Optimizing Away JavaScript Obfuscation

Adrian Herrera
Defence Science and Technology Group
adrian.herrera@dst.defence.gov.au

***Abstract*—JavaScript is a popular attack vector for releasing malicious payloads on unsuspecting Internet users. Authors of this malicious JavaScript often employ numerous obfuscation techniques in order to prevent the automatic detection by antivirus and hinder manual analysis by professional malware analysts. Consequently, this paper presents SAFE-DEOBS, a JavaScript deobfuscation tool that we have built. The aim of SAFE-DEOBS is to automatically deobfuscate JavaScript malware such that an analyst can more rapidly determine the malicious script's intent. This is achieved through a number of static analyses, inspired by techniques from compiler theory. We demonstrate the utility of SAFE-DEOBS through a case study on real-world JavaScript malware, and show that it is a useful addition to a malware analyst's toolset.**

***Index Terms*—javascript, malware, obfuscation, static analysis**

I. INTRODUCTION

The past decade has seen JavaScript's popularity steadily increase (and continue to increase): the language is supported by all modern web browsers and used in 95% of all websites [1]; it constantly ranks within the top ten most popular programming languages [2–4]; and thousands of libraries and frameworks have been built on top of it [5]. However, this pervasiveness has a dark side: the ubiquity of JavaScript on the Internet has also made it popular amongst people with malicious intent. For example, JavaScript is commonly used for gaining initial code execution via a browser or PDF reader vulnerability [6, 7], installing cryptocurrency miners [8], and in cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks.

The rise of malicious JavaScript ("JavaScript malware") has resulted in a renewed focus by both antivirus companies and security researchers [7–12]. In turn, authors of JavaScript malware have increasingly turned to *obfuscation* as a means of (i) hiding from automatic detection by anti-virus products, and (ii) hindering manual analysis by professional malware analysts. Thus, tools are required to undo this obfuscation so that JavaScript malware can be detected by antivirus engines and more easily inspected and understood by malware analysts.

We propose and prototype a tool that accomplishes these goals. Our tool, SAFE-DEOBS, is a static analyzer built on top of the SAFE framework [13, 14]. It repurposes a number of common compiler optimizations for the purpose of deobfuscating JavaScript malware in order to make it more understandable by subsequent automatic and manual analyses. This paper summarizes our experiences designing, implementing, and evaluating SAFE-DEOBS. Our primary contributions are:

- Applying techniques rooted in compiler theory to the task of deobfuscating JavaScript malware;
- The design and implementation of SAFE-DEOBS, an open-source tool to assist malware analysts to better understand JavaScript malware; and
- An evaluation of SAFE-DEOBS on a large corpus of real-world JavaScript malware.

Unless otherwise stated, all malicious code used in this paper is taken from real-world malware.

II. BACKGROUND AND RELATED WORK

Software obfuscation has many legitimate uses: digital rights management, software diversity (for software protection), and tamper protection, to name a few. However, software obfuscation is being increasingly co-opted by malware authors to thwart program analysis (both automated and manual).

When discussing JavaScript obfuscation (and obfuscation of other scripting languages; e.g., PHP, PowerShell), it is important to differentiate obfuscation from *minification*. Minification reduces code size by removing unnecessary characters/strings (e.g., whitespace and comments) and shortening variable names. Small code size means less data to download over the Internet, which leads to reduced web page load times. In contrast, the aim of obfuscation is to make the code difficult to read and understand. Undoing most minification (commonly referred to as "beautification") is trivial—e.g., by (re)inserting whitespace—and many tools exist to do this (e.g., UglifyJS [15], which we applied to the code in Listing 1). In contrast, deobfuscation often requires advanced program analyses (e.g., symbolic execution [16] and abstract interpretation [17]). Like Lu and Debray [10], we do not consider minification as obfuscation.

```
1 // From 20151226_9474ac02eae3bbe9bcf19d94c8e68a25
2 var str = "5553515E0A0D0108174A0E05010"/* ... */;
3 var k6 = ';', e5 = '%")+', h3 = '', w9 = eval,
4 /* 218 more variables */;
5
6 h3 += d1;
7 h3 += x0;
8 h3 += d7;
9 /* 215 more operations */
10 w9(h3);
```

Listing 1: A snippet of (beautifed) obfuscated JavaScript. Comments have been inserted by us.

Listing 1 shows a snippet of an obfuscated JavaScript malware sample. To a human (even a highly-trained malware

analyst), it is not immediately obvious what the code in Listing 1 does: this code defines 222 variables and performs 219 operations on these variables (most of which are not shown, due to space limitations). An automated tool attempting to either signature the malware or extract features (e.g., callback URLs, API calls) will also face difficulties.

Furthermore, JavaScript presents the opportunity to apply obfuscation techniques that are generally unapplicable to compiled languages (e.g., C, C++, and Java). For example, previous work [9, 10] has observed the following obfuscation techniques actively used in the wild:

String splitting: The conversion of a single string into the concatenation of several substrings, as observed on Lines 6 to 8 in Listing 1.

Keyword substitution: Storing keywords in variables, such as `eval` being stored in `w9` on Line 3 and called via `w9` on Line 10 in Listing 1.

String encoding: Encode strings so that they are not readable, e.g., via escaped ASCII characters, hexadecimal, or Base64 representations.

Finally, JavaScript's "*quirky semantics*" [13] deserve special mention. These include:

Variable hoisting: JavaScript allows variable declarations *after* the variable has been used. These variable declarations are moved ("hoisted") to the top of the scope in which they are used.

`with` scoping: Extends the scope of the current statement, which is specially noted in the Mozilla developer documentation "*as it may be the source of confusing bugs*" [18].

`eval`: Dynamically executes JavaScript code and is a known security risk [19].

These semantics can be abused (by both malicious and benign code alike) to make both static analysis and deobfuscation difficult [7, 13].

To this end, existing work on JavaScript (de)obfuscation has mostly focused on detecting (and preventing) malicious JavaScript, often using machine learning [11, 20–23] or program analysis [7] techniques. Machine learning approaches (e.g., random forests, support vector machines) have been shown to be effective when combined with *semantic* features (e.g., control flow and program dependency graphs, as used by JSTAP [22]). Similarly, program analysis techniques (such as abstract interpretation) that operate on (an abstraction of) JavaScript's semantics have also been effective at detecting JavaScript malware [7]. This is likely due to the fact that obfuscation must maintain the original code's *meaning* (i.e., its semantics), while syntactic features are easier to manipulate. However, this focus on *detection* has traditionally eschewed *readability*: the focus of deobfuscation. Our work therefore complements much of this existing research.

A. Related Work

Lu and Debray [10] combine dynamic analysis (to capture a JavaScript execution trace) and static analysis (backward slicing of the execution trace to create a simplified Abstract Syntax Tree) to produce *observationally equivalent* output JavaScript. Dynamic analysis allows for code "hidden" in an `eval` call to be analyzed and deobfuscated. However, dynamic analysis can also be thwarted by environmental checks that may not be satisfied when the malware is under analysis. Their prototype is also not publicly available.

JSNICE uses "big code" and machine learning to predict program properties (e.g., variable names and type annotations) for JavaScript code [24]. While not open-source, the JSNICE authors have created a website[1] that promises to "*make even obfuscated JavaScript code [uploaded to the website] readable*". Unfortunately, uploading the code from Listing 1 resulted in the code in Listing 2: while the type annotations are accurate, this code is no-more readable than that in Listing 1.

```
1 'use strict';
2 /** @type {string} */
3 var str = "5553515E0A0D0108174A0E05010"/* ... */;
4 /** @type {string} */
5 var k6 = ";";
6 /** @type {string} */
7 var e5 = '%")+';
8 /** @type {string} */
9 var h3 = "";
10 /** @type {function(string): *} */
11 var w9 = eval;
12 /* 218 more variables */
13
14 /** @type {string} */
15 h3 = h3 + d1;
16 /** @type {string} */
17 h3 = h3 + x0;
18 /** @type {string} */
19 h3 = h3 + d7;
20 /* 215 more operations */
21 w9(h3);
```

Listing 2: A snippet of the JavaScript from Listing 1 "deobfuscated" by JSNICE. Comments enclosed within `/** */` (i.e., the type annotations) were generated by JSNICE. Other comments have been inserted by us.

Other tools, such as "deobfuscate javascript"[2] perform deobfuscation by intercepting calls to `eval` and `write`. This approach is able to successfully deobfuscate the sample in Listing 1 (due to the `eval` substitution on Line 10). However, as noted on their website: "*some malicious scripts may not employ these functions and may therefore infect your browser*".

This opens the door for a new, open-source tool for JavaScript malware deobfuscation. To this end, the following sections discuss the design and implementation of our contribution to this field.

III. DESIGN AND IMPLEMENTATION

The SAFE-DEOBS workflow is illustrated in Fig. 1. The obfuscated JavaScript is parsed into an Abstract Syntax Tree

[1] http://www.jsnice.org/

[2] http://deobfuscatejavascript.com/

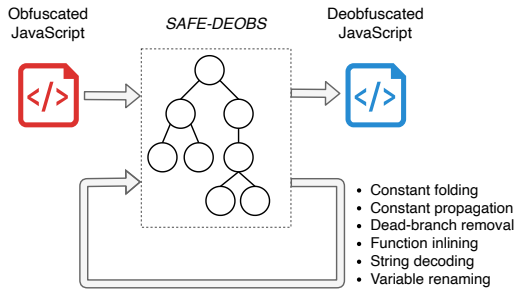


Fig. 1: The SAFE-DEOBS workflow.

(AST), upon which a number of deobfuscation passes are performed (much like a compiler performing a set of optimization passes). These passes are continuously applied until a fixed point is reached, at which point the AST is serialized back into JavaScript source code. The set of deobfuscation passes that we developed (listed in Fig. 1) will be described in Section III-B. But first, the JavaScript must be transformed into a form amendable to analysis.

A. Preprocessing

We reuse existing tools to parse and preprocess the input JavaScript. Specifically, we use the Scalable Analysis Framework for ECMAScript (SAFE v2.0) [13, 14] to parse JavaScript into an AST that can be further analyzed.

SAFE is a scalable analysis for JavaScript[3] that provides different levels of intermediate representations (IR). We selected SAFE because it is open-source and "*especially designed as a playground for advanced research in JavaScript web applications*" [14]. Furthermore, SAFE is primarily written in Scala, a functional programming language. Features standard in functional programming languages—such as pattern-matching and support for efficient recursion—lend themselves well to writing analyses/transformations that operate on trees (i.e., ASTs).

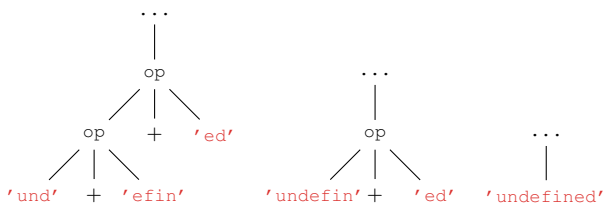
While SAFE provides three levels of IR, we only use the lowest level: the AST. The higher levels (the Intermediate Representation and Control Flow Graph) are not used because they only support one-way translation: from lower levels up to higher levels. As the higher IRs are aimed at analysis, rather than transformation, there is no mechanism to translate higher IRs back to JavaScript.

Finally, we make use of two features in SAFE's AST translator that simplify further analysis: the `Hoister` and `WithRewriter`. The `Hoister` lifts variable declarations so that they appear at the top of the current scope, separating declarations from initializations. The `WithRewriter` conservatively eliminates `with` statements such that lexical scoping remains valid. After this, the AST is ready for deobfuscation.

[3]We use ECMAScript and JavaScript interchangeably throughout this paper.

```
if (typeof ifopracxa == 'und' + 'efin' + 'ed')
```

(a) The original JavaScript code.



(b) The AST as constant folding is applied (from left to right).

Fig. 2: Constant folding example.

B. Deobfuscation Passes

After preprocessing, SAFE-DEOBS performs a number of "deobfuscation passes" ("phases" in SAFE parlance) on the AST until a fixed point is reached. These phases include: (i) constant folding; (ii) constant propagation; (iii) dead-branch removal; (iv) function inlining; (v) string decoding; and (vi) variable renaming, totalling 2474 LOC of Scala. The first four phases should be familiar to those with an understanding of common compiler optimizations, while the last two are specific to scripting languages. We describe each of these phases in the following sections.

1) Constant Folding: Constant folding aims to recognize and evaluate constant expressions. This process is illustrated in Fig. 2, where *string splitting* has been applied to a string literal (Fig. 2a) in an `if` condition. The original string can be recovered by traversing the AST (Fig. 2b) and rewriting nodes where an arithmetic operation occurs on two constant nodes.

SAFE's AST representation and Scala's pattern-matching feature make this straightforward to implement: Listing 3 gives an example of one such rule (the concatenation of strings with integers). While we have implemented 59 rules, JavaScript's quirky semantics (Section II) means that corner-cases may remain unhandled. Fortunately, these rules are straightforward to extend.

```
1 case InfixOpApp(StringLiteral(quote, str, false),
2                 Op("+"), IntLiteral(int)) =>
3   StringLiteral(quote, s"${str}${int}", false)
```

Listing 3: An example of a constant folding rewrite rule. This rule matches on a string concatenated with an integer, resulting in a single string literal (according to JavaScript's semantics).

2) Constant Propagation: Constant propagation is the substitution of known literal values into expressions. Unlike constant folding (Section III-B1), constant propagation requires (i) maintaining state while the AST is traversed, and (ii) multiple traversals of the AST.

The state of a variable's constness must be tracked during AST traversal: value substitution can no longer occur once

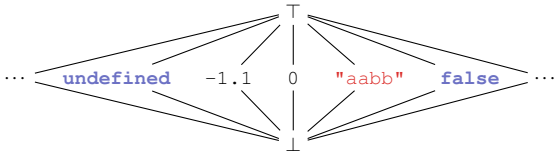


Fig. 3: The constant propagation lattice.

a variable's constness can no longer be guaranteed. We implement this as an abstract interpretation over the three-level lattice typically used for constant propagation [25] (Fig. 3). This lattice contains an infinite number of middle elements, representing constant values. A variable goes to ⊤ once its value is no longer constant.

Once constants have been propagated through the AST, the AST is traversed again to remove redundant variable assignments.

3) Dead-branch Removal: Dead code is often inserted to confuse and distract a malware analyst. This dead code is often revealed by other deobfuscation phases (in particular, constant folding and propagation). The simplest form of dead code removal is when a constant (i.e., `Literal` AST node) appears in an **`if`** condition. In this instance, we can statically determine which branch will *always* execute, and remove the other branch. We apply the same technique to **`switch`** statements, removing unreachable **`case`**s.

4) Function Inlining: Function inlining expands trivial function calls, where the complexity of the analysis determines what is "trivial". Like the dead code discussed in Section III-B3, obfuscation may introduce (i) functions that are never called, and/or (ii) functions that perform trivial operations, but nevertheless adds to an analyst's cognitive load.

Our prototype inlines functions where the function's AST consists of a single `Return` statement that returns a `Literal` expression. Examples of such functions are given in Listing 4. The first function is trivial to inline. The second function (`Ph`) becomes inlinable after string decoding and constant propagation (Sections III-B2 and III-B5 respectively). Again, Scala's pattern-matching feature allows for a concise rule to find such functions (Listing 5).

```
1 // From 20170124_a0b2eeedbc9c6187927e32645700d1d2
2 function zdykuvpobrenusdegvusipasad/*- ... -*/() {
3   return [ /*- ... -*/ , "ing", /*- ... -*/];
4 }
5
6 // From 20190808_536f24111b28ff9febcdaef4ceb47adb
7 function Ph() {
8   var fHC=String.fromCharCode(6688/88+0);
9   nDO = fHC + String.fromCharCode(2600/52-0);
10  /* 7 more operations */
11  oOw = Oy + String.fromCharCode(16*5);
12  return oOw;
13 }
```

Listing 4: Examples of inlinable functions.

```
1 // Match the last statement in the function body
2 // Precondition: funcBody contains a single stmt
3 funcBody.last match {
4   // Function returns a literal expression
5   case Return(Some(lit: Literal)) => Some(lit)
6   // Function returns nothing
7   case Return(None) => EmptyExpr
8   // Function returns one of its parameters
9   case Return(Some(VarRef(id))
10    if params.exist(_ == id) =>
11      // Return the param with the given id
12  // Cannot inline
13  case _ => None
14 }
```

Listing 5: Pattern-matching rules for an inlinable function.

```
1 var lI11II;
2 if (IIIlII() == lI11I1)
3   lI11lI();
4
5 lI11II = 11I1I(lI11I1);
```

(a) The original code.

```
1 var dog; // lI11II
2 if (cat() == parrot)
3   lion();
4
5 dog = tiger(parrot);
```

(b) After variable renaming.

Fig. 4: An example of variable/function renaming.

Once inlinable functions are found (by traversing the AST), the AST is traversed (again) so that all `FunApp` (i.e., function application) nodes that call an inlinable function are replaced with the `Literal` expression returned by the function. For example, calls to the first function in Listing 4 are replaced with the returned array literal, while calls to the second function are replaced with the literal value in `oOw` (after string decoding and constant propagation have been applied).

5) String Decoding: As discussed in Section II, strings can be encoded such that they are unreadable by most analysts. Common string encoding schemes include hexadecimal (typically found in string literals, e.g., `'\x68\x65\x6c\x6c\x6f'`) and unicode (commonly decoded using the `String.fromCharCode` function, as used in Lines 8 to 11 in Listing 4). These encoding patterns are straightforward to find—by looking for `StringLiteral` AST nodes that contain hexadecimal-encoded strings and calls to `String.fromCharCode`, respectively—and rewrite.

The encoding schemes thus far have all utilized language features built into the JavaScript language. Of course, malware authors are free to implement other encoding (e.g., Base64) or encryption (e.g., RC4) schemes. While statically detecting these encoding/encryption schemes is impossible in the general-case, it may still be possible to employ heuristics to detect such functionality. While our prototype does not support this, SAFE provides an ideal environment to develop and experiment with such heuristics.

6) Variable Renaming: Finally, variables (and functions) can be given complex, confusing and/or similar names to increase an analyst's cognitive load. This is demonstrated in Fig. 4a.

Variable names that share common prefixes complicate vari-

able renaming via regular expressions (depending on the order in which variables are renamed). Therefore, we developed an optional[4] SAFE phase that renames all variables to animal names. Animals are deterministically selected so that repeated deobfuscation of the same sample produces the same result. The original variable names are placed alongside renamed variable definitions, in case analysts are required to refer back to the original malware sample, as in Fig. 4b.

IV. Evaluation

Here we present (i) a case study on a particular malware sample (Section IV-A); and (ii) a high-level analysis over 39 450 malware samples (Section IV-B). Both of these discussion will use the open-source dataset from Hynek Petrak [26].

A. Case Study

This case study is based on the JavaScript malware sample `20170110_9330ee612a9027120543d6cd601cda83`, which is publicly available from our dataset.

This particular sample has not been minified and consists of 475 LOC, contains 14 functions, and defines 214 variables. This sample makes for an interesting case study because it is one of the few samples that does not use `eval`, and "deobfuscate javascript" (Section II) is therefore unable to deobfuscate it (due to its reliance on hooking `eval`). Interestingly, JSNice is unable to infer any of the 14 functions' return types, despite all of these functions being inlinable (according to our inline rules, described in Section III-B4) and therefore relatively straightforward to analyze.

Listing 6 shows the deobfuscated sample. All 14 functions have been inlined and 211 variables have been eliminated through a repeated combination of constant folding and propagation (a fixed point was reached after four iterations). The number of lines has been reduced by 97 % to 12 LOC.

Unfortunately, `hamster` (Line 6) remains because we do not model the Document Object Model (DOM, of which the `window` object is an element of). Nevertheless, it is now much easier to reason about the sample's behavior. Alas, this behavior primarily consists of executing a string of obfuscated PowerShell (Line 10). Clearly, SAFE-Deobs would benefit from integration with other malware analysis/deobfuscation tools.

B. Generalizability

We examined all 39 450 malware samples in our dataset to obtain a high-level understanding of our tool's efficacy. First, we removed 7109 invalid samples (18.02 % of the dataset) that escomplex [27]—a tool for performing software complexity analysis on JavaScript ASTs—failed to parse. Second, we "normalized" the dataset by running all remaining 32 341 through SAFE's `astRewrite` phase, which hoists variable definitions, rewrites `with` statements, and beautifies the code (i.e., undoes minification). This allowed us to remove duplicate samples (identified by SHA512 checksum), leaving 28 285 samples (71.70 % of the original dataset).

[4]Occasionally, malware will declare variables such as `var exploitation`, which are useful names.

```
1 var lion; // edeb
2 var hamster; // uvacdykadq
3 var chinchilla; // cqorobcit
4
5 lion = WScript;
6 hamster = typeof window == "undefined";
7 {
8   chinchilla = lion.CreateObject('WScript.Shell');
9   if (hamster) {
10    chinchilla["run"]('cmd.exe /c \"powershell
          $ojogo=\'^dimas.top\';$hetfo=\'^-Scope  Pr
          \';$pobbi=\'^,$path); \';$innypu=\'^ocess;
           $p\';$monsucm=\'^y Bypass \';$binkucb=\'^
          h\';$ykpyffy=\'^Start-Pro\';$ykjygr=\'^:
          temp+\'\'\b\';$uzmez=\'^e\'\');(New-\';
          $xzymo=\'^Set-Execu\';$ulirgo=\'^tp://laro
          \';$eqtem=\'^ath=($env\';$evyvz=\'^).
          Downloa\';$ogxow=\'^Webclient\';$utkyjv
          =\'^/777.exe\'\'\';$gsydibv=\'^tionPolic
          \';$upoh=\'^stem.Net.\';$zceqmi=\'^Object
          Sy\';$cepsuhm=\'^ipbafa.ex\';$qfyzko=\'^
          dFile(\'\'ht\';$awysqe=\'^cess $pat\';
          Invoke-Expression ($xzymo+$gsydibv+
          $monsucm+$hetfo+$innypu+$eqtem+$ykjygr+
          $cepsuhm+$uzmez+$zceqmi+$upoh+$ogxow+
          $evyvz+$qfyzko+$ulirgo+$ojogo+$utkyjv+
          $pobbi+$ykpyffy+$awysqe+$binkucb);\"', 0);
11  }
12 }
```

Listing 6: Case study sample after deobfuscation.

TABLE I: Complexity metrics before and after deobfuscation (using the normalized, deduplicated samples from our dataset).

Metric	Before	After	% decrease
Total physical LOC	46 724 630	45 491 768	2.64
Total num. functions	324 441	241 091	25.69
Mean cyclomatic complexity	10.58	8.68	17.96
Mean Halstead length	5994.62	4297.03	28.31

Finally, we ran SAFE-Deobs over the 28 285 deduplicated samples and used escomplex to compare common software complexity metrics before and after deobfuscation. These complexity metrics include: (i) physical lines of code (LOC); (ii) number of functions; (iii) cyclomatic complexity [28]; and (iv) Halstead length [29]. The results are presented in Table I.

We used both escomplex's report and manual inspection to verify deobfuscation correctness. Unfortunately, JavaScript malware is difficult to verify via dynamic behavioral analysis because: (i) the malware's output may vary depending on its intent (e.g., downloading a second-stage implant, exploiting a vulnerability) and may not be readily apparent; (ii) the malware may require complex "trigger conditions" [30] to activate the intended behavior; and (iii) the malware may target a particular browser. Nevertheless, we found that SAFE-Deobs successfully processed our malware corpus and greatly reduced the complexity of the code contained within.

V. Conclusion

In this paper we present SAFE-DEOBS, a static analyzer for deobfuscating JavaScript malware. While none of the techniques that we propose are particularly novel in their own right—indeed, Garba and Favaro [16] also proposed "*deobfuscation by optimization*"—little work has been published on applying these techniques to JavaScript malware. We have demonstrated SAFE-DEOBS' utility by applying it to a large corpus of real-world malware, and shown that it makes for a useful addition to a malware analyst's toolset. SAFE-DEOBS is open-source and available to malware analysts at https://github.com/DSTCyber/safe-deobs.

References

[1] W^{3}Techs, "Usage statistics of JavaScript on websites," https://w3techs.com/technologies/details/cp-javascript, 2019.

[2] Stack Overflow, "Developer survey results," https://insights.stackoverflow.com/survey/2019, 2019.

[3] TIOBE, "TIOBE index for November 2019," https://www.tiobe.com/tiobe-index, Nov. 2019.

[4] Github, "The state of the Octoverse," https://octoverse.github.com, 2019.

[5] Module Counts, "Module counts," http://www.modulecounts.com, 2019.

[6] S. Groß, "CVE-2019-11707," May 2019.

[7] A. Jordan, F. Gauthier, B. Hassanshahi, and D. Zhao, "Unacceptable behavior: Robust PDF malware detection using abstract interpretation," in *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, ser. PLAS'19, 2019.

[8] Trend Micro, "JavaScript malware in spam spreads ransomware, miners, spyware, worm," Online, Jan. 2019.

[9] W. Xu, F. Zhang, and S. Zhu, "The power of obfuscation techniques in malicious JavaScript code: A measurement study," in *2012 7th International Conference on Malicious and Unwanted Software*, Oct. 2012.

[10] G. Lu and S. Debray, "Automatic simplification of obfuscated JavaScript code: A semantics-based approach," in *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, ser. SERE '12, 2012.

[11] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "JAST: Fully syntactic detection of malicious (obfuscated) JavaScript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2018.

[12] A. Fass, M. Backes, and B. Stock, "HideNoSeek: Camouflaging malicious JavaScript in benign ASTs," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, 2019.

[13] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu, "SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript," in *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages*.

[14] J. Park, Y. Ryou, J. Park, and S. Ryu, "Analysis of JavaScript web applications using SAFE 2.0," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017.

[15] M. Bazon, "UglifyJS," http://lisperator.net/uglifyjs, 2018.

[16] P. Garba and M. Favaro, "SATURN – software deobfuscation framework based on LLVM," in *Proceedings of the 3rd ACM Workshop on Software Protection*, ser. SPRO'19, 2019.

[17] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi, "Opaque predicates detection by abstract interpretation," in *Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology*, ser. AMAST'06, 2006.

[18] Mozilla Developer Network, "with statement," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with, Mar. 2020.

[19] ——, "eval," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval, May 2020.

[20] K. Rieck, T. Krueger, and A. Dewald, "Cujo: Efficient detection and prevention of drive-by-download attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10, 2010.

[21] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "ZOZZLE: Fast and precise in-browser JavaScript malware detection," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11, 2011.

[22] A. Fass, M. Backes, and B. Stock, "JSTAP: A static pre-filter for malicious JavaScript detection," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19, 2019.

[23] J. W. Stokes, R. Agrawal, G. McDonald, and M. Hausknecht, "ScriptNet: Neural static analysis for malicious JavaScript detection," 2019.

[24] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, 2015.

[25] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 2, Apr. 1991.

[26] H. Petrak, "Javascript malware collection," https://github.com/HynekPetrak/javascript-malware-collection, 2019.

[27] P. Booth, "escomplex," https://github.com/escomplex/escomplex, 2017.

[28] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308—320, Jul. 1976.

[29] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. USA: Elsevier Science Inc., 1977.

[30] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, *Automatically Identifying Trigger-based Behavior in Malware*. Boston, MA: Springer US, 2008, pp. 65–88.