情報処理学会論文誌 Vol.54 No.1 349–356 (Jan. 2013)

推薦論文

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

神薗 雅紀[1,a)] 西田 雅太[1] 小島 恵美[1] 星澤 裕二[1,2]

受付日 2012年4月12日, 採録日 2012年10月10日

概要：近年の不正サイトは，マルウェアなどにより自動生成されたポリモーフィックな JavaScript が利用され，他の URL に誘導する手法が多くみられる．著者らはこのような JavaScript の動的解析システムを開発したが，条件分岐やタイマ処理による遅延処理，さらにはイベント処理などにより JavaScript のすべての挙動を網羅して得ることができないという動的解析技術の課題も存在した．そこで本稿では動的解析技術を用いず，JavaScript を分析するための新たな特徴点として，抽象構文解析木を用いる手法を提案する．そして，この手法の検証として，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う．

キーワード：マルウェア，難読化 JavaScript，抽象構文木，動的解析，静的解析，ドライブバイダウンロード

Categorizing Hostile JavaScript Using Abstract Syntax Tree Analysis

Masaki Kamizono[1,a)] Masata Nishida[1] Emi Kojima[1] Yuji Hoshizawa[1,2]

Received: April 12, 2012, Accepted: October 10, 2012

Abstract: In the recent years, many hostile websites have been using polymorphic JavaScript in order to conceal its code. The author of this article had previously proposed and developed a system based on dynamic analysis to process and detect such types of JavaScript. However, a challenge often encountered with that approach is the mandatory preparation of very detail-oriented environments that may also require specific user-driven events for the hostile JavaScript to execute properly as it was designed to. As an alternative solution, this paper will propose the use of an abstract syntax tree based on structural analysis of polymorphic JavaScript to detect and categorize hostile JavaScript. This paper will also elaborate on test results based on the proposed algorithm.

Keywords: malware, obfuscated JavaScript, abstract syntax tree, dynamic analysis, static analysis, drive-by download

1. はじめに

近年のマルウェアを配布する不正サイトは，機械的に生成され難読化が施されたポリモーフィックな JavaScript を利用し，他の URL に誘導する手法が多くみられる．特に，Gumblar などに代表される Web サイトにインジェクトされた JavaScript は，同じアルゴリズムで難読化され，構造は似ているが使われる文字列が異なるため，検知が難しく爆発的に被害が拡大した．

これらのインシデントを詳細に把握し検知するために，著者らは難読化が施された JavaScript をエミュレーションし，難読化を解除する動的解析システム [1], [2] を開発した．開発システムは擬似的な Document Object Model（以下，DOM と称する）環境と関数フック処理を JavaScript

[1] 株式会社セキュアブレイン先端技術研究所
Advanced Research Laboratory, SecureBrain Ltd., Chiyoda, Tokyo 102–0083, Japan
[2] 横浜国立大学
Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
[a)] masaki_kamizono@securebrain.co.jp

本稿の内容は 2011 年 10 月のコンピュータセキュリティシンポジウム 2011（CSS2011）にて報告され，同プログラム委員長により情報処理学会論文誌ジャーナルへの掲載が推薦された論文である．

Interpreter 上に準備し，JavaScript をエミュレートすることで動的解析を実現する．そして，動的解析により抽出された難読化を解除した最終的な悪意のある JavaScript コードを取得し，どのような脆弱性を利用しているかを判別する．しかし，同システムは条件分岐やタイマ処理による遅延処理，イベント処理などにより JavaScript のすべての挙動を網羅して得ることができないという動的解析技術の課題も存在した．これにより，動的解析技術の向上とともに，動的解析技術を利用しない不正な JavaScript の検知および分析技術の向上が希求されている．

難読化が施されたポリモーフィックな JavaScript の特徴を分析すると，アルゴリズムにより機械的にコードが生成されているため文字列の変化による難読化のパターンは異なるが，その構造自体は変化していない．例をあげると，unescape 関数や replace 関数の引数に異なる値を設定することで，変数名をランダムにすることや bracket 書式を利用することなどで，難読化パターンを変更している [3], [4]．また，JavaScript の構造の一部だけを変更したものも存在する．つまり，引数の設定値や変数名を変更することで難読化のパターンを変更しているが，JavaScript そのものの構造は同じである．したがって，JavaScript の特徴を抽出することができれば，機械的に生成され難読化が施されたポリモーフィックな JavaScript を効果的に検知および分析できると考えられる．

そこで本稿では，抽象構文解析木を JavaScript の特徴点として扱う手法を提案する．そして，JavaScript から導出した抽象構文解析木がどのような特徴を表現するものであるかを検証するために，実際に自動生成され難読化が施されたポリモーフィックな JavaScript を利用して，その評価を行う．

2. 機械的に生成されたポリモーフィックな JavaScript

機械的に生成され難読化が施されたポリモーフィックな JavaScript の一例を**図 1**，**図 2** に示す．図 1，図 2 から分かるとおり，JavaScript の構造は同じであるが，関数オブジェクト名や関数の引数などが異なっており，これらが難読化の多様なパターンとなっている．また，一部構造が異なるだけの JavaScript も複数存在した．実際に Gumblar が猛威を振るった際は，このような同じ構造やよく似た構造の JavaScript が多数の正規サイトにインジェクトされ，難読化のパターンが異なるため検知が困難になり，被害が拡大した．

3. 関連研究

不正な JavaScript の検知手法としては，Curtsinger らによる Zozzle [5] があげられる．Zozzle は，JavaScript の構文要素を抽象構文解析木ととらえ，その階層的特徴をベイズ

```
try{
 window.onload = function(){
  var A84jbd5xsu = document.createElement('script');
  A84jbd5xsu.setAttribute('type', 'text/javascript');
  A84jbd5xsu.setAttribute('id', 'myscript1');
  A84jbd5xsu.setAttribute('src', 'h^&(t)$t(!^p($^(:$@(/!!)/@x!()@n&!$x&!!@x^!&(-(c
#)o#!m!!(.$n^!#(u^((.@)#n&!(!$@@.#^@@w$()3&)(-$o)#r$^{g}@!&.@#&g(^o!&!!)&d!
                                                                              
m@^@/##!)#r&@i!#n^$(c))!$o#&)n@(!d^&e&!#&@v&)a($$!g@^)#o#&^!.((^c@)&&
o!!)!m!/)$'.replace(/@|¥^|¥)|#|&|¥$|¥(|¥!/ig, '));
  A84jbd5xsu.setAttribute('defer', 'defer');
  document.body.appendChild(A84jbd5xsu);
 }
} catch(e) {}
```

図 1 機械的に生成されたポリモーフィックな JavaScript（pattern I）

Fig. 1 Polymorphic JavaScript generated mechanically (pattern I).

```
try{
 window.onload = function(){
  var Q236s4ic4454clw = document.createElement('script');
  Q236s4ic4454clw.setAttribute('type', 'text/javascript');
  Q236s4ic4454clw.setAttribute('id', 'myscript1');
  Q236s4ic4454clw.setAttribute('src', 'h(t)!^t^))p#@:&&/(##/&$#c^$$!^@)(i&(c$^)k))#
$s^o$#r!^)^-$$$&c@$o#^m$!#.#&(e((a!ls)(@t)&m((o@^^n!$!e&^&(y$#).#&c$@o$@!
                                                                              
e!#^@)s$.^c^&#o((&m&/)(&@!&()i(@n)(k$@h&e)@$(l)$p^!e)$!$r$#.)&c!&n($@/$g#o^
@&o!$$g$^!^&#@e$.&&!c#o@$$m(/$$'.replace(/¥(|¥!|&|#|¥$|¥)|@|¥^/ig, ''));
  Q236s4ic4454clw.setAttribute('defer', 'defer');
  document.body.appendChild(Q236s4ic4454clw);
 }
} catch(e) {}
```

図 2 機械的に生成されたポリモーフィックな JavaScript（pattern II）

Fig. 2 Polymorphic JavaScript generated mechanically (pattern II).

分類機を利用して悪意のある JavaScript であるか否か判定する手法である．Zozzle はブラウザの JavaScript エンジンを拡張する形で提案されており，難読化された JavaScript を実行し特定の関数をフックすることで得られる難読化を解除した JavaScript を解析対象としている．本稿は，Web サイトにインジェクトされた難読化 JavaScript そのものを対象とし，抽象構文解析木を利用してその特徴をとらえ，検知および分類することを目的とする．

また，抽象構文解析木を利用した JavaScript の分類では，Gregory ら [6]，宮本ら [7] の論文があげられる．Gregory らの論文は，スクリプトから学習した構造を部分木検索することによって，悪意のあるスクリプトに利用される難読化と，そうではない難読化について分類する手法を提案している．また，宮本らは，Chilowicz ら [8] の提案する抽象構文木に基づいて生成される Fingerprint を指標として用い，Gregory らの論文で使用された抽象構文解析木を利用して JavaScript の類似性を判定している．Chilowicz らの提案する抽象構文木に基づいて生成される Fingerprint とは，木構造における Node ごとに，その Node の重み，Node のハッシュ値，Node の名称，Node の親の名称という 4 つの値を文字列結合した内容により表現される．

Gregory ら，宮本らの手法は，抽象構文解析木を利用することで JavaScript の類似性や，悪意のあるスクリプトに利用される難読化と，そうでない難読化についての分類手法を提案している．

本稿は，上述のアプローチとは異なり，自動生成された悪意のあるポリモーフィックな JavaScript に着目する．そして，JavaScript の抽象構文解析木からさらに難読化のパターンに利用されている引数の設定値や変数名などを除外した抽象構文解析木を作成し，これを JavaScript の特徴点とする．この特徴点を利用して，自動生成され難読化が施されたポリモーフィックな JavaScript の検知および分類を実現する．

4. 抽象構文解析木による JavaScript の特徴抽出手法

続いて，本稿の提案手法である抽象構文解析木による JavaScript の特徴抽出手法を述べる．最初に，提案手法では JavaScript の構造を導出するため JavaScript Parser を用いて構文解析を行い，対象となる JavaScript の構文解析木を作成する．続いて，構文解析木の抽象化処理を行い，JavaScript の特徴点となる抽象構文解析木を導出する．

4.1 JavaScript の構文解析

JavaScript の構文解析木を導出するため，提案手法では JavaScript parser である RKelly [9] を利用する．RKelly は JavaScript を parse するライブラリであり，対象の JavaScript をインプットすると，Node の属性と Node の値からなる構文解析木をオブジェクトとして作成する．

次に，RKelly を利用して導出した構文解析木のオブジェクトを可視化する．構文解析木の可視化には Graphviz [10] を利用した．**図 3** に機械的に生成され難読化が施されたポリモーフィックな JavaScript を，**図 4** 左側に導出された構文解析木を示す．一部の Node の値は，文字数の関係から先頭部分のみを表示している．図 4 左側からも分かるとおり，当該構文解析木には関数名や引数の値などの情報が含まれているため，難読化のパターンが異なれば値も異なるが，構造は変化しない．それゆえ，難読化のパターンに依存することなく検知を実現するためには，次節で述べる抽象化を行う必要がある．

4.2 抽象構文解析木の導出

JavaScript の構造を特徴点とするために，4.1 節で作成した構文解析木の抽象化を行う．抽象化処理は各 Node の値を除外することで実現する．値を除外する Node の一例を**表 1** に示す．Node の分類は RKelly が定義した NodeName に従う．

図 4 左側の構文解析木に対して，上記の抽象化処理を施した，構文解析木を図 4 右側に示す．以上の処理により，

```
document.write(unescape('%3CspBscjBrjBispBppHtjB%20srCXHcjB%3DspB%2FpH%2FjB
t3y%3E').replace(/3y|CW|jB|pH|CXH|ze6|spB/g,""));
```

図 3 解析対象 JavaScript
Fig. 3 The target JavaScript.

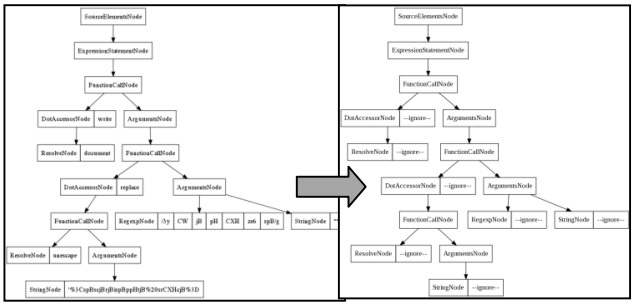


図 4 JavaScript の構文解析木例
Fig. 4 The syntax analysis tree and abstract syntax analysis tree of JavaScript.

表 1 値を除外する NodeName 一覧
Table 1 The NodeName list which excepts a value.

Node Name	概要
DotAccessorNode	オブジェクトが関数を利用する際の「.」表記の値を表す Node であり，値には関数名が入る．
EmptyStatementNode	「;」のみの処理が記載されていない Node であり，値は「;」が入る．
FunctionDeclNode	関数宣言を表す Node であり，値には関数名が入る．
NumberNode	数値オブジェクトを表す Node であり，値には数値が入る．
ParameterNode	関数の引数を表す Node であり，オブジェクト名が入る．
PostfixNode	演算子などの情報を表す Node であり，「++」などが入る．
RegexpNode	正規表現を表す Node であり，値には正規表現のルールが入る．
ResolveNode	関数を呼ぶ変数の名前を表す Node であり，値には変数名が入る．
StringNode	String オブジェクトを表す Node であり，値には実際の文字列が入る．
ThisNode	this を表す Node であり，値には「this」が入る．
TryNode	try/catch を表す Node であり，error オブジェクト名が入る．
VarDeclNode	変数宣言を表す Node であり，値には変数名が入る．
FunctionExprNode	function により関数（メソッド）定義を表す Node であり，値には「function」が入る．

構文解析木の各 Node の値を除外することで抽象化した構文解析木を，本稿では「抽象構文解析木」と定義する．

5. 抽象構文解析木を利用した JavaScript 比較処理

機械的に生成され，難読化が施されたポリモーフィックな JavaScript を検知および分類する際は，検査対象の

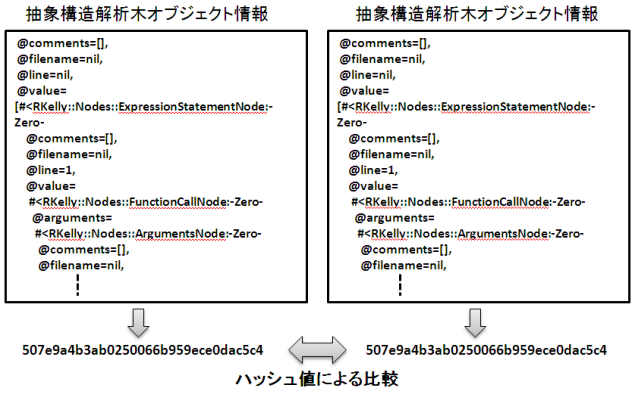


図 5 抽象構文解析木の比較方法

Fig. 5 The comparison method of an abstract syntax tree.

JavaScript の抽象構文解析木を導出し，すでに不正であると判断された抽象構文解析木と比較することで実現する．これにより，難読化のパターンに依存しない JavaScript の特徴による比較を実現する．

比較処理は抽象構文解析木の完全一致を効率的に判断する手法と，一部の Node の差異を吸収し，類似の抽象構文解析木を判定する簡易的な木探索アルゴリズムの 2 つの手法で実現する．

まず，前者の抽象構文解析木の完全一致を効率的に判断する手法は，4.2 節で述べたように，表 1 で示した Node の値を除外した抽象構文解析木オブジェクトを TEXT 化する．また，抽象構文解析木は Ruby 言語のオブジェクトとして作成しているため，オブジェクトごとに一意となるアイデンティティ（オブジェクト ID）が付与されている．アイデンティティ情報はハッシュ値による比較では不要な情報となるため，これらを削除し MD5 値を算出することで，抽象構文解析木の比較を実現する．提案手法による比較処理の概要を**図 5** に示す．本手法により，高速な比較を実現する．

次に，後者となる類似の抽象構文解析木を判定する簡易的な木探索アルゴリズムの手法を述べる．ただし，機械的に生成され難読化が施されたポリモーフィックな JavaScript は子 Node を持たない末端 Node に差異が多く見られたため，本手法は比較する抽象構文解析木どうしがどの程度近い構造であるか類似度を判定するのではなく，子 Node を持たない末端 Node を考慮し，どちらかの抽象構文解析木がもう一方に包含される，もしくは末端 Node のみが異なる形であるか判定するものとする．具体的には，図 5 に示すように抽象構文解析木オブジェクトは各 Node が子 Node を持つか否か，容易に判断できる形式となっている．つまり，子 Node を持たない Node が末端 Node と判断することができる．そして，Node 情報の前にあるスペースにより，その Node の深さを判別することが可能である．これらの形式を利用し，比較対象の TEXT 化した抽象構文解析木オブジェクトを検索し，同じ位置の NodeName を比較する．NodeName に差異が発生した場合，どちらかが末端 Node であれば，差異を無視する．これらの処理により，どちらかの抽象構文解析木がもう一方に包含される，もしくは末端 Node のみが異なる構成であるかを判定する．

表 2 評価環境

Table 2 Evaluation environment.

CPU	Intel Core i7
クロック	2.8 GHz
メモリ	8 GB 1333 MHz DDR3
仮想マシン	VMware Fusion 4.1.1
ホスト OS	Mac OS X Lion 10.7.3
ゲスト OS	Fedora（Linux version 2.6）

表 3 平均処理時間

Table 3 Average handling time.

分類	gumblar 系(s)	8080injection 系(s)
構文解析木オブジェクト作成処理	0.00524	0.10226
抽象構文解析木の導出処理	0.02431	0.41548
抽象構文解析木の画像作成処理	0.19403	3.28567
ハッシュ値生成処理	0.00232E-2	0.00247E-2

6. 分析事例

本章では提案手法により導出した抽象構文析木を利用して，自動生成され難読化が施されたポリモーフィックな JavaScript を検知することができるか，その有効性を検証する．今回利用するポリモーフィックな JavaScript は，独自のクライアント型ハニーポット [11] を利用して収集したもの，および “D3M_2011 データセット”[12] から抽出した gumblar 系 40 パターンと 8080 injection 系 68 パターンの計 108 パターンを対象とする．

評価環境を**表 2** に，各処理の平均処理時間を**表 3** に示す．平均処理時間は JavaScript から構文解析木オブジェクトを作成する処理，抽象構文解析木の導出処理，導出した抽象構文解析木を画像（png）形式で表記する処理，構文解析木オブジェクトのハッシュ値生成処理に要した時間に分けて表記する．表 3 より，抽象構文解析木の画像作成処理が他の処理より時間を要しているが，JavaScript の構造を画像で確認する必要がなくハッシュ値のみで比較する場合は，本処理を実施する必要はない．

6.1 分析事例 1

Gumblar 系のポリモーフィックコードを提案手法により分析する．**図 6**，**図 7** に Gumblar 系ポリモーフィックコードの比較対象とする JavaScript を示す．図からも分かるとおり，難読化のパターンはそれぞれ異なっている．

次に，上記 JavaScript から導出した抽象構文解析木を示す．2 つの JavaScript から導出した抽象構文解析木は一

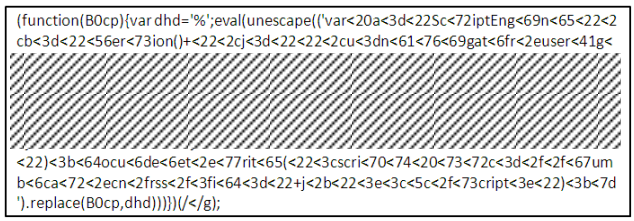


図 6 Gumblar 系 JavaScript（pattern I）
Fig. 6 JavaScript of a Gumblar system (pattern I).

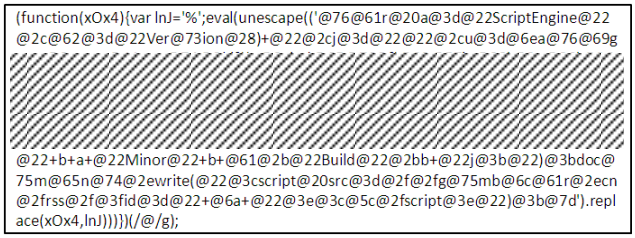


図 7 Gumblar 系 JavaScript（pattern II）
Fig. 7 JavaScript of a Gumblar system (pattern II).

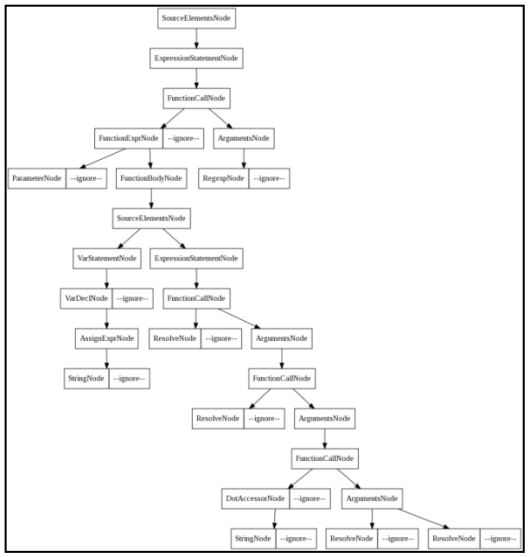


図 8 Gumblar 系 JavaScript 抽象構文解析木
Fig. 8 The abstract syntax tree of Gumblar system JavaScript.

致した．図 8 に一致した抽象構文解析木を示す．これにより，アルゴリズムにより機械的に生成されたポリモーフィックな難読化 JavaScript の構造をシグネチャ化することができたといえる．このシグネチャを利用することで，同アルゴリズムで生成された未知の JavaScript に対しても検知，分類が可能となる．

6.2 分析事例 2

次に，8080 injection のポリモーフィックコードを提案手法により分析する．図 9，図 10 に 8080 injection ポリモーフィックコードの比較対象とする JavaScript を示す．先ほどと同様，難読化のパターンはそれぞれ異なっていることが分かる．

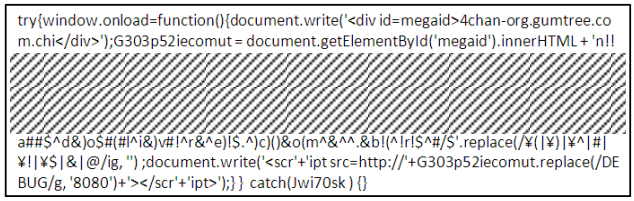


図 9 8080 injection 系 JavaScript（パターン I）
Fig. 9 JavaScript of a 8080 injection system (pattern I).

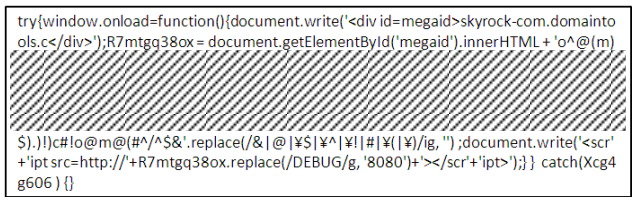


図 10 8080 injection 系 JavaScript（パターン II）
Fig. 10 JavaScript of a 8080 injection system (pattern II).

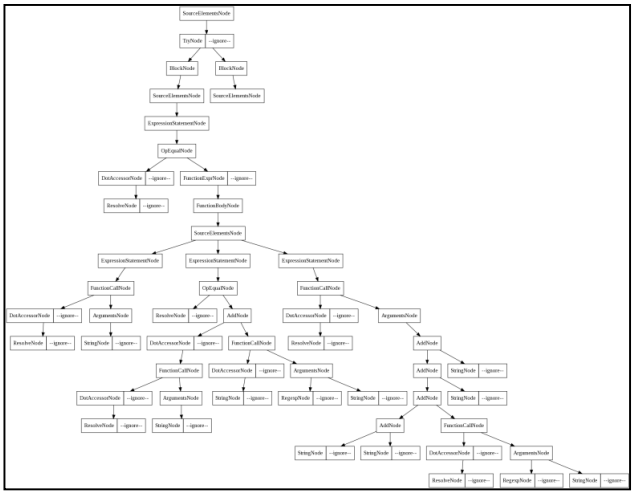


図 11 8080 injection 系 JavaScript 抽象構文解析木
Fig. 11 The abstract syntax tree of 8080 injection system JavaScript.

次に，提案手法により各ポリモーフィックコードの抽象構文解析木を導出する．2 つの JavaScript から導出した抽象構文解析木は一致した．抽象構文解析木を図 11 に示す．これにより 6.1 節と同様に，導出した抽象構文解析木をシグネチャとすれば，同じ構造の未知の 8080 injection ポリモーフィックコードを検知することが可能となる．

6.3 分析事例 3

6.2 節と同様，8080 injection ポリモーフィックコードを提案手法により分析する．図 12，図 13 に 8080 injection ポリモーフィックコードの比較対象とする JavaScript を示す．難読化のパターンはそれぞれ異なっていることが分かる．

次に，提案手法により各ポリモーフィックコードの抽象構文解析木を導出する．2 つの JavaScript から導出した抽

図 12 8080 injection 系 JavaScript（パターン I）
Fig. 12 JavaScript of a 8080 injection system (pattern I).

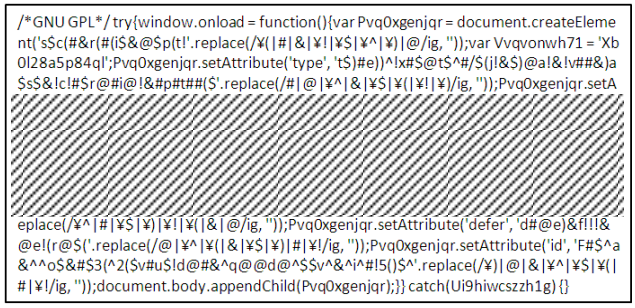


図 13 8080 injection 系 JavaScript（パターン II）
Fig. 13 JavaScript of a 8080 injection system (pattern II).

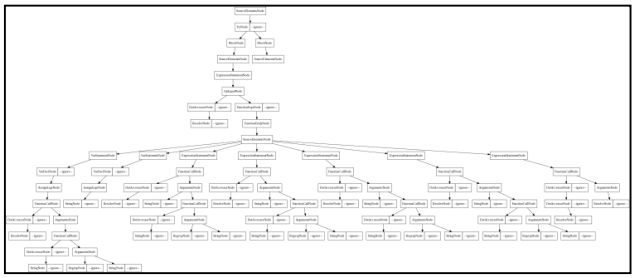


図 14 8080 injection 系 JavaScript 抽象構文解析木
Fig. 14 The abstract syntax tree of 8080 injection system JavaScript.

象構文解析木は一致した．抽象構文解析木を図 14 に示す．これにより 6.1 節 6.2 節と同様に，導出した抽象構文解析木をシグネチャとすれば，同じ構造の未知の 8080 injection ポリモーフィックコードを検知することが可能となる．

6.4 分析事例 4

図 15，図 16 に示す JavaScript は，構造が異なる Gumblar 系ポリモーフィックコードである．各 JavaScript の抽象構文解析木を図 17 に示す．抽象構文解析木で表現することで，構造が一部のみ変更されており，その他の箇所は構造が同じであることが容易に判断できる．

以上より，JavaScript だけでは判断が困難であるが抽象構文解析木で構造を表現することで，構造が似ている JavaScript を分類することができた．したがって，導出した抽象構文解析木をシグネチャとすれば，アルゴリズムを

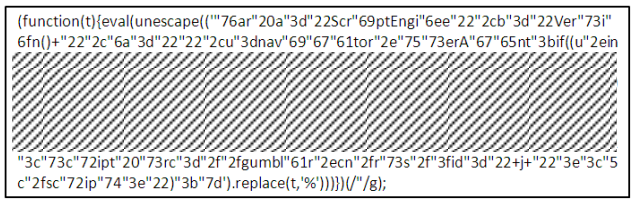


図 15 Gumblar 系 JavaScript I
Fig. 15 Gumblar system (pattern I).

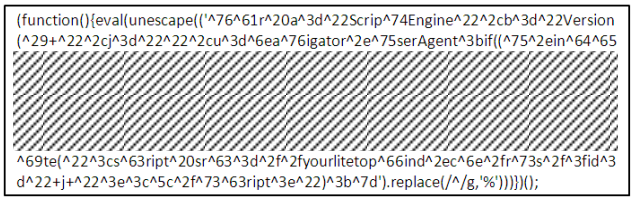


図 16 Gumblar 系 JavaScript II
Fig. 16 JavaScript of a Gumblar system (pattern II).

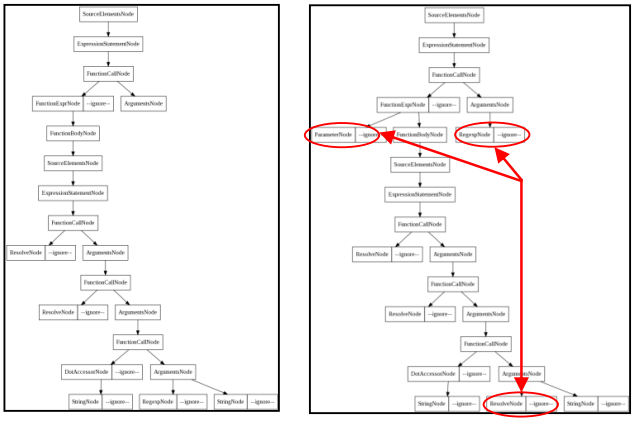


図 17 Gumblar 系 JavaScript I/II 抽象構文解析木
Fig. 17 The abstract syntax tree of Gumblar system JavaScript I/II.

変更され構造の一部が変更された JavaScript に対しても，検知や分類ができるといえる．また本抽象構文解析木は，5 章で述べた類似の抽象構文解析木を判定する簡易的な木探索アルゴリズムにより，末端 Node のみが異なるものであると判定することができた．

7. 考察および今後の発展

6 章の分析事例より，提案手法を用いて JavaScript の特徴点を抽出し，同じ構造の JavaScript を検知できることを証明した．さらに，抽象構文解析木を利用することで，構造が似ている JavaScript の判定も容易に実現した．また，今回の分析調査では，各々異なる 108 パターンのポリモーフィックな JavaScript に対して，提案手法により抽象構文解析木を導出すると 65 パターンまで集約することができた．構造が似たものを含めるとさらに集約することが可能である．これにより，本提案は機械的に生成され難読化が施されたポリモーフィックな JavaScript に対し，検知およ

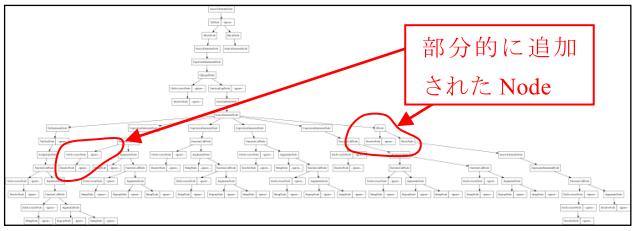


図 18 8080 injection 系 JavaScript 抽象構文解析木
Fig. 18 The abstract syntax tree of 8080 injection system JavaScript.

び分類としての特徴点を表す有効的な手法であるといえる．

また，5 章で述べた単純な木探索アルゴリズムを利用して構造が近い JavaScript の判定を試みたが，集約できたのは図 17 で示す 1 パターンのみであった．この理由としては，抽象構文解析木の差異が末端 Node のみではなく，特定の深さまではまったく同じであるが途中の Node 以降がすべて改変されているものや，途中に異なる Node が組み込まれたものが多く存在したためである．図 14 の 8080 injection 系 JavaScript において構造が近い例を**図 18** に示す．特定の Node 以降に新しい Node 構造が追加され，さらに Node の途中に新しい Node が追加されていることが分かる．このような特定の Node 以降が異なるものや，途中に異なる Node が組み込まれているものを類似パターンであると仮定すると，108 パターンのポリモーフィックな JavaScript は 43 パターンまで集約できることが分かった．ただし，このような集約は，5 章で述べた簡易的な木探索アルゴリズムでは検知が困難であるため，これをふまえ提案技術の今後の展開を以下に考察する．

本提案技術の今後の発展としては，抽象構文解析木の部分マッチにより汎用的に JavaScript の分類や検知を実現すること，および，抽象構文解析木のさらなる抽象化である．まず前者は，不正な JavaScript に共通する構造などがある場合は，その共通部分のみを判定することで，より汎用的に分類や検知を実現できると想定される．また，本稿で分析したようなインジェクションされた JavaScript の判定だけでなく，細かいコードスニペットを抽象構文解析木として扱い，その組合せにより不正判定や機能分類を実現できると考えられる．**図 19** に heap spray 攻撃や shellcode を含む不正な JavaScript を示す．たとえば，インジェクトされた JavaScript の heap spray 攻撃部の抽象構文解析木を導出する．この導出した抽象構文解析木と解析対象とする JavaScript の抽象構文解析木とを部分比較し，同様の構造が検出されれば，対象としている JavaScript は heap spray 攻撃を行うものであると判断できる．shellcode なども同様である．つまり，様々な JavaScript の特徴を導出し，部分比較を行うことで，どのような処理を行う JavaScript であるか，分類や検知を実現できると想定される．ただし，heap spray 攻撃のみでも構造が完全に一致するケースは少

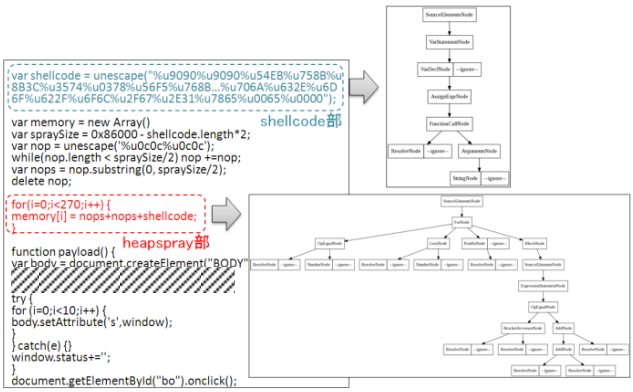


図 19 shellcode や heapspray などの抽象構文解析木例
Fig. 19 The example of an abstract syntax analysis tree of shellcode/heapspray.

なく，似た構造を持つ場合が多く存在した．そこで，先ほどの発展の後者である抽象構文解析木のさらなる抽象化となる．たとえば，本稿で提案したように構造解析の値のみ除外するのではなく，特定の構造も除外することで，抽象構文解析木をさらに抽象化できると想定される．そして，抽象構文解析木の構造そのものも抽象化することで，構造が似ているものを判定し，より汎用的に不正な JavaScript の分類や検知を実現できると考えられる．

また，本稿では Node の値を一律に除外しているが，検知や分類の精度向上，そして誤検知対策として任意の Node 情報を残して比較することも実現している．ただし，どの Node 情報を残せば効果的に検知や分類できるかは，今後調査する必要がある．これらの発展課題を今度の研究課題として進めていきたい．

8. おわりに

本稿では，JavaScript から特徴点を抽出する手法を提案し，実際に自動生成されたポリモーフィックな JavaScript に対し，分類および検知という観点において提案手法が特徴点となることの有効性を証明した．また，本稿は JavaScript から特徴点を抽出する手法の提案を主眼としているため，7 章で述べた今後の発展に重きをおき，研究を進めていくことを今後の課題とする．

参考文献

[1] 神薗雅紀，西田雅太，星澤裕二：動的解析を利用した難読化 JavaScript コード解析システムの実装と評価，MWS2010 (2010).
[2] 神薗雅紀，西田雅太，星澤裕二：動的解析を利用した PDF マルウェア解析システムの実装と評価，ICSS2011 (2011).
[3] 神薗雅紀，星澤裕二：ウェブ改ざんはこうして起こるその攻撃手法の解説，AVAR2009 IN KYOTO (2009).
[4] 神薗雅紀，西田雅太，小島恵美，星澤裕二：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，MWS2011 (2011).
[5] Curtsinger, C., Livshits, B., Zorn, B. and Seifert, C.: Zozzle: Low-overhead Mostly Static JavaScript Malware

Detection, No.MSR-TR-2010-156 (Nov. 2010).
[6] ブラン・グレゴリー，宮本大輔，秋山満昭：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，MWS2011 (2011).
[7] 宮本大輔，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，MWS2011 (2011).
[8] Chilowicz, M. et al.: Syntax tree nger printing: A foundation for source code similarity detection, *Proc. IEEE ICPC*, pp.243–247 (May 2009).
[9] Parsing Javascript Parser – Rkelly, available from ⟨http://tenderlovemaking.com/2007/12/24/parsing-javascript-parser.html⟩.
[10] Home Graphviz - Graph Visualization Software, available from ⟨http://www.graphviz.org/⟩.
[11] 星澤裕二，川守田和男，太刀川剛，神薗雅紀：自律型クライアントハニーポットの提案，ICSS2009 (2009).
[12] 畑田充弘，中津留勇，秋山満昭：マルウェア対策のための研究用データセット—MWS 2011Datasets，MWS2011 (2011).

推薦文

マルウェア配布に利用される不正サイトにおいて，難読化されたポリモーフィックな JavaScript が，攻撃検知の妨げとなっている．本稿は，JavaScript の構文解析木の各 Node の値を除外した抽象構文解析木の使用と，抽象構文解析木のオブジェクト情報のハッシュ値比較による高速な比較・分類が提案手法の特徴である．実際の攻撃に使われる難読化されたポリモーフィックな JavaScript の具体例をもとに提案手法における解析結果の事例を複数示し，有効性の高さがうかがえるため，推薦論文として推薦する．

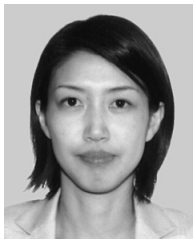
（コンピュータセキュリティシンポジウム 2011 プログラム委員長　四方順司）

神薗 雅紀 （正会員）

2003 年徳島大学工学部知能情報工学科卒業．2005 年同大学院修士課程修了．2009 年（株）セキュアブレイン入社．主に不正サイトの検知・分析，およびマルウェアの静的・動的解析に関する研究開発に従事．

西田 雅太

2002 年電気通信大学電気通信学部情報工学科卒業．2004 年同大学院修士課程修了．2009 年（株）セキュアブレイン入社．主に不正サイトの検知・分析，およびマルウェアの静的・動的解析に関する研究開発に従事．



小島 恵美

1996 年お茶の水女子大学理学部数学科卒業．2008（株）セキュアブレイン入社．主に官公庁，民間企業向けのセキュリティコンサルティングに従事．

星澤 裕二 （正会員）

2004 年 10 月から（株）セキュアブレイン入社．日本におけるウイルス研究の第一人者としての地位を確立し，コンピュータのセキュリティに関して多くの IT 関連出版物に寄稿している．また，Virus Bulletin や EICAR，AVAR 等の国際会議でセキュリティ問題に関する研究発表も行っている．著書に『ウイルスの原理と対策』，『アナライジング・マルウェア』．2007 年に経済産業省商務情報政局長表彰を受賞．