抽象構文木を用いた Javascript ファイルの分類に関する一検討

宮本 大輔 †　　ブラン グレゴリー ‡　　秋山 満昭 §

† 東京大学
113-8658 東京都文京区弥生 2-11-16
daisu-mi@nc.u-tokyo.ac.jp

‡ 奈良先端科学技術大学院大学
630-0192 奈良県生駒市高山町 8916-5
gregory@is.naist.jp

§NTT 情報流通プラットフォーム研究所
180-8585 東京都武蔵野市緑町 3-9-11
akiyama.mitsuaki@lab.ntt.co.jp

近年，悪意のあるウェブページが様々な問題を引き起こすことが知られており，この対策が急務である．しかし，これらのウェブページは難読化された Javascript によって記述されることが多く，問題の対策への大きな妨げとなっている．悪意のあるウェブページの解読を効率的に行うため，本研究では抽象構文木を使ったプログラムの類似性を測定する研究に着目する．そして，Chilowicz [1] ら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，マルウェア対策のための研究用データセットから抽出した Javascript コードの比較及び分類を試み，この結果を分析する．

A consideration for categorizing Javascript files based on Abstract Syntax Tree Fingerprinting

Daisuke Miyamoto †　　Gregory Blanc ‡　　Mitsuaki Akiyama §

†The University of Tokyo
2-11-16 Yayoi, Bunkyo, Tokyo 113-8658, JAPAN
daisu-mi@nc.u-tokyo.ac.jp

‡Nara Advanced Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192, JAPAN
gregory@is.naist.jp

§NTT Information Sharing Platform Laboratories
3-9-11 Midorimachi, Musashino, Tokyo 180-8585, JAPAN
akiyama.mitsuaki@lab.ntt.co.jp

Abstract In this paper, we discuss a classification method for obfuscated Javascript source codes. Recently, malicious webpages often employs obfuscation techniques based on Javascript to make it difficult to analyze the threat. This paper focuses on identifying the types of obfuscation techniques, rather than decoding the obfuscated code. Our proposed method uses Abstract Syntax Tree(AST), which is designed for detecting the similarity of multiple source code. We employ Chilowicz's [1] AST Fingerprinting as a metric of the similarity, compare and categorize source codes extracted from MWS research dataset, and show the results of our analysis.

1　はじめに

近年，悪意のあるウェブページが様々な問題を引き起こしている．例えば，エンドユーザのコンピュータをマルウェアに感染させる Drive-by-Download 攻撃が知られているが，この攻撃は悪意のあるウェブページを介して実行される．また，エンドユーザを騙すことによりエンドユーザの個人情報を盗むフィッシング攻撃では，あたかも本物のウェブページのように見える偽サイト，すなわちフィッシングサイトが用いられる．

こうした悪意のあるウェブページが，果たしてどのような悪意を抱いているのかを解析する必要は急務である．本来，ウェブページは HTML 文法で記述されており，その解析は容易と思われてきた．しかし近年は Javascript に代表されるスクリプト言語によってウェブページを難読化する技術が広く知られている．悪意のあるウェブページもこうした難読化技術を採用する傾向があり，悪意の解析は困難なものとなりつつある．

そこで我々の研究グループは，難読化された Javascript に使われた難読化手法の特定を目的とする．難読化技術も多様であるとされているが，仮に複数のソースコードが同じ手法によって難読化されていた場合，その難読化されたソースコードには何らかの類似性を持つのではないかと考えられる．そこで，我々はプログラムのソースコードの類似性を発見する研究領域に着目した．この分野の先行研究は，複数のソースコードの類似性を何らかの指標によって計測することにより，ソースコードの盗用を発見するために用いられている．

本研究では Javascript コードの抽象構文木の Fingerprinting を用いて解析する手法 [1] に着目する．この構造の類似性を調べることにより，難読化技術の特定，あるいはファイルに含まれる悪意の内容を調査できるかについて，実験を通じた検討を行う．

以下，2 節では関連研究を，3 では比較実験について述べ，データセットの分類を 4 節で試みる．5 節において考察を行い，最後に 6 節にまとめと今後の課題を述べる．

2　関連研究

複数のプログラムのソースコードの間に類似性を見出す最も簡単な手法は，プログラムのソースコードにおいて同じ文字列を使っているものがないかを比較するという方式である．この方式では，例えば変数名などの識別子を変更することで全く異なるソースコードとして判断され得るため，ソースコードの盗用を検知するといった目的には性能が不十分であるとされている．より近代的な類似性判定の研究としては，トークンによるもの，プログラム依存グラフによるもの，そして抽象構文木 (Abstract Syntax Tree, AST) を用いるものが挙げられる．

トークンによる手法 [2,3] には，プログラムを抽象構文で表現した上で，各構文をトークン化する特徴がある．抽象構文を用いることにより，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ため，文字列の名称変更といった類似性判定を逃れる行為についての耐性があると考えられている．なお，抽象化された文字列を比較する例としては，レーベンシュタイン距離を用いて文字列同士を比較するといった方式が挙げられる．

プログラム依存グラフとは，プログラム内における各文の間の依存関係を有向グラフによって表現したものである．ここで表現される依存関係とは，プログラムの制御における関係やデータの定義や参照といった関係である．ただし，複雑なプログラムは膨大なグラフを持つため，この検索は容易ではないとされており，比較対象とするグラフの数を減らす研究 [4] が行われている．

AST とは，プログラムを文，式，識別子の単位で抽象構文として構成される構文木である．Chilowicz らは [1]，抽象構文木における部分木同士の検索という点において，トークンによる手法より柔軟性が高い検索ができるとし，AST に基づいた類似性の調査を行う AST Fingerprinting(ASTF) 方式を提案している．この方式では，木構造におけるノードごとに，そのノードの重み，ノードのハッシュ値，ノードの名称，ノードの親の名称という 4 つの値を文字列結合した内容を ASTF として用い，木構造同士の部

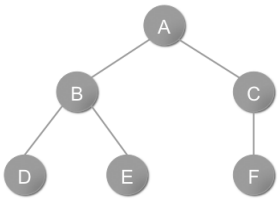

図 1: AST の例

```
function test() {
 var a = "test";
 var b = "test";
 if (a == b){
  document.write("abc");
 }
}
```

図 2: ソースコード J1

分的な検索を行う手法を提案している．

例として図 1 に示す AST の ASTF を作成する．ノードの重みとは，このノード配下の子の総数に 1 を加算したものであり，ノード A の重みは 6，ノード B の重みは 3 である．次に各ノードのハッシュ値を計算する．Chilowicz らの提案では，ハッシュ関数 $\mathcal{H}$ として SHA1 や MD5，Karp-Rabin [5] などのアルゴリズムを用い，後方のノードから計算していく．図 1 の場合では，最下層のノードである D，E，F の抽象構文における名称のハッシュ値 $\mathcal{H}(D)$，$\mathcal{H}(E)$，$\mathcal{H}(F)$ を最初に計算する．ノード B のハッシュ値は，ノード B の名称 B 及び $\mathcal{H}(D)$，$\mathcal{H}(E)$ を文字列結合し，その値をさらにハッシュ化したもの，すなわち $\mathcal{H}(B+\mathcal{H}(D)+\mathcal{H}(E))$ となる．同様に，ノード A のハッシュ値は $\mathcal{H}(A+\mathcal{H}(B)+\mathcal{H}(C))$ である．ノードの名称は A は A，というように自明である．ノードの親の名前は，ノード B からすればノード A となり，ノード A は根にあたる部分であり，便宜上，ハイフン記号 $(-)$ によって表現される．

Chilowicz らは，これら 4 つの値を各ノード毎に計算し，さらに文字列結合した値を AST Fingerprint(ASTF) と定義している．例えばノード A の場合は，$(6, \mathcal{H}(A), A, -)$，ノード B の場合は $(3, \mathcal{H}(B), B, A)$ となる．

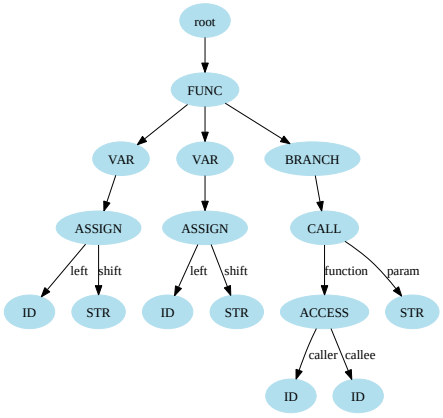

図 3: ソースコード J1 の AST

3 比較実験

3.1 予備実験

本節では ASTF の可用性を調べるため，3 つの Javascript のソースコードを用いて予備調査を行う．図 2 に示すソースコード J1 は簡易な Javascript の例であり，ソースコード J2 は，ソースコード 1 に小さな関数を一つ加えたものである．ソースコード J3 は，ソースコード 1，2 とも全く異なるものであり，Google のホームページに用いられているものを使用した．これらのソースコード群に対して複数の難読化技術を試すことにより，ソースコードが似ている場合であっても難読化技術が異なる場合は類似性が低くなり，ソースコードが似ていない場合であっても難読化技術が同じ場合は類似性が高くなるか否かを調査する．

難読化技術としては，CuteSoft 社の Free Javascript Obfuscator(FJO) [6] と Daft Logic 社の Online Javascript Obfuscator(OJO) [7] を用いた．前者では変数を別の名前に置き換えることにより難読化が行われ，後者ではプログラムが符号化され，この内容を復号しつつ最終的に得られた文字列を Javascript の `eval` 関数によって実行するという難読化が行われていることを確認した．便宜上，ソースコード 1 を J1，FJO により難読化したものを J1’，OJO により難読化したものを J1” と呼ぶ．

次に，9 種類のソースコードから AST 及び ASTF を作成する．本研究では Gregory [8] ら

表 1: AST Fingerprint の度数分布による比較

	J1 - J2	J1 - J2'	J1 - J3	J1" - J3"
$5 \leq w < 10$	2	0	0	2
$10 \leq w$	0	0	0	3

の実装を使用し，図 3 に示すような AST を作成する．ASTF の作成には，ハッシュ関数として SHA1 を用いる．また，ハッシュ値を計算する際に用いるノードの名称は `BRANCH` や `LOOP` など，実装 [8] が用いる抽象構文を用いる．この結果，ソースコード 1 から 9 個の，FJO により難読化したものから 15 個の，OJO により難読化したものから 26 個の ASTF が得られた．

二つのスクリプトの類似性の指標について，Chilowicz [1] らは，一致した ASTF についての重みをしきい値に分けて区分している．例えば表 1 は，一致した ASTF の重み w の値が，5 以上から 10 未満のもの，10 以上のもので分類している．例えば J1 から派生した J2 は似たソースコードであり，難読化を行う前は何らかのの同一性があることが伺えるが，J1 と FJO により難読化を行った J2" は同一性の部分が損なわれる．一方で，J1 と J3 は全く異なるソースコードであるが，どちらも OJO で難読化を行ったところ，同一の部分が多くみられる．

ただし，Chilowicz らはしきい値の決め方については定義しておらず，値は経験的にしか決まらないと考えられる．そこで，本研究ではより簡単な類似性の指標として，二つのファイルから生成される ASTF の最も高い重みを調べる．しかし，木構造のノード数が増えるた場合，全体的には一致していなくても重みが高くなる可能性がある．反対に，少ない重みの部分木が多数一致しているような木構造の場合，全体的には一致していても重みが低くなる可能性がある．この点を考慮し，全体的な一致率についても同様に調べる．例えば，J1 と J2 から作成した ASTF 間において，一致した ASTF の中で重みが最も高いものは 15 であった．また，一意な ASTF は，J1 に 11 個，J2 に 12 個みられ，そのうち 10 個が共通していた．この二つの木構造における各ノードを全体としてみた場合，一致する割合は 0.77 $(= 10/(11 + 12 - 10))$ となる．この場合，J1 と J2 の類似性を (15, 0.77) と設定する．また，J1" と J3" では (13, 0.43) であり多くの一致がみられる．ただし，J1' と J3' の類似性 (5, 0.06) のように，難読化技術が同じだからとはいても似た構造となるとは言えない場合があった．

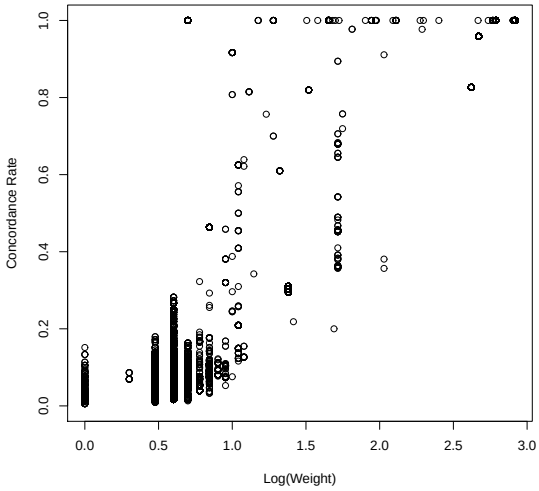

図 4: 研究用データセットの重みと一致率

3.2 研究用データセットによる比較実験

次に，研究用データセットである D3M 2011 [9] の検体より Javascript と思わしき 252 ファイルを個別に保存した．これらのうち，78 ファイルはその内容が全く同じものであった．

そこで残りの 174 ファイルについて ASTF を比較する．まず，二つのファイル間で一致した ASTF について，最も高い重みのものを調べる作業を 15051 $(= 174 \times 173/2)$ 回行った．この観測された重みの平均は 38.06，標準偏差は 142.70 であった．観測された中で最も高い重みは 829 であった．一致率についても同様に調査を行ったところ，平均は 0.15，標準偏差は 0.24，最大は 1.00 であった．図 4 に重みと一致率についての関係を示す，二つのファイルの ASTF 間において，x 軸は一致した ASTF における最も高い重みについての底を 10 とする対数であり，y 軸は ASTF の一致率を示す．一致率が高い ASTF であれば重みも高くなり，反対に一致率が低いほど重みも低くなるという傾

表 2: 研究用データセットのクラスタ分析結果

クラスタ	サンプル数	類似性
1	5	(584, 1.00)
2	5	(806, 1.00)
3	4	(588, 1.00)
4	1	-
5	2	(194, 0.98)
6	1	-
7	1	-
8	23	(612, 1.00)
9	23	(829, 1.00)
10	109	(6.33, 0.15)

向が観測された．なお，一致する ASTF の重みが 829 となるファイル間では，ASTF だけでなく Javascript ファイルの内容も類似していることを確認した．同様に，一致率が同じ場合も，同様に類似を確認した．従って，ASTF の一致した重みや ASTF の一致率から，類似性の有無を調べることは可能であると思われる．

4 研究用データセットの分類

本節では ASTF を用いたクラスタ分析を行い，データセットの分類を試みる．本研究では，クラスタ分析の際に必要となる特徴変数に，ASTF の有無に基づいた値を設定した．まずデータセット全体で一意な ASTF を調べ，その個数を特徴変数の数とする．ここで，任意の $\mathrm{ASTF_i}$ に対し，各ファイル毎の ASTF における $\mathrm{ASTF_i}$ の出現回数 n_i を調べる．この値に，$\mathrm{ASTF_i}$ の重み w_i を乗算した値を，特徴変数の値とする．

例えば，D3M データセットにおける 174 ファイルから得られた ASTF の一意な値を調べたところ，1826 種類の ASTF が得られた．仮に $\mathrm{ASTF_i}(1 \leq \mathrm{i} \leq 1826)$ の重み w_i が 10 とすると，任意のファイルに $\mathrm{ASTF_i}$ に該当するノードが 3 箇所発見された場合，ファイルの特徴変数の値は 30 $(= 10 \times 3)$ となる．このように値を計算し，174 行 1826 列からなる特徴ベクトルを得て，EM 法 [10] によるクラスタ分析を行った．なお，今回の実験ではクラスタ数を 10 とした．

分析結果を表 2 に示す．クラスタ 10 を除き，複数のファイルが分類されているクラスタでは，そのクラスタ内におけるファイル間の類似性が高くなっている．なお，ASTF によらず直接ファイルの内容を目視し，またブラウザのレンダリングエンジンにより動的解釈した結果を観測したが，このファイル群の内容及び悪意の内容も類似していることが確認された．例えばクラスタ 5 に分類された 2 個のファイルは，符号化されたデータを `getElementByID` メソッドを通じて HTML 文書から取得，この内容を解釈し，新たに Javascript で書かれたファイルを出力するという内容であった．この出力結果も似通っていたが，どちらも内容は Drive-by-Download 攻撃を行うものであった．

なお，クラスタ内ではなくクラスタ間については，類似性はあまり発見されなかった．ただし，クラスタ 1 とクラスタ 8，クラスタ 2 とクラスタ 9 では，それぞれ類似性が (418, 0.83) と (469,0.96) となり，似通っていた．クラスタ 1 で用いられている関数名と，クラスタ 8 で用いられている関数名にも一致する箇所がみられた．また，クラスタ 2 とクラスタ 9 では同じライブラリを用いている様子が見られた．本研究の範囲ではないが，難読化技術の亜種の判定が可能かについても検討したい．

5 考察

本研究では，同じ難読化手法を用いたファイルは，その構造が似てくるという仮説のもと，その構造解析を行っている．しかし， 3.1 節に述べた予備実験では異なるコード J1' と J3' が同じ難読化手法を用いても，重み，一致率が (5, 0.06) であり似通っているとは言いがたい．これは，全く違うソースコードの関数名や変数名を置き換えても全く違うソースコードになるためである．ASTF が扱いやすい難読化方式，そうでない方式を発見する必要があると思われる．

また，ASTF 同士の比較による類似性から，クラスタ分析による分類を試みている．この問題の課題としては，最適なクラスタ数の選定が挙げられる．例えば表 2 の結果では，クラスタ 10 に 109 ファイルが分類されているが，これらのファイルをさらに分類するべく，クラスタ数の再定義が必要になると思われる．また，クラスタに分類して，プログラムの類型化，ひいて

は難読化手法の類型化を行っている．しかし，類似しているか否かの判断は主観的な評価に留まっている．Javascript の動的解析の併用や，より多くの専門家らに判断してもらう等，こうした判断結果の信頼性を高める必要があると思われる．

6 おわりに

本研究では，Javascript ファイルの類似性を分析するため，プログラムの抽象構文木 (AST) から，Chilowicz [1] らの提案する AST Fingerprint(ASTF) を算出し，ASTF 間の比較を行った．対象としたファイルは，D3M 2011 の攻撃検体から抽出した 174 ファイルであり，指標としては二つのファイルにおいて一致した ASTF のうち最も高い重みと一致率，すなわち一意な ASTF 集合に対する一致した集合の占める割合を用いた．

この結果，重みが高い場合，一致率が高い場合には ASTF の類似性が確認できた．この結果を受けて ASTF を用いたクラスタ分析を行ったところ，同じクラスタに分類されたファイルでは，重みと一致率が高いだけでなく，その内容も似通っていることを確認した．

本研究の課題としては，考察で述べた課題に加え，多くのデータセットでの追試が考えられる．また，無意味なプログラムを挿入し，構文木を故意に歪めることにより，同じ悪意をもつプログラムの類似性を低くする，あるいは全く違う悪意を持つプログラムの類似性を高くするといった妨害行為が考えられる．そうしたノードの除去，あるいは各ノードにおける抽象構文の意味上の概念から重みを調節するといった対策を検討したい．

参考文献

[1] M. Chilowicz et al. Syntax tree fingerprinting: a foundation for source code similarity detection. In *Proceedings of IEEE ICPC*, pp. 243–247, May 2009.

[2] T. Kamiya et al. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. on Software Engineering*, Vol. 28, No. 7, pp. 654–670, July 2002.

[3] L. Prechelt et al. Finding Plagiarisms among a Set of Programs with JPlag. *J. of Universal Computer Science*, Vol. 8, No. 11, pp. 1016–1038, 2000.

[4] C. Liu et al. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *Proceedings of ACM SIGKDD ICKDDM*, pp. 872–881, August 2006.

[5] R. M. Karp et al. Efficient randomized pattern-matching algorithms. *IBM J. of Research and Development*, Vol. 31, pp. 249–260, March 1987.

[6] CuteSoft Components Inc. Free Javascript Obfuscator. `http://javascriptobfuscator.com`.

[7] Daft Logic. Online Javascript Obfuscator. `http://www.daftlogic.com/projects-online-javascript-obfuscator.htm`.

[8] G. Blanc et al. Identifying Characteristic Syntactic Structures in Obfuscated Scripts by Subtree Matching. マルウェア対策研究人材育成ワークショップ, 2011 年 10 月. (to appear).

[9] 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 〜. マルウェア対策研究人材育成ワークショップ, 2011 年 10 月. (to appear).

[10] A.P. Dempster et al. Maximum Likelihood from Incomplete Data via the EM Algorithm. *J. of the Royal Statistical Society Series(B)*, Vol. 39, No. 1, pp. 1–38, 1977.