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

命令列に着目した名前難読化の逆変換手法
Reverse Conversion Method for Name Obfuscation Methods based on Method Instructions

匂坂　勇仁 *　　玉田　春昭 †
Hayato Sagisaka　　Haruaki Tamada

あらまし 近年，ソフトウェアに対して違法コピーや無断ダウンロードが問題となっている．それらの攻撃を未然に防ぐため，様々なソフトウェア保護手法が提案されている．保護手法の中には，プログラムの理解が困難になるように変更する難読化がある．難読化の中でも，本稿では世の中で広く使われている名前難読化に着目する．名前難読化とは，メソッド名やクラス名を意味のない名前に変更する手法である．しかし，名前難読化はソースコードの名前のみを変更する手法のため，処理は変化しない．そのため，変数以外全く同じといった脆弱性が含まれる可能性が存在する．そこで本研究では，堅牢性を確認するため，名前難読化手法自体の攻撃耐性を評価する．具体的な手法として，大量のメソッドを DB に構築する．その DB 内のメソッドの命令と名前難読化後のメソッドの命令を比較する．比較した結果，類似度が一定以上だったらソースコードを見比べることで，一致性を確認する．その結果，どの程度類似していたら同一のメソッドと判断できるかを検証する．

キーワード 名前難読化，類似度評価

1　はじめに

近年，ソフトウェアに対して違法コピーや無断ダウンロードが問題となっている．そのソフトウェアの悪用を防ぐために様々なソフトウェア保護手法が提案されている．ソフトウェア保護手法は，プログラム中の秘密情報を保護するため，プログラムを何らかの決まりに従って変換する．そのような保護手法の一つに難読化がある [1]．難読化とは，理解が困難になるよう，また，プログラムの入出力仕様を変更せずに，プログラムを変換する．プログラムのどの部分に着目して変換するかにより，様々な難読化手法が提案されている．我々は，その中でも，世の中で広く使われている名前難読化に焦点を置く．名前難読化とは，メソッド名やクラス名を意味のない名前に変更する手法である．プログラム中の名前は，プログラムを理解するための重要な手掛かりであるため，意味のない名前に変換することで，プログラムの解析を困難にできる．

従来，様々な名前難読化が提案され，ツールとして実装されている．例えば，ProGuard[1]，Dash-O[2]などがある．それらツールは，例えデコンパイルされたとしても，プログラムの理解が困難であると主張しているものの，実際に理解の困難さを測定した研究は存在しない．

そこで，本研究では，名前難読化の評価の一つとして，名前難読化の攻撃耐性を評価する．名前難読化が適用されたプログラムを攻撃し，仮に名前難読化を無効化できれば，名前難読化は脆弱な難読化手法であると言える．また，完全に無効化できないまでも，どの程度の割合で難読化された名前が元に戻せるのかの指標を測定することで，名前難読化の評価指標が確立できる．

名前難読化の攻撃には，名前の逆変換を行う．つまり，意味のない名前にされた名前を元の名前に復元することを目指す．もちろん，名前難読化により，多くの情報が削られているため，完全な復元は不可能であろう．例えば，変数名，メソッド名，クラス名が意味のない名前に変換されており，さらに，メソッド内で呼び出している標準 API 以外のメソッドも意味のない名前に変換されている．そこで，あらかじめソフトウェアリポジトリから難読化されていないプログラムの情報を DB に入れておく．次に，難読化されたプログラムの情報を DB に問

* 京都産業大学大学院　先端情報学研究科 Division of Frontier Informatics, Graduate School of Kyoto Sangyo University.
† 京都産業大学 コンピュータ理工学部 Faculty of Computer Science and Engineering, Kyoto Sangyo University.

[1] https://www.guardsquare.com/en/proguard
[2] https://www.preemptive.com/products/dasho/

い合わせて見つかった名前を元の名前として復元する．

2 関連研究

名前難読化は，プログラム中の名前（変数名やメソッド名，クラス名など）の持つ意味がプログラム理解に貢献するという前提の元，意味のない理解し難い名前に変換することでプログラムを保護している [2]．この難読化手法では，どのように名前を変換するかにより，手法ごとに特徴付けされる．単純に名前をアルファベット 1 文字にする方法や，スペースやタブなどの不可視文字にする方法，予約語にする方法，できるだけ名前を重複させる方法 [3] や，名前の定義部分ではなく，使用する部分を隠す方法 [4] など，一口に名前難読化と言っても数多くの手法が提案されている．

名前難読化以外にも別の難読化として，暗号鍵を含むプログラムにおいて鍵データを隠蔽するのに有効なデータ難読化や秘密データの位置に関する手掛かりを減らしたり，サブルーチンやアルゴリズムの隠蔽に役立つ制御フロー難読化 [5] がある．

名前難読化の逆変換手法は Cimato らにより，提案されている [6]．Cimato らの手法では，フィールドの型情報に着目し，難読化された変数名を，型名を元に基づいた命名を行うことで，逆変換を実現している．しかし，対象としているものはフィールドのみである．提案手法の主な対象はメソッドであり，Cimato らの手法とは対象が異なる．

新たなメソッドを作成した際に，そのメソッドがどの API で使用できるかを推薦してくれる API 集合推薦手法 [7] がある．早瀬らの手法では，開発者が新規作成したいメソッドの名前を記述したときに，そのメソッド本体で使用される可能性のある API の集合を推薦することで，API の選択を支援する手法を提案する．使用される可能性のある API の推薦には大規模なソースコード集合から抽出した相関ルールを使用する．相関ルールには，メソッド名などの識別子から得た情報と，使用された API との関係が記録されている．

しかし，メソッドの名前を難読化された場合，この推薦手法が使用できなくなる．

3 提案手法

3.1 キーアイデア

名前難読化が適用されたプログラムの名前を復元するには，名前難読化されても変化しない情報に着目する必要がある．名前難読化では，プログラム中の名前，すなわち，クラス名，フィールド名，メソッド名，ローカル変数名が変更される．さらに，名前難読化対象となったプログラムから，別の名前難読化対象となるプログラムを参照している場合，その名前も変更される．

一方，名前難読化は，プログラム中の名前を単純に置き換えるだけとも言える．すなわち，名前以外の情報は元のプログラムと変化しない．そこで，メソッド中の命令列の並びは，名前難読化が適用されたとしても変化しないため，メソッドの命令列を用いて名前の復元を行う．

しかし，命令列のみから適切な名前の導出は不可能である．命令列が適切な名前のための糸口であったとしても，命令列のみからそのメソッドの意味を推し量ることは非常に困難である．そのため，名前を導出するため，命令列と名前を結びつける何らかの情報源が必要である．そこで，ソフトウェアリポジトリに置かれているプログラムに着目する．近年では，ソフトウェアリポジトリから必要なライブラリを自動的にダウンロードしてソフトウェア開発が行われる．例えば，Maven Central Repository[3]や Ruby Gems[4]，npm[5]などである．

あらかじめ，ソフトウェアリポジトリ上のプログラム（ライブラリ）を分析しておき，命令列と名前を結びつけた DB を構築する．名前難読化されたプログラムが与えられた時，各メソッドから命令列を抽出し，DB に問い合わせる．類似した命令列が DB 中に存在すれば，その名前が復元の候補となる．

3.2 名前復元法

図 1 に提案手法の模式図を示す．まず，あるソフトウェアリポジトリ R が与えられたとき，R に含まれるプログラム（ライブラリ）の集合を P_R とする．P_R はプログラム p_i の集合である（$P_R = \{p_1, p_2, ..., p_n\}$）．そして，プログラム p_i はプログラム名 c_i と，メソッド情報の集合 $M_i = \{m_{i,1}, m_{i,2}, ..., m_{i,m}\}$ を持つ（$p_i = \{c_i, M_i\}$）．メソッド情報 $m_{i,j}$ はメソッド名 $n_{i,j}$ と命令列（opcode 列）$S_{i,j}$ を含む（$m_j = \{n_{i,j}, S_{i,j}\}$）．命令列は $S_{i,j} = \{s_1, s_2, ..., s_k\}$ で表される．そして，各 p_i から $S_{i,j}$ を抽出し，$d_{i,j} = \{S_{i,j}, c_i, n_{i,j}\}$ の組を DB に保存し，集合 $D = \{d_1, d_2, ..., d_x\}$ を得る．

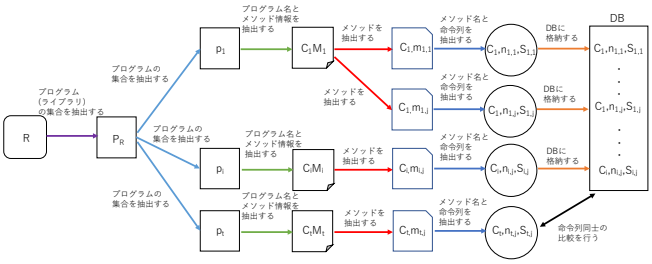


図 1: 提案手法

[3] http://central.maven.org/maven2/
[4] https://rubygems.org
[5] https://www.npmjs.com

一方，名前難読化されたプログラム p_t が与えられたとき，M_t のそれぞれの要素から，$S_{t,j}$ を抽出し，$\mathcal{S} = \{S_{t,1}, S_{t,2}, ..., S_{t,y}\}$ を得る．DB から $S_{t,j}$ と類似する命令列を検索し，対応する $c_i, n_{i,j}$ を元に $n_{t,j}$ の名前を復元する．なお，類似性の計算法 $\mathrm{sim}(S_{i,j}, S_{t,k})$ は別途定める．

3.3 類似性計算法

さて，第 3.2 節で述べたように，名前の復元には，与えられた 2 つの命令列 $S_{i,j}$, $S_{t,k}$ から類似性を導出する必要がある．ここでは以下の 3 つの類似性計算法を定める．

1. Levenshtein 距離（$\mathrm{levenshtein}(S_i, S_j)$）[8]
2. Jaccard 係数（$\mathrm{jaccard}(S_i, S_j)$）
3. コサイン類似度（$\mathrm{cosine}(S_i, S_j)$）

Levenshtein 距離は編集距離とも呼ばれる．一方のシーケンスから，もう一方のシーケンスに変換する手順の数で距離を測定する手法である．ただし，Levenshtein 距離が算出するのは距離であって，類似度ではない．そこで，$1 - \dfrac{\mathrm{levenshtein}(S_i, S_j)}{\max(|S_i|, |S_j|)}$ を計算することで類似度として扱う．

Jaccard 係数は 2 つの命令列 S_i, S_j の積集合を和集合で割ったものである．つまり，$\dfrac{|S_i \cap S_j|}{|S_i \cup S_j|}$ で計算できる．

最後のコサイン類似度は，命令列 S を，命令を要素としたベクトルとして捉え，2 つのベクトルの為す角のコサイン値を類似度としたものである．すなわち，命令列 $S' = \{s'_1, s'_2, ..., s'_l\} (l \leq k)$ を S から重複した要素を取り除いたシーケンスとする．そして，a_x を s'_x が S 中に現れる回数とする．$V = \{(s'_1, a_1), (s'_2, a_2), ..., (s'_l, a_l)\}$ を命令列 S をベクトル化したものとする．このとき，2 つの命令列 S_i, S_j からそれぞれベクトル V_i, V_j を導出し，為す角のコサイン値を $\dfrac{\sum_{k=1}^{l_i, l_j} a_{k_i} a_{k_j}}{\sqrt{\sum_{k=1}^{l_i} a_{k_i}^2} \sqrt{\sum_{k=1}^{l_j} a_{k_j}^2}}$ で求める．

4 評価実験

4.1 概要

本稿では，メソッドの命令列を用いた名前難読化の逆難読化を目的とする．そこで，Java を対象としたメソッドの命令列を用いる．抽出した命令列は，第 3.3 節で述べたように 3 つの指標で比較を行う．

はじめに S の長さと類似度の関係性について検証する．命令列の長さが短いと単純な処理のメソッドが複数存在し，誤推薦が大幅に増えると予測できる．そのため，特徴的な命令のみが比較するには，S の長さををどのくらいにすれば良いのかを検証する．

表 1: 命令列の詳細

Product	Abbr.	Version
ASM	ASM	5.1
JavaBitcoindRpcClient	jbrc	0.9.3
kennisfabriek-tools-github	ktg	1.0.3

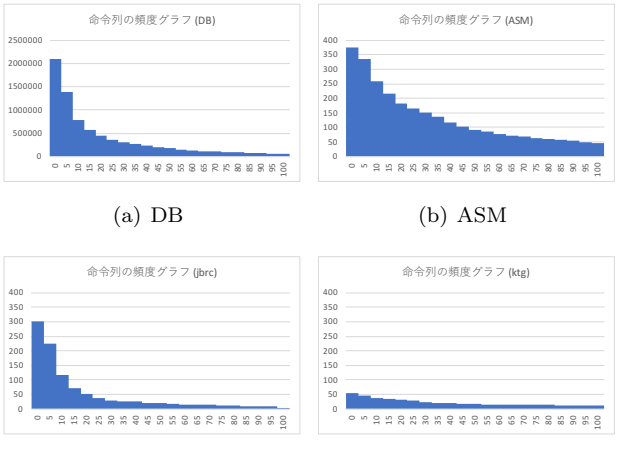
次に，DB に含まれていないプロジェクトを対象に類似度評価を検証する．名前難読化は変数名やメソッド名を変更するため，抽出した命令列に影響はない．そのため，命令列が類似していたら同一の処理をするメソッドである可能性が高く，メソッド名が異なっていても同一の処理とみなして良いと考える．

最後に，難読化後での類似度評価を検証する．名前難読化だけではなく別の難読化が施されていた場合，難読化ごとにどれほど類似性があるかを検証する．類似性があれば別の難読化を施されていても名前だけは逆難読化できる可能性が高くなる．

なお，全ての実験において，DB の構築には，Maven Central Repository を用いた [9]．

4.2 命令列の比較に対する適正長さ

ここでは，S の長さが類似度にどれほどの影響があるかを検証する．また，DB とプロジェクトの S の長さの頻度分布を図 2 に表す．図 2 は縦軸を頻度で，横軸を命令列の長さで表している．図からわかるように，全てのプロジェクトで S の長さが短いものは，数多く存在する．これは，S の長さが短いもので検索すると，誤推薦が多くなることが予測できる．そこで，ここでは，閾値 t を設定し，$|S| \geq t$ となるように DB をフィルタリングした上で推薦の正答率がどう変化するかを確認する．この時の部分集合は，$D_s = \mathrm{s}_D(D, t) = \{d_i | d_i \in D \cap |S_i| \geq t\}$ とする．同じく，$\mathcal{S}_s = \mathrm{s}_{\mathcal{S}}(\mathcal{S}, t) = \{S_i | S_i \in \mathcal{S} \cap |S_i| \geq t\}$ とする．



(a) DB　(b) ASM

(c) jbrc　(d) ktg

図 2: S の頻度分布

検索結果が少なくとも 1 件は存在するように，DB 内に入っているプロジェクトを用いて検索を行った．本実験で対象としたプロジェクトを表 1 に示す．表 1 は DB に含まれていないプロジェクトであるため，DB に追加してから検索を行った．

なお，正答は，第 3.3 節で述べた 3 つの指標で類似度 1 であること，検索結果が 1 件であることとする．さらに，検索結果が元のクラス名，メソッド名と同じでなければ誤検出とみなす．結果を図 3 に表す．横軸は命令列の長さであり，縦軸は前述の正答した割合を示している．

図 3 を見ると，図 3(a) ASM 以外は，S の長さが 10 以下の場合は，正答した割合が非常に低くなっている．また，全体的に，S の長さが 15 を超えると正答の割合が急上昇することがわかる．このことから，S の適正な長さは 15 以上が一つの基準と考えられる．

図 3(a) ASM や図 3(c) ktg を見ると，命令列の長さが長いほど正答の割合は良くなり，図 3(b) jbrc では命令列の長さが 20 以上になると，正答の割合は悪くなっている．

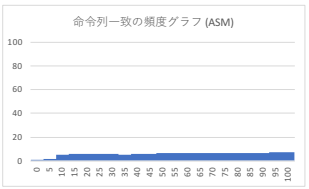
また，図 2 を見ると対象となるプロジェクトの 100 以上の命令列を持つメソッドは少数ということがわかる．これにより命令列をあまりに長くしすぎることは，かえって比較が難しくなる可能性がある．

図 3(a) ASM では，正答の割合が全体的に低くなった．これは DB 内に含まれるプロジェクトが，ASM を含むものが多く存在したため，検索結果が 1 件以上になる場合が非常に多かったことが挙げられる．

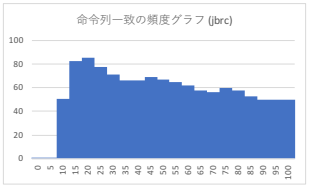
4.3 類似度評価

ここでは互いに異なるメソッド同士で S を比較し，類似度を計測することで，どの程度のメソッドの名前が復元できるかを確認する．

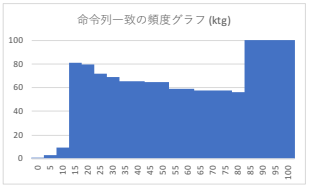
使用するプロジェクトと DB は第 4.2 節と同様のものを使用する．



(a) ASM



(b) jbrc



(c) ktg

図 3: DB との一致度

表 2: 命令列の詳細

t	$\lvert s_D(D,t)\rvert$	$\lvert s_S(\mathcal{S}_{\mathrm{ASM}},t)\rvert$	$\lvert s_S(\mathcal{S}_{\mathrm{jbrc}},t)\rvert$	$\lvert s_S(\mathcal{S}_{\mathrm{ktg}},t)\rvert$
15	575,098	216	71	34
60	133,010	78	16	16
100	6,0942	46	5	11

表 3: t ごとの一致数

	t	完全一致	全て 0.9 以上	編集距離	jaccard 係数	cos 類似度
ASM	15	3,596	6,628	6,870	13,894	212,6107
	60	1,183	1,783	1,787	1,977	25,2683
	100	587	1,003	1,006	1,073	8,4354
jbrc	15	15	41	67	2,179	438,563
	60	12	12	12	15	59,415
	100	6	6	6	6	10,686
ktg	15	18	121	123	4,115	252,683
	60	17	102	102	966	163,800
	100	0	0	0	3	79,655

ただし，DB には表 1 に示したプロジェクトは含めていない．また，今回の比較対象となる S の長さは，第 3.3 で示したように $t = 15, 60, 100$ とした．それぞれの閾値での $|S|$ を表 2 に示す．

比較結果は完全一致，3 つの指標全てが 0.9 以上，3 つの指標のどれかが 0.9 以上の計 5 パターンで比較を行った．類似度がそれぞれの条件に合致する DB の中のメソッドの数を一致数と呼ぶ．その結果を表 3 に表す．

表 3 を見ると，類似度が高いものがいくつか発見できた．3 つの指標のうち cos 類似度はより多く発見できたが，他の指標が小さすぎたため，本研究の類似度評価には適さなかった．

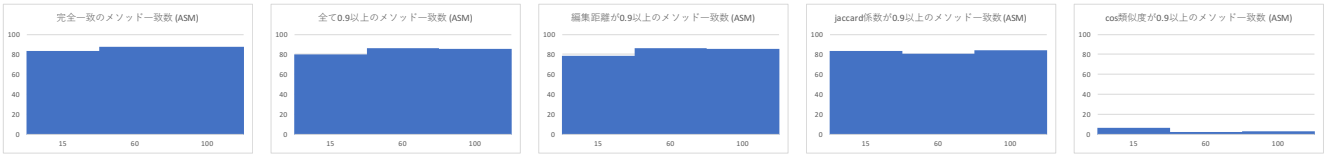
ASM は，第 3.3 節でも述べたように，DB 内にあるプロジェクトのいくつかが ASM を内包していたため，類似度評価でも多くのメソッドが検出された．

プロダクトごとに一致数を算出し，その割合を図 4，5，6 に示す．図 4，5，6 は，それぞれのプロジェクトを対象とした 5 つのパターンでメソッド一致数を表している．横軸は命令数の長さを，縦軸は一致数の割合を表している．

図 4 を見ると，全体的に同じメソッドが多いことがわかる．また，命令列の長さによって多少の違いがあるが大きな変化は見られなかった．cos 類似度 4(e) では，無関係のメソッドを大量に類似していると判断しているため，このような結果になった．

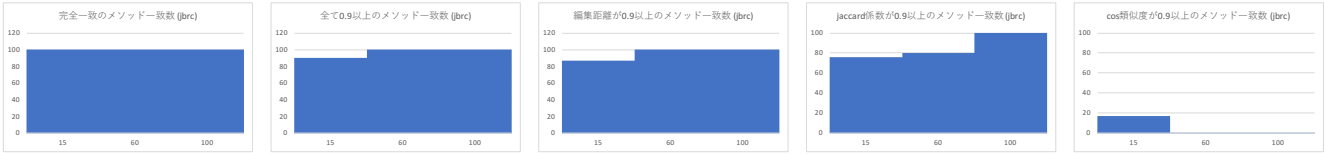
ASM を内包しているプロジェクト内の ASM では，メソッド名が難読化されているため，一致するメソッドに含まれていない．そのため，それらを加えるとこのグラフの値は高くなる．これらの結果により，類似度が高いものはほとんどが関係のあるメソッド同士の命令列ということがわかった．

次に検出した結果がどのようなメソッドであるかを見る．jbrc で検索した結果のうち，全ての指標が一致しつつも，クラス名もしくはメソッド名が異なるメソッドを



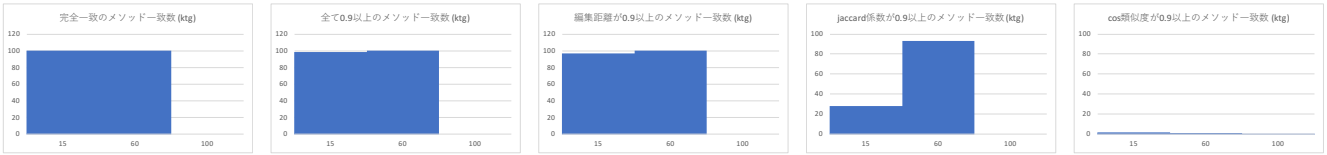
(a) 完全一致　(b) 全て 0.9 以上　(c) 編集距離　(d) jaccard 係数　(e) cos 類似度

図 4: DB と ASM のメソッド一致度



(a) 完全一致　(b) 全て 0.9 以上　(c) 編集距離　(d) jaccard 係数　(e) cos 類似度

図 5: DB と jbrc のメソッド一致度



(a) 完全一致　(b) 全て 0.9 以上　(c) 編集距離　(d) jaccard 係数　(e) cos 類似度

図 6: DB と ktg のメソッド一致度

確認した．それらは全て Base64 の encode 同士，decode 同士であった．Base64 は作成者が異なっていても，アルゴリズムが同じであるため，処理に大きな違いは生まれない．そのため，今回の実験においては，一致しなかったものの，意味的には一致するものと判断できる．一方，3 つの指標全てが 0.9 以上では，類似度が高いものは初期化をする`<init>`メソッド（コンストラクタ）同士や`<clinit>`メソッド（static イニシャライザ）同士が検出された．3 つの指標全てが 0.9 以上で，処理内容が異なるものは，命令列が 15 以上の場合で 1 件だけ検出された．

ktg では，命令列が 100 以上の jaccard 係数が高いメソッドは編集距離が小さいため，類似度が高いメソッドは検出されなかった．別の命令列の長さでは，完全に一致したメソッドは別々のクラスを指しているが，ほとんどが`<clint>`メソッド同士が検出された．しかし，命令列が 15 以上に，1 件だけクラス名もメソッド名も違うメソッドが検出された．

3 つの指標全てが 0.9 以上で，命令列が 60 以上では，`<clint>`メソッド同士しか検出されなかったが，命令列が 15 以上では，メソッド名が違うものもいくつか検出された．このことから，S が短ければ，処理内容に関係のないメソッドが検出される恐れがある．

4.4 難読化前後での比較

あるプログラムにいくつかの難読化を施して難読化前後で提案手法での検索を行う．難読化前のプロジェクトを DB に入れ，難読化後のプロジェクトを比較し，難読化前のプロジェクトのメソッドが検索できるかを確認する．一般的に名前難読化は S の変更は行わないが，名前難読化以外の難読化手法が併用される可能性もある．

そこで，実際に世に出回っている表 4 に示すツールを使い難読化を行う．これらのツールは，Maven リポジトリに入っていないものを選択するため，Sonartype Releases Repository[6]から取得した．難読化前の名前を難読化後の S を使って検索できるかを確認する．対象プロジェクトは表 1 に示した通りである．

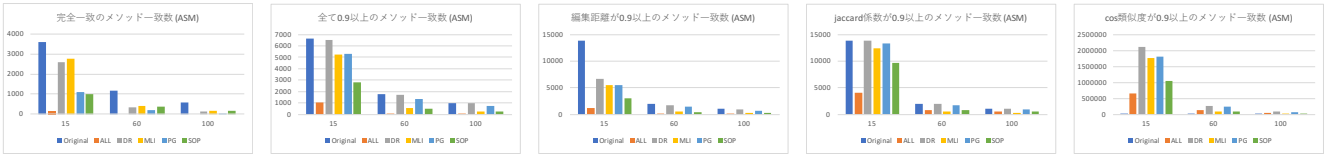
表 4: 利用した難読化ツール

Tools	Abbr.	Overview
オリジナルプログラム	ORI	難読化前のプログラム
Allatori Java Obfuscator	ALL	商用の難読化ツール
ProGurad	PG	OSS の難読化ツール
Sandmark		研究用の難読化ツール
Duplication registers	DR	代入を重複させる．
Merge local integers	MLI	2 つの int 変数を long に収める．
Simple Opaque Predicate	SOP	常に真となる条件文を加える．

表 5: 類似度結果

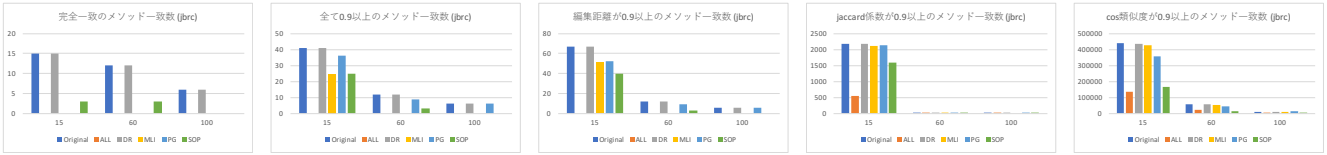
	t	ORI	ALL	DR	MLI	PRO	SOP
ASM	15	216	223	216	216	219	223
	60	78	85	79	84	83	102
	100	46	51	47	66	50	69
jbrc	15	71	98	71	71	86	81
	60	16	20	16	18	19	27
	100	5	9	5	8	8	14
ktg	15	34	37	34	34	55	36
	60	16	18	16	17	26	22
	100	11	11	11	11	20	14

[6] https://oss.sonatype.org/content/repositories/releases



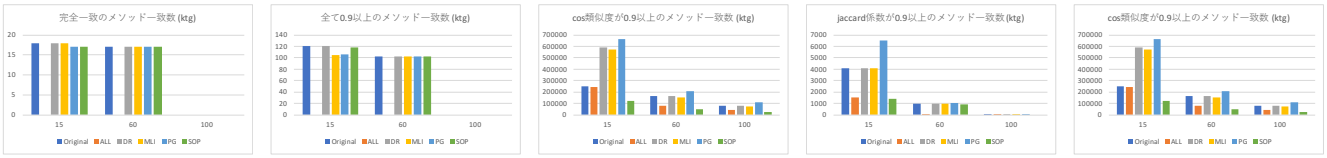
(a) 完全一致　(b) 全て 0.9 以上　(c) 編集距離　(d) jaccard 係数　(e) cos 類似度

図 7: DB と ASM のメソッド一致度



(a) 完全一致　(b) 全て 0.9 以上　(c) 編集距離　(d) jaccard 係数　(e) cos 類似度

図 8: DB と jbrc のメソッド一致度



(a) 完全一致　(b) 全て 0.9 以上　(c) 編集距離　(d) jaccard 係数　(e) cos 類似度

図 9: DB と ktg のメソッド一致度

表 6: ASM の難読化前後の類似度

	t	完全一致	全て 0.9 以上	編集距離	jaccard 係数	cos 類似度
ORI	15	3,596	6,628	6,870	2,126,107	13,894
	60	1,183	1,783	1,787	252,683	1,977
	100	587	1,003	1,006	84,354	1,073
ALL	15	153	1,038	1,221	4,091	652,847
	60	0	65	95	833	141,181
	100	0	8	29	566	54,912
DR	15	2,596	6,493	6,735	13,874	2,112,628
	60	350	1,712	1,716	1,963	260,200
	100	124	965	968	1,068	87,366
MLI	15	2,775	5,260	5,499	12,372	1,757,111
	60	400	571	572	617	90,170
	100	155	235	235	257	24,679
PG	15	1,105	5,280	5,532	13,274	1,805,194
	60	190	1,337	1,407	1,693	241,053
	100	41	734	737	937	78,578
SOP	15	994	2,794	2,968	9,671	1,037,820
	60	358	486	490	786	93,289
	100	168	240	243	527	24,870

表 7: jbrc の難読化前後の類似度

	t	完全一致	全て 0.9 以上	編集距離	jaccard 係数	cos 類似度
ORI	15	15	41	67	2,179	438,563
	60	12	12	12	15	59,415
	100	6	6	6	6	10,686
ALL	15	0	0	0	551	135,749
	60	0	0	0	7	22,422
	100	0	0	0	3	7,749
DR	15	15	41	67	2186	433,331
	60	12	12	12	20	58,999
	100	6	6	6	6	10,837
MLI	15	0	25	51	2,103	424,954
	60	0	0	0	3	53,244
	100	0	0	0	0	8,081
PG	15	0	36	52	2,129	355,811
	60	0	9	9	12	43,355
	100	0	6	6	9	13,204
SOP	15	3	25	40	1,583	164,701
	60	3	3	3	6	13,464
	100	0	0	0	3	6,490

DB は第 4.3 と同じ DB を使用し，対象のプロジェクトを登録しない．その後，難読化後のプロジェクトを検索し，検索結果を確認する．それぞれの結果を表 6，7，8 に示す．また，表の内容を図 7，8，9 に示す．横軸は命令列の長さ，縦軸はその頻度である．

表 6，7，8 の各行を見ると，難読化によって類似度が異なることがわかった．また，表 6，7，8 ごとに異なる結果となっている．すなわち，プロジェクトによって異なる類似度が示されたことがわかる．これは難読化の対象となるメソッドの違いによって現れたと考える．別の観点では，命令例の長さによって大きくグラフの形が変わることがないことがわかった．

ALL (Allatori Obfuscator) では，命令列に変化があり，完全一致や 3 つの指標全てが 0.9 以上では該当する件数が少なかった．つまり，この難読化手法は，大幅に命令列を変更するため類似度検査で発見することは難しい．一方，DR では，命令列に変化があるときと変化がないときがあったまた，3 つの指標全てが 0.9 以上ではオリジナルとほぼ同じ値になり，ほとんどが類似していることがわかった．特徴として，編集距離が小さく，jaccard 係数や cos 類似度が高いことから命令列の並びに影響がある難読化ということがわかる．MLI と PG，SOP では，命令列に変化があり，3 つの指標全てが 0.9 以上ではオリジナルに近い値になった．しかし，類似していないメソッドも存在するため，そのメソッドに対しての検知は難しい．このことから，現状の $\mathrm{sim}(S_{i,j}, S_{t,k})$ の比

表 8: ktg の難読化前後の類似度

	t	完全一致	全て 0.9 以上	編集距離	jaccard 係数	cos 類似度
ORI	15	18	121	123	4,115	252,683
	60	17	102	102	966	163,800
	100	0	0	0	3	79,655
ALL	15	0	0	0	1,550	241,756
	60	0	0	0	96	81,498
	100	0	0	0	1	45,420
DR	15	18	121	123	4,115	592,066
	60	17	102	102	966	163,800
	100	0	0	0	3	79,655
MLI	15	18	105	107	6,870	574,751
	60	17	102	102	960	156,196
	100	0	0	0	1	75,491
PG	15	17	106	110	6,504	662,618
	60	17	102	102	1,026	206,371
	100	0	0	0	10	112,302
SOP	15	17	118	118	1,429	125,902
	60	17	102	102	901	48,280
	100	0	0	0	0	23,965

較方法では，$S_{t,k}$ に影響のある難読化手法は適切な名前を推薦できない．

5 まとめ

本稿では，名前難読化の逆変換を目標に，命令列の類似度からメソッド名の推薦を試みた．提案手法は，予めソフトウェアリポジトリ内の全てのプログラムから，メソッドごとに命令列を抽出しておき，データベースを構築しておく．次いで，名前を復元したいプログラムからも同様にメソッドごとに命令列を抽出し，データベースを検索する．命令列が規定の閾値よりも大きければ，検索して得られたメソッド名が復元された名前の候補となる．

評価実験として，3 つの実験を実施した．はじめに，類似度検出に適した命令列の長さを評価した．その結果，メソッドの命令列の長さが短い場合，満足に推薦できず命令列の長さは 15 以上が必要であることがわかった．次にデータベースに存在しない難読化されたプロダクトのメソッド名を復元できるかを確認する実験を実施した．データベースにコードが存在しない場合，いくつかのアルゴリズム，例えば，Base 64 などは検索できるものの，一般的なメソッドは検索できないことがわかった．最後に，名前難読化とそれ以外の難読化手法が適用されたプログラムが提案手法で検索できるかを確認した．結果は特定の難読化手法は問題なく検出できたものの，一般的に難しいことがわかった．

今後は難読化されても検索できるようにする．具体的には，盗用されたプログラムを検出するバースマーク手法を調査し，提案手法に適用する．

参考文献

[1] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Proc. Principles of Programming Languages 1999, POPL'99*, pp. 311–324, January 1999. (San Antonio, TX).

[2] Paul M. Tyma. Method for renaming identifiers of a computer program. United States Patent 6,102,966, August 2000. Filed: Mar.20, 1998.

[3] PreEmptive Solutions. Java obfuscation DashO. `http://www.preemptive.com/products/dasho` (Last Access: 2015/06/22).

[4] 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, October 2007. (In Japanese).

[5] 門田暁人, Clark Thomborson. ソフトウェアプロテクションの技術動向 (前編). *IPSJ Magazine*, Vol. 46, pp. 431–437, April 2005.

[6] S. Cimato, A. De Santis, and U. Ferraro Petrillo. Overcoming the obfuscation of java programs by identifier renaming. *Journal of Systems and Software*, Vol. 78, pp. 60–72, October 2005.

[7] 早瀬康裕, 鬼塚勇弥, 山本哲夫, 石尾隆, 井上克郎. Api 呼び出しとメソッド周辺の識別子の実績に基づいた pai 集合推薦手法. 情報処理学会論文誌, Vol. 56, No. 8, pp. 692–700, February 2015.

[8] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, Vol. 10, No. 8, pp. 707–710, February 1966.

[9] Steven Raemaekers, Arie van Deursen, and Joost Visser. The maven repository dataset of metrics, changes and dependencies. In *Proc. of the 10th Working Conference on Mining Software Repositories (MSR 2013)*, pp. 221–224, 2013.