2012 26th International Conference on Advanced Information Networking and Applications Workshops

Characterizing Obfuscated JavaScript using Abstract Syntax Trees: Experimenting with Malicious Scripts

Gregory Blanc
Graduate School of Information Science
Nara Institute of Science and Technology
Ikoma, Japan
Email: gregory@is.aist-nara.ac.jp

Daisuke Miyamoto
Information Technology Center
The University of Tokyo
Tokyo, Japan
Email: daisu-mi@nc.u-tokyo.ac.jp

Mitsuaki Akiyama
NTT Information Sharing Platform Laboratories
Musashino, Japan
Email: akiyama.mitsuaki@lab.ntt.co.jp

Youki Kadobayashi
Graduate School of Information Science
Nara Institute of Science and Technology
Ikoma, Japan
Email: youki-k@is.aist-nara.ac.jp

***Abstract*—Obfuscation, code transformations that make the code unintelligible, is still an issue for web malware analysts and is still a weapon of choice for attackers. Worse, some researchers have arbitrarily decided to consider obfuscated contents as malicious although it has been proven wrong.**

Yet, we can assume than some web attack kits only feature a fraction of existing obfuscating transformations which may make it easy to detect malicious scripting contents. However, because of the undecidability on obfuscated contents, we propose to survey, classify and design deobfuscation methods for each obfuscating transformation.

In this paper, we apply abstract syntax tree (AST) based methods to characterize obfuscating transformations found in malicious JavaScript samples. We are able to classify similar obfuscated codes based on AST fingerprints regardless of the original attack code. We are also able to quickly detect these obfuscating transformations by matching these in an analyzed sample's AST using a pushdown automaton (PDA). The PDA accepts a set of subtrees representing obfuscating transformations previously learned. Such quick and lightweight subtree matching algorithm has the potential to detect obfuscated pieces of code in a script, to be later extracted for deobfuscation.

***Keywords*-abstract syntax tree; JavaScript; obfuscation**

I. Introduction

JavaScript is the glue of the AJAX framework which is a widespread framework for many Web 2.0 applications. Its prevalence has benefited from the presence of built-in interpreters in the major web browsers, allowing developers to provide dynamic and interactive contents without the need for users to install third-party plug-ins. JavaScript is mostly used on the client-side where it manipulates the Document Object Model (DOM), among other tasks. This is precisely on the client-side that attackers are able to hijack the control-flow of an application to their own benefit. This is often done by means of developing scripts to be injected in benign web pages through an SQL injection or an XSS vulnerability. In order to prevent the execution of unsafe scripting contents, pattern matching is often applied to detect attack vectors. However, attackers have been able to evade such detectors by obfuscating their scripts.

If obfuscation has been extensively used by attackers, it is not restricted to a malicious usage. Developers have also made great use of obfuscators in order to protect their intellectual property or compress their voluminous scripting contents. Therefore, it would be counterproductive to consider every obfuscated script as malicious. From this observation, we sought to survey obfuscation patterns themselves to classify obfuscation techniques, and gain knowledge on which techniques are likely to be used by attackers. Still, attackers may shift towards techniques exclusively used by web developers, further clouding the issue. It would then be impossible to decide on the nature of an obfuscated script without prior deobfuscation, which is often hindered by anti-analysis tricks that render dynamic techniques ineffective. Ultimately, we seek to enumerate all obfuscation techniques, assuming there is a finite set of these, which will allow us to design an execution-less deobfuscation technique for each obfuscating transformation. That way, deobfuscation can be automated and used on a large scale.

This paper offers a first look at the variety of obfuscating transformations collected during a short-period crawling of malicious websites. Since string matching cannot be applied to identify obfuscating transformations, abstracting scripting contents becomes necessary. Abstract syntax trees are a straightforward transformation of programs and allow to reduce the randomization introduced by variable renaming while highlighting structural or hierarchical features of programs. Here, we apply techniques developed in adjacent areas of expertise to extract, learn and detect these features in JavaScript programs.

This paper is organized as follows: first, we set back this

978-0-7695-4652-0/12 $26.00 © 2012 IEEE
DOI 10.1109/WAINA.2012.140

project in its context in Section II, then in Section III, we cover past research works on the subject of obfuscation as well as related to the techniques we employ in our proposal, which is developed in Section IV. As we indicated, we evaluated our proposal against malicious JS scripts collected on a short period; the experiment is detailed in Section V. We discuss the limitations of our proposal in Section VI and conclude in Section VII.

II. Context

As stated in the introduction, identifying obfuscated pieces of code will allow to precisely extract code that needs to be deobfuscated. In fact, this project forms part of a wider research work that aims to provide a mostly static analysis of web scripting malware.

Most malicious script detectors fail to consider the issue of obfuscation, that is any transformation that alters the code to evade signature matching. In this paper, we consider web script obfuscation, which is different from binary obfuscation. More information is available in Section III-C.

Often, obfuscation is considered an indicator of malice: detecting malice amounts to detecting obfuscation [1], [2]. On the contrary, some other approaches have ruled out obfuscation as an issue, considering that obfuscation is a terminating process that can be trivially addressed through execution [3]. The latter approach actually carries out analysis on deobfuscated contents.

Provos et al. [4] have long proven that obfuscation does not indicate malice which invalidate approaches that rely solely on obfuscation detection. Moreover, a recent report from Google [5] demonstrates that although terminating, obfuscation is not necessarily a convergent process. By using cloaking techniques to detect analysis environments, malicious obfuscated contents may be deobfuscated into benign code when analysis is suspected. Finally we argue that executing malicious code is not recommended and therefore isolating suspicious code seems a safe approach.

Our research work provides static analysis of code that has been priorly deobfuscated using rewriting logic [6]. To automate deobfuscation, it is necessary to precisely extract suspicious code to be rewritten. This requires to reliably detect obfuscated contents and any routine in the code that may be of use in deobfuscating these contents. Obviously, signature matching is not an option and we have surveyed alternative ways in this projet as detailed in next section.

III. Related Works

Related works in the area of JavaScript analysis have seen newly-proposed methods involving the use of abstract syntax trees (ASTs). Authors of Zozzle [3] claim to be the first to have used hierarchical features extracted from ASTs but notice that their method suffers from false positives and should rather be used as a first stage for their other tool, Nozzle [7]. Moreover, Zozzle needs deobfuscated JavaScript to perform acurately. On the contrary, our work rather concentrates on obfuscation and how to identify obfuscating transformations using code similarity detection methods.

A. Plagiarism Detection

Methods for finding similar pieces of code among several programs are used for two distinct purposes. One is clone detection. It aims to detect and remove duplicated pieces of code in order to reduce software maintenance costs. The other one is plagiarism detection. It is often used in the case a company wants to check if someone is stealing their code. It has also been used in cases where a company wants to avoid plagiarizing some open source projects when developing their own products.

The string-based method is the basic type of plagiarism detection method. Each statement is treated as a string, and a program is represented as a sequence of strings. Two programs are compared to find sequences of same strings [8]. However, it can be evaded by renaming identifiers.

In the token-sequence based method [9], source codes are tokenized to provide sequences of tokens that are supplied to string matching algorithms. The tokenized representation is generally abstracted, so the identifiers cannot be distinguished. Due to this, the method is robust to identifier renaming.

Another approach is the Abstract Syntax Tree (AST) based method. A program is parsed into an AST with variable names and literal values discarded. The method attempts finding duplicate subtrees to detect plagiarism.

Aside from these methods, a Program Dependency Graph(PDG)-based method has also been developed [10], [11]. A PDG of a program is a graph that has nodes assigned to each statement of the program. Directed edges represent a dependence. The PDG-based method can detect complex disguises. Unfortunately, finding similarity between subgraphs is very costly: it is a NP complete problem [12].

B. AST Fingerprinting

Abstract Syntax Tree Fingerprinting (ASTF) was proposed by Chilowicz et al. [12]. An ASTF is composed of four elements: namely, weight, hash value, and pointers to the subtree root node and to the parent node. Given a subtree t, the weight ($w(t)$) is the size of the subtree. The pointers are the abstract syntax representations at the root of subtree t, $r(t)$ and its parent, $p(t)$.

If a subtree t has n child subtrees, $t_1, \cdots t_n$, a hash value of subtree t, ($\mathcal{H}(t)$), is $\mathcal{H}(r(t) \cdot \mathcal{H}(t_1) \cdots \mathcal{H}(t_n))$ where $\mathcal{H}$ is a cryptographic hash function, such as SHA-1 [13] or MD5 [14]. Note that the hash variables are calculated in bottom-up manner, i.e., leaves are calculated first. Once computed, the abstract syntax tree is stored in a repository for further analysis.

C. Obfuscation

For readers unfamiliar with obfuscation, we would like to point out interesting surveys by Collberg et al. [15] and Craioveanu [16]. The former survey is a taxonomy of obfuscating transformations in Java bytecode, it classifies these transformations following criteria as diverse as the kind of protection, the quality of the obfuscation, or the information targeted by the transformation (data, layout, control flow). The latter concentrates on JavaScript obfuscation and provides an overview of current techniques.

The issue of obfuscation is rarely raised and was almost considered an artifact of malicious contents until Provos et al. [4] ruled it out. Nonetheless, such way of thinking has partly remained in the community. We can point out two notable paper on the seldom dealt-with issue of obfuscation in JavaScript code.

Techniques to detect obfuscation in web scripts[1], [2] have been proposed recently. The goal is usually to detect malware assuming obfuscation is an indicator of malice. Both proposals make use of machine learning methods whether on statistical string features (byte occurence, entropy, word size) or semantic features (JS keywords and symbols).

Contrary to these related works, we are not considering the obfuscated string itself, but rather the obfuscating transformation or combination of obfuscating transformations, often implemented as automated tools. This proposal is not a method for deobfuscation though, but seeks to exhibit significant features of obfuscating transformations in order to automate their detection and deobfuscation.

IV. Proposal

The contribution of this paper lies in applying AST-based methods to demonstrate significant regularities in obfuscated JavaScript programs. Our proposal borrows principles from source code analysis, in particular pattern matching. But contrary to many previous proposals, our matching unit is the subtree. Being able to quickly identify subtree patterns of known obfuscating transformations can allow an analyst to detect and extract only obfuscated contents. To be more precise, we are identifying decoding routines, regardless of the obfuscated string. Such analysis can lead to the automation of deobfuscation for each given obfuscating transformation class as described in [6].

A. Overview

In our approach, we consider the abstract syntax trees of scripts to analyze. We distinguish two distinct phases: first, we attempt to learn obfuscating transformations as subtrees in obfuscated scripts' ASTs; then, we try to identify the presence of such obfuscating transformations by matching learned patterns in candidate scripts' ASTs. The expression of a script as an AST is common to both phases and is obtained by simply parsing the script. Our subtree matching prototype developed in Ruby takes advantage of the *johnson* library [17] which supports the SpiderMonkey [18] JavaScript engine.

B. Abstract Syntax Tree

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 its abstract syntax tree. However, the AST used here is different in some aspects from similar approaches [3]:

- values are discarded and replaced by generic types: *NUM* for numeric values, *STR* for string values, *ID* for identifiers;
- some identifiers for core objects, e.g., `document`, and functions, e.g., `String.fromCharCode()`, are preserved in order to make explicit operations such as overriding or aliasing of core objects;
- conditions of branching and looping are discarded 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 child nodes, which are 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.).

Moreover, we are not tokenizing the parse tree representation as it has been done for pattern matching purpose in [3] but we rather focus on the hierarchical properties of tree-like structures as in [3].

Figure 1 displays a sample AST for the following eval unfolding loop:

```
for(i=0;i<str.length;i++){
  str2 = str2 +
  String.fromCharCode(str.charCodeAt(i)^1);
  str3 = str3 + str3};
eval (str2);
```

The LOOP statement contains two instructions represented by two paths stemming from the LOOP node: one being the actual decoder using the `String.fromCharCode` function and the other path being a dummy operation on an unrelated variable.

C. AST Fingerprinting

In a first step, we generate AST fingerprints (ASTFs) for each JS file present in our learning dataset. Each JS file is parsed and the corresponding AST is generated by applying the abstraction described in the previous section. The obtained AST is then processed by a Perl script that computes its fingerprint as explained in Section III-B. The

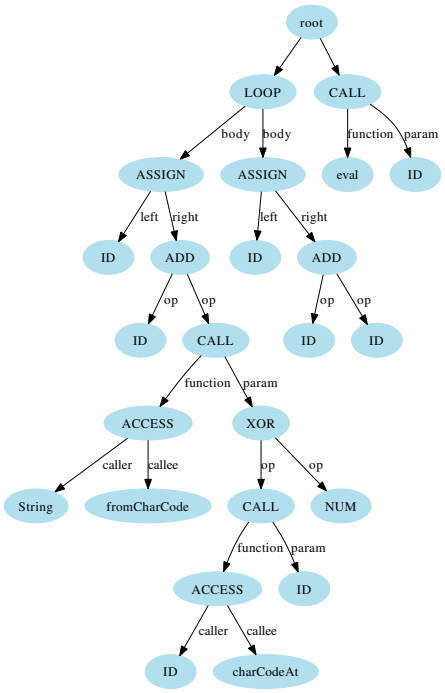

Figure 1. Abstract syntax tree of an eval unfolding transformation.

output of such processing provides a fingerprint for each subtree composing the script's AST. We assume that recurring fingerprints among samples denote recurring obfuscating transformations, provided our dataset is composed of obfuscated JavaScript files only. Fingerprints vary for slight variations of the code, this ensures that only invariant sections of obfuscating transformations are learned. However, some false negatives can occur.

Then, we manually pick representative subtrees. To elect these, we cluster samples on unique occurrences of ASTFs. We further reduce the number of clusters by merging clusters using similarity concordance rates computed between each pair of samples. These clusters usually offer a least common denominator, which is the set of commonly shared ASTFs from which we pick our subtrees to be matched in the next phase.

D. Subtree Matching Pushdown Automaton

A pushdown automaton (PDA) accepting the selected subtrees is then built. The analyzed script, transformed into an AST, is fed to the PDA in order to match occurrences of the learned subtrees. This technique has been proposed by Flouri et al. [19], who aim to apply or adapt algorithms, commonly applied to strings and sequences, to the field of tree structures. Strings are usually processed using finite state machines as the model of computation and this naturally led to the use of pushdown automata towards trees, since the addition of a stack accommodates recursion. Thanks to syntactic rules that limit the number of combination between symbols in a programming language, employing a subtree automaton seems an effective solution that can scale with a large number of subtrees. Still, the size of these subtrees can badly affect performance and should be limited.

Constructing a deterministic pushdown automata 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-known PDA construction algorithms which are further detailed in [19]. In this research, we implemented a script that parses an XML file listing recurring subtrees and generates the transition rules for the deterministic subtree matching PDA. We also modified a program emulating a finite state machine to accommodate 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 its prefix notation to be fed to the subtree matching PDA. Whenever, there is a match, the matched subtree is reported, allowing to identify characteristic subtrees while traversing the script's AST.

V. Experiment

The experiment evaluates jointly ASTF and subtree matching methods in characterizing a dataset of obfuscated JavaScript programs, in particular ones collected from malicious websites. Using ASTF, we are able to discover recurring subtrees that are assumed to be characteristic of obfuscating transformations. To that extent, human knowledge is also involved. We also evaluate the accuracy and performance of subtree matching using pushdown automata on a set of subtrees relatively large compared to the dataset of ASTs it is extracted from.

A. Dataset Description

We used URLs listed by MalwareDomainList.com (MDL) [20] which publishes a large URL blacklist related to malware infected web pages. We collected HTTP sessions on different domains on February 8th, 14th and 16th, 2011. The data collection procedure is as follows. A client honeypot crawls all the URLs on the list and detects malicious ones. The honeypot's environment runs Internet Explorer 6.0 on a Windows XP SP2 platform [21]. These web browser and OS versions include exploitable vulnerabilities, most of which can be attacked by many exploit packs, so they are suitable as the basis of a honeypot system. Vulnerable versions of Adobe Reader, Flash Player, JRE, WinZip, and QuickTime plug-ins were also installed on the system. During the anti-Malware engineering WorkShop (MWS) [22], this dataset is

shared by academic and industrial researchers for research promotion.

Additionally, we added another dataset comprised of JS files from randomly chosen 25 domains among Alexa Top 100 [23]. This dataset serves mostly the role of control set.

B. Evaluation

In this paper, we evaluate the precision of our methods by attempting to identify, in a dataset of obfuscated scripts, obfuscating transformations learned in a previous dataset. As suggested here, we apply the two techniques presented in Section IV at two distinct stages. Recurrent ASTFs indicate recurring subtrees in the learning set, which we assume to be patterns of obfuscation. Obviously, having *clean* datasets of obfuscated samples will enhance accuracy. Assuming the set of obfuscating transformations, as well as patterns, is finite, we wish to constitute an exhaustive knowledge base.

Using AST fingerprints, we calculated the concordance rate between each 2 files of the learning dataset (Day 1 (February 8th) of the MWS dataset). From these concordance rates, we were able to regroup samples, including duplicates, in 16 clusters. From each cluster, we manually picked one or several subtrees based on the recurrence of its corresponding ASTF and some additional criteria:

- we excluded subtrees rooted at the root of an AST, as it is not a subtree;
- we excluded subtrees of which ASTF weight, i.e., the size of the subtree, is less than 7;
- we excluded subtrees of which ASTF weight is more than 150, in order to reduce the number of transitions for the automata to be built;
- we tried to pick one subtree for a given set when the set was made up of non-duplicates and the subtree was the most recurring one among samples;
- we tried to pick several subtrees when no subtree were outstandingly recurring in a set. 2 or 3 subtrees would be taken based on their location in the AST or on their length. In particular, we tried, as much as possible, to select subtrees that were not too big, in order to reduce the number of false negatives, and not too small in order to avoid false positives.

In Table I, we expressed the learned subtrees on two lines. *Subtree ID* indicates an individual subtree while *obfuscation ID* (from A to P) indicates a particular obfuscating transformation. This corresponds to one of the 16 clusters since each cluster regroups files that displays similar subtrees, thus similar obfuscating transformations.

Overall, we extracted 32 subtrees from the 16 clusters of the learning set. These 32 subtrees were as small as 7 nodes and as big as 127 nodes. Generating the transitions for the pushdown automata yielded 37730 transitions.

During the second stage, we generated prefix notations for JS samples from the testing datasets (Day 2 and 3 from MWS datasets). We further grouped matched samples by detected subtrees. We call *obfuscation pattern*, or simply *pattern*, the combination of 1 or several obfuscating transformations. In Table I, patterns are expressed as a combination of 1 or several subtrees from 1 or several obfuscating transformations. We identified 19 different obfuscation patterns (pattern ID 1 to 19). 55 samples out of 82 in Day 2 (February 14th) were found to contain subtrees we learned, encompassing 15 different obfuscation patterns. The Day 3 dataset had 100 samples out of 116 that contained around 15 different obfuscation patterns. As a way to control the accuracy of the identification, we also tested the Alexa dataset that contained both unobfuscated and obfuscated benign samples and it resulted that 19 samples were detected as obfuscated out of 148. And among these 19 matched samples, 18 displayed the same pattern. The results are displayed in Table I where each testing dataset is shown independently. For each pattern we discovered in the testing phase, we counted the number of matched samples. In particular, we can observe that some patterns are exclusive to one day or the other (patterns 6, 11 and 12 for Day 2; patterns 16, 17 and 18 for Day 3). Obviously, the domains visited are different but this can also indicates that new tools were used to generate these obfuscated scripts.

Overall, identifying trees has been straightforward and accurate as shown by results on Day 2 and Alexa 25 datasets. Day 2 comprised JS samples quite different from the learning dataset which was much more similar to Day 3 dataset. Samples matched in Alexa 25 dataset express the fact that the obfuscating transformation L may not be relevant (7 nodes, the smallest subtree of the learning set) while the pattern ID 19 only relies on a single subtree (partial matching of transformation E). In fact, much more credit should be given to complete match of obfuscating transformations involving every subtree for a given ID.

On the other hand, pattern IDs also give some insights on the accuracy of our manual feature selection. Partial matching of obfuscating transformation, where one subtree is left out, may indicate that this particular subtree may not be relevant to the corresponding obfuscation ID. Some subtrees were also never matched which indicate that they are not indicators of obfuscation or that they just did not occur in testing datasets. On the contrary, subtrees occurring in the control dataset might either indicate that a learned obfuscating transformation has been detected or that the subtree is actually not an indicator of obfuscation.

C. Performance

Considering the high number of transitions, it was expected that the process may take time and it is a challenge to design the smallest and most expressive set of subtrees able to characterize obfuscating transformations of a given dataset. We ran our automata on top of a Mac OS X platform with a Dual-Core Intel Xeon (2 x 2.66GHz) with 8GB of DDR2 RAM. The script is written in Python and is far from

subtree ID	02QPBJU8_0001-0/0	2QBFJLZQ_0001-0/0	2QBFJLZQ_0001-0/1	2QBFJLZQ_0001-0/2	5LYZW3J2_0001-0/0	5LYZW3J2_0001-0/1	8OOEADDL_0001-0/0	9EZ2MI1N_0001-0/0	9EZ2MI1N_0001-0/1	9EZ2MI1N_0001-0/2	C3KZZ9ZF_0001-0/0	C3KZZ9ZF_0001-0/1	C57RJU4Y_0001-0/0	C57RJU4Y_0001-0/1	CHPMQKKU_0003-0/0	CHPMQKKU_0003-0/1	GKFQY90R_0001-0/0	GKR3AJ4H_0001-0/0	GKR3AJ4H_0001-0/1	LI9RG7YR_0004-0/0	LI9RG7YR_0004-0/1	LI9RG7YR_0004-0/2	OHWM3F4A_0001-0/0	Q6AML43K_0002-0/0	Q6AML43K_0002-0/1	VVID7HEC_0001-0/0	VVID7HEC_0001-0/1	Y3ZC6N6X_0029-0/0	Y3ZC6N6X_0029-0/1	Y3ZC6N6X_0029-0/2	Y9CODFTS_0001-0/0	Y9CODFTS_0001-0/1	samples matched
obfuscation ID	A	B			C		D	E			F		G		H		I	J		K			L	M		N		O			P		
pattern ID	Day 2 (82 samples)																																55
1	X																																8
2																		X	X					X	X								7
3																																	11
4																									X								4
5																												X	X	X			2
6										X																							1
7											X	X																					1
8					X	X																											2
9																				X	X	X											2
10		X		X																					X								1
11													X	X									X										2
12													X	X																			1
13																							X										11
14																	X																1
15																										X	X						1
pattern ID	Day 3 (116 samples)																																100
1	X																																9
2																								X	X								9
3																		X	X														11
4																									X								8
5																												X	X	X			1
7											X	X																					1
8					X	X																											2
9																				X	X	X											2
10		X		X																					X								1
13																							X										6
14																	X																1
15																										X	X						1
16															X	X																	46
17									X	X																							1
18																															X	X	1
pattern ID	Alexa 25 (148 samples)																																19
13																							X										18
19									X																								1

Table I
MATCHED SAMPLES CLUSTERED BY OBFUSCATING TRANSFORMATIONS

being optimized. Nonetheless, the script processed the 82 samples from Day 2 with an average time of 243ms, the 116 of Day 3 in 244ms in average and the Alexa dataset containing 148 samples in 38.44 seconds which represents an average processing time of nearly 260ms. As expected, the Alexa dataset, containing much bigger JavaScript files, took a greater time to be processed by the PDA but still remains in an acceptable range. Nonetheless, the PDA can scale with a large number of subtrees to be matched provided we limit their size.

VI. DISCUSSION

One problem inherent to obfuscated programs is the presence of dummy instructions, i.e., instructions that do not affect the true flow of execution but rather are here to misdirect an analyst. Their impact on the AST representation is obviously high on the distance between two programmatically equivalent ASTs. Moreover, it affects both vertical (insertion of intermediate nodes) and horizontal (insertion of sibling nodes or subtrees) distances. One proposal we can make to tackle vertical insertion of dummy nodes relies on the principles of automata theory. Our PDA is governed by transition rules generated from learned subtrees. If we upgrade each state's transitions with a transition function that accommodates unexpected input symbols to remain on the same state, as would do an exception clause in a regular expression, we can achieve a more flexible subtree matching. Nonetheless, this transition function can not accept infinite input words and we would need to define a limit.

On the other hand, although we can imagine a similar solution for horizontal insertion, it may not scale with the limit on the number of excepted input symbols. Using induced subtrees instead of bottom-up subtrees can reduce the impact of horizontal insertion without relying on wildcards. Indeed, our approach lacks flexibility in that it makes use of bottom-up subtrees [24], i.e., all nodes composing a

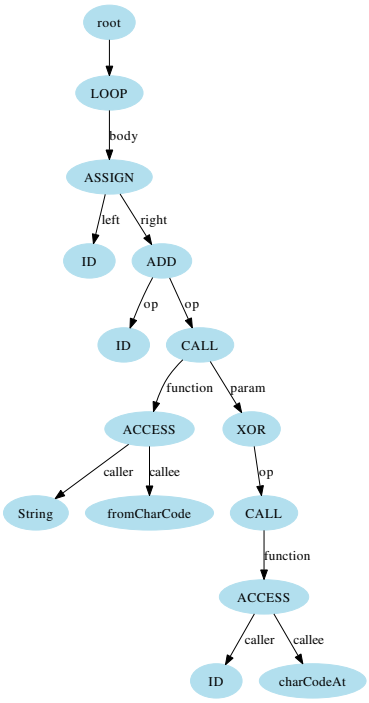

Figure 2. Bottom-up subtree of the example eval unfolding.

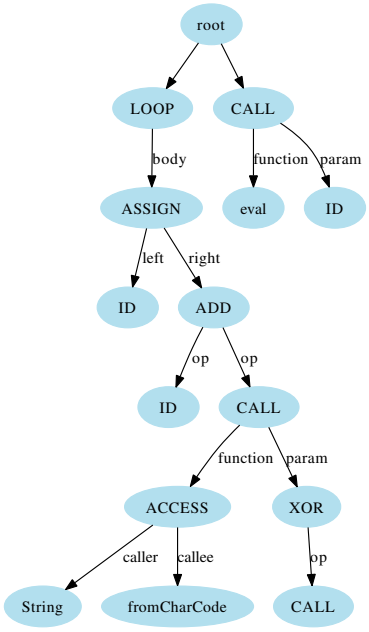

Figure 3. Induced subtree of the example eval unfolding.

subtree stemming from a given node. Figure 2 shows a bottom-up subtree of the example AST presented in Figure 1. Therefore, if the set of instructions at a given block-depth may vary upon insertion of instructions, the subtree changes and our method is not able to identify this subtree anymore. With induced subtrees [24], i.e., subtrees stemming from a given node, from which leaf nodes are arbitrarily removed, it is possible to ignore part of the code, likely invariants, and preserve tree structures that are really specific of an obfuscating transformation. Figure 3 shows an induced subtree of the example AST in Figure 1. Performing such pruning can be arbitrary (no knowledge involved) or can be formalized through specific rules obtained by mining (knowledge involved). The latter method may also rely on subjective decisions from an expert.

As for subtree selection, there were cases where representative subtrees were not obvious, so we had to arbitrarily agree on some rules as listed in Section V-B. However, there is potential to extrapolate co-clustering results to automatically extract recurring and representative subtrees. In particular, it helps when samples containing the same obfuscating transformation are very different, since the only similar code snippet becomes the very obfuscating transformation itself.

To achieve the daunting task of learning induced subtrees, we can rely on an open-source tool, Varro [25], which is able to discover frequently occurring subtrees in a set of trees. Varro introduces condensed canonically ordered trees for efficiently discovering frequently recurring unordered subtrees. Though it minimizes memory use so that moderately large treebanks are tractable on commonly available hardware, 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. We initially thought of making use of Varro but because of these drawbacks, it resulted unsuitable to efficiently process programming language-based treebanks.

VII. Conclusion

Abstract syntax trees allow to cancel the effects of string randomization, allowing to structurally match two programmatically equivalent scripts. In this paper, we have been successful in applying abstract syntax tree methods to scripting languages, allowing us to cluster similar obfuscated contents regardless of the original code. Learning remains a rigid process which needs continuous update but demonstrates unrivaled accuracy. In particular, our subtree matching system has been able to quickly classify samples based on the type of obfuscating transformation. It has also demonstrated some trends in combining several obfuscating transformations to encode different parts in a code, as well as the emergence of new combinations upon time. The small number of conbinations discovered let us think that many automated tools are used and that there is a finite set of these which we may identify. One major limitation of our approach is the model used: bottom-up subtrees are not enough flexible to match quasi-similar subtrees and may generate false negatives. Additionally, an appropriate subtree

selection should be implemented to ensure a low false positive rate. We plan on exploring feature selection methods to automate and maybe improve our learning results.

With current limitations being identified, we may be able to refine our proposal to improve its efficiency. Precisely detecting deobfuscation routines can allow to extract these and the related obfuscated strings. Gaining knowledge on obfuscating transformations will yield methods to reverse these, thus leading to the automation of deobfuscation, which is one of our goals. Applications of automated deobfuscation seemed limited to security purposes and encompass surveying obfuscations, assisting analysts and providing realtime safe browsing.

ACKNOWLEDGEMENTS

This experiment is part of a project supported by a grant from the NEC C&C Foundation.

REFERENCES

[1] Y. Choi, T. Kim, S. Choi, and C. Lee, "Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis," in *1st International Conference on Future Generation Information Technology*. Springer-Verlag, 2009, pp. 160–172.

[2] P. Likarish, E. Jung, and I. Jo, "Obfuscated Malicious JavaScript Detection using Classification Techniques," in *4th International Conference on Malicious and Unwanted Software*, Oct. 2009, pp. 47 –54.

[3] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection," in *20th USENIX Security Symposium*, Aug. 2011.

[4] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, "The Ghost In The Browser Analysis of Web-based Malware," in *1st Workshop on Hot Topics in Understanding Botnets*, 2007.

[5] M. A. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt, "Trends in Circumventing Web-Malware Detection," Available at: http://research.google.com/archive/papers/rajab-2011a.pdf, Google, Inc., Tech. Rep. rajab-2011a, Jul. 2011.

[6] G. Blanc and Y. Kadobayashi, "A Step Towards Static Script Malware Abstraction: Rewriting Obfuscated Script with Maude," *IEICE Transactions of Information and Systems*, vol. E94-D, no. 11, Nov. 2011, to appear.

[7] P. Ratanaworabhan, B. Livshits, and B. Zorn, "NOZZLE: A Defense against Heap-spraying Code Injection Attacks," in *18th USENIX Security Symposium*. USENIX Association, 2009, pp. 169–186.

[8] B. S. Baker, "On Finding Duplication and Near-duplication in Large Software Systems," in *Second Working Conference on Reverse Engineering*, Jul. 1995, pp. 86–95.

[9] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[10] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *8th Working Conference on Reverse Engineering*, Oct. 2001, pp. 301–309.

[11] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis," in *12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Aug. 2006, pp. 872–881.

[12] M. Chilowicz, E. Duris, and G. Roussel, "Syntax Tree Fingerprinting: a Foundation for Source Code Similarity Detection," in *17th IEEE International Conference on Program Comprehension*, May 2009, pp. 243–247.

[13] National Institute of Standards and Technology, "Secure Hash Standard," Available at: http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[14] R. L. Rivest, "MD5 Message-Digest Algorithm," Available at: http://tools.ietf.org/html/rfc1321.

[15] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Department of Computer Science, University of Auckland, Tech. Rep. 148, Jul. 1997.

[16] C. Craioveanu, "Server-side Script Polymorphism: Techniques of Analysis and Defense," in *3rd International Conference on Malicious and Unwanted Software*, Oct. 2008, pp. 9 –16.

[17] J. Barnette, "johnson," Available at: http://github.com/jbarnette/johnson/.

[18] B. Eich, "SpiderMonkey," Available at: http://www.mozilla.org/js/spidermonkey/.

[19] T. Flouri, J. Janoušek, and B. Melichar, "Subtree Matching by Pushdown Automata," *Computer Science and Information Systems*, vol. 7, no. 2, pp. 331–357, Apr. 2010.

[20] "Malware Domain List," Available at: http://malwaredomainlist.com.

[21] M. Akiyama, M. Iwamura, Y. Kawakoya, K. Aoki, and M. Itoh, "Design and Implementation of High Interaction Client Honeypot for Drive-by-Download Attacks," *IEICE Transactions on Communications*, vol. E93-B, no. 5, pp. 1131–1139, 2010.

[22] "anti-Malware engineering WorkShop (MWS)," Available at: http://www.iwsec.org/mws/2011/ (in Japanese).

[23] "Alexa Top 500 Global Sites," Available at: http://www.alexa.com/topsites.

[24] Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok, "Frequent Subtree Mining - An Overview," *Fundamenta Informaticae*, vol. 66, pp. 161–198, Nov. 2004.

[25] S. Martens, "Varro: An Algorithm and Toolkit for Regular Structure Discovery in Treebanks," in *23rd International Conference on Computational Linguistics: Posters*. Association for Computational Linguistics, 2010, pp. 810–818.