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

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 into analyzed samples using a pushdown automaton (PDA). The PDA accepts a set of subtrees representing obfuscating transformations we previously learned. Such quick and lightweight subtree matching algorithm has the potential to identify parts of a script that are obfuscated in order to extract these 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 distinguish obfuscation techniques, and gain knowledge on which techniques were 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 task is often hindered by anti-analysis tricks that render ineffective dynamic techniques. 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 occurring during a short-time crawling of malicious websites. Since string matching cannot be applied to identify obfuscating transformations, abstracting scripting contents is needed. Abstract syntax trees are a straightforward transformation of programs and allow to reduce 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: in Section II, 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 III. As we indicated, we evaluated our proposal against malicious JS scripts collected on a short period; the experiment is detailed in Section IV. We discuss the limitations of our proposal in Section V and conclude in Section VI.
II. 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 [1] 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 [2]. Indeed, Zozzle needs deobfuscated JavaScript samples to carry out malicious code detection. 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 removed 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 [3]. However, it can be evaded by renaming identifiers.

In the token-sequence based method [4], 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 [5], [6]. 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 [7].

B. AST Fingerprinting

Abstract Syntax Tree Fingerprinting (ASTF) was proposed by Chilowicz et al. [7]. 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 the subtree \( t \) (\( \rho(t) \)) and its parent (\( p(t) \)).

If a subtree \( t \) has \( n \) child subtrees, \( t_1, \ldots, t_n \), a hash value of subtree \( t \), \( \mathcal{H}(t) \), is \( \mathcal{H}(\rho(t)) \cdot \mathcal{H}(t_1) \cdot \ldots \cdot \mathcal{H}(t_n) \) where \( \mathcal{H} \) is a cryptographic hash function, such as SHA-1 [8] or MD5 [9]. 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. [10] and Craioveanu [11]. 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 currently-known techniques.

The issue of obfuscation is rarely raised and was almost considered an artifact of malicious contents until Provos et al. [12] 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 issues of obfuscation in JavaScript code.

In [13], the authors have observed that malicious web pages include obfuscated code 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 only identified 6 different obfuscation patterns. Oddly enough, their method was not able to detect an eval unfolding string as obfuscated.

In [14], 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.

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.

III. 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 [15].

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 assuming they are recurring subtrees in obfuscated scripts; then, we try to identify the presence of such obfuscating transformations by matching their 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 [16] which supports the SpiderMonkey [17] 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 the abstract syntax tree. However, the AST used here is different in some aspects from similar approaches [18], [1]:

- values are discarded and replaced by generic types: \texttt{NUM} for numeric values, \texttt{STR} for string values, \texttt{ID} for identifiers;
- some identifiers for core objects, e.g., \texttt{document}, and functions, e.g., \texttt{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, I \rangle where \texttt{S} represents a symbol (either \texttt{BRANCH} or \texttt{LOOP}) and \texttt{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 [18] but we rather focus on the hierarchical properties of tree-like structures like in [1].

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

```
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 decoding using the \texttt{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 introduced in the previous section. The obtained AST is then processed by a Perl script that computes its fingerprint as explained in Section II-B. The output of such processing provides a list of fingerprints of each subtree composing the script’s AST. We assume that recurring fingerprints among samples denote recurring subtrees. These subtrees should be the expression of obfuscating transformations provided our dataset is composed of obfuscated JavaScript files. Fingerprints varying 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 of which members are similar using 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

After discovering frequent recurring subtrees appearing in a learning dataset, we will build transition rules from the learned subtrees. 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 automaton, which is analogous to a finite state machine with a stack. 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 leaded 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 accommodating 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 [19]. 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 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 as the script’s AST is browsed.

IV. 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 script ASTs, obfuscating transformations learned in a previous dataset. As suggested here, we apply the two techniques presented in Section III at two distinct stages: recurrent ASTFs indicate frequently occurring subtrees in the learning set, which we assume to be obfuscation routines. Obviously, having clean datasets of obfuscated samples will enhance accuracy, while we wish the learning set to be as exhaustive as possible in order to cover all obfuscating transformations, assuming the size of the subtree, is less than 150, in order to reduce the number of transitions for the automata to be built.

\[ \text{conc} \]

- 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.

Overall, we extracted 33 subtrees divided into 17 different obfuscating transformations, based on the set they were discovered from. These 33 subtrees were as small as 7 nodes and as big as 127 nodes. Generating the transitions for the pushdown automaton yielded 37730 transitions.

During the second stage, we generated prefix notations for JS samples from datasets other than the learning one. These prefix notations are based on the AST of these JS files and are feeded to the pushdown automaton. Each time, a subtree is matched, a message indicates which subtree has been discovered. Doing so, we are able to detect samples that are obfuscated with transformations discovered in the learning dataset. We identified around 19 different transformations or combination of transformations. In particular, 55 samples out of 82 in Day 2 (February 14th) were found to contain subtrees we learned, encompassing 15 different obfuscating transformations. The Day 3 dataset had 100 samples out of 116 that contained around 15 different obfuscating transformations. 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 transformation. One can argue that this transformation’s AST may be prone to false positives. The results are displayed in Table I where each testing dataset is shown independently. To better synthesize the results, samples are clustered by obfuscating transformations, which are a single subtree or a combination of subtrees. Table I also indicates for each cluster the presence or absence of a given subtree and the number of samples where the given combination of subtrees has been identified. In particular, we can observe that some obfuscating transformations are exclusive to one day or the other (6, 11 and 12 for Day 2; 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 subtree L may not be relevant (7 nodes, the smallest subtree of the learning set) while the obfuscating transformation denoted by the ID 19 only relies on a single subtree. In fact, much more credit should be given to obfuscating transformations involving every subtree of a given obfuscation ID.

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 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.

V. 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 programatically 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 subtree stemming from a given node. Figure 3 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 4 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 IV-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.

VI. CONCLUSION

Abstract syntax trees allow to cancel the effects of string randomization, allowing to structurally match two programatically 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, and of the string-edit distance. Learning remains a rigid process
which needs continuous update but demonstrates unrivaled accuracy. In particular, our subtree matching system has been
to quickly classify samples based on the type of obfuscating transformations the sample implements. 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 combination upon time. The small number of combinations discovered let us think that many
automated tools are used and that there is a finite set of these which we can therefore identify. One major limitation
of our approach is the model used: bottom-up subtree 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 safesurfing.

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

REFERENCES
Fast and Precise In-Browser JavaScript Malware Detection,”
Figure 4. Induced subtree of the example eval unfolding.