抽象構文解析木の符号化による不正なJavascriptの分類手法の提案

上西 拓真† 神薗 雅紀‡ 西田 雅太‡ 森井 昌克†

†神戸大学大学院工学研究科
657-8501 兵庫県神戸市灘区六甲台町 1-1
johnishi.y@stu.kobe-u.ac.jp, mmorii@kobe-u.ac.jp

‡株式会社セキュアブレイン 先端技術研究所
102-0083 東京都千代田区麹町 2-6-7 RK ビル 4F
{masaki_kamizono, masata_nishida}@securebrain.co.jp

あらまし 近年，難読化 Javascript で記述されたマルウェアが増加しておりその解析に関する研究が盛んに行われている．その一つに類似するコードの抽象構文解析木を導出し明確に特徴点とすることで検知や解析に応用する研究がある．その結果より類似する木構造を持つコードが複数存在することがわかっている．しかし既存の研究では抽象構文解析木の類似度を判定し，分類することは主眼としてなかった．本稿では，抽象構文解析木をより抽象化して符号化することで類似するコードを分類する手法を提案する．そして提案手法を用いて MWS のデータセットの不正な Javascript を分類し，提案手法の有効性を示す．

Classification of Hostile Javascript based on Encoding Abstract Syntax Tree

Takuma Johnishi† Masaki Kamizono‡ Masata Nishida‡ Masakatu Morii†

†Graduate School of Engineering, Kobe University
Rokkodai 1-1, Nada-ku, Kobe-shi, Hyogo, 657-8501, JAPAN
johnishi.y@stu.kobe-u.ac.jp, mmorii@kobe-u.ac.jp

‡Advanced Reserch Laboratory, Securebrain Corporation
Kojimachi RK Bldg,6-7 Kojimachi 2-chome,Chiyodaku, Tokyo, 102-0083, JAPAN
{masaki_kamizono, masata_nishida}@securebrain.co.jp

Abstract

In the recent years, the number of malware written in obfuscated Javascript has been increasing. Abstract syntax tree(AST) is used as a measure of classifying these malware to counter these malware. In previous research, it is proposed to use an AST based on structural analysis of polymorphic Javascript. However, Classification of Javascript has not been performed in the research. In this paper, we propose a method which enable to classifying Javascript by further abstracting AST and by encoding AST. Furthermore experimental results with the malware samples provided by MWS DATAset show that our proposed method can exactly classify and summarize hostile Javascript that is obfuscated in a similar way.

1 はじめに

近年，Javascript により記述されたマルウェアが増加している．その一つに Drive-by download 攻撃があげられる．Drive-by download 攻撃とは，不正なコードを Web ページに埋め込むことによりそのページを訪れたユーザを不正サイトへ誘導し，自動的にマルウェアの実行ファイルをダウンロードさせる攻撃である．また標的型攻撃などで利用されている PDF タイプのマルウェアにも Javascript は使用されている．これらの不正な Javascript には，アンチウイルスソフトの検知やマルウェア解析者の解析を妨害するという目的で難読化が施されている．難読化を解除してマルウェアを解析する手法として動

的解析が行われてきた．しかしタイマー処理やイベント処理等の影響により，動的解析では全ての挙動を正確に把握することは難しい．このように動的解析には精度の面で課題が残るため，難読化を解除することなく不正な Javascript を解析，検知する研究が行われている．神薗らは構文解析木を用いて不正な Javascript の特徴点を表す抽象構文解析木を提案し，機械的に作成されたポリモーフィックなコードの検知や解析を実現している [1]．また宮本，グレゴリーらは抽象構文木 (Abstract Syntax Tree:AST) を利用し，不正な Javascript における難読化アルゴリズムの推定，分類を行っている [2][3].

本稿では神薗らの手法に着目し，同じような難読化が施された不正な Javascript の分類を行う．この手法の課題は，構造が類似するコードに対して網羅的に分類を行えない点である．我々は神薗らが提案する抽象構文解析木をさらに抽象化することによって，より汎用性の高い分類を行う手法を提案する．またマルウェア検体として MWS のデータセットを用いて PDF ファイルに埋め込まれている不正な Javascript を分類し，提案手法の有効性を検証する．

2 抽象構文解析木による不正な Javascript の解析

本章では構文解析木を利用した Javascript コードの分類に関する研究について述べる．まず構文解析木について説明し，次に神薗らが提案する抽象構文解析木について述べる．

構文解析木とはプログラムを構文解析することで構文を構成する様々な表現を属性として与え，それを単位として構成する構文木のことである．図 1 に構文解析木の例を示す．構文解析木はノードごとに属性が割り当てられており，属性によっては内容を表す値を持つ．

神薗らは不正な Javascript が引数の設定値や変数名の定義を変更することで難読化のパターンを変更していることに着目し，構文解析木の抽象化を行った．抽象化により不正な Javascript の構文解析木におけるノードの値を考慮しないことで，引数の設定値や変数名の定義に対応する情報を集約している．そしてコードの構造が類似する不正な Javascript の特徴点となる抽象構文解析木を提案し，同じような難読化が施されたコードの抽象構文解析木が一致，または類似することを示している．抽象構文解析木が一致する場合，コードが類似するもの同士を特定することができる．しかしコードが類似していても抽象構文解析木が一致しない場合，木構造が類似することは確認されたがそれを類似するものと判断して分類を行うことは難しい．

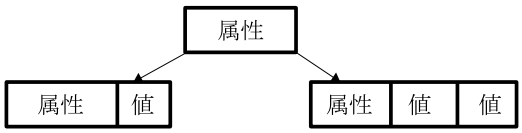


図 1: 構文解析木

3 不正な Javascript の分類手法

本稿では不正な Javascript における抽象構文解析木の構造変化に対応する分類を行う．神薗らの手法では構文解析木のノードの属性を考慮しており，類似する構造を定量的に表して分類することが困難である．我々は抽象化を強めることで類似する構造を分類できると考え，ノードの属性だけでなくノードの値も考慮しないことで構文解析木の形状のみに着目して符号化を行う．これにより，抽象構文解析木を定量化しコード間の類似度を算出することで分類を行う手法を提案する．3.1 節にて抽象構文解析木を抽象化する手法について延べ，3.2 節にて符号化によるコード間の類似度算出法について説明する．

3.1 抽象構文解析木の更なる抽象化

本節では神薗らが提案する抽象構文解析木をさらに抽象化する手法について述べる．3.1.1 項にて Javascript の構文解析木の導出法について説明し，3.1.2 項にて構文解析木の抽象化を行う．

3.1.1 構文解析木の導出

Javascript の構文解析を行うため，本稿では UglifyJS[4] を用いる．UglifyJS は Javascript からノードの属性と，ノードの値から構成される構文解析木を生成する．構文解析の結果は木構造の一般的な表現方式である BP 表現 [5] で表わされる．図 2 及び図 3 に難読化された Javascript の一部とそれを構文解析した結果を示す．また図 2 に示すコードの構文解析木を図 4 に示す．

3.1.2 構文解析木の抽象化

神薗らの手法では構文解析木の各ノードの値を考慮しないことで抽象化を行っている．それに対し提案手法では各ノードの属性，及びノードの値の両方の情報を考慮しないことにより抽象化を強める．図 5 に，図 4 を抽象化した木構造を示す．本稿では提案手法により導出された構文木を抽象構文解析木と定義する．これを用いて同じような難読化が施されているコードに対して類似度を算出し，それを指標として分類を行う．

```
var lookManyDummy = lookedHave.replace(/@B¥$¥^#]/g, '')
```

図 2: 難読化された Javascript の例

```
[ 'toplevel',
  [ [ 'var',
      [ [ 'lookManyDummy',
          [ 'call',
            [ 'dot', [ 'name', 'lookedHave' ], 'replace' ],
            [ [ 'regexp', '@B\\$\\^#]', 'g' ], [ 'string', '' ] ] ] ] ] ] ];
```

図 3: UglifyJS による構文解析結果

3.2 抽象構文解析木の比較処理

本節では抽象構文解析木を比較し，類似性を示す距離 L と類似度 S を導出する．抽象構文解析木の共通部分を抽出する手法として，本稿では Longest Common Substring 手法 (LCS 手法) を用いる．LCS 手法を適用するため 3.2.1 項にて抽象構文解析木の符号化を行い，3.2.2 項にて LCS 手法を用いた符号間の共通部分文字列の抽出手法について説明する．そして導出した共通部分文字列を使用し 3.2.3 項にて符号間の距離 L と類似度 S を算出する．

3.2.1 抽象構文解析木の符号化

抽象構文解析木の符号列は，構文解析木の BP 表現に対し「 [」を「 1 」,「] 」を「 0 」に変換し抽出することにより導出される．例えば図 3 の BP 表現に対応する抽象構文解析木を符号化すると「11111111001101000000000」となる．木構造を符号列に変換して扱うことでコード間の類似度を表すことができ，異なる抽象構文解析木の構造を定量的に比較できる．

3.2.2 LCS 手法による共通部分文字列の抽出

LCS 手法とは 2 つの文字列を比較し，共通する部分文字列のうち最も長いものを抽出するアルゴリズムである．例えば {abcd} と {acdb} の LCS は {cd} となる．本稿では LCS 手法を用いて符号間における共通部分文字列を複数抽出することで，後に示す符号間の類似度をより正確に得る．また符号間の特徴を表す共通部分文字列のみを抽出するため，あらかじめ最小重み t を設定し重みが t を超える共通部分文字列のみを抽出する．それらの重みの和を用いて符号間の類似度算出を行う．類似度算出の流れを図 6 に示す．まず比較する 2 つの文字列 A_0, B_0 に対して LCS 手法を適用し共通部分文字列 C_0 を抽出する．次に A_0, B_0 から抽出した共通部分文字列を取り除き，新たな文字列 A_1, B_1 を得る．これらの

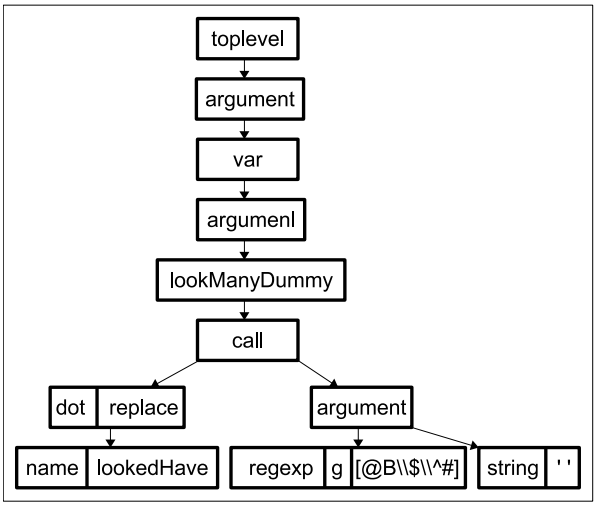


図 4: Javascript の構文解析木の例

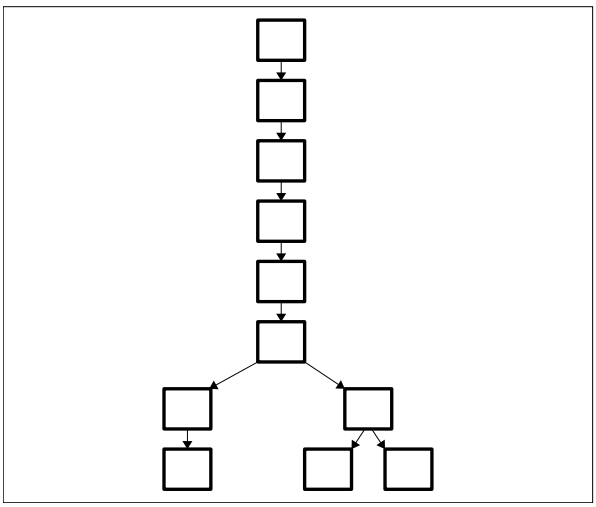


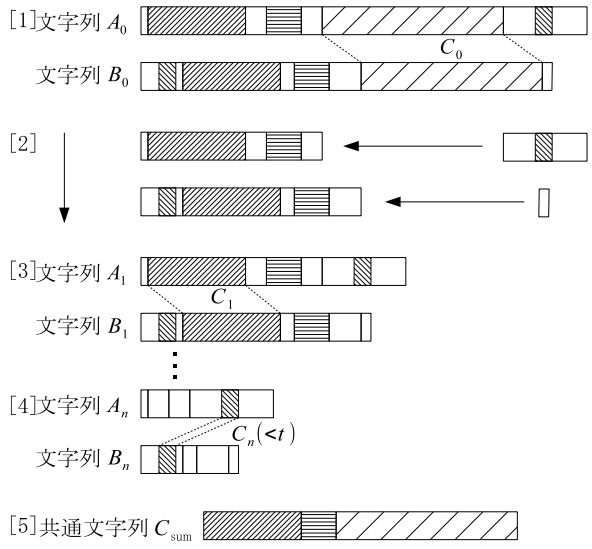
図 5: Javascript の抽象構文解析木の例

文字列に対しても同様に LCS 手法を適用し共通部分文字列 C_1 を抽出する．上記の処理を繰り返し行い，抽出した共通部分文字列の重み C_{n} が t を下回れば抽出処理は終了する．ここで共通文字列の和である C_{sum} を

$$C_{\mathrm{sum}} = C_0 + C_1 + \cdots\cdots + C_{\mathrm{n}-1}$$

と定義し，C_{sum} の重みを文字列 A_0, B_0 の共通部分の重みとする．

例として本稿で用いた不正な Javascript コードのうち，同じような難読化が施されたコード間の共通部分の重みを導出する．コードの抽象構文解析木を表す符合列の重みはそれぞれ 236，266 である．コード間における最大共通部分文字列の重みは 103 である．それに対し本手法を用いて算出した共通部分の重みは 197 である．このように複数の共通部分文字列を用いることでより多くの共通部分を抽出する．



[1] A_0, B_0の最大共通部分文字列 C_0を導出.
[2] A_0, B_0から C_0 を除去し，結合.
[3] A_1, B_1の最大共通部分文字列C_1を導出．以下[1]，[2]の繰り返し.
[4] $|C_n|<t$ を満たせば終了.
[5]. C_0からC_{n-1}の部分文字列を結合しA_0, B_0 の共通文字列 C_{sum} を得る

図 6: 共通部分文字列の導出の流れ

3.2.3 符号間の距離及び類似度算出

前述した文字列 A_0, B_0 の共通部分 C_{sum} を用いて，A_0, B_0 間の類似度 S と距離（非類似度）L を次のように定義する．

$$S = \frac{|C_{\mathrm{sum}}|}{|A_0| + |B_0| - |C_{\mathrm{sum}}|} \qquad (0 \leq S \leq 1)$$

$$L = 1 - S \qquad (0 \leq L \leq 1)$$

文字列 A_0, B_0 を不正な Javascript コードの抽象構文解析木を表す符号列とすると，S はコード間の類似度を表し L はコード間の距離を表す．実際に分類を行う際はコード間の距離 L を用いてクラスター分析を行うため，以降はコード間の関係を距離 L を用いて表す．L が 0 の場合は 2 つのコードの抽象構文解析木の構造が完全に一致することを意味する．L が大きくなるにつれて共通部分は減り，L が 1 の場合は共通部分が全く無いことを意味する．

距離 L の定義の妥当性について考察する．それぞれ互いの距離 L が異なる不正な Javascript コードの一部を図 7〜図 9 に示す．図 7，図 8 に示されるコードは，構造が大きく異なっているため異なる難読化が施されているといえる．一方図 8，図 9 に示されたコードは，変数名や識別子の設定が異なるが全体的な構造は似ているため同じような難読化が施されているといえる．しかし図に示す部分の構文の構造が異なるため，神薗らが提案する抽象構文解析木は一致しない．提案手法を用いて図 7, 図 8 に示されるコード間の距離 L を求めた結果，$L = 0.355$ となった．また図 8, 図 9 に示されるコード間の距離 L を求めた結果，$L = 0.978$ となった．同じような難読化が施されているコード間の距離はそうでないものと比べ明らかに小さい．よって距離 L にしきい値を与えることにより，コードが類似しているかを判定することができると考えられる．

PDF文書に埋め込まれたJavascript

```
function dlVtf(dlVtf){var VEzeI=mBw1=0,mBw2;for(;mE
return VEzeI;}
function rlkfb(mEi2,rlkfb){if(rlkfb==0){return 1;}
var jaxnF=mEi2;for(var Lo1s=1;Lo1s<rlkfb;Lo1s++){ja
return jaxnF;}
function xbvtG(GvS1,GvS0){while(GvS1.GvS0gth*2<GvS0
return GvS1.substring(0,GvS0/2);}
function u71L(u711,u712){util['\u0070\u0072\u0069\u
function S7aL(u713,u714){Collab['\u0067\u0065\u0074
function s3nT(SzAIi){var DDGfx=new Array();if(SzAIi
```

```
if(SzAIi>8.103&&SzAIi<=9){DDGfx
[1]='011010001010100111011010011111101001001100000
[0]='000111101100100010101000010111101111011011001
DDGfx;}}
```

図 7: PDF に埋め込まれた不正 Javascript コード (pattern 1)

4 提案手法を用いた不正な Javascript の分類評価

本章では提案手法を用いた不正な Javascript の分類結果について述べる．分類する対象として MWS のデータセットを使用し，PDF ファイルに埋め込まれている不正な Javascript を分類する．PDF ファイルから不正な Javascript を抽出する際には JSunpack[6] を用いる．データセットの内訳は，D3M 2010 のデータセット [7] から 147 種，D3M 2011 のデータセット [8] から 25 種，D3M 2012 のデータセットから 31 種の計 203 種である．

4.1 実験内容の説明

各検体に対して 3 章にて述べた手法を適用し検体間の距離を算出する．それを用いて全ての検体間における距離の組み合わせを表す距離行列を作成する．ただし共通部分文字列を得るための最小重みのしきい値 t は 25 とする．この値は false positive を軽減

図 8: PDF に埋め込まれた不正 Javascript コード (pattern 2)

するため，コードの構造が全く異なる検体同士を比較した際に算出される最大の共通部分文字列の重みを上回るように設定した．導出した距離行列を利用して階層的クラスター分析を実施する．分類手法はメディアン法を用いる．

4.2 結果

検体間の距離 L が 0 である検体，すなわちコードの構文の構造が完全に一致する検体を 1 つのクラスタとしてまとめた場合，クラスタ数は 52 であった．それに対してクラスタ間の距離 L が小さいものから順に 1 つのクラスタとしてまとめた場合，クラスタ間の距離 L が 0.637 以下の場合は同じような難読化が施されたコードを正確に分類を行うことができた．その際のクラスタリング結果を表 1 に示す．この際のクラスタ数は 38 となり，クラスタ内の距離 L を 0 として分類した結果と比べ 14 個のクラスタを集約することができた．表 1 における検体内の距離が 0 でないクラスタは，コードの構造が異なる検体を分類したことを示す．また複数のデータセットにわたって検体を含むクラスタは 3 個しか存在しなかった．とくにクラスタ 3 に分類された検体は，属するデータセットごとに構文の構造が変わらない程度に難読化が施されていることがわかった．このことから不正な Javascript の種類は年々変化し続けているといえる．

図 9: PDF に埋め込まれた不正 Javascript コード (pattern 3)

次にクラスタごとにファイルの内容を解析した結果を示す．全てのクラスタにおいてクラスタ内の検体は同じような難読化が施されていることがわかった．例えば図 8，図 9 に示す不正な Javascript はクラスタ 6 に分類された検体である．このクラスタの検体はすべて以下に示す手順で動作するものであった．まず PDF のヘッダー内に符号化されたデータが定義されており，この符号列を Javascript 内に読み込み replace 関数を用いて不必要な部分を削除する．次に符号列に隠されている 16 進コードを 1 つずつ unescape 関数を用いて復号する．最後に復号された Javascript コードを eval 関数で実行する．

各クラスタ内においては正確に同じような難読化が施された検体のみを分類できた．しかし異なるクラスタ間においても同じようなコードの構造を持つ検体が存在した．そのためクラスタの集約が完全に行えていたわけではなかった．例えばクラスタ 21 とクラスタ 30 の検体は，共に Javascript 内に定義された文字列から不要な記号を削除し eval 関数で実行するというものであった．しかしこれらのコードは構文の構造が非常に煩雑なため，十分な共通点が抽出できなかったと考えられる．

表 1: D3M データセットのクラスタ分析結果

クラスタ	検体数	クラスタ内の距離	検体の内訳		
			D3M 2010	D3M2012	D3M2012
1	32	0.229	32		
2	18	0.572	18		
3	14	0.080	6	6	2
4	14	0.120	14		
5	12	0.000			12
6	10	0.355	10		
7	9	0.000			9
8	8	0.000	8		
9	8	0.135		8	
10	7	0.000	7		
11	7	0.000	7		
12	7	0.000		3	4
13	5	0.000		5	
14	5	0.000	5		
15	4	0.000			4
16	4	0.000	4		
17	4	0.000	1	3	
18	4	0.000	4		
19	4	0.000	4		
20	3	0.000	3		
21	3	0.637	3		
22	2	0.394	2		
23	2	0.168	2		
24	2	0.000	2		
25	2	0.000	2		
26	1	0.000	1		
27	1	0.000	1		
···	···	···	···		
38	1	0.000	1		

5 考察

神薗らの手法に比べ構文解析木の情報をさらに抽象化することで，不正な Javascript の抽象構文解析木を定量的に扱うことを可能にした．そして提案手法により，異なるクラスタ間の距離 L にしきい値を与えることで MWS のデータセットに含まれる不正な Javascript をより集約して分類した．しかし本稿ではクラスタ間の距離 L のしきい値を主観的な判断から決定している．そのため同じような難読化が施されたコード間の距離とそうでないコード間の距離の違いを明確にし，より適切なしきい値を求める必要がある．また今回の手法では構文解析木に強い抽象化を施すことで不正な Javascript の構文構造の差異を無視したため，多くの情報を使用できていない．今後の課題は構文解析木の適切な抽象化を行い部分マッチングにより重要なノードに重みを与えることで，さらに分類の精度を高めること等が挙げられる．

6 まとめ

本稿では神薗らの抽象構文解析木をさらに抽象化した．これにより同じような難読化が施された不正な Javascript コードの特徴を符号化し，LCS 手法を用いることでコード間の距離 L を定量的に表した．また距離 L を用いて D3M 2010〜2012 のデータセットに存在する PDF ファイルに埋め込まれている不正な Javascript を分類した．その結果マルウェア間の距離 L が 0.64 以下の検体を 1 つのクラスタとしてまとめた場合，同じような難読化が施された検体を分類することができた．そして同じような難読化が施されているコードに対して，より集約度を高めることを可能にした．また不正な Javascript の種類が年々変化し続けていることを確認した．しかし，同じような難読化が施されているにもかかわらず異なるクラスタに分類された検体が存在した．今後は考察で述べた課題を踏まえて分類の精度を高め，より一般性の高い分類を目標とする．

参考文献

[1] 神薗雅紀，西田雅太，小島恵美，星澤祐二，“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,” CSS2011

[2] 宮元大輔，ブラン グレゴリー，秋山満昭，“抽象構文木を用いた Javascript ファイルの分類に関する一検討,”CSS2011

[3] ブラン グレゴリー，秋山満昭，宮元大輔，門林雄基，“難読化されたスクリプトにおける特徴的な構文構造のサブツリー・マッチングによる同定,”CSS2011

[4] Javascript parser - UglifyJS ‘‘`https://github.com/mishoo/UglifyJS`’’

[5] J.I.Munro, V. Raman : “Succint Representation of Balanced Parentheses and Static Trees,” SIAM Journal on Computing, 31(3):762-776,2001

[6] JSUNPACK ‘‘`http://jsunpack.jeek.org/`’’

[7] 畑田充弘，他，マルウェア対策のための研究用データセット 〜 MWS 2010 Datasets 〜, MWS2010, (2010 年 10 月)

[8] 畑田充弘，他，マルウェア対策のための研究用データセット 〜 MWS 2011 Datasets 〜, MWS2011, (2011 年 10 月)