2012 IEEE Sixth International Conference on Software Security and Reliability

Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach

Gen Lu Saumya Debray
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
Email: {genlu, debray}@cs.arizona.edu

***Abstract*—JavaScript is a scripting language that is commonly used to create sophisticated interactive client-side web applications. However, JavaScript code can also be used to exploit vulnerabilities in the web browser and its extensions, and in recent years it has become a major mechanism for web-based malware delivery. In order to avoid detection, attackers often take advantage of the dynamic nature of JavaScript to create highly obfuscated code. This paper describes a semantics-based approach for automatic deobfuscation of JavaScript code. Experiments using a prototype implementation indicate that our approach is able to penetrate multiple layers of complex obfuscations and extract the core logic of the computation, which makes it easier to understand the behavior of the code.**

***Keywords*-web security; deobfuscation; dynamic analysis; program slicing**

I. Introduction

Recent years have seen a dramatic increase in web-based malware delivery. This is typically done via a process, known as *drive-by-downloading*, that exploits vulnerabilities in web browsers and/or their extensions, and can be delivered through a variety of web-based mechanisms [1]. Drive-by-downloads often rely on JavaScript, a scripting language widely used for generating dynamic web content. Attackers often take advantage of the dynamic nature of JavaScript to create highly obfuscated code to avoid detection [2]. The growing prevalence of malicious JavaScript code is exemplified by the Gumblar worm, which uses dynamically generated and heavily obfuscated JavaScript to avoid detection and identification, and which at one point was considered to be the fastest-growing threat on the Internet [3].

Identifying malicious JavaScript code is not easy, however. The mere presence of obfuscated JavaScript does not, in itself, signal the presence of malicious content, since benign web pages also to use code obfuscation to protect intellectual property [4], [5]. Moreover, attackers often use server-side scripting to deliver randomly obfuscated code where each instance is syntactically different from the next (*server-side polymorphism*). For these reasons, static signature-based heuristics (e.g., "'eval(' *and* 'unescape(' *within 15 bytes of each other*" [3]) have limited success when dealing with obfuscated JavaScript. Traditional anti-virus tools that process web pages typically rely on such syntactic heuristics and so tend to produce a high misidentification rate [2], [4], [5].

A better solution would be to use semantics-based techniques that focus on code behavior. Unfortunately, current behavioral analysis techniques for obfuscated JavaScript typically require considerable manual intervention, e.g., to modify the code in specific ways or to monitor its execution within a debugger [6]–[8]. There has been some recent work on automated behavioral analyses of obfuscated JavaScript that in many cases has a "deobfuscator" component [4], [5], [9], [10], as well as standalone JavaScript deobfuscation tools [11]–[14]. These deobfuscators all rely on some simple and intuitive assumptions about the obfuscation and the structure of the obfuscated code. Although these assumptions seem plausible, it is not difficult to construct obfuscations that violate them and thereby defeat the corresponding deobfuscators. This is illustrated in Section IV.

This paper proposes a different approach to analyze obfuscated JavaScript code. We collect bytecode execution traces from the target program and use dynamic slicing and semantics-preserving code transformations to automatically simplify the trace, then reconstruct deobfuscated JavaScript code from simplified trace. The code so obtained is *observationally equivalent* to the original program for the execution path considered, but has obfuscations simplified away, thereby exposing the core logic of the computation performed by the original code. The resulting code can then be examined manually or fed to other analysis tools for further processing. Our approach differs from existing approaches in that it makes no assumptions about the structure of the obfuscation and uses semantics-based techniques to reveal the behavior of the code.

In previous work [15], we have described a semantics-based approach to deobfuscating "core" JavaScript, i.e. code that runs on a standalone JavaScript engine. This does not account for interactions between the executing JavaScript code and the host browser, which provides the Document Object Model (DOM) for manipulating HTML documents and the ability to interact with external websites. This paper, by contrast, focuses on the more challenging task of deobfuscating JavaScript code in the full context of the browser's execution environment. This is much more complex than the "core" JavaScript considered in [15], and admits many more options for obfuscation, but is able to handle the full range of behaviors of real JavaScript malware. We evalute our approach using a prototype implemention and test it againest both synthetic programs and real malware code. The results show that our approach can penetrate multiple layers of complex obfuscations, some of which cannot be handled by existing techniques, and extract the core logic of the underlying computation.

978-0-7695-4742-8 2012
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/SERE.2012.13

IEEE computer society

II. Background

This section provides an overview of the JavaScript language and the host environment of web browser and describes some widely used real-world code obfuscation techniques.

A. JavaScript Basics

The term "JavaScript," commonly used to refer to a scripting language used for client-side programming of dynamic websites, consists of the core programming language together with the host enviroment, namely, the Document Object Model (DOM) provided by web browser.

The core JavaScript language provides a set of data types (e.g. Boolean, String, Object), a collection of built-in objects and functions (e.g. RegExp, Math, Date), and a prototype-based inheritance mechanism, among other things. Like most scripting languages, JavaScript is highly dynamic in nature. It is dynamically typed, which means that a variable can take on values of different types at different points in a program. Properties of (associative-array based) objects can be added/deleted on the fly, and code can be generated from strings at runtime using the built-in eval function.

The Document Object Model (DOM) is an API that abstracts HTML documents as a structural representation of objects and provides a mechanism for manipulating this abstraction, thereby enabling JavaScript code to modify and interact with the content of web pages dynamically. For example, write method of the document object can be used to dynamically write HTML expressions or JavaScript code to a document. In contrast to the built-in objects defined in core JavaScript, objects defined in the DOM specification are called "host objects", and are provided as a part of the host environment by the web browser.

B. JavaScript Runtime

At the implementation level, JavaScript typically uses an expression-stack-based byte-code interpreter; modern implementations of these interpreters usually come with JIT compilers. For example, Mozilla's popular FireFox web browser uses an open source JavaScript interpreter, SpiderMonkey, written in C/C++ [16]. This is a single dispatching function that steps through the bytecode one instruction at a time.

As discussed in previous sections, client-side JavaScript programs have the ability to generate code at runtime, using various mechanisms provided by both the interpreter and browser. Further, dynamic code generation can be multi-layered, e.g., a string that is eval-ed may itself contain calls to eval, and such embedded calls to eval can be stacked several layers deep. We refer to such dynamic code generating as *code unfolding*, and for each piece of code generated by runtime unfolding, we call it a *code context*. Functions that are defined in JavaScript (using the keyword 'function') are called *non-native functions*; and functions provided by the interpreter or browser (e.g. built-in functions and methods of host objects) are called *native functions*. Unlike non-native functions, native functions do not generate a bytecode trace when executed.

```
1  <div id='x'>HelloWorld</div>
2  <script>
3  a = document; b = "Id";
4  var c = a[f()]('x')['i'+'nn'+'erH'+'TML'];
5  function f(){
6    var p='ent'+'By'+b;
7    var q='get'+'Elem';
8    return q+p;
9  }
10 </script>
```

Fig. 1. Example of obfuscated JavaScript code

C. JavaScript Obfuscation Techniques

The dynamic nature of Javascript code makes possible a variety of obfuscation techniques. What's particularly challenging is the combination of the ability to execute a string using code unfolding, as described above, and the fact that the string being "executed" may be obfuscated in a wide variety of ways. Howard discusses several such techniques in more detail [2]. For example, the characters in the string can be encoded in various ways, e.g., using %-encoding (a as %61, b as %62, ...), Unicode (a as \u0061, b as \u0062, ...), Base-64, etc. The string can be kept in encrypted, compressed, or permuted form. It can be constructed at runtime by concatenating other strings together. Besides, in addition to the traditional "dot notation" (obj.property) for object access, one can use a "bracket notation" (obj["property"]) instead. In the latter case, moreover, the use of a string as an array index makes applicable all of the string obfuscation techniques mentioned earlier.

The host environment of web browser also provides various options for obfuscation. One approach is to split code into several parts, either in the same file or even into multiple files stored among web servers. This technique is frequently seen with web-based malware. Another approach takes advantage of DOM interaction. For example, data can be stored in the HTML file, outside the <script> block, then retrieved using document.getElementById() at runtime. And, of course, document.write() is a more powerful weapon than eval(), which can be used in combination of those obfuscation techqniues mentioned above, to generate script, document elements storing data and pointers to external documents, all at runtime.

Figure 1 presents an example of JavaScript code obfuscated using some of the techniques discussed above. Line 3 of this code snippet uses bracket notation to reference object property, using the strings 'getElementById' (obtained as the concatenation of the strings 'get', 'Elem', 'ent', 'By', and 'Id') and 'innerHTML' (obtained as the concatenation of the strings 'i', 'nn', 'erH', and 'TML') as array indices instead of using the more straightforward dot-notation to obtain the corresponding property values. Furthermore, it uses the DOM method document.getElementById() to retrieve data, namely, the string 'HelloWorld.' For simplicity of exposition, this code uses a very straightforward obfuscation of the array index strings, namely, concatenation of a few smaller strings; the code could, however, just as easily have used arbitrarily more complex obfuscations to construct these strings. The script is equivalent to var c = document.getElementById('x').innerHTML. The value finally assigned to variable c is a string 'HelloWorld' which is retrived from the HTML element with ID 'x'.

Those obfuscation techniques can be combined in arbitrary

ways with multi-layered code unfolding, which makes it difficult to determine the intent of a JavaScript program from a static examination of the program text. To make it more challenging, the payload can be scattered in multiple code contexts at different levels, with each piece using various obfuscation techniques and hidden in the garbage code whose only purpose is to confuse deobfuscators. We will show in Section IV this trick can defeat existing JavaScript deobfuscators, which assume the unobfuscated, complete payload is revealed in one of the (typically the last) unfolded JavaScript code contexts.

There are also tools available for reducing the size of scripts [17], [18], usually by removing unnecessary whitespaces and comments, and renaming symbols. This technique is called code compression or minification, although it makes code difficult to read, the behavior is still apparent. Therefore, we don't consider code minification as obfuscation.

III. Semantics-Based Deobfuscation

In this section, we describe the concept of semantics-based deobfuscation and the architecture of our prototype system in more detail. In particular, we discuss how we collect execution information of JavaScript programs at runtime, and how we use dynamic slicing technique to identify "semantically relevant" code from obfuscated script.

A. Overview

Semantics-based approach. Deobfuscation refers to the process of simplifying a program to remove obfuscation code and produce a simpler and functionally equivalent program. In general, we cannot expect deobfuscation to produce the original source code for the program, either because the source code is unavailable, or due to code transformations applied during obfuscation. All we can require, then, is that the process of deobfuscation must be semantics-preserving: i.e., that the code resulting from deobfuscation be *equivalent* to the original program. For the analysis of potentially-malicious code, a reasonable notion of equivalence is that of *observational equivalence*, where two programs are considered equivalent if they behave—i.e., interact with their execution environment—in the same way. Since a program's runtime interactions with the external environment occur through system calls, this means that two programs are observationally equivalent if they execute identical sequences of system calls (together with the argument vectors to these calls).

This notion of equivalence suggests a simple approach to deobfuscation: identify code that directly or indirectly affects the values of the arguments to system calls; these instructions are "semantically relevant". Any remaining instructions, which are by definition semantically irrelevant, may be discarded. For the JavaScript code considered in this paper, the actual system calls are typically made from built-in browser routines that appear as native functions. Our implementation therefore uses native functions as a proxy for system calls: this is sound, but potentially conservative since not all native functions lead to system calls. Then, to identify instructions that affect the values of native function arguments, we use dynamic slicing, applied at the byte-code level. One of the advantages of doing analysis at byte-code level is that the JavaScript compiler does part of the job for us: many obfuscation techniques used to confuse human analysts or automated script parsers can be revealed or removed after compilation. Examples of such tricks are discussed in [8], [19].

System overview. Our approach to deobfuscating JavaScript code consists of the following steps, as shown in Figure 2:

1) Use an instrumented web browser to obtain an execution trace for the JavaScript code under analysis.
2) Construct a dynamic control flow graph from collected trace to determine the structure of the executed code.
3) Use our deobfuscation slicing algorithm to identify *semantically relevant instructions*, i.e., instructions that affect the externally-observable behavior of the program. As previously discussed, externally-observable behavior is carried out by native functions.
4) Decompile the dynamic control flow graph to an abstract syntax tree (AST) and label all the nodes constructed from resulting set of relevant instructions.
5) Use semantics-preserving transformations to eliminate goto statements. Finally, generate deobfuscated source code by traversing the AST and printing only labeled (relevant) syntax tree nodes.

Our current implementation seperates trace collection from the remaining steps: the generated trace is written out to a file, which is then read by the trace analyzer. This is purely for convenience, since it is conceptually straightforward to build the analysis facilities directly into the web browser. Our current implementation writes out the abstract syntax tree obtained at the end of the above process in the form of JavaScript source code, but one can also imagine directly applying other malware analysis tools to the syntax tree itself.

B. Trace Collection

We use an instrumented Mozilla FireFox web browser to collect the program's execution trace. FireFox first compiles JavaScript source code to bytecode and then executes it using its embeded SpiderMonkey interpreter. Since obfuscations commonly used by malware take advantage of the built-in functionality of JavaScript interpreter as well as document related operations provided by the browser (see Section II-C), our instrumentation covers both the interpreter and DOM.

Each byte-code instruction instance generated by our instrumented web browser includes instruction's address, opcode mnemonic, length (in bytes), and operands, together with any additional information about the instruction that may be relevant. In particular, we print the following information, which is used for subsequent steps of the deobfuscation:

- *function calls*: the reference to the callee (function object) and the number of arguments being passed, together with a flag indicating whether the callee is a native function;
- *global variables, array elements, and object property accesses*: which property of which object is being defined or used.
- *function references*: the reference to the function object in which this intruction belongs to.

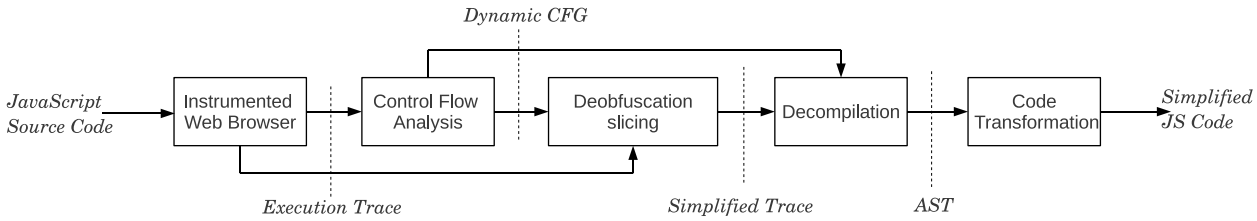

Fig. 2. Semantics-based JavaScript deobfuscation: System overview

– document.write *flags*: a flag indicating current instruction is a call to document.write() function;
– *document elements*: the reference to the document element that is created or accessed by functions such as getElementById();
– *unfolded code*: string passed to code generating operations (i.e. eval(), document.write(), etc.)

As discussed in Section II-B, the execution of a non-native function generates a bytecode trace while a call to a native function does not. However, the call to the non-native function can not be determined merely by the existence of bytecode trace. There are native functions take other functions as arguments, i.e. callbacks. These callback functions are invoked implicitly by the native function and generate bytecode trace, makes it similar to the execution of a non-native function. One such example is *string.*replace(), it takes a callback function as argument, the callback will be invoked after the match has been performed. The callback result (return value) will be used as the replacement string. Therefore, a flag is used to distinguish calls to native and non-native functions, and for each instruction instance in the trace, we use the function reference to indicate in which function this instruction belongs, in order to associate the execution and definition of callback functions.

The references to document elements and document.write flags are used to handle obfuscations involving DOM operations, which is opaque to JavaScript interpreter, but is crucial for the purpose of deobfuscating JavaScript in web pages: HTML document elements can be created and modified dynamically, and are often used for storing data by obfuscated JavaScript programs (e.g., see Figure 1). Unlike eval(), which is directly translated to an "eval" bytecode instruction, a call to document.write() is indistinguishable from other native function calls. The document.write flag is used by our deobfuscation slicing algorithm to establish the connection between HTML document and JavaScript code.

C. Control Flow Graph Construction

In principle, obtaining the static control flow graph (CFG) for a JavaScript program is possible. JavaScript source code is compiled into bytecode before execution, and it is straightforward to decompile this bytecode to an abstract syntax tree. In practice, the control flow graph so obtained may not be very useful if the intent is to simplify obfuscations away. The reason for this is that dynamic constructs such as eval(), commonly used to obfuscate JavaScript code, are essentially opaque in the static control flow graph: their runtime behavior—which is what we are really interested in—cannot be easily determined from an inspection of the static control flow graph. For this reason, we opt instead for a dynamic control flow graph, which is obtained from an execution trace of the program. However, while the dynamic control flow graph gives us more information about the runtime behavior of constructs such as eval(), it does so at the cost of reduced code coverage.

The algorithm for constructing a dynamic CFG from an execution trace is adapted from the algorithm for static CFG construction, found in standard compiler texts [20], modified to deal with dynamic execution traces, plus the standard dominator analysis to identify the loops. A more detailed discussion of issues about recovering the CFG from dynamic execution trace, such as handling different instances of the same instruction and the lack of information about functions, are given in our previous paper [15].

One particular challenge for JavaScript dynamic CFG construction is how to deal with code generated dynamically. In order to distinguish code used for generating other code at runtime, from code used for other computation, we treat dynamically unfolded code similar to the way we handle non-native functions: a seperate CFG is constructed for each piece of dynamically unfolded code, as the function body; and the unfolding instruction (e.g. eval()) is treated as a call to the function. This turns out to be conceptually simple and also reflects the way in which the eval() construct is handled in the underlying implementation.

D. Semantic-Based Deobfuscation

Given the execution trace and control flow graph, the next step is to identify the instructions that are semantically relevant to the program's externally observable behavior. For this, we use a variation on a program analysis technique known as dynamic slicing.

In general, dynamic slicing is the problem of identifying, for a given execution of a program P, which instructions (or statements) in P actually affect the value of a given variable at a given point in P. Intuitively, this involves tracing dependencies between uses and definitions of variables; the issue is somewhat more complicated in stack-based interpreters because the instructions that use the expression stack typically do not have their operands represented explicitly. Dynamic slicing in such situations has been investigated by Wang and

Input: A dynamic trace T; an instruction instance $instr \in T$; a dynamic control flow graph G;
Output: A slice S;

```
1  S := ∅;
2  currFrame := lastFrame := NULL;
3  LiveSet := ∅;
4  stack := a new empty stack;
5  I := instruction instance at the last position in T;
6  while true do
7      inSlice := false;
8      Uses := locations of data used by I;
9      Defs := locations of data defined by I;
10     if I is property access by bracket notation ∧ all instances
       of the corresponding instruction access the same name then
11         Uses := Uses - location of the string argument;
12     end
13     inSlice := I is instr;
14     if I is a return instruction then
15         push a new frame on stack;
16     else if I is an interpreted function call then
17         lastFrame := pop(stack);
18     else
19         lastFrame := NULL;
20     end
21     currFrame := top frame on stack;
22     if I is an interpreted function call ∧ I is not eval ∧ I is
       not code-unfolding document.write then
23         inSlice := inSlice ∨ lastFrame is not empty;
24     else if I is a control transfer instruction then
25         for each instruction J in currFrame s.t. J is
           control-dependent on I do
26             inSlice := true;
27             remove J from currFrame;
28         end
29     end
30     inSlice := inSlice ∨ (LiveSet ∩ Defs ≠ ∅);
31     LiveSet := LiveSet − Defs;
32     if inSlice then
33         add I into S;
34         add I into currFrame;
35         LiveSet := LiveSet ∪ Uses;
36     end
37     if I is not the first instruction instance in T then
38         I := previous instruction instance in T;
39     else
40         break;
41     end
42 end
```

Algorithm 1: Deobfuscation-Slicing algorithm

Roychoudhury in the context of slicing Java byte-code traces [21]. We adapt their algorithm in two ways, both having to do with the dynamic features of JavaScript used extensively for obfuscation. The first is that while Wang and Roychoudhury use a static control flow graph, we use the dynamic control flow graph discussed in Section III-C. The reason for this is that in our case a static control flow graph does not adequately capture the execution behavior of exactly those dynamic code unfolding constructs, such as eval() and document.write(), that we need to handle when dealing with obfuscated JavaScript. The second is in the treatment of dynamic constructs during slicing, such as code-unfolding and the bracket notation. Consider a statement eval(s): in the context of deobfuscation, we have to determine the behavior of the code obtained from the string s; the actual construction of the string s, however, is simply part of the obfuscation process and is not directly relevant for the purpose of understanding the functionality of the program. When slicing, therefore, we do not follow dependencies through any code unfolding statements.

A different approach is used to handle object property accesses using bracket notation. This access mechanism is useful in situations where different executions of the same piece of bracket-notation code access different object properties, e.g., when iterating over a list of properties in a loop. In such situations, the code used to construct the various strings used for the bracket-notation access is relevant to understanding the behavior of the program, and should be included in the slice. On the other hand, if every instance of the bracket-notation code accesses the same property, then this access mechanism provides no additional benefit compared to the more common dot-notation access; arguably, it makes the code a little harder to understand (especially if the string being used for the bracket-notation access is constructed dynamically), and so is obfuscatory. For this reason, given an instruction instance I in the trace that uses a bracket-notation access, we check whether all the instances of I in the trace access the same property name s: if so, the dependency from I to the code that constructs s is not followed; instead, the constructed name s is used in decompilation stage. Otherwise, if different instances of I use different property names s, the dependency from I to the code that constructs s is treated normally and the string construction code is included.

The key to any dynamic slicing based technique is to accurately capture the data and control flow of the target program, which is especially challenging for slicing JavaScript code in web pages: DOM enables the interaction between JavaScript code and the host HTML document, but the document related information is opaque to the JavaScript interpreter. For example, a program can store data in a HTML element and retrieve it later, similar to the usage of variables (e.g., see Figure 1). Furthermore, JavaScript program can also perform code unfolding at runtime using DOM methods (e.g. document.write()), either directly or from an external source. Therefore, to precisely slice JavaScript program, it is important to keep track of the connection between the byte-code used by interpreter and DOM operations. To this end, our slicing algorithm considers extra information provided by instrumented DOM, as discussed in Section III-B. More specifically, the reference to the document element is printed at the point where it is accessed by DOM methods, which recovers the data dependence hidden in the native functions.

In additon, we treat document.write() specially. document.write() method writes a string of text to the HTML document, which could be used both for basic document manipulation and code unfolding. Those two cases are handled differently, depends on whether new code is introduced and executed. If JavaScript code is dynamically generated and executed by calling this method, then our slicing algorithm handles the call exactly the same way as eval(): cut the dependence between document.write() and the code generated. Otherwise, we consider it a regular native function call which is used for output purpose.

We refer to this algorithm as deobfuscation-slicing. The

pseudocode is shown in Algorithm 1. Lines $1-5$ are initialization. The algorithm traverses the execution trace backwards, processing each instruction in order from the last instruction to the first. Lines $8-9$ extracts from the trace the set of memory locations on the stack that are read and/or written by the instruction, and similarly for properties and DOM elements. Lines $10-12$ cut the dependency for property access using bracket notation, as discussed above. If we encounter a return instruction, this instruction must be in a callee function, and since the trace is being traversed backwards we push a new frame on the stack (line 14); analogously, when we encounter a call to an interpreted function (native functions are not traced), we pop the stack because the call instruction is in the caller (line 16). The underlying implementation handles dynamic code generation via eval() and document.write() like a function call; line 22 of our algorithm ignores code unfolding, as discussed above. The handling of callback functions requires extra processing, the details of which are omitted due to space constraints; interested readers are referred to the full version of the paper, which is available online [22].

Input: A dynamic trace T; a dynamic control flow graph G;
Output: All relevant instructions R;

```
1  R := ∅;
2  U := ∅;
3  for i := length of T to 1 do
4  |  instr := i-th instruction instance in T;
5  |  if instr is a code unfolding instruction then
6  |  |  U := U ∪ Deobfuscation-Slicing(T, instr, G);
7  |  end
8  end
9  for i := length of T to 1 do
10 |  instr := i-th instruction instance in T;
11 |  if instr is a native function call ∧ instr ∉ U then
12 |  |  R := R ∪ Deobfuscation-Slicing(T, instr, G);
13 |  end
14 end
```

Algorithm 2: Semantics-based deobfuscation algorithm.

Deobfuscation-slicing solves only half of the puzzle; we still have to determine which instructions affect program's behavior, i.e. on which instructions to apply deobfuscation-slicing. Our approach of semantics-based deobfuscation consists of two basic steps, both steps rely on the deobfuscation-slicing algorithm. The pseudocode is shown in Algorithm 2:

1) *identify all instructions relevant to code unfolding (line 3-8).* First, the algorithm traverses the execution trace in order from the last instruction instance to the first, for each instance of dynamic code unfolding instructions in trace (e.g. eval() and call to document.write()), the deobfuscation-slicing algorithm is applied on it to identify instruction instances relevant to code unfolding, which include native function calls contribute to dynamic code generation. After this step, set U contains all the instructions in trace T that are relevant to code unfolding.
2) *identify all instructions relevant to observable behavior (line 9-14).* The algorithm traverses the trace backwards, applying the deobfuscation-slicing algorithm on each call to the native function which is irrelevant to code unfolding (those not in set U as identified in the first step). The resulting set R contains instructions semantically relevant to the observable behavior of the script.

E. Decompilation

The slicing step described above identifies instructions in the dynamic trace that directly or indirectly affect arguments to native function calls, which includes functions that invoke system calls. Instead of recomputing a control flow graph considering only those relevant instructions, we adopt a simpler approach for decompilation: transform the *original* control flow graph to the higher-level representation such as an abstract syntax tree (AST), and label those AST nodes constructed from relevant instrucions. This way, we avoid the complexity of handling protential problems caused by slicing, for example, basic blocks might be scattered and the branching target instruction might not in the slice.

A program in the byte-code representation of SpiderMonkey can not be directly converted into valid JavaScript source code, due to the existence of those low level branch instructions, e.g. ifne, goto, etc. Therefore, as the first step, we use goto statement to represent those branch operations in AST. Since the CFG has already been processed using loop analysis and function indentification, we need to construct an abstract syntax tree for each function. The basic blocks of the CFG are traversed in depth first order on the corresponding dominance tree, goto node is created in two cases: at the end of basic block that doesn't end with a brach instruction, or whenever a branch instruction is encountered. In addition to storing information of target block in goto nodes, we also keep track of a list of preceding goto nodes in each target node. Once every basic block has been translated to an AST node, loop structures are constructed by creating infinite while loop node which, initially, contains only the nodes of corresponding natural loop obtained from section III-C. Once we have an extened AST with goto nodes, additional code transformation is applied to generate valid JavaScript soure code. Basic block node and loop node in AST will be refered as *block node*.

F. Code Transformation

Introducing goto statments during decompilation allows us to apply a straightforward algorithm to construct AST, but JavaScript source code generated directly from this AST is invalid. To recover valid code, we need to transform the extended AST to eliminate goto statements, without changing the logic of the program.

Joelsson proposed a goto removal algorithm for decompilation of Java byte-code with irreducible CFGs, the algorithm traverses the AST over and over and applies a set of transformations whenever possible [23]. We adapt this algorithm to handle JavaScript and the instruction set used by the SpiderMonkey JavaScript engine [16]. The basic idea is to transform the program so that each goto is either replaced by some other construct, or the goto and its target are brought closer together in a semantics-preserving transformation. The transformation stops when none of the rules above can be applied to the AST. The resulting syntax tree is traversed one last time, for each node labeled by the decompiler described

```
function f(n){
  var t1=n;var t2=n;var k;
  var s4 = "eval('k=t1+t2;');";
  var s3 = "t1=f(t1-1);eval(s4);";
  var s2 = "t2=f(t2);eval(str3);";
  var s1 = "if(n<2){k=1;}\
       else{t2=t2-2;eval(s2);}";
  eval(s1);
  return k;
}
var x = 3;
var y = f(x);
alert(y);
```

(a) P_1

```
function fib(i){
  var k;var x = 1;var f1 = "fib(";
  var f2 = ")";var s1 = "i-";
  var s2 = "x";
  if(i<2)
    eval("k="+eval("s"+
        (x*2).toString()));
  else
    eval("k="+f1+s1+x.toString()+
        f2+"+"+f1+s1+(x*2).toString()+f2);
  return k;
}
var y = fib(3);
alert(y);
```

(b) P_2

Fig. 3. The test programs P_1 and P_2

in section III-E, corresponding source code has been printed out. Again, the detailed description of our transformation rules can be found in [15] .

G. Attacking our Algorithm

Intuitively, there are two ways by which an attacker might attempt to evade our approach to deobfuscation. The first is to hide relevant instructions, by adding fake dependency between them and strings to be unfolded. Our approach is immune to this technique, because an unfolded string s depends on some code v doesn't automatically exclude v from the resulting slice; if the real workload depends on v, then v would be added to slice regardless of the connection with code unfolding operation. In other words, only code which is solely used for obfuscation would be eliminated. The second evasion technique is to disguise the obfuscation code as relevant by adding extra irrelevant native function calls and creating dependencies between the obfuscation code and those irrelevant calls. Our semantics-based approach can not automically simplify away this kind of disguised obfuscation because, in general, the additional native function calls potentially change the observable behavior of the program. One approach to mitigating such attacks is to select, either manually or automatically, a (possibly proper) subset of the native function calls in the program that are used as the basis for the slicing process described above. A detailed discussion of this issue is beyond the scope of this paper.

IV. Experimental Evaluation

We evaluated the efficacy of our ideas using a prototype implementation based on Mozilla's open source FireFox web browser, which uses SpiderMonkey as its JavaScript engine. We tested this prototype on three synthetic programs as well as an actual JavaScript malware sample obtained from the Internet. First, we used two versions of the familiar Fibonacci program: this was chosen, first because it contains a variety of language constructs, including conditionals, recursive function calls, and arithmetic; and second because it is small and familiar, which makes it easy to assess the quality of deobfuscation. Our third synthetic test case is a very simple program that obfuscated so as to distribute its "payload" over multiple code contexts. This poses a problem for most existing

```
var cl=[168,183,176,165,182,171, ... … ,106,187,107,125];
var str='<script>';
for(ii=0;ii<cl.length;ii++)str+= String.fromCharCode(cl[ii]-66);
document["wr"+"ite"](str+"</script>");
```

(a)

```
eval(function(p,a,c,k,e,d){e=function(c){return
c};if(!''.replace(/^/,String)){while(c--){d[c]=k[c]||c}k=[function(e){return
d[e]}];e=function(){return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new
RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('17 8(9){ ... … 8(14);16(12);' ,
10,21, 'var||||t2| ... … |return|else'.split('|'),0,{}))
```

(b)

Fig. 4. Fragments of obfuscated versions of the program P_1

```
function f (arg0) {
  var local_var0,local_var1,local_var2;
  local_var0 = arg0;
  local_var1 = arg0;
  if(arg0<2)
    local_var2 = 1;
  else {
    local_var1 = local_var1-2;
    local_var1 = f(local_var1);
    local_var0 = f(local_var0-1);
    local_var2 =
       local_var0+local_var1;
  }
  return local_var2;
}
var x,y;
x = 3;
y = f(x);
alert(y);
```

(a)

```
function fib (arg0) {
  var local_var0,local_var1;
  local_var1=1;
  if(arg0<2)
    local_var0=local_var1;
  else
    local_var0=
      fib(arg0-1)+fib(arg0-2);
  return local_var0;
}
var y;
y=fib(3);
alert(y);
```

(b)

Fig. 5. Deobfuscator outputs for programs P_1 and P_2

JavaScript deobfuscators, which assume that the entirety of the payload is contained in one of the unfolded JavaScript code contexts. Finally, we tested our prototype using a sample of actual malicious code obtained from the Internet by using the 'wget' command to retrieve the contents of a URL extracted from a spam email sent to one of the authors.

Figure 3 shows two version of Fibonacci number computation programs. The first one, P_1 is shown in Figure 3(a), this program is hand-obfuscated to incorporate multiple nested levels of dynamic code generation using eval for each level of recursion. The second program, P_2, as shown in Figure 3(b), is also hand-obfuscated, in which we added dependency between real workload and the value used by eval (local variable x in function fib). Three versions of each of these programs are used—the program as-is as well as two obfuscated versions—one using an obfuscator we wrote ourselves that uses many of the obfuscation techniques described in Section II-C, including DOM operation; and an online obfuscator [24]. Figures 4 shows the fragments of obfuscated programs corresponding to P_1; the obfuscated code for P_2 are very similar and not shown separately due to space constraints.

The output of our deobfuscator for these programs is shown in Figure 5. Figure 5(a) shows the deobfuscated code for

```
var cl=[167,184,163, ..... ,191,107,107];var ii=0; var str=';';
for(ii=0;ii<cl.length;ii++) str+=String.fromCharCode(cl[ii]-66);
ii=0; eval(str); alert(b);
```

(a)

```
eval(function(p,a,c,k,e,d){e=function(c){return c.toString(36)}; b=0;
if(!''.replace(/^/,String)){while(c--){d[c.toString(a)]=k[c]||
c.toString(a)}k=[function(e){return d[e]}];e=function()
{return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)
+'\\b','g'),k[c])}}return p}('h f(6){0 8=6;0 4=6;0 7;0 9="5(\'7=8+4;\');";0 c="8=f(8-
1);5(9);";0 a="4=f(4);5(c);";0 b="e(6<2){7=1}\\g{4=4-2;5(a)}";5(b);d 7}0
i=f(3);',19,19,'var||||t2|eval|n|k|t1|s4|s2|s1|s3|return|if||else|function|
y'.split('|'),0,{}));eval(function(p,a,c,k,e,d){e=function(c){return
c.toString(36)};if(!''.replace(/^/,String)){while(c--){d[c.toString(a)]=k[c]||
c.toString(a)}k=[function(e){return d[e]}];e=function(){return'\\w+'};c=1}; ++b;
while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}
('e b(i){0 5;0 3=1;0 6="b(";0 a=")";0 9="i-";0
d="3";f(i<2)7("5="+7("c"+(3*2).8()));g 7("5="+6+9+3.8()+a+"+"+6+9+(3*2).8()
+a);h 5}0 j=b(4);',20,20,'var|||x||k|f1|eval|toString|s1|f2|fib|s|s2|function|if|else|
return||z'.split('|'),0,{}));
```

(b)

```
function f(n){var t1=n;var t2=n;var k;var s4="eval('k=t1+t2;');";var s3="t1=f(t1-
1);eval(s4);"; var s2="t2=f(t2);eval(s3);";var s1="if(n<2){k=1}\else{t2=t2-
2;eval(s2)}";eval(s1);return k}var y=f(3);
```

(c)

```
function fib(i){var k;var x=1;var f1="fib(";var f2=")";var s1="i-";var s2="x";
if(i<2) eval("k="+eval("s"+(x*2).toString()));else eval("k="+f1+s1+x.toString()
+f2+"+"+f1+s1+(x*2).toString()+f2);return k}var z=fib(4);
```

(d)

Fig. 6. Unfolded code contexts from obfuscated version of programs P_3. the original code of P_3 is highlighted. Some smaller code contexts are omitted.

all three versions of P_1 (the original code, shown in Figure 3(a), as well as the two obfuscated versions shown in Figure 4). Figure 5(b) shows the deobfuscated code for all three versions of P_2. For each of P_1 and P_2, the deobfuscator outputs are the same for all of the three versions. It can be seen that the recovered code is very close to the original, and expresses the same functionality. The results obtained show that the technique we have described is effective in simplifying away obfuscation code and extracing the underlying logic of obfuscated JavaScript code, which means it can handle server-side polymorphism regardless of syntactical difference of obfuscations. This holds even when the code is heavily obfuscated with multiple different kinds of obfuscations, including runtime decryption of strings and multiple levels of dynamic code generation and execution using eval() and document.write(), In particular, from simplified code of P_2 (Figure 5(b)), we could see that our approach handles those code intented to be "hidden" by eval correctly.

All obfuscations shown in Figure 3 and 4 are typical techniques widely used in the wild, they also satisfy the assumption made by current deobfuscators: the unobfuscated, complete payload is revealed in one of the unfolded JavaScript code contexts, i.e. if the deobfuscator simply examines every string passed to code unfolding operations such as eval() and document.write(), the unobfuscated payload can be directly identified in one of them. Our next test program, P_3, is purposely constructed to violate this assumption. For illustrative purpose, we make the original logic of P_3 very simple, which consists of only three statements:

```
function f0 (a0, a1, a2, a3, a4, a5) {
  b=0;
}
function f4 (a0, a1, a2, a3, a4, a5) {
  ++b;
}
f0(0,0,0,0,0,0)
f4(0,0,0,0,0,0)
alert(b);
```

Fig. 7. Deobfuscator outputs for programs P_3

```
laKKs='mCha';jJt='';n9gs="37G51G67G105G102G114G97G109G1
                    ...6 lines deleted...
7G105G102G114G97G109G101G37G51G69";ngs=document;n9gs=
n9gs["split"]('G');for(i=0;i<n9gs.length;i++)
jJt+=String['fro'+laKKs+'rCode'](n9gs[i]);
ngs["w"+"rite"](unescape(jJt));
```

Fig. 8. Source code for malware sample P_4

b=0; ++b; alert(b);

This code is manually obfuscated by hiding each statement into obfuscated variants of P_1 and P_2, in four steps. First, we remove the calls to native function alert() in P_1 and P_2 in Figure 3. Next, the modified P_1 and P_2 are obfuscated using the online obfuscator [24]. Then we insert first two statements of P_3 into these two obfuscated programs, as if a part of the obfuscation process, and concatenate them together. Last, we apply one more level of obfuscation to the code from last step, and attach the third statement of P_3 to its end. Figure 6 shows the unfolded code contexts of obfuscated P_3 as described above. Figure 6(a) is the topmost level obfuscation, statement alert(b) of P_3 resides in this context. Figure 6(b) is the context unfolded by the eval() in Figure 6(a). This context consists of obfuscated Fibonacci number programs, and the first two statements of P_3 are hidden in these two obfuscation processes consecutively. Figure 6(c) and (d) present the logic of Fibonacci number computation, from P_1 and P_2, both unfolded by context of Figure 6(b). Some of the smaller code contexts generated are not shown here.

Although P_3 is extremely simple, identifying its original logic from unfolded contexts in Figure 6 is still challenging: the original code is scattered among different code contexts at different obfuscation levels, hidden in garbage code; and it is easy to misidentify P_3 as Fibonacci computation. Therefore, as we can see, deobfuscators adopt the simple "context-unfolding" technique is very ineffective against the obfuscation which does't satisfy its assumption. In comparison, Figure 7 presents the output of our deobfuscator, in which most of the obfuscation and garbage code are removed, recovered code is very close to the original P_3, and expresses the same functionality. The extra code (function f0 and f4) is introduced because of the control dependency, which can be simplified away by further analysis, e.g. in this case, since none of the arguments is relevant, the invocation of the functions can be simply replaced by their body. We leave this for future work.

Finally, we evaluated our prototype system using a JavaScript malware sample, P_4, which we collected from the Internet as described earlier. The complete code contexts and deobfuscation output of P_4 is not presented here, due

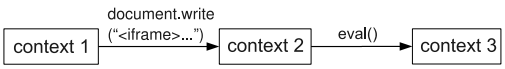

Fig. 9. Execution flow of malware sample P_4

```
lo cal_var0 = 0;
w hile (local_var0 < navigator.plugins.length) {
    local_var1= navigator.plugins[local_var0].name;
    if (local_var1.indexOf("Adobe Reader") != -1)
       document.write("<iframe src='./f3256c.pdf' width='1' height='1'
                          frameborder=0></iframe>");
    local_var0++; continue;
}
```

Fig. 10. A fragment of deobfuscator output for malware sample P_4

to the space constraint; Figure 8 shows the initial obfuscated JavaScript, while Figure 9 shows its high-level dynamic structure. Context 1 resides in the web page opened by web browser; it is a small piece of obfuscated JavaScript code (see Figure 8) that invokes document.write() method to dynamically insert a hidden iFrame, and cause the load of an external web page. This newly loaded web page contains more obfuscated code, which consists context 2 in Figure 9. Similarly, context 2 causes one more level of code unfolding using eval() and generates context 3. context 3 is the intended payload, it opens a PDF file that exploits a vulnerability in Adobe Reader, this action is also conducted using a dynamically created hidden iFrame. Figure 10 shows a fragment from the output of our deobfuscator. The full recovered code is very close to hand deobfuscated result and captures the essence of the malicious behavior of P_4. The call to document.write() in context 3 is part of the simplified code since it is used as a method for *output* as discussed in Section III-D. For complete result of P_4, please refer to the full version of this paper [22].

Performance

Thus far we have focused our efforts on implementing functionality in our deobfuscation tool instead of performance. Nevertheless, current performance seems acceptable. For all the test programs described earlier, the average overhead of trace collection is $2.5\mu s$ per instruction executed. Table I presents the performance for the deobfuscation process: with traces ranging from 582 instructions to 25,330 instructions, our tool takes an average of about $76\mu s$ per trace instruction. In particular, the trace for the our malware sample was $12,225$ instructions long and required $1.126s$ to analyze, which works out to about $92\mu s$/instruction.

V. Related Work

Most current approaches to dealing with obfuscated JavaScript typically require a significant amount of manual intervention, e.g., to modify the JavaScript code in specific ways or to monitor its execution within a debugger [6]–[8]. There are also approaches, such as Caffeine Monkey [11], intended to assist with analyzing obfuscated JavaScript code, by instrumenting JavaScript engine and logging the actual string passed to eval. Similar tools include several browser extensions, such as the JavaScript Deobfuscator extension for Firefox [12]. The disadvantage of such approaches is that they show all the code that is executed and do not separate out the code that pertains to the actual logic of the program from the code whose only purpose is to deal with obfuscation.

TABLE I
Running time of the deobfuscator for test programs

Test program	Length of trace ($instructions$)	Total time (μs)	Avg. time ($\mu s/instr.$)
P_1 obf1	6166	221949	35.9
P_1 obf2	582	40514	69.6
P_2 obf1	5755	155537	27.0
P_2 obf2	587	50209	85.5
P_3	25330	3793117	149.7
P_4	12225	1125874	92.1

Recently a few authors have begun looking at automatic analysis of obfuscated and/or malicious JavaScript code. Cova *at al.* [9] and Curtsinger *et al.* [4] describe the use of machine learning techniques based on a variety of dynamic execution features to classify Javascript code as malicious or benign. Such techniques typically do not focus on automatic deobfuscation, relying instead on the heuristics based on behavioral characteristics. Since obfuscation can also be found in benign code and really is simply an indicative of a desire to protect the code against casual inspection, classifiers that rely on obfuscation-oriented features are not reliable indicators of malicious intent. Our automatic deobfuscation approach can potentially increase the accuracy of such techniques by exposing the actual logic of the code. Saxena *et al.* discuss dynamic symbolic execution of JavaScript code using constraint-solving over strings [25]. Hallaraker and Vigna describe an approach to detecting malicious JavaScript code by monitoring the execution of the program and comparing the execution to a set of high-level policies [26]. All of these works are very different from the approach discussed in this paper.

There is a rich body of literature dealing with dynamically generated ("unpacked") code in the context of native-code malware executables [27]–[29]. Much of this work focuses on detecting unpacking and identifying the unpacked code. By contrast, the work described here is not concerned with the identification and extraction of dynamically-generated code *per se*, but focuses instead on identifying instructions that are relevant to the externally-observable behavior of the program.

VI. Discussion and Future Work

Our approach requires instrumenting the JavaScript interpreter within a web browser to write out a trace of the byte code being executed. Because this requires inserting code into the interpreter, our current prototype is implemented in the context of the open-source Firefox web browser. However, none of our ideas are Firefox-specific, and in principle they could be adapted in a straightforward way to any browser whose source code is available.

A potential concern with dynamic approaches, such as ours, is that of code coverage: in theory, static analyses can examine all of the code in a program while dynamic analyses can only examine the code that lies on a particular execution path. In practice, however, current static analyses for JavaScript are not

able to actually penetrate constructs such as eval() and analyze the code generated at runtime; rather, they rely on syntactic heuristics such as the presence of redirection, calls to eval(), and code/data entropy, to classify whether or not the code is potentially malicious. Examination of the simplified JavaScript code obtained from a tool such as ours can make it easier to identify inputs that would cause the code to execute alternative execution paths. We leave this to future work.

Another area of future research is the extension our approach to allow identification of attack code that does not have any other semantic significance, e.g.,code whose only purpose is to position heap buffers for a heap spray attack [30], [31]. Intuitively, what is going on is that our slicing algorithm starts with a notion of a set of "interesting" data values and identifies all of the other code that affects these values; the problem is that our current notion of "interesting values," limited to arguments to native function calls, is too narrow to capture such attack code. We intend to explore ways to address this issue by broadening the notion of "interesting" data values appropriately.

Finally, there are a few minor aspects of JavaScript and its DOM interactions that we have not yet had time to implement fully within our decompiler. For example, our JavaScript decompiler currently does not handle exception-handling via try-catch statements. This is straightforward implementation work and does not present significant conceptual challenges.

VII. Conclusions

The common use of JavaScript code for web-based malware delivery makes it important to be able analyze the behavior of JavaScript programs and, possibly, classify them as benign or malicious. For malicious JavaScript code, it is useful to have automated tools that can help identify the functionality of the code. However, such JavaScript code is usually highly obfuscated, and use dynamic language constructs that make program analysis difficult.

We present a semantics-based approach for automatic deobfuscation of JavaScript code. We use dynamic analysis and program slicing techniques to simplify away the obfuscation and expose the underlying logic of the JavaScript code in web pages. Moreover, this approach does not make any assumptions about the structure of the obfuscation. Experiments using a prototype implementation indicate that our technique is effective even against highly obfuscated JavaScript programs.

Acknowledgement

This work was supported in part by the National Science Foundation via grant nos. CNS-1016058 and CNS-1115829, and the Air Force Office of Scientific Research via grant no. FA9550-11-1-0191.

References

[1] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose, "All your iFRAMEs point to us," in *Proc. 17th USENIX Security Symposium*, 2008, pp. 1–15.

[2] F. Howard, "Malware with your mocha: Obfuscation and anti emulation tricks in malicious JavaScript," September 2010, http://www.sophos.com/security/technical-papers/malware_with_your_mocha.pdf.

[3] A. Kirk, "Gumblar and more on Javascript obfuscation," sourcefire Vulnerability Research Team. http://vrt-blog.snort.org/2009/05/gumblar-and-more-on-javascript.html. May 22, 2009.

[4] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Fast and precise in-browser JavaScript malware detection," in *USENIX Security Symposium*, 2011.

[5] S. Kaplan *et al.*, ""NOFUS: Automatically Detecting"+ String. fromCharCode (32)+ "ObFuSCateD ".toLowerCase()+ "JavaScript Code"," Microsoft Research, Tech. Rep., 2011.

[6] P. Markowski, "ISC's four methods of decoding JavaScript + 1," Mar. 2010, http://blog.vodun.org/2010/03/iscs-four-methods-of-decoding.html.

[7] J. Nazario, "Reverse engineering malicious JavaScript," CanSecWest 2007, http://cansecwest.com/csw07/csw07-nazario.pdf.

[8] D. Wesemann, "Advanced obfuscated JavaScript analysis," Apr. 2008, http://isc.sans.org/diary.html?storyid=4246.

[9] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: A fast filter for the large-scale detection of malicious web pages," in *Proc. 20th International Conference on World Wide Web*, 2011, pp. 197–206.

[10] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code," in *Proc. 19th International Conference on World Wide Web*, 2010, pp. 281–290.

[11] B. Feinstein and D. Peck, "Caffeine Monkey: Automated collection, detection and analysis of malicious javaScript," *Black Hat USA*, 2007.

[12] W. Palant, "FireFox add-on: JavaScript deobfuscator 1.5.7," https://addons.mozilla.org/en-US/firefox/addon/javascript-deobfuscator/.

[13] "jsunpack: A generic JavaScript unpacker," http://jsunpack.jeek.org/.

[14] B. Spasic, "Malzilla," http://malzilla.sourceforge.net/.

[15] G. Lu, K. Coogan, and S. Debray, "Automatic simplification of obfuscated JavaScript code (extended abstract)," in *Proc. ICISTM-12 Workshop on Program Protection and Reverse Engineering (PPREW)*, Mar. 2012, full version available at: http://www.cs.arizona.edu/~debray/Publications/js-deobf-full.pdf.

[16] Mozilla, "SpiderMonkey," https://developer.mozilla.org/en/SpiderMonkey.

[17] Google, "Minify," http://code.google.com/p/minify/.

[18] Yahoo!, "YUI compressor," http://developer.yahoo.com/yui/compressor/.

[19] B. Zdrnja, "Advanced JavaScript obfuscation (or why signature scanning is a failure)," Apr. 2009, http://isc.sans.edu/diary.html?storyid=6142.

[20] A. Aho, R. Sethi, and J. Ullman, *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, 1986.

[21] T. Wang and A. Roychoudhury, "Dynamic slicing on java bytecode traces," *ACM TOPLAS*, vol. 30, no. 2, p. 10, 2008.

[22] G. Lu and S. Debray, "Automatic simplification of obfuscated JavaScript code: A semantics-based approach," Tech. Rep., Jan. 2012, http://www.cs.arizona.edu/~genlu/pub/js-deobf-web.pdf.

[23] E. Joelsson, "Decompilation for visualization of code optimizations," Master's thesis, Royal Institute of Technology, 2003.

[24] "Online javascript obfuscator," http://www.daftlogic.com/projects-online-javascript-obfuscator.htm.

[25] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for JavaScript," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2010, pp. 513–528.

[26] O. Hallaraker and G. Vigna, "Detecting Malicious JavaScript Code in Mozilla," in *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, Shanghai, China, June 2005, pp. 85–94.

[27] L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware," in *Proc. ACSAC '07*, Dec. 2007.

[28] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *Proc. ACSAC '06*, 2006, pp. 289–300.

[29] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proc. Fifth ACM Workshop on Recurring Malcode (WORM 2007)*, Nov. 2007.

[30] M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with javascript," in *Proc. Second USENIX Workshop on Offensive Technologies (WOOT)*, 2008.

[31] A. Sotirov, "Heap feng shui in JavaScript," in *Black Hat USA 2007*, Jul. 2007, https://www.blackhat.com/presentations/bh-usa-07/Sotirov/Whitepaper/bh-usa-07-sotirov-WP.pdf.