# 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

ブラン グレゴリー† 秋山 満昭‡ 宮本 大輔§ 門林 雄基†

†奈良先端科学技術大学院大学 情報科学研究科
〒 630-0192 生駒市高山町 8916-5
{gregory,youki-k}@is.aist-nara.ac.jp

‡NTT 情報流通プラットフォーム研究所
〒 108-8585 東京都武蔵野市緑町 3-9-11
akiyama.mitsuaki@lab.ntt.co.jp

§東京大学 情報基盤センター
〒 113-8658 東京都文京区弥生 2-11-16
daisu-mi@nc.u-tokyo.ac.jp

**あらまし** 悪意のあるウェブサイトを解析する研究は，サイトが難読化されたスクリプトを用いている場合に悪意があるとみなす傾向にあるが，難読化は必ずしも悪意を持って用いられてはいない．本研究で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，そうでない難読化についての分類手法を提案する．本手法は，情報量の増加を抑制するために抽象構文木を用い，頻出する木構造を探索して学習を行う．また，抽象構文木の同定を行う検査では，プッシュダウン・オートマトンを用いた，複数の木構造の部分木検索を試みる．このように自然言語処理及び文字列処理の技術を用い，分類結果を考察する．

# Identifying Characteristic Syntactic Structures in Obfuscated Scripts by Subtree Matching

Gregory Blanc† Mitsuaki Akiyama‡ Daisuke Miyamoto§
Youki Kadobayashi†

†Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-0192, JAPAN
{gregory,youki-k}@is.aist-nara.ac.jp
‡NTT Information Sharing Platform Laboratories
3-9-11 Midorimachi, Musashino, Tokyo 180-8585, JAPAN
akiyama.mitsuaki@lab.ntt.co.jp
§Information Technology Center, The University of Tokyo
2-11-16 Yayoi, Bunkyo, Tokyo 113-8658, JAPAN
daisu-mi@nc.u-tokyo.ac.jp

**Abstract** Many approaches to web malware detection tend to consider obfuscated scripts as malicious, although it is has been demonstrated that obfuscator does not indicate malice. In a bid to distinguish obfuscation techniques used in malicious and benign scripts, we propose a subtree matching technique to identify learned structural patterns in analyzed scripts. Our proposal implements two techniques from the realm of natural language processing and string algorithms. We advocate the use of abstract syntax trees to reduce the entropy introduced by string randomization and rather focus on code structure patterns. In a learning phase, we discover frequently occurring trees in abstract syntax treebanks, while in the testing phase, we attempt to identify these trees within a candidate AST by using a pushdown automata that accepts the set of learned trees.

# 1 Introduction

Though obfuscation in scripting contents has been studied for some time now, it still remains a difficult task to reverse it. Of course, simple schemes can easily be defeated by hooking critical sinks and executing the then-modified program. However, more advanced and sophisticated techniques increase the complexity of the output to the point where some obfuscated contents need to be brute-forced. Obfuscation is further hardened through the use of redirection techniques: obfuscated contents are not provided as a single piece of code to reverse but are often scattered among several files and origins, the original script rebuilding itself through several layers of deobfuscation and linking.

Readers not familiar with the matter of obfuscation in web scripting languages can refer to a survey of obfuscation schemes[1] and this taxonomy compiled by Collberg et al.[2] where obfuscating transformations are divided into 3 main categories: layout obfuscations which remove formatting information from the source code; data obfuscations which obscure data and data structures (storage, encoding, aggregation or ordering); control-flow obfuscations which affect the aggregation, ordering or computations of the flow of the code. We are focusing on the last two types. What we propose is to identify these obfuscating transformations by exhibiting characteristic structures only found in these.

As an example, let's take the well-known Dean Edwards' packer which is used among both benign and malicious scripts. Even though it is insufficient to detect it to decide on the dangerosity of a script, we believe it is still important to acknowledge its existence within a script, and so is for other kinds of obfuscating transformations more likely to occur in malicious scripts or in benign scripts. The Dean Edwards' packer is easily recognizable by its signature:

```
eval(function(p,a,c,k,e,d){
  e=function(c){
    return(
```

However, what if someone tweak the code to obtain a different signature than `p,a,c,k,e,d`. Would the current detection systems still acknowledge its presence? On the other hand, this packer performs some operations that are well-known to security analysts and the part of the structure of the abstract syntax tree (AST), resulting from parsing the packer, can be captured. In particular, the previous signature can be abstracted to the following (extended signature in prefix notation:

```
CALL eval CALL FUNC ASSIGN ID FUNC RET
```

Such representation allows us to capture different layers of nesting as well. In this paper, we explain the different algorithms and steps taken to build an automaton able to capture several of such structures at a same time.

# 2 Related Works

The issue of obfuscation is rarely raised and was almost considered an artifact of malicious contents. As a testament of such way of thinking, we can point out two notable papers on the seldom dealt-with issues of obfuscation in JavaScript code.

In [3], the authors have observed that malicious web pages include obfuscated code used to circumvent signature-based detection systems. The authors have designed three metrics (byte occurrence, entropy, word size) to distinguish obfuscated strings from deemed benign unobfuscated ones. However, they recognize that obfuscated JavaScript is not itself a good indicator of malice. They only identified 6 different obfuscation patterns. Oddly enough, their method was not able to detect an `eval` unfolding string as obfuscated.

In [4], the authors clearly assumed that obfuscated strings are malicious in most of the cases. They proposed a machine-learning-based classification technique. With an overall number of 65 features, including 50 JS keywords and symbols, they came up to the conclusion that human-readable features perform better and considered reinforcing these in priority.

Contrary to these related works, we are not considering the obfuscated string itself, but rather the

obfuscated string as an output of an obfuscating transformation or a combination of obfuscating transformations, often implemented as automated tools. This is not a method for deobfuscation though.

# 3 Proposal

We propose to circumvent the issue of (tokenized) string matching by capturing the structure of the program through its parse tree. Since we wish to know about characteristic structures of obfuscating transformations, we propose to mine frequently occurring subtrees in a treebank. Eventually, we will be able to gain knowledge of the structures of obfuscating transformations and then use this knowledge to identify such structures in future datasets.

## 3.1 AST Representation

Since we are concerned with reducing the entropy of script contents for the purpose of analysis, it became necessary to abstract the script code in order to get rid of the randomization introduced in the identifiers and values. An accurate and abstract representation of a program is the abstract syntax tree. However, the AST used here is different in some aspects:

- values are not kept and replaced by generic types: *NUM* for numeric values, *STR* for string values, *ID* for identifiers;
- some identifiers for core objects and functions are preserved in order to make explicit operations such as overriding or aliasing of core components;
- conditions of branching and looping are not preserved and the whole construct is replaced by a couple $\langle S, \mathcal{I} \rangle$ where $S$ represents a symbol (either *BRANCH* or *LOOP*) and $\mathcal{I}$ the set of instructions that form the body of the branch or loop;
- all instructions in a block are represented at the same level by sibling nodes, children of a node representing the containing block (which can be a branching control, a loop, a function definition, etc.).

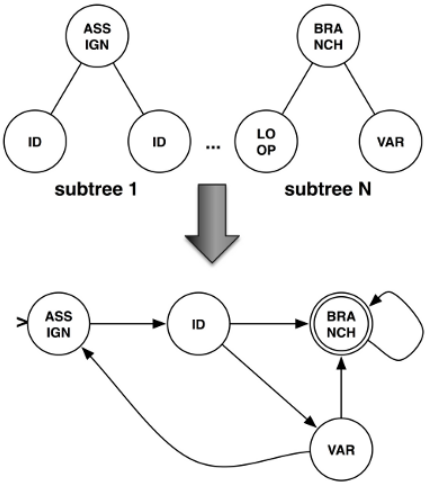


図 1: An automaton accepting all the subtrees is built.

Moreover, we are not tokenizing the AST representation as it has been done for pattern matching purpose in [5] but we focus on the properties of tree-like structures.

## 3.2 Subtree Matching by PDA

After discovering frequent recurring subtrees appearing in a learning dataset (or treebank), we will build transition rules from the learned subtrees (Fig. 1. The objective is to quickly identify the learned subtrees in the script being analyzed. The analyzed script, transformed into an AST, is fed to a pushdown automata, which is analogous to a finite state machine with a stack. This technique has been proposed by Flouri et al.[6] as an output to a new discipline of research dubbed *arbology*[10], which aims to apply or adapt algorithms on strings and sequences (also known as *stringology*) to the field of tree structures. Stringology made use of finite state machine as the model of computation and this naturally yielded to the use of pushdown automata towards trees, since the addition of the stack accomodates recursion.

Constructing a deterministic pushdown automata

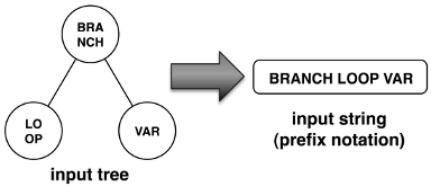


図 2: Each JS file is transformed to an AST and then to its prefix notation.

accomodating a set of subtrees is done in three distinct steps:

1. construction of a PDA accepting a set of subtrees in their prefix notation
2. construction of a nondeterministic subtree matching PDA for a set of subtrees in their prefix notation
3. transformation of the nondeterministic subtree matching PDA to an equivalent deterministic PDA

Each step employs well-know PDA construction algorithms which are further detailed in [6]. In this research, we implemented a script that parses an XML file listing frequently occurring subtrees and generates the transition rules for the deterministic subtree matching PDA. We also modified a program emulating a finite state machine to accomadate a stack. This program would parse the file containing the transition rules, as well as a file containing the script to be analyzed. The script to be analyzed is transformed into an AST and then expressed as a prefixed string (see Fig. 2) which is passed as input string to the subtree matching PDA. Whenever, there is a match, the matched subtree is reported allowing to identify characteristic subtree as the script's AST is browsed.

# 4 Experiment

In this section, we briefly expose the settings of our experiment and explain its implementation. The results are displayed in a straightforward manner and commented thereafter.

## 4.1 Data Preprocessing

MWS2011 D3M datasets[7] are composed of 3 pcap files spanning the period of 3 days during February 2011. Each file features several HTTP conversations. An HTTP conversation takes place between the user agent (the browser) and one or several web pages (belonging to the same or distinct origins) during the span of a same browsing session. By definition, an HTTP conversation is constituted of several HTTP transactions (request and response).

Once we have been able to disambiguate the conversations, we apply a script to reconstitute JS files scattered among several origins or pages. The script takes each HTTP transaction individually, extract the JS contents and follow the linked contents through consecutive transactions. This allows us to recover fragmented scripts as a single file.

By such processing, we were able to recover over 60 files for Day 1, 82 for Day 2 and 116 for Day 3. As a mean of comparison, we also included a randomly handpicked dataset composed of transactions towards 25 domains of Alexa top 100. This last datasets is comprised of around 150 files.

## 4.2 Overview

The experiment is roughly divided between a learning and a testing phase. The learning dataset will provide subtrees to be matched on later datasets during the testing phase. Day 1 has been designed as the testing dataset. Due to time constraints, we settled for picking subtrees manually instead of an automated way. More details about such choice are provided in Section **??**. We selected 35 subtrees from Day 1 ASTs ranging from 3 to 43 nodes, representing 1 to 3 instructions and 1 to 3 layers of nesting. Obviously, the longer the more specific.

These 35 subtrees were stored as ASTs in an XML file. The file was processed using the algorithms referred in Section 3.2 to generate states and transitions for our deterministic subtree matching pushdown automaton. The resulting automa-

ton comprises 379 states (among which only 39 are accepting) and 17057 transitions.

Concerning the other datasets (Day 2, Day 3 and Alexa25), we generated a prefix AST notation for each extracted JS file. These files were then inputted to the subtree matching PDA. Results include the number of subtrees identified (which is nearly equivalent to the number of a time an accepting state is reached, modulo possible false positives due to the entanglement of the PDA transitions), the matched subtrees, the time it took to process the JS file (in seconds).

### 4.3 Results

Among the 82 scripts, of the Day 2 dataset, passed to the PDA, 56 were found to contain one or several occurrences of a subtree picked in the Day 1 dataset which represents around 68% of the scripts. For the data of Day 3, 92 scripts of the 116 extracted were found to contain subtrees identified by the PDA, which is almost 80%. On the other hand, while confronting the data from the Alexa25 scripts, which contain a mixture of unobfuscated and obfuscated benign scripts, the ratio falls down to 50%. While this sounds to be still high, it is to be noted that most of the accepted subtrees are actually of two kinds, which revealed to be the shortest subtrees, thus more likely to occur. Additionally, the distribution is heavily polarized between these two subtrees for the Alexa25 dataset, while Day 2 and Day 3 datasets exhibit a more varied distribution of subtrees.

We were able to cluster JS samples around different type of subtree combinations: single subtree, combination of subtree from the same family, combination of subtrees from different families. A family is a set of subtrees extracted from the same sample at learning stage and therefore indicating with a high level of confidence that the JS sample contains a given obfuscating transformation when both subtrees are found together. This rules out cluster #22 from being a good cluster since its samples only contain a partial family (besides, the subtree is only 3 nodes long which explains its commonness). On the other hand, clusters #01, #03 are characteristics of the Dean Edwards' packer since it contains all or most of the subtrees for family A (Dean Edwards' packer). Cluster #12 is another such good example of a identified obfuscations.

In terms of performance, the PDA is quite an efficient technique. As a matter of fact, processing times for scripts of Day 2 range from 0.098803s to 0.279856s and an average of 0.10638s. For Day 3 dataset, minimum time is 0.098565s, maximum time is 0.151935 and average time is 0.103685s. For an average processing time barely excessing 100ms, one must say that the PDA is quite efficient. Not to mention that these results were obtained using scripts written in Ruby, which is not praised for its time performance. On the other hand, the PDA was still tractable, and it may be possible that it grows to thousands of states and transitions in worst-case scenarios.

## 5 Discussion

The results have shown that we need to take into consideration some subtleties inherent to automata in order to avoid false positives.

We thought to apply data mining methods to past datasets in order to discover the most frequent subtrees. In fact, we started experiencing with a tool called *Varro*[8] which identifies and counts regularities in treebanks. *Varro* usually processes treebanks representing syntactically-annotated natural language sentences and extract frequent induced[9] unordered subtrees. Our method, using prefix notation of ASTs, is limited to bottom-up subtrees[9], which offers less flexibility.

*Varro* minimizes memory use but the worst case memory performance is $O(nm)$ where $n$ is the number of vertices in the treebank and $m$ is the largest frequent subtree found in it. This has been the major drawback in our attempt to automate the discovery of frequently occurring subtrees in the MWS 2011 datasets as some scripts can span several thousand nodes.

表 1: Clusters obtained from PDA identification (partial)

| | Learned subtrees (35 subtrees divided in 17 families) | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | |
| # subtrees | 5 | 4 | 1 | 1 | 4 | 2 | 1 | 1 | 1 | 2 | 3 | 3 | 1 | 2 | 1 | 1 | 2 | |
| avg. nodes | 13 | 9 | 6 | 10 | 7 | 20 | 12 | 12 | 22 | 13 | 7 | 22 | 44 | 11 | 10 | 6 | 11 | |
| ID | Day 2 (56 samples matched) | | | | | | | | | | | | | | | | | # |
| 01 | 4 | | | 1 | 1 | | | | | | | | | | | | | 3 |
| 02 | | 1 | | | | | | | | | | | | | | | | 6 |
| 03 | 5 | | | | 1 | | | | | | | | | | | | | 2 |
| 04 | | | | | | | | | | | | | 3 | | | | | 11 |
| 05 | 3 | | | | 1 | | | | | | | | | | | | | 2 |
| 06 | | | | | | 2 | | | | | | | | | | | | 4 |
| 07 | | | | | | | | | | | | | | | | | 2 | 2 |
| 08 | | | | 1 | | | | | | | 1 | | | | | | 1 | 2 |
| 09 | | | | | 1 | | | | | | | | | | | | | 4 |
| 12 | | | | | 4 | | | | | | | | | | | | | 2 |
| 13 | | | | | | | | | | | | | 1 | | | | | 2 |
| 15 | | | | | | | | | 1 | | | | | | | | | 3 |
| 16 | | | | | 1 | | | | | | | | | | | | 1 | 5 |
| ID | Alexa Top 100 Random 25 (75 samples matched) | | | | | | | | | | | | | | | | | # |
| 22 | | | | | 1 | | | | | | | | | | | | | 51 |
| 23 | | | | | 1 | | | | | | | | | | | | 1 | 11 |
| 25 | | | | | | | | | | | | | | | | | 1 | 8 |
| 27 | | | | | 2 | | | | | | | | | | | | | 3 |

# 6 Conclusion

In this paper, we have proposed a way to reduce the entropy induced by string randomization by focusing on the tree-like structures of programs that commonly occur during parsing time. Using an abstract syntax tree representation, we can express characteristic structures found in obfuscating transformations. We were also able to cluster samples around obfuscating transformations (expressed as a set (or family) of subtrees) and showed that clusters issued from benign samples are likely to be false positives, since the size of the subtree is too small to be characteristic by itself.

We are well aware that this is just a step towards classification of obfuscated samples based on the obfuscating transformations they employ. Indeed, we need to characterize more each subtree by, for example, assigning a grade based on their occurrence and length. Larger subtrees will then have more weight than short subtrees but short subtrees that occur consecutively a certain number times may indicate obfuscation. Then, we can combine several subtrees to represent an obfuscating transformation (for example, in this research, up to 4 subtrees were used to represent different part of the Dean Edwards' packer) and then highly reducing the number of false positives.

**Acknowledgements.** This research is supported by a grant from the NEC C&C Foundation.

# 参考文献

[1] C. Craioveanu, *Server-side script polumorphism: Techniques of analysis and defense*, Proc. $3^{rd}$ Intl. Conf. on Malicious and Unwanted Soft. (2008).

[2] C. Collberg et al., *A Taxonomy of Obfuscating Transformations*, Tech. Rep. #148, The University of Auckland, July 1997.

[3] Y.H. Choi et al., *Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis*, Proc. $1^{st}$ Intl. Mega Conf. on Fut. Gen. Inf. Tech. (2009).

[4] P. Likarish et al., *Obfuscated Malicious JavaScript Detection using Classification Techniques*, Proc. $4^{th}$ Intl. Conf. on Malicious and Unwanted Soft. (2009).

[5] C. Curtsinger et al., *ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection*, Proc. $20^{th}$ USENIX Sec. Symp. (2011).

[6] T. Flouri et al., *Aho-Corasick like multiple subtree matching by deterministic pushdown automata*, Proc. $20^{th}$ USENIX Sec. Symp. (2010).

[7] M. Hatada et al., *Datasets for Anti-Malware Research - MWS 2011 Datasets*, マルウェア対策研究人材育成ワークショップ, October 2011 (to appear).

[8] S. Martens, *Varro: An Algorithm and Toolkit for Regular Structure Discovery in Treebanks*, Proc. $23^{rd}$ Intl. Conf. on Comp. Ling. (2010).

[9] Y. Chi et al., *Frequent Subtree Mining - An Overview*, Fund. Inf., vol.66, no.1-2, pp.161–198, November 2004.

[10] *Arbology*, `www.arbology.org`