SCIS 2017 2017 Symposium on
Cryptography and Information Security
Naha, Japan, Jan. 24 - 27, 2017
The Institute of Electronics,
Information and Communication Engineers

使用クラスに基づいた *tf-idf* による難読化の不自然さ評価
Artificiality Evaluation of Obfuscation Methods based on Used Classes

横井 昂典 * 玉田 春昭 *
Takanori Yokoi Haruaki Tamada

あらまし プログラムの保護手法に難読化が提案されている．難読化とは，プログラムを意図的に分かりにくくする手法である．しかし，難読化を行うことにより，プログラムに不自然な特徴が追加されることがある．そして，追加された不自然な特徴により，プログラムが難読化されている事実が明らかになってしまう恐れがある．プログラムを不正に利用しようとする者に難読化されている事実が明らかになると，プログラムを解析する手がかりを与えてしまう．そこで，本稿では，難読化されたプログラムの特徴の不自然さ（artificiality）の定量的な評価尺度を提案する．プログラムにおけるクラスは，他のクラスの機能を利用することで実装される場合が多い．そのため，クラスが利用している他のクラスの情報を用いることで，当該クラスの不自然さが求められる．その不自然さを数値化するために，利用しているクラスの情報を用いて *tf-idf* を求める．そして，難読化されたプログラムと元のプログラムで *tf-idf* の比較を行うことで，不自然さを評価する．本稿で行った実験では，プログラムに難読化手法を適用し，難読化前後で不自然さを比較した．その中の一つの結果では，不自然さが難読化前後で 1.77 から 2.37 になり，難読化後のプログラムの不自然さが増加した．

キーワード 不自然さ評価，使用 API

1 はじめに

近年，様々な分野でソフトウェアの需要が高まっている．しかしそれと同時に，ソフトウェアを不正に利用しようとするものが増え，ソフトウェアのセキュリティの重要度も高まっている．そこで，ソフトウェアを保護するための手法として，数々のソフトウェア保護手法が提案されている．その中の一つに難読化が存在する．難読化とは，プログラムの外部的な振る舞いは変更せずに，内部処理を意図的にわかりにくくする手法である．ソフトウェア開発者の視点からすると，難読化などの保護手法が，どれだけソフトウェアを保護出来ているのかを把握することは重要である．ソフトウェアの保護の強さの度合いを知るために，保護されたソフトウェアへの攻撃のコストの高さを測定する必要がある．

Collberg らによると，ソフトウェアへの攻撃は (1) 保護機構の発見，(2) 保護機構の理解と改ざん，(3) 動作テストの 3 ステップで行われる [1]．また，保護機構さえ見つけてしまえば，攻撃は容易になるとも述べている．本稿では，３ステップのうち，(1) 保護機構の発見に着目し，ソフトウェアへの攻撃のコストを保護機構の発見の困難さによって測定する．そして，保護機構の発見の困難さは，難読化を適用することにより元のプログラムと比べてどれほど不自然であるかによって求める．不自然さを測定するために，本稿では，プログラムの利用クラスに着目する．

近年は，非常に高機能な様々なプログラム部品が標準的に提供されている．そして，それらの部品を使ってプログラムが作成される．例えば，Java 言語では標準ライブラリに，数多くのクラスが用意されており，様々な機能が標準的に提供されている．そのため，プログラムが利用しているライブラリの情報を用いることで，当該プログラムの不自然さが求められる．そして，利用しているライブラリの情報を用いて *tf-idf* を求めることにより，プログラムの不自然さを数値化する．*tf-idf* の値が高いほど，他のプログラムより目立つことになり不自然と言える．そして，難読化されたプログラムと元のプログラムで *tf-idf* の比較を行う．この比較によって，プログラムが難読化を適用することでどれほど不自然になったかを評価する．

以降，第 2 節は，本稿の関連研究や定義を説明する．第 3 節では，提案手法について記述する．第 4 節では，

* 京都産業大学コンピュータ理工学部，京都市北区上賀茂本山，Motoyama, Kamigamo, Kita-ku, Kyoto-shi, Kyoto, Japan.

実験の準備について記述する．第 5 節は，既存の保護手法が適用されたプログラムの不自然さを提案手法によって評価するケーススタディを記す．最後に第 6 節で，本稿をまとめ，今後の課題について述べる．

2 準備

2.1 コードの不自然さ評価

神崎らは，n-gram モデルを利用してコードの不自然さの評価手法を提案している [2]．n-gram は，既存の多くのソフトウェアのアセンブリコードを元にしたコーパスから構築される．提案モデルに基づいて，ソフトウェア保護のために追加・変形されたコードの生成確率を求めている．すなわち，当該部分のもっともらしさを評価している．もし，もっともらしさが低い場合，一般的なコンパイラが作成するコードではないため，不自然であることになる．

大滝らは，神崎らの手法を拡張して，オブジェクト指向言語を対象とした不自然さの評価手法を提案している [3]．オペコードおよびオペランドを言語体系とみなして，確率的言語モデルを用いて不自然さを評価した．大滝らは，Java 言語を対象に提案手法によって不自然さ評価を行い，神崎らの手法では測定が難しい難読化手法の不自然さを敏感に測定できることを示している．つまり，プログラムの不自然さにも様々な測定方法があり，難読化手法によって測定できるもの，測定できないものがあることを示している．実際に，神崎らの手法，大滝らの手法の両方で測定できない難読化手法も存在する．

そこで本稿では，神崎ら，大滝らの手法とは異なる手法でコードの不自然さを測定する方法を提案する．具体的には，コード中のライブラリ名に着目した不自然さの評価指標を構築する．なお，本稿での対象言語は Java としているが，一般的なオブジェクト指向言語にも適用可能である．

2.2 利用クラス

今日，アプリケーションを作成するときには，既に存在するライブラリを利用することが当たり前になっている．本稿では，プログラムが利用しているライブラリの名前を用いて不自然さを測定する．例えば，Java 言語では標準ライブラリに，数多くのクラスが用意されており，様々な機能が標準的に提供されている．あるプログラムが利用している標準ライブラリのクラス名を，そのプログラムの不自然さを測定する手掛かりとする．この標準ライブラリのクラス名を，以降**利用クラス**（UC; Used Classes）と呼ぶ．なお，標準ライブラリの範囲は適宜与えられるものとする．

例としてファイルを読み込み標準出力に出力する Java のプログラム `Cat` を作成した．ソースコードを図 1 に示す．

```
import java.io.*;

public class Cat {
  public void run(String[] args) throws IOException{
    for(String arg: args){
      cat(arg);
    }
  }
  private void cat(String file) throws IOException{
    try(BufferedReader br =
          new BufferedReader(new FileReader(file))){
      String line;
      while((line = br.readLine()) != null){
        System.out.println(line);
      }
    }
  }
  public static void main(String[] args)
          throws IOException{
    Cat cat = new Cat();
    cat.run(args);
  }
}
```

図 1: サンプルプログラム（`Cat.java`）

図 1 から抽出できる利用クラスは次の通りである．ここでは，標準ライブラリを Java SE 8[1]に属するクラス，すなわちパッケージ名が `java` もしくは `javax` で始まるものとした[2]．

- `java.io.FileReader`
- `java.io.BufferedReader`
- `java.io.Reader`
- `java.io.PrintStream`
- `java.lang.System`
- `java.lang.Object`
- `java.lang.String`

上記の利用クラスの中には図 1 に示す `Cat` のソースコード中に明示的に現れないクラスも利用クラスとして挙げている．この不自然さ評価は，難読化されたプログラムを主な対象とする評価方法である．難読化されたプログラムは，基本的にソースコードが手に入らない．そのため，バイナリを分析の対象とする．バイナリでは，ソースコード中に明示的に現れないクラスであっても出現する場合がある．例えば，`java.io.PrintStream` は `System.out` の型であり，`println` メソッドの呼び出し時に利用されている．また，利用クラスの中には同じクラス内で複数回出現するものも存在するが，本稿では一つにまとめる．

2.3 *tf-idf*

不自然さを測定するために，各クラスから抽出した利用クラスを用いて何らかの指標を導出しなければならない．そのために，本稿では *tf-idf* を利用する．

[1] `http://www.oracle.com/technetwork/java/javase`

[2] もちろん，`org.omg.CORBA` や `org.w3c.dom`，`org.xml.sax` なども Java SE 8 に含まれるが，今回は簡単化のため本文中のように `java` もしくは `javax` から始まるもののみとした．

tf-idf とは，文書における単語の重みの一種であり，文書の中でどの単語が重要であるかを数値として示す指標である．文書は文字列から成る単語を要素として持つ．本稿では，プログラムを文書，各プログラムから抽出された利用クラスを単語として扱う．なお，プログラムはクラスファイル単位やパッケージ単位，もしくは jar ファイル単位のように適宜与えられるものとする．

プログラムを p，プログラムの集合を $P = \{p_1, p_2, ..., p_n\}$ とした時，p_i から抽出した利用クラスを $u(p_i) = U_i = \{u_{i,1}, u_{i,2}, ..., u_{i,m}\}(1 \leq i \leq n)$ とする．この時，p_i 中の $u_{i,j}$ の *tf*（term frequency; 単語出現頻度）の値 $tf(u_{i,j}, p_i)$，*idf*（inverse document frequency）の値 $idf(u_{i,j}, P)$ を次のように定義する．

$$
\begin{aligned}
tf(u_{i,j}, p_i) &= \frac{freq(u_{i,j}, p_i)}{\sum_{k=1}^{m} freq(u_{i,k}, p_i)} \\
df(u_{i,j}, P) &= \{p | p \in P \wedge u_{i,j} \in u(p)\} \\
idf(u_{i,j}, P) &= \log_2 \frac{|P|}{\sum_{k=1}^{m} |df(u_{i,k}, P)|}
\end{aligned}
$$

ここで，$freq(u_{i,j}, p_i)$ は p_i 中に出現する $u_{i,j}$ の数を表しており，$df(u_{i,j}, P)$ は P に属するプログラムのうち，$u_{i,j}$ を 1 つ以上含むプログラムの個数を表す．以上で導出した $tf(u_{i,j}, p_i)$ と $idf(u_{i,j}, P)$ を元に $tf\text{-}idf(u_{i,j}, p_i, P)$ を次式のように定義する．

$$
tf\text{-}idf(u_{i,j}, p_i, P) = tf(u_{i,j}, p_i) \times idf(u_{i,j}, P) \quad (1)
$$

式 (1) の値が大きくなると，利用クラス $u_{i,j}$ はプログラムの集合全体から考えて，少数のプログラムにしか出現しないことになる．つまり，目立つクラスと言える．また，プログラム p_i で，多く出現することにもなり，プログラムの重要な要素となる．逆に式 (1) の値が 0 のとき，全てのプログラムで出現することになり，目立たないクラスと言える．

3 提案手法

ここでは，保護が適用されたプログラムの不自然さの評価手法を提案する．提案手法の概要を図 2 に示す．図 2 に示す通り，提案手法は次のようなステップで評価する．

1. 学習データからデータベースを構築する．
2. 対象データから利用クラスを抽出する．
3. 抽出した利用クラスとデータベースの情報から *tf-idf* を求める．

まず，学習データからデータベースを構築する．この学習データは第 2.3 節で述べた *tf-idf* を算出するときの

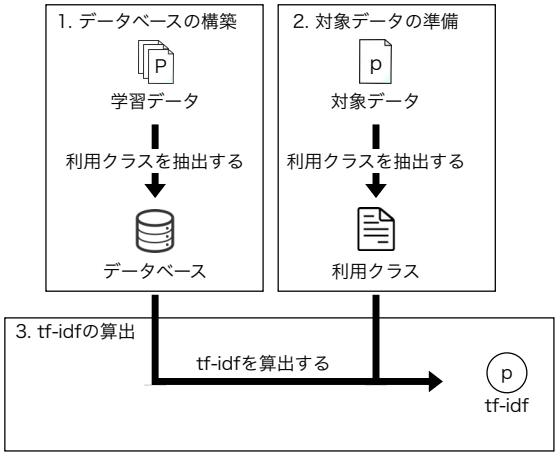


図 2: 提案手法の概要図

P に相当するものである．ここで集めたプログラムから，利用クラスを抽出し，結果をデータベースに保存する．対象とするプログラムは，OSS プロジェクトから収集する．

次に対象データ，すなわち，不自然さを評価したいプログラム（集合）を準備する．ここで用意したデータからも同じく，各プログラムから利用クラスを抽出し，結果を保存しておく．

最後に，学習データから構築したデータベースと対象データから抽出した利用クラスを用いて *tf-idf* を求める．

4 実験準備

ここでは，評価実験で用いる設定や指標を整理する．実験では，難読化前後で提案手法による *tf-idf* がどの程度変化したのかを確認する．

以降，データベース構築のために用いた学習データと，実験で用いるデータ，適用した保護手法，そして，評価指標である変化量について述べる．その後，具体例を用いて変化量について説明する．

4.1 学習データと対象データ

まず，学習データは Maven Central Repository[3]の各プロジェクトから最新のバージョンの jar ファイルを収集した [4]．収集した jar ファイル 17,637 個を学習用データとする．なお，各 jar ファイルからは利用クラスを抽出し，データベースに保存している．

次に，対象データは Maven Central Repository 以外の Maven のリポジトリである Sonatype Releases Repository[4] から，最新バージョンの jar ファイルを無作為に収集した．収集した jar ファイルとその概要を表 4 に示す．

[3] http://central.maven.org/maven2
[4] https://oss.sonatype.org/content/repositories/releases

表 1: Sandmark で実装されている難読化一覧

名前	略称	概要	名前	略称	概要
Array Folder	AF	1 次元配列を多次元配列に変換する	Array Splitter	AS	1 つの配列を 2 つの配列に分割する
Block Marker	BM	Basic Block Marker を適用する	Bludgeon Signatures	BS	すべての static メソッドの引数と返り値の型を Object 型に変換する
Boolean Splitter	BSp	すべての boolean 変数と配列の仕様と定義を変更する	Branch Inverter	BI	if 文と else 文を交換する
Buggy Code	BC	基本ブロックのコピーを作成し，その中にバグを埋め込む．その部分は恒偽式で無視する	Class Encrypter	CE	クラスファイルを暗号化し，実行中に復号する
Duplicate Registers	DR	ローカル変数を取得し，新しい変数とそれへの参照を分割する	Dynamic Inliner	DI	実行時に使用するブランチを決定し，非静的メソッドをインライン展開する
False Refacter	FR	親クラスを追加することで 2 つのクラスをマージする	Field Assignment	FA	各クラスに偽のフィールドを追加し，クラス全体でのフィールドに割り当てを行う
Inliner	IL	static メソッドをインライン展開する	Insert Opaque Predicates	IOP	すべての条件式に恒偽式ライブラリによって提供される恒偽式を付加する
Integer Array Splitter	IAS	配列を 2 つに分割する	Interleve Methods	IM	2 つのメソッドを 1 つにまとめる．実行パスは，新しいメソッドに追加されたパラメータとして入力された恒偽式に基づく
Merge Local Integers	MLI	2 つの int 型を 1 つの long 型にする	Method Merger	MM	1 つの大きなメソッドに各クラスで同じシグネチャを持つ public static メソッドのすべてをマージする
Objectify	Obj	Object 型のフィールドを持つ参照型で，全てのフィールドを置き換える	Opaque Branch Insertion	OBI	空の if 文を挿入する
Param Alias	PA	各クラスにフィールドを追加し，クラスのランダムなメソッドのパラメータにそのフィールドに割り当てる	Promote Primitive Registers	PPR	ローカル変数を Object に昇格する
Promote Primitive Types	PPT	すべてのプリミティブ型をラッパクラスのオブジェクトにする	Publicize Fields	PF	すべてのフィールドとメソッドを public にする
Random Dead Code	RDC	メソッドの終わりに無意味なコードを追加する	Rename Registers	RR	ローカル変数の名前を変える
Reorder Instructions	RI	基本ブロックの命令を並び替える	Reorder Parameters	RP	メソッドの引数の順番を並び替える
Simple Opaque Predicates	SOP	単純な条件式を追加する	Transparent Branch Insertion	TBI	空の if 文を追加する

表 2: Sandmark で実装されている電子透かし一覧

名前	略称	概要	名前	略称	概要
Add Expression	AE	偽の式として透かしを埋め込む	Add Initialization	AI	透かしを分割し、定数プールに埋め込む
Add Method and Field	AMF	透かしを 2 つに分割し、新たなフィールドとメソッドとして埋め込む	Dividson / Myhrvoid	DM	基本ブロックを再整理することで埋め込む
Monden	MD	ダミーメソッドに命令を再配置することで埋め込む	Register Types	RT	ローカル変数の型と数字を結びつけて数字を埋め込む
Static Arboit	SA	恒偽式による電子透かしの埋め込みを行う	String Constant	SCo	透かしを分割し、文字列に埋め込む
Steganography	Stg	偽の式として透かしを埋め込む	Stern	Str	透かしを分割し、定数プールに埋め込む

表 3: その他ツール実装されている難読化手法

名前	略称	概要			
APIBlinder	Ap	動的名前解決難読化手法．プログラムの使用部に現れる名前を文字列として暗号化しておき，実行時元に戻す（DonQuixote）．	Name Obfuscation	NO	名前を意味のない名前に変更する（ProGuard，yGuard，Allatori）．
String Encryption	SE	文字列を暗号化する（Allatori）．	Control Flow Obfuscation	CFO	制御フローを複雑にする（Allatori）．
Shrink	Shr	プログラムを圧縮する（ProGuard，yGuard）．	Optimize	Opt	プログラムの最適化を行う（ProGuard）．

表 4: 対象データとして収集した jar ファイルの説明

jar ファイル名	概要
cuid-java-0.1.1.jar	衝突困難性をもつ id を生成する
generic-archiver-1.1.0.jar	アーカイブフォルダに書き込むための汎用ライブラリ
GoodbyDao-0.0.2.jar	汎用 DAO ライブラリ
jauter-1.7.jar	逆方向ルーティングのためのライブラリ
jsnprsr-0.1.jar	JSON パーサ
livelib-crawler-1.2.0.jar	書籍管理ツール
opal-0.0.4.jar	オープンソースプログラムの解析ライブラリ
rest-storage-1.2.3.jar	CRUD REST サービス
ridioc-core-0.1.jar	Java のための IoC ツール
swinginsulation-1.0.jar	Swing アプリケーションの保護ツール

4.2 保護手法

対象データとして収集した各 jar ファイルに保護手法の適用を行った．保護手法の適用に使用したツールは，Sandmark[5]，DonQuixote[5, 6]，yGuard[6]，ProGuard[7]，Allatori[8]の 5 つである．

使用した保護アルゴリズムをツールごとに名称と略称，概要を表にまとめている．Sandmark の難読化アルゴリズムは表 1 に，電子透かしアルゴリズムを表 2 に，その他のツールを表 3 にまとめる．概要に示しているツール名は，当該アルゴリズムをそのツールが提供していることを表している．

4.3 変化量

提案手法で得られた *tf-idf* 値が，難読化前後でどのように変化するのかを確認するため，変化量を定義する．対象とするプログラム p に，ある難読化手法を適用し p' を得る．それぞれから抽出した利用クラスごとに *tf-idf* 値を求めたとき，各利用クラスの変化量 $A_c(u)$，プログラム全体の変化量 $A(p)$ をそれぞれ次のように定義する．ただし，$A(p_i, p'_i, P)$ を計算するときの $u_{i,j}$ は $u(p_i)$ に属するものとする（$u_{i,j} \in u(p_i)$）．

最後に，難読化を適用したことによる変化量 A'(p) を求める．このとき，$A(p_i, p'_i, P)$ を計算するときの $u_{i,j}$ は $u(p_i)$ と $u(p'_i)$ の和集合とする（$u_{i,j} \in u(p_i) \cup u(p'_i)$）．

$$
\begin{aligned}
A_c(u_{i,j}, p_i, p'_i, P) &= \textit{tf-idf}(u_{i,j}, p_i, P) - \textit{tf-idf}(u_{i,j}, p'_i, P) \\
A(p_i, p'_i, P) &= \sum_{j=1}^{m} |A_c(u_{i,j}, p_i, p'_i, P)|
\end{aligned}
$$

[5] http://sandmark.cs.arizona.edu
[6] https://www.yworks.com/products/yguard
[7] http://proguard.sourceforge.net
[8] http://www.allatori.com

表 5: Cat.class から抽出される利用クラス

	利用クラス名	*tf-idf*	$A_c(u_i, p, p', P)$
$u(p)$	FileReader	$t_1 = 0.885$	0.885
	BufferedReader	$t_2 = 0.749$	0.749
	Reader	$t_3 = 0.558$	0.558
	PrintStream	$t_4 = 0.521$	0.521
	System	$t_5 = 0.409$	0.409
	Object	$t_6 = 0.024$	-0.031
	String	$t_7 = 0.052$	-0.068
$u(p')$	String	$t'_1 = 0.120$	-0.068
	Object	$t'_2 = 0.055$	-0.031
	Integer	$t'_3 = 0.879$	-0.879

4.4 変化量の具体例

ここでは，例として，図 1 に示した Cat に DonQuixote の Ap (API Blinder) を適用した時の変化量を求める．オリジナルのプログラムを p，難読化後のプログラムを p' とした時，p, p' から抽出した利用クラスは表 5 に示す通りである（パッケージ名は省略している）．

Ap は動的名前解決難読化と呼ばれる手法で，全てのメソッド呼び出しをリフレクションを使った動的呼び出しに変換する [5, 6]．動的呼び出しに変換することで，クラス名，メソッド名を文字列として扱うことが可能になる．そこで，文字列を暗号化することによって，呼び出すメソッドを隠蔽する難読化手法である．表 5 からわかるように，$u(p')$ では，$u(p)$ で得られる利用クラスのほとんどが得られなくなっている．$u(p) \cap u(p') = \{\texttt{String}, \texttt{Object}\}$ である．その 2 つのクラスの変化量は，$A_c(\texttt{String}, p, p', P) = t_7 - t'_1 = -0.068$，$A_c(\texttt{Object}, p, p', P) = -0.031$ となった．その他の利用クラスは，一方のみにしか現れていない．そのため，$u(p)$ のみに現れる利用クラスの $A_c(u, p, p', P)$ は正の値，$u(p')$ のみに現れる利用クラスの $A_c(u, p, p', P)$ は負の値になっている．

また，p と p' のプログラム全体の変化量はそれぞれ $A(p, p', P) = 3.221$, $A(p', p, P) = 0.978$ となり，難読化されたプログラムは利用クラスが削除されていることが数値上からも確認できる．

最後に，p に Ap を適用したことによる変化量 $A'(p)$ は $A'(p) = 4.1$ となる．

5 ケーススタディ

ここでは，難読化手法や電子透かし手法などの保護手法が適用されたプログラムの不自然さを提案手法によって評価する．一般的にプログラムはファイル単位で考えられる．ただし，ファイル単位と一言で言ったとしても，ソースファイルで考えた場合と，実行ファイルで考えた場合に単位が異なる．Java 言語であれば，ソースファイル単位で考えた場合，一般的に単一のクラスファイル単位になる．一方，実行ファイル単位で考えた場合，jar

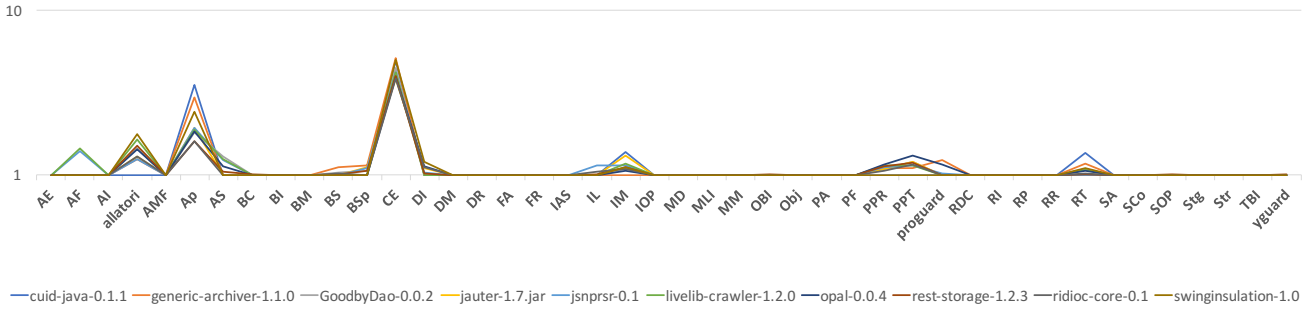


図 3: 保護手法ごとの変化量

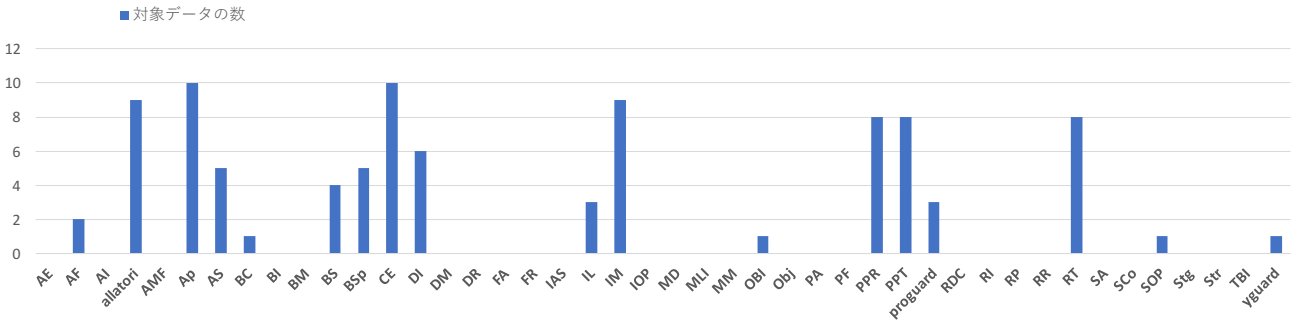


図 4: 保護手法ごとの不自然になった対象データの頻度

ファイル単位となる．どちらのケースであっても，不自然さ評価は必要であり，さらに学習データの単位もそれぞれに合わせる必要がある．そこで，ここでは，(1) プログラムを jar ファイルとして扱った場合，(2) プログラムをクラスファイルとして扱った場合それぞれで不自然さ評価を行う．

5.1 プログラムを jar ファイルとして扱った場合の不自然さ評価

ここでは，プログラムを jar ファイルとして扱い，提案手法によりプログラム全体，jar ファイル単位の不自然さ評価を行う．

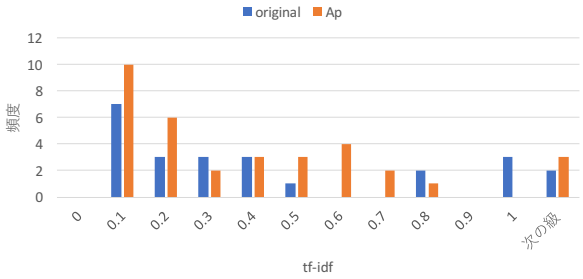
まず，学習データと対象データは第 4.1 節で用意した jar ファイルとする．第 4.3 節で定義した変化量 $A'(p)$ の測定を行なった結果を図 3 に示す．横軸は保護手法の名前，縦軸は変化量を対数で表している．ただし，対数は 0 を表現できないため，変化量を変更し $A'(p, p', P) = A(p, p', P) + 1$ としている．また，対象データ別に折れ線の色で分類している．この値が高いほど，保護を適用することでプログラムがより不自然になったことを示している．

図 3 から，提案手法では不自然さが測定できる保護手法と測定できない保護手法があることがわかる．

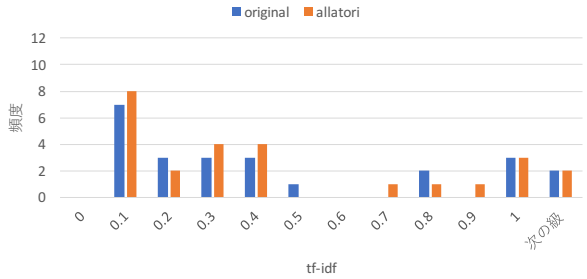
一方，変化量 $A'(p)$ が 0 より大きくなった対象データの頻度を図 4 に示す．この値が高いほど，より多くの対象データ（jar ファイル）で不自然さが測定できたことを示している．

図 3，図 4 を見ると，3 つの保護手法 CE (Class Encyption) と Ap (API blinder)，Allatori が他に比べて高い値を示している．CE と Ap は共に暗号化された情報を実行時に復号している．そのため，復号を行うクラスを新たに利用するようになった．それが原因で不自然さが向上したと考えられる．一方，Allatori は NO (Name Obfuscation) の他，SE (String Encription) と CFO (Control Flow Obfuscation) を採用している．NO はクラス名，メソッド名を意味のない名前に変更するが，標準 API は対象外であるため，本手法に影響を与えない．一方，SE は対象こそ違えど，CE，Ap と同じく，暗号化を行う難読化手法である．そのため，CE，Ap と同じく復号を行うクラスを利用するようになった．このことから，CE，Ap，SE が同じような特徴となり，不自然さが向上したと考えられる．

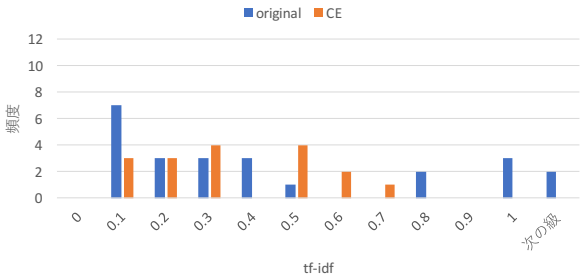
一方，AE (Add Expression) や BI (Branch Inverter) など，いくつかの保護手法では，提案手法での不自然さは全く変化していない．例えば，BI は，if 文と else 文を交換するアルゴリズムである．このようにプログラムの命令列のみに着目したアルゴリズムは，提案手法では不自然さを測定できない．



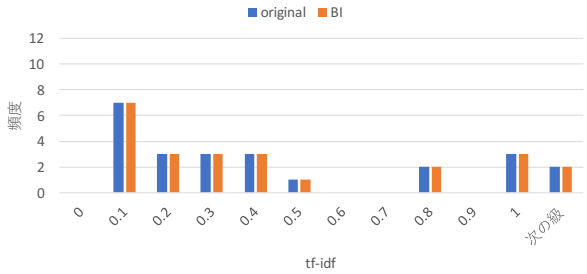
(a) API Blinder (Ap)



(b) Allatori



(c) Class Encryption (CE)



(d) Branch Inverter (BI)

図 5: オリジナルと難読化手法の *tf-idf* のヒストグラム

5.2 プログラムをクラスファイルとして扱った場合の不自然さ評価

ここでは，クラスファイル単位で提案手法を適用し，不自然さ評価を行う．まず，学習データは第 4.1 節で利用したデータと同じである．対象データには，表 4 に示すプログラムのうち，`generic-archiver-1.1.0.jar` を選択した．ただし，学習データの全ての jar ファイルからクラスファイルを抽出し，データベースを再構築しており，対象データからもクラスファイルを抽出して分析している．

ここでは，第 5.1 節で不自然さを評価した保護手法において，利用クラスの数の変化を調べる．4 つの保護手法，Ap，Allatori，CE，そして，BI の *tf-idf* のヒストグラムをそれぞれ図 5a，5b，5c，5d に示す．横軸は *tf-idf* の値であり，縦軸はその頻度を表している．

提案手法で不自然さを測定できた Ap（図 5a），Allatori（図 5b），CE（図 5c）はオリジナルと難読化されたプログラムとで，*tf-idf* の頻度が変わっていることがわかる．一方，提案手法では不自然さを測定できなかった BI（図 5d）をみると，オリジナルと *tf-idf* の頻度が全く変わっていないことがわかる．実際に $u(p)$ と $u(p')$ を調べたところ，全く同じことがわかった．そのため，利用クラスを変更しない保護手法の場合，提案手法による不自然さはオリジナルと全く同じになる．

5.3 命令列を対象とした不自然さ評価手法との比較

命令列を対象に不自然さを測定する手法は，神崎ら，大滝らによって提案されている [2, 3]．本研究の目的は同じ不自然さの測定であるが，測定のために着目する情報が異なっている．本節では，着目する情報によって得られる不自然さの特徴について考察する．

大滝らの手法では保護手法 AF (Array Folder) と AS (Array Splitter) の不自然さは測定できなかった．一方，提案手法では，図 3 を見ると，不自然さが向上していることがわかる．

AF は配列を多次元配列に統合する保護手法，AS は配列を分解する保護手法である．どちらも，命令列に変更はあるものの，配列を操作する命令の並びは不自然なプログラムではないことから，大滝らの手法では不自然さが測定できなかったと考えられる．一方，提案手法の場合は，AF も AS も新たに難読化後の配列を扱うクラスが追加される．例えば，AF の場合，`Folder` が追加されているため，提案手法での不自然さが向上したと考えられる．

このように，難読化を適用後に命令の並びが自然であっても，クラスが追加されることにより不自然さが測定できる．以上より提案手法は，大滝らの手法と補い合う関係にあると言える．

6 まとめ

本稿では Java 言語で記述されたプログラムを対象として，保護手法を適用した場合の不自然さの定量的な評価尺度の提案を行った．プログラムが利用している標準ライブラリのクラス名を用いて *tf-idf* を算出し，その値をプログラムの不自然さとした．そして，プログラムに保護手法を適用することによって，*tf-idf* がどのように変化するのかを数値で表した．ケーススタディでは，その評価尺度を OSS の jar ファイルに，いくつかの保護手法を適用した場合の不自然さの評価を行った．その結果，クラスが追加された場合，提案手法は高い不自然さを示した．特に，暗号化を利用した保護手法においては，非常に高い不自然さを示した．そして，関連研究で不自然さが測定できなかった保護手法のいくつかで，不自然さを測定できた．

しかし，提案手法でも，関連研究で提案された手法においても，不自然さが測定できない手法が存在する．例えば，BM (Block Marker) や MM (Method Merger) である．今後は，それらの手法の不自然さの測定に取り組む．

参考文献

[1] Christian Collberg and Jasvin Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Program Protection*. Addison-Wesley Profes-sional, 2009.

[2] 神崎雄一郎, 尾上栄浩, 門田暁人. コードの「不自然さ」に基づくソフトウェア保護機構のステルシネス評価. 情報処理学会論文誌, Vol. 55, No. 2, pp. 1005–1015, February 2014.

[3] 大滝隆貴, 大堂哲也, 玉田春昭, 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予稿集 (SCIS2014), pp. CD–ROM (2D2–2), January 2014.

[4] Steven Raemaekers, Arie van Deursen, and Joost Visser. The maven repository dataset of metrics, changes, and dependencies. In *Proc. 10th IEEE Working Conference on Mining Software Repositories (MSR 2013)*, May 2013.

[5] 玉田春昭, 中村匡秀, 松本 健一門田 暁人. 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, October 2007.

[6] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Kenichi Matsumoto. Introducing dynamic name resolution mechanism for obfuscating system-defined names in programs. In *Proc. IASTED International Conference on Software Engineering (IASTED SE 2008)*, pp. 125–130, February 2008.