SCIS 2015 The 32nd Symposium on
Cryptography and Information Security
Kokura, Japan, Jan. 20 - 23, 2015
The Institute of Electronics,
Information and Communication Engineers

メタプログラミング技法を用いた偽装難読化手法
A Camouflage Method with Meta Programming Techniques

福田 收真 * 稲垣 賢一 † 玉田 春昭 ‡
Kazumasa Fukuda Kenichi Inagaki Haruaki Tamada

あらまし 我々は過去にメソッド呼び出しを偽装する難読化・変数へのアクセスを偽装する難読化手法を提案した．これら手法では，本来のメソッド呼び出し・変数へのアクセスの処理を，偽のメソッド呼び出し・変数へのアクセスに置き換え，実行時に本来のメソッド呼び出し・変数のアクセスへと呼び出すように変更する．これにより，攻撃者が静的解析を行ったとき，偽のメソッド呼び出し・変数へのアクセスであると誤解すると期待できる．この二つの手法では，攻撃者を間違った情報へと誘導することでで難読化のコストを高くする．本稿では，過去の手法に加え，新たにクラスを偽装する難読化手法を提案し，これら 3 つの手法を 1 つの手法としての再定義を行う．そして，この難読化手法の効果や耐性について評価・議論を行う．

キーワード メタプログラミング，クラス偽装，メソッド偽装，フィールド偽装

1 はじめに

プログラム解析を行うためのツールは，世の中に多数公開されている [1, 2, 3]．特に高機能なツールを用いると，本来，高度な知識が必要となるプログラム解析が，特別な知識なく行えてしまう現状がある．そのような中，安易な攻撃を防ぐためには，既存のツールによる攻撃を防ぐ必要がある．従来，ソフトウェア中に含まれる秘密情報を隠すためのプログラムの難読化手法が提案されている [4, 5]．プログラムの難読化手法とは，プログラムを意図的に理解し難いプログラムへ変換することで，攻撃から秘密情報を保護する手法である．この難読化手法の中でも，プログラムの中の要素を偽装する方法に着目する．プログラムの要素を偽装することで，解析時には，偽の要素を出力し，実行時に本来の要素として実行する．私たちのグループでは，過去に，メソッド呼び出し偽装，変数アクセス偽装について難読化手法を提案した．メソッド呼び出し偽装では，メソッドフックを用いて，プログラムに書かれていないメソッドを呼び出す．変数アクセス偽装では，Instrumentation 機構により，フック処理を加えることで，他の要素へアクセスする [6]．これらの手法によって，攻撃者は，偽の要素が処理されていると錯覚する．

本稿では，過去の手法に加え，新たにクラスを偽装する手法を提案する．クラスを偽装することにより，別のオブジェクトが利用されていると誘導することが期待できる．また，過去の手法と新たな手法を統括し，偽装難読化手法として再定義する．それぞれ 3 つの手法について，議論・評価を行う．

2 プログラム難読化

プログラム難読化とは，プログラムを意図的に理解し難いプログラムへ変換することで，攻撃から秘密情報を保護する手法である．一般的な難読化として，プログラムの秘密情報を難解な形へと変換することで，攻撃から秘密情報を保護する．難読化手法は次のように定義される．

定義 1 (プログラムの難読化) 秘密情報 X を隠蔽する難読化をプログラム p に対して適用し，難読化後ソフトウェア p' を作成したとき，以下の条件を満たす．以下の条件では，I はプログラムの入力を表し，$r(p, I)$ は I を入力とした時の p の実行結果を表す．また，$c(p, X)$ は p から X を理解するために要するコストを表す．

* 京都産業大学大学院先端情報研究科 〒 603-8555 京都府京都市北区上賀茂本山. Division of Frontier Informatics, Graduate School of Kyoto Sangyo University. Motoyama, Kamigamo, Kita-ku, Kyoto-shi, Kyoto, Japan, 603-8555. i1358095@cse.kyoto-su.ac.jp
† 京都産業大学コンピュータ理工学部 〒 603-8555 京都府京都市北区上賀茂本山. Faculty of Computer Science and Engineering, Kyoto Sangyo University. Motoyama, Kamigamo, Kita-ku, Kyoto-shi, Kyoto, Japan, 603-8555.
‡ 京都産業大学コンピュータ理工学部 〒 603-8555 京都府京都市北区上賀茂本山. Faculty of Computer Science and Engineering, Kyoto Sangyo University. Motoyama, Kamigamo, Kita-ku, Kyoto-shi, Kyoto, Japan, 603-8555. tamada@cse.kyoto-su.ac.jp

条件 1 $r(p, I) = r(p', I)$

条件 2 $c(p, X) < c(p', X)$

条件 1 では，難読化前後では，プログラムの実行結果が変わらないことを表す．つまり，難読化では，外部仕様を保持しなければならない．条件 2 では，p' から X を理解する際，p から X を理解するよりコストが高くなることを表す．つまり，X を攻撃から保護する．

難読化は必ずしも p のソースコードに適用されるとは限らない．一般に保護対象となるソフトウェアはソースコードを公開せず，実行コードのみがリリースされることが多いためである．

3 提案手法

3.1 概要

従来の難読化では，保護対象となる情報を隠ぺいすることを目的としている．対して，提案手法は，保護対象となる情報を別の情報へ誤認させることを目的とする．本稿では，過去に我々のグループで提案したメソッドの偽装，変数の偽装手法に，新たに提案するクラスの偽装手法を加える．そして，３つの手法を偽装難読化手法として統括する．この手法は，本来参照される要素 (メソッド・変数・クラス) とは，異なる要素にアクセスすると誤認させることにより，実際に実行される処理を隠ぺいする．ただし，難読化であるため，偽装されたプログラムの挙動ではなく，本来の挙動が必要がある．そのため，偽装された要素へのアクセスをフックし，本来の挙動に戻す．これにより，プログラムは本来の挙動を示し，かつ攻撃者にプログラムの誤った理解を促せるようになる．誤った理解を促すことにより，正しいプログラムを理解するコストをより高くすることが期待できる．

図 1 はキーアイデアの模式図である．オリジナルプログラム p にこの手法を適用する手順を次に示す．難読化されたプログラムを p'，対象要素 e_t，偽の要素を e_f とする．オリジナルプログラムを左側に，難読化後のプログラムを右側に示している．オリジナルプログラムでは，直接 e_t にアクセスし，難読化後のプログラムでは，e_t へのアクセスが偽の要素 e_f へのアクセスへと変わっている．しかし，e_f へのアクセスの直前に，フックにより処理 h が実行される．この時，h では，e_f へのアクセスを e_t へのアクセスへと変換する．つまり，フックにより，難読化後のプログラムはオリジナルプログラムと同じ挙動をするようになる．

3.2 偽装難読化が満たす要件

偽装難読化は以下の要件を満たすことを目指す．

要件 1 偽のプログラムはフック機能を削除したとしても実行可能である．

要件 2 実行時，フック機能により，本来の処理へ変換する．

要件 3 偽装部分の実行後，偽のプログラムに戻す．

要件 1 は，効率的に難読化を行うための条件である．提案手法は，攻撃者の誤認を促すことを目的としている．そのため，フックを使わない場合であっても，偽のプログラムを実際に実行可能なプログラムにすることによって難読化の効果向上を狙う．要件 2 は，本来の処理を実行するために必要な要件である．偽のプログラムのまま実行すると，当然，偽のプログラムの処理になるため，本来の処理へと変換する．要件 3 は，次のプログラム実行のための要件である．プログラムを攻撃するときには，幾度もプログラムの実行を行うと予想される．そのため，本来のプログラムに戻されている時間をできるだけ短くすることが望ましい．

3.3 キーアイデア

ここでは，提案手法のキーアイデアを，難読化処理と復帰処理の２つの処理に分けて説明する．難読化処理は，プログラムの実行前に静的にプログラムを変換する処理である．この変換により，プログラムは本来の挙動とは異なる挙動になる．そして，実行時に本来の挙動を行うためにプログラムに仕掛けを施す必要がある．これを復帰処理と呼ぶ．ここでは，オリジナルプログラム p に提案手法を適用する手順について述べる．難読化されたプログラムを p'，偽装対象の要素を e_t，偽装後の要素を e_f とする．また，フックを h_1, h_2 とする．h_1 は，e_f を e_t へと置き換える処理 $(e_f \to e_t)$，h_2 は，h_1 の置き換えを戻す処理 $(e_t \to e_f)$ である．h_1 と h_2 が実行されるタイミングをそれぞれ，t_1，t_2 とする．

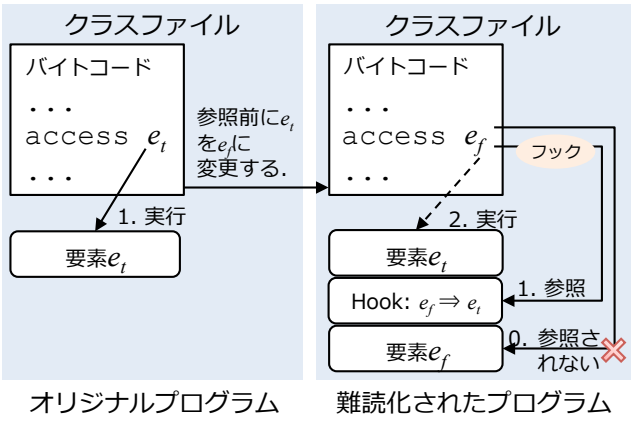

図 1: 偽装難読化の模式図

3.4 難読化処理

難読化処理は，先に述べたように，プログラムの実行前に行う処理であり，次の手順で実行する．

1. ユーザーは難読化対象の要素 e_t と偽装後の要素となる e_f を選択する．
2. プログラム中での e_t へのアクセスを e_f へのアクセスへ置換する．
3. p' に e_f を破棄し e_t に置き換える処理 h_1 を追加する．
4. p' に h_1 の逆変換を行う処理 h_2 を追加する．
5. e_f へのアクセスの前に処理 h_1 が起動するようフックを導入する．
6. e_f へのアクセス後に処理 h_2 が起動するようフックを導入する．

ユーザーが対象となる要素 e_t と e_f を選択する．一般に，開発者自身は偽装したい対象を知っている．その対象を任意に選択する．ただし，e_t が e_f に入れ替わったとしても文法的に正しいものでなければならない．次に，プログラム中の e_t へのアクセスを e_f へのアクセスへと置換する．つまり，プログラム上での記述は，偽の要素 e_f へのアクセスとなる．p' に h_1 を追加する．h_1 は，$e_f \rightarrow e_t$ への変換する処理が行われる．同じく，h_1 の逆変換を行う $h_2(e_t \rightarrow e_f)$ を p' に追加する．h_1 と h_2 は e_f へのアクセス前後で行われる．最後に，h_1 と h_2 が e_f へのアクセス前後で行われるようにフックを導入する．

3.5 復帰処理

ここでは，実行時にどのようなタイミングで本来の処理内容に戻すのか，また，偽装された処理内容に戻すのかについて述べる．

1. プログラム実行のタイミングが t_1 のときに h_1 を実行し，復帰処理を開始する．
2. h_1 により，e_f へのアクセスを e_t へのアクセスに置き換える．
3. プログラム実行のタイミングが t_2 のときに h_2 を実行し，再偽装処理を開始する．
4. h_2 により，h_1 の逆変換を行い，e_t へのアクセスを e_f へのアクセスにする．

まず，変換されていない状態では，e_f が呼び出される．t_1 のタイミングで h_1 を呼び出し，復帰処理を行う．この処理により，e_f が e_t へと変換され，本来の処理を行うプログラムへと戻る．また，t_2 では，逆変換を行う．この処理により，e_f が e_t へと再偽装され，本来のプログラムに戻っている時間をできるだけ短いものにする．

なお，t_1 は，常に t_2 の後であるとする．また，t_1，t_2 は予め難読化ツールによって決められた任意のタイミングである．例えば，t_1 がプログラム実行時，t_2 がプログラム終了時といったパターンや．t_1 が e_f へのアクセス直前，t_2 が e_f へのアクセス直後，などといったパターンが考えられる．ただし，t_1 から t_2 の期間では，プログラムが本来の処理内容が暴露されている状態であるため，できるだけ短い期間であることが望ましい．

図 2 は，本手法の実行時のシーケンス図である．復帰処理を適用しない場合と適用する場合の 2 種類を示している．復帰処理を適用しない場合は，e_f へのアクセスがそのまま e_f となり，本来の挙動にはならない．一方，復帰処理を行った場合は，上記で説明した内容が行われ，e_f へのアクセスにより，e_t へのアクセスに置き換わるため，プログラムの本来の挙動となる．

4 偽装難読化手法

ここでは，最初に我々が過去に提案した 2 つの偽装難読化手法である，メソッド偽装難読化手法 [7, 8] と，変数偽装難読化手法 [9] を説明する．その後，本稿で提案するクラス偽装難読化手法について説明する．なお，これらの手法は，主に Java 言語に対する手法として提案しているものの，他のプラットフォームに対してもキーアイデアは適用可能である．

4.1 メソッド偽装難読化手法

メソッド偽装難読化手法は，偽のメソッド呼び出しにより，ツールで得られる情報を意味のないものにすることを目指す．具体的には，ツールにより得られるコールグラフを偽のコールグラフに偽装する手法である．コールグラフは解析の初期段階で参照されることが多いため，コールグラフを偽装することで理解を大幅に困難にさせることが期待できる．

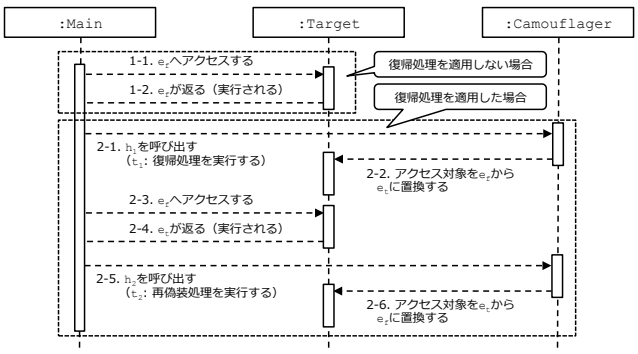

図 2: 実行時のシーケンス図

実装方法は，invokedynamic 命令を用いる方法 [7, 8] と Instrumentation 機構を用いる方法 [9] を提案した．invokedynamic 命令では，invokedynamic 命令に対応していないツールはクラッシュして，逆変換を妨げられ，Instrumentation 機構では，すべてのツールが偽装メソッドを呼び出すことが確認できた．

この手法では，偽のメソッドは任意に選択できる．しかし，実際に呼び出せない（文法的に正しくない）偽のメソッドを選択することは好ましくない．実行可能ではないプログラムでは，攻撃者に不審感を持たれ結果的に提案手法が見破られる恐れがあるためである．そこで，偽のメソッドは，文法的に正しいものであることはもちろん，意味的にも齟齬の少ないものであることが望まれる．

4.2 変数偽装難読化手法

変数偽装難読化手法は，偽の変数アクセスへと誘導することによって，メソッド呼び出し偽装では対応できないより詳細な処理の偽装を行う．具体的には，setter/getter の処理内容や，通常の変数アクセスを偽装する．メソッド偽装難読化手法を用いて，setter/getter を別の setter/getter に偽装することにより，対象となる変数の偽装が可能である．しかし，この方法であっても，setter/getter メソッド内でアクセスする変数を偽装できない．変数偽装難読化手法は，変数のアクセス自体を偽装することにより，ソースコードを復元するデコンパイラなどのツールに対する保護効果が期待できる．

この手法は，Instrumentation 機構を用いて実装する [9]．変数アクセスに対するフックが存在しないためである．Instrumentation 機構では，デコンパイラに対して，偽の変数アクセスの結果を得ることができた．

この手法でも，アクセスできない変数を偽の変数とすることができる．しかし，メソッド呼び出しと同様で，意味的に正しい変数を偽装変数とすることで，より効果的な難読化となる．

4.3 クラス偽装

本稿で，新たにクラスの偽装を行う難読化手法を提案する．提案手法は，偽のクラスのインスタンス作成を行い，本来とは異なるクラスのオブジェクトが作成されるようにする．提案手法により，クラス図作成ツールでのリバースエンジニアリング攻撃を妨害できるようになる．提案手法の実現には，オブジェクト指向の特徴である継承を利用する．共通の親クラス，もしくは，共通のインターフェースを持つ 2 つの異なるクラスを対象クラス C_t と偽装クラス C_f とする．共通の親クラスもしくは，共通のインターフェースを C_p とする．この場合，C_t オブジェクトも C_f オブジェクト共に C_p の型に代入可能であることを利用する．このようにクラスを偽装すること

```
01: import java.util.*;
02:
03: public class Queue {
04:     private List<String> list = new ArrayList<>();
05:     public void enqueue(String value) {
06:         list.add(0, value);
07:     }
08:     public String dequeue() {
09:         String value = list.get(list.size() - 1);
10:         list.remove(list.size() - 1);
11:         return value;
12:     }
13: }
```

図 3: 難読化前のプログラム

```
01: import java.util.*;
02:
03: public class Queue {
04:     private List<String> list = new LinkedList<>();
05:     public void enqueue(String value) {
06:         list.add(0, value);
07:     }
08:     public String dequeue() {
09:         String value = list.get(list.size() - 1);
10:         list.remove(list.size() - 1);
11:         return value;
12:     }
13: }
```

図 4: 難読化語のプログラム

で，偽のクラスへと誘導することができると期待できる．

例えば，`ArrayList` を `LinkedList` へと偽装する．図 3 を図 4 へと偽装する．変更点は 3 行目であり，`ArrayList` が `LinkedList` に変わっている．これはどちらも `List` インターフェースを実装しており，どちらも文法的に正しいプログラムである．しかし，`ArrayList` と `LinkedList` では，データの順序が違うため，異なる処理となる．

実装方法は，invokedynamic 命令と Instrumentation 機構による 2 つの実装を行う．具体的な実装方法は，第 5 節で示す．

5 実装

5.1 提案手法の実装

本節では，偽装難読化ツールについて説明する．難読化対象の言語は，Java 言語とする．提案手法の実現には，コンストラクタのフックが必要になる．Java 言語でフック処理を行うには，invokedynamic 命令を用いる方法と，Instrumentation 機構を用いる 2 つの方法がある．どちらの方法を適用するにしても，クラスファイルを書き換える必要がある．その処理内容について，順に説明する．なお，クラスファイルの書き換えには，ASM を利用する [10]．

5.2 invokedynamic による実装

invokedynamic 命令は，Java 7 で新たに実装された JVM 命令である．この命令を使用することにより，メソッド呼び出しをフックできる．そのため，メソッド偽装難読化手法，クラス偽装難読化手法に利用できる．その一方で，invokedynamic 命令は，メソッドを対象とした命令であるため，変数偽装難読化手法には利用できない，また，Java 7 以降の実行環境でしか使用できないという制約がある．しかし，JVM 命令で用意されているため高速で実行されるというメリットが存在する．

5.3 Instrumentation 機構による実装

Instumentation 機構は，実行時にロードされたバイトコードを再定義する機構である [11]．この機構により，実行時に，任意のタイミングでバイトコードを書き換えることができ，Java 言語であっても自己書き換えが実現可能である [6]．そして，バイトコードの書き換えにより，任意のタイミングで任意の処理を織り込めるようになる．そこで，フックを行いたい箇所に，フック処理を織り込むことにより，擬似的にフックを行えるようになる．

この方法は，実行時にバイトコードの再定義を行うため，大きな遅延が発生する．しかし，Java 5 以降であれば，invokedynamic 命令が使えない環境があっても提案手法が実現可能である．

6 評価

6.1 クラス偽装の評価

新たに提案したインスタンス作成の偽装を行う難読化について，有意性を評価するため，ツールによる攻撃を想定した評価を行う．対象プログラムは，図 4 で示した Queue クラスを使用する．

ツールによる攻撃では，クラス図作成ツールを用いた評価を行う．クラス図作成ツールとは，プログラムからクラス図を作成するツールである．本評価では，UDoc[12] を用いて攻撃を行った．このツールは，クラスファイルを入力として受け取り，クラス図を作成するツールである．

図 5 は，難読化前のクラス図である．また，図 6 は難読化後のクラス図である．クラス図は UDoc によって作成されている．図 5 では，`ArrayList` を参照している．しかし，図 6 では，`LinkedList` が参照されている．つまり，UDoc による解析では偽のクラス図が評価されていることが確認できた．

6.2 ステルス性の評価

攻撃者は，攻撃の初期段階において，どのような難読化手法が適用されているかを特定しようとする．難読化手法が特定されれば，その難読化手法を無効化，もしくは回避するような解析手法が選択できるためである．そのため，難読化手法にステルス性が備わっていれば，すなわち，難読化が適用されたことの認識が困難であれば，攻撃も困難になる．

そこで，インスタンス作成の偽装を行う難読化について，ステルス性を評価を行う，ステルス性の評価としてをパープレキシティを利用する．本研究では，平均分岐数を表すパープレキシティ（perplexity）という指標を用い，ステルス性を評価する．パープレキシティは，大滝らによって提案されているプログラムの不自然さの評価手法として利用されている [13]．パープレキシティの値が低いほど，ステルス性が高くなり，攻撃者に難読化が発見されにくい．

難読化前（図 3）と難読化後（図 4 に加えてフックを行う処理を含む）を比較する．また，代表的な難読化手法を図 3 のプログラムに適用し，従来の難読化手法との比較を行う．

結果を図 7 に示す．図 7 では，左端のバーが難読化前のパープレキシティを，左から 2 番目のバーが提案手法

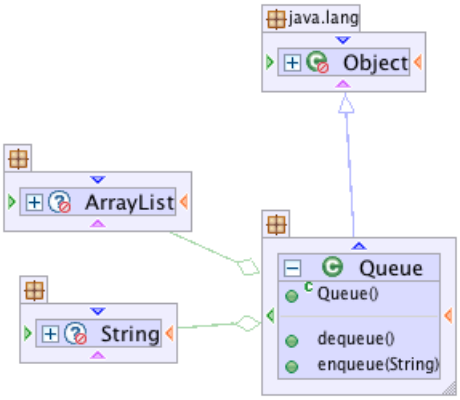

図 5: 難読化前のクラス図

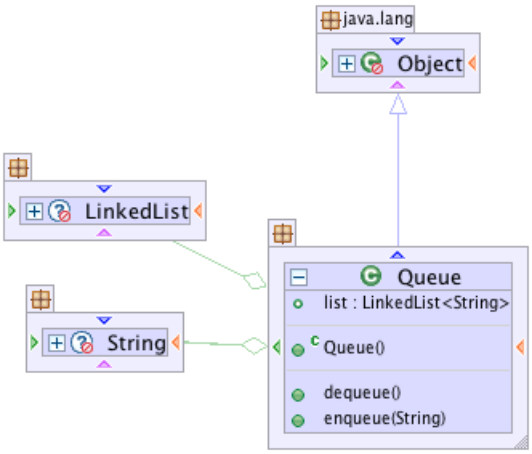

図 6: 難読化後のクラス図

のパープレキシティを表している．それ以降のバーは，既存の難読化手法のパープレキシティを表している．他の難読化手法に比べて，パープレキシティは低い．

7　議論

7.1　静的解析に対する耐性

提案手法は，ツールを利用した静的解析に対して堅牢な難読化手法である．過去の研究で，メソッド偽装難読化では，コールグラフ作成ツールに対する堅牢性，変数偽装難読化では，逆コンパイルツールを用いた堅牢性を確認している．本稿での提案手法は，クラスの偽装であるため，クラス図作成ツールを用いた評価を行った．その結果，既存の静的解析ツールでは，偽のコードが出力されることが確認できている．この偽のコードは，文法的に正しいプログラムであり，実際にも実行可能である．

既存の難読化では，冗長な処理が追加されたり，通常のプログラミングでは書かないような不自然な処理が埋め込まれることが多い．例えば，名前難読化では，変数名が意味のない名前に変更され，コントロールフロー難読化では，不自然な分岐処理が追加される．不自然なコードでは，攻撃者により強い警戒心を与え，詳細な解析を促してしまう危険性がある．そこで，本手法では，逆コンパイルにおいて自然なコードのみを出力することによって，攻撃者に警戒心を与えず，偽のコードに誘導できる．

また，提案手法は，プログラムの一部のみに適用することにより，より大きな効果を得ることができる．もし，プログラムの大部分に本手法を適用した場合，プログラムの動作と逆コンパイルにより得たコードの動作が違うことに気がつきやすくなる．プログラムの一部，特に，重要な部分のみに適用することにより，秘密情報を隠蔽することができる．

もし，フック処理を逆コンパイルした場合，評価実験でも確認したように，変換処理が書かれたコードが出力される．攻撃者が，フック処理と共に難読化されたプログラムを解析すると，実際の動作が読み解かれる．しかし，フック処理は特殊な処理が多く，読み解くコストが通常のプログラムに比べ高い．さらに，フック処理自身に提案手法を適用することも可能である．この場合であってもフック処理が自分自身を書き換える．Instrumentation 機構は，書き換え対象を制限することがないためである．また，２種類の実装方法を併用することも可能である．併用により，２種類の難読化手法を同時に解析する必要が出てくるため，より解析を困難にすることができる．

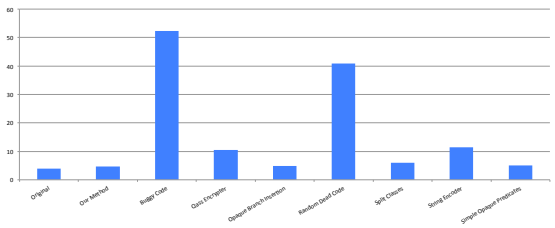

図 7: パープレキシティの比較

7.2　動的解析に対する耐性

攻撃者がプログラムを攻撃するとき，静的解析だけでなく，動的解析も併用して行われることが多い．例えば，アスペクト指向プログラミング (AOP; Aspect Oriented Programming) を利用して，メソッド呼び出しやフィールドアクセス時にログ出力させることも可能である [14]．このような攻撃が行われたとき，いくら難読化手法を用いて偽装していたとしても，実際にアクセスしたメソッドやフィールドが暴露するため，提案手法は無効化される．

しかしながら，Java 言語においては，AOP によるコードの織り込みは，コンパイル時，もしくはクラスファイルのロード時に行われる．本稿における実装ではプログラムの変更は，実行時，クラスファイルのロード後，必要なときに随時行われる．つまり，AOP がたとえクラスファイルのロード時に織り込みを行ったとしても，提案手法ではその後，プログラムを変更する機会がある．そのため，AOP による変更を無効化することも可能となる．

更に，AOP も同じ機構を使い，ロード後，様々なタイミングで書き換えてくることが考えられる．たとえ，そのような攻撃が行われた場合であっても，複数の偽装を行い，復帰処理，逆変換を複数回おこなうことで，どれが本来のアクセスであるのかを隠ぺいできる．

このように，提案手法は動的解析に対しても保護の余地があると言える．ただし，どの程度堅牢であるのか更なる議論や評価が必要である．

8　まとめ

本稿では，過去に提案したメソッド偽装難読化，変数偽装難読化に加え，新たにクラス偽装難読化を提案し，３つの手法を統括した．そして，ツールによる堅牢性の評価とステルス性の評価を行った．その結果，静的解析による攻撃に対して，高い堅牢性があることが確認できた．また，他の手法に比べて，ステルス性の増加が抑えられていることが確認できた．今後の課題として，動的解析を用いた評価・対策の議論を行う必要がある．

謝 辞

本研究の一部は，JSPS 科研費 若手研究 (B) 25730087 の助成を受けて行われた．

参考文献

[1] Pavel Kouznetsov. jad —the fast Java decompiler, Feb 2004. http://www.kpdus.com/jad.html (Last Access: 2014/12/14).

[2] Mike Strobel. Procyon. https://bitbucket.org/mstrobel/procyon (Last Access: 2014/12/14).

[3] Hex-Rays. IDA Pro. https://www.hex-rays.com/products/ida/index.shtml (Last Access: 2014/12/14).

[4] Jien-Tsai Chan and Wuu Yang. Advanced obfuscation techniques for Java bytecode. *Journal of Systems and Software*, Vol. 71, No. 1-2, pp. 1–10, April 2004.

[5] 門田暁人, 高田義広, 鳥居宏次. ループを含むプログラムを難読化する方法の提案. 電子情報通信学会論文誌, Vol. J80-D, No. 7, pp. 644–652, Jul 1997.

[6] 玉田春昭, 神崎雄一郎, 門田暁人. Java 言語を対象とした実行時多様化の試み. 2012 年 暗号と情報セキュリティシンポジウム予稿集 (SCIS2012), pp. CD–ROM, January 2012.

[7] 福田收真, 玉田春昭. メソッド呼び出し関係隠蔽のための引数順序の入れ替えによる難読化. 2014 年暗号と情報セキュリティシンポジウム予稿集 (SCIS2014), pp. CD–ROM (2D2–3), January 2014.

[8] Kazumasa Fukuda and Haruaki Tamada. An obfuscation method to build a fake call flow graph by hooking method calls. In *14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2014)*, pp. CD–ROM, June 2014. Las Vegas, U.S.A.

[9] 福田收真, 玉田春昭. フックを用いた変数アクセス偽装難読化に向けて. ソフトウェア工学の基礎 XXI, 日本ソフトウェア科学会 FOSE2014, pp. 81–86, December 2014.

[10] Eric Bruneton. ASM 4.0: A Java bytecode engineering library. http://download.forge.objectweb.org/asm/asm4-guide.pdf (Last Access: 2014/12/14).

[11] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java^{TM} Virtual Machine Specification Java SE 7 Edition*. Addison-Wesley Pub Co, Feburary 2013.

[12] Christopher Deckers. Udoc, 2006. http://sourceforge.net/projects/udoc/ (Last Access: 2014/12/14).

[13] 大滝隆貴, 大堂哲也, 神崎雄一郎, 門田暁人. Java バイトコード命令のオペコード、オペランドを用いた難読化手法のステルシネス評価. 2014 年暗号と情報セキュリティシンポジウム (SCIS 2014)(講演番号 2D2-2), January 2014.

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect oriented programming. In *Proc. European Conference on Object-Oriented Programming (ECOOP) (Springer-Verlag LNCS)*, Vol. 1241, June 1997.