Computer Security Symposium 2018
22 - 25 October 2018

ソースコードの類似度に基づく悪性JavaScriptの検知

三須 剛史[1,a)]　巻島 和雄[1]　岩本 一樹[1]

概要：Web 媒介型攻撃の 1 つとして Drive by Download 攻撃があり，JavaScript(JS) を用いて攻撃を行うため，これらを適切に分類・検知する手法が求められている．先行研究 [1] においては，Paragraph Vector(PV) を用いた類似度に基づく悪性 JS の分類を行った．実験の結果，同一クラスター内のデータには共通の傾向があることを明らかにした．本研究では PV を用いた手法に加えて，ソースコードを抽象構文木に変換し，構造や構文などの情報を特徴量とする．実験では悪性 JS と良性 JS を用いて，各手法における検知率を評価した．その結果，各手法では検知可能な JS に違いがあることを明らかにした．

キーワード：悪性 JavaScript，Drive by Download 攻撃，抽象構文木，Paragraph Vector

A Study on Detection of Malicious JavaScript Based on Source Code Similarity

Takeshi Misu[1,a)]　Kazuo Makishima[1]　Kazuki Iwamoto[1]

Abstract: Drive by Download, one of the threats via Web pages, uses JavaScript (JS) to perform attacks. Therefore, a method to classify and detect attacking JS is required. Previous study[1] classified malicious JS by similarity based on Paragraph Vector (PV). Experiments revealed common characteristics in data belonging to the same cluster. In this study, in addition to the PV-based method, we adopt a set of structure and syntax as features which is generated by converting JS source code to an abstract syntax tree. Experiments with benign and malicious JS evaluate the proposed methods and show that each method differs in types of JS that it can detect.

Keywords: Malicious JavaScript，Drive by Download Attack，Abstract syntax tree，Paragraph Vector

1. はじめに

Web を媒介とした攻撃は未だ重大な脅威であり，被害が後を絶たない．Cisco の 2018 年次サイバーセキュリティレポート [2] によれば，2016 年 4 月から 2017 年 10 月までの 1 年半に渡って Web 攻撃手法を分析した結果，悪意のある Web コンテンツが攻撃に利用される例が増加していたことがわかった．また，同レポートでは Web コンテンツ内に悪意のある JavaScript が含まれている例が大量に検出されていることも確認されており，悪意のある JavaScript を検知する手法が求められている．Web を媒介とした攻撃の 1 つとして Drive by Download 攻撃（以下，DbD 攻撃と呼ぶ.）がある．**図 1** に示すように，DbD 攻撃は，改ざんされた Web サイト，中継サイト，攻撃サイト，マルウェア配布サイトの 4 つからなる．DbD 攻撃は 3 つの手順で行われる．

(1) 攻撃者が一般の Web サイトの改ざん（以下，改ざんサイトと呼ぶ）を行う．
(2) ユーザが改ざんサイトにアクセスすると，複数の中継サイトへリダイレクトされる．
(3) 攻撃サイトで脆弱性を確認し，マルウェア配布サイトからマルウェアがダウンロードされる．

[1] 株式会社セキュアブレイン
SecureBrain Corporation
[a)] *takeshi_misu@securebrain.co.jp*

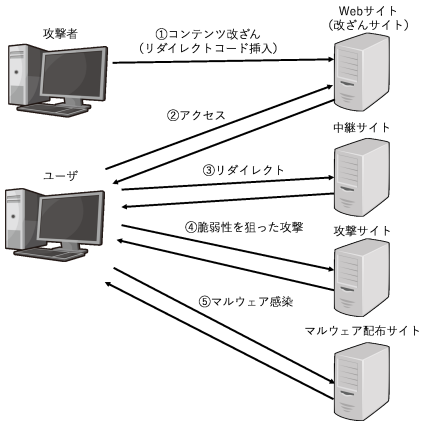

図 1 ドライブ・バイ・ダウンロード攻撃

DbD 攻撃では，中継サイトへのリダイレクトコードを含む JavaScript を埋め込こんでいる場合や，脆弱性の有無を判断する目的で Web ブラウザやプラグインのバージョンを取得する JavaScript を埋め込んでいる場合，また，マルウェアをダウンロードする JavaScript を埋め込んでいる場合があり，これらは検知を困難にする目的で難読化されていることもある．

また，DbD 攻撃に必要な基盤を容易に構築できる ExploitKit と呼ばれるツールがあり，ExploitKit が作成する攻撃コードはテンプレートに基づいて作成されるため，作成された JavaScript の関数名や URL に類似性があることが報告されている [3].

我々は，解析者が効率よく分析を行うことを目的として DbD 攻撃に含まれる悪性 JavaScript のソースコードを Paragraph Vector に適用し，特徴量の抽出を行った．また，抽出した特徴量を用いて類似度に基づいたクラスタリングを行い，結果を可視化する手法について検討した [1]．MWS 2017 Datasets 内にある Drive-by Download Data by Marionette Dataset[4]（以下，D3M Dataset と呼ぶ）を用いた評価実験により，同じような機能を持つ悪性 JavaScript が同じクラスタに属しており，分析の足がかりとなるような情報が得られることがわかった．

本研究では，先行研究 [1] にて行った Paragraph Vector を用いた特徴抽出手法に加えて，ソースコードを抽象構文木に変換し，構造や構文などの情報を特徴量とする手法について述べる．評価実験では良性および悪性 JavaScript を用いて検知率について評価を行う．考察では各手法における識別結果を元に JavaScript のソースコードを調査する．

以下に，本論文の構成を示す．2 章で，関連技術について説明する．3 章で，関連研究について述べる．4 章で，提案手法について述べる．5 章で，評価実験の結果について述べる．6 章で，5 章で行った評価実験の考察について述べる．7 章で，まとめと今後の課題について述べる．

2. 関連技術

本章では，本研究で用いる Paragraph Vector と，抽象構文木について説明する．

2.1 Paragraph Vector

ドキュメントを数値ベクトルとして表現し，類似度の計算や分類等様々な分析手法を適用しやすい形式とすることは自然言語処理の分野で頻繁に用いられる手法である．このような方法は自然言語に限らず，一定の語彙と文法を持つデータに対してであれば適用することが出来る．

ベクトル表現を得る方法として，従来 Bag of Words（以下，BoW と呼ぶ.）が広く用いられてきたが，Mikolov らによってニューラルネットワークを利用した Paragraph Vector が提案された [5]．後述する Word Vector[6] で得られた特徴を持つベクトル表現を，文章についても得ることが出来るよう拡張したものが Paragraph Vector である．Paragraph Vector の利点として，可変長の入力文章から一定長のベクトル表現を得られる点，語順を考慮したベクトル表現が得られる点，ベクトルが文章の意味を反映しておりベクトル間距離が文章の意味的距離を反映する点が挙げられる．

2.1.1 Word Vector

Word Vector は Paragraph Vector の基礎となる技術であり，文章ではなく一単語のベクトル表現を求める手法である．Word Vector では，単語のベクトル表現はその単語の存在する文脈によって決定されるため，単語の意味的表現を考慮した手法である．具体的な Word Vector の導出法としては Countinuous Bag of Words（以下，CBoW と呼ぶ）と skip-gram の 2 つの方法が提案されている．

2.1.2 CBoW

CBoW では，ある単語の前後それぞれ c 単語を入力とし，元の語を推測するニューラルネットワークを構築することで Word Vector を得る．入力される $2c$ 個の単語はそれぞれ BoW で用いられるような，語彙数を要素数とする One-hot 表現で入力され，低次元の文脈ベクトルに収束される．文脈ベクトルから中心の語を予測する BoW 表現を展開し，結果が実際の文章と一致するようニューラルネットワークを訓練する．訓練の結果得られたニューラルネットワークの重みが各単語を表現する Word Vector となる．

2.1.3 skip-gram

skip-gram では，CBoW とは逆に中心の 1 語から周囲に出現する単語を予測する．1 単語のみの入力から CBoW と同様の方法で出現する単語を予測する BoW 表現を計算し，文書中で周辺に出現する単語の出力が高くなるようニューラルネットワークを訓練する．

```
1 var hello = 'Hello';
```

図 2　JavaScript ソースコード

```
{
  "type": "Program",
  "start": 0,
  "end": 22,
  "body": [
    {
      "type": "VariableDeclaration",
      "start": 0,
      "end": 20,
      "declarations": [
        {
          "type": "VariableDeclarator",
          "start": 4,
          "end": 19,
          "id": {
            "type": "Identifier",
            "start": 4,
            "end": 9,
            "name": "hello"
          },
          "init": {
            "type": "Literal",
            "start": 12,
            "end": 19,
            "value": "Hello",
            "raw": "'Hello'"
          }
        }
      ],
      "kind": "var"
    }
  ],
  "sourceType": "module"
}
```

図 3　JavaScript AST 出力結果

2.2　抽象構文木 (Abstract Syntax Tree, AST)

抽象構文木は，プログラムを文，式，識別子の単位で抽象構文として表現する．コンパイラやインタプリタなどの言語処理系の中間処理に用いられる手法であり，構文木とは異なり，プログラムの意味に不必要な部分を省略して木構造として表現する．例えば，括弧の省略やソースコード内のコメントがこれに挙げられる．

プログラムを抽象構文木で表現した上で各構文をトークン化することで，プログラムにおける同じ型の変数は同じ構文として表現される．よって，文字列の名称変更といった類似性判定を逃れる難読化について効果があると考えられる．

2.2.1　JavaScript AST

JavaScript AST は JavaScript のソースコードを木構造に変換し，JSON 形式で表現する手法である．**図 2** のソースコードを JavaScript AST で表現すると**図 3** のようになる．

JavaScript AST で表現することにより，コードの意味が抽象的に表現される．例えば，name の hello の type は Identifier（つまり，変数名）であり，value の Hello の type は Literal（つまり，変数の値）であることがわかる．また，start と end はソースコードの何文字目から何文字目に現れたかがわかる．4 章で述べる特徴抽出手法 3 では，JavaScript AST で表現される type やソースコード全体のどの位置に現れたかなどの情報を特徴量として抽出する．

3.　関連研究

本章では，マルウェアの動的解析結果やソースコードの検知に関する関連研究について述べる．セキュリティ分野において，マルウェアの検知・分類に関する様々な研究が行われている [7], [8], [9].

佐藤らの研究 [7] では，マルウェアの呼び出す API 群とその引数に着目し，Paragraph Vector を用いて抽出した特徴量を基に，マルウェアがどのマルウェアの亜種であるかを推定する手法を提案している．佐藤らは API 群と引数を Paragraph Vector に適用しているのに対して，本研究では悪性 JavaScript のソースコードに Paragraph Vector を適用し検知するという点で異なる．

岩本らの研究 [8] では，API コールのパターンを用いたマルウェアの分類方法を提案している．この研究では 32 ビット Windows のマルウェアを逆アセンブルすることで制御フロー解析を行ってグラフを取得し，API が呼ばれた時に次に呼ばれる可能性のある API（API 推移）を検体の特徴としている．岩本らは API 推移だけに注目しているのに対して，本研究では JavaScript のメソッド呼び出しに限らずソースコード内の単語全般を対象としている点で異なる．

神薗らの研究 [9] では，抽象構文解析木を用いて自動生成されたポリモーフィックな JavaScript の検知および分類を行っている．この研究では，JavaScript の構造に着目し，構文解析木の特徴量を抽出し比較を行うことで構造の類似した JavaScript の判定を容易にしている．神薗らは抽象構文解析木オブジェクト情報をハッシュ値（MD5）に変換して比較しているのに対して，本研究では木構造同士の編集距離を計算し，比較しているという点で異なる．

4.　提案手法

本章では悪性 JavaScript の検知を行うための特徴抽出手法について述べる．JavaScript のソースコードの特徴量を抽出する手法として本研究では以下の 3 つの手法を提案する．

4.1　特徴抽出手法 1

悪性 JavaScript のソースコードに対して Paragraph Vector を用いて特徴量を計算する．2 章で述べたように，Paragraph Vector の利点は，入力文が可変長であり，入力文の長さによって処理が変わらない点が挙げられる．加えて，意味を考慮した特徴量の抽出を行うため単純な文字の出現頻度を特徴量とした場合と比較して，文章の意味的情報を保ちつつ，特徴量を抽出できるという点も挙げられる．

4.2　特徴抽出手法 2

JavaScript のソースコードを JavaScript AST に変換し，

出力された JSON ファイルから括弧等を取り除いたものについて特徴抽出手法 1 と同様に Paragraph Vector を計算し，特徴量とする．

ソースコードを AST へ変換するメリットとして，プログラムの実行において意味のない部分を省略できることが挙げられる．また，語の type と内容がセットになった状態のテキストを Paragraph Vector に渡すことで変数名や関数名のような任意に命名できる情報に Identifier や FunctionDeclaration といった情報が加わるため，変数名や関数名を変更して可読性を低下させる種類の難読化への効果が期待される．

4.3 特徴抽出手法 3

AST の利点としてソースコードの構造が木の形で明示される事が挙げられるが，Paragraph Vector はパラメータ window によって指定される前後幾つかの単語から文脈を推定する手法である．したがって，ソースコード全域に渡って存在する構造を反映することが難しいため，以下の 2 つを木構造情報の特徴量として取得した．

- 木構造から表層的な特徴量を取得

  表 1 に示すように，表層的な特徴量として手動で決定した 20 次元の特徴を取得する．なお，**表 2** は表 1 の root の子ノードの type13 種類をリストにしたものである．予備実験で，難読化が施された Java Script に対しては検知の精度が落ちる傾向が確認されたため，難読化された JavaScript に特有の構造が特徴として現れたパラメータを特徴量として定義した．
- それぞれの木構造間の編集距離を計算し，距離行列を一定次元の特徴量に変換

  木構造における編集距離は，2 つの木が与えられた時一方の木をもう一方の木と同じ形にするために必要な操作（ノードの削除、ノードの挿入、ノードのラベルの変更）の回数で定義される．木構造の編集距離を求めるには膨大な計算量が必要なため，近似アルゴリズムとして pq-gram[10] を用いた．得られた距離行列に Multi Dimensional Scaling(MDS) を適用し，超平面上の座標を特徴量とする．本研究では 100 次元の特徴量を取得した．

なお，上記 2 つの特徴量は抽象化されたものであり単独で検知に用いるには情報が不足していたため，特徴抽出手法 2 で得られた Paragraph Vector の結果と併用することで最終的な特徴量とした．

5. 評価実験

本章では，評価実験に用いた JavaScript と評価指標について説明し，最後に実験結果について述べる．なお実験には Paragraph Vector の実装である Doc2vec[11] を用いた．抽象構文木の生成には JavaScript のパーサである Acorn[12] を用いた．

表 1 特徴抽出手法 3 で決定した特徴量

項目	次元数
ソースコードの全長	1
root の子ノードの数	1
root の子ノードが一つのみであるか	1
root の子ノードの type	13
root の子ノードのうち最大のものがコード中どれだけの割合を占めているか	1
root の子ノードのうち最大のものがコード中どの位置に出現するか	1
木の深さの最大値	1
木の深さの平均値	1

表 2 root の子ノードの type13 種類

No.	項目
1	ClassDeclaration
2	VariableDeclaration
3	FunctionDeclaration
4	IfStatement
5	SwitchStatement
6	EmptyStatement
7	WhileStatement
8	ForInStatement
9	ForStatement
10	ExpressionStatement
11	TryStatement
12	BlockStatement
13	WithStatement

5.1 評価実験に用いた JavaScript

悪性 JavaScript の取得には，MWS Dataset 2018[13] 内の D3M Dataset を調査し，2,816 件を取得した．D3M Dataset は，高対話型のクライアント型ハニーポット Marionette で DbD 攻撃を検知した際の悪性通信データや，その際にマルウェアが行った通信データが収録されている．良性 JavaScirpt の取得には，調査した 3,600 サイトの中から 3,831 件を取得した．

5.2 評価指標

検知率の評価実験には，一般に機械学習の評価指標として用いられる混同行列を用いた．混同行列とは，データに対するモデルの予測結果を，TP（True Positive），FP（False Positive），TN（True Negative），FN（False Negative）の 4 つで分類し，それぞれに当てはまる予測結果を表にしたものである．

TP は悪性と予測した JavaScript が実際に悪性であった数であり，FP は悪性と予測した JavaScript が実際には良性であった数である．また，TN は良性であると予測した JavaScript が実際に良性であった数であり，FN は良性であると予測した JavaScript が実際には悪性であった数で

表 3　Doc2vec パラメータ

項目	設定値	説明
dm	0	PV-DM(1),PV-DBoW(0)
alpha	0.003	学習率
iter	200	反復学習の回数
min_count	5	単語の最低出現回数
window	16	同じ文脈とする前後の単語数
size	300	ベクトル次元数
sample	0.0001	高頻出単語の抑制率

表 4　pq-gram パラメータ

項目	設定値	説明
p	2	抽出するプロファイルのサイズ（親ノード）
q	3	抽出するプロファイルのサイズ（子ノード）

ある．

先に述べた混同行列から導出される指標が Precision，Recall，F-measure である．式 1 に示すように，Precision は悪性であると予測した JavaScript の内，実際に悪性であったものの割合である．

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

式 2 に示すように，Recall は悪性 JavaScript の中で悪性であると予測できたものの割合である．

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

式 3 に示すように，F-measure は，Precision と Recall の調和平均である．

$$F\text{-}measure = \frac{2Recall * Precision}{Recall + Precision} \tag{3}$$

5.3　実験パラメータ

Doc2vec の設定パラメータを**表 3** に示す．Paragraph Vector のモデルには PV-DBoW を用いた．pq-gram のパラメータを**表 4** に示す．

5.4　JavaScript の検知結果

機械学習のモデルにはニューラルネットワークを用いた．特徴抽出手法 1（Paragraph Vector を用いた場合），特徴抽出手法 2（抽象構文木を用いた場合），特徴抽出手法 3（特徴抽出手法 2 と手動で抽出した特徴量，および編集距離を組み合わせた場合）の 3 パターンでの実験を行った．実験結果は 10 分割の交差検定を 3 回行った結果の平均を求めた．

それぞれの結果を**表 5**，**表 6**，**表 7**，**表 8** で表す．表 8 に示すように，特徴抽出手法 1 の F-measure が 81.7%と他の手法と比較して高い精度を示している．また，Recall を比較すると特徴抽出手法 1 が 76.7%であるのに対して，特徴抽出手法 3 は 74.1%であり，その差は 2.6%である．一方で Precision は特徴抽出手法 1 が 87.4%，特徴抽出手法 3 が 83.0%と，その差は 4.4%である．

表 5　特徴抽出手法 1 の結果

	悪性（結果）	良性（結果）
悪性（予測）	2162	309
良性（予測）	653	3521

表 6　特徴抽出手法 2 の結果

	悪性（結果）	良性（結果）
悪性（予測）	2045	431
良性（予測）	771	3399

表 7　特徴抽出手法 3 の結果

	悪性（結果）	良性（結果）
悪性（予測）	2089	429
良性（予測）	726	3402

表 8　Precision,Recall,F-measure 算出結果

	Precision	Recall	F-measure
特徴抽出手法 1	87.4%	76.7%	81.7%
特徴抽出手法 2	82.5%	72.6%	77.2%
特徴抽出手法 3	83.0%	74.1%	78.3%

特徴抽出手法 2 と 3 を比較すると F-measure は 1%程向上している．これは，特徴抽出手法 2 が構文木のみに着目していたのに対し，特徴抽出手法 3 ではそれ以外に，編集距離や手動で抽出した特徴量を追加したため精度が向上したと考えられる．

各手法において特徴抽出手法 1 が高い精度を示しているが，特徴抽出手法 1 でのみ識別できなかった JavaScript と特徴抽出手法 3 でのみ識別できなかった JavaScript を比較すると差が生じていた．なお，本稿で述べる “識別できなかった JavaScript” とは False Negative と False Positive を指す．具体的には，どちらの手法でも識別できなかった JavaScript が 338 件，特徴抽出手法 1 でのみ識別できなかった JavaScript が 263 件，特徴抽出手法 3 でのみ識別できなかった JavaScript が 323 件あった．

6 章では各手法において識別できなかった JavaScript に着目し，ソースコードにどのような特徴があるのかを調査した結果について述べる．

6.　考察

本章では，5 章で行った評価実験について考察する．特徴抽出手法 1 と特徴抽出手法 3 のどちらでも識別できなかった JavaScript，特徴抽出手法 1 で識別できなかった JavaScript，特徴抽出手法 3 で識別できなかった JavaScript について分析を行った．

6.1　特徴抽出手法 1 および特徴抽出手法 3 で識別できなかった JavaScript の例

ソースコードの量が少ない JavaScript が存在した．ソースコードの内容に大きな特徴はなく，変数を 1 つ定義しているだけのもの，関数を 1 つだけ呼び出しているものなど

```
var poYmwvXMxt7;
poYmwvXMxt7=document.getElementById("MLUsVUWgRV2").innerText;
tzwKyxlLUC6.open('G'+'E'+unescape('%54'),poYmwvXMxt7,false);
tzwKyxlLUC6.send();
faoWyrRHTc4.type = 1;
faoWyrRHTc4.open();
faoWyrRHTc4.Write(tzwKyxlLUC6.responseBody);
CniOsRXnld8 = "[redacted].exe";
faoWyrRHTc4.SaveToFile(CniOsRXnld8,2);
```

図 4 識別できなかった JavaScript の例（特徴抽出手法 1）

```
document.write(("<iframe src=[redacted] width=12 height=11
frameborder=0 marginheight=0 marginwidth=0 scrolling=no> </" + "iframe>"));
```

図 5 識別できなかった JavaScript の例（特徴抽出手法 3）

様々であった．単体で機能する JavaScript ではなく，呼び出し元の JavaScript が他にあると推測される．

識別できなかった理由としては，どちらの手法でもソースコードの量が少ない場合，得られる特徴量も比例して少なくなるため十分な特徴量が抽出できなかったと考えられる．これらの特徴を持つ JavaScript を識別するためには，ソースコードの量が少ない場合でも特徴量を抽出する手法の検討が必要である．

6.2 特徴抽出手法 1 で識別できなかった JavaScript の例

図 4 に示すようにソースコードが難読化された悪性 JavaScript が存在した．ソースコードの特徴として変数名や関数名が全体的に無意味な文字列に難読化されており，可読性を下げていることがわかった．

特徴抽出手法 1 で識別できなかった理由としては，Paragraph Vector ではソースコードを文章として扱い，文脈を特徴量として計算する．よって，ソースコード内の変数や関数が難読化されていた場合，文章の意味を十分にとらえることが出来ず識別できなかったと考えられる．

6.3 特徴抽出手法 3 で識別できなかった JavaScirpt の例

図 5 に示すように document.write を使って iframe を埋め込む悪性 JavaScript が存在した．特徴抽出手法 3 で識別できなかった理由としては，ソースコードの木構造をみた場合，良性の JavaScript にも悪性 JavaScript と同じ木構造が多く存在したため，その特徴が強く反映されてしまったと考えられる．

7. まとめと今後の課題

本章では，本研究のまとめと今後の課題について述べる．本研究では，Paragraph Vector に加えて抽象構文木を用いてソースコードの構文を抽出し，ベクトル表現への変換を行う手法について検討した．評価実験では良性および悪性 JavaScript を用いて検知率について評価した．

評価実験の結果，各特徴抽出手法で識別できる JavaScript のソースコードに差があることがわかった．今後は JavaScript の検知精度を高めていくために，より詳細な分析を実施する必要があるため，各特徴抽出手法の結果を元に分類を行う予定である．

謝辞 本研究は，国立研究開発法人情報通信研究機構の委託研究「Web 媒介型攻撃対策技術の実用化に向けた研究開発」の成果の一部です．ご協力いただいた皆様に，深く感謝します．

参考文献

[1] 三須剛史，巻島和雄，岡田晃市郎，岩本一樹 “ソースコードの類似度に基づく悪性 JavaScript の分類に関する一検討，” コンピュータセキュリティシンポジウム 2017，pp.378-384，Oct．2017.

[2] シスコ 2018 年次サイバーセキュリティレポート，https://www.cisco.com/c/dam/m/ja_jp/offers/183/sc-04/cisco-acr2018-ja.pdf

[3] 笠間貴弘，神薗雅紀，井上大介，“ExploitKit の特徴を用いた悪性 Web サイト検知手法の提案，” コンピュータセキュリティシンポジウム 2013，pp.603-610，Oct．2013.

[4] Mitsuaki Akiyama, Kazufumi Aoki, Yuhei Kawakoya, Makoto Iwamura, and Mitsuataka Itoh, “Design and Implementation of High Interaction Client Honeypot for Drive-by-download Attacks,” IEICE Transactions on Communication, Vol.E93-B No.5 pp.1131-1139, May. 2010.

[5] Mikolov,Tomas,et al,“Distributed representations of sentences and documents,”Proceedings of the 31st International Conference on Machine Learning (ICML-14), 2014.

[6] Mikolov,Tomas,et al,“Efficient estimation of word representations in vector space,”International Conference on Leaning Representations(ICLR), 2013.

[7] 佐藤拓未，後藤滋樹，武部嵩礼，“Paragraph Vector を用いたマルウェアの亜種推定法，” コンピュータセキュリティシンポジウム 2016，pp.298-304，Oct．2016.

[8] 岩本一樹，和崎克己，“静的解析により抽出された API 推移に基づくマルウェアの分類，” 情報処理学会論文誌，Vol．54．No．3．pp.1199-1210，Mar．2013.

[9] 神薗雅紀ら，“抽象構文解析木による不正な JavaScript の特徴点抽出手法の提案，” 情報処理学会論文誌，Vol．54．No．1．pp349-356，Jan．2013.

[10] AUGSTEN, Nikolaus,et al, “The pq-gram distance between ordered labeled trees,”ACM Transactions on Database Systems (TODS), 2010, 35.1: 4.

[11] Doc2vec，https://radimrehurek.com/gensim/models/doc2vec.html

[12] Acorn，https://github.com/acornjs/acorn

[13] 高田雄太ら，“マルウェア対策のための研究用データセット ～ MWS 2018 Datasets ～，” 情報処理学会，Vol．2018-CSEC-82，No．38，2018 年 7 月．