Computer Security Symposium 2019
21 - 24 October 2019

Javaバイトコードを対象とした命令の頻度解析による適用難読化ツールの特定

玉田 春昭[1]　神崎 雄一郎[2]

概要：近年様々なソフトウェア保護手法が提案・リリースされている．しかしながら，それらについての評価が十分であるとは言い難いのが現状である．本研究では JVM プラットフォーム向けの既存の難読化ツールに焦点を当て，逆変換の困難さの評価を目指す．逆変換のためには，どのようなツール/手法が適用されたかを特定する必要がある．そのために，逆難読化の困難さ評価の足掛かりとして，本稿では適用された難読化ツールの特定を試みる．ツールの特定には，あらかじめ多くのツールで難読化しておいたソフトウェアを用意する．それらのソフトウェアから命令列の k-gram の頻度を記録しておく．そして，ソフトウェアが与えられた時，その命令列の k-gram の頻度を抽出し，記録しておいたものと比較することで，適用された難読化ツールの特定を試みる．評価実験の結果，Allatori の製品版は特定が可能である反面，ProGuard，yGuard は命令列の変更が少ないため，判定が困難であることがわかった．

キーワード：難読化，逆変換，ソフトウェア保護手法，頻度分析

Identifying applied obfuscation tools by analyzing opcode frequencies for the JVM platform

Haruaki Tamada[1]　Yuichiro Kanzaki[2]

Abstract: Currently, many software protection methods and tools were proposed and released. However, enough evaluation is not conducted for those methods and tools. This paper focuses on the existing obfuscation tools for the JVM platform and evaluates tolerance against de-obfuscation. Generally, we must identify the applied tools/methods at the first step for de-obfuscation. Therefore, this paper tries to identify applied obfuscation tools. The proposed method extracts the frequencies of opcode k-grams from many obfuscated software, and store them. Then, we try to identify the method by matching the opcode k-grams from the given software protected by some tool. From our experimental evaluation, the proposed method succeeded to identify Allatori; however, could not identify ProGuard and yGuard because ProGuard and yGuard change only little opcode sequences.

Keywords: obfuscation, de-obfuscation, software protection method, frequency analysis

1. はじめに

近年のソフトウェアは内部に秘密情報を持つものが多く存在する．それら秘密情報を保護するために，解析を困難にするプログラム難読化技術が用いられている．特に，Android プラットフォームでは，開発環境に標準で難読化ツール ProGuard[*1] が用意されており，難読化する方法が公式ドキュメントで説明されている[*2]．

一方で，昨今，難読化技術の評価についての研究が行われている．Collberg らは難読化の評価には，理解の困難さ（Potency），逆難読化の困難さ（Resilience），発見の困難

[1] 京都産業大学
Kyoto Sangyo University.
[2] 熊本高等専門学校
National Institute of Technology, Kumamoto College.

[*1] https://www.guardsquare.com/en/products/proguard
[*2] https://developer.android.com/studio/build/shrink-code

© 2019 Information Processing Society of Japan

表 1　調査対象の難読化ツール [2]

Tool Name	Ver.	URL
Allatori	6.9	`http://www.allatori.com/`
ProGuard	6.1.1	`https://www.guardsquare.com/en/products/proguard`
yGuard	2.7.1	`https://www.yworks.com/products/yguard`
Zelix Klass-Master (ZKM)	12.0.2	`http://www.zelix.com/klassmaster/`

さ（Stealth），実行効率のオーバヘッド（Cost）の 4 つの指標があると述べている [1]．本稿では，逆難読化の困難さ評価の足掛かりとして，適用された難読化手法の特定を試みる．逆難読化のためには，どの手法が適用されたのかを検知し，その手法に合わせた逆難読化手法を適用する必要がある．そのため，逆難読化の困難さの評価には，適用手法特定の困難さの測定がまず必要である．逆に，何らかの適用難読化特定手法で特定できた場合，その難読化手法には一定の脆弱性があると言える．そのため，命令列から難読化ツールの特定を試みる．

我々は先行研究として，難読化ツールの難読化性能評価を行っている [2]．また，それぞれの難読化ツールの特徴調査も我々の研究グループで行っている [3], [4]．本稿はこれらの内容を踏まえ，どのツールが使われているかの自動判別を試みるものである．

2. 提案手法

2.1 難読化ツール

本稿では，世の中に数多く出回っている Java クラスファイルを対象とした難読化ツールを対象とする．その中で，先行研究として調査した 4 つの難読化ツールに着目する [2]．これら難読化ツールの名前，バージョン，URL を表 1 に示す．yGuard，ProGuard は OSS であり，Allatori，ZKM は 商用の難読化ツールである．

それぞれの難読化ツールが実施する難読化手法を表 2 に示す．表 2 に示す難読化手法はそれぞれ表 3 に示している．これらは，表 1 に示した各ツールのホームページに記載されている内容から導出したものである．表 2 の各行には難読化ツール名と，その難読化ツールがサポートする難読化手法を示しており，サポートする手法の欄に ✓ マークを入れている．なお，OTHERS 欄は表 3 で示していない手法を実施しているものである．OTHERS の具体的な手法は，各難読化ツールごとに異なっている．Dash-O は命令パターンの変換や，ダミーコードの挿入, 電子透かしの挿入などが実施される[*3]．Allatori は電子透かし，yGuard は行番号表の値を変更することでデバッグを困難にする手法を採用している．

*3 `https://www.agtech.co.jp/products/preemptive/obfuscate.html`

表 2　各難読化ツールが提供する難読化アルゴリズム [2]

	NO	SE	CO	DO	LO	OTHERS
Allatori	✓	✓	✓		✓	✓
ProGuard	✓				✓	
yGuard	✓				✓	✓
ZKM	✓	✓	✓		✓	

表 3　難読化手法 [2]

Abbr.	Description
NO	名前難読化手法を表す．変数名やメソッド名，クラス名を意味のない名前に変換する．
SE	文字列暗号化手法を表す．予め文字列リテラルを暗号化しておき，参照する直前に復号する．
CO	コントロールフロー難読化手法を手法を表す．Opaque Predicate やループの複雑化 [5]，制御構造の平坦化 [6] を行う場合が多い．
DO	データ難読化を表す．2 つの `int` 型をマージして扱う手法 [7] のほか，計算式を複雑化する方法 [8]，複数の boolean 型変数を 1 つの変数にまとめる手法 [9] などがある．
LO	レイアウト難読化を表す．クラスの統合・分割や，メソッドを他のクラスへの移動 [10] などを行う．また，不要なメソッドや変数の削除を行う場合もある．

ただし ZKM は，生成したバイトコードの解析もライセンスで禁止しているため，本稿の分析対象から除外する．

2.2 キーアイデア

本稿では適用された難読化手法を特定するために，図 1 に示すような手順を用いる．最初に，ソフトウェア群（$\mathcal{S}$:　ソフトウェア A_1〜A_z）を用意し，それらをいくつかの難読化ツール（O_1〜O_m）でそれぞれ難読化する（$O_i(A_j) (1 \leq i \leq m, 1 \leq j \leq n)$．これをまとめて，$O_i(\mathcal{S})$ とする）．結果として，$n \times m$ 個のソフトウェアを得る．そして，それらから命令列の k-gram の頻度を測定しておき，保存しておく．

次に，難読化手法を特定したいソフトウェア（S）が与えられる．S からも，同様に命令列の k-gram の頻度を抽出し，保存していたデータと比較する．こうすることにより，難読化手法の特定を試みる．

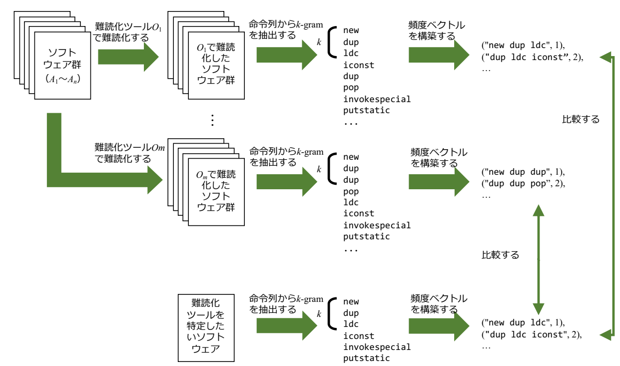


図 1　提案手法の模式図

2.3 頻度ベクトル

あるソフトウェア S には複数のプログラム部品 p_i が含まれるものとする（$S=(p_1,p_2,...,p_n)$）．プログラム部品 p_i には複数のメソッド（$r_{i,j}$）が含まれる（$p_i=(r_{i,1},r_{i,2},...,r_{i,m})$）．そして，各メソッド $r_{i,j}$ には複数の命令（$C_{i,j}=(c_1,c_2,...,c_l)$）が含まれる．ここで，各 $r_{i,j}$ から命令列の k-gram を抽出し，k-gram のリストを得る（$R_{i,j}=\{G_1,G_2,...,G_x\}(x=l-k+1)$）．

一方，対象言語で有効な命令の集合を $\mathcal{C}=\{c_1,c_2,...,c_w\}$ とし，$\mathcal{C}$ から k 個の命令列の組み合わせの集合 $\mathcal{C}_k$ を取り出す（$\mathcal{C}_k=\{G_1,G_2,...,G_y\}(y={}_wC_k=\frac{w!}{(w-k)!k!})$）．そして，$R_{i,j},\mathcal{C}_k$ から頻度ベクトル $v_{i,j}$ を構築する（$v_{i,j}=\{(G_1,d_{i,j,u}),(G_2,d_{i,j,u}),...,(G_y,d_{i,j,u})\}$，$|v_{i,j}|=|\mathcal{C}_k|=y$）．ただし，$d_{i,j,u}(1\le u\le y)$ は，$R_{i,j}$ での G_u の出現回数とする．

次に，各メソッドから取り出した $v_{i,j}$ の各要素を加算し，v を構成する（$v=\{(G_1,N_1),...,(G_y,N_y)\}$）．ただし，$N_u=\sum_{i=1,j=1}^{i\le n,j\le m} d_{i,j,u}$ で求めるものとする．このように S から v を取り出す関数を vectorize とし，$v=\text{vectorize}(S)$ として表す．

一方，あるソフトウェア集合 $\mathcal{S}=\{A_1,...,A_z\}$ から k-gram の頻度集合 $V=\{v_1,...,v_z\}$ を取り出す（$v_l=\text{vectorize}(A_l)=\{(G_1,N_{1,l}),...,(G_y,N_{y,l})\}$）．各要素を加算して $\mathcal{V}$ を導出する（$\mathcal{V}=\{(G_1,\mathcal{N}_1),...,(G_y,\mathcal{N}_y)\}$）．ここで $\mathcal{N}_t=\sum_{l=1}^{l<z} N_{t,l}$ とする．

2.4 コサイン類似度

ソフトウェア S から頻度ベクトル v，ソフトウェア集合 $\mathcal{S}$ から頻度ベクトル $\mathcal{V}$ をそれぞれ得る．これら 2 つのベクトルのなす角のコサインを元のソフトウェアとソフトウェア集合の類似度 $sim(S,\mathcal{S})$ とする．具体的には，$sim(S,\mathcal{S})=\frac{v\cdot\mathcal{V}}{|v||\mathcal{V}|}$ で求める．

3. 評価実験

3.1 実験準備

3.1.1 対象ソフトウェア

ここでは，ソフトウェア集合として，`mvnrepository.com` でよく用いられるのカテゴリのうち，Bytecode libraries，Command line parsers からそれぞれソフトウェア集合 $\mathcal{S}_b$ と $\mathcal{S}_c$ を構築する．それぞれの集合には，各カテゴリからダウンロード数が多いものから 10 個程度集めている．具体的な利用ソフトウェアとバージョンを表 4 に示す．

また，S を cojen 2.2.5（S_{cojen}）と bcel 6.3.1（S_{bcel}）とした．これら 2 つは mvnrepository.com で Bytecode libraries に分類されるソフトウェアであり，$S_{\text{cojen}}\notin\mathcal{S}_b$，$S_{\text{bcel}}\in\mathcal{S}_b$ である．

表 4 ソフトウェア集合として用いたソフトウェア

$\mathcal{S}_b$ (Bytecode libraries)		$\mathcal{S}_c$ (Command line parsers)	
asm	7.1	airline	0.8
baksmali	2.3	argparse4j	0.8.1
bcel	6.3.1	args4j	2.33
byte-buddy	1.9.14	java-getopt	1.0.13
bytecode	1.2	jcommander	1.72
cglib-nodep	3.2.12	jopt-simple	5.0.4
com.ibm.wala.shrike	1.5.3	jsap	2.1
javassist	3.25.0-GA	picocli	4.0.1
jitescript	0.4.1	scopt	3.7.1
scalap	2.13.0		
serp	1.15.1		
smali	2.3		

3.1.2 適用する難読化ツール

ここで，$\mathcal{S}_b$，$\mathcal{S}_c$ の各ソフトウェアを Allatori，ProGuard，yGuard でそれぞれ難読化し，オリジナルを含めて計 10 個のソフトウェア集合を得る．それぞれ $O_x(\mathcal{S}_y)(x=\{a,p,y\},y=\{b,c\})$ である．加えて，S_{cojen}，S_{bcel} も同難読化ツールを適用し，難読化されたソフトウェアを得る（$O_x(S_z)(x=\{a,p,y\},z=\{\text{cojen},\text{bcel}\})$）．難読化ツールによっては難読化のレベルを選択可能であるが，今回はデフォルトの難読化レベルを適用した．なお，本稿で用いたツールのうち ProGuard，yGuard は OSS であり，Allatori は製品版を用意した．

3.2 評価結果

$\mathcal{S}_b$，$\mathcal{S}_c$，S_{cojen}，S_{bcel}，および，それらを難読化したものから命令列の k-gram を抽出した．そして，それらを相互に比較し，コサイン類似度を算出した．$k=1,2,3,4,5,6$ の結果をヒートマップにしたものを図 2 に示す．各図の横軸には S_{cojen}，S_{bcel}，縦軸に $\mathcal{S}_b$，$\mathcal{S}_c$ を並べており，区切りとして白線を引いている．また，S_{cojen}，S_{bcel}，$\mathcal{S}_b$，$\mathcal{S}_c$，それぞれは，オリジナル，Allatori，ProGuard，yGuard で難読化したものを並べ，相互の比較結果（コサイン類似度）をヒートマップのセルとして示している．そして，各比較結果を表す色は図 3 に示しており，左端が 0，右端が 1 を表しており，0〜1 への分布を示している．

図 2 の (a)〜(f) を見ると，各図には同じようなパターンが 4 つ含まれていることがわかる．これはつまり，S が $\mathcal{S}$ に含まれているか否かは評価結果に大きな影響はないと考えられる．

また，k が増加するごとに各セルの色の差がはっきりしてきている．つまり，k が大きい方が適用された難読化手法を特定しやすいと言える．

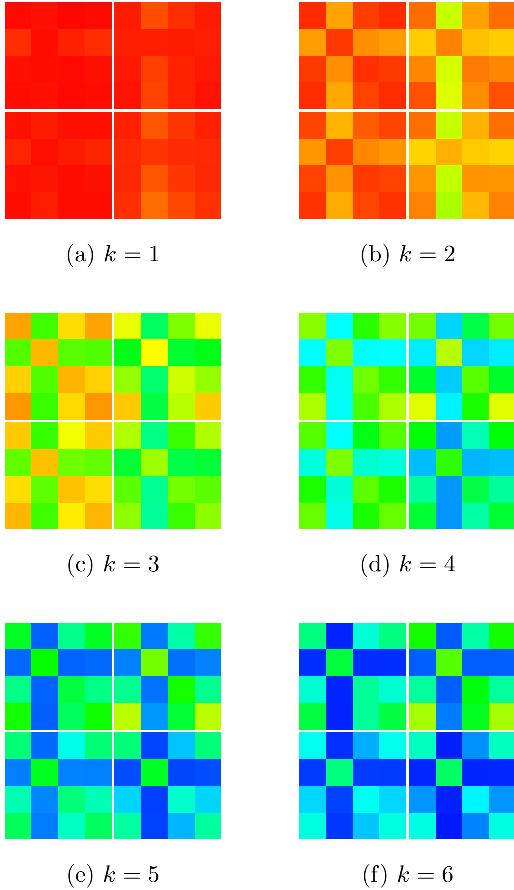
図 2 (f) では，Allatori がオリジナル，他の難読化ツールを適用したもので顕著な差が見られる．逆に，ProGuard，yGuard はオリジナルとの差が見られず，少なくとも提案手法では適用難読化手法の特定が難しいと言える．ただ

表 5　評価結果（$k=6$）

6-gram			$\mathcal{S}_{\mathrm{cojen}}$ Original	$\mathcal{S}_{\mathrm{cojen}}$ $O_a(\mathcal{S}_{\mathrm{cojen}})$	$\mathcal{S}_{\mathrm{cojen}}$ $O_p(\mathcal{S}_{\mathrm{cojen}})$	$\mathcal{S}_{\mathrm{cojen}}$ $O_y(\mathcal{S}_{\mathrm{cojen}})$	$\mathcal{S}_{\mathrm{bcel}}$ Original	$\mathcal{S}_{\mathrm{bcel}}$ $O_a(\mathcal{S}_{\mathrm{bcel}})$	$\mathcal{S}_{\mathrm{bcel}}$ $O_p(\mathcal{S}_{\mathrm{bcel}})$	$\mathcal{S}_{\mathrm{bcel}}$ $O_y(\mathcal{S}_{\mathrm{bcel}})$
$\mathcal{S}_b$	Original		0.375	0.034	0.287	0.374	0.518	0.090	0.337	0.518
	Allatori	$O_a(\mathcal{S}_b)$	0.044	0.442	0.043	0.044	0.092	0.583	0.091	0.092
	ProGuard	$O_p(\mathcal{S}_b)$	0.296	0.035	0.351	0.296	0.347	0.093	0.490	0.347
	yGuard	$O_y(\mathcal{S}_b)$	0.435	0.034	0.348	0.435	0.662	0.123	0.470	0.662
$\mathcal{S}_c$	Original		0.266	0.046	0.171	0.266	0.311	0.030	0.142	0.311
	Allatori	$O_a(\mathcal{S}_c)$	0.055	0.375	0.057	0.055	0.036	0.401	0.035	0.036
	ProGuard	$O_p(\mathcal{S}_c)$	0.211	0.063	0.261	0.210	0.147	0.031	0.241	0.147
	yGuard	$O_y(\mathcal{S}_c)$	0.283	0.053	0.195	0.282	0.279	0.025	0.129	0.279

し，我々の先行研究においても，ProGuard，yGuard は共に命令列を大きく変更しないことがわかっている [2]．そのため，命令列以外の手段を用いて特定が必要となろう．

表 5 に図 2 (f) の具体的な数値を示す．表 5 を見ると，$O_a(\mathcal{S}_b)$ と $O_a(\mathcal{S}_{\mathrm{cojen}})$ の比較結果は 0.442 と，$O_a(\mathcal{S}_b)$ の行，$O_a(\mathcal{S}_{\mathrm{cojen}})$ の列の他のセルに比べて顕著に高い．そのため，Allatori が適用された判定は他の難読化ツールとの差により判断できると言える．



(a) $k=1$　(b) $k=2$

(c) $k=3$　(d) $k=4$

(e) $k=5$　(f) $k=6$

図 2　評価結果のヒートマップ（$k=1,2,3,4,5,6$）

図 3　色の分布

表 6　*sim* と *ave* の一覧と自動判別結果（$k=6$）

		O_x	*sim*	*ave*	$\varepsilon_s=0.9$, $\varepsilon_a=0.05$ ∧	$\varepsilon_s=0.9$, $\varepsilon_a=0.05$ ∨	$\varepsilon_s=0.4$, $\varepsilon_a=0.05$ ∧	$\varepsilon_s=0.4$, $\varepsilon_a=0.05$ ∨
$\mathcal{S}_b$	$\mathcal{S}_{\mathrm{cojen}}$	Original	0.375	0.187				
		Allatori	0.442	0.030		✓	✓	✓
		ProGuard	0.351	0.166				
		yGuard	0.435	0.295				
$\mathcal{S}_c$	$\mathcal{S}_{\mathrm{cojen}}$	Original	0.266	0.132				
		Allatori	0.375	0.042		✓		✓
		ProGuard	0.261	0.116				
		yGuard	0.282	0.136				
$\mathcal{S}_b$	$\mathcal{S}_{\mathrm{bcel}}$	Original	0.518	0.262				✓
		Allatori	0.583	0.075		✓	✓	✓
		ProGuard	0.490	0.217				✓
		yGuard	0.662	0.285				✓
$\mathcal{S}_c$	$\mathcal{S}_{\mathrm{bcel}}$	Original	0.311	0.037		✓		✓
		Allatori	0.401	0.007		✓	✓	✓
		ProGuard	0.241	0.026		✓		✓
		yGuard	0.279	0.037		✓		✓

3.3　議論 ―自動的に判別するには―

本研究の最終目的は，適用された難読化ツールの自動判別である．自動判別のために 2 つの閾値 ε_s と ε_a を導入する．そして，表 5 のような結果一覧が得られたものと考える．ε_s はコサイン類似度の閾値であり，ε_s を超えるペアがあった場合，当該難読化ツールが適用されたものと考える（$sim > \varepsilon_s$）．一方，当該ペア以外のペアの類似度の平均を *ave* とした時，ε_a は *ave* の閾値を表し，$ave < \varepsilon_a$ となるペアがあった場合に，当該難読化ツールが適用されたものと考える．

3.2 節にて，Original，ProGuard，yGuard は提案手法での判別が困難であると結論付けた．そのため Allatori を自動判別するための ε_s と ε_a を導出する．そこで，$k=6$（表 5）での *sim* と *ave* の一覧を表 6 に示す．なお，表 6 では，3.2 節で判別が困難であると結論付けた項目はグレーで表している．表 6 の *sim* 欄は，$\mathcal{S}_y$ と $\mathcal{S}_z$ の比較のうち，O_x に対応するペア同士の類似度を表す．同様に *ave* 欄は，$\mathcal{S}_y$

と S_z の比較のうち，O_x 以外のペアの類似度の平均を示している．そして，続く列には，ε_s と ε_a の具体的な値と，$\mathrm{sim} > \varepsilon_s$，$\mathrm{ave} < \varepsilon_a$ の論理積（∧），論理和（∨）で判定した時に，判別できた項目にチェック（✓）を入れている．

表 6 を見ると，$\varepsilon_s = 0.9, \varepsilon_a = 0.05$ の論理積の場合，判別できてほしい Allatori が判別できていない．同様に論理和の時，判別不能である Original，ProGuard，yGuard で判別できている．$\varepsilon_s = 0.4, \varepsilon_a = 0.05$ の論理和の場合も同様に，$\mathcal{S}_c$ と S_{bcel} の比較で，判別不能であるはずの項目が判定できている．一方，$\varepsilon_s = 0.4, \varepsilon_a = 0.05$ の論理積においても，$\mathcal{S}_c$ と S_{cojen} の Allatori が判別できていない．以上のことから，この例においては $\varepsilon_s = 0.4, \varepsilon_a = 0.05$ の論理積での判定がより有効であろうと考えられる．ただし，具体的な数値はより大規模な評価実験を通じて決定する必要があり，最終的な値はユーザによって決められる必要があろう．

4. 関連研究

従来から特定の難読化アルゴリズムの評価は様々な方法で行われている．人の短期記憶をキューに見立ててプログラムの理解しやすさの評価 [11] やその評価方法を用いて，様々な難読化アルゴリズムを比較した研究 [12] がある．また，Anckaert らは，複雑さメトリクスを用いて難読化アルゴリズムを評価している [13]．二村らは理解の困難さを，コルモゴロフ複雑度を用いて計測した [14], [15]．

また，難読化手法の発見の困難さを測定する手法として，命令列の k-gram を用いる手法が提案されている [16], [17]．こちらも多くの難読化アルゴリズムを対象に評価が行われている．本稿は，特定のアルゴリズムではなく，難読化ツールを対象とした評価である点に違いがある．また，与えられたプログラムが難読化されているか否かを機械学習で判定する研究も行われている [18]．Wang らは iOS を対象にどのような難読化ツールが使われているかを人手で調査した [19]．調査では，1,145,582 個の iOS アプリの中から難読化が適用された 6,600 個のアプリを自動で抽出し，手作業で解析している．その結果，601 個の難読化された iOS アプリ（元は 539 個の iOS アプリ）を特定している．ただし，この研究は難読化ツール・アルゴリズムの性能を評価するものではない点に留意されたい．

一方，Salem らは Malware に適用された難読化を機械学習を用いて特定している [20]．ただし，対象は C 言語であり，Tigress[*4]で難読化されたプログラムを手法ごとに分類している．最終的な目的は，本稿と同じであるものの，対象とする言語やツール，アプローチが異なっている．

*4 http://tigress.cs.arizona.edu/

5. まとめ

本稿では難読化されたソフトウェアから適用された難読化ツールを特定するための簡易な手法を提案した．提案手法は，あらかじめ難読化しておいたソフトウェア群と，特定したいソフトウェアそれぞれから抽出した命令列の k-gram の頻度ベクトルを比較するものである．Java 言語を対象とする Allatori，ProGuard，yGuard を対象に評価を行った．特定のカテゴリからソフトウェア群を構築し，適用された難読化ツールが特定できるかを試みた．その結果として以下のようなことが考えられる．

- $k = 6$ で Allatori が特定できる．
- ProGuard，yGuard は命令列を大きく変更しないため，提案手法では特定が難しい．
- Bytecode libraries と Command line parser のカテゴリ間で大きな違いは見られなかった．
- ソフトウェア群に対象ソフトウェアが含まれているか否かは結果に大きな影響を与えない．

今後の課題として評価実験のスケールアップや異なるカテゴリ，難読化ツールの拡充に取り組む．また，未知の難読化ツールを如何に特定するかについても今後の課題とする．

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

参考文献

[1] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Program Protection.* Addison-Wesley Professional, 2009.

[2] 玉田春昭, 神崎雄一郎．オペコードの編集距離を用いた JVM 向け難読化ツールの難読化性能の評価．2019 年暗号と情報セキュリティシンポジウム予稿集 (SCIS 2019), pp. 3D2–1, January 2019.

[3] 匂坂勇仁, 玉田春昭．適用保護手法特定の試み —不自然さ評価方法を用いて—．信学技報, 第 IEICE-115 巻, pp. 63–68, July 2015. 札幌.

[4] Hayato Sagisaka and Haruaki Tamada. Identifying the applied obfuscation method towards de-obfuscation. In *In Proc. 15th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2016)*, pp. 873–878, July 2016. Oakayama, Japan.

[5] T. Hou, H. Chen, and M. Tsai. Three control flow obfuscation methods for java software. *IEE Proceedings-Software*, Vol. 153, No. 2, pp. 80–86, 2006.

[6] T. László and A. Kiss. Obfuscating c++ programs via control flow flattening. Technical report, Annales Univ. Sci. Budapest. Sect. Comp., 2009.

[7] J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proc. the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pp. 67–78, 1995.

[8] 佐藤弘紹, 門田暁人, 松本健一．データの符号化と演算子の変換によるプログラムの難読化手法．信学技報, 情報セキュリティ研究会, pp. 13–18, Mar. 2002.

[9] L. Badger, L. D'Anna, D. Kilpatrick, B. Matt, A. Reisse,

and T. van Vleck. Self-protecting mobile agents obfuscation techniques evaluation report. Technical report, Network Associates Laboratories, Ja. 2002.

[10] 福島和英, 桜井幸一. メソッド分散による java 言語の難読化手法の提案. 情報処理学会シンポジウム論文集, 第 2002 巻, pp. 191–196, 2002.

[11] M. Nakamura, A. Monden, T. Itoh, K. Matsumoto, Y. Kanzaki, and H. Satoh. Queue-based cost evaluation for mental simulation process in program comprehension. In *Proc. 9th International Software Metrics Symposium (METRICS 2003)*, pp. 351—360, Sept. 2003.

[12] H. Tamada, K. Fukuda, and T. Yoshioka. Program incomprehensibility evaluation for obfuscation methods with queue-based mental simulation model. In *Proc. 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2012)*, pp. 498–503, Aug. 2012.

[13] B. Anckaert, M. Madou, B. Sutter, B. Bus, K. Bosschere, and B. Preneel. Program obfuscation: a quantitative approach. In *Proc. of the 2007 ACM workshop on Quality of protection (QoP 2007)*, pp. 15–20, Oct. 2007.

[14] 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. 命令の乱雑さに基づくプログラム理解性の評価. ソフトウェア工学の基礎 XIX, 日本ソフトウェア科学会 FOSE2012, pp. 151–160, Dec. 2012.

[15] 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, Sept. 2013.

[16] 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, Feb. 2014.

[17] 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), Jan. 2014. CD-ROM (2D2-2).

[18] 北岡哲哉, 神崎雄一郎, 森川みどり, 門田暁人. ランダムフォレストを用いた難読化されたコードのステルス評価の検討. 第 18 回情報科学技術フォーラム（FIT2019）, September 2019.

[19] P. Wang, Q. Bao, L. Wang, S. Wang, Z. Chen, T. Wei, and D. Wu. Software protection on the go: A large-scale empirical study on mobile app obfuscation. In *Proc. 2018 ACM/IEEE 40th International Conference on Software Engineering (ICSE 2018)*, pp. 26–36, 2018.

[20] Aleieldin Salem and Sebastian Banescu. Metadata recovery from obfuscated programs using machine learning. In *Proc. the 6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW 2016)*, No. 1, 2016.