Computer Security Symposium 2020
26 - 29 October 2020

バイナリの類似性を利用した命令列の差分検出

森田 悟大[1,a)]　岡村 真吾[1,b)]

概要： プログラムの処理の類似性はコンピュータセキュリティやプログラム解析の分野における重要な指標であり，既知の脆弱性を持つコードと同様の処理が行われている箇所の検出や，マルウェアのようなプログラムに固有の処理の発見に用いられている．このとき，類似するが一致はしない処理についてその差分を明らかにすることは処理の内容を明らかにする上で重要になる．本研究では，プログラムの類似性を用いて複数の似通った命令列の異なる箇所を明らかにする手法を提案する．また難読化のための VM（Virtual Machine）に対して本研究を適用し，複数の命令の差分から行われている処理を特定する例を示す．

Application of similarity of binaries to show the difference of instructions

Godai Morita[1,a)]　Shingo Okamura[1,b)]

Abstract: The similarity of the procedure is an important metric in the field of cybersecurity and program analysis. The way is used for detection that uses known vulnerable code and for detection of the domain-specific process such as malicious software. At this time, to turn out the difference of the similar procedures is important to make sense what the procedure processes. In this paper, we present a way to show the difference between the two similar procedures. Additionally, we apply the proposed way to the obfuscation VM(Virtual Machine) to show the difference between instructions.

1. はじめに

既知の脆弱性を内包するプログラムや，類似の脆弱性を含みうるプログラムの検出の指針として，またマルウェアのような悪意あるソフトウェアに共通して見られる挙動の評価基準として，バイナリの類似性の算出が近年注目されている [7] ．これらの手法では複数のプログラムに対して，処理が共通，あるいは類似していると思われる部分について，それらがどの程度類似しているかを数値化して示すことができる．

一方，類似した複数の処理について，それらがどのような差異を持っているかということもまた，プログラムを解析し脆弱性やその挙動を評価する上で重要な指標となりうる．特に多くのプログラムには共通して利用されている処理が存在し，それらはプログラミング言語のライブラリや，デザインパターンとして普遍的に利用されている．すなわちあるプログラムの目的や挙動を解析するためには，複数のプログラムに共通した処理でなく，他のプログラムと異なる処理について解析することが有効であると予想した．プログラムの処理を解析されにくくする手法として，仮想マシン（VM) を用いた難読化が存在する．プログラムを用いて仮想的に独自の命令セットを持つ CPU やメモリを実装し，難読化を施したい機械語列を実装した VM の命令列に変換する．難読化 VM でこの変換した命令列を実行すると，変換前の機械語列と等価な内容が VM 上で処理される．このとき，実行可能バイナリの機械語としては難読化 VM のものだけが存在し，もともとのプログラムは難読化 VM の命令列に変換されているため，プログラムを解析する際にはまずはじめに VM の構造や命令セットを解析する必要が生じ，難読化が施されるというものである．VM はその機械語列が大きくなりやすいが各命令の実装に共通する処理も多く，解析時には命令同士の差分に注目することになる．

[1] 奈良工業高等専門学校
National Institute of Technology, Nara College
[a)] godai@info.nara-k.ac.jp
[b)] okamura@info.nara-k.ac.jp

本研究では，仮想マシンのようなプログラムの解析手法として，バイナリの類似度を指標として用い，プログラムの複数の類似する処理に対してそれぞれの差分を抽出する手法について提案する．文献 [4] などの手法を用いて命令列から処理の類似性を評価した上で，複数のアルゴリズムを用いて処理同士を比較し，一致していると判断できる部分を取り除くことで命令列中の処理毎に異なる部分を明らかにする．

2. 提案手法

2.1 概要

あるプログラム中の 2 つの類似する処理からその差分を得る手法として，本研究では 2.2 節で述べる strand への分割と 2.3 節で述べる最長共通部分文字列を用いる．strand とは機械語命令列の基本単位の一つである Basic Block を更に分割した単位であり，Basic Block に比べて操作の意味やデータの流れに対応した単位となるため意味のある単位での分割を行うことができる他，類似した処理の中のある strand は一致しやすいという特徴を持つ [3]．この strand の一致度合いから処理同士の類似度を算出し，類似度の最も高い組を類似した処理として検出する．そしてこの処理のうち一致している箇所を最長共通部分文字列を用いて抽出し，削除する．これにより strand の有する意味に基づいて差分を抽出する．

2.2 strand を用いた処理の類似度算出

提案手法では，プログラム中の複数の類似する処理に対して，それぞれの差分を検出する．このため，まず第一にプログラム中の処理の中から類似する処理を選択する必要がある．プログラムの類似度を算出する手法は多数提案されているが，本研究では Yaniv David らの手法 [3], [5] を採用した．これはプログラムの命令列を LLVM IR に代表されるような中間表現に逆変換したうえで，最適化や一般化を行うことでコンパイルオプションやターゲットアーキテクチャから生じる差異を吸収する．そのうえで，Basic Block をデータフロー解析することで更に strand と呼ばれる単位に分割し，strand の一致度によって処理ごとの類似度を算出するというものである．中間表現への逆変換によって位置独立実行コード化などによって生成される相対アドレスを参照する処理を同一の表現に集約することができ，また分解した strand に対して一般化を行うことによって，処理の配置や呼び出しの文脈によるスタック位置や使用するレジスタの違いを修正することができる．

図 1 は Basic Block を strand に分解する例である [5]．図中上段にある Basic Block をデータフローに基づいて，rbx レジスタに関連する処理からなる strand である Strand 1 と r13 レジスタに関連する処理からなる strand である Strand 2 に分解している．ある類似する複数の処理においては，利用するレジスタやインデックスが前後の処理に基づいて異なる可能性があるが，strand に分解してから一般化を行うことで，処理が共通している部分を明らかにすることができる．

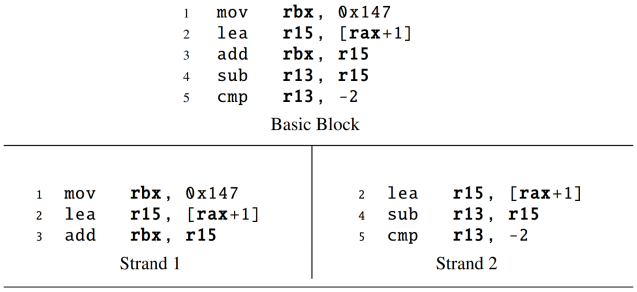

図 1 Basic Block の strand への分割 [5]
Fig. 1 Baisc Block and extracted strands [5]

strand を用いたある 2 つの処理同士の類似度 $S_{(P,R)}$ は [5] に基づき，次の式 1 で算出する．

$$S_{(P,R)} = \frac{|P \cap R|}{|P| + |R|} \tag{1}$$

ここで，P, R は比較対象となっている処理を strand に分割したものの集合であり，その大きさは含まれる strand の命令数の総和である．

2.3 最長共通部分文字列を用いた一致部分の削除

本節では，最長共通部分文字列アルゴリズム（longest common substring)[2], [13] を用いて 2 つの strands を比較し，その重複部分を削除することで差分を抽出する手法について述べる．ある配列に対して，その連続する一部分だけを取り出したものを部分文字列といい，ある 2 配列の最長共通部分文字列とはその 2 配列に共通する部分文字列であって，かつ最長のものを言う．2 配列 $seq1, seq2$ の長さをそれぞれ n, m として，およそ $\mathcal{O}((n+m)min(n,m))$ の時間計算量で $seq1, seq2$ の最長共通部分文字列を求めるアルゴリズム 1 が知られている．

提案手法では，前節のアルゴリズムを用いて類似していると判断した 2 つの strand に対して，その最長共通部分文字列を求め，その部分を取り除くことによって 2 つの strand の差分を明らかにする．例えば，図 2 は与えられた配列の要素の相加平均を算出するプログラムと，相乗平均を算出するプログラムの機械語列の比較である．図中に網掛けで示したとおり，この 2 つのプログラムは前処理や繰り返しの構造は共通しており，その内部で行われている処理と後処理のみが異なる．この 2 つの機械語列について長さが 2 以上の共通部分文字列が存在する間，このアルゴリズムを用いて最長共通部分文字列を削除し続けることでその差分を抽出することができる．

Algorithm 1 Longest Common Substring

Input: seq1, seq2
Output: longest common substrig of seq1 and seq1

```
1: max_length := 0
2: max_offset := 0
3: for offset = −|seq1| to |seq2| do
4:     current_length := 0
5:     for i = max(offset, 0) to min(|seq1|+offset, |seq2|) do
6:         if current_length > max_length then
7:             max_offset ← i − current_length − offset
8:             max_length ← current_length
9:         end if
10:        if seq1[i − offset] = seq2[i] then
11:            current_length ← current_length + 1
12:        else
13:            current_length ← 0
14:        end if
15:    end for
16: end for
17: return seq1[max_offset . . . (max_offset + max_length)]
```

```
f1:                                     f2:

push   rbp                              push   rbp
mov    rbp,rsp                          mov    rbp,rsp

                                        sub    rsp,0x20

mov    QWORD PTR [rbp-0x18],rdi         mov    QWORD PTR [rbp-0x18],rdi
mov    DWORD PTR [rbp-0x1c],esi         mov    DWORD PTR [rbp-0x1c],esi

mov    DWORD PTR [rbp-0x4],0x0          mov    DWORD PTR [rbp-0x4],0x1

mov    DWORD PTR [rbp-0x8],0x0          mov    DWORD PTR [rbp-0x8],0x0
jmp    7f8 <f1+0x38>                    jmp    84b <f2+0x42>
mov    eax,DWORD PTR [rbp-0x8]          mov    eax,DWORD PTR [rbp-0x8]
cdqe                                    cdqe
lea    rdx,[rax*4+0x0]                  lea    rdx,[rax*4+0x0]
mov    rax,QWORD PTR [rbp-0x18]         mov    rax,QWORD PTR [rbp-0x18]
add    rax,rdx                          add    rax,rdx
mov    eax,DWORD PTR [rax]              mov    eax,DWORD PTR [rax]

add    DWORD PTR [rbp-0x4],eax          mov    edx,DWORD PTR [rbp-0x4]
                                        imul   eax,edx
                                        mov    DWORD PTR [rbp-0x4],eax

add    DWORD PTR [rbp-0x8],0x1          add    DWORD PTR [rbp-0x8],0x1
mov    eax,DWORD PTR [rbp-0x8]          mov    eax,DWORD PTR [rbp-0x8]
cmp    eax,DWORD PTR [rbp-0x1c]         cmp    eax,DWORD PTR [rbp-0x1c]
jl     7db <f1+0x1b>                    jl     828 <f2+0x1f>

mov    eax,DWORD PTR [rbp-0x4]          pxor   xmm0,xmm0
cdq                                     cvtsi2sd xmm0,DWORD PTR [rbp-0x1c]
idiv   DWORD PTR [rbp-0x1c]             movsd  xmm1,QWORD PTR [rip+0x14c]
pop    rbp                              divsd  xmm1,xmm0
                                        pxor   xmm0,xmm0
                                        cvtsi2sd xmm0,DWORD PTR [rbp-0x4]
                                        call   650 <pow@plt>
                                        cvttsd2si eax,xmm0
                                        leave
ret                                     ret
```

図 2　相加平均（左）と相乗平均（右）の機械語列の比較
Fig. 2　The comparison of machine code of mean(left) and geometric mean(right).

3. 実験

本節では，提案手法の妥当性を示すために行った実験について述べる．

3.1 実験内容

VM では与えられた命令列から次に実行する命令をフェッチする処理や，命令列のデコード結果に基づいて命令を実行する処理，仮想的に実装されたメモリやレジスタに値を書き込む処理，プログラムカウンタの役割を担う変数をインクリメントする処理などが必要になる．VM の各命令に対応する機械語列ではこれらの処理が実装されおり，その大部分は共通すると考えられる．また，ある 2 つの命令を実装した機械語列を比較した時，その差分として実装された命令の差が現れると考えられる．本研究では難読化 VM を模したプログラムを作成し，このプログラムに対して提案手法を適用することで，難読化 VM の持つ命令にどのようなものが存在するかを明らかにする実験を行った．

表 1　難読化 VM の構成
Table 1　Propeties of obfuscation VM

命令数	25
命令長	可変長
計算モデル	レジスタ型
記述言語	C
コンパイラ	gcc version 4.9.4
ターゲット	x86_64-linux-gnu
最適化オプション	-O0

3.2 難読化 VM

表 1 に難読化 VM の構成を示す．命令としてはレジスタ同士，またはレジスタと即値をオペランドにとる算術演算，論理演算の他，メモリ操作系の命令，ジャンプ命令，条件つきジャンプ命令を実装した．表 2 に VM に実装した命令の一覧を示す．

3.3 提案手法の実装

提案手法の実装を行った．実装したものは，3.2 節の VM の実行可能ファイルと VM 本体に当たる関数のアドレスを受け取り，2.2 節の内容に基づいて strand に分割したものを LLVM IR 形式で出力する C++製のプログラムと，その出力を受け取って，2.2 節の内容に基づいて差分を表示する Python のスクリプトである．用いる中間言語の選択肢として，他には VEX-IR や BAP IR が存在したが，逆変換プログラムの精度や，既に最適化や一般化のためのツールが存在することから LLVM IR を採用した．また機械語命令列から LLVM IR への逆変換には retdec[9] を利用した．

3.4 実験結果

表 3 に実験結果を示す．VM のそれぞれの命令がどの命令と類似していると判断されたか，差分をとることで命令数をどの程度削減できたかを示している．類似する命令がないと判断された場合，すなわち式 1 で算出した類似度が 0 より大きくなるような命令が存在しなかった場合には横線で示した．算術，論理，比較演算命令に関しては同様の演算命令と一致していると判断され，差分が得られている一方，メモリに関する演算を行う"STORE REG1, REG2"や，ジャンプ系の命令"JMP IMM", "JIF REG,IMM"は類似する

表 2　VM の命令一覧
Table 2　VM instruction list

命令	命令長
NOP	1
ADD REG1,REG2,REG3	4
ADD REG1,REG2,IMM	7
SUB REG1,REG2,REG3	4
SUB REG1,REG2,IMM	7
MUL REG1,REG2,REG3	4
MUL REG1,REG2,IMM	7
DIV REG1,REG2,REG3	4
DIV REG1,REG2,IMM	7
AND REG1,REG2,REG3	4
OR REG1,REG2,REG3	4
XOR REG1,REG2,REG3	4
NOT REG1,REG2	3
STORE REG1,REG2	3
LOAD REG1,REG2	3
MOV REG1,REG2	3
MOV REG1,IMM	6
LT REG1,REG2,REG3	4
LE REG1,REG2,REG3	4
GT REG1,REG2,REG3	4
GE REG1,REG2,REG3	4
EQ REG1,REG2,REG3	4
NEQ REG1,REG2,REG3	4
JMP IMM	2
JIF REG,IMM	3

```
+ %143 = add i32 %140, %142
+ %146 = add i32 %144, %145
+ store i32 %197, i32* %200
—
- %143 = sub i32 %140, %142
- %146 = sub i32 %144, %145
- store i32 %197, i32* %200
```

図 3　ADD 命令と SUB 命令の差分
Fig. 3　Difference between ADD and SUB instruction.

命令が存在しないと判断されている．

図 3 は典型的な結果である，"ADD REG1, REG2, REG3"を実装した機械語列と類似する機械語列"SUB REG1, REG2, REG3"との差分である．一方が ADD 命令，もう一方は SUB 命令を実装していることが差分から読み取れる．

一方，"ADD REG1, REG2, IMM"に対しては"SUB REG1, REG2, IMM"との差分が取られ，その結果は図 4 のとおりであった．値が VM 上でレジスタ由来のものか即値由来のものかという情報は，それぞれの命令では共通するため削除されている．この他にも論理演算や比較演算の命令について，それぞれ類似する命令との差分が得られた．

表 3 では，"DIV REG1, REG2, REG3"は同じく除算を実装している"DIV REG1, REG2, IMM"と類似していると判断されている．これは除算の処理の共通部分が同じオペランドを持つ別の命令との共通部分よりも大きかった結果であると考えられる．この 2 命令の差分を図 5 に示した．除算を行っている部分は共通しているため削除されて現れないが，即値として現れている 4，7 が命令長に由来していることが読み取れる．

表 3　類似する命令とその差分
Table 3　similar instructions and difference

命令	類似する命令	命令数	差分
NOP	-	-	-
ADD REG1,REG2,REG3	SUB	294	3
ADD REG1,REG2,IMM	SUB	284	3
SUB REG1,REG2,REG3	ADD	294	3
SUB REG1,REG2,IMM	ADD	284	3
MUL REG1,REG2,REG3	ADD	281	8
MUL REG1,REG2,IMM	ADD	271	8
DIV REG1,REG2,REG3	DIV	285	21
DIV REG1,REG2,IMM	ADD	223	11
AND REG1,REG2,REG3	ADD	285	4
OR REG1,REG2,REG3	ADD	285	5
XOR REG1,REG2,REG3	ADD	285	4
NOT REG1,REG2	DIV	229	17
STORE REG1,REG2	-	-	-
LOAD REG1,REG2	NOT	206	115
MOV REG1,REG2	NOT	193	3
MOV REG1,IMM	-	-	-
LT REG1,REG2,REG3	LE	16	2
LE REG1,REG2,REG3	LT	20	5
GT REG1,REG2,REG3	LT	22	7
GE REG1,REG2,REG3	LT	18	3
EQ REG1,REG2,REG3	LT	16	3
NEQ REG1,REG2,REG3	LT	18	5
JMP IMM	-	-	-
JIF REG,IMM	-	-	-

```
+ %137 = add i32 %134, %136
+ %140 = add i32 %138, %139
+ store i32 %191, i32* %194
—
- %137 = sub i32 %134, %136
- %140 = sub i32 %138, %139
- store i32 %191, i32* %194
```

図 4　即値の ADD 命令と SUB 命令の差分
Fig. 4　Difference between ADD immediate and SUB immediate instruction.

"LOAD REG1,REG2"命令は同じく 2 オペランドを持つ"NOT REG1,REG2"命令と類似していると判断されたが，strand の半分程度は異なった内容となっており，意味のある差分は得られなかった．算出した類似度に対して閾値を設けることでこのような結果となることは回避できると考えられるが，適切な閾値は取り扱うプログラムによっても変化すると予想されるため，本研究では扱っていない．

```
+ %87 = add i32 %86, 3
+ %89 = add i32 %88, 3
+ %93 = xor i32 3, %87
+ store volatile i64 4119, i64* @0
+ %105 = add i32 %104, 3
+ %107 = add i32 %106, 3
+ %111 = xor i32 3, %105
+ store volatile i64 4128, i64* @0
+ store volatile i64 4130, i64* @0
+ store volatile i64 4132, i64* @0
+ store volatile i64 4134, i64* @0
+ store volatile i64 4136, i64* @0
+ %158 = add i32 %157, 4
+ %160 = add i32 %159, 4
+ %164 = xor i32 4, %158
+ store volatile i64 4139, i64* @0
+ %176 = add i32 %175, 4
+ %178 = add i32 %177, 4
+ %182 = xor i32 4, %176
+ store volatile i64 4142, i64* @0
+ store i32 %193, i32* %196
—
- store volatile i64 4118, i64* @0
- store volatile i64 4120, i64* @0
- %116 = add i32 %115, 7
- %118 = add i32 %117, 7
- %122 = xor i32 7, %116
- store volatile i64 4129, i64* @0
- %134 = add i32 %133, 7
- %136 = add i32 %135, 7
- %140 = xor i32 7, %134
- store volatile i64 4132, i64* @0
- store i32 %151, i32* %154
```

図 5 レジスタ同士の DIV 命令と即値との DIV 命令の差分
Fig. 5 Difference between DIV register and DIV immediate instruction.

3.5 考察

前節までの実験の結果から，提案手法は目的のとおりにプログラムの類似する処理の間の差分を抽出できていることが明らかとなった．図 3 と図 4 の結果のように，共通する処理の内容が変わっても同じ差分が得られていることから strand への分割，また共通部分文字列の削除というアプローチの有効性が伺える．この実験結果のみを用いて難読化 VM の解析を行うことや，既知の脆弱性と類似した処理が実際にその脆弱性を有しているかという判断をすることは難しいが，これらの処理を自動化するにあたって有効な手がかりになり得る．

4. 関連研究

本研究で用いたバイナリの類似度を算出する手法は，既知の脆弱性の検出や悪意あるソフトウェアの動作の検出といった用途で既に用いられており，様々な手法は存在する．また実験でとりあえげた難読化 VM の解析についてはこれまで本研究と同様のアプローチは存在しなかったが，自動的に解析あるいは難読化解除を行う手法が提案されている．

4.1 バイナリの類似度算出

何をもって類似しているかや，どの程度類似しているかといった適切な指標はその利用目的によって異なる．完全一致している一部分のコードの発見には Smith-Waterman の手法 [12] や，それを拡張した Needleman-Wunsch の手法 [10] が用いられる．一方，処理の大部分が一致しているかどうかなどを見る場合には処理間のギャップをスコアとして計上して類似度を算出する手法が存在している [15]．また，これらの手法を用いて，バイナリを構造的に比較する手法 [6] や制御フローグラフと組み合わせて効率的に類似箇所を検出する手法 [14] が提案されている．

4.2 難読化 VM の解析

VM 難読化に対するアプローチは，難読化 VM 自体を解析する手法の他に，シンボリック実行や tainting を用いて，難読化 VM の影響を回避して難読化前のプログラムを復元する手法 [11] が存在しており，VM 難読化による難読化を施されたプログラムに対しての難読化解除のアプローチとしては現状最も効果的とされている．VM 自体を解析する手法としては，[1] のような意味論的アプローチの他に VM に特有の VPC などの構造を静的解析で解析することによって明らかにする手法 [8] が存在する．

5. まとめと今後の課題

類似する機械語列を内包する複数の処理に対して，データの依存に基づく strand に分割し，strand の共通部分文字列を削除することによって差分を表示する手法を提案し，VM に対して手法を適用することで有効性を検証した．今後の課題としては，提案手法を応用してバージョンの異なるバイナリの更新箇所を明らかにするシステムや，VM の自動的な解析システムの構築，また LLVM IR に変換した strand として表示されている差分を元の機械語の差分として表示するための試みなどがある．

参考文献

[1] Coogan, K., Lu, G. and Debray, S.: Deobfuscation of Virtualization-Obfuscated Software A Semantics-Based Approach, pp. 275–284 (online), DOI: 10.1145/2046707.2046739 (2011).

[2] Crochemore, M., Iliopoulos, C. S., Langiu, A. and Mignosi, F.: The longest common substring problem, *Mathematical Structures in Computer Science*, Vol. FirstView, pp. 1–19 (online), DOI: 10.1017/S0960129515000110 (2015).

[3] David, Y., Partush, N. and Yahav, E.: Statistical Similarity of Binaries, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, Association for Computing Machinery, p. 266–280 (online), DOI: 10.1145/2908080.2908126 (2016).

[4] David, Y., Partush, N. and Yahav, E.: Similarity of Binaries through Re-Optimization, *Proceedings of the 38th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, New York, NY, USA, Association for Computing Machinery, p. 79–94 (online), DOI: 10.1145/3062341.3062387 (2017).

[5] David, Y., Partush, N. and Yahav, E.: Similarity of Binaries through Re-Optimization, *SIGPLAN Not.*, Vol. 52, No. 6, p. 79–94 (online), DOI: 10.1145/3140587.3062387 (2017).

[6] Flake, H.: Structural Comparison of Executable Objects, *In Proceedings of the IEEE Conference on Detection of Intrusions and Malware Vulnerability Assessment (DIMVA*, pp. 161–173 (2004).

[7] Haq, I. U. and Caballero, J.: A Survey of Binary Code Similarity, *ArXiv*, Vol. abs/1909.11424 (2019).

[8] Kinder, J.: Towards Static Analysis of Virtualization-Obfuscated Binaries, *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, USA, IEEE Computer Society, p. 61–70 (online), DOI: 10.1109/WCRE.2012.16 (2012).

[9] Křoustek, J.: Retargetable Analysis of Machine Code, Ph.d. thesis, Brno University of Technology, Faculty of Information Technology (2015).

[10] Needleman, S. B. and Wunsch, C. D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins, *Journal of Molecular Biology*, Vol. 48, No. 3, pp. 443 – 453 (online), DOI: https://doi.org/10.1016/0022-2836(70)90057-4 (1970).

[11] Salwan, J., Bardin, S. and Potet, M.-L.: Symbolic Deobfuscation: From Virtualized Code Back to the Original, *Detection of Intrusions and Malware, and Vulnerability Assessment* (Giuffrida, C., Bardin, S. and Blanc, G., eds.), Cham, Springer International Publishing, pp. 372–392 (2018).

[12] Smith, T. and Waterman, M.: Identification of common molecular subsequences, *Journal of Molecular Biology*, Vol. 147, No. 1, pp. 195 – 197 (online), DOI: https://doi.org/10.1016/0022-2836(81)90087-5 (1981).

[13] Starikovskaya, T. and Vildhøj, H. W.: Time-Space Trade-Offs for the Longest Common Substring Problem, *Combinatorial Pattern Matching* (Fischer, J. and Sanders, P., eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 223–234 (2013).

[14] 羽田大樹，後藤厚宏：BinGrep: 制御フローグラフの比較を用いた関数の検索によるマルウェア解析の効率化の提案，技術報告 5，情報セキュリティ大学院大学／ NTT コムセキュリティ株式会社, 情報セキュリティ大学院大学 (2016).

[15] 中島明日香，岩村　誠，矢田　健：機械語命令の類似度算出による複製された脆弱性の発見手法の提案，コンピュータセキュリティシンポジウム 2015 論文集，Vol. 2015, No. 3, pp. 304–309 (2015).