメソッドの呼出し元であるため、そのままでは braek や continue の本来の遷移先のコードを実行できない。そ

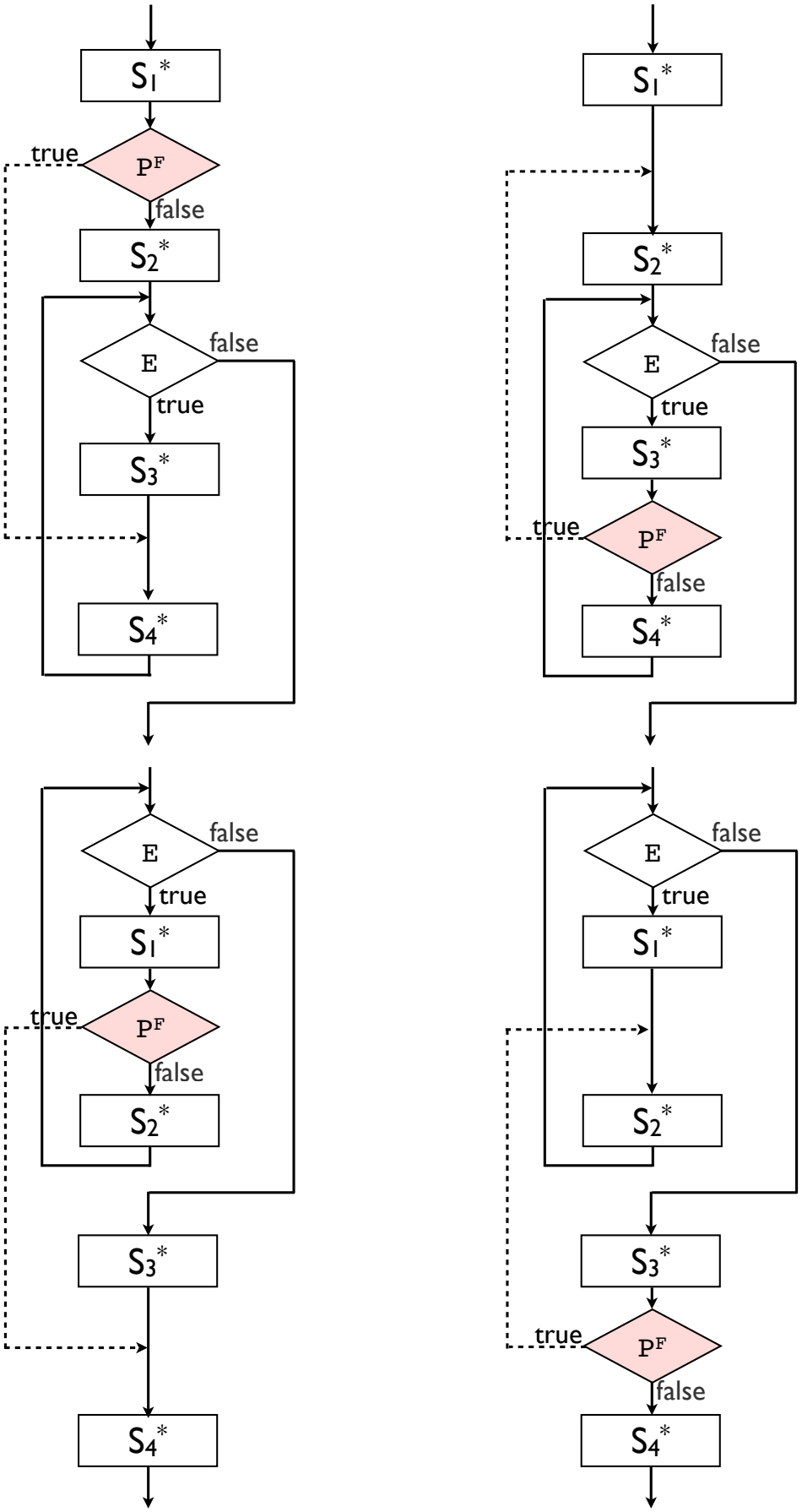



図 3.19　制御構造パターンの破壊による難読化のパターン

$S^{loop}$ = $\{s_3, s_4\}$
$v$ : $s_2$と $S^{loop}$が参照する変数全て
$P^F$ : 恒偽となる条件式
$C_{body}$ : $s_1, s_2, S^{loop}$を含むクラス本体

1. $C_{body}$ に以下のように定義される内部クラス $O$, $O^F$, $O^T$, $O^L$ を追加

```
abstract static class O {
    abstract type m(v);
}

static class O^F extends O{
    type m(v) {
        s2
        oloop.m(v);
    }
}

static class O^T extends O{
    type m(v) {
        s4
        o_loop.m(v);
    }
}

static class O^L extends O{
    type m(v) {
        if (E) {
            s3
            o[true].m(v);
        }
    }
}
```

2. $C_{body}$ に以下のように定義されるフィールド $o$,$o_{loop}$ を追加

```
static final O[] o = new O[Boolean];
static final O o_loop = new O^L();
```

3. $o[true]$ をクラス $O^T()$ のインスタンス、$o[false]$ をクラス $O^F()$ のインスタンスで初期化
4. メソッド中の $\{s_2, S^{loop}\}$ を $o[P^F].\mathrm{m}(v)$ で置換

図 3.20 仮想メソッド呼出しを用いた制御構造パターンの破壊の変換規則

のため、グローバルなフィールドなどを利用し、移動先のメソッドでの break や continue に相当するリターンと、移動された文の終了による本来のリターンを区別し、break や continue に相当するリターンならば移動元メソッドにおいて break や continue を行う必要がある。図 3.21 に、break を含んだ文を提案手法に基づき難読化を行った例を示す。

制御構造パターンの破壊による難読化の対象となる文に break や continue を含む場合、break や continue の対象となるループが、難読化の対象となるループである場合と、難読化の対象外である外側のループである場合が考えられる。

難読化の対象となるループである場合の break は、ループを相互再帰にする手法では return を、ループを残す手法ではループ内で break を行えばよい。難読化の対象となるループである場合の continue は、ループを相互再帰にする手法ではループの条件式を行うメソッドの呼出しを、ループを残す手法ではループ内で continue を行えばよい。図 3.22 に、continue を含んだ文をループを相互再帰にする提案手法に基づき難読化を行った例を示す。

難読化の対象外である外側のループである場合は、先のデッドコード挿入と同様にグローバルなフィールドを利用する必要がある。

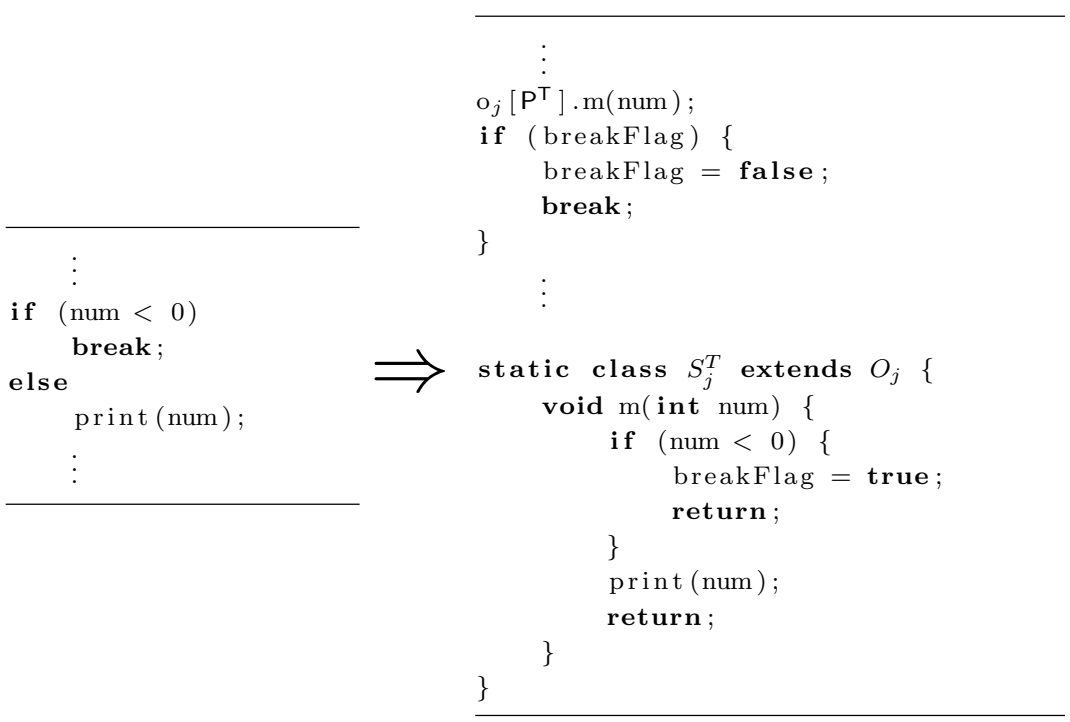



図 3.21　デッドコード挿入による難読化における break の対応

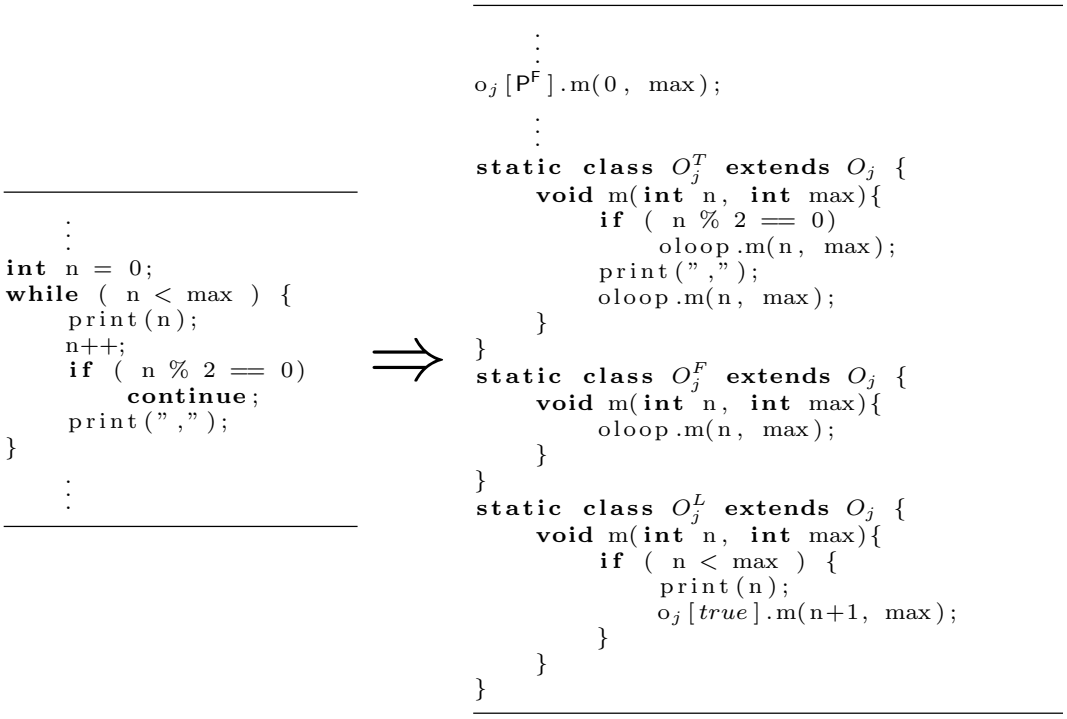



図 3.22　制御構造パターンの破壊による難読化における continue の対応

#### 文に return を含む場合

難読化の対象となる文に reutrn を含む場合、移動先のメソッドから移動前のメソッドのリターン処理を行うためには、一度移動先のメソッドをリターンで抜けてから移動前のメソッドのリターンを行う必要がある。そのため、先と同様にグローバルなフィールドを利用する必要がある。

## 3.5　Java プログラムへの提案手法の適用

関連研究として示した通り、Java 言語で書かれたプログラムは難読化の有用性があり、更に JIT コンパイラによって動的最適化が行われることが多い。そこで本節では、Java 言語で書かれたプログラムコードに対して提案手法を適用するための方法について述べる。

### 3.5.1　ブール値を添字とする連想配列の代用

3.1 節 (p.13) の図 3.1 に示す通り、提案手法ではブール値を要素とする連想配列を利用して仮想メソッド呼出しを行う。しかし、Java 言語の標準パッケージで提供される連

想配列である HashMap クラス [24] は、Boolean クラスの TRUE オブジェクトおよび FALSE オブジェクトのハッシュコード値から配列の添字を求めるため、難読化として用いるには効率が悪い。

そこで、ブール値の true/false を 0/1 として扱い、1 次元 2 要素の配列を利用して仮想メソッド呼出しを行う手法が考えられる。これは、true のハッシュコード値を 0、false のハッシュコード値を 1 としたときの HashMap クラスが行う連想配列の処理を単純化したものである。図 3.23 に、1 次元 2 要素の配列を仮想メソッド呼出しを行うオブジェクトとして利用し、条件式の真ならば 0 番目の要素、偽ならば 1 番目の要素を参照するプログラムのコードとその制御フローグラフを示す。ここで、制御フローグラフに現れる変数 x は配列参照用の一時変数である。

1 次元 2 要素の配列を利用する手法の問題点として、制御フローグラフが図 3.3(p.16) で示した条件分岐を用いたデッドコード挿入による難読化を行った制御フローグラフとほぼ等価になるため、仮想メソッド呼出しを用いる利点が薄れてしまう。そのため、条件分岐や条件演算子を使わない手法を考える必要がある。

そこで、あらかじめ条件式の計算した結果を配列にキャッシュする手法が考えられる。条件式の引数の個数を次元とする配列を用意し、条件式の評価した結果を引数に対応した添字を要素とする配列に代入する。また参照時に剰余計算を行うことで、無限である引数の取る範囲を有限の配列長に抑えることができる。2 つの引数 x,y を持つ恒真条件式 $P^T(x, y)$ に対し、キャッシュした条件分岐を利用したプログラムコード表現とその制御フローグラフを図 3.24 に、キャッシュした仮想メソッド呼出しを利用したプログラムコード表現とその制御フローグラフを図 3.25 に示す。ここで、$v$ は変数バインディングで参

```
O[] o = {oA, oB}
    ⋮
o[PT ? 0 : 1].m(v);
```

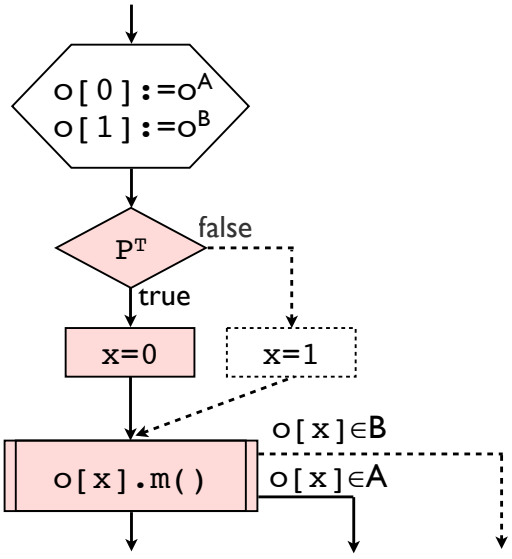



図 3.23　1 次元 2 要素の配列を利用した仮想メソッド呼出し

照可能な変数の集合、$v_x$ と $v_y$ はそのような変数を表す。

条件式をキャッシュする手法の問題点として、配列の初期化処理や参照のための剰余計算や絶対値の計算が新たな実行時間に対するオーバヘッドになること、利用する恒真恒偽の条件式の引数の数や難読化を行う箇所の数に比例してメモリの使用量が増加することが考えられる。

### 3.5.2 ローカル変数の扱い

Java 言語ではメソッドの引数が値渡しされるため、副作用を持つ文をメソッド化するとローカル変数への代入ができなくなる。例として、図 3.26 に示すメソッドの 4 行目に対しデッドコード挿入の難読化を行うためには、ローカル変数 z への代入を考慮する必要がある。これを解決する手法は次の二通りが考えられる。

```
boolean[] b = new boolean[n][n];
for (int x = 0; x < n; x++)
    for(int y = 0; y < n; y++)
        b[x][y] = P^T(x,y);
    ⋮
if(b[v_x ∈ v mod n][v_y ∈ v mod n]) {
    ...
} else {
    ...
}
```

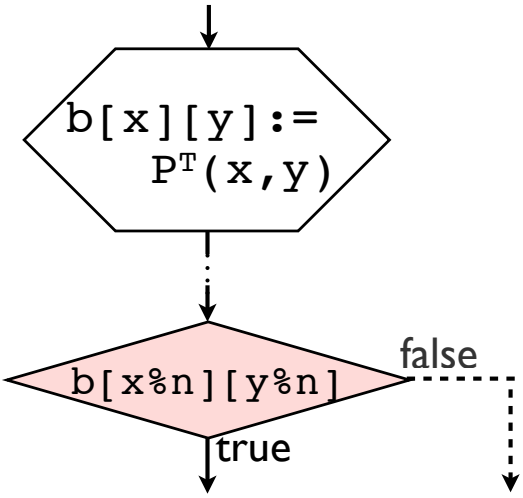



図 3.24 $\mathsf{P}^{\mathsf{T}}(\mathsf{x},\mathsf{y})$ の代用として 2 次元の配列を利用した条件分岐

```
O[] o = new O[n][n];
for (int x = 0; x < n; x++)
    for(int y = 0; y < n; y++)
        o[x][y] = P^T(x,y)?o^A:o^B;
    ⋮
o[v_x ∈ v mod n][v_y ∈ v mod n].m(v);
```

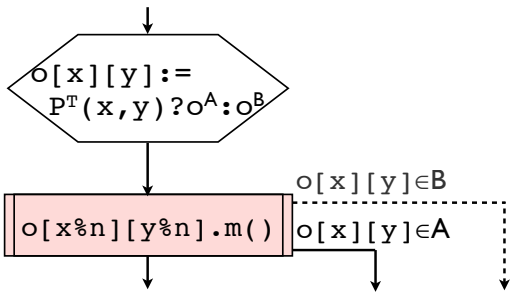



図 3.25 $\mathsf{P}^{\mathsf{T}}(\mathsf{x},\mathsf{y})$ の代用として 2 次元の配列を利用した仮想メソッド呼出し

```
1 int test(int x, int y){
2     int z = 0;
3
4     z += x / y;
5     return z;
6 }
```

図 3.26　副作用を持つ文のあるメソッドの例

### メソッドの返り値による代入

メソッド内部に移動された文が参照する変数をメソッドの仮パラメータから参照し、副作用のある文の計算式の値をメソッドの返り値とする。呼出し元では仮想メソッド呼出しの返り値を利用して元の副作用を起こす。

図 3.27 に、図 3.26 の 4 行目の文をメソッドの返り値を利用してメソッド化したときのプログラムコードを示す。図 3.27 において、4 行目における o[···] の実行時の型を常にクラス A とすると、クラス A のメソッド m の仮想呼出しが行われ、10 行目の x/y の結果がメソッドのリターン値となり、z に加算される。しかし、解析者にとっては 4 行目における o[···] の型がクラス A かクラス B か分からないため、10 行目の x/y を計算するか、16 行目の y/x を計算するかは分からない。

```
1 int test(int x, int y){
2     int z = 0;
3
4     z += o[···].m(x,y);
5     return z;
6 }
7
8 static class A extends O {
9     int m(int x, int y) {
10        return x / y;
11    }
12 }
13
14 static class B extends O {
15    int m(int x, int y) {
16        return y / x;
17    }
18 }
```

図 3.27　メソッドの返り値による代入の例

メソッドの返り値を利用する手法の問題点として、副作用のある文を 1 つしかメソッド化できない。これは、メソッドの返り値は 1 つしか返すことができないからである。また、ダミーとなる文の処理が制限されるという問題も考えられる。例として、図 3.27 の難読化では変数 z に加算される値を分からなくすることはできるが、変数 z に加算をするか減算をするか分からなくさせることや、変数 z と変数 y のどちらに加算をするのか分からなくさせることはできない。

### ローカル変数のフィールド化

難読化対象となる文を含むメソッドのローカル変数と仮パラメータに対し、それらをフィールドとする変数参照用のクラスを新たに生成する。難読化対象メソッドの先頭で変数参照クラスのインスタンス生成を行い、難読化による仮想メソッド呼出しでは生成した変数参照オブジェクトを渡す。メソッド化された文は変数参照オブジェクトのフィールドに対して副作用を起こすことで、呼出し元メソッドから副作用を受けた変数を参照することができる。この手法は、プリミティブ型を参照型に変換するボックス化 (boxing) の考えに似ている。

図 3.28 に、図 3.26 の 4 行目の文をローカル変数のフィールド化を利用してメソッド化したときのプログラムコードを示す。図 3.28 において、2 行目は変数参照用クラスのインスタンス生成、3 行目と 4 行目は変数参照オブジェクト v へのメソッド引数値 x と y の格納、5 行目は図 3.26 の 2 行目でのローカル変数 z の初期化に相当する処理を行う。7 行目の仮想メソッド呼出しにおいて、実行時の o[···] の型を常にクラス A とすると、クラス A のメソッド m の仮想メソッド呼出しが行われ、19 行目の文が実行され v.z には v.x/v.y の結果が加算される。しかし、解析者にとっては 7 行目の o[···] の型がクラス A かクラス B か分からないため、19 行目の v.z+=v.x/v,y を実行するか、25 行目の v.y-=v.x/v.z を実行するか分からない。

ローカル変数のフィールド化する手法の問題点として、変数をオブジェクトのフィールド扱いするため、メモリのヒープ領域を使うため実行速度の低下が考えられる。しかしこれは、Java Hotspot VM の研究プロジェクト [11] である、エスケープ解析を用いたフィールドのローカル変数化やオブジェクトのスタック割当てによって最適化される可能性が考えられる。

```
1  int test(int x, int y){
2      V v = new V();
3      v.x = x;
4      v.y = y;
5      v.z = 0;
6
7      o[···].m(v);
8      return v.z;
9  }
10
11 static class V {
12     int x;
13     int y;
14     int z;
15 }
16
17 static class A extends O {
18     void m(V v) {
19         v.z += v.x / v.y;
20     }
21 }
22
23 static class B extends O {
24     void m(V v) {
25         v.y -= v.x / v.z;
26     }
27 }
```

図 3.28　ローカル変数のフィールド化の例

# 第 4 章

# 変換器の実装

本章では、3.2 節 (p.15) で示した仮想メソッド呼出しを用いたデッドコード挿入による難読化を Java ソースプログラムに対して行い、難読化された Java ソースプログラムを出力する変換器の実装について簡単に述べる。

## 4.1 変換器の構成

パーサジェネレーターと構文解析器ジェネレーターを生成する JavaCC と JJTree[22] をベースとし、Java ソースプログラムから難読化された Java ソースプログラムを生成する変換器を生成した。Java の構文解析器ジェネレーターは、Java 1.5 用の既存の設定ファイルを利用した [23]。

実装した変換器の構成を図 4.1 に示す。まず、JavaCC と JJTree が生成したパーサと構文解析器を利用して、難読化を行う Java プログラムコードの抽象構文木を得る。この抽象構文木に対して難読化を行うが、JJTree が生成した構文解析器によって得られる抽象構文木では、難読化処理やコードの出力を行うことが難しい。そこで、JJTree が生成

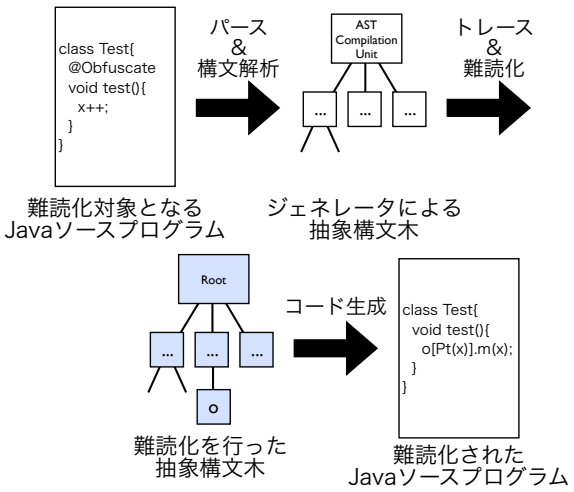



図 4.1 実装した変換器の構成

した抽象構文木を元に、難読化処理やコードの出力が行いやすい独自の抽象構文木に変換する。独自の抽象構文木に対して難読化処理を行い、これを元に難読化された Java ソースプログラムのコードを生成する。

## 4.2 変換器が行う難読化

実装した変換器では、図 3.8 の変換規則を元にプログラムの難読化を行っている。ここで、以下に示す方法で変換を行っている。

- 難読化の対象となる文は、@Obfuscate アノテーションがついているメソッドの別の文への制御遷移を含まない文全て
- ローカル変数と仮パラメータは 3.5.2 節 (p.34) で述べた通り、それら変数をフィールドとするクラスを新たに生成し参照
- 変数の初期化を行っているローカル変数の宣言文は、フィールドへの代入文に変換
- opaque な条件式で参照する変数は、式の変数スコープ内の整数変数からランダムに選択
- 難読化により生成されるクラスの名前は、既存のクラスとの重複が起きないようダラー記号 ($) を利用する
- ローカル変数と仮パラメータの名前を、$\$var i (i \geq 0)$ の形に統一
- ダミーとなるクラスのメソッドには “DummyCode” というコメントのみ記述

## 4.3 プログラムの変換例

プログラム 4.1 に、変換器で変換を行う Java プログラムコードを示す。18 行目の test メソッドが難読化の対象である。

プログラム 4.1 変換の対象となるプログラムコード

```
1
2  public class Test {
3      public static void main(String[] args) {
4          int x, y;
5
6          try {
7              x = Integer.parseInt(args[0]);
8              y = Integer.parseInt(args[1]);
9          } catch (Exception e) {
10             x = 3;
11             y = 2;
12         }
```

```
13
14          test(x, y);
15      }
16
17      @Obfuscate
18      static void test(int x, int y) {
19          int z = x + y;
20
21          if (z > 0) {
22              System.out.println("x:" + x);
23          } else {
24              System.out.println("y:" + y);
25          }
26
27          System.out.println("z:" + z);
28      }
29  }
```

プログラム 4.2 に、実際に変換器によって難読化された Java プログラムコードを示す。実装した変換器では、プログラム 4.1 における 19 行目、22 行目、24 行目、27 行目の 4 カ所の文を対象に、恒真な条件式として $7x^2 - 1 \neq y^2$ を利用して難読化を行った。プログラム 4.2 の 32 行目、46 行目、60 行目、74 行目が難読化により移動された文である。

プログラム 4.2　変換器により難読化されたプログラムコード

```
1  public class Test {
2      public static void main(String[] args) {
3          int x, y;
4          try {
5              x = Integer.parseInt(args[0]);
6              y = Integer.parseInt(args[1]);
7          } catch (Exception e) {
8              x = 3;
9              y = 2;
10         }
11         new Test().test(x, y);
12     }
13
14     void test(int $var0, int $var1) {
15         $V $v = new $V();
16         $v.$var0 = $var0;
17         $v.$var1 = $var1;
18         $o0[(7 * $v.$var0 * $v.$var0 - 1 != $v.$var1 * $v.$var1) ? 0 :
               1].m(this, $v); //$v.$var2=$v.$var0+$v.$var1
19         if ($v.$var2 > 0) {
20             $o1[(7 * $v.$var1 * $v.$var1 - 1 != $v.$var1 * $v.$var1) ? 0
                   : 1].m(this, $v); //System.out.println("x:"+$v.$var0)
21         }
22         else {
```

```
23              $o2[(7 * $v.$var0 * $v.$var0 - 1 != $v.$var1 * $v.$var1) ? 0
                    : 1].m(this, $v); //System.out.println("y:"+$v.$var1)
24          }
25          $o3[(7 * $v.$var1 * $v.$var1 - 1 != $v.$var1 * $v.$var1) ? 0 :
                1].m(this, $v); //System.out.println("z:"+$v.$var2)
26      }
27
28      private static final $O[] $o3 = { $O.$o3a, $O.$o3b };
29
30      static class $3A extends $O {
31          void m(Test2 $this, $V $v) {
32              System.out.println("z:" + $v.$var2);
33          }
34      }
35
36      static class $3B extends $O {
37          void m(Test2 $this, $V $v) {
38              //DummyCode
39          }
40      }
41
42      private static final $O[] $o2 = { $O.$o2a, $O.$o2b };
43
44      static class $2A extends $O {
45          void m(Test2 $this, $V $v) {
46              System.out.println("y:" + $v.$var1);
47          }
48      }
49
50      static class $2B extends $O {
51          void m(Test2 $this, $V $v) {
52              //DummyCode
53          }
54      }
55
56      private static final $O[] $o1 = { $O.$o1a, $O.$o1b };
57
58      static class $1A extends $O {
59          void m(Test2 $this, $V $v) {
60              System.out.println("x:" + $v.$var0);
61          }
62      }
63
64      static class $1B extends $O {
65          void m(Test2 $this, $V $v) {
66              //DummyCode
67          }
68      }
69
70      private static final $O[] $o0 = { $O.$o0a, $O.$o0b };
```

```
71
72      static class $0A extends $O {
73          void m(Test2 $this, $V $v) {
74              $v.$var2 = $v.$var0 + $v.$var1;
75          }
76      }
77
78      static class $0B extends $O {
79          void m(Test2 $this, $V $v) {
80              //DummyCode
81          }
82      }
83
84      static class $V {
85          int $var0;
86          int $var1;
87          int $var2;
88      }
89
90      static abstract class $O {
91          abstract void m(Test2 $this, $V $v);
92
93          static final $O $o0a = new $0A(), $o0b = new $0B(),
94                  $o1a = new $1A(), $o1b = new $1B(),
95                  $o2a = new $2A(), $o2b = new $2B(),
96                  $o3a = new $3A(), $o3b = new $3B();
97      }
98  }
```

# 第 5 章

# 評価

第 3 章の冒頭で述べた通り、本研究では既存の難読化手法の性能を保ちながらもオーバヘッドを減らす手法を提案した。そこで、提案手法の評価では難読化の有効性を考慮せず、実行時間に関するオーバヘッドのみ考慮する。そのため、本章では以下のような条件を元に難読化を行い、難読化されたメソッドの実行時間を測定した。

- デッドコード挿入による難読化の対象となる文は、難読化対象のメソッドにおける制御遷移を含まない文の全て (代入文やメソッド呼出しなど)
- 難読化で利用する恒真条件式は $7x^2 - 1 \neq y^2$ で固定

表 5.1 に評価実験を行った環境を示す。

メソッドの実行時間の測定は System クラスの nanoTime メソッドを利用した。nanoTime メソッドは、利用できる最も正確なシステムタイマーの現在の値をナノ秒単位で返すメソッドである [25]。プログラム 5.1 に、*testMethod* メソッドの実行時間を nanoTime メソッドを利用して測定し表示するプログラムコードを示す。2 行目で *testMethod* メソッドを呼出す前の時間を取得し、4 行目で *testMethod* メソッドを呼出した後の時間を取得している。上で取得した時間の差から、7 行目でメソッドの実行に要した実行時間をミリ秒単位で標準出力に出力している。

表 5.1　実行環境

| 環境名 | 規格 | 製造会社 |
|---|---|---|
| OS | Mac OS X 10.7.5 | Apple |
| CPU | Intel Core i7 2.4 GHz | Intel |
| メモリ | 8 GB 1333 MHz DDR3 | |
| 端末エミュレータ | ターミナル 2.2.3 (303.2) | Apple |
| Java 仮想マシン | Java(TM) SE Runtime Environment (build 1.7.0_04-b21) | Oracle |
| | Java HotSpot(TM) 64-Bit Server VM (build 23.0-b21, mixed mode) | |

プログラム 5.1　System.nanoTime メソッドを利用した実行時間測定コード

```
1     //testMethodの実行時間測定
2     long start = System.nanoTime();
3     testMethod();
4     long end = System.nanoTime();
5
6     //testMethodの実行時間をミリ秒で表示
7     System.out.println( (end - start ) / 1000.0 / 1000.0 + "[ms]");
```

## 5.1　デッドコード挿入による難読化への適用に対する評価

3.2 節 (p.15) で述べたデッドコード挿入による難読化を行い、メソッドの実行時間を測定した。

### 5.1.1　フィールド変数の加算を行うプログラムの難読化

難読化の対象となるメソッドのプログラムコードをプログラム 5.2 に示す。

プログラム 5.2　フィールド変数の加算を行うプログラム

```
1 static void test0(int n) {
2     for(int i=0;i<n;i++)
3         for(int j=0;j<10;j++)
4             for(int k=0;k<10;k++)
5                 val++;
6 }
```

このメソッドは、フィールド変数 val の値を n×100 回加算する処理を行う。

プログラム 5.2 の 5 行目の文を対象に、既存手法である条件分岐を用いたデッドコード挿入による難読化を行ったときのプログラムコードをプログラム 5.3 に示す。

プログラム 5.3　プログラム 5.2 を既存手法により難読化したコード

```
1 static void test1(int n) {
2     for(int i=0;i<n;i++)
3         for(int j=0;j<10;j++)
4             for(int k=0;k<10;k++){
5                 if(Pt(j,k))
6                     val++;
7                 else
8                     val--;
9             }
10 }
```

ここで、メソッド Pt は以下のように定義されている。

```
public static boolean Pt(int x, int y){
    return 7*x*x-1!=y*y;
}
```

プログラム 5.3 の 5 行目の条件式 Pt(j, k) は必ず真であるため、6 行目の変数 val のインクリメントが必ず実行される。しかし、条件式 Pt(j, k) は必ず真であることを解析者が分からないならば、6 行目の変数 val のインクリメントと、7 行目の変数 val のデクリメントのどちらが実行されるか分からない。

プログラム 5.2 の 5 行目の文を対象に、提案手法である仮想メソッド呼出しを用いたデッドコード挿入による難読化を行ったときのプログラムコードをプログラム 5.4 に示す。ここでは、オブジェクト配列として 1 次元 2 要素の配列を利用し、ローカル変数は文に副作用がないためメソッドの引数として渡している。

プログラム 5.4　プログラム 5.2 を提案手法により難読化したコード

```
1  static abstract class O{
2      abstract void m();
3
4      static O o1=new A(),o2=new B();
5  }
6  static class A extends O{
7      void m(){
8          Test.val++;
9      }
10 }
11 static class B extends O{
12     void m(){
13         Test.val--;
14     }
15 }
16 static final O[] oAry1 = {O.o1, O.o2};
17
18 static void test2(int n) {
19     for(int i=0;i<n;i++)
20         for(int j=0;j<10;j++)
21             for(int k=0;k<10;k++){
22                 oAry1[Pt(j,k)?0:1].m();
23             }
24 }
```

プログラム 5.4 の 22 行目の条件式 Pt(j, k) は必ず真であるため、式 Pt(j ,k)?0:1 は必ず 0 を返し、oAry1[0] の要素であるクラス A のインスタンス O.o1 に対して m の仮想メソッド呼出しを行うため、8 行目の変数 val のインクリメントが必ず実行される。しかし、条件式 Pt(j, k) は必ず真であることを解析者が分からないならば、式 Pt(j ,k)?0:1 が必ず 0 を返すことが分からないため、22 行目の m の仮想メソッド呼出しを行うオブジェクト

の型はクラス A かクラス B か分からず、8 行目の変数 val のインクリメントと、13 行目の変数 val のデクリメントのどちらが実行されるか分からない。

プログラム 5.3 の条件式を予め計算し配列にキャッシュ化したときのプログラムコードをプログラム 5.5 に示す。

プログラム 5.5　プログラム 5.2 をキャッシュを利用した既存手法により難読化したコード

```
1  static final boolean[][] bAry;
2  static {
3      bAry = new boolean[10][10];
4      for(int x=0;x<10;x++){
5          for(int y=0;y<10;y++){
6              bAry[x][y]=Pt(x,y);
7          }
8      }
9  }
10
11 static void test3(int n) {
12     for(int i=0;i<n;i++)
13         for(int j=0;j<10;j++)
14             for(int k=0;k<10;k++){
15                 if(bAry[j][k])
16                     val++;
17                 else
18                     val--;
19             }
20 }
```

プログラム 5.5 では、スタティックイニシャライザを利用して条件式を予め計算している。また、ローカル変数 x,y の範囲は 0 から 9 であり、配列の要素数も 9 までにしたため、剰余計算は行っていない。

プログラム 5.4 の条件式を予め計算し配列にキャッシュ化したときのプログラムコードをプログラム 5.6 に示す。

プログラム 5.6　プログラム 5.2 をキャッシュを利用した提案手法により難読化したコード

```
1  static void test4(int n) {
2      for(int i=0;i<n;i++)
3          for(int j=0;j<10;j++)
4              for(int k=0;k<10;k++){
5                  oAry2[j][k].m();
6              }
7  }
8  static final O[][] oAry2;
9  static {
10     oAry2 = new O[10][10];
11     for(int x=0;x<10;x++){
12         for(int y=0;y<10;y++){
```

```
13                oAry2[x][y]=Pt(x,y)?O.o1:O.o2;
14            }
15        }
16 }
17
18 static abstract class O{
19        abstract void m();
20
21        static O o1=new A(),o2=new B();
22 }
23 static class A extends O{
24        void m(){
25            Test.val++;
26        }
27 }
28 static class B extends O{
29        void m(){
30            Test.val--;
31        }
32 }
```

n の値を 1 万から 1,000 万まで変えたときのプログラム 5.2、プログラム 5.3、プログラム 5.4、プログラム 5.5、プログラム 5.6 のメソッドの実行時間を表 5.2 に示す。また、横軸を n の値、縦軸をメソッドの実行時間としたときのグラフを図 5.1、図 5.2、図 5.3 に示す。

図 5.1 のグラフより、n の値が小さくプログラム全体の実行時間が短い場合は、予め条件式を計算した既存手法が一番速く、予め条件式を計算し仮想メソッド呼出しを行う提案手法が逆に一番遅いことが分かる。

しかし図 5.3 のグラフから、n の値を増やしプログラム全体の実行時間を増やすことで逆転し、最終的にはオブジェクトをキャッシュ化した提案手法が一番速くなることが分かった。予め条件式の計算を行わない場合では、既存手法の条件分岐よりも提案手法の仮想メソッド呼出しの方が速くなり、予め条件式を計算した既存手法が一番遅くなった。この逆転が起きた理由として、プログラム全体の実行時間が短い場合は JIT コンパイラにより最適化される前までの遅い処理、逆にプログラム全体の実行時間が長い場合は JIT コンパイラによって最適化された後の処理が、実行時間の大きな要因となるからと考えられる。

これら結果から、opaque な要因として条件分岐の代わりに仮想メソッド呼出しを利用した場合、実際に実行時間に関するオーバヘッドを削減できることを確認することができた。しかし、具体的にどのような最適化が行われたことで条件分岐より仮想メソッド呼出しの方が実行時間が短くなったのか、詳しい理由は分かっていない。参考として、プログラム 5.3 を実行したときの Java JIT コンパイラが生成した機械語コードを付録

A.1(p.65) に、プログラム 5.6 を実行したときの Java JIT コンパイラが生成した機械語コードを付録 A.2(p.66) に示す。

表 5.2　test メソッドの平均実行時間 [ms]

| n | 10000 | 20000 | 30000 | 40000 | 50000 |
|---|---|---|---|---|---|
| 難読化前 | 2.2438 | 2.6251 | 3.0281 | 3.4739 | 3.8965 |
| 既存手法：if(PT(x,y)) | 4.2131 | 6.1309 | 7.9808 | 9.8864 | 11.9611 |
| 提案手法：o[PT(x,y)].m() | 5.2007 | 6.9344 | 8.7938 | 10.5937 | 12.6127 |
| 既存手法：if(b[x][y]) | 3.9231 | 5.9393 | 7.8944 | 10.2521 | 12.5615 |
| 提案手法：o[x][y].m() | 6.1781 | 7.0619 | 7.9223 | 9.1207 | 9.8834 |
| n | 60000 | 70000 | 80000 | 90000 | 100000 |
| 難読化前 | 4.227 | 4.7416 | 5.0213 | 5.3976 | 5.7557 |
| 既存手法：if(PT(x,y)) | 13.6409 | 15.477 | 17.407 | 19.1887 | 21.3545 |
| 提案手法：o[PT(x,y)].m() | 14.2681 | 16.0803 | 17.8928 | 19.6245 | 21.5083 |
| 既存手法：if(b[x][y]) | 14.0372 | 15.9242 | 17.9319 | 20.2095 | 21.9875 |
| 提案手法：o[x][y].m() | 10.8181 | 11.5903 | 12.5827 | 13.5054 | 14.3781 |
| n | 100000 | 200000 | 300000 | 400000 | 500000 |
| オリジナルプログラム | 5.7557 | 9.7036 | 13.4937 | 17.4231 | 21.6555 |
| 既存手法：if(PT(x,y)) | 21.3545 | 39.9028 | 58.3628 | 77.2104 | 96.3126 |
| 提案手法：o[PT(x,y)].m() | 21.5083 | 39.6975 | 57.6541 | 75.8960 | 94.2066 |
| 既存手法：if(b[x][y]) | 21.9875 | 41.9914 | 62.0998 | 81.8391 | 103.0158 |
| 提案手法：o[x][y].m() | 14.3781 | 23.5788 | 32.6560 | 41.6991 | 51.6381 |
| n | 600000 | 700000 | 800000 | 900000 | 1000000 |
| オリジナルプログラム | 25.6823 | 29.6149 | 33.8286 | 36.8731 | 40.6785 |
| 既存手法：if(PT(x,y)) | 117.2559 | 136.2614 | 154.3089 | 169.5008 | 188.5589 |
| 提案手法：o[PT(x,y)].m() | 113.8223 | 132.9583 | 151.3988 | 165.6149 | 183.5827 |
| 既存手法：if(b[x][y]) | 123.7062 | 144.0811 | 161.9901 | 179.8423 | 199.6059 |
| 提案手法：o[x][y].m() | 61.5756 | 70.8397 | 80.2361 | 86.7280 | 93.3250 |
| n | 1000000 | 2000000 | 3000000 | 4000000 | 5000000 |
| オリジナルプログラム | 40.6785 | 78.9799 | 117.4772 | 157.0820 | 196.5372 |
| 既存手法：if(PT(x,y)) | 188.5589 | 375.4199 | 558.6095 | 748.2129 | 931.2643 |
| 提案手法：o[PT(x,y)].m() | 183.5827 | 363.7859 | 542.3055 | 725.4057 | 902.5978 |
| 既存手法：if(b[x][y]) | 199.6059 | 397.7397 | 592.6802 | 792.5668 | 982.1994 |
| 提案手法：o[x][y].m() | 96.3250 | 186.3497 | 276.3129 | 370.0315 | 458.4873 |
| n | 6000000 | 7000000 | 8000000 | 9000000 | 10000000 |
| オリジナルプログラム | 235.6053 | 273.1366 | 311.9753 | 351.5051 | 387.6964 |
| 既存手法：if(PT(x,y)) | 1126.3967 | 1298.3641 | 1486.6254 | 1671.4277 | 1852.1168 |
| 提案手法：o[PT(x,y)].m() | 1094.6980 | 1258.2994 | 1437.8846 | 1617.6505 | 1800.4962 |
| 既存手法：if(b[x][y]) | 1189.8242 | 1376.5270 | 1575.4295 | 1763.2672 | 1961.9001 |
| 提案手法：o[x][y].m() | 554.0526 | 639.5750 | 729.8027 | 820.4169 | 909.3283 |

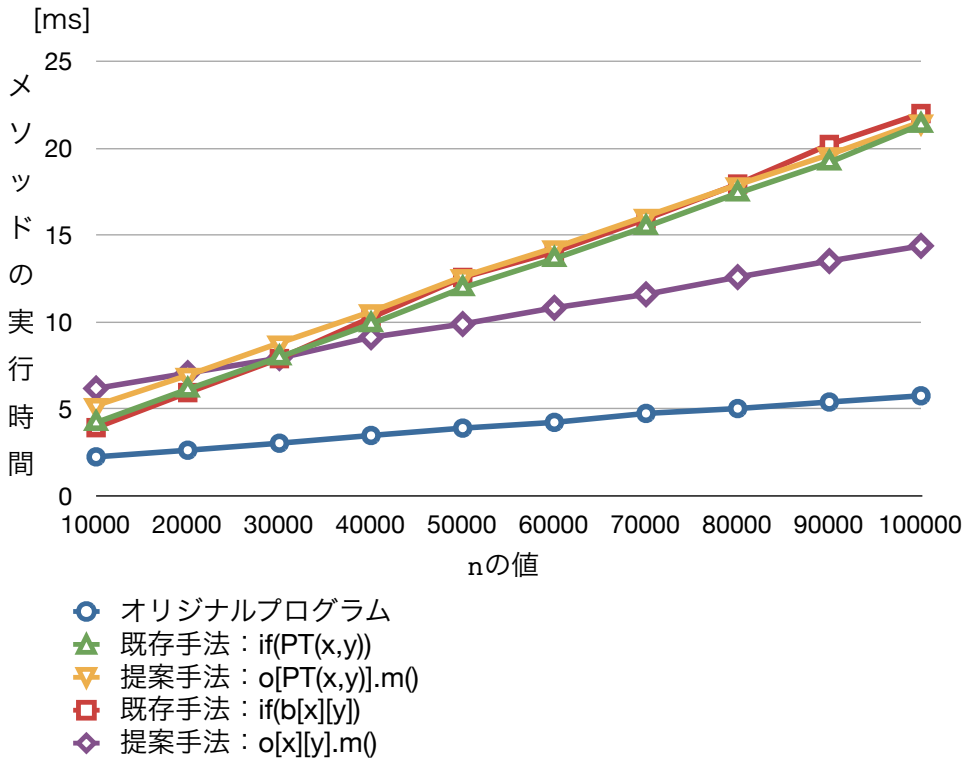



図 5.1　表 5.2 の実行時間グラフ (n:1 万〜10 万))

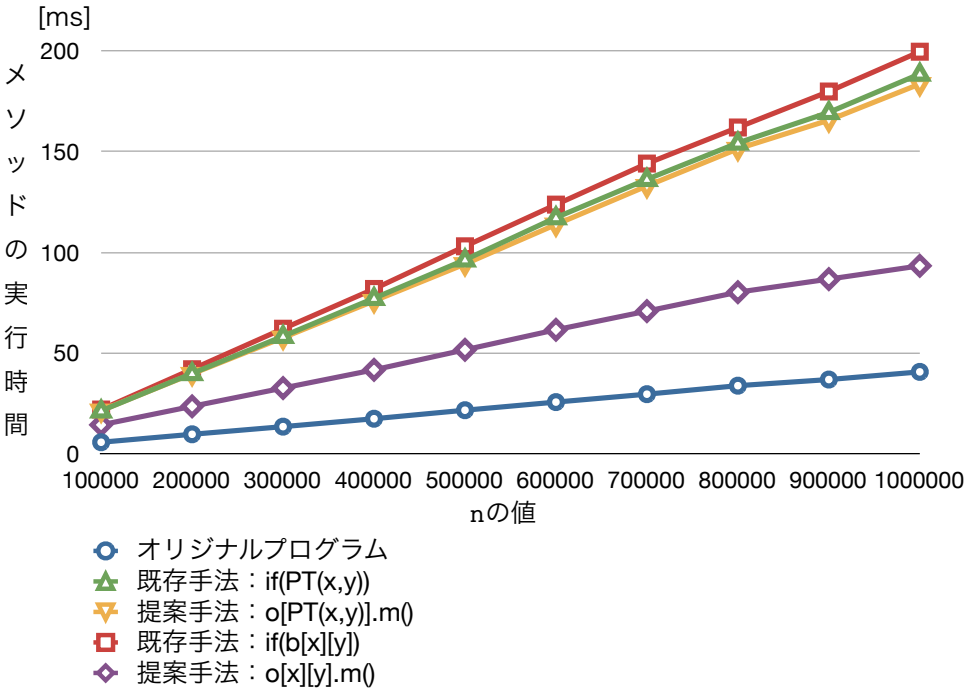



図 5.2　表 5.2 の実行時間グラフ (n:10 万〜100 万)

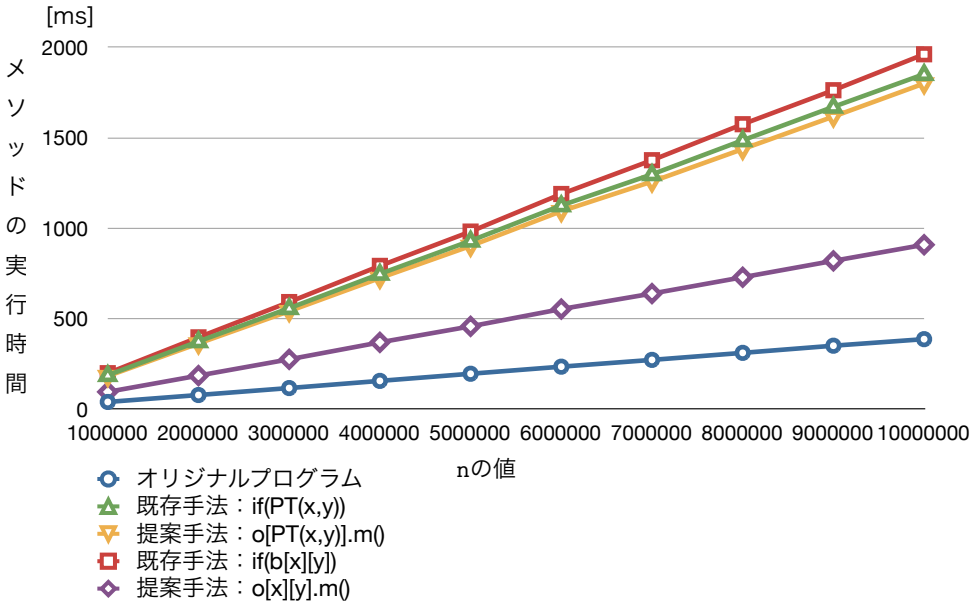



図 5.3　表 5.2 の実行時間グラフ (n:100 万〜1,000 万)

### 5.1.2　配列に格納された値を入れ替えるプログラムの難読化

より複雑なプログラムコードにおいて、提案手法に基づいた難読化のオーバヘッドが削減できるかどうかの検証を行った。難読化の対象となるメソッドのプログラムコードをプログラム 5.7 に示す。これは、配列に格納された値を入れ替えるプログラムである。

プログラム 5.7　配列に格納された値を入れ替えるプログラム

```
1  private static void swap(int i, int j) {
2      int tmp = array[i];
3      array[i] = array[j];
4      array[j] = tmp;
5  }
6
7  @Obfuscate
8  public void test() {
9      for (int index = 1; index < n - 1; index++) {
10         swap(index, n - 1 - index);
11         swap(index + 1, n - 1 - index);
12         swap(index - 1, index + 1);
13         swap(index, n - 1 - index);
14         swap(index, index + 1);
15         swap(index, index - 1);
16     }
17     return;
18 }
```

このプログラムコードに対し、5.1.1 節 (p.42) と同様の実験を行った。難読化は第 4 章で述べた変換器を利用して行った。なお、キャッシュ配列のサイズは $10 \times 10$ 固定とし、配列のインデックスは任意の整数変数に対し 10 の剰余計算した値とした。

配列の要素数を 100 万から 1,000 万まで変えたときの swap メソッドの実行時間を表 5.3 に示す。また、横軸の配列の要素数、縦軸をメソッドの実行時間としたときのグラフを図 5.4 に示す。

プログラム 5.7 に対する難読化では、予め条件式を計算せずに仮想メソッド呼出しを行う提案手法が最終的には一番速くなり、次に条件式を予め計算して仮想メソッド呼出しを行う提案手法が速く、条件分岐を利用した既存手法はどちらも遅くなった。

プログラムの実行時間以外に関するオーバヘッドについて述べる。表 5.4 に、本実験で用いたプログラムをコンパイルすることで生成されたクラスファイルの数とファイルサイズの合計を示す。既存手法により難読化されたプログラムでは、難読化前と比べクラスファイルのサイズはほぼ同じである。しかし、提案手法により難読化されたプログラムでは、難読化前と比べ多くのクラスファイルが作られ、ファイルサイズも増加している。これは、難読化にあたり内部クラスを多く追加しているためである。

表 5.3　配列要素交換処理を行う平均実行時間 [ms](配列要素数:100 万〜1,000 万)

| 配列要素数 | 1000000 | 2000000 | 3000000 | 4000000 | 5000000 |
|---|---|---|---|---|---|
| オリジナルプログラム | 17.0442 | 21.3803 | 24.9423 | 29.3391 | 33.3949 |
| 既存手法：if(PT(x,y)) | 24.4797 | 36.4136 | 47.8872 | 58.8952 | 70.8451 |
| 提案手法：o[PT(x,y)].m() | 27.6225 | 35.5885 | 43.9519 | 52.3603 | 60.3266 |
| 既存手法：if(b[x][y]) | 38.4335 | 61.0617 | 82.7814 | 104.7766 | 127.479 |
| 提案手法：o[x][y].m() | 40.2356 | 49.3714 | 58.1564 | 66.7373 | 75.8386 |

| 配列要素数 | 6000000 | 7000000 | 8000000 | 9000000 | 10000000 |
|---|---|---|---|---|---|
| オリジナルプログラム | 36.8067 | 40.6246 | 44.7076 | 48.7441 | 53.0305 |
| 既存手法：if(PT(x,y)) | 81.5581 | 91.0017 | 102.4655 | 113.54 | 125.1057 |
| 提案手法：o[PT(x,y)].m() | 66.9448 | 75.0014 | 82.5756 | 91.1252 | 98.4736 |
| 既存手法：if(b[x][y]) | 146.4372 | 168.3785 | 189.5099 | 211.6159 | 232.8911 |
| 提案手法：o[x][y].m() | 83.3206 | 91.8011 | 99.6005 | 108.1379 | 117.1024 |

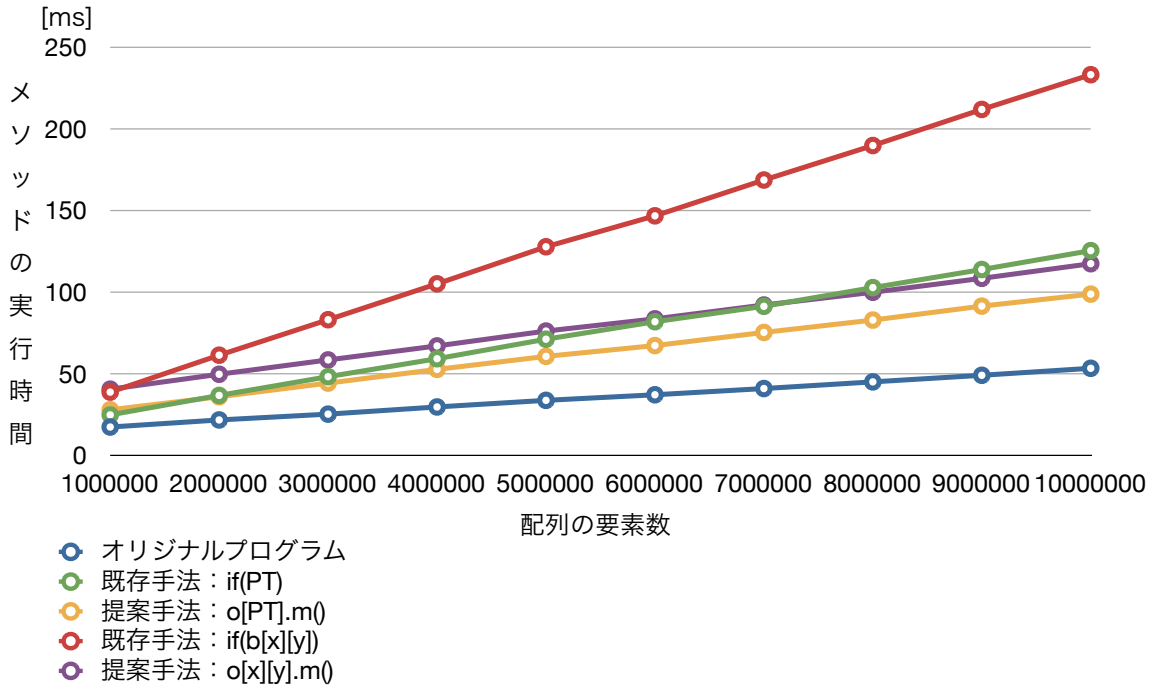



図 5.4　表 5.3 の実行時間グラフ

表 5.4　配列要素交換処理を行うプログラムにおけるクラスファイルの数とサイズ

| | クラスファイル数 | ファイルサイズ合計 [kB] |
|---|---|---|
| オリジナルプログラム | 1 | 1.65 |
| 既存手法：if(PT(x,y)) | 1 | 1.93 |
| 提案手法：o[PT(x,y)].m() | 15 | 11.21 |
| 既存手法：if(b[x][y]) | 1 | 2.65 |
| 提案手法：o[x][y].m() | 15 | 11.43 |

### 5.1.3 配列に格納された値をソートするプログラムの難読化

難読化の対象となるメソッドのプログラムコードをプログラム 5.8 に示す。これは、配列に格納された値をバブルソートを利用して昇順に並び替えるプログラムである。

プログラム 5.8 配列に格納された値をソートするプログラム

```
1  private static void swap(int i, int j) {
2      int tmp = array[i];
3      array[i] = array[j];
4      array[j] = tmp;
5  }
6
7  @Obfuscate
8  public void sort() {
9      for (int i = 0; i < size; i++) {
10         boolean isSwapped = false;
11         for (int j = size - 1; j > i; j--) {
12             if (array[j] < array[j - 1]) {
13                 swap(j, j - 1);
14                 isSwapped = true;
15             }
16         }
17         if (!isSwapped) {
18             break;
19         }
20     }
21 }
```

このプログラムコードに対し、5.1.2 節 (p.48) と同様の実験を行った。

配列の要素数を 1,000 から 10 万まで変えたときの sort メソッドの実行時間を、表 5.5 に示す。また、横軸の配列の要素数、縦軸をメソッドの実行時間としたときの片対数グラフを図 5.5 と図 5.6 に示す。

プログラム 5.8 に対する難読化では、予め条件式を計算しない提案手法が最終的には一番速くなり、予め条件式を計算した提案手法は予め条件式を計算しない既存手法よりも遅くなった。この結果は、5.1.1 節 (p.42) とは異なる結果になったと言える。

表 5.5　配列ソート処理を行う平均実行時間 [ms]

| 配列要素数 | 1000 | 2000 | 3000 | 4000 | 5000 |
|---|---|---|---|---|---|
| オリジナルプログラム | 5.5134 | 7.1794 | 9.877 | 13.9155 | 19.0787 |
| 既存手法：if(PT(x,y)) | 6.1966 | 10.7644 | 17.7547 | 28.2276 | 41.1239 |
| 提案手法：o[PT(x,y)].m() | 14.3722 | 17.5893 | 22.8141 | 29.5233 | 38.7724 |
| 既存手法：if(b[x][y]) | 10.3309 | 19.6058 | 34.6512 | 55.6807 | 81.996 |
| 提案手法：o[x][y].m() | 17.6388 | 23.4125 | 32.7063 | 45.934 | 63.9016 |
| 配列要素数 | 6000 | 7000 | 8000 | 9000 | 10000 |
| オリジナルプログラム | 25.3482 | 33.0937 | 41.4315 | 54.1329 | 62.4431 |
| 既存手法：if(PT(x,y)) | 57.3279 | 76.3311 | 98.2685 | 123.3483 | 151.1416 |
| 提案手法：o[PT(x,y)].m() | 50.1888 | 63.0559 | 78.2228 | 96.5532 | 115.7257 |
| 既存手法：if(b[x][y]) | 115.318 | 157.8254 | 200.6802 | 252.3476 | 312.2414 |
| 提案手法：o[x][y].m() | 83.8901 | 108.4478 | 146.1715 | 181.1166 | 219.4936 |
| 配列要素数 | 10000 | 20000 | 30000 | 40000 | 50000 |
| オリジナルプログラム | 62.4431 | 237.9383 | 525.0771 | 929.3687 | 1448.1882 |
| 既存手法：if(PT(x,y)) | 151.1416 | 594.0898 | 1316.7777 | 2334.4219 | 3643.3615 |
| 提案手法：o[PT(x,y)].m() | 115.7257 | 423.1196 | 928.5516 | 1633.4433 | 2544.3405 |
| 既存手法：if(b[x][y]) | 312.2414 | 1226.1216 | 2670.691 | 4771.9183 | 7629.5496 |
| 提案手法：o[x][y].m() | 219.4936 | 837.1205 | 1838.0331 | 3249.5194 | 5105.524 |
| 配列要素数 | 60000 | 70000 | 80000 | 90000 | 100000 |
| オリジナルプログラム | 2092.3147 | 2830.1559 | 3691.1285 | 4700.3877 | 5832.4268 |
| 既存手法：if(PT(x,y)) | 5278.0421 | 7157.409 | 9363.559 | 11858.0975 | 14659.9518 |
| 提案手法：o[PT(x,y)].m() | 3685.503 | 5037.4957 | 6576.4622 | 8461.5104 | 10462.7402 |
| 既存手法：if(b[x][y]) | 11015.6226 | 14967.2474 | 19432.0098 | 24643.3495 | 30482.7646 |
| 提案手法：o[x][y].m() | 7411.0179 | 10146.392 | 13293.2082 | 16986.224 | 21161.679 |

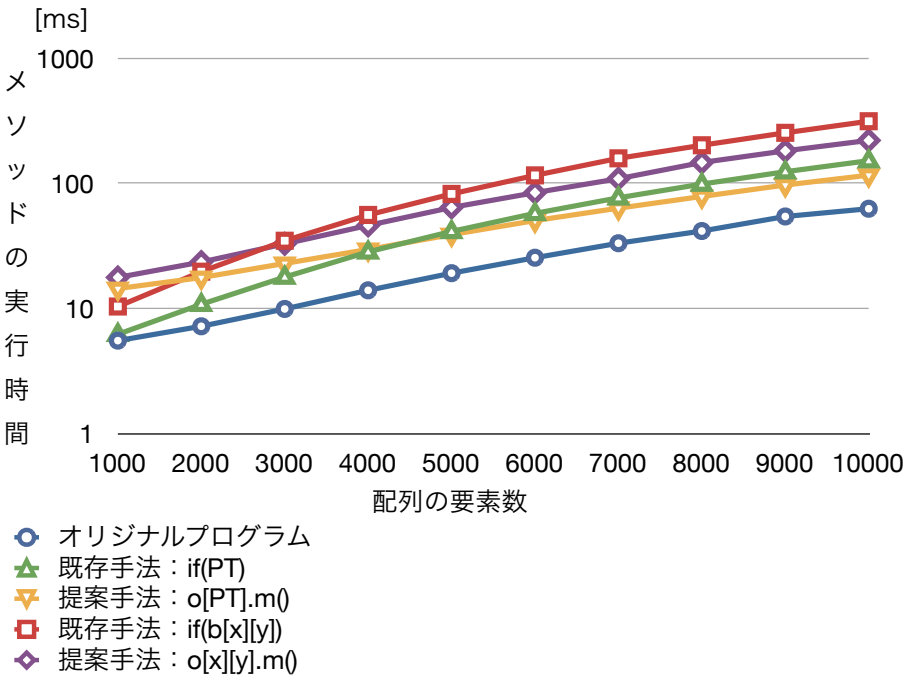



図 5.5　表 5.5 の実行時間グラフ (配列要素数:1,000〜1 万)

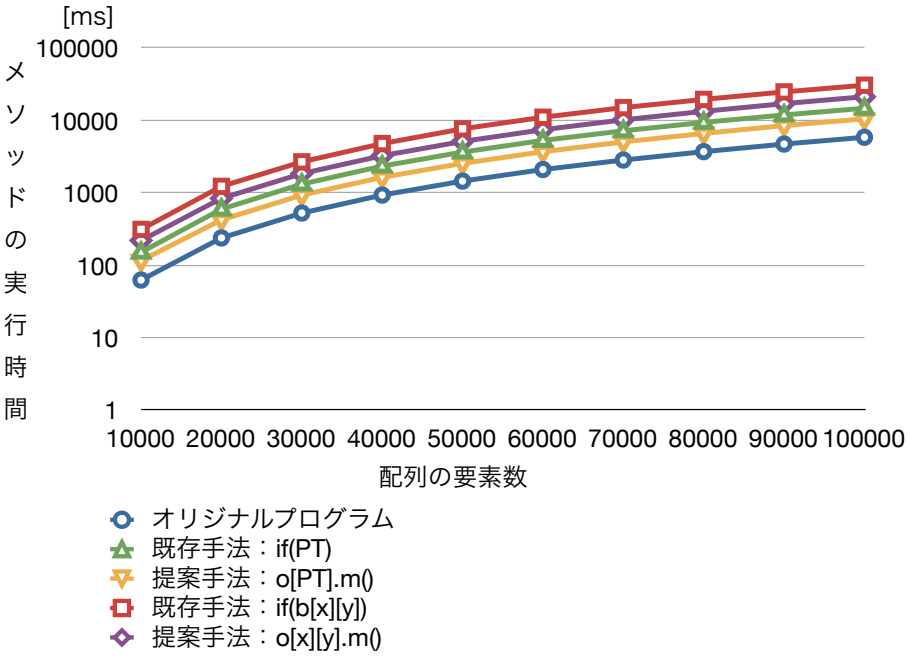



図 5.6　表 5.5 の実行時間グラフ (配列要素数:1 万〜10 万)

## 5.2 制御構造パターンの破壊による難読化への適用に対する評価

プログラム 5.9 に、制御構造パターンの破壊による難読化を行うプログラムコードを示す。

プログラム 5.9 制御構造パターン破壊による難読化用のテストプログラムのコード

```
1  static long loop(int max) {
2      for (int i = 0; i < max; i++) {
3          cnt = (int) Math.sqrt(cnt);
4          while (cnt < 5) {
5              val1 += max;
6              val2 *= cnt;
7              cnt++;
8          }
9      }
10     return val1 + val2;
11 }
```

制御構造パターンの破壊による難読化として、3 行目の代入文の前に 5 行目と 6 行目への実行されないフローを追加する。既存手法となる条件分岐を利用して制御構造パターンの破壊による難読化を行ったプログラムコード表現をプログラム 5.10 に示す。なおこのプログラムにおいて、Java 言語では goto 文を扱えないため、実際はコンパイルされ得られたクラスファイルの書換えを行った。

プログラム 5.10 プログラム 5.9 に対し既存手法の制御構造パターン破壊による難読化を行ったコード

```
1  static long loop1(int max) {
2      for (int i = 0; i < max; i++) {
3          if (Pf(max, cnt)) {
4              goto label; //クラスファイル書換えにより挿入
5          }
6          cnt = (int) Math.sqrt(cnt);
7          while (cnt < 5) {
8              val1 += max;
9  label:
10             val2 *= cnt;
11             cnt++;
12         }
13     }
14     return val1 + val2;
15 }
```

プログラム 5.10 に対し、提案手法となる仮想メソッド呼出しを利用して制御構造パターンの破壊による難読化を行ったプログラムコードをプログラム 5.11 とプログラム 5.12 に示す。プログラム 5.11 が相互再帰を使わずに難読化を行ったときのコード、プログラム 5.12 が相互再帰を用いて難読化を行ったときのコードである。

プログラム 5.11　プログラム 5.9 に対し相互再帰を使わない提案手法の制御構造パターン破壊による難読化を行ったコード

```
1  static long loop2(int max) {
2      for (int i = 0; i < max; i++) {
3          o2[Pf(max, cnt) ? 0 : 1].m(max);
4      }
5      return val1 + val2;
6  }
7
8  static final O2[] o2 = { O2.ot, O2.of };
9  static abstract class O2 {
10     abstract void m(int max);
11     static O2 ot = new O2T(), of = new O2F(),
12         oloop = new O2Loop(), os = new O2S();
13 }
14
15 static class O2T extends O2 {
16     @Override
17     void m(int max) {
18         os.m(max);
19         oloop.m(max);
20     }
21 }
22 static class O2F extends O2 {
23     @Override
24     void m(int max) {
25         cnt = (int) Math.sqrt(cnt);
26         oloop.m(max);
27     }
28 }
29
30 static class O2Loop extends O2 {
31     @Override
32     void m(int max) {
33         while (cnt < 5) {
34             val1 += max;
35             os.m(max);
36         }
37     }
38 }
39
40 static class O2S extends O2 {
41     @Override
```

```
42      void m(int max) {
43          val2 *= cnt;
44          cnt++;
45      }
46  }
```

プログラム 5.12　プログラム 5.9 に対し相互再帰を使った提案手法の制御構造パターン破壊による難読化を行ったコード

```
 1  static long loop3(int max) {
 2      for (int i = 0; i < max; i++) {
 3          o3[Pf(max, cnt) ? 0 : 1].m(max);
 4      }
 5      return val1 + val2;
 6  }
 7
 8  static final O3[] o3 = { O3.ot, O3.of };
 9  static abstract class O3 {
10      abstract void m(int max);
11      static O3 ot = new O3T(), of = new O3F(), oloop = new O3Loop();
12  }
13
14  static class O3T extends O3 {
15      @Override
16      void m(int max) {
17          val2 *= cnt;
18          cnt++;
19          oloop.m(max);
20      }
21  }
22  static class O3F extends O3 {
23      @Override
24      void m(int max) {
25          cnt = (int) Math.sqrt(cnt);
26          oloop.m(max);
27      }
28  }
29
30  static class O3Loop extends O3 {
31      @Override
32      void m(int max) {
33          if (cnt < 5) {
34              val1 += max;
35              ot.m(max);
36          }
37      }
38  }
```

プログラム 5.12 のプログラムは相互再帰呼出しを行うため、max の値によってはスタックオーバフローが発生する。そこで、Java 仮想マシンが利用するスタックサイズを 1

ギガバイトまで増やし、評価を行った。

max の値を 100 万から 1,000 万まで変えたときのメソッドの実行時間を表 5.6 に示す。また、横軸を max の値、縦軸のメソッドの実行時間としたときのグラフを図 5.7 に示す。

表 5.6 と図 5.7 から、相互再帰を使わずに仮想メソッド呼出しを用いて難読化を行った場合は、条件分岐を用いて難読化を行うより実行時間が速くなることがあることを確認できた。しかし、相互再帰となる仮想メソッド呼出しを用いて難読化を行った場合は、既存手法より速くなることは無かった。これは、再帰である仮想メソッド呼出しのインライン展開がうまく行われなかったからであると考えられる。

表 5.6　loop メソッドの平均実行時間 [ms](max:100 万〜1,000 万)

| | 1000000 | 2000000 | 3000000 | 4000000 | 5000000 |
|---|---|---|---|---|---|
| オリジナルプログラム | 14.4753 | 25.8893 | 37.4707 | 48.5101 | 60.2148 |
| 既存手法 | 15.481 | 28.2392 | 40.6167 | 53.5074 | 65.9713 |
| 提案手法 (相互再帰無し) | 18.7033 | 30.7197 | 42.8865 | 54.9463 | 66.9975 |
| 提案手法 (相互再帰有り) | 20.242 | 33.6619 | 46.9271 | 60.4515 | 73.7488 |

| | 6000000 | 7000000 | 8000000 | 9000000 | 10000000 |
|---|---|---|---|---|---|
| オリジナルプログラム | 71.6097 | 83.0217 | 94.4556 | 105.5763 | 117.1517 |
| 既存手法 | 78.1424 | 90.8345 | 103.5639 | 115.3845 | 128.5306 |
| 提案手法 (相互再帰無し) | 78.9219 | 91.2076 | 103.06 | 114.797 | 127.0462 |
| 提案手法 (相互再帰有り) | 86.8281 | 100.673 | 114.5603 | 127.7455 | 140.4476 |

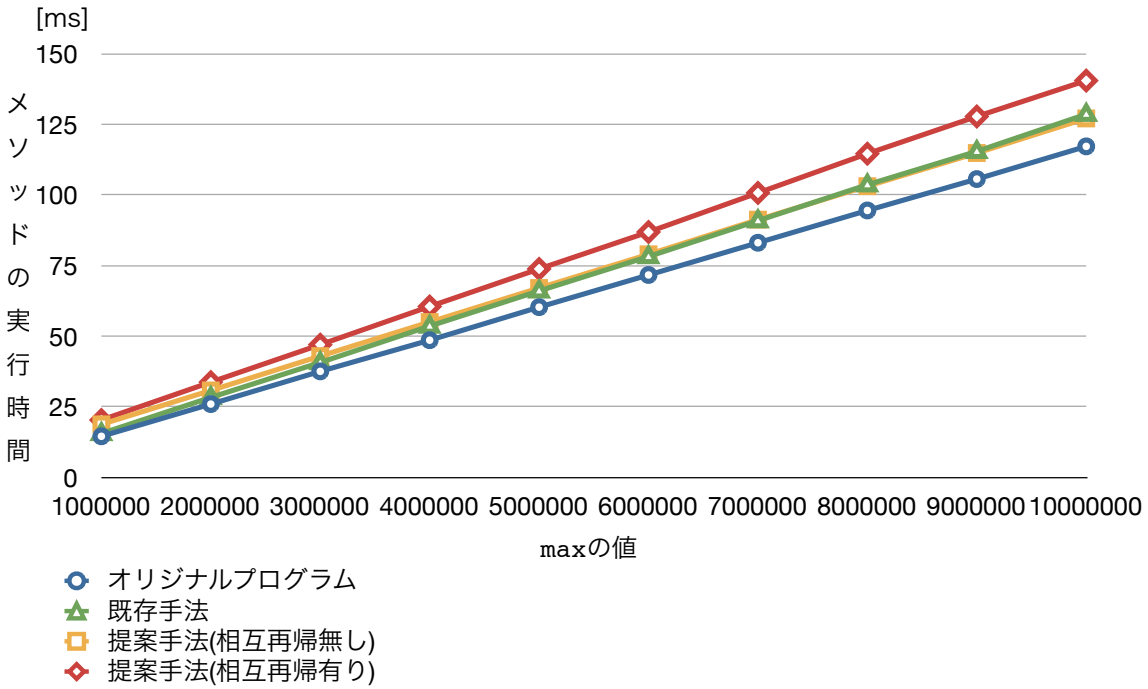



図 5.7　表 5.6 の実行時間グラフ (max:100 万〜1,000 万)

## 5.3 評価のまとめ

表 5.7 に、これまでの結果から既存手法と提案手法に基づいた難読化によって加わる実行時間のオーバヘッドのまとめを示す。5.1.1 節 (p.42) の変数加算プログラムに対する難読化の評価では、条件式を予め計算しない既存手法と、条件式を予め計算した提案手法で比較を行った。それ以外の評価では、条件式を予め計算しない既存手法と提案手法で比較を行った。

表 5.7 から、デッドコード挿入による難読化では条件分岐を用いた既存手法と比べ、仮想メソッド呼出しを用いた提案手法ではオーバヘッドを 236〜478% から 179%〜235% に減らすことができ、1.19 倍〜2.04 倍のプログラム高速化に成功したと言える。

提案手法の問題点として、プログラムのファイルサイズが大幅に増えてしまう点がある。これは、難読化によって実行用の内部クラスとダミー用の内部クラス、それらの親となる抽象クラスと変数参照用のクラスを作成するためである。これらクラスは実行時に読込まれるため、これらクラスファイルロードがプログラム全体に加わるオーバヘッドとなる可能性が考えられる。

表 5.7　難読化による実行時間オーバヘッドのまとめ

| 評価 | 実行時間オーバヘッド | | 性能向上比 |
|---|---|---|---|
| | 既存手法 | 提案手法 | |
| 5.1.1 節 (p.42)(n の値:1,000 万) | 478% | 235% | 2.04 倍 |
| 5.1.2 節 (p.48)(配列要素数:1,000 万) | 236% | 186% | 1.19 倍 |
| 5.1.3 節 (p.51)(配列要素数:10 万) | 251% | 179% | 1.40 倍 |
| 5.2 節 (p.54)(max の値:1,000 万) | 110% | 108(120)% | 1.01(0.92) 倍 |

# 第 6 章

# 結論と今後の課題

## 6.1 結論

本研究では、既存の条件分岐を利用した制御フローの難読化によってプログラムに加わるオーバヘッドを削減するため、JIT コンパイラによって動的に最適化される仮想メソッド呼出しを利用する手法を提案した。また、条件分岐を利用した制御フローの難読化として、デッドコード挿入による難読化と制御構造の破壊による難読化を取り上げ、これらに対して提案手法を適用した難読化手法の変換規則を示した。複数のプログラムに対し、条件分岐を用いる既存手法と仮想メソッド呼出しを用いる提案手法の実行時間を計測した結果、最大で実行時間に加わるオーバヘッドを 478% から 235% に削減することができた。また、プログラムの実行時間が長いほど提案手法によるオーバヘッド削減の効果が大きくなることが分かった。

## 6.2 今後の課題

現在、条件分岐の代わりとして仮想メソッド呼出しを利用した場合に、実行時間に関するオーバヘッドを削減することができることは分かったが、どのような動的最適化が行われることで実行時間が速くなったかの原因が詳しく分かっていないため、今後の課題とする。

また、デッドコード挿入による難読化を行う変換器の実装は行ったが、制御構造の破壊による難読化を行う変換器の実装はまだ行っていないため、今後に実装を行う予定である。

# 謝辞

本研究を遂行するにあたり、様々な方々にお世話になりました。

指導教員である小宮常康先生には、毎週行われたゼミなど日頃からの熱心なご指導、ご助言、そして鞭撻を賜わりました。ここに厚く御礼申し上げます。

また、年数回行われた講座内進捗においてでは、多田好克先生、鶴岡行雄先生、末田欣子先生、佐藤喬には研究を進める上での貴重な意見や助言を頂きました。ここに感謝の意を表します。

そして本研究を終えることができたことは、共に研究生活を送ってきた基盤ソフトウェア学講座の学生諸氏のおかげであります。更に、群馬工業高等専門学校への入学から電気通信大学大学院の卒業までの約 10 年、私が情報工学について学ぶことができたことは、両親や友人、先生や先輩や後輩やアルバイト先の人々など、これまでの人生で出会った方々のおかげであります。皆様への心から感謝の気持ちと御礼を申し上げたく、謝辞にかえさせていただきます。

2014 年 1 月吉日 石田峰文

# 参考文献


[1] 門田暁人 and Clark 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 —” *IPSJ Magazine*, vol. 46, no. 4, pp. 431–437, 2005.

[2] Christian Collberg, Clark Thomborson and Douglas Low: “A Taxonomy of Obfuscating Transformation”, *Technical Report 148, Dept. of Computer Science, Univ. of Auckland*, 1997.

[3] Jan Cappaert and Bart Preneel: “A General Model for Hiding Control Flow”, *Proceedings of the tenth annual ACM workshop on Digital rights management*, pp. 35–42, 2010.

[4] Chenxi Wang, Jack Davidson, Jonathan Hill and John Knight: “Protection of Software-based Survivability Mechanisms”, *Proc. International Conference on Dependable Systems and Networks*, pp. 193–202, 2001.

[5] Martha Mercaldi: “Using Exceptions to Obstruct Analysis of Control Flow Structure”, Bachelor’s Thesis, Harvard College, Cambridge, Massachusetts, 2002.

[6] Christian Collberg, Clark Thomborson and Douglas Low: “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs”, *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 184–196, 1998.

[7] 齋藤鐵男, 鈴木貢, 渡邊坦: “非可約な制御フローグラフのための簡潔で高速な支配木と支配辺境の検出算法”, 情報処理学会論文誌. プログラミング, vol. 43, no. 8, pp. 72–86, 2002.

[8] 滝本宗宏: “言語処理系特論”, 東京理科大学理工学研究科情報科学専攻, 2010 年度テキスト, `http://www.cs.is.noda.tus.ac.jp/~mune/master/10/loop.pdf`, 2014/1/24 閲覧.

[9] 刑部裕介, 双紙正和, 宮地充子: “オブジェクト指向言語の難読化の提案”, 電子情報通信学会技術研究報告. *ISEC,* 情報セキュリティ, vol. 102, no. 71, pp. 33–38, 2002.

[10] Oracle Corporation: “Java SE HotSpot at a Glance”, `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html`, 2014/1/16 閲覧.

[11] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell and David Cox: “Design of the Java HotSpot$^{\text{TM}}$Client Compiler for Java 6”, *ACM Transactions on Architecture and Code Optimization*, vol. 5, article 7, 2008.

[12] IBM: “東京基礎研究所 研究紹介 Java JIT compiler”, `http://www.research.`

ibm.com/trl/projects/jit/, 2014/1/16 閲覧.

[13] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu and Toshio Nakatani: “Overview of the IBM Java Just-in-Time Compiler”, *IBM Systems Journal*, vol. 39, pp. 175–193, 2000.

[14] 石崎一明: “Java Just-In-Time コンパイラにおける最適化手法”, 早稲田大学大学院理工学研究科 博士論文, http://dspace.wul.waseda.ac.jp/dspace/handle/2065/337, 2002.

[15] Google: “V8 JavaScript Engine”, Google Project Hosting, http://code.google.com/p/v8/, 2014/1/16 閲覧.

[16] Mozila: “IonMonkey in Firefox 18”, http://blog.mozilla.org/javascript/2012/09/12/ionmonkey-in-firefox-18/, 2014/1/16 閲覧.

[17] David Detlefs and Ole Agesen: “Inlining of Virtual Methods”, *ECOOP ’99 LNCS 1628*, pp. 258–277, 1999.

[18] Junpyo Lee,Byung-Sun Yang, Suhyun Kim, Seungil Lee, Yoo C. Chung, Heugbok Lee, Je hyung Lee, Soo-Mook Moon, Kemal Ebcioğlu and Eril Altman: “Reducing Virtual Call Overheads in Java VM Just-In-Time Compiler”, *The 4th Annual Workshop on Interaction between Compilers and Computer Architectures*, pp. 21–33, 2000.

[19] Masahide Nakamura, Akito Monden, Tomoaki Itoh, Ken-ichi Matsumoto, Yuichiro Kanzaki and Hirotsugu Satoh: “Queue-Based Cost Evaluation of Mental Simulation Process in Program Comprehension”, *Proceedings of the 9th International Symposium on Software Metrics*, ISBN 0-7695-1987-3, pp. 351–360, 2003.

[20] 二村 阿美, 門田 暁人, 玉田 春昭, 神崎 雄一郎, 中村 匡秀, 松本 健一: “命令のランダム性に基づくプログラム難読化の評価”, コンピュータソフトウェア, vol. 30, no. 3, pp. 18–24, 2013.

[21] Peiyi Tang: “Complete Inlining of Recursive Calls: Beyond Tail-Recursion Elimination”, *Proceeding og the 44th annual Southeast regional conference*, pp. 579–584, 2006.

[22] “Java Compiler Compiler™(JavaCC™) - The Java Parser Generator”, https://javacc.java.net, 2014/1/23 閲覧.

[23] Sun Microsystems: “Java1.5.jjt”, powerjava Project, http://code.google.com/p/powerjava/source/browse/trunk/powerJava0.2/Java1.5.jjt?r=2, 2010/5/21 作成.

[24] Oracle: “HashMap”, Java Platform, Standard Edition 7 API 仕様, http://docs.

oracle.com/javase/jp/7/api/java/util/HashMap.html, 2014/1/19 閲覧.

[25] Oracle: “System.nanoTime()”, Java Platform, Standard Edition 7 API 仕様, http://docs.oracle.com/javase/jp/7/api/java/lang/System.html#nanoTime(), 2014/1/20 閲覧.

# 付録 A

# 機械語コード

## A.1 プログラム 5.3 が JIT コンパイルされ得られた機械語

プログラム A.1 プログラム 5.3 が JIT コンパイルされ得られた機械語

```
    134     1 %           Test::test1 @ 17 (69 bytes)
                            @ 25   Test::Pt (20 bytes)   inline (hot)
Decoding compiled method 0x000000010b443910:
Code:
[Entry Point]
[Verified Entry Point]
[Constants]
  # {method} 'test1' '(I)V' in 'Test'
  0x000000010b443a60: callq  0x000000010abaf236  ;   {runtime_call}
  0x000000010b443a65: data32 data32 nopw 0x0(%rax,%rax,1)
  0x000000010b443a70: sub    $0x18,%rsp
  0x000000010b443a77: mov    %rbp,0x10(%rsp)
  0x000000010b443a7c: mov    0x10(%rsi),%ebp
  0x000000010b443a7f: mov    0x18(%rsi),%ebx
  0x000000010b443a82: mov    (%rsi),%r14d
  0x000000010b443a85: mov    0x8(%rsi),%r13d
  0x000000010b443a89: mov    %rsi,%rdi
  0x000000010b443a8c: movabs $0x10ac03450,%r10
  0x000000010b443a96: callq  *%r10                ;*iload_3
                                                  ; - Test::test1@17 (line 65)
  0x000000010b443a99: movabs $0x7e5a08f60,%r10    ;   {oop(a 'java/lang/Class' = 'Test')}
  0x000000010b443aa3: jmp    0x000000010b443ac4
  0x000000010b443aa5: data32 data32 nopw 0x0(%rax,%rax,1)
  0x000000010b443ab0: add    $0x1,%r8
  0x000000010b443ab4: mov    %r8,0x80(%r10)       ;*goto
                                                  ; - Test::test1@53 (line 65)
  0x000000010b443abb: inc    %r14d                ; OopMap{r10=Oop off=94}
                                                  ;*goto
                                                  ; - Test::test1@53 (line 65)
  0x000000010b443abe: test   %eax,-0xd4aac4(%rip)        # 0x000000010a6f9000
                                                  ;*iload_3
                                                  ; - Test::test1@17 (line 65)
                                                  ;   {poll}
  0x000000010b443ac4: cmp    $0xa,%r14d
  0x000000010b443ac8: jge    0x000000010b443afa   ;*if_icmpge
                                                  ; - Test::test1@20 (line 65)
  0x000000010b443aca: mov    %r14d,%r11d
  0x000000010b443acd: imul   %r14d,%r11d
  0x000000010b443ad1: mov    0x80(%r10),%r8       ;*getstatic val
                                                  ; - Test::test1@31 (line 67)
  0x000000010b443ad8: mov    %r13d,%r9d
  0x000000010b443adb: shl    $0x3,%r9d
  0x000000010b443adf: sub    %r13d,%r9d
  0x000000010b443ae2: imul   %r13d,%r9d
  0x000000010b443ae6: dec    %r9d
  0x000000010b443ae9: cmp    %r11d,%r9d
  0x000000010b443aec: jne    0x000000010b443ab0   ;*if_icmpeq
                                                  ; - Test::Pt@11 (line 5)
                                                  ; - Test::test1@25 (line 66)
  0x000000010b443aee: dec    %r8
  0x000000010b443af1: mov    %r8,0x80(%r10)       ;*putstatic val
                                                  ; - Test::test1@47 (line 69)
  0x000000010b443af8: jmp    0x000000010b443abb   ;*goto
                                                  ; - Test::test1@59 (line 64)
  0x000000010b443afa: inc    %r13d                ; OopMap{r10=Oop off=157}
                                                  ;*goto
                                                  ; - Test::test1@59 (line 64)
  0x000000010b443afd: test   %eax,-0xd4ab03(%rip)        # 0x000000010a6f9000
                                                  ;*goto
                                                  ; - Test::test1@59 (line 64)
                                                  ;   {poll}
  0x000000010b443b03: cmp    $0xa,%r13d
  0x000000010b443b07: jge    0x000000010b443b0e   ;*iconst_0
```

```
                                                  ; - Test::test1@15 (line 65)
  0x000000010b443b09: xor    %r14d,%r14d
  0x000000010b443b0c: jmp    0x000000010b443ac4  ;*goto
                                                  ; - Test::test1@65 (line 63)
  0x000000010b443b0e: inc    %ebp                ; OopMap{r10=Oop off=176}
                                                 ;*goto
                                                  ; - Test::test1@65 (line 63)
  0x000000010b443b10: test   %eax,-0xd4ab16(%rip)        # 0x000000010a6f9000
                                                 ;*goto
                                                  ; - Test::test1@65 (line 63)
                                                  ;   {poll}
  0x000000010b443b16: cmp    %ebx,%ebp
  0x000000010b443b18: jge    0x000000010b443b1f  ;*if_icmpge
                                                  ; - Test::test1@4 (line 63)
  0x000000010b443b1a: xor    %r13d,%r13d
  0x000000010b443b1d: jmp    0x000000010b443b09
  0x000000010b443b1f: add    $0x10,%rsp
  0x000000010b443b23: pop    %rbp
  0x000000010b443b24: test   %eax,-0xd4ab2a(%rip)        # 0x000000010a6f9000
                                                  ;   {poll_return}
  0x000000010b443b2a: retq                        ;*iload_3
                                                  ; - Test::test1@17 (line 65)
```

# A.2 プログラム 5.6 が JIT コンパイルされ得られた機械語

プログラム A.2 プログラム 5.6 が JIT コンパイルされ得られた機械語

```
     66    1 %           Test::test4 @ 17 (52 bytes)
                           @ 30   Test$A::m (9 bytes)   inline (hot)
Decoding compiled method 0x000000010f5be410:
Code:
[Entry Point]
[Verified Entry Point]
[Constants]
  # {method} 'test4' '(I)V' in 'Test'
  0x000000010f5be580: callq  0x000000010ed32236  ;   {runtime_call}
  0x000000010f5be585: data32 data32 nopw 0x0(%rax,%rax,1)
  0x000000010f5be590: mov    %eax,-0x14000(%rsp)
  0x000000010f5be597: push   %rbp
  0x000000010f5be598: sub    $0x30,%rsp
  0x000000010f5be59c: mov    0x10(%rsi),%r13d
  0x000000010f5be5a0: mov    0x18(%rsi),%ebp
  0x000000010f5be5a3: mov    0x8(%rsi),%r14d
  0x000000010f5be5a7: mov    (%rsi),%ebx
  0x000000010f5be5a9: mov    %rsi,%rdi
  0x000000010f5be5ac: movabs $0x10ed86450,%r10
  0x000000010f5be5b6: callq  *%r10                ;*iload_3
                                                  ; - Test::test4@17 (line 95)
  0x000000010f5be5b9: mov    %ebx,%r10d
  0x000000010f5be5bc: inc    %r10d
  0x000000010f5be5bf: movabs $0x7e5a08f60,%rcx  ;   {oop(a 'java/lang/Class' = 'Test')}
  0x000000010f5be5c9: xor    %r11d,%r11d
  0x000000010f5be5cc: cmp    %r11d,%r10d
  0x000000010f5be5cf: cmovl  %r11d,%r10d
  0x000000010f5be5d3: mov    $0xa,%r11d
  0x000000010f5be5d9: cmp    %r11d,%r10d
  0x000000010f5be5dc: mov    $0xa,%r8d
  0x000000010f5be5e2: cmovg  %r8d,%r10d
  0x000000010f5be5e6: movabs $0x7e5a0ebb8,%rsi  ;   {oop(a 'Test$O'[][10] )}
  0x000000010f5be5f0: jmp    0x000000010f5be610
  0x000000010f5be5f2: mov    (%rsp),%ebp          ;*goto
                                                  ; - Test::test4@42 (line 94)
  0x000000010f5be5f5: inc    %r14d                ; OopMap{rcx=Oop rsi=Oop off=120}
                                                 ;*goto
                                                  ; - Test::test4@42 (line 94)
  0x000000010f5be5f8: test   %eax,-0xdd05fe(%rip)        # 0x000000010e7ee000
                                                 ;*goto
                                                  ; - Test::test4@42 (line 94)
                                                  ;   {poll}
  0x000000010f5be5fe: cmp    $0xa,%r14d
  0x000000010f5be602: jge    0x000000010f5be77b  ;*iconst_0
                                                  ; - Test::test4@15 (line 95)
  0x000000010f5be608: xor    %ebx,%ebx
  0x000000010f5be60a: mov    $0x1,%r10d           ;*iload_3
                                                  ; - Test::test4@17 (line 95)
  0x000000010f5be610: cmp    $0xa,%ebx
  0x000000010f5be613: jge    0x000000010f5be5f5  ;*if_icmpge
                                                  ; - Test::test4@20 (line 95)
  0x000000010f5be615: mov    0x80(%rcx),%rdi     ;*getstatic val
                                                  ; - Test$A::m@0 (line 121)
                                                  ; - Test::test4@30 (line 96)
  0x000000010f5be61c: cmp    $0xa,%r14d
  0x000000010f5be620: jae    0x000000010f5be7d1
```

```
0x000000010f5be626: mov    0x10(%rsi,%r14,4),%eax  ;*aaload
                                                   ; - Test::test4@27 (line 96)
0x000000010f5be62b: mov    0xc(%r12,%rax,8),%r8d  ; implicit exception: dispatches to 0
    x000000010f5be839
0x000000010f5be630: cmp    %r8d,%ebx
0x000000010f5be633: jae    0x000000010f5be7f0
0x000000010f5be639: mov    %ebp,(%rsp)
0x000000010f5be63c: lea    (%r12,%rax,8),%r9
0x000000010f5be640: mov    0x10(%r9,%rbx,4),%ebp  ;*aaload
                                                   ; - Test::test4@29 (line 96)
0x000000010f5be645: mov    0x8(%r12,%rbp,8),%r11d  ; implicit exception: dispatches to 0
    x000000010f5be829
0x000000010f5be64a: cmp    $0xf5790a7b,%r11d   ;   {oop('Test$A')}
0x000000010f5be651: jne    0x000000010f5be811  ;*invokevirtual m
                                                   ; - Test::test4@30 (line 96)
0x000000010f5be657: add    $0x1,%rdi           ;*ladd
                                                   ; - Test$A::m@4 (line 121)
                                                   ; - Test::test4@30 (line 96)
0x000000010f5be65b: mov    %rdi,0x80(%rcx)     ;*putstatic val
                                                   ; - Test$A::m@5 (line 121)
                                                   ; - Test::test4@30 (line 96)
0x000000010f5be662: inc    %ebx                ;*iinc
                                                   ; - Test::test4@33 (line 95)
0x000000010f5be664: cmp    %r10d,%ebx
0x000000010f5be667: jge    0x000000010f5be66e  ;*if_icmpge
                                                   ; - Test::test4@20 (line 95)
0x000000010f5be669: mov    (%rsp),%ebp
0x000000010f5be66c: jmp    0x000000010f5be61c
0x000000010f5be66e: mov    $0xa,%r10d
0x000000010f5be674: cmp    %r8d,%r10d
0x000000010f5be677: cmovg  %r8d,%r10d
0x000000010f5be67b: mov    %r10d,%r11d
0x000000010f5be67e: add    $0xfffffffffffffffd,%r11d
0x000000010f5be682: cmp    %r11d,%r10d
0x000000010f5be685: mov    $0x80000000,%r10d
0x000000010f5be68b: cmovl  %r10d,%r11d
0x000000010f5be68f: cmp    %r11d,%ebx
0x000000010f5be692: jge    0x000000010f5be738
0x000000010f5be698: nopl   0x0(%rax,%rax,1)    ;*aaload
                                                   ; - Test::test4@27 (line 96)
0x000000010f5be6a0: mov    0x10(%r9,%rbx,4),%r10d  ;*aaload
                                                   ; - Test::test4@29 (line 96)
0x000000010f5be6a5: mov    0x8(%r12,%r10,8),%ebp  ; implicit exception: dispatches to 0
    x000000010f5be829
0x000000010f5be6aa: cmp    $0xf5790a7b,%ebp    ;   {oop('Test$A')}
0x000000010f5be6b0: jne    0x000000010f5be7a6  ;*invokevirtual m
                                                   ; - Test::test4@30 (line 96)
0x000000010f5be6b6: movslq %ebx,%rdx
0x000000010f5be6b9: mov    0x14(%r9,%rdx,4),%r10d  ;*aaload
                                                   ; - Test::test4@29 (line 96)
0x000000010f5be6be: mov    %rdi,%rbp
0x000000010f5be6c1: add    $0x1,%rbp
0x000000010f5be6c5: mov    %rbp,0x80(%rcx)     ;*putstatic val
                                                   ; - Test$A::m@5 (line 121)
                                                   ; - Test::test4@30 (line 96)
0x000000010f5be6cc: mov    0x8(%r12,%r10,8),%ebp  ; implicit exception: dispatches to 0
    x000000010f5be829
0x000000010f5be6d1: cmp    $0xf5790a7b,%ebp    ;   {oop('Test$A')}
0x000000010f5be6d7: jne    0x000000010f5be795  ;*invokevirtual m
                                                   ; - Test::test4@30 (line 96)
0x000000010f5be6dd: mov    0x18(%r9,%rdx,4),%r10d  ;*aaload
                                                   ; - Test::test4@29 (line 96)
0x000000010f5be6e2: mov    %rdi,%rbp
0x000000010f5be6e5: add    $0x2,%rbp
0x000000010f5be6e9: mov    %rbp,0x80(%rcx)     ;*putstatic val
                                                   ; - Test$A::m@5 (line 121)
                                                   ; - Test::test4@30 (line 96)
0x000000010f5be6f0: mov    0x8(%r12,%r10,8),%ebp  ; implicit exception: dispatches to 0
    x000000010f5be829
0x000000010f5be6f5: cmp    $0xf5790a7b,%ebp    ;   {oop('Test$A')}
0x000000010f5be6fb: jne    0x000000010f5be7ac  ;*invokevirtual m
                                                   ; - Test::test4@30 (line 96)
0x000000010f5be701: mov    0x1c(%r9,%rdx,4),%r10d  ;*aaload
                                                   ; - Test::test4@29 (line 96)
0x000000010f5be706: mov    %rdi,%rdx
0x000000010f5be709: add    $0x3,%rdx
0x000000010f5be70d: mov    %rdx,0x80(%rcx)     ;*putstatic val
                                                   ; - Test$A::m@5 (line 121)
                                                   ; - Test::test4@30 (line 96)
0x000000010f5be714: mov    0x8(%r12,%r10,8),%ebp  ; implicit exception: dispatches to 0
    x000000010f5be829
0x000000010f5be719: cmp    $0xf5790a7b,%ebp    ;   {oop('Test$A')}
0x000000010f5be71f: jne    0x000000010f5be79b  ;*invokevirtual m
                                                   ; - Test::test4@30 (line 96)
0x000000010f5be721: add    $0x4,%rdi           ;*ladd
                                                   ; - Test$A::m@4 (line 121)
                                                   ; - Test::test4@30 (line 96)
0x000000010f5be725: mov    %rdi,0x80(%rcx)     ;*putstatic val
                                                   ; - Test$A::m@5 (line 121)
                                                   ; - Test::test4@30 (line 96)
0x000000010f5be72c: add    $0x4,%ebx           ;*iinc
```

```
                                                  ; - Test::test4@33 (line 95)
0x000000010f5be72f: cmp    %r11d,%ebx
0x000000010f5be732: jl     0x000000010f5be6a0  ;*if_icmpge
                                                  ; - Test::test4@20 (line 95)
0x000000010f5be738: cmp    $0xa,%ebx
0x000000010f5be73b: jge    0x000000010f5be5f2
0x000000010f5be741: data32 xchg %ax,%ax        ;*aaload
                                                  ; - Test::test4@27 (line 96)
0x000000010f5be744: cmp    %r8d,%ebx
0x000000010f5be747: jae    0x000000010f5be7ed
0x000000010f5be74d: mov    0x10(%r9,%rbx,4),%r10d  ;*aaload
                                                  ; - Test::test4@29 (line 96)
0x000000010f5be752: mov    0x8(%r12,%r10,8),%ebp  ; implicit exception: dispatches to 0
    x000000010f5be829
0x000000010f5be757: cmp    $0xf5790a7b,%ebp    ;   {oop('Test$A')}
0x000000010f5be75d: jne    0x000000010f5be817  ;*invokevirtual m
                                                  ; - Test::test4@30 (line 96)
0x000000010f5be763: add    $0x1,%rdi           ;*ladd
                                                  ; - Test$A::m@4 (line 121)
                                                  ; - Test::test4@30 (line 96)
0x000000010f5be767: mov    %rdi,0x80(%rcx)     ;*putstatic val
                                                  ; - Test$A::m@5 (line 121)
                                                  ; - Test::test4@30 (line 96)
0x000000010f5be76e: inc    %ebx                ;*iinc
                                                  ; - Test::test4@33 (line 95)
0x000000010f5be770: cmp    $0xa,%ebx
0x000000010f5be773: jge    0x000000010f5be5f2  ;*if_icmpge
                                                  ; - Test::test4@20 (line 95)
0x000000010f5be779: jmp    0x000000010f5be744  ;*goto
                                                  ; - Test::test4@48 (line 93)
0x000000010f5be77b: inc    %r13d               ; OopMap{rcx=Oop rsi=Oop off=510}
                                                  ;*goto
                                                  ; - Test::test4@48 (line 93)
0x000000010f5be77e: test   %eax,-0xdd0784(%rip)        # 0x000000010e7ee000
                                                  ;*goto
                                                  ; - Test::test4@48 (line 93)
                                                  ;   {poll}
0x000000010f5be784: cmp    %ebp,%r13d
0x000000010f5be787: jge    0x000000010f5be81d  ;*if_icmpge
                                                  ; - Test::test4@4 (line 93)
0x000000010f5be78d: xor    %r14d,%r14d
0x000000010f5be790: jmpq   0x000000010f5be608
0x000000010f5be795: lea    (%r12,%r10,8),%rbp
0x000000010f5be799: jmp    0x000000010f5be7a2
0x000000010f5be79b: lea    (%r12,%r10,8),%rbp  ;*aaload
                                                  ; - Test::test4@29 (line 96)
0x000000010f5be79f: add    $0x2,%ebx
0x000000010f5be7a2: inc    %ebx                ;*iinc
                                                  ; - Test::test4@33 (line 95)
0x000000010f5be7a4: jmp    0x000000010f5be7b3
0x000000010f5be7a6: lea    (%r12,%r10,8),%rbp
0x000000010f5be7aa: jmp    0x000000010f5be7b3
0x000000010f5be7ac: lea    (%r12,%r10,8),%rbp  ;*aaload
                                                  ; - Test::test4@29 (line 96)
0x000000010f5be7b0: add    $0x2,%ebx
0x000000010f5be7b3: mov    $0xfffffffe,%esi
0x000000010f5be7b8: mov    %r13d,0x4(%rsp)
0x000000010f5be7bd: mov    %r14d,0x8(%rsp)
0x000000010f5be7c2: mov    %ebx,0xc(%rsp)
0x000000010f5be7c6: nop
0x000000010f5be7c7: callq  0x000000010f598020  ; OopMap{rbp=Oop off=588}
                                                  ;*invokevirtual m
                                                  ; - Test::test4@30 (line 96)
                                                  ;   {runtime_call}
0x000000010f5be7cc: callq  0x000000010ed32236  ;   {runtime_call}
0x000000010f5be7d1: mov    $0xfffffffe4,%esi
0x000000010f5be7d6: mov    %r13d,(%rsp)
0x000000010f5be7da: mov    %ebx,0x8(%rsp)
0x000000010f5be7de: mov    %r14d,0xc(%rsp)
0x000000010f5be7e3: callq  0x000000010f598020  ; OopMap{off=616}
                                                  ;*aaload
                                                  ; - Test::test4@27 (line 96)
                                                  ;   {runtime_call}
0x000000010f5be7e8: callq  0x000000010ed32236  ;*aaload
                                                  ; - Test::test4@27 (line 96)
                                                  ;   {runtime_call}
0x000000010f5be7ed: mov    (%rsp),%ebp
0x000000010f5be7f0: mov    $0xfffffffe4,%esi
0x000000010f5be7f5: mov    %r13d,(%rsp)
0x000000010f5be7f9: mov    %r14d,0x4(%rsp)
0x000000010f5be7fe: mov    %eax,0xc(%rsp)
0x000000010f5be802: mov    %ebx,0x10(%rsp)
0x000000010f5be806: nop
0x000000010f5be807: callq  0x000000010f598020  ; OopMap{[12]=NarrowOop off=652}
                                                  ;*aaload
                                                  ; - Test::test4@29 (line 96)
                                                  ;   {runtime_call}
0x000000010f5be80c: callq  0x000000010ed32236  ;   {runtime_call}
0x000000010f5be811: shl    $0x3,%rbp
0x000000010f5be815: jmp    0x000000010f5be7b3
0x000000010f5be817: lea    (%r12,%r10,8),%rbp  ;*aaload
```

```
                                                  ; - Test::test4@29 (line 96)
0x000000010f5be81b: jmp     0x000000010f5be7b3
0x000000010f5be81d: add     $0x30,%rsp
0x000000010f5be821: pop     %rbp
0x000000010f5be822: test    %eax,-0xdd0828(%rip)        # 0x000000010e7ee000
                                                  ;   {poll_return}
0x000000010f5be828: retq
0x000000010f5be829: mov     $0xfffffff6,%esi
0x000000010f5be82e: nop
0x000000010f5be82f: callq   0x000000010f598020  ; OopMap{off=692}
                                                  ;*invokevirtual m
                                                  ; - Test::test4@30 (line 96)
                                                  ;   {runtime_call}
0x000000010f5be834: callq   0x000000010ed32236  ;   {runtime_call}
0x000000010f5be839: mov     $0xfffffff6,%esi
0x000000010f5be83e: mov     %ebx,%ebp
0x000000010f5be840: data32  xchg %ax,%ax
0x000000010f5be843: callq   0x000000010f598020  ; OopMap{off=712}
                                                  ;*aaload
                                                  ; - Test::test4@29 (line 96)
                                                  ;   {runtime_call}
0x000000010f5be848: callq   0x000000010ed32236  ;*aaload
                                                  ; - Test::test4@29 (line 96)
                                                  ;   {runtime_call}
```