情報処理学会論文誌 Vol.60 No.4 1063–1074 (Apr. 2019)

ランダムフォレストを用いた名前難読化の耐タンパ化性能の評価

磯部 陽介[1] 玉田 春昭[2,a)]

受付日 2018年8月1日, 採録日 2019年1月15日

概要：ソフトウェア内部に含まれる秘密情報を保護するために，ソフトウェア保護技術が広く用いられている．その 1 つに，プログラム中の秘密情報を秘匿するために，プログラムを読みにくく変更する難読化手法がある．プログラム中のどの特徴に着目して，読みにくく変更するかにより，異なる手法が提案されている．そのなかでも特に広く使われている難読化手法が名前難読化である．プログラム中に含まれる識別子名を意味のない名前に変更することで，可読性を下げる手法である．しかし，名前難読化の耐タンパ化性能はこれまでに議論されたことはない．もし，識別子名の復元が可能であれば，名前難読化手法は脆弱な手法であり，そのことが知られないまま使い続けられていることになる．そこで，名前難読化の耐タンパ化性能の評価のために，逆変換を試みる．特にメソッド名に着目しての復元を試みる．復元のために，難読化手法では変更されにくいメソッドの命令列，そして，メソッド引数の型と戻り値の型に着目する．大量のプログラムを用意し，これらのデータを機械学習にかけ，復元モデルを構築する．そして，復元したいメソッドをモデルに適用し，メソッド名の復元を試みる．Maven Central Repository の Java のデータからモデルを構築し，モデル構築に含まれなかった Java プログラムを対象に復元を行った．その結果，全体の 31.62%の動詞の復元に成功した．また，動詞の意味的な類似度に基づいた評価では，同義語では 33.94%，上位語の関係では 40.07%のメソッド名の動詞を復元できた．

キーワード：逆難読化，名前難読化，ランダムフォレスト，ソフトウェア保護

An Evaluation for the Performance of Tamper Resistance Transformations for the Identifier Renaming Obfuscation Method Using Random Forest

Yosuke Isobe[1] Haruaki Tamada[2,a)]

Received: August 1, 2018, Accepted: January 15, 2019

Abstract: The software protection is often used to protect the secret information in software. The one of the protection technique is the program obfuscation method. The obfuscation methods change the programs into hard to understand by preserving the input/output specification in order to hide the secret information. Various obfuscation methods were proposed focuses on a specific part of programs. One of them, there is the identifier renaming method. The identifier renaming method (IRM) changes the names of identifiers to meaningless names. IRM is often used because it is easy to implement. However, the performance of tamper resistance transformation of the IRM is not discussed. If we can restore the identifier names, the IRM has a serious vulnerability. Therefore, this paper evaluates the performance of tamper resistance transformation of the IRM by de-obfuscation. Especially, we try to restore the method names. We focus on opcodes, parameter types and return type as the clues of the restoration, which are hard to change by the identifier renaming method. The restoration model is constructed by the random forest from them. In the proposed method, we succeeded in restoration at the rate of 31.62%. On the evaluation based on the semantic similarity of the verb, in the synonym, the method restored at 33.94%. In the relation of hypernym, the restoration was succeeded at 40.07%.

Keywords: de-obfuscation, the identifier renaming method, random forest, software protection method

1. あらまし

今日，復号された鍵やパスワード，アルゴリズムなど，秘密情報を内部に持つソフトウェアは少なくない．そのようなソフトウェア内部の機密情報を保護するために，難読化が用いられることがある [1]．難読化とは，プログラムの入出力の仕様を保ったまま，理解が困難になるようにプログラムの可読性を下げる技術である．これにより，プログラム中のデータやアルゴリズムの理解，機能の改変を困難にすることが目的である．このような性質を持つソフトウェアのことを耐タンパソフトウェアといい [2]，難読化はソフトウェアを耐タンパ化する一般的な手法である．

プログラムのどの部分を変更するかによって，様々な難読化手法が提案されている [3], [4], [5]．なかでも，名前難読化は識別子名に着目した難読化で，識別子名を意味を持たない名前に変更する難読化手法である [6], [7]．識別子名を変更することで，識別子の持つ意味を秘匿し，プログラムの内容を識別子名の意味から推測されにくくすることで，プログラムの可読性を下げる．この難読化手法は，実装が容易なことから広く用いられており，ほぼすべての難読化ツールに実装されている（Dash-O [*1] や ProGuard [*2] など）．

しかしながら，名前難読化の耐タンパ化性能については十分に評価されないまま利用され続けている．たとえば，変更された識別子名を元に戻すことができれば，名前難読化の保護は無効化される．また，完全に元に戻すことができなくても，識別子名の一部を復元できれば，名前難読化の少なくとも一部が無効化されることになる．つまり，名前難読化によって提供された耐タンパ性が損なわれることになる．このような議論や評価を経ずに利用され続けている現状は，非常に危険な状態である．

そのため，本論文では，名前難読化の耐タンパ化性能の評価を目的とした，名前難読化されたプログラムの逆変換手法を提案する．逆変換が実現できれば，名前難読化の耐タンパ化性能は脆弱であるといえる．特に，本論文では，メソッド名の復元に着目する．もちろん，名前難読化では，クラス名やメソッド名，変数名など，プログラム中に現れるすべての識別子名が変更される．しかし，クラス名は変数名に依存することが多く，変数名はプログラムのドメインに依存することが多い．一方で，メソッド名の多くは，メソッドの働きに依存する．そのため，本論文では，メソッド名の復元に着目する．しかし，メソッド名を完全に復元することは難しい．なぜならば，メソッド名は一般的に動詞および目的語で構築され，目的語はクラス名やフィールド名に依存するためである．ゆえに，本研究では，メソッド名の動詞の復元を行う．

復元は，メソッド情報の機械学習によって行う．大量のメソッドを用意し，オペコード列や引数の型，戻り値の型情報を学習させる．学習した結果を復元モデルとし，名前難読化されたプログラムに適用することで，メソッド名の動詞を復元する．

以降，本論文の構成を述べる．2 章では，難読化および名前難読化を定義し，3 章で提案手法について述べる．続いて，4 章で実験と提案手法の評価について，5 章で本提案手法の妥当性について議論する．さらに，6 章で関連研究を紹介する．最後に，7 章でまとめと今後の課題を述べる．

2. 準備

2.1 難読化

難読化とは，プログラムを読みにくく変換することで，プログラム中の秘密情報を保護するための手法である．ここでは従来の定義に従い，難読化について整理する [8]．難読化の定義の前に，プログラム理解に関するコストを次のように定義する．

定義 1（プログラム理解に要するコスト）．p を与えられたプログラム，X を p に含まれるある情報の集合とする．このとき，ユーザが何らかの方法によって，p から X を取り出せたとき，ユーザは X について p を理解したとする．このときにユーザが理解に要したコストを $c(p, X)$ と表記する．

ここで，コストとは，解析に要する時間，労力，必要な知識，設備などを含むものとする．次にプログラム難読化を次のように定義する．

定義 2（プログラム難読化）．p を与えられたプログラム，X を p に含まれるある情報の集合とする．次に，p に入力 I を与えたときに得られる出力の集合を $r(p, I)$ とする．このとき，p の X に関する難読化とは，あるプログラム変換 T を p に適用し，以下の 2 つの条件を満たす $p' = T(p)$ を得ることである．

条件 1. $r(p, I) = r(p', I)$

条件 2. $c(p, X) < c(p', X)$

条件 1 は，難読化前後で入出力の仕様が保たれることを表す．すなわち，プログラムの外部的な振舞いは，難読化によって影響を受けないことを保証するものである．続く条件 2 は，難読化後のプログラム p' から X を取り出すのは，p から X を取り出すよりコストがかかるようになることを示している．

2.2 名前難読化

次に名前難読化について定義する．この名前難読化は，難読化手法の 1 つであり，プログラム中に現れる名前（識

[1] 京都産業大学大学院
Graduate School of Kyoto Sangyo University, Kyoto 603–8555, Japan
[2] 京都産業大学
Kyoto Sangyo University, Kyoto 603–8555, Japan
[a)] tamada@cc.kyoto-su.ac.jp
[*1] https://www.preemptive.com/products/dasho/
[*2] https://www.guardsquare.com/en/products/proguard

```
public class Reverse{
  public void run(String[] args){
    for(int i = 0; i < args.length; i++){
      System.out.printf("%s %s%n",
          args[i], reverse(args[i]));
    }
  }
  private String reverse(String string){
    StringBuilder sb = new StringBuilder();
    for(int i = 0; i < string.length(); i++){
      sb.append(
          string.charAt(string.length() - i - 1));
    }
    return new String(sb);
  }
  public static void main(String[] args){
    Reverse app = new Reverse();
    app.run(args);
  }
}
```

図 1 プログラム例
Fig. 1 An example program.

```
public class a{
  public void b(String[] c){
    for(int d = 0; d < c.length; d++){
      System.out.printf("%s %s%n",
          c[d], e(c[d]));
    }
  }
  private String e(String f){
    StringBuilder g = new StringBuilder();
    for(int h = 0; h < f.length(); h++){
      g.append(f.charAt(f.length() - h - 1));
    }
    return new String(g);
  }
  public static void main(String[] i){
    a j = new a();
    j.b(i);
  }
}
```

図 2 図 1 に名前難読化を適用した例
Fig. 2 An example program applied an identifier renaming method to the program shown in Fig. 1.

別子）を別のものに付け替える手法である．プログラム言語における名前は，計算機にとっては互いに区別できれば十分である一方，人間にとっては，プログラムを理解するための重要な手がかりとなる [9]．つまり，識別子が持つ意味を削除することにより，プログラム理解を困難にする難読化手法である．

定義 3（名前難読化）．p を与えられたプログラム，U_p を p に現れるすべての名前の集合，$N_p(\subset U_p)$ を難読化対象となる任意の名前の集合とする．ここで，p の名前難読化とは，名前 $n \in N_p$ を別の名前 $n' = \mathcal{C}(n)$ に変換し，別のプログラム p' を得るものとする．

U_p は主に，クラス名，フィールド名，メソッド名，ローカル変数名などの宣言部，使用部に現れる．名前難読化では，U_p からいかに N_p を選択するか，また，$\mathcal{C}$ をいかに実装するかが鍵となる．一般的に用いられる名前難読化は，宣言部に現れる名前を $\mathcal{C}$ により変換し，それにともなって使用部も変更する手法である．使用部に着目して難読化を行う手法もあるが [3], [8]，一般的に利用される名前難読化とは実現方法が大きく異なるため，本論文では対象外とする．

2.3 名前難読化の例

ここでは具体的なプログラム例を元に名前難読化を説明する．なお，説明の簡単化のためソースコードで難読化例を示すが，多くの場合バイナリを対象に実施される．**図 1**，**図 2** に名前難読化前後のプログラムを示す．図 1 は名前難読化前のプログラムで，コマンドライン引数に与えられた各文字列を逆順に表示するプログラムである．図 2 は，図 1 を名前難読化したプログラムである．$\mathcal{C}$ は，アルファベット 1 文字を出現順に付けるものとした．

図 2 から分かるように，このプログラム中で宣言されたクラス名，メソッド名，ローカル変数名がアルファベット 1 文字に変換されている．それにともない，その変数を利用している部分も名前が変更されている．一方で，標準 API で提供される名前である `System` や `String` は変更されていない．この部分を変更すると，プログラムが動作しなくなるためである．このように名前難読化では，プログラム中の名前から意味を削除することで，プログラムを読みにくくする．

3. 提案手法

3.1 キーアイデア

本論文では，名前難読化されたメソッド名の復元を行う．メソッド名の復元のために，機械学習によって復元モデルを作成する．機械学習には，大量の学習データが必要となる．そのため，本研究では，インターネット上のソフトウェアリポジトリに着目する．

近年のソフトウェア開発には，オープンソースソフトウェア（OSS）を用いることが一般的となっている．インターネット上のリポジトリから，開発に必要な OSS をダウンロードする．インターネット上のリポジトリとして，Maven Repository [*3] や Ruby Gems [*4]，GitHub [*5] などがあげられる．本研究では，これらのインターネット上の OSS を，復元モデルを作成するための学習データとする．

難読化されたプログラムから得られる情報は少ない．一方で，難読化され難い情報も存在する．その 1 つが命令列である．名前難読化はメソッドの動作を変更しないため，メソッドの処理の流れを表している命令列は変更されない．また，名前難読化ではほかにも，引数の型や数，戻り値の型を難読化しない．本手法では，これらの情報を手がかりとし，メソッド名の復元を行う．

一般的にメソッド名は，動詞および目的語で構成される．目的語は，クラス名やフィールド名に依存することが多く，名前難読化されたプログラムから推測することは難しい．そのため，本研究では，メソッド名の動詞に着目して復元を目指す．

[*3] https://mvnrepository.com
[*4] https://rubygems.org
[*5] https://github.com

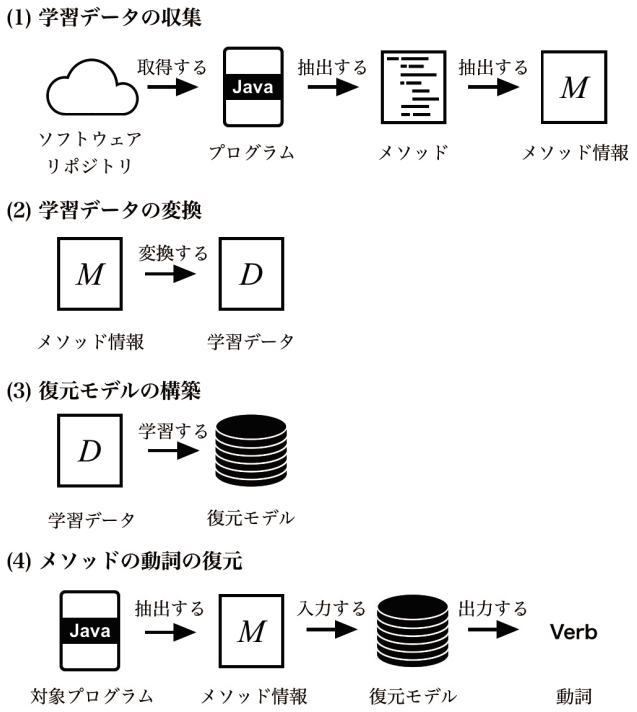


図 3 提案手法の概要図

Fig. 3 The overview of the proposed method.

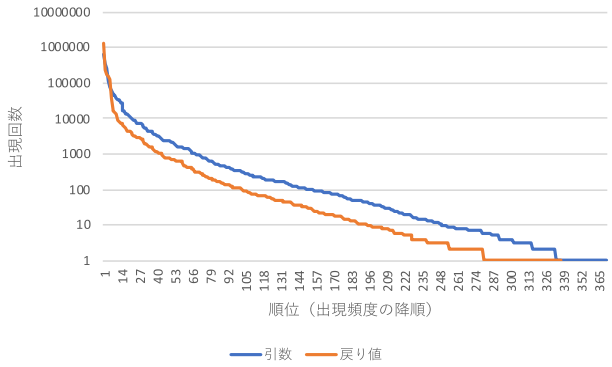


図 4 引数の型と戻り値の型の出現頻度

Fig. 4 The frequencies of the types of arguments and return variables.

3.2 全体の流れ

3.2.1 提案手法の概観

提案手法の概要を**図 3** に示す．本研究では，名前難読化されたプログラムに登場するメソッド名の動詞部分を復元する．復元には，機械学習によって構築された復元モデルを用いる．大量のプログラムから復元モデルを作成し，名前難読化されたプログラムに適用する．提案手法は次の 4 つのフェーズで構成される．

(1) 学習データの収集
(2) 学習データ変換
(3) 復元モデルの構築
(4) メソッドの動詞の復元

なお，本研究では Java 言語を対象に行い，以降で述べるプログラムはバイナリコード，すなわち，class ファイルを指すものとする．本研究の対象は，難読化されたプログラムであり，ソースコードが手に入るとは考えにくいためである．なお，バイナリからメソッド名，およびメソッドのシグネチャ，そしてメソッドの命令列の取得が可能であれば，他言語への応用が可能である．

3.2.2 学習データの収集

本フェーズでは，機械学習に用いるデータを収集する．まず，ソフトウェアリポジトリからプログラム集合 P を取得する（$P = \{p_1, p_2, \ldots, p_n\}$）．次に，プログラム p_i（$1 \leq i \leq n$）から，メソッド情報集合 M_i を取り出す（$M_i = \{m_{i,1}, m_{i,2}, \ldots, m_{i,l}\}$）．$m_{i,j}$（$1 \leq i \leq n, 1 \leq j \leq l$）には，メソッド名 $c_{i,j}$，引数の型 $A_{i,j} = (a_{i,j,1}, a_{i,j,2}, \ldots, a_{i,j,x})$，引数の数 $|A_{i,j}|$，戻り値の型 $r_{i,j}$，オペコード列 $O_{i,j} = (o_{i,j,1}, o_{i,j,2}, \ldots, o_{i,j,y})$ の 5 つの情報が含まれる（$m_{i,j} = \{c_{i,j}, A_{i,j}, r_{i,j}, O_{i,j}\}$）．これらの情報を復元モデルの学習データとする．

なお，引数の型や戻り値の型がユーザ定義のクラスである場合，難読化されている可能性がある．また，あまりに多くの型を対象にすると学習効率が低下するため，プリミティブ型および，`java.lang`，`java.util`，`java.io` に含まれるクラスのみを対象とする．一般的にこれらのような標準ライブラリに含まれるクラスや，プリミティブ型は，難読化の対象にならない．なお，この削除に先立ち，予備調査として Maven Central Repository [10] を対象に，型の利用頻度を調査した．その結果，上記プリミティブ型，`java.lang`，`java.util`，`java.io` に所属するクラスで全体の 91.43%を占めたため，この削除による影響は少ないと考えられる．

なお，**図 4** に学習データにおける引数の型と戻り値の型の出現頻度を示す．横軸が型を表す数値であり，その数値は，型の出現頻度を降順に並べたときの順位である．また，縦軸はその出現頻度を対数で示している．このように出現頻度には非常に大きな偏りがあることから，出現頻度の上位のみを対象とする．すなわち，引数の型の上位 n_a 件，戻り値の型の上位 n_r 件を利用するものとする．

3.2.3 学習データの変換

本フェーズでは，3.2.2 項で収集した学習データに対して，機械学習が可能なフォーマットへの変換を行う．すなわち，引数の型 A および戻り値の型 r，オペコード列 O を，出現頻度のベクトル形式に変換する．

なお，本研究の目的は，メソッドの動詞の復元である．そのため，メソッド名 c を目的変数に変換する処理もこのフェーズで行う．

変換フェーズは 4 つのステップによって構成される．

Step 1. 動詞の抽出
Step 2. 引数および戻り値の型の絞り込み
Step 3. 引数および戻り値のベクトル化
Step 4. オペコード列のベクトル化

ステップ 1 では，メソッド名 c を目的変数に変換するため，先頭にある動詞を抽出する．抽出結果となる動詞を v とし，verb(m) で得られるものとする．なお，前提として，メソッド名はキャメルケースで記述されているものとし，メソッド名を単語に分割する．次に，先頭の単語を一般的な単語辞書で検索し，検索結果の品詞に動詞が含まれていれば動詞であると判定している．たとえば，register のように名詞，動詞など複数の品詞を持つ単語の場合，動詞が含まれていれば動詞として扱う．なお，キャメルケースで記述されていない場合や，メソッド名の先頭単語に動詞が含まれない場合，学習データとして用いない．また，得られた動詞が `get` および `set` の場合も，学習データとして用いない．`get` および `set` は，一般的に，getter および setter に用いられる動詞である．getter/setter は Project Lombok [*6]で作成されるメソッドのように，単純にフィールドの値を取得/代入するメソッドとして作られることが多いと考えられる．このようなメソッドは処理が短く，命令列が短くなる．一方で，getter/setter でありながら，命令列が長いメソッドも存在する．そのようなメソッドは，getter/setter という名称が相応しいかどうかの議論が生まれる可能性がある．すなわち，getter/setter として作成されたメソッドの中に，潜在的な命名ミスや改名の余地がある可能性のあるメソッドが混在してることが考えられる．そのため，本実験では，`get` および `set` を学習データから除外した．

ステップ 2 では，本手法では用いない型を削除する．3.2.2 項で述べたとおり，本手法では学習コストの削減のため，学習に用いる型を限定する．本ステップで，プリミティブ型および，`java.lang`，`java.util`，`java.io` に含まれるクラス以外の型を削除する．

ステップ 3 では，引数の型 A および戻り値の型 r の型情報から，出現頻度を元にしたベクトル $\mathcal{A}$,$\mathcal{R}$ を次の手順で作成する．3.2.2 項で述べたように，出現頻度の上位 n_a 件，n_r 件となる型を取り出す．このときの型を $\mathcal{T}_a = (\tau_{a,1}, \tau_{a,2}, \ldots, \tau_{a,n_a})$，$\mathcal{T}_r = (\tau_{r,1}, \tau_{r,2}, \ldots, \tau_{r,n_r})$ とする．このとき，$\tau_{a,i}$ の A での出現回数を $\mathrm{count}(\tau_{a,i}, A)$ として，ベクトル $\mathcal{A} = (\mathrm{count}(\tau_{a,1}, A), \mathrm{count}(\tau_{a,2}, A), \ldots, \mathrm{count}(\tau_{a,n_a}, A))$ を得る．r，$\mathcal{T}_r$ についても同様に変換し，ベクトル $\mathcal{R} = (\mathrm{count}(\tau_{r,1}, r), \mathrm{count}(\tau_{r,2}, r), \ldots, \mathrm{count}(\tau_{r,n_r}, r))$ を得る．

たとえば仮に $n_a = 3$，$n_r = 2$ とし，$\mathcal{T}_a = (\texttt{String}, \texttt{int}, \texttt{Object})$，$\mathcal{T}_r = (\texttt{void}, \texttt{boolean})$ であるとする．このとき，$A = (\texttt{String}, \texttt{List}, \texttt{String})$，$r = \texttt{int}$ であれば，$\mathcal{A} = (2, 0, 0)$，$\mathcal{R} = (0, 0)$ となる．同様に，$r = \texttt{java.awt.Image}$ のように，ステップ 2 で削除された型の場合，$\mathcal{R} = (0, 0)$ となる．

ステップ 4 では，オペコード列 O を各オペコードの出現頻度を元に以下の手順でベクトルを導出する．Java のオペコードには，`istore` と `lstore` など，よく似た意味を持つ命令がある [11]．そのため本手法では，似た意味を持つカテゴリにオペコードを分類する．分類結果となる 36 カテゴリは**表 1** に示す．また，我々の先行研究から，オペコードの 2-gram での復元が有効であることが分かっている [12], [13]．以上のことから，各 $o_{i,j,k}$ を表 1 に従って，統合して $o'_{i,j,k}$ とし，2 つのオペコード $o'_{i,j,k}$ と $o'_{i,j,k+1}$ をまとめて，2-gram としたオペコード列 $B_{i,j} = (b_{i,j,1}, b_{i,j,2}, \ldots, b_{i,j,y-1})$ を得る．そして，$B_{i,j}$ における $b_{i,j,k}$ の出現頻度を元にベクトル $\mathcal{B}_{i,j} = ((b_{i,j,k}, \mathrm{freq}(b_{i,j,k}, B_{i,j}))|b_{i,j,k} \in B_{i,j})$ を導出する．なお，$\mathrm{freq}(b_{i,j,k}, B_{i,j})$ は $B_{i,j}$ に出現する $b_{i,j,k}$ の出現頻度を返す関数とする．

3.2.4 復元モデルの構築

本フェーズでは，学習データの変換フェーズ（3.2.3 項）で得られたデータから復元モデルをランダムフォレストを用いて構築する．メソッド情報 M のうち，v を目的変数とし，$\mathcal{A}$ および $|\mathcal{A}|$，$\mathcal{R}$，$\mathcal{B}$ を説明変数とする．構築されたモデルは，$\mathcal{A}$ および $|\mathcal{A}|$，$\mathcal{R}$，$\mathcal{B}$ を入力とすることで v を出力する．

3.2.5 対象プログラムの復元

本フェーズでは，名前難読化されたプログラムの復元を行う．難読化されたメソッドから，引数の型 A および引数の数 $|A|$，戻り値の型 r，オペコード列 O を取り出す．これらの情報を，3.2.3 項と同様の手順で，復元モデルに適用できるように変換する．変換した情報を，3.2.4 項で構築した復元モデルに入力することで，メソッド名の動詞が復元される．

3.3 変換の例

ここでは，3.2.3 項で述べた処理内容を具体的な例を示して再度説明する．変換前のメソッド M と，変換後のデータの例を**図 5** と**図 6** に示す．なお，対象となる型は $n_a = 5$，$n_r = 3$ とする．メソッド M は，Apache Cocoon Pagination プロジェクトに含まれている `Paginator` クラス[*7]のメソッドである．

まず，ステップ 1 を適用し，メソッド名 `addPaginateTags` が動詞 `add` に変換される．次に，ステップ 2 により，引数から，指定されたもの以外が削除される．すなわち `AbstractTransformer` は指定されたパッケージに属さないため削除され，引数の数が 5 となる．続いて，ステップ 3 を適用し，出現頻度の上位 n_a にある型それぞれが引数の中で何度現れたかを数え，ベクトル化する．この例では，$n_a = 5$ としたため，対象となる引数の型は $(\texttt{String}, \texttt{int}, \texttt{Object}, \texttt{Writer}, \texttt{boolean})$ となった．そのた

[*6] https://projectlombok.org

[*7] `org.apache.cocoon.transformation.pagination` パッケージ

表 1 各オペコードと集約規則
Table 1 Opcodes and unified rules.

ID	グループ名	オペコード
1	STACK	nop
2	CONSTANT	aconst_null, iconst_X, lconst_X, fconst_X, dconst_X, bipush, sipush, ldc, ldc_w, ldc2_w
3	LOAD	iload, lload, fload, dload, aload, iload_X, lload_X, fload_X, dload_X, aload_X
4	ARRAY_LOAD	iaload, laload, faload, daload, aaload, baload, caload, saload
5	STORE	istore, lstore, fstore, dstore, astore, istore_X, lstore_X, fstore_X, dstore_X, astore_X
6	ARRAY_STORE	iastore, lastore, fastore, dastore, aastore, bastore, castore, sastore
7	POP	pop, pop2
8	DUP	dup, dup_x1, dup_x2, dup2, dup2_x1, dup2_x2
9	SWAP	swap
10	ADD	iadd, ladd, fadd, dadd, iinc
11	SUBTRACT	isub, lsub, fsub, dsub
12	MULTIPLY	imul, lmul, fmul, dmul
13	DIVIDE	idiv, ldiv, fdiv, ddiv
14	REMAIN	irem, lrem, frem, drem
15	NEGATE	ineg, lneg, fneg, dneg
16	SHIFT_LEFT	ishl, lshl
17	SHIFT_RIGHT	ishr, lshr
18	USHIFT_RIGHT	iushr, lushr
19	AND	iand, land
20	OR	ior, lor
21	XOR	ixor, lxor
22	CAST	i2l, i2f, i2d, l2i, l2f, l2d, f2i, f2l, f2d, d2i, d2l, d2f, i2b, i2c, i2s
23	COMPARE	lcmp, fcmpl, fcmpg, dcmpl, dcmpg
24	BRANCH	ifeq, ifne, iflt, ifge, ifgt, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, if_acmpeq, if_acmpne, goto, goto_w, jsr, jsr_w, ifnull, ifnonnull
25	RETURN	ireturn, lreturn, freturn, dreturn, areturn, return, ret
26	SWITCH	tableswitch, lookupswitch
27	FIELD	getstatic, putstatic, getfield, putfield
28	INVOKE	invokevirtual, invokespecial, invokestatic, invokeinterface, invokedynamic
29	NEW	new
30	NEW_ARRAY	newarray, anewarray, multianewarray
31	ARRAYLENGTH	arraylength
32	CHECKCAST	checkcast
33	INSTANCEOF	instanceof
34	MONITOR	monitorenter, monitorexit
35	WIDE	wide
36	THROW	athrow

メソッド名	c	addPaginateTags
引数の数	$\|A\|$	6
引数	A	Integer[], int, int, int, String, AbstractTransformer
戻り値の型	R	void
オペコード列	O	bb, 59, b6, ..., 01, b6, b1 (295 項目)

図 5 メソッド M の情報
Fig. 5 Information of method M.

動詞	v	add
引数の数	$\|\mathcal{A}\|$	5
ベクトル化した引数	$\mathcal{A}$	1, 3, 0, 0, 0
ベクトル化した戻り値の型	$\mathcal{R}$	1, 0, 0
ベクトル化したオペコード列	$\mathcal{B}$	0, 0, ..., 30, 25, ..., 0, 0 1,296 次元（$= 36^2$）

図 6 フェーズ 2 によって変換されたメソッド M の情報
Fig. 6 Information of method M (translated by phase 2).

め，ベクトル化した結果は，$(1, 3, 0, 0, 0)$ となる．対象の型に Integer の配列が含まれないため，結果的にこれも削除される．

同様に，$n_r = 3$ となる型は，(void, boolean, String) であるため，戻り値の型もベクトル化し，$(1, 0, 0)$ を得る．最後に，ステップ 4 を適用し，オペコードの 2-gram をベクトル化する．最終的に得られるオペコードの 2-gram のベクトルは，1,296 次元（36^2）となる（意味別にまとめた 36 次元の 2-gram であるため）．

4. 実験

4.1 予備実験

難読化手法は識別子を変更する名前難読化だけではなく，様々な方法が提案されている．もしオペコード列が大きく変更されていると，提案手法によるメソッド名の動詞の復元が困難になる．そこで，提案手法の評価の前に，現実に利用される難読化ツールによりオペコード列がどの程度変更されるのかを確認する．

Stack overflow の質問と，回答から得られた記事を元に，現実に利用されているであろう難読化ツールとして ProGuard，yGuard [*8] を選択した[*9,*10]．これらのツールは，無料で利用でき，難読化およびコードの最適化も行う．そして，これらを用いて，dw-jdbc-0.4.jar [*11,*12] を難読化し，難読化前後のオペコード列を編集距離 [14] で比較した．クラス数は 78，メソッド総数は 1,536，ソースコードの総行数は 11,257 である．なお，難読化ツールの難読化レ

[*8] https://www.yworks.com/products/yguard
[*9] https://stackoverflow.com/questions/2537568/best-java-obfuscator
[*10] https://www.excelsior-usa.com/articles/java-obfuscators.html
[*11] https://mvnrepository.com/artifact/world.data/dw-jdbc/0.4
[*12] https://github.com/datadotworld/dw-jdbc

表 2 難読化前後のオペコード列の編集距離
Table 2 The edit distances of the opcode sequences between before and after obfuscation.

Range of ED	ProGuard	yGuard
0	82.61%	90.29%
1	90.71%	96.89%
2	92.93%	98.57%
3	93.56%	98.65%
4	94.32%	98.73%
5	95.36%	99.20%
6–10	96.74%	99.60%
11–15	97.64%	99.84%
16–20	98.20%	99.84%
21–30	98.75%	99.84%
31–40	99.10%	99.84%
41–50	99.24%	99.84%
51–70	99.58%	99.92%
71–90	99.72%	100.00%
91–110	99.86%	100.00%
111–150	100.00%	100.00%

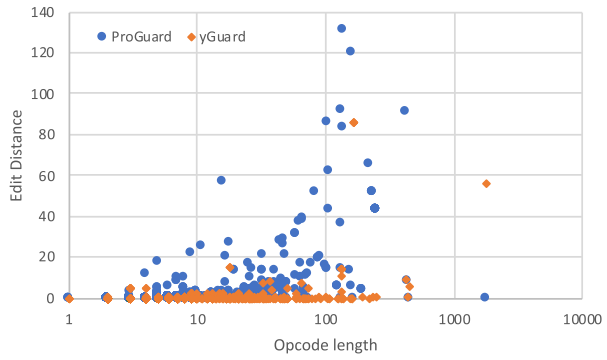


図 7 難読化前後でのオペコード列の変化の散布図
Fig. 7 Differences of the opcode sequences before/after obfuscation.

ベル，最適化レベルはそれぞれデフォルトを適用した．

予備実験の結果を**表 2** に示す．Range of ED は編集距離の範囲を表す．そして，ProGuard，yGuard 欄は，対応する編集距離以下であったメソッドの割合を示す．表 2 を見ると，ProGuard，yGuard ともに，95%以上のメソッドが編集距離 5 以下であることが分かる．また，**図 7** に難読化前後でのオペコード列の変化を散布図で示す．横軸は難読化前のオペコード列長を対数で表し，縦軸は対応するメソッドの難読化前後でのオペコード列の編集距離である．丸型のプロットが ProGuard での難読化結果を表し，ひし形のプロットが yGuard での難読化結果を表す．図 7 を見ると，ProGuard において，難読化前のオペコード列長が 100 を超えたものから編集距離が大きくなっている．しかし，編集距離が 20 以下のものが 98%であることが表 2 から確認できる．一方，yGuard はいくつかのメソッドは編集距離が 40 を超えるものの，ほとんどのメソッドで編集距離は 20 以下であることが図 7 からも確認できる．以上のことから，少なくとも，ProGuard と yGuard では提案手法への影響が少ないと考えられる．

4.2 学習データの収集

この実験では，Java プログラムを対象とし，Maven Central Repository から学習データを取得した．Maven Central Repository に含まれる全プロジェクトの最新バージョンを取り出し，17,714 個の jar ファイルを得た．すべての jar ファイルからメソッドを取り出し，21,738,029 個のメソッドを得た．この中から，極端に処理の短いメソッドおよび長いメソッドを，オペコード列の長さを基準に学習データから除く．短いメソッドは，単純な処理が多く，復元に必要な情報が含まれないためである．また，長いメソッドは類似するメソッドがごく少数となり，機械学習に適さないためである．そこで，コード行数が 10 行から 300 行になるようなオペコード長で実験を行う．その結果，オペコード長は $[30, 1000)$ となるもののみとする．動詞の復元を目的として考えたときの，意味のあるメソッド行数を元にこのオペコード長を用いた．しかし，場合によっては，より範囲を狭めての実験が有効な場合もあろう．また，`main` メソッドおよび `static` イニシャライザのメソッド名は自明であり，コンストラクタはクラス名に依存するため学習データから除く．これらのフィルタリングにより，2,896,071 個のメソッドを得た．

得られたメソッドからメソッド情報を取り出し，3.2.3 項のとおり，学習データに適用可能なフォーマットへデータ変換する．本実験では，動詞の復元を目的とするため，メソッド名の先頭に動詞を持たないメソッドは学習データとして用いない．ただし，一般的にプログラム中で動詞と同様な扱いとされる `new` および `setup`，`cleanup`，`to`，`as` は本論文では動詞として扱う [15], [16]．また，`calc` や `init` などの短縮系は元の動詞と同じ扱いとし，`analyse` と `analyze` のような表記揺れのあるものは統一する．なお，これら短縮系の判定ならびに元の動詞への変換や，表記揺れの統一は，人手で行った．2,896,071 個のメソッドのうち，メソッド名の先頭に動詞のあったメソッドは 2,404,277 個であった．なお，全引数の使用頻度の 90%以上になるよう調整し，$n_a = 20$，$n_r = 6$ とした．

4.3 復元モデルの構築

4.2 節で取得，変換した学習データをもとに復元モデルの作成を行う．復元モデルの作成には，Python 2.7.10 の scikit-learn [*13] を用いた．実験に用いた PC は MacBook Pro '15，macOS High Sierra（10.13.5），Intel Core i5 2.7 GHz，16 GB RAM である．

ただし，学習データに現れる動詞の数は，1,813 個と非

[*13] http://scikitlearn.org/stable/

表 3 復元対象のプログラム
Table 3 The test program of the experiments.

ファイル名	データサイズ
BTSync-Java-0.1.jar	5.2 MB
DynamicJasper-core-fonts-1.0.jar	2.9 MB
acm-2.0.0-preview-5.jar	251 KB
amqp-scala-client_2.12-2.0.0.jar	555 KB
api-doc-0.0.34.jar	554 KB
bitcoinj-core-0.15-cm04.jar	1.5 MB
codedeploy-notifications_2.11-0.2.1.jar	208 KB
dw-jdbc-0.4.jar	167 KB
elasticsearch-5.0.0-beta1.jar	8.9 MB
flink-kudu-connector-1.0.jar	34 MB
geopackage-core-1.3.1.jar	314 KB
itk-payloads-0.5.jar	1.8 MB
ixa-pipe-chunk-1.1.0.jar	5.3 MB
ixa-pipe-parse-1.1.1.jar	59 MB
mlapi_2.12-0.0.1.jar	367 KB
monetdb-java-lite-2.33.jar	6.4 MB
no-exceptions_2.11-1.0.1.jar	173 KB
openfin-desktop-java-adapter-6.0.1.0.jar	310 KB
orbit-runtime-1.1.0.jar	1.9 MB
payara-microprofile-1.0-4.1.2.172.jar	37 MB
phtree-0.3.1.jar	336 KB
scala-expect_2.12-6.0.0.jar	199 KB
scenery-0.2.2.jar	1.5 MB
schemaspy-maven-plugin-1.2.1.jar	280 KB
semanticvectors-5.8.jar	12 MB
siren-join-2.4.5.jar	210 KB
uaiMockServer-1.2.5.jar	579 KB
vldocking-3.0.4.jar	392 KB

常に多く，すべてのデータを用いた学習はメモリ不足のためできなかった．また，一般的にプログラムに用いられる動詞は限られていることから，使用頻度の高い動詞のみを対象とし，復元モデルの作成を行った．使用頻度上位 20 件から上位 200 件まで，10 件ずつ増加させ，19 個の復元モデル（$\mathcal{L} = (L_1, \ldots, L_{19})$）を作成した．つまり，$L_i$ の復元モデルには使用頻度上位 $10 + 10i$（$1 \leq i \leq 19$）件の動詞が使用されている．

4.4 テストデータ

復元対象のプログラムを**表 3** に示す．これらは Maven Sonartype Repository から無作為に取得し，学習データに含まれないデータとした．これらのデータは名前難読化されていないため，提案手法により復元された動詞と，元のメソッドが持つ動詞を比較できる．この比較により，本手法の有効性を検証する．また，動詞の比較のため，テストデータは動詞から始まる名前を持つメソッドとし，オペコード長が $[30, 1000)$ のメソッドとした．テストデータは 28 個のプログラムであり，復元対象となるメソッドは 38,685 個となった．

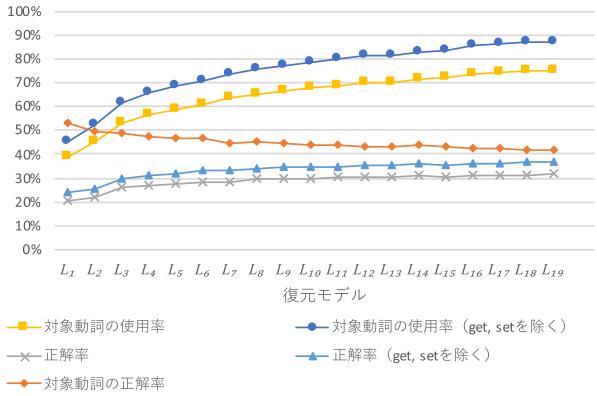


図 8 モデルごとの正解率と対象動詞の使用率
Fig. 8 The correct rates and the usage rates of the target verbs by each model.

4.5 評価指標

提案手法の評価には，正解率，対象動詞の正解率，対象動詞の使用率を用いる．まず，対象動詞を $V_k = \{v_1, v_2, \ldots, v_k\}$ と表す．k が対象動詞の数を表す．次に，テストデータから取り出したメソッド一覧を $\mathcal{M} = \{m_1, m_2, \ldots, m_x\}$ としたとき，メソッド名に対象動詞を含むメソッドの集合を $\mathcal{M}_V = \{m_i | \mathrm{verb}(m_i) \in V_k \wedge m_i \in \mathcal{M}\}$ とする．また，提案手法により復元された動詞を $v' = \mathrm{restore}(m)$ とする．そして，提案手法により，復元に成功したメソッドを $\mathcal{M}_S = \{m_i | \mathrm{match}(\mathrm{verb}(m_i), \mathrm{restore}(m_i)) = \mathit{true} \wedge m_i \in \mathcal{M}\}$ とする．なお，$\mathrm{match}(v_i, v_j)$ 関数は与えられた 2 つの動詞が一致するかを判定する関数であり，同じ動詞であれば *true*，そうでなければ *false* を返すものとする．具体的なアルゴリズムは与えられるものとする．

ここで，正解率は，$\frac{|\mathcal{M}_S|}{|\mathcal{M}|}$，対象動詞の正解率は，$\frac{|\mathcal{M}_S|}{|\mathcal{M}_V|}$ とする．そして，対象動詞の使用率は，テストデータのメソッド一覧（$\mathcal{M}$）のうち，メソッド名に対象動詞（V_k）を使用しているメソッド（$\mathcal{M}_V$）の割合を表す．すなわち，対象動詞の使用率を $\frac{|\mathcal{M}_V|}{|\mathcal{M}|}$ で表す．

4.6 対象プログラムの復元

4.3 節で作成した復元モデルを用いて，実際に復元を行う．なお，復元結果の評価である $\mathrm{match}(v_i, v_j)$ 関数は $v_i = v_j$ の場合に *true* となるものとする．また，評価指標は，正解率，対象動詞の使用率に加えて，`get`，`set` を除いた正解率，対象動詞の使用率についても算出している．3.2.3 項で述べたように，`get`，`set` は学習データから除外している．一方で，テストデータからは除外していない．つまり，テストデータに含まれる `get`，`set` から始まるメソッド名は必ず復元に失敗し，正解率を下げることになる．そのため，これらのメソッドを除外した評価も行う．

4.3 節で作成した各復元モデルでの復元結果を**図 8** に示す．縦軸は各折れ線が示すデータが全体に含まれる割合，横軸は利用した復元モデルである．図 8 では，評価指標で

ある正解率，対象動詞の正解率，対象動詞の使用率に加え，get，set を除いた正解率，対象動詞の使用率の 5 つを折れ線で示している．

図 8 を見ると，L_i の i が増加するにつれて，すなわち，復元対象の動詞が増加するにつれて，全体の正解率は上がる傾向にあることが分かる．一方で，対象動詞の正解率は下がる傾向にあることが分かる．また，復元モデル L_{11}（120 動詞）を利用すると，メソッド名に用いられている動詞の約 30%を復元することができた．このときの対象動詞の使用率は 68.67%である．さらに，復元対象をモデル作成に用いた動詞に限ると，すべてのモデルにおいて 40%以上のデータが正しく復元された．最も高い正解率は 20 動詞を用いた復元モデルで 53.16%，最も低いものは 190 動詞で 42.02%であった．

前述のとおり，復元モデルには get および set は含まれていないため，これらが復元されることはない．図 8 を見ると，get，set を復元対象としない場合，L_{12}（130 動詞）以上で対象動詞の使用率は 80%以上であることが分かる．一方で，L_1（20 動詞）では，対象動詞の使用率は 50%を下回っており，適用できるデータは少ない．しかしながら，対象動詞を持つデータの正解率は，すべてのモデルの中で最も高くなった．また，L_{11}（120 動詞）では，対象動詞の使用率は 69.85%であり，全体の 34.91%の復元に成功した．

4.7 動詞の類似度に基づいた評価

プログラム中に使われる動詞には，用途の似たものがある．たとえば compute と calculate や，execute と perform などがあげられる．このような場合，どちらが復元されても本質的な意味は大きく変わらないと考えられる．そのため，動詞の意味に着目した評価も実施する．復元された動詞と元の動詞の意味的な類似度を元に評価する．評価には，英単語の意味辞書である WordNet [*14] を用いた．また，類似度の計算には，path_similarity を用いた [17]．なお，正解率を計算する $\mathrm{match}(v_i, v_j)$ は，$\mathrm{match}_{\mathrm{sim}}(v_i, v_j) \geq \varepsilon$ の場合に *true* となるものとする．そして，$\mathrm{match}_{\mathrm{sim}}(v_i, v_j)$ は 2 つの動詞の path_similarity を返すものとする．ここでは，ε が，1，$\frac{1}{2}$，$\frac{1}{3}$ の場合で正解率を算出する．

WordNet を用いた評価を**図 9** に示す．2 動詞が同義語となる $\varepsilon = 1$ の場合，60 動詞以上を用いることで，全体の正解率が 30%以上となった．最も高い正解率は 200 動詞で 33.94%であった．また，get および set を復元対象としない場合，最高で 39.37%の復元に成功した．2 動詞が直接の上位語の関係となる $\varepsilon = \frac{1}{2}$ とした場合，全体の正解率が最大で 40.07%となった．get および set を復元対象としない場合，最高で 59.60%と 60%近くの動詞の復元に成功した．$\varepsilon = \frac{1}{3}$ とした場合，全体で 60%，get および set を除く場合で 70%を超える正解率となった．しかし，$\varepsilon = \frac{1}{3}$ は同義語，上位語の関係に加え，上位語を介した別の下位語や上位語のさらに上位語の関係が含まれる．**表 4** に，$\varepsilon = \frac{1}{3}$ となった復元例を示す．表 4 では，start と run や，translate と convert のように，比較的意味の似た動詞も $\varepsilon = \frac{1}{3}$ となった．一方で，read と write や，accept と remove のように，意味が正反対の動詞や，まったく意味の異なる動詞も正解に含まれている．そのため，直感に合わない動詞であっても正解となる場合があり，この指標の評価も別途必要となる．

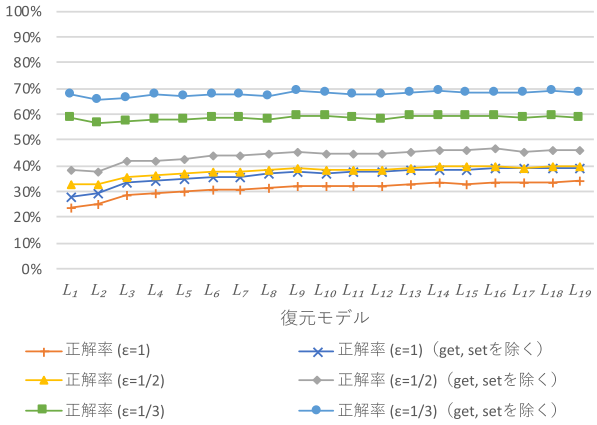


図 9 WordNet を用いた評価結果
Fig. 9 The correct rates by using WordNet.

表 4 提案手法による復元の例
Table 4 Examples of the restration by the proposed method.

正解動詞	復元された動詞
start	run
translate	convert
read	write
accept	remove

これらの結果から，本手法は，約 30%の割合でメソッド名の動詞を元に戻すことができることが分かる．また，get および set を復元対象としない場合，約 35%の割合で元に戻すことができた．さらに，同義語の動詞であれば，約 40%の割合で復元できた．上位語の関係にある動詞の復元は，約 60%の割合で成功した．このことから，名前難読化の耐タンパ化性能には一定の脆弱性があるといえる．

4.8 議論

提案手法では，全体の 31.62%の動詞の復元に成功し，同義語では 33.94%，上位語では 40.07%のメソッド名の動詞を復元できたことを前節までに示した．この結果は決して高い復元率であるとはいえない．また，復元できるものもメソッド名の動詞のみと限定されている．しかし，それでも名前難読化での保護が安全でない可能性があることを示した点において意味があるといえよう．

1 章や 2.2 節にも述べているように，一般的な名前難読

[*14] https://wordnet.princeton.edu

化は実装が容易であり，変更箇所も分かりやすいため，多くの難読化ツールで採用されている．しかし本論文では，名前をいくら分かりにくいものに変更したとしても，最高で 40%程度は動詞を復元できる可能性を示した．つまり，名前難読化単体での保護は，破られる可能性があることを示したといえる．そのため，ソフトウェアの保護を考えるとき，名前難読化だけではなくコントロールフローの変更 [18], [19] やデータ難読化 [20] も併用する必要があろう．本研究での評価では，オペコード列は難読化されていないという前提であるため，その前提を崩せば提案手法が無効化できる可能性があるためである．ただし，4.1 節で示したように，ツールによってはオペコード列を大きく変更させないものもある．そのため，本当に保護できているか，少なくともオペコード列の変更が行われているかの確認が必要であるといえる．

5. 妥当性への脅威

5.1 内的妥当性への脅威

5.1.1 オペコード長の制限

本実験では，メソッドのオペコード長に制限を設けた．対象としたオペコード長は $[30, 1000)$ としたが，この制限の範囲の妥当性を十分に裏付けるための根拠は乏しい．オペコード長が長くなるにつれ，サンプル数が少なくなるため，復元のために十分なデータが確保されていない可能性がある．また，メソッド行数をおおむね 10 行程度を下限値とした．しかし，さらに下限値を低くすることで，対象となるデータが増える．一方で，まとまった意味を持つメソッドでなければ，復元は難しい．そのため，オペコード長の制限について議論が今後の課題となる．

5.1.2 対象とした型

本実験では，学習効率の向上のため，プリミティブ型および，`java.lang`，`java.util`，`java.io` に含まれるクラスのみを対象とした．今回対象とした型は，引数，戻り値としての使用頻度が高く，全データの 91.43%を占めていることから，復元結果に大きな影響はないと考えられる．しかし，対象とする型を絞った結果，対象外となった型のみを持つメソッドは，引数の数が 0 となる．本実験では，これらの影響に関して，十分な検証を行っていない．そのため，対象とする型を絞ることによる，復元への影響を検証する必要がある．

5.1.3 `get`, `set` の扱い

本論文では，`get` および `set` を対象外とし復元モデルを作成した．`get` および `set` は，一般的に getter/setter メソッドに用いられる動詞であり，単純な処理かつ，多数定義されている．なお，テストデータにおける setter/getter は 72,231 メソッドであり，メソッド全体の 29.04%（72,231/248,712 メソッド）を占める．実験で対象にしたメソッド，すなわちオペコード長が $(30, 1000]$ に限ったとしても，全体の 13.80%（5,340/38,685 メソッド）を占める．

しかし，学習データやテストデータには，単なるフィールド変数の取得，代入以外に `get` および `set` を使う例があった．そのため，今後，`get` および `set` を対象外とすることに対する妥当性の議論が必要となる．

5.1.4 復元による難読化への影響

本研究では，約 30〜60%の動詞の復元に成功した．名前難読化の耐タンパ化性能には一定の脆弱性があることが分かったが，この復元結果が名前難読化にどの程度影響を及ぼすのか十分な議論をしていない．そのため，今後，実際に名前難読化されたプログラムを用いて，復元されたプログラムを検証し復元の効果を定量的に示す必要がある．

5.2 外的妥当性への脅威

5.2.1 学習データとテストデータ

本実験では，Maven Repository から学習データおよびテストデータを取得した．その際，プロジェクトのドメインなどを考慮していない．ドメインによって，メソッドの処理に対する動詞の使われ方が異なる場合がある．そのため，ドメインごとにモデルを作成することで，復元結果が大きく異なる可能性がある．

また，プロジェクトによっては，多数のオーバライドやオーバロードが行われている場合がある．このような場合，復元結果がプロジェクトに依存してしまう可能性がある．さらに，学習データおよびテストデータの名前の正しさについての検証は行っていない．つまり，メソッドの名前と内容の適合性は検証していない．これらを精査し，対象メソッドを除外することで，復元結果に影響が出る可能性もある．

5.2.2 類似度による評価

動詞間の類似度を測るために WordNet を用いた．しかし，WordNet は一般的な意味での単語間の類似度を算出する．一方で，プログラム中に用いられる動詞は，複数の意味で使われることは少ない．また，一般的な意味とは異なる意味で用いられる場合もある．そのため，WordNet による評価の妥当性を検証する必要がある．今後，プログラムで用いられる意味に特化した意味辞書などを作成し，評価に用いることで解決できると考えられる．

5.2.3 対象の難読化手法

本研究では，名前難読化では難読化されない情報を用いて復元を行った．しかし，難読化には様々な手法がある．なかには，オペコード列を大きく変更する難読化手法も存在する．加えて，現実問題として，単一の難読化手法（名前難読化）で保護されているとは限らない．保護手法を幾重にも適用し，厳重に保護する可能性もある．そのため，本手法による復元が難しくなる可能性がある．4.1 節にて，予備実験として ProGuard と yGuard は，95%のメソッドでオペコードの変更量が僅かであることを確認した．しか

し，他の難読化ツールについても今後調査が必要である．

一方で，提案手法の一部では 2-gram を用いてモデルを構築している．オペコードの 2-gram は k-gram バースマークとして提案されている [21]．バースマークはソフトウェアの盗用を検出するための手法であり，難読化や最適化などの攻撃に対して一定の耐性を持つと確認されている．このことから，提案手法も多くの場合に有効であると期待できる．しかし，上にも述べたように，具体的にどの難読化手法に対して有効であるかの調査も今後必要である．

6. 関連研究

名前難読化以外の難読化手法についても，逆変換や評価に取り組まれている．たとえば，Udupa らは，動的解析，静的解析を組み合わせて Basic block flattering 難読化（コントロールフロー難読化の 1 つ）の逆変換を試みている [22]．データフロー難読化についても，メトリクスが定義され，難読化自体の評価が行われている [23]．また，Kanzaki らは，命令列の不自然さ（Artificiality）からプログラムのステルス性（Stealthiness）を測定することで，難読化の評価を狙っている [24]．一方，名前難読化は Cimato らが逆変換手法を提案しているものの [25]，対象はフィールドであり，動作を理解するためのメソッド名は対象外となっている．

一方で，名前に着目したプログラムの分析が行われている．たとえば，メソッド名の動詞を推薦する手法が提案されている [15]．名前推薦の手がかりとして，他の名前（クラス名，メソッド名）を利用している．また，メソッド名とメソッドの内容の適合性を確認する手法が提案されている [26]．これらの対象はソースコードであり，難読化の逆変換を目的とする場合には，そのまま適用が難しい．

7. まとめと今後の課題

本研究では，名前難読化されたプログラムに対して動詞の復元を行った．名前難読化では難読化されないオペコードや引数の型，戻り値の型を復元のための情報とした．これらの情報を機械学習させ，復元モデルを作成した．

本手法によって，31.62%のメソッドに対して正しい動詞の復元に成功した．復元モデルに用いた動詞に限ると，40%以上のメソッドを復元できた．WordNet による動詞の意味的な類似度に基づいた評価では，同義語（$\varepsilon = 1$）であれば，33.94%のメソッドの動詞が復元に成功した．$\varepsilon = \frac{1}{2}$ とした場合，40.07%のメソッド名の動詞を復元できた．すなわち，名前難読化手法の耐タンパ化性能には一定の脆弱性があるといえる．そのため，ソフトウェア内部に存在する保護したい情報は，名前難読化以外の保護方法も併用し，保護する必要がある．

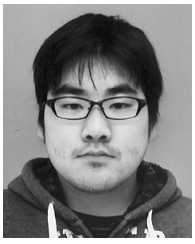
本実験では，学習データに含まれるすべての動詞を用いた復元モデルの作成ができなかった．一方で，対象動詞を増やすことで復元結果が向上する結果となった．そのため，今後さらに対象動詞を増加させた実験が必要となる．そして，名前難読化手法に加えて，他の難読化手法が適用された場合の評価についても今後評価が必要である．加えて，どの難読化手法が安全であるのかを確認するためにも，他の難読化手法についても耐タンパ化性能の評価が必要である．

謝辞 本研究の一部は JSPS 科研費 17K00196，17K00500，17H00731 の助成を受けたものです．

参考文献

[1] Collberg, C. and Nagra, J.: *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, Addison-Wesley (2009).
[2] Aucsmith, D.: Tamper resistant software: An implementation, *Proc. 1st International Workshop on Information Hiding*, LNCS, Vol.1174, pp.317–333 (1996).
[3] Tamada, H., Nakamura, M., Monden, A. and Matsumoto, K.: Introducing Dynamic Name Resolution Mechanism for Obfuscating System-Defined Names in Programs, *Proc. IASTED International Conference on Software Engineering* (*IASTED SE 2008*), pp.125–130 (2008).
[4] Qing, S., Zhi-yue, W., Wei-min, W., Jing-liang, L. and Zhi-wei, H.: Technique of Source Code Obfuscation Based on Data Flow and Control Flow Tansformations, *Proc. 7th International Conference on Computer Science & Education* (*ICCSE 2012*) (2012).
[5] Fukuda, K. and Tamada, H.: To Prevent Reverse-Enginnering Tools by Shuffling the Stack Status with Hook Mechanism, *International Journal of Software Innovation* (*IJSI*), Vol.3, pp.14–25 (2015).
[6] Høst, E.W. and Østvold, B.M.: The Programmer's Lexicon, Volume I: The Verbs, *Proc. 7th IEEE International Working Conference on Source Code Analysis and Manipulation* (*SCAM 2007*), pp.193–202 (2007).
[7] Tyma, P.M.: Method for renaming identifiers of a computer program, United States Patent 6,102,966 (2000).
[8] 玉田春昭，中村匡秀，門田暁人，松本健一：API ライブラリ名隠ぺいのための動的名前解決を用いた名前難読化，電子情報通信学会論文誌，Vol.J90-D, No.10, pp.2723–2735 (2007).
[9] Chaudhary, B.D. and Sahasrabuddhe, H.V.: Meaningfulness as a factor of program complexity, *Proc. ACM 1980 Annual Conference*, pp.457–466 (1980).
[10] Raemaekers, S., Deursen, A. and Visser, J.: The Maven repository dataset of metrics, changes, and dependencies, *Proc. 2013 10th IEEE Working Conference on Mining Software Repositories* (*MSR 2013*), pp.221–224 (2013).
[11] Lindholm, T., Yellin, F., Bracha, G. and Buckley, A.: *The Java Virtual Machine Specification*, Oracle, Inc., Java SE 7 edition (2013).
[12] 磯部陽介，玉田春昭：ランダムフォレストによる名前難読化の逆変換，ソフトウェア工学の基礎 XXIV，pp.93–98 (2017).
[13] Isobe, Y. and Tamada, H.: Are Identifier Renaming Methods Secure?—An Evaluation Focuses on Opcodes using Random Forest, *19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing* (*IEEE/ACIS SNPD 2018*) (2018).

[14] Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals, *Soviet Physics Doklady*, Vol.10, No.8, pp.707–710 (1966) (online), available from ⟨http://sascha.geekheim.de/wp-content/uploads/2006/04/levenshtein.pdf⟩.
[15] Kashiwabara, Y., Onizuka, Y., Ishio, T., Hayase, Y., Yamamoto, T. and Inoue, K.: Recommending verbs for rename method using association rule mining, *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering* (*CSMR-WCRE*), pp.323–327 (2014).
[16] Kashiwabara, Y., Ishio, T., Hata, H. and Inoue, K.: Method Verb Recommendation Using Association Rule Mining in a Set of Existing Projects, *IEICE Trans. Information and Systems*, Vol.E98-D, No.3, pp.627–636 (2015).
[17] Pedersen, T., Patwardhan, S. and Michelizzi, J.: WordNet::Similarity: Measuring the relatedness of concepts, *Proc. HLT-NAACL-Demonstrations 2004*, pp.38–41 (2004).
[18] Hou, T., Chen, H. and Tsai, M.: Three control flow obfuscation methods for Java software, *IEE Proc. Software*, Vol.153, No.2, pp.80–86 (2006).
[19] László, T. and Kiss, A.: Obfuscating C++ programs via control flow flattening, *Proc. 10th Symposium on Programming Languages and Software Tools* (*SPLST 2007*), pp.15–29 (2007).
[20] Collberg, C., Thomborson, C. and Low, D.: Breaking Abstractions and Unstructuring Data Structures, *Proc. 1998 International Conference on Computer Languages* (*ICCL 1998*) (1998).
[21] Myles, G. and Collberg, C.: K-gram based software birthmarks, *Proc. 20th Annual ACM Symp. Applied Computing*, pp.314–318 (2005).
[22] Udupa, S.K., Debray, S.K. and Madou, M.: Deobfuscation: Reverse engineering obfuscated code, *Proc. 12th Working Conference on Reverse Engineering* (2005).
[23] Demissie, B.F., Ceccato, M. and Tiella, R.: Assessment of Data Obfuscation with Residue Number Coding, *1st International Workshop on Software Protection* (*SPRO2015*), pp.38–44 (2015).
[24] Kanzaki, Y., Monden, A. and Collberg, C.: Code Artificiality: A Metric for the Code Stealth Based on an N-gram Model, *1st International Workshop on Software Protection* (*SPRO2015*), pp.31–37 (2015).
[25] Cimato, S., Santis, A.D. and Petrillo, U.F.: Overcoming the obfuscation of Java programs by identifier renaming, *Journal of Systems and Software*, Vol.78, No.1, pp.60–72 (2005).
[26] 鈴木 翔，阿萬裕久，川原 稔：決定木を用いた Java メソッドの名前と実装の適合性評価法の提案，ソフトウェア工学の基礎 XXIV，pp.63–72 (2017).



磯部 陽介

2017 年京都産業大学コンピュータ理工学部卒業．同大学大学院先端情報学博士前期課程在学中．

玉田 春昭 （正会員）

2006 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．同年同大学産学官連携研究員．2007 年同大学情報科学研究科特任助教．2008 年京都産業大学コンピュータ理工学部助教．2013 年同大学コンピュータ理工学部准教授．2018 年同大学情報理工学部准教授．ソフトウェアセキュリティ，エンピリカルソフトウェア工学の研究に従事．IEEE，IEICE 各会員．