Automatic Simplification of Obfuscated JavaScript Code*

Gen Lu, Kevin Coogan, and Saumya Debray

Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
{genlu, kpcoogan, debray}@cs.arizona.edu

Abstract. Javascript is a scripting language that is commonly used to create sophisticated interactive client-side web applications. It can also be used to carry out browser-based attacks on users. Malicious JavaScript code is usually highly obfuscated, making detection a challenge. This paper describes a simple approach to deobfuscation of JavaScript code based on dynamic analysis and slicing. 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.

1 Introduction

A few years ago, most malware was delivered via infected email attachments. As email filters and spam detectors have improved, however, this delivery mechanism has increasingly been replaced by web-based delivery mechanisms, e.g., where a victim is lured to view an infected web page from a browser, which then causes malicious payload to be downloaded and executed. Very often, such "drive-by downloads" rely on JavaScript code; to avoid detection, the scripts are usually highly obfuscated [7]. For example, the Gumblar worm, which in mid-2009 was considered to be the fastest-growing threat on the Internet, uses Javascript code that is dynamically generated and heavily obfuscated to avoid detection and identification [9].

Of course, the simple fact that a web page contains dynamically generated and/or obfuscated JavaScript code does not, in itself, make it malicious [4]; to establish that we have to figure out what the code does. Moreover, the functionality of a piece of code can generally be expressed in many different ways. For these reasons, simple syntactic rules (e.g., "*search for '*`eval(`*' and '*`unescape(`*' within 15 bytes of each other*" [9]) turn out to be of limited efficacy when dealing with obfuscated JavaScript. Current tools that process JavaScript typically rely

* This work was supported in part by the National Science Foundation via grant nos. CNS-1016058 and CNS-1115829, the Air Force Office of Scientific Research via grant no. FA9550-11-1-0191, and by a GAANN fellowship from the Department of Education award no. P200A070545.

on such syntactic heuristics and so tend to be quite imprecise: we found, for example, the obfuscated version of the fibonacci program discussed in Section 4 was identified as "infected" by an email system simply because it was obfuscated.

A better solution would be to use semantics-based techniques that focus on the behavior of the code. This is also important and useful for making it possible for human analysts to easily understand the inner workings of obfuscated JavaScript code so as to deal quickly and effectively with new web-based malware. Unfortunately, current techniques for behavioral analysis of 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 [10, 13, 17]. Recently, some authors have begun investigating automated approaches to dealing with obfuscated JavaScript. Cova *et al.* extract a set of features via runtime monitoring of the code, then apply machine learning techniques to classify these features as malicious or benign [3]. Saxena *et al.* use symbolic execution to reason about string operations in JavaScript programs [15].

This paper takes a different approach to reasoning about obfuscated JavaScript code: we use run-time monitoring to extract execution trace(s) from the obfuscated program, apply semantics-preserving code transformations to automatically simplify the trace, then reconstruct source code from the simplified trace. The program so obtained is observationally equivalent to the original program for the execution considered, but has the obfuscation simplified away, leaving only the core logic of the computation performed by the code. The resulting simplified code can then be examined either by humans or by other software. The removal of the obfuscation results in code that is easier to analyze and understand than the original obfuscated program. Experiments using a prototype implementation indicate that this approach is able to penetrate multiple layers of complex obfuscations and extract the core logic of the underlying computation.

2 Background

2.1 JavaScript

Despite the similarity in their names and their object-orientation, JavaScript is a very different language than Java. A JavaScript object consists of a series of name/value pairs, where the names are referred to as *properties*. Another significant difference is that while Java is statically typed and has strong type checking, JavaScript is dynamically typed. This means that a variable can take on values of different types at different points in a JavaScript program. JavaScript also makes it very convenient to extend the executing program. For example, one can "execute" a string s using the construct **eval(*s*)**. Since the string s can itself be constructed at runtime, this makes it possible for JavaScript code to be highly dynamic in nature.

There are some superficial similarities between the two languages at the implementation level as well: e.g., both typically use expression-stack-based

byte-code interpreters, and in both cases modern implementations of these interpreters come with JIT compilers. However, the language-level differences sketched above are reflected in low-level characteristics of the implementations as well. For example, Java's static typing means that the operand types of each operation in the program are known at compile time, allowing the compiler to generate type-specific instructions, e.g., **iadd** for integer addition, **dadd** for addition of double-precision values. In JavaScript, on the other hand, operand types are not statically available, which means that the byte code instructions are generic. Unlike Java, the code generated for JavaScript does not have an associated class file, which means that information about constants and strings is not readily available. Finally, JavaScript's **eval** construct requires runtime code generation: in the SpiderMonkey implementation of JavaScript [11], for example, this causes code for the string being **eval**ed to be generated into a newly-allocated memory region and then executed, after which the memory region is reclaimed.

The dynamic nature of Javascript code makes possible a variety of obfuscation techniques. Particularly challenging is the combination of the ability to execute a string using the **eval** construct, 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 [7]. 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. Moreover, these string obfuscation techniques can be combined in arbitrary ways, e.g., a string can be constructed by concatenating together a collection of strings that have been encoded in various ways; the string so constructed can be permuted or reversed to construct a second string; and this can then be decrypted to obtain the string that is actually used. Further, dynamic code generation via **eval** 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. Such obfuscation techniques can make it difficult to determine the intent of a JavaScript program from a static examination of the program text.

2.2 Semantics-Based Deobfuscation

Deobfuscation refers to the process of simplifying a program to remove obfuscation code and produce a functionally equivalent program that is simpler (or, at least, no more complex) than the original program relative to some appropriate complexity metric. To motivate our approach to deobfuscation, consider the semantic intuition behind any deobfuscation process. In general, when we simplify an obfuscated program we cannot hope to recover the code for the original program, either because the source code is simply not be available, or due to code transformations applied during compilation. All we can require, then, is that the process of deobfuscation must be semantics-preserving: i.e., that the code resulting from deobfuscation be semantically equivalent to the original program.

For the analysis of potentially-malicious code, a reasonable notion of semantic equivalence seems to be 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 are carried out 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 program equivalence suggests a simple approach to deobfuscation: identify all instructions that directly or indirectly affect 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 (examples of such semantically-irrelevant code include dead and unreachable code used by malware to change their byte-signatures in order to avoid detection). The crucial question then becomes that of identifying instructions that affect the values of system call arguments: for the JavaScript code considered in this paper, we use dynamic slicing, applied at the byte-code level, for this.

3 JavaScript Deobfuscation

3.1 Overview

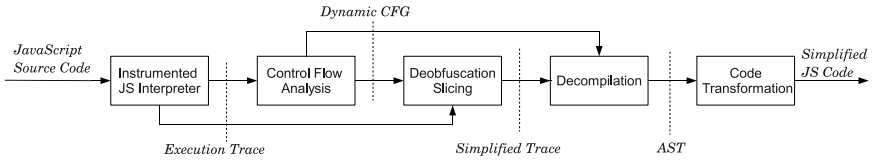

Fig. 1. Our approach to JavaScript deobfuscation: Overview

Our approach to deobfuscating JavaScript code is shown in Figure 1. It consists of the following sequence of steps:

1. Use an instrumented interpreter to obtain an execution trace for the JavaScript code under consideration.
2. Construct a control flow graph from this trace to determine the structure of the code that is executed.
3. Use our dynamic slicing algorithm to identify instructions that are relevant to the externally-observable behavior of the program. Ideally, we would like to compute slices for the arguments of the system calls made by the program. However, the actual system calls are typically made from external

library routines that appear as native methods. As a proxy for system calls, therefore, our implementation computes slices for the arguments passed to any native function.
4. Decompile excution trace to an abstract syntax tree, and label all the nodes constructed from resulting set of relevant instructions.
5. Apply transformation rules to eliminate **goto** statements. Then traverse AST to generate deobfuscated source code, by printing only labeled syntax tree nodes.

In our current implementation, the first step (trace collection) is separate from the remaining steps (trace analysis, decompilation): the generated trace is written out to a file, which is then read by the trace analyzer and decompiler. This is purely for convenience, since it is conceptually straightforward to build the trace collection and analysis facilities directly into the JavaScript engine, making it unnecessary to write/read a trace file. 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 could also imagine directly applying malware analysis tools to the syntax tree itself.

3.2 Instrumentation and Tracing

We instrument the JavaScript interpreter to collect a trace of the program's execution. Each byte-code instruction is instrumented to print out the instruction's address, operation mnemonic, and length (in bytes) together with any additional information about the instruction that may be relevant. In particular, we print the following information, which is used for control and data dependence analysis and to construct the control flow graph of the program:

- expression stack: set of memory locations on the stack that are read and/or written by the instruction;
- constants: encoded as part of the byte-code instruction, could be an integer or a reference to an object (i.e., string and floating point number), for the latter case, the actual value of constant is retrieved and printed instead of the reference;
- global variables: the name of the variable;
- local variables: an index specifying which variable amongst the function's locals is being referenced;
- function calls: the reference to the callee and the number of arguments being passed, and wether the callee is native function;
- conditional and unconditional branches: the offset (relative to the current instruction) of the branch target.
- global variables, array elements, and object property accesses: which property of which object is being defined or used. (This information is useful for the slicing step described next.)

The length information for instructions is used to distinguish calls to native methods from calls to interpreted methods: since native methods do not produce

a byte-code trace, a call instruction I at address A of length n bytes is a native-method call if the next instruction in the trace has address $A + n$, i.e., is the instruction that physically follows I in the program byte-code and therefore is the instruction that the call returns to. This proceeds as one would expect, except for one case: it turns out that tracing the code generated dynamically for the **eval** construct needs to be handled specially. There are two issues that arise in tracing dynamically generated code: code addresses and local variables. We discuss each of these below.

Dynamic code generation for an **eval**ed string typically allocates a buffer to hold the code. Code is generated into this buffer and executed, after which the buffer is deallocated. If a string is **eval**ed repeatedly at the same program point, e.g., over different loop iterations, the buffers allocated for the different **eval**s may be at different addresses. Tracing this code naively would give different addresses for the code generated into different such buffers. These would then be treated as different code fragments by later stages of the decompilation pipeline, leading to a great deal of repetition and redundancy in the high-level program representations recovered from the code. To deal with this, we maintain a table of **eval**ed strings indexed by the address of the **eval** construct. We then post-process the instructions in the trace collected from the execution and use this table to adjust the addresses of code resulting from dynamic code generation. This is done by making a pass over the instructions in the execution trace to find instances of code resulting from **eval**. Whenever such a code instance is found, we check to see whether the eval-string table mentioned above already contains the same executed string for the **eval** at that address. If this turns out to be the case, we adjust the addresses of the instructions within the **eval**ed code for that string by an offset that accounts for any difference between the addresses of the different buffers allocated. It is not difficult to see that this works as desired even if we have nested **eval**s, i.e., where a string being **eval**ed itself contains embedded **eval**s.

The second issue arises from the fact that in some JavaScript implementations, local variables may be represented differently in dynamically generated code than in statically generated code. For example, in Mozilla's SpiderMonkey JavaScript engine [11], in the code generated during the "regular" compilation process, each local variable in a function is represented using an index (a small integer), while the code resulting from dynamic compilation of **eval**ed strings uses the names of the variables regardless of whether they are local or global. It is necessary to resolve this discrepancy prior to program analysis, because otherwise the different representations for local variables may cause incorrect program slices and syntax tree to be computed. This means that when processing an instruction such as **setname**, whose operand is given as a name, the instrumentation code has to check to see whether the name resolves to a local variable, and retrieve and write out the corresponding index value if it does.

3.3 Control Flow Graph Construction

In principle, the (static) control flow graph for a JavaScript program can be obtained fairly easily. The byte-code for each function in a JavaScript program can be obtained as a property of that function object, and it is straightforward to decompile this byte-code 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 control flow graph from an execution trace is a straightforward adaptation of the algorithm for static control flow graph construction, found in standard compiler texts [2, 12], modified to deal with dynamic execution traces, plus the standard dominator analysis to identify the loops. The issue is that while an instruction at a particular memory address may occur many times in an execution trace, it should appear just once in a control flow graph. To this end, the first time an instruction I at memory address A is encountered, it is added to the control flow graph in the usual way. If instruction I at address A is subsequently encountered again, we do not create a new instance of that instruction in the control flow graph, but instead add a control flow edge to previously-created instance of I at A.

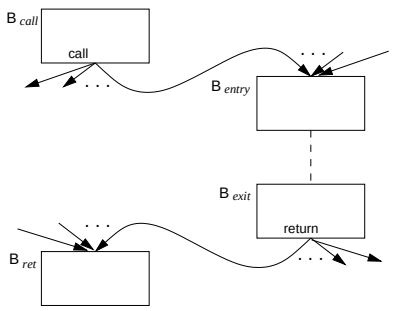

Fig. 2. Control flow graph structure for function calls/returns

Function Identification. The dynamic control flow graph obtained is missing some information necessary for transformation to a higher-level representation. In particular, the basic blocks in the program are not grouped into different

functions. In order to identify the basic blocks belonging to each function, we first associate each function call block with the corresponding block to which control returns from the call.

Figure 2 shows the structure of control flow edges for function calls and returns in the control flow graph constructed after slicing. Consider a basic block B_{call} ending in a **call** instruction. For each target for this call (in general there may be more than one) there is a control flow edge to the entry block B_{entry} of the callee. Suppose that control transfer from this callee back to the caller at the end of this call occurs from a basic block B_{exit} in the callee to a block B_{ret} in the caller, as shown in Figure 2. In general, a call may target more than one callee; a function may be called from more than one call site; and different calls to a function may return from different basic blocks within that function. Thus, B_{call} and B_{exit} may have multiple outgoing edges, and B_{entry} and B_{ret} may have multiple incoming edges. Our objective is to link together the call block B_{call} and the corresponding return block B_{ret} to which control returns at the end of the call. We proceed as follows:

1. Suppose that the call instruction at the end of B_{call} is at address A and is n bytes long. This means that the instruction that comes immediately after it in the program's byte-code, and to which control returns from the call, is at address $A + n$. We search the control flow graph to find the basic block B_{ret} whose first instruction is at address $A + n$.
2. We remove all the outgoing edges from B_{call} (however, we retain a list of the call targets for the call block) and the incoming edges into B_{ret}. We add an edge from B_{call} to B_{ret}; this edge indicates that the call from B_{call} returns to B_{ret}.

After all call blocks have been processed in this way, the control transfer edges out of each call block and the control transfer edges into the corresponding return block are replaced by a single edge from the call block to the return block. For each function, the basic blocks belonging to that function are then computed as the set of blocks that are reachable from the entry node for the function.

3.4 Deobfuscation Slicing

As mentioned in Section 2.2, we use dynamic slicing to identify instructions that directly or indirectly affect arguments passed to native functions, which has been investigated by Wang and Roychoudhury in the context of slicing Java byte-code traces [16]. We adapt the algorithm of Wang and Roychoudhury 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 3.3. 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 constructs, such as **eval**, that we need to handle when dealing with obfuscated JavaScript. The second is in the treatment of the **eval** construct during slicing. Consider a statement **eval(**s**)**:

Input: A dynamic trace T; a slicing criterion C; 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 := memory addresses and property set used by I;
9  |  Defs := memory addresses and property set defined by I;
10 |  inSlice := I ∈ C ;            /* add all instructions in C into S */
11 |  if i is a return instruction then
12 |  |  push a new frame on stack;
13 |  else if I is an interpreted function call then
14 |  |  lastFrame := pop(stack);
15 |  else
16 |  |  lastFrame = NULL;
17 |  end
18 |  currFrame := top frame on stack;
   |  // inter-function dependence: ignore dependency due to eval
19 |  if I is an interpreted function call ∧ I is not eval then
20 |  |  inSlice := inSlice ∨ lastFrame is not empty;
21 |  else if I is a control transfer instruction then
   |  |  // intra-function control dependency
22 |  |  for each instruction J in currFrame s.t. J is control-dependent on I do
23 |  |  |  inSlice := true;
24 |  |  |  remove J from currFrame;
25 |  |  end
26 |  end
27 |  inSlice := inSlice ∨ (LiveSet ∩ Defs ≠ ∅) ;          // data dependency
28 |  LiveSet := LiveSet − Defs;
29 |  if inSlice then                                  // add I into the slice
30 |  |  add I into S;
31 |  |  add I into currFrame;
32 |  |  LiveSet := LiveSet ∪ Uses;
33 |  end
34 |  if I is not the first instruction instance in T then
35 |  |  I := previous instruction instance in T;
36 |  else
37 |  |  break;
38 |  end
39 end
```

Algorithm 1: Deobfuscation slicing

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—for example, by decryption of some other string or concatenation of a collection of string fragments—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 **eval** statements. We have to note that because an **eval**ed 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 **eval**. In other words, only code which is solely used for obfuscation would be eliminated. Therefore, an obfuscator cannot simply insert **eval**s into the pragram's dataflow to hide relevant code. We refer to this algorithm as deobfuscation-slicing.

The pseudocode of deobfuscation-slicing 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. 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 11); 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 13). The underlying implementation handles dynamic code generation via **eval** like a function call; line 19 of our algorithm ignores **eval**, as discussed above.

3.5 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 programs 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 3.3. 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*.

3.6 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 [8]. We adapt this algorithm to handle JavaScript and the instruction set used by the SpiderMonkey JavaScript engine [11]. 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 fact that SpiderMonkey always generates byte-code with reducible CFGs (due to lack of an aggressive code optimization phase) and the difference between JavaScript byte-code and Java byte-code, makes it possible for our algorithm to have a smaller set of tranformation rules. But it would be straightforward to add more rules, if necessary, to handle highly optimized JavaScript byte-code with possibly irreducible CFGs.

The following transformation rules are applied by our algorithm on extended AST.

(a) **Move target block after preceding goto.** This transformation moves a block node to the location right after its preceding **goto** node. To apply this transformation, node m and its preceding **goto** node n must satisfy following conditions:
- n must be the only preceding **goto** node of m, and
- m is not already located after n, and
- m is neither a function entry nor a loop header node, and
- if m is being moved into a loop node o from outside, then there must be a **goto** node p immediately follows o, and a **goto** node q at the end of m, such that m and n have the same target node.

(b) **Convert goto to continue.** To convert a **goto** node n to **continue**,
- n and its target node m must resides in the same loop node o, and
- m must be the loop header of o.

When this transformation applied, n has to be removed from m's predeceding **goto** list.

(c) **Convert goto to break.** To convert a **goto** node n to **break**,

- n must resides in the a loop node o, and
- there must be a **goto** node p immediately follows o, and,
- n and p have the same target node.

When this transformation applied, n has to be removed from its target node's predeceding **goto** list.

(d) **Combine infinite loop and if-else.** This transformation, if applied, removes an **if-else** statement and moves the **goto** out the loop. To combine a **while** loop p and **if-else** node q, following conditions must be satisfied:
- p is an infinite loop, and
- q is the first statement in p, and
- the **else** branch of q only has a **goto** node n, and target of n is not part of loop p.

then, the p and q are combined, by substituting loop condition of p with predicate of q, substituting q with statements in the **if** branch of q, and moving n to the location immediately follows and outside of p. This rule also has a symmetric case, in which **if** and **else** is switched.

(e) **Move goto out of if-else.** This transformation could move **goto**s out of blocks and reduce the number of **goto** nodes as well. Given a **if-else** node p, in which the **if** branch ends with node m and **else** branch ends with node n, if both m and n are **goto** nodes and have the same target, then m is moved to the location right after p, and n is removed from AST as well as the preceding **goto** list of its target.

The transformation stops when none of the rules above could be applied to the AST. Then the syntax tree is traversed again, for each **goto** node n, we examine its target node t, if t is the node immediately following n, then n is removed from syntax tree. The resulting syntax tree is traversed one last time, for each node labeled by the decompiler described in section 3.6, corresponding source code has been printed out.

4 Experimental Results

We evaluated our ideas using a prototype implementation based on Mozilla's open source JavaScript engine SpiderMonkey [11]. Here we present results for two versions of Fibonacci number computation program. We chose them for two reasons: 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. The first of these, P_1, is shown in Figure 3(a); this program is was 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

```
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);
print(y);
```

(a) Program 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);
print(y);
```

(b) Program P_2

Fig. 3. The test programs P_1 and P_2

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 by Howard [7]; and an online obfuscator [1]. Figures 4 and 5 show the obfuscated programs corresponding to input programs P_1 and P_2 respectively.

The output of our deobfuscator for these programs is shown in Figure 6. Figure 6(a) shows the deobfuscated code for all three versions of program P_1 (the original code, shown in Figure 3(a), as well as the two obfuscated versions shown in Figure 4). Figure 6(b) shows the deobfuscated code for all three versions of the program P_2 (the original, shown in Figure 3(b), as well as the obfuscated versions shown in Figure 5). For both P_1 and P_2, the deobfuscator outputs are the same for each 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. 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, in particular, from simplified code of P_2 (Figure 6(b)), we could see that our approach handles those code intented to be "hidden" by **eval** correctly.

5 Related Work

We are not aware of much work on automatic simplification of obfuscated JavaScript. 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 [10, 13, 17]. There are also approaches, such as Caffeine Monkey [5], intended to assist with analyzing obfuscated JavaScript code, by instrumenting JavaScript

```
var cl=[168,183,176,165,182,171,177,176,98,168,171,164,106,176,107,189,184,163,180,98,182,
  115,127,176,125,184,163,180,98,182,116,127,176,125,184,163,180,98,173,125,184,163,180,
  98,181,182,180,118,98,127,98,100,167,184,163,174,106,105,173,127,182,115,109,182,116,125,
  105,107,125,100,125,184,163,180,98,181,182,180,117,98,127,98,100,182,115,127,168,171,164,
  106,182,115,111,115,107,125,167,184,163,174,106,181,182,180,118,107,125,100,125,184,163,
  180,98,181,182,180,116,98,127,98,100,182,116,127,168,171,164,106,182,116,107,125,167,184,
  163,174,106,181,182,180,117,107,125,100,125,184,163,180,98,181,182,180,115,98,127,98,100,
  171,168,106,176,126,116,107,189,173,127,115,125,191,167,174,181,167,189,182,116,127,182,
  116,111,116,125,167,184,163,174,106,181,182,180,116,107,125,191,100,125,75,167,184,163,174,
  106,181,182,180,115,107,125,75,180,167,182,183,180,176,98,173,125,191,184,163,180,98,186,
  98,127,98,117,125,184,163,180,98,187,98,127,98,168,171,164,106,186,107,125,178,180,171,
  176,182,106,187,107,125];
var ii=0;
var str=';';
for(ii=0;ii<cl.length;ii++){
  str+= String.fromCharCode(cl[ii]-66);
}
eval(str);
```

(a) Obfuscated code using our obfuscator.

```
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){0 6=9;0 4=9;0 7;0
  11="5(\'7=6+4;\');";0 10="6=8(6-1);5(11);";0 13="4=8(4);5(10);";0
  15="18(9<2){7=1;}20{4=4-2;5(13);}";5(15);19 7}0 14=3;0
  12=8(14);16(12);',10,21,'var||||t2|eval|t1|k|f|n|str3|str4|y|str2|x|str1
  |print|function|if|return|else'.split('|'),0,{}))
```

(b) Obfuscated code using online obfuscator.

Fig. 4. Obfuscated versions of the program P_1

engine and log the actual string passed to **eval**. Similar tools include several browser extensions, such as the JavaScript Deobfuscator extension for Firefox [14]. 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. Recently a few authors have begun looking at automating the analysis of obfuscated and/or malicious JavaScript code. Cova *at al.* describe Prophiler, which uses machine learning techniques to identify behavioral characteristics of malicious Javascript code [3]. The classification of Javascript code as malicious or benign is based on runtime characteristics consisting of both browser-based behavior, such as the number and target of web page redirections, browser personality and history-based differences, as well as JavaScript execution characteristics such as ratio of string definitions and string uses, number of dynamic code executions, number of bytes allocated through string operations, and length of dynamically evaluated code. Zozzle by Curtsinger *et al.* [4], identifies malicious JavaScript using a classifier based on context-sensitive features. They use known malicious and benign JavaScript to train a classifier, unlike [3], it must be run on unobfuscated scripts to reliably detect malicious code. While these techneques usually have a deobfuscation component, they simply rely on the assumption that the malicious workload has to be unpacked at last, and still heavily based on manual inspection for labeling training data. Besides, as with all classification problems,

```
var cl=[168,183,176,165,182,171,177,176,98,168,171,164,106,171,107,189,184,163,180,98,173,
  125,184,163,180,98,186,98,127,98,115,125,184,163,180,98,168,115,98,127,98,100,168,171,164,
  106,100,125,184,163,180,98,168,116,98,127,98,100,107,100,125,184,163,180,98,181,115,98,127,
  98,100,171,111,100,125,184,163,180,98,181,116,98,127,98,100,186,100,125,171,168,106,171,
  126,116,107,167,184,163,174,106,100,173,127,100,109,167,184,163,174,106,100,181,100,109,
  106,186,108,116,107,112,182,177,149,182,180,171,176,169,106,107,107,107,125,167,174,181,
  167,189,167,184,163,174,106,100,173,127,100,109,168,115,109,181,115,109,186,112,182,177,
  149,182,180,171,176,169,106,107,109,168,116,109,100,109,100,109,168,115,109,181,115,109,
  106,186,108,116,107,112,182,177,149,182,180,171,176,169,106,107,109,168,116,107,125,191,180,
  167,182,183,180,176,98,173,125,191,184,163,180,98,187,98,127,98,168,171,164,106,117,107,125,
  178,180,171,176,182,106,187,107,125];
var ii=0;
var str=';';
for(ii=0;ii<cl.length;ii++){
  str+= String.fromCharCode(cl[ii]-66);
}
eval(str);
```

(a) Obfuscated code using our obfuscator.

```
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};while(c--){if(k[c])
 {p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}
 ('f a(i){0 k;0 4=1;0 6="a(";0 8=")";0 9="i-";0 d="4";c(i<2)7("k="+7
 ("e"+(4*2).5()));g 7("k="+6+9+4.5()+8+"+"+6+9+(4*2).5()+8);h k}0
 b=a(3);j(b);',21,21,'var||||x|toString|f1|eval|f2|s1|fib|y|
 if|s2|s|function|else|return||print|'.split('|'),0,{}))
```

(b) Obfuscated code using online obfuscator.

Fig. 5. Obfuscated versions of the program P_2

```
function f (arg0) {
  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;
}
(x = 3);
(y = f(x));
print(y);
```

(a) Deobfuscated P_1

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

(a) Deobfuscated P_2

Fig. 6. Deobfuscator outputs for programs P_1 and P_2

features based on obfuscation techniques are not reliable indicators of malicious code, given the fact that obfuscation is also common in benign code. Our technique of automatic deobfuscation not only could reduce workload of manual analysis, it also could potentially increase the accuracy of such machine learning techniques. Saxena *et al.* discuss dynamic symbolic execution of JavaScript code using constraint-solving over strings [15]. 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 [6]. All of these works are very different from the approach discussed in this paper.

6 Conclusions

The prevalence of web-based malware delivery methods, and the common use of JavaScript code in infected web pages to download malicious code, 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. This paper describes an approach for dynamic analysis of JavaScript code to simplify away the obfuscation and expose the underlying logic of the code. Experiments using a prototype implementation indicate that our technique is effective even against highly obfuscated programs.

References

1. Online javascript obfuscator. http://www.daftlogic.com/projects-online-javascript-obfuscator.htm.
2. A.V. Aho, R. Sethi, and J.D. Ullman. Compilers: principles, techniques, and tools. *Reading, MA,*, 1986.
3. D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th international conference on World wide web*, pages 197–206. ACM, 2011.
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. B. Feinstein, D. Peck, and I. SecureWorks. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. *Black Hat USA*, 2007, 2007.
6. O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 85 – 94, june 2005.
7. F. Howard. Malware with your mocha: Obfuscation and antiemulation tricks in-malicious javascript, 2010.
8. E. Joelsson. Decompilation for visualization of code optimizations. 2003.
9. 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.

10. P. Markowski. ISC's four methods of decoding Javascript + 1, March 2010. http://blog.vodun.org/2010/03/iscs-four-methods-of-decoding.html.
11. Mozilla. Spidermonkey javascript engine. https://developer.mozilla.org/en/SpiderMonkey.
12. Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.
13. J. Nazario. Reverse engineering malicious Javascript. CanSecWest 2007, http://cansecwest.com/csw07/csw07-nazario.pdf.
14. W. Palant. Javascript deobfuscator 1.5.7. https://addons.mozilla.org/en-US/firefox/addon/javascript-deobfuscator/.
15. Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. *Security and Privacy, IEEE Symposium on*, 0:513–528, 2010.
16. T. Wang and A. Roychoudhury. Dynamic slicing on java bytecode traces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):10, 2008.
17. D. Wesemann. Advanced obfuscated javascript analysis, April 2008. http://isc.sans.org/diary.html?storyid=4246.