

# 目 次

第 1 章 序論	1
第 2 章 マルウェアの解析手法	2
2.1 マルウェア解析の目的 . . . . .	2
2.2 マルウェア解析の種類 . . . . .	2
2.3 FFRI Dataset . . . . .	3
第 3 章 圧縮アルゴリズム LZT	7
3.1 LZT の系譜 . . . . .	7
3.1.1 LZ77 . . . . .	7
3.1.2 LZ78 . . . . .	7
3.2 LZW の概要 . . . . .	8
3.3 LZW による圧縮の手順 . . . . .	9
3.4 LZT の概要 . . . . .	10
3.5 LZT による圧縮の手順 . . . . .	10
第 4 章 提案手法と実験	12
4.1 マルウェア亜種分類手法の提案 . . . . .	12
4.2 API コール列の可視化 . . . . .	13
4.2.1 Graphviz . . . . .	13
4.2.2 API コール列の繰り返しの特徴 . . . . .	14
4.3 実験結果 . . . . .	18
第 5 章 結論	24
謝辞	25
参考文献	26



# 第1章 序論

近年、様々な攻撃対象に向けてのマルウェアが増加傾向にあり、McAfee の脅威レポート<sup>[1]</sup>からも、2021年第1期だけでも 87.6 百万もの新種のマルウェアが発見されている。マルウェアにコンピュータが感染すると、個人情報の流出によってサイバー犯罪の被害者になるだけでなく、本人の気付かない間に感染したコンピュータが不正アクセスや Dos(Denial of Service) 攻撃などに悪用され、サイバー犯罪の加害者側になる恐れもある。その被害事例として、2015 年に東京大学が管理する業務用 PC がマルウェアに感染し、不正アクセスによって情報の流出被害が確認されている。流出した個人情報は 3 万 6300 件に及ぶと報告されている<sup>[2]</sup>。被害者が攻撃者になってしまう例としては、2016 年にアメリカで DNS サービスを提供する Dyn 社が、IoT 機器に感染するマルウェアによる大規模な DDos 攻撃を受け、Twitter や Amazon、PayPal、Netflix などの Dyn 社のサービスを利用していた企業でのサービスに不具合が起こるという事例がある<sup>[3]</sup>。

マルウェアの種類は膨大であるが、その大半はマルウェアを一部改変した亜種である。そのため、膨大な量のマルウェアを解析する際に、亜種であるマルウェアを効率的に自動で分類することによって、セキュリティ対策として非常に有効であると考えられる。マルウェアの解析には大別すると、表層解析、動的解析、静的解析の 3 つの手法がある。本研究では動的解析によって得られた API コールから特徴を見出すことによってマルウェア亜種分類を行う手法を提案する。具体的には、マルウェアが呼び出した API コールを辞書式圧縮アルゴリズム LZT によって圧縮し、その圧縮率を特徴量として用いる。分類は教師あり学習モデルの 1 つである SVM(Support Vector Machine) によって行い、正解率を求めることで、提案したマルウェア亜種分類手法の有効性を評価する。

本稿では、2 章でマルウェアの解析手法について説明する。3 章では本研究で用いる圧縮アルゴリズムである LZT について説明する。4 章では本研究の提案手法と実験手順と結果を説明し、最後に 5 章で本論文の結論について述べる。

## 第2章 マルウェアの解析手法

本章では、マルウェアの解析手法について説明する。2.1節でマルウェア解析の目的について説明する。2.2節でマルウェア解析の種類について説明する。2.3節で本研究で用いた動的解析結果であるFFRI Datasetについて説明する。

### 2.1 マルウェア解析の目的

マルウェア解析とは、マルウェアに内在している情報を得るための行為である。例えば、マルウェアが備えている機能を明らかにすることや、マルウェアが作成された目的を明らかにするために行われている。

### 2.2 マルウェア解析の種類

マルウェアの解析手法<sup>[4]</sup>には、いくつかの解析プロセスが存在する。

1つ目は、表層解析である。表層解析ではマルウェアの表層的な特徴（ファイル名、ファイル種別、ハッシュ値等）を元に、インターネットや過去の解析データ等を調査する方法である。表層解析では、解析対象に対してツールを使用して情報を取得する。短時間に解析結果を取得できることや、解析者に要求されるスキルレベルが高度なものではないという長所がある。しかしながら、得られる情報が限定的であることや難読化されたマルウェアからは十分な解析結果を得ることが難しいという短所がある。

2つ目は動的解析（ブラックボックス解析）である。動的解析では実際にマルウェアを動作させ、その挙動をモニタリングする手法である。実際の挙動から対策や復旧に有用な情報が得られる。短時間で解析結果を取得できることや難読化されたマルウェアからも解析結果を取得できることなどの長所がある。しかしながら、動的解析を行うためにはマルウェアが必要である点、マルウェアを実行するための安全な解析環境を構築する必要がある点、解析時に実行されなかったコードの振る舞いは分からぬ点が短所である。動的解析では、プログラムコードを詳細に分析しないためブラックボックス解析とも呼ばれる。

3つ目は静的解析(ホワイトボックス解析)である。静的解析はマルウェアのファイルをリバースエンジニアリングし、プログラムのコードを読み解く手法である。マルウェアを動作させる必要がないため、動的解析で実行されないコードの動作を把握できることやマルウェアに備わっている機能の詳細なアルゴリズムを解明することができるという長所がある。しかしながら、実際のアセンブリを読み解いて解明するため、CPU アーキテクチャなどに対する解析者の専門性と、非常に時間がかかることが短所であると言える。静的解析では、プログラムコードを詳細に分析するためホワイトボックス解析とも呼ばれる。

### 2.3 FFRI Dataset

本研究ではマルウェア対策研究人材育成ワークショップ 2020(MWS2020)<sup>[5]</sup> が提供しているデータセットのうち FFRI 社が独自に収集したマルウェアの動的解析結果である FFRI Dataset2013～2017 を用いる<sup>[6]</sup>。このデータセットは FFRI 社が独自に収集したマルウェアの動的解析ログであり、FFRI Dataset2013 には 2644 検体、FFRI Dataset2014,2015 には 3000 検体、FFRI Dataset2016 には 8242 検体、FFRI Dataset2017 には 6251 検体の情報がそれぞれ含まれている。FFRI Dataset は、Cuckoo Sandbox というオープンソース(一部非公開)のマルウェア解析システムを利用している。1 検体あたり 90 秒間実行しており、実行した解析結果のログファイルは json 形式で保存されている<sup>[7]</sup>。

FFRI Dataset の項目とその内容を表 2.1 に示す。本研究では、virustotal 内に記載されている Kaspersky による命名規則に則ったマルウェアファミリ名と behavior 内に記載されている API 名を使用する。また、亜種の数が 100 検体以上あるファミリを対象として、他のマルウェアを意味するマルウェアファミリ名 “Torojan.Win32.Generic”、”DangerousObject.Multi.Generic”、”Torojan.Script.Generic” を除くこととする。以上の条件で選択されたマルウェアファミリと、その検体数を表 2.2 に示す。

表 2.1 FFRI Dataset の項目とその内容

項目	内容
info	解析の開始、終了時刻、id 等
signatures	ユーザ定義シグニチャとの照合結果
virustotal	VirusTotal の検査履歴との照合結果
static	検体のファイル情報(インポート API、セクション構造等)
dropped	検体が実行時に生成したファイル
behavior	検体実行時の API ログ
processstree	検体実行時のプロセスツリー(親子関係)
summary	検体が実行時にアクセスしたファイル、レジストリなどの概要情報
target	解析対象検体のファイル情報(ハッシュ値等)
debug	検体解析時の Cuckoo Sandbox のデバッグログ
strings	検体中に含まれる文字列情報
network	検体が実行時に行った通信の概要情報

表 2.2 マルウェアファミリ名と検体数

No.	マルウェアファミリ名	検体数
1	Bakdoor.Win32.Androm	559
2	Bakdoor.Wi32.DarkKomet	683
3	Bakdoor.Win32.Matsnu	110
4	Downloader.Win32.LMN	132
5	Hoax.Win32.Archsms	305
6	Packed.Win32.Tpyn	309
7	Trojan.Win32.Agent	460
8	Trojan.Win32.Agentb	175
9	Trojan.Win32.Inject	538
10	Trojan.Win32.Jorik	172
11	Trojan.Win32.Kovter	174
12	Trojan.Win32.Llac	104
13	Trojan.Win32.Poweliks	114
14	Trojan.Win32.Reconyc	110
15	Trojan.Win32.Scar	131
16	Trojan.Win32.Waldek	688
17	Trojan.Win32.Yakes	618
18	Trojan-Banker.Win32.Banbra	287
19	Trojan-Banker.Win32.Shifu	103
20	Trojan-Downloader.Win32.Upatre	117

表は次ページに続く

前ページからの続き

No.	マルウェアファミリ名	検体数
21	Trojan-Downloader.Win32.Vb	118
22	Trojan-Dropper.Win32.Injector	189
23	Trojan-Psw.Win32.Fareit	326
24	Trojan-Psw.Win32.Tepfer	249
25	Trojan-Ransom.Win32.Blocker	633
26	Trojan-Ransom.Win32.Foreign	228
27	Trojan-Ransom.Win32.Locky	124
28	Trojan-Ransom.Win32.Zerber	702
29	Trojan-Spy.Win32.Zbot	635
30	Virus.Win32.Parite	139
31	Worm.Win32.Vobfus	120
32	Worm.Win32.Wbna	309

以上

## 第3章 圧縮アルゴリズム LZT

本章では本研究で用いたデータ圧縮アルゴリズム LZT について説明する。3.1 節で LZT に関するアルゴリズムである LZ77 と LZ78 について説明する。3.2 節で LZW の概要について説明し、3.3 節で LZW による圧縮のアルゴリズムについて説明する。3.4 節で LZT の概要について説明し、3.5 節で LZT による圧縮のアルゴリズムについて説明する。

### 3.1 LZT の系譜

この節では、辞書を用いたデータ圧縮アルゴリズムについて述べる。LZ77 と LZ78 はそれぞれ Ziv と Lempel が 1977 年と 1978 年に提案した可逆圧縮法である<sup>[8]</sup>。本研究で使用する LZT は LZ78 のバリエーションの 1 つに含まれる。3.1.1 節では LZ77 について説明し、3.1.2 節では LZT の土台となっている LZ78 について説明する。

#### 3.1.1 LZ77

LZ77 について述べる<sup>[8],[9]</sup>。LZ77 は入力された記号列をバッファに格納し、それを辞書として利用する。バッファの大きさは有限のため、新しく記号列を読み込んだ分だけ古い記号列を削除する必要がある。この動作がバッファがスライドしていくように見えることから、このバッファは「スライド窓」と呼ばれている。LZ77 は符号化する際に既に入力されている記号列から符号化部と最も長く一致する部分列（最長一致系列）を探す。しかし、復号する際には最長一致系列の探索を行う必要が無い。このことから、LZ77 は符号化には時間をするが復号は高速に行えるデータ圧縮アルゴリズムである。

#### 3.1.2 LZ78

LZ78 について述べる<sup>[8],[10]</sup>。LZ78 は LZ77 の辞書の局所性という問題を回避するために作成された。LZ78 符号は、辞書としてスライド窓を使うのをやめ、これまでに入力された記号列の一覧表を辞書とすることで、大域的な辞書を作成する。また、入力された記号列を

増分分解と呼ばれる方法で部分記号列に分解し、得られた部分記号列を辞書に登録し、この辞書を利用することで符号化を行っている。LZ77 符号の問題点を解決しただけではなく、圧縮/展開が速く、プログラム化が容易であるという特徴から現在のところ最も一般的な圧縮手法であるといえる。

### 3.2 LZW の概要

LZWについて述べる<sup>[8]</sup>。LZWはLZ78の改良の一つで、1984年にWelchらによって提案された辞書式圧縮アルゴリズムである。LZWの前身であるLZ78は辞書の作成や管理の手法に難点があったため、それを改善するように開発された。LZWには1記号からなる語をすべて辞書に前もって登録しておくことによって、符号語中に含まれる記号を取り除き、常にポインタで済ましている。LZ78とLZWの符号語に対応する記号列と辞書に登録される記号列との違いをそれぞれ図3.1、図3.2に示す。LZ78では、符号語に対応する記号列が必ず辞書に登録されているが、LZWでは、符号語に対応する記号列の最後に、次に符号化される1記号を続けたものが辞書に登録されている。以上に述べた仕組みにすることにより、LZW符号では符号語中に直接記号が含まれない仕組みになっている。

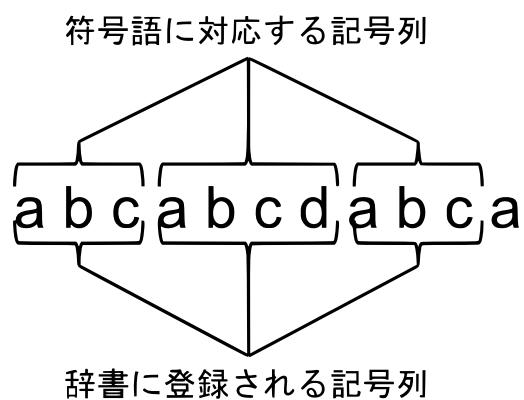


図 3.1 LZ78における入力列の分解

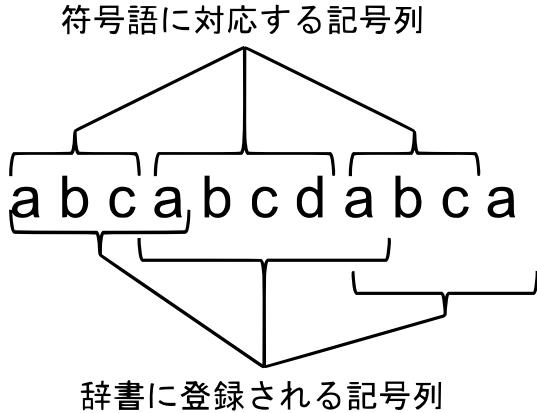


図 3.2 LZW における入力列の分解

### 3.3 LZW による圧縮の手順

LZW のアルゴリズムを次に示す<sup>[8],[11]</sup>。

#### (1) 初期化

まず初めに、文字列に出現するすべての単語を辞書に登録する。本実験では API コール列に出現するアルファベット 52 種類(大文字、小文字)、0 ~ 9 の数字 10 種類、”\_”、”“ の合計 64 単語を登録する。また、現在符号化を行っている単語の位置を示すポインタ  $i$  を

$$i \leftarrow 1$$

に設定する。さらに、登録されている語の数を示す変数を  $j$  とする。

#### (2) $j$ 番目の出力

$i$  番目の記号から始まる記号列に対して、最長一致系列を辞書から探す。出力される符号語は、最長一致系列の参照番号  $R_j$  を出力する。次に、最長一致系列の長さを  $l$  としたとき、位置を表すポインタ  $i$  を

$$i \leftarrow i + l + 1$$

によって更新する。

## (3) 辞書の更新

$j + 1$  番目の辞書に、出力した  $R_j$  に次の 1 単語をつなげたものを登録する。次に、辞書に登録してある数を表す変数  $j$  を

$$j \leftarrow j + 1$$

によって更新した後、(2) に戻る。

### 3.4 LZT の概要

本論文で用いる LZT について述べる [8], [12], [13]。LZT は 1987 年に Tischer によって開発された圧縮アルゴリズムであり、LZW と同様に圧縮の際に辞書を用いている。LZW の圧縮アルゴリズムに加えてこのアルゴリズムの主な改良点は、辞書が満杯になったときに最も長い間使われていない（最長時間未使用：Least Recently Used）語を取り除くことによって辞書の空きスペースを確保することである。これによって符号化と復号に時間がかかることになるが、不必要的語が辞書から外されるため、データの局所的な偏在も辞書に反映することができる。

### 3.5 LZT による圧縮の手順

LZT のアルゴリズムを次に示す [8], [12], [13]。

## (1) 初期化

LZW と同様に、文字列に出現するすべての単語を辞書に登録する。また、現在符号化を行っている単語の位置を示すポインタ  $i$  を

$$i \leftarrow 1$$

に設定する。さらに、登録されている語の数を示す変数  $j$  とする。

(2)  $j$  番目の出力

LZW と同様に  $i$  番目の記号から始まる記号列に対して、最長一致系列を辞書から探す。出力される符号語は、最長一致系列の参照番号  $R_j$  を出力する。本実験では圧縮した結

果を出力しないが、代わりに辞書を出力する。次に、最長一致系列の長さを  $l$  としたとき、位置を表すポインタ  $i$  を

$$i \leftarrow i + l + 1$$

によって更新する。

### (3) 辞書の更新

#### (a) 辞書が埋まっている場合

最も長い間参照されていない辞書を削除することで辞書に空きを作り、削除された辞書以降の参照番号をそれぞれ-1 することで再割り当てし、 $j+1$  番目の辞書に、出力した  $R_j$  に次の 1 単語をつなげたものを登録した後、(2) に戻る。

#### (b) 辞書が埋まっていない場合

LZW と同様に  $j+1$  番目の辞書に、出力した  $R_j$  に次の 1 単語をつなげたものを登録する。次に、辞書に登録してある数を表す変数  $j$  を

$$j \leftarrow j + 1$$

によって更新した後、(2) に戻る。

## 第4章 提案手法と実験

本章では、本実験で行ったマルウェア亜種分類の手法と実験結果を示す。4.1節でマルウェア亜種分類手法の提案について説明し、4.2節で、API コール列の可視化について説明し、4.3節で実験結果を示す。

### 4.1 マルウェア亜種分類手法の提案

本研究では LZT の辞書による圧縮率を用いたマルウェア亜種分類手法を提案する。まず、 $N$  個のマルウェアの動的解析結果  $D_1, D_2, \dots, D_N$  をデータセットとする。表 2.1 に示す動的解析結果  $D_n$  の項目 behavior から API コールを、項目 virustotal からマルウェアファミリ名を得る。これによって作成されたファミリ名ごとに、スペース区切りで API コールが羅列された「API コール列」を LZT の圧縮アルゴリズムによって圧縮し、辞書  $B$  を作成する。この時初期辞書として API コール列に使われている文字 ( $A, B, \dots, Z, a, b, \dots, z, 0, 1, \dots, 9, “_”, “”$ ) 64 種類を登録する。また、本研究では辞書の大きさは 12bit としている。

- (1) API コール列に使われている文字を初期辞書  $B_1 = A, B_2 = B, \dots, B_{63} = “_”, B_{64} = “”$  として登録する。
- (2) 本研究では機械学習の際に各ファミリの圧縮率を特徴ベクトルとして与える。そのため、API コール列で構成されるデータセットから特徴抽出用検体を選択する。ここで選択された特徴抽出用検体を、LZT の圧縮アルゴリズムに沿って辞書に登録する。例えば、文字列 “AC” については

$$“AC” = B_1 B_3$$

であり、辞書に登録されていなければ

$$B_{65} = AC$$

として、新たな辞書  $B_{65}$  に登録する。ここでは、辞書の作成を目的としているため、符号語は出力しない。

- (3) 特徴抽出用検体に選択されなかった検体について、(2)で作成した Lzt 辞書のみを用いて圧縮を実施する。これによって、特徴抽出用検体と類似した検体が与えられた場合はよく圧縮され、似ていない検体が与えられた場合はそれほど圧縮されないことで、機械学習においての局所性を得る。ここでの圧縮率の計算方法を以下に示す。ここでは文字列の長さが短い検体による影響を小さくすることを目的としてスムージングパラメータ  $\gamma$  を用いる。

$$\text{圧縮率} = \frac{\text{圧縮後のデータサイズ} + \gamma}{\text{元のデータサイズ} + \gamma}$$

- (4) 得られた特徴ベクトルを、対象ファミリであるか否かを示す教師データとともに、教師あり学習の SVM を用いて実験を行う。ファミリ毎に 4 分割交差検証を行い、その平均値をファミリの正解率とし、それぞれのファミリの正解率を平均した、平均正解率で評価する。

## 4.2 API コール列の可視化

本実験では、同じファミリの検体同士では API コール列中の API コールから API コールへの遷移に同じ繰り返しの特徴があると考え、繰り返し構造に着目した圧縮アルゴリズムである Lzt を用いてマルウェア亜種分類を行っている。そこで 4.2.1 節では、テキストデータをグラフ画像に変換し可視化することができるツールである「Graphviz」について説明し、4.2.2 節では、「Graphviz」で API コールの繰り返しを可視化することによって得られた、API コール列の特徴について説明する。

### 4.2.1 Graphviz

Graphviz(Graph Visualization Software) とは、AT&T 研究所が開発したオープンソースのグラフ視覚化ツールであり、DOT 言語で記述されたグラフ構造(ノードとエッジから成るネットワーク構造)を描画することができる<sup>[14]</sup>。DOT 言語とは、データ構造としてのグラフを表現するためのデータ記述言語の一種であり、コンピュータで処理しやすく、目で見ても分かり易い単純化された形式となっている。

#### 4.2.2 API コール列の繰り返しの特徴

本研究では同じファミリの検体同士では API コール列中の API コールから API コールへの遷移に同じ繰り返しの特徴があると考えている。そこで、同じファミリ同士の検体を可視化した場合を図 4.1、図 4.2 に示し、別のファミリの検体を可視化した場合を図 4.3、図 4.4 に示す。

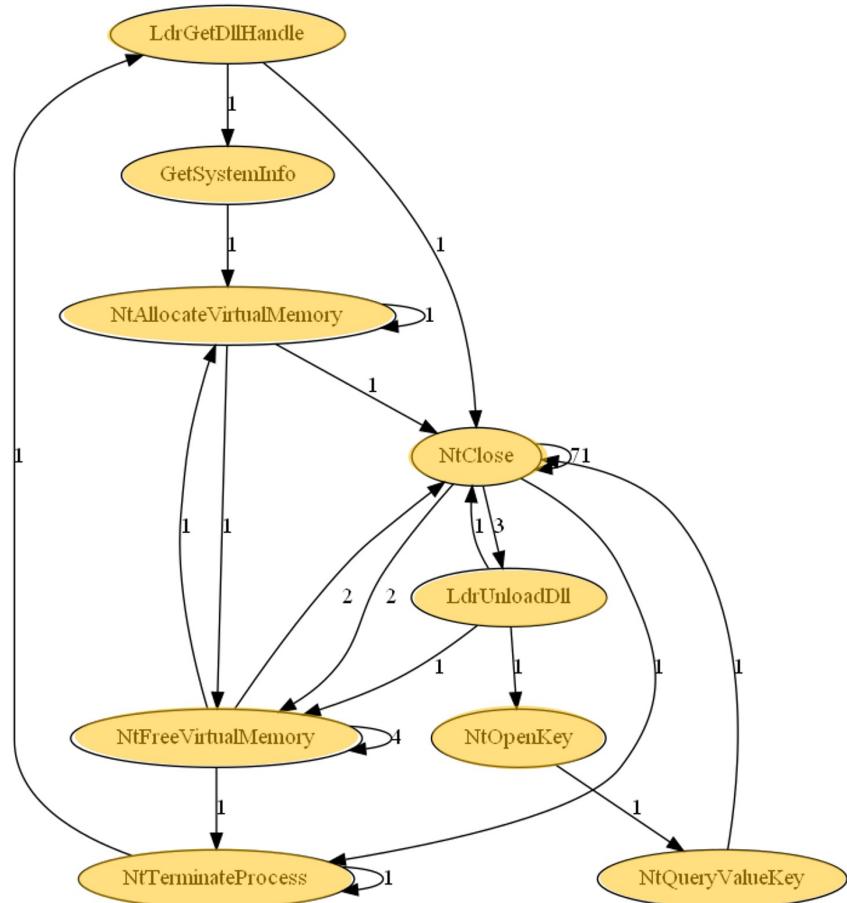


図 4.1 Trojan.Win32.Agent の検体 (1)

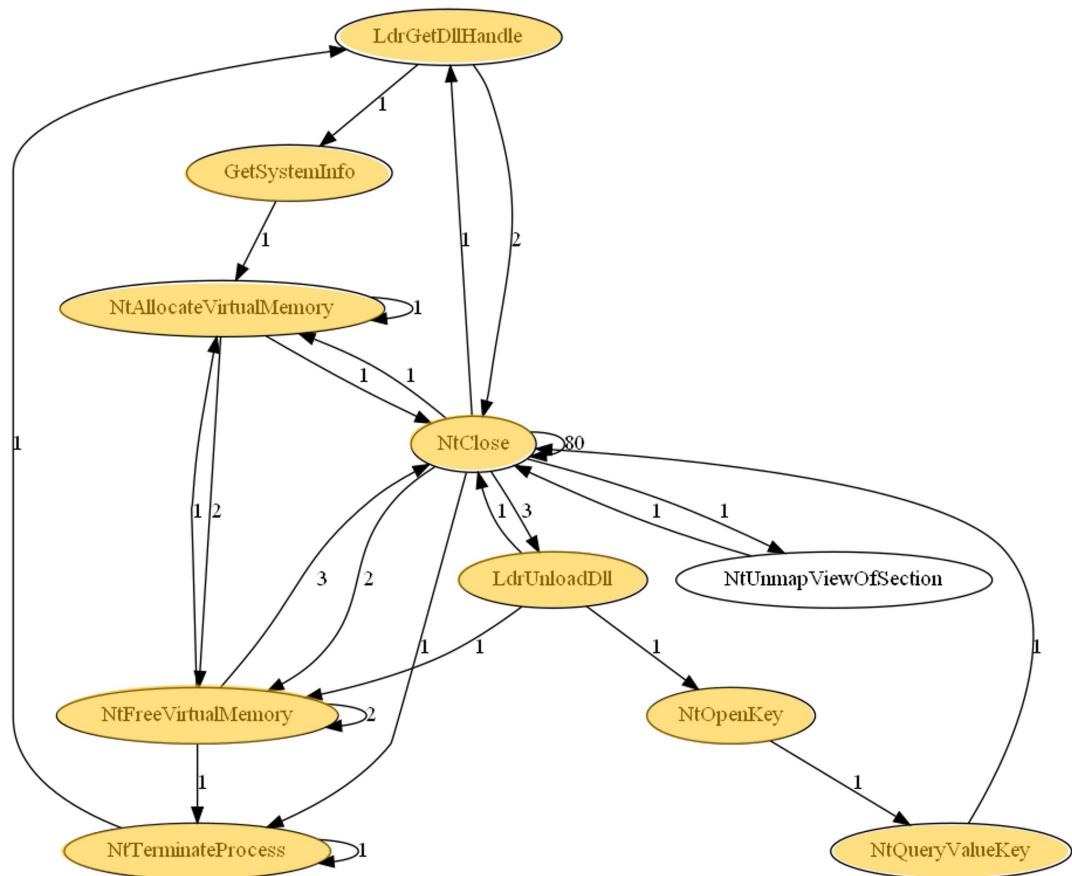


図 4.2 Trojan.Win32.Agent の検体 (2)

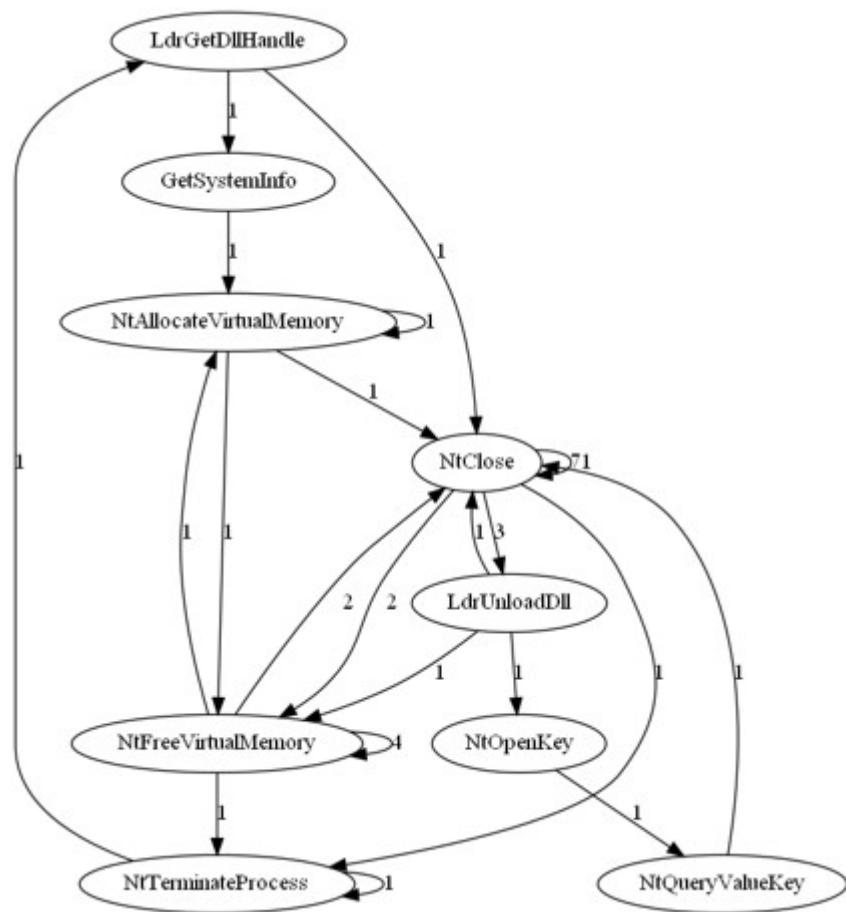


図 4.3 Trojan.Win32.Agent の検体 (1)

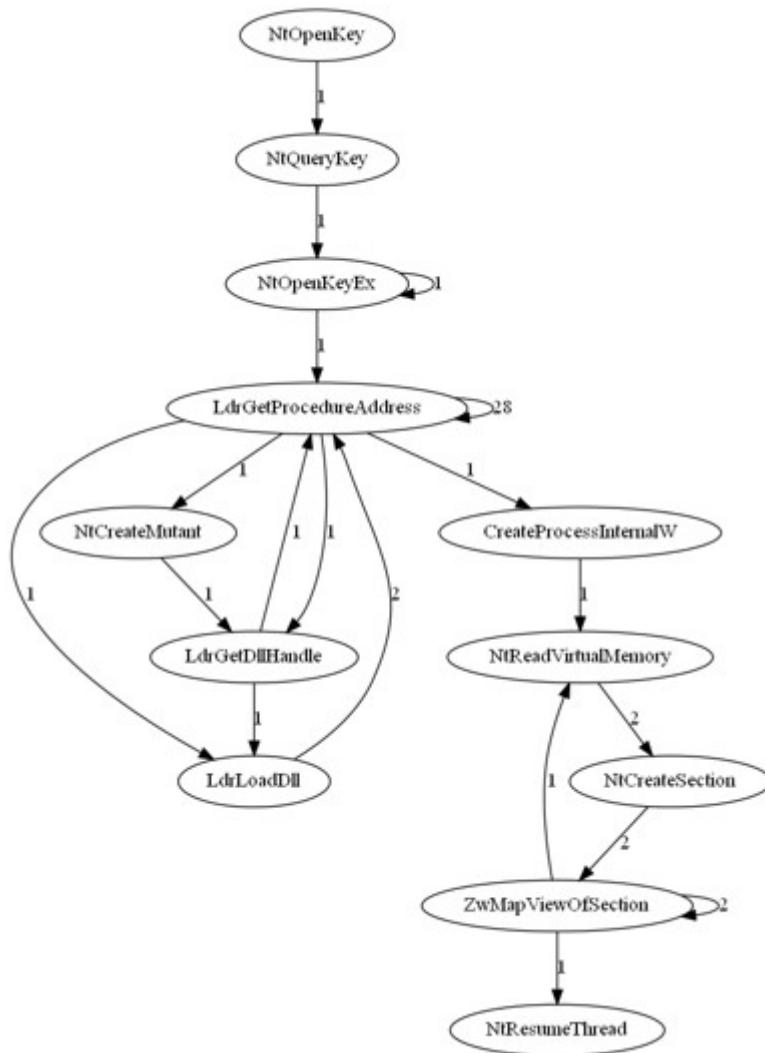


図 4.4 Backdoor.Win32.Androm の検体 (1)

図 4.1、図 4.2 はマルウェアファミリ Trojan.Win32.Agent の、ある 2 つの検体の API コールの遷移を表している。この図の例ではそれを比較したときに、色を付けている頂点が両グラフで共通していることを表しており、API コールから API コールへの遷移も同じ繰り返しの特徴を持っていることが確認できる。

図 4.3、図 4.4 は Trojan.Win32.Agent の図 4.1 に示した検体とマルウェアファミリ Backdoor.Win32.Androm のある検体の API コールの遷移を表している。この図の例ではそれを比較したときに、同じファミリの検体同士で比較した場合に比べて、呼び出されている API コールが異なっていることが確認できる。以上のことから、繰り返し構造に着目する LZT の辞書を用いて圧縮することで、同じマルウェアファミリの検体で作成された辞書と異なるマルウェアファミリの検体で作成された辞書では圧縮率に差が生じる。

### 4.3 実験結果

4分割交差検証により、LZTの辞書を用いたマルウェア亜種分類を検証した。以下の表4.1に検証4回分のそれぞれの正解率と平均正解率を示す。

表4.1 LZT辞書を用いたマルウェア亜種分類の実験結果

No.	マルウェアファミリ名	1回目	2回目	3回目	4回目	正解率
1	Bakdoor.Win32.Androm	82.2	79.92	82.58	78.24	80.74
2	Bakdoor.Wi32.DarkKomet	86.5	90.8	88.04	89.51	88.71
3	Bakdoor.Win32.Matsnu	91.67	91.3	89.13	89.13	90.31
4	Downloader.Win32.LMN	85.42	84.78	95.65	95.65	90.38
5	Hoax.Win32.Archsms	95	94.29	95	98.55	95.71
6	Packed.Win32.Tpyn	90	90	88.57	94.93	90.87
7	Trojan.Win32.Agent	73.39	73.39	77.31	75.93	75.01
8	Trojan.Win32.Agentb	84.62	87.18	87.18	85.53	86.13
9	Trojan.Win32.Inject	81.44	78.79	83.71	80.15	81.02
10	Trojan.Win32.Jorik	94.87	85.9	92.31	89.47	90.64
11	Trojan.Win32.Kovter	98.72	94.87	92.31	97.37	95.82
12	Trojan.Win32.Llac	68.75	86.96	82.61	84.78	80.77
13	Trojan.Win32.Poweliks	100	100	100	95.65	98.91
14	Trojan.Win32.Reconyc	83.33	78.26	84.78	82.61	82.25
15	Trojan.Win32.Scar	75	67.39	65.22	73.91	70.38
16	Trojan.Win32.Waldek	93.25	94.48	92.64	93.21	93.39
17	Trojan.Win32.Yakes	81.08	8.095	82.31	81.63	81.49
18	Trojan-Banker.Win32.Banbra	94.35	96.77	95.16	92.74	94.76
19	Trojan-Banker.Win32.Shifu	81.25	95.65	89.13	86.96	88.25
20	Trojan-Downloader.Win32.Upatre	83.33	84.78	84.78	82.61	83.88
21	Trojan-Downloader.Win32.Vb	85.42	86.96	93.48	95.65	90.38
22	Trojan-Dropper.Win32.Injector	80.77	78.21	76.92	85.53	80.36
23	Trojan-Psw.Win32.Fareit	85.9	79.49	83.12	86.36	83.72

表は次ページに続く

前ページからの続き

No.	マルウェアファミリ名	1回目	2回目	3回目	4回目	正解率
24	Trojan-Psw.Win32.Tepfer	80.91	76.85	77.78	75.93	77.87
25	Trojan-Ransom.Win32.Blocker	92.26	91.29	90	92.9	91.61
26	Trojan-Ransom.Win32.Foreign	80.91	80.56	78.7	75.93	79.02
27	Trojan-Ransom.Win32.Locky	95.83	84.78	70.74	78.26	82.65
28	Trojan-Ransom.Win32.Zerber	98.25	98.54	98.24	97.65	98.17
29	Trojan-Spy.Win32.Zbot	79.68	76.77	79.35	81.94	79.44
30	Virus.Win32.Parite	98.39	98.39	100	98.39	98.79
31	Worm.Win32.Vobfus	85.42	91.3	93.48	89.13	89.83
32	Worm.Win32.WBNA	92.86	93.57	94.29	94.93	93.91
平均正解率						<b>87.04</b>

以上

表4.1から、LZTの辞書を用いたマルウェア亜種分類の結果、平均87.04%の正解率となった。また、本研究と同じデータセットを用いてLZWの辞書を特徴としたParagraph Vectorを用いる関連研究<sup>[15]</sup>の正解率の平均値83.80%よりも高い正解率を確認できた。

マルウェアファミリ名“Trojan.Win32.Scar”は、他のマルウェアファミリに比べて比較的正解率が低いという結果が得られた。また、マルウェアファミリ名“Trojan.Win32.Poweliks”は、最も正解率が高いという結果が得られた。ここで、マルウェアファミリ名“Trojan.Win32.Scar”と“Trojan.Win32.Poweliks”的本実験で用いた特徴抽出用検体をそれぞれ図4.5、図4.6に示す。

```
"trojan.win32.scar": [
    "./Dataset/trojan.win32.scar/117.txt",
    "./Dataset/trojan.win32.scar/22.txt",
    "./Dataset/trojan.win32.scar/123.txt",
    "./Dataset/trojan.win32.scar/127.txt",
    "./Dataset/trojan.win32.scar/11.txt",
    "./Dataset/trojan.win32.scar/63.txt",
    "./Dataset/trojan.win32.scar/26.txt",
    "./Dataset/trojan.win32.scar/10.txt",
    "./Dataset/trojan.win32.scar/13.txt",
    "./Dataset/trojan.win32.scar/30.txt"
],
```

図4.5 “Trojan.Win32.Scar”的特徴抽出用検体

```
"trojan.win32.poweliks": [
    "./Dataset/trojan.win32.poweliks/20.txt",
    "./Dataset/trojan.win32.poweliks/89.txt",
    "./Dataset/trojan.win32.poweliks/80.txt",
    "./Dataset/trojan.win32.poweliks/76.txt",
    "./Dataset/trojan.win32.poweliks/3.txt",
    "./Dataset/trojan.win32.poweliks/10.txt",
    "./Dataset/trojan.win32.poweliks/7.txt",
    "./Dataset/trojan.win32.poweliks/56.txt",
    "./Dataset/trojan.win32.poweliks/107.txt",
    "./Dataset/trojan.win32.poweliks/18.txt"
],
```

図4.6 “Trojan.Win32.Poweliks”的特徴抽出用検体

それぞれのマルウェアファミリの特徴抽出用検体から無作為に2検体選び、選んだ検体を4.2.2節と同様にそれぞれ可視化してみる。“Trojan.Win32.Scar”の2検体を可視化したグラフをそれぞれ図4.7、図4.8、“Trojan.Win32.Poweliks”の検体を可視化したグラフをそれぞれ図4.9、図4.10に示す。

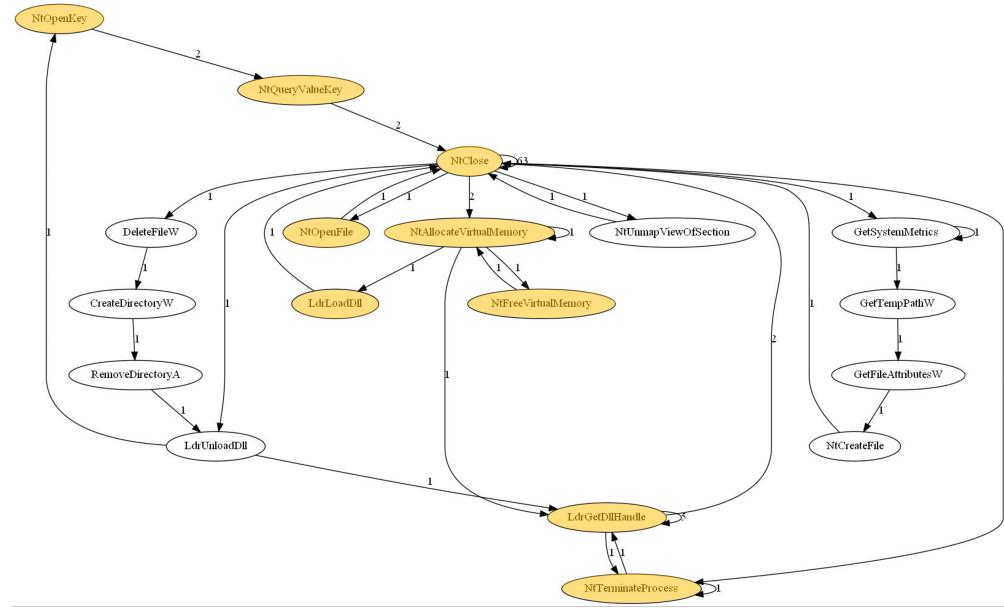


図 4.7 “Trojan.Win32.Scar” の特徴抽出用検体(1)

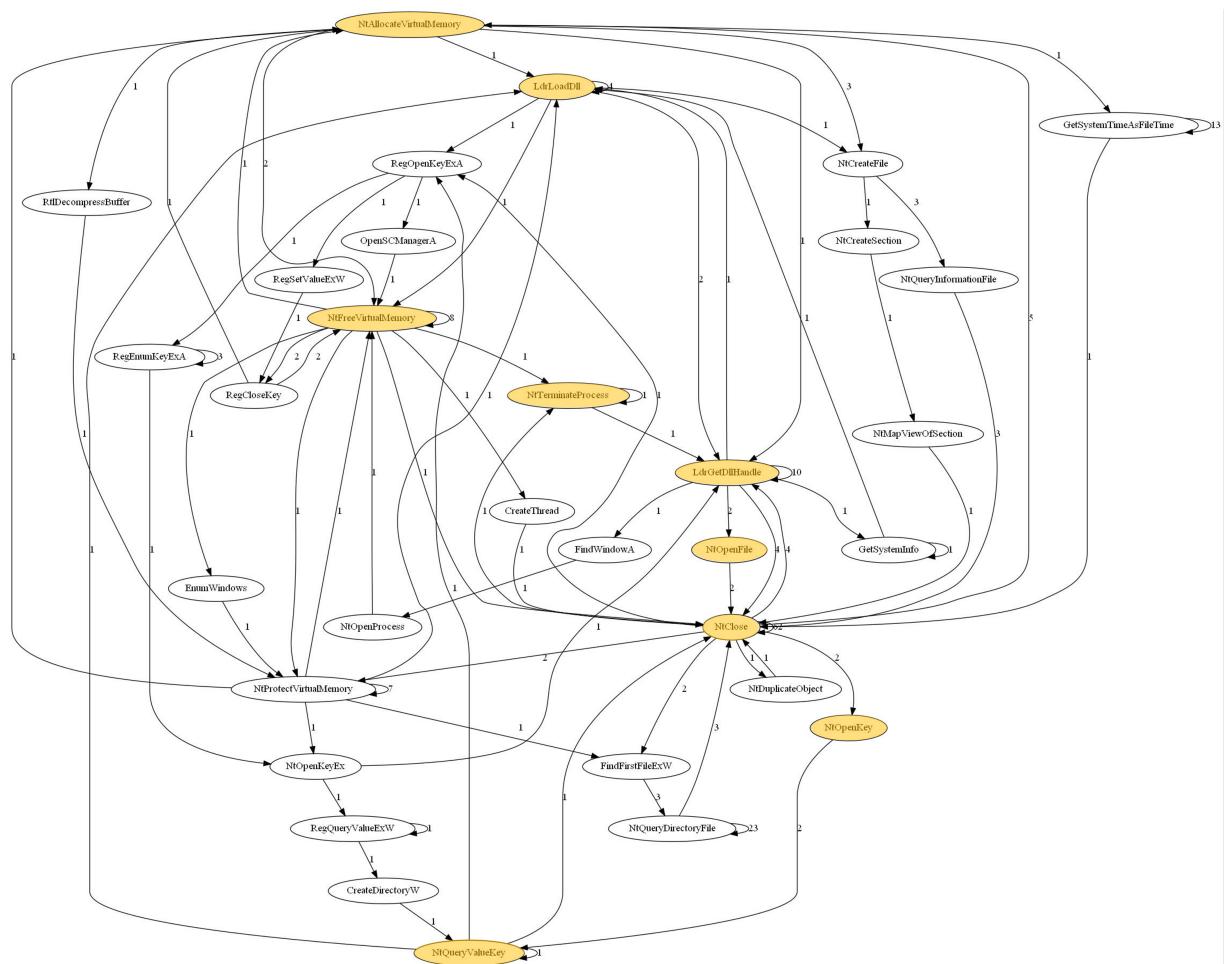


図 4.8 “Trojan.Win32.Scar”の特徴抽出用検体 (2)

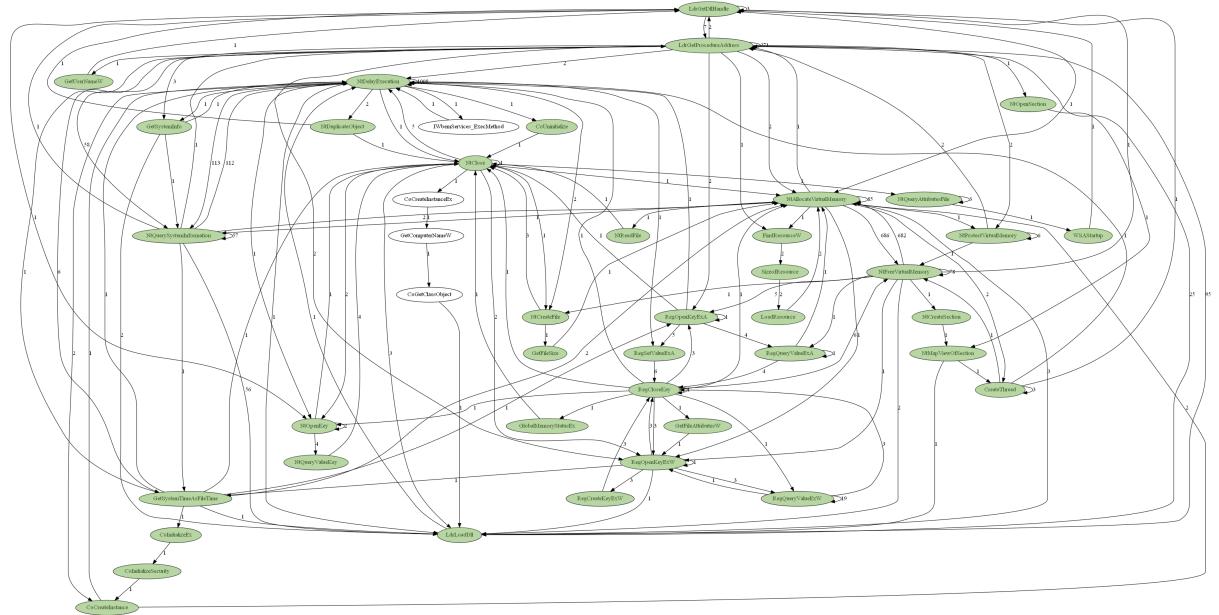


図 4.9 “Trojan.Win32.Poweliks” の特徴抽出用検体 (1)

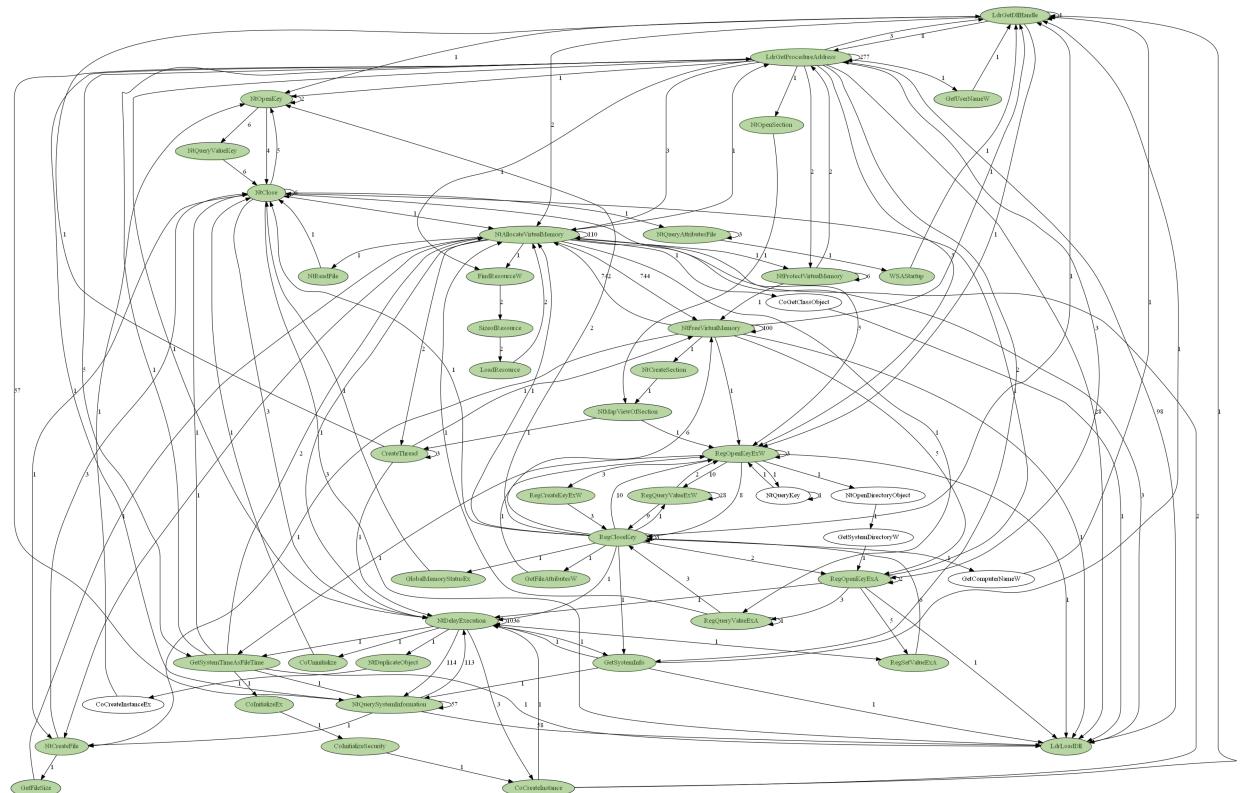


図 4.10 “Trojan.Win32.Poweliks” の特徴抽出用検体 (2)

図4.7、図4.8から、亜種分類の正解率が低いマルウェアファミリの検体同士では呼び出されるAPIコールやAPIコールからAPIコールへの遷移に同じ繰り返しの特徴が少ないと分かる。また、図4.9、図4.10から、正解率が高いマルウェアファミリの検体同士では呼び出されるAPIコールやAPIコールからAPIコールへの遷移に多くの同じ繰り返しの特徴を持っていることが分かる。以上のことから、APIコール列に類似した繰り返しの特徴が比較的少ないマルウェアファミリでは、繰り返し構造をよく圧縮するLZTの辞書による圧縮もされにくいため、亜種分類の正解率が低くなっていると考えられる。

## 第5章 結論

本研究ではマルウェアの動的解析結果に含まれる API コール列を圧縮アルゴリズムである LZT により圧縮し、その辞書による各マルウェアファミリの検体の圧縮率を特徴として、教師あり学習の SVM を用いて亜種判定を行った。その結果、平均 87.04% の高い正解率を得られた。しかしながら、亜種分類の正解率が比較的低いマルウェアファミリでは正解率が高いマルウェアファミリに比べて、API コールから API コールへの同じ繰り返しの特徴が少ない無いことが判明した。

今後の課題としては、さらなる分類精度の向上を目指し、マルウェアの特徴ベクトルをより特徴が表れるように改善していくことである。

## 謝辞

本研究を遂行するにあたり、常日頃より多大な助言や懇切丁寧な指導を頂きました高橋寛教授、甲斐博准教授、王森レイ講師、に深謝致します。そして本研究をご審査いただきました遠藤慶一准教授、宇戸寿幸准教授に深謝致します。最後に、常日頃から相談に応じていただきました浅沼和希先輩、助言や励ましを頂きました同研究室の皆様に深謝致します。

## 参考文献

- [1] McAfee 脅威レポート：2021年第1四半期  
<https://www.mcafee.com/enterprise/ja-jp/assets/reports/rp-threats-jun-2021.pdf> (参照：2022-01-19)
- [2] 東京大学への不正アクセスによる情報流出被害について  
[https://www.u-tokyo.ac.jp/focus/ja/articles/n\\_z1201\\_00001.html](https://www.u-tokyo.ac.jp/focus/ja/articles/n_z1201_00001.html)  
(参照：2022-02-09)
- [3] DNS サービス「Dyn」への大規模 DDos 攻撃、発信源は 10 万台の IoT 機器  
<https://project.nikkeibp.co.jp/idg/atcl/idg/14/481542/102800290/>  
(参照：2022-01-19)
- [4] 八木毅, 青木一史, 秋山満昭, 幾世知範, 高田雄太, 千葉大紀, “実践サイバーセキュリティモニタリング,” コロナ社, 2016.
- [5] MWS 2020  
<https://www.iwsec.org/mws/2020/> (参照：2022-02-07)
- [6] 寺田真敏, 他：マルウェア対策のための研究用データセット MWS Datasets～コミュニティへの貢献とその課題～, 情報処理学会, Vol.2020-IFAT-139 No.8, 2020年7月.
- [7] FFRI Dataset 2016 のご紹介,  
<http://www.iwsec.org/mws/2016/20160530-ffri-dataset-2016.pdf>  
(参照：2022-01-24)
- [8] 植松友彦, “文書データアルゴリズム入門, ” CQ 出版 (1994)
- [9] Ziv,J. and Lampel,A. : A Universal Algorithm for Sequential Data Compression, Vol.~23, No.~3, pp.337-343 (1977).
- [10] Ziv,J. and Lampel,A. : Compression of individual sequences via variable-rate coding, *IEEE Transactions on Information Theory*, Vol.~24, No.~5, pp.530-536(1978).

- [11] Welch,T.A. : “A Technique for High-Performance Data Compression,”  
*IEEE Computer*,Vol.17,No.6,pp.8-19 (1984).
- [12] P.Tischer, “A modified Lempel-Ziv-Welch data compression scheme.”,  
Aust.Comp.Sci.Commun,9,1,pp.262-272,1987.
- [13] Algorithms with Python / LZT 符号 - NCT  
[http://www.nct9.ne.jp/m\\_hiroi/light/pyalgo35.html](http://www.nct9.ne.jp/m_hiroi/light/pyalgo35.html) (参照 : 2022-01-21)
- [14] Graphviz-Graph Visualization Software  
<https://graphviz.org/> (参照 : 2022-01-19)
- [15] 浅沼和希, 甲斐博, 森井昌克, : “API コール列と LZW を用いたマルウェア亜種分類の一検討”, 情報処理学会第 83 回全国大会講演論文集 (2021)

## 付録A プログラムリスト

プログラム A.1 LZT\_trie.py

```
1 import sys
2
3 DICT_SIZE = 12
4 HEAD = 0
5
6 class TrieNode(object):
7     """
8         Trie 木の 1 ノードを表すクラス
9     """
10    WORD_SIZE = 64 ## underbar + space + alphabets + number ##
11    def __init__(self, depth = 0, item = None):
12        self.item = item
13        self.depth = depth
14        self.children = [-1 for i in range(TrieNode.WORD_SIZE)]
15
16    def is_leaf(self):
17        for index in self.children:
18            if index != -1:
19                return False
20        return True
21
22
23 class Trie(object):
24
25     def __init__(self, init_alphabets=None):
26         ## 追加可能ノード数の計算 ##
27         self.max_dict_cap = 2 ** DICT_SIZE
28         ## 根ノードの作成 ##
29         root = TrieNode()
30         self.nodes = [root]
31         # root の index は 0 #
32         node_index = 0
33         ## 初期辞書の登録 ##
34         if init_alphabets == None:
35             init_alphabets = list(
36                 ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_
37             )
38         ## 出力文字列 (int) ##
39         for item_alp in init_alphabets:
```

```
38         char_num = self._get_char_num(item_alp)
39         new_node = TrieNode(depth = 1, item = item_alp)
40         # ノード追加後なので、1からWORD_SIZE-1 が帰ってくる #
41         next_node_index = self._add_node(new_node)
42         self.nodes[node_index].children[char_num] = next_node_index
43
44     def _get_char_num(self, c):
45
46         # 文字のidを返す(配列実装のため)#
47         A:0, B:1, ..., Z:25, a:26, b:27, ..., z:51, 0:52, ..., 9:61, -
48             :62, " ":63
49         """
50
51         ## 出現する特別な文字 ##
52         if c == "_":
53             return 62
54         if c == " ":
55             ## space の id は 63 ##
56             return 63
57         ## アルファベット ##
58         if c.isalpha():
59             if c.isupper():
60                 return ord(c) - ord('A')
61             if c.islower():
62                 return ord(c) - ord('a') + 26
63         ## 数字 ##
64         if c.isdecimal():
65             return ord(c) - ord('0') + 52
66         ## いずれかに分類されない場合は読み込みエラー ##
67         print(f"読み込めない文字:{c}")
68         sys.exit(1)
69
70     def _add_node(self, node):
71         """
72             nodes のノードを追加する
73         """
74         self.nodes.append(node)
75         return len(self.nodes) - 1
76
77     def _delete_node(self, index):
78         """
79             指定index のノードを削除する
80             1. ノード自身の削除
81             2. 親ノードの子ノードリストからの削除
82             3. 子ノードリストの番号の再割り当て (なくなった番号以降のノードイン
83                 デックスを 1減算する)
84
85             ## ノード自身の削除 ##
86             del self.nodes[index]
87             ## 親ノードの子ノードリストから削除 ##
88
```

```
86     error_flg = True ## 子ノードの削除検知 ##
87     parent_node_index = self._ret_parent_node_index(index)
88     for i, child_index in enumerate(self.nodes[parent_node_index].
89         children):
90         if child_index == index:
91             ## initialize ##
92             self.nodes[parent_node_index].children[i] = -1
93             error_flg = False
94             break
95     ## 子ノードリストから何も削除されない場合は異常 ##
96     if error_flg:
97         print("Unexpected error:親ノードの保有する子ノードリストからの
98             削除に失敗")
99         sys.exit(1)
100    ## 子ノードリストの番号再割り当て ##
101    for i, t_node in enumerate(self.nodes):
102        for j, child_index in enumerate(t_node.children):
103            if child_index == index:
104                print("Unexpected error:親ノードの保有する子ノードリスト
105                    からの削除が不完全")
106                self.output_tree(0)
107                sys.exit(1)
108            if child_index > index:
109                self.nodes[i].children[j] -= 1
110
111    def encoding(self, word, char_index = 0, node_index = 0, stdout=False
112    ):
113        """
114        Trie に新しい単語を"登録せず"に符号後を返す
115        Parameter は insert_encoding と同じ
116        """
117        output_data = []
118        while True:
119            parent_depth = self.nodes[node_index].depth
120            char_num = self._get_char_num(word[char_index])
121            next_node_index = self.nodes[node_index].children[char_num]
122            if next_node_index == -1: ## 辿れなくなった場合 ##
123                ## 符号語の出力 & 出力の保存 ##
124                if stdout:
125                    print("{0}".format(node_index), end=" ")
126                    output_data.append(node_index)
127                node_index = 0
128                continue
129            else: ## 辿れる時 ##
130                if char_index < len(word) - 1:
131                    char_index += 1
132                    node_index = next_node_index
133                    continue
134                else: ## 最後の文字であれば ##
135
```

```
132         if stdout:
133             print(next_node_index)
134             output_data.append(next_node_index)
135             break
136     return output_data
137
138 def output_tree(self, node_index, depth = 0):
139     """
140         Trie 木の表示
141         debug 及び 動作確認 用
142     """
143     #####
144     def __my_space(depth):
145         for i in range(depth):
146             if i != 0:
147                 print(" ", end="")
148     #####
149
150     child_list = []
151     if node_index == 0:
152         print("\n親ノード:root\n")
153     else:
154         print()
155         __my_space(depth)
156         print("Node: {}".format(node_index))
157         __my_space(depth)
158         print(" depth: {}".format(self.nodes[node_index].depth))
159         __my_space(depth)
160         print(" item: {}".format(self.nodes[node_index].item))
161         for itr, x in enumerate(self.nodes[node_index].children):
162             if x != -1: ## 次の頂点が存在すれば
163                 __my_space(depth)
164                 print(" child node: {}".format(x))
165                 child_list.append(x)
166             for x in child_list:
167                 self.output_tree(x, depth+1)
168
169 ## 双方向リストによるキュー ##
170 class Queue:
171
172     def __init__(self):
173         self.prev = [None] * (2 ** DICT_SIZE + 1)
174         self.next = [None] * (2 ** DICT_SIZE + 1)
175         self.prev[HEAD] = HEAD
176         self.next[HEAD] = HEAD
177         self.data_num = 0
178
179     ## 最後尾に追加 ##
180     def insert(self, x):
181         last = self.prev[HEAD]
```

```
182         self.prev[x] = last
183         self.next[x] = HEAD
184         self.next[last] = x
185         self.prev[HEAD] = x
186         self.data_num += 1
187
188     ## 削除 ##
189     def delete(self, x):
190         p = self.prev[x]
191         q = self.next[x]
192         self.next[p] = q
193         self.prev[q] = p
194
195     ## 巡回 ##
196     def traverse(self):
197         n = self.next[HEAD]
198         while n != HEAD:
199             yield n
200             n = self.next[n]
201
202     def slide_number(self, n):
203         for i, item in enumerate(self.prev):
204             if item == None:
205                 continue
206             if item > n:
207                 self.prev[i] -= 1
208             if i > n:
209                 self.prev[i-1] = self.prev[i]
210
211         for i, item in enumerate(self.next):
212             if item == None:
213                 continue
214             if item > n:
215                 self.next[i] -= 1
216             if i > n:
217                 self.next[i-1] = self.next[i]
218
219
220     ### LZT 関連 ###
221     class Trie_LZT(Trie):
222
223         def __init__(self, init_alphabets=None):
224             super().__init__(init_alphabets)
225             self.queue = Queue()
226             for index in range(1, len(self.nodes)):
227                 self.queue.insert(index)
228
229         def insert_encoding(self, word, char_index=0, node_index=0, stdout=False):
230             output_data = []
```

```
231     while True:
232         ## 子ノードの深さを与えるために、親の深さを取得 ##
233         parent_depth = self.nodes[node_index].depth
234
235         ## 文字の番号表現を取得 ##
236         char_num = super().__get_char_num(word[char_index])
237
238         ## 現在のノードの子ノードに、次の文字が存在するかを確認 ##
239         next_node_index = self.nodes[node_index].children[char_num]
240
241         ## 辿れなくなった場合 ##
242         if next_node_index == -1:
243             if stdout:
244                 print("{0}".format(node_index), end = " ")
245                 output_data.append(node_index)
246
247         ## 辞書が埋まった場合 ##
248         if len(self.nodes)-1 == self.max_dict_cap:
249             for s_node_index in self.queue.traverse():
250
251                 ## 葉であるかの確認 ##
252                 if self.nodes[s_node_index].is_leaf() and self.
253                     nodes[s_node_index].depth != 1:
254
255                     ## 使用されていない辞書情報の削除 ##
256                     super().__delete_node(s_node_index)
257                     self.queue.delete(s_node_index)
258                     self.queue.slide_number(s_node_index)
259                     if node_index > s_node_index:
260                         node_index -= 1
261                         break
262
263         ## 辿れなくなった語を木に追加 ##
264         new_node = TrieNode(parent_depth+1, item = word[char_index]
265                             ])
266         next_node_index = super().__add_node(new_node)
267         self.nodes[node_index].children[char_num] = next_node_index
268
269         ## 登録した辞書番号をqueueに追加 ##
270         self.queue.insert(next_node_index)
271         node_index = 0
272         continue
273
274         ## 辿れる場合 ##
275     else:
276         if char_index < len(word) - 1:
277             self.queue.delete(next_node_index)
278             self.queue.insert(next_node_index)
279             char_index += 1
```

```
279             node_index = next_node_index
280             continue
281
282         ## 最後の文字 ##
283     else:
284         if stdout:
285             print(next_node_index)
286             output_data.append(next_node_index)
287             break
288
289     return output_data
290
291
292 if __name__ == '__main__':
293
294     trie = Trie_LZT()
295     apicalls = "aaaaa ababa 12abc ababa abcde"
296     out = trie.insert_encoding(apicalls, stdout=True)
297     outsize = 12 * len(out)
298     print(f"size:{outsize}")
299 }
```

---

プログラム A.2 api\_graph.py

---

```
1 import collections
2 from graphviz import Digraph
3
4 ## API コール列の読み込み ##
5 with open(r"C:\Users\Nagao\Desktop\Feature_Dataset\trojan.win32.poweliks
6 \18.txt", 'r') as f:
7     apicalls = f.read()
8
9 apicall_list = apicalls.split(" ")
10
11 ## 連想配列の生成 ##
12 apicall_graph = collections.OrderedDict()
13 for i in range(len(apicall_list)-1):
14     api_arrow = apicall_list[i] + "->" + apicall_list[i+1]
15     if api_arrow in apicall_graph.keys():
16         apicall_graph[api_arrow] += 1
17     else:
18         apicall_graph[api_arrow] = 1
19
20 ## グラフの作成 ##
21 graphviz = Digraph(format = "png")
22
23 if len(apicall_graph) > 1:
24
25     for edge, num_edge in apicall_graph.items():
26         call_1, call_2 = edge.split("->")
27         graphviz.edge(call_1, call_2, label=str(num_edge))
28
29
30 graphviz.render(r"C:\Users\Nagao\Desktop\graph\poweliks_18_graph")
```

---