Automatic Simplification of Obfuscated JavaScript Code
(Extended Abstract)[*]

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 [8]. 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 [11].

Of course, the simple fact that a web page contains dynamically generated and/or obfuscated JavaScript code does not, in itself, make it malicious [5]; 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(***'*

[*] 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.

within 15 bytes of each other" [11]) turn out to be of limited efficacy when dealing with obfuscated JavaScript. Current tools that process JavaScript typically rely on such syntactic heuristics and so tend to be quite imprecise.

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 [13, 17, 22]. Recently, some authors have begun investigating automated approaches to dealing with obfuscated JavaScript, e.g., using machine learning techniques [3] or symbolic execution of string operations [20]; Section 5 discusses these in more detail. This paper takes a different approach to the problem: 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. Some of the details of this work have been omitted from this paper due to space constraints; interested readers are referred to the full version of the paper, which is available online [12].

In addition to obfuscated JavaScript code, web-based malware may also use other techniques, such as DOM interactions, to hamper analysis [8]. In such situations, simplification of obfuscated JavaScript code, while necessary, may not be sufficient to give a complete picture of what the malware is doing. This paper focuses on dealing with obfuscations involving dynamic constructs in JavaScript core language; additional issues, such as objects provided by DOM and the interactions between JavaScript and DOM, are beyond the scope of this paper and are considered to be future work.

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 [15], 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 [8]. 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

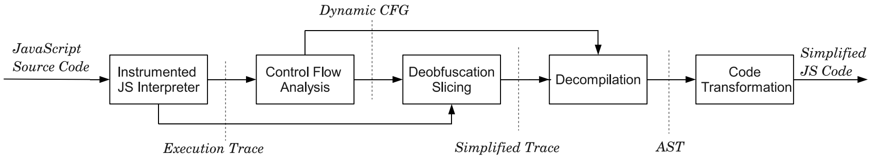
Deobfuscation refers to the process of simplifying a program to remove obfuscation code and produce a functionally equivalent program that is simpler (or, at least, no more complex) than the original program relative to some appropriate complexity metric. To motivate our approach to deobfuscation, consider 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. 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



Our approach to deobfuscating JavaScript code, shown above, consists of the following 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 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 (AST), and label all the nodes constructed from resulting set of relevant instructions.
5. Eliminate **goto** statements from the AST, then traverse it to generate deobfuscated source code by printing only labeled syntax tree nodes.

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, including expression stack usage, encoded constants, variable names/IDs, branch offets and object related data. Due to the space constraints, detailed description of the format and processing of the execution trace is not presented.

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, 16], modified to deal with dynamic execution traces.

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 [21].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**)**: 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

```
Input: A dynamic trace T; a slicing criterion $C$; a dynamic control flow graph
       $G$;
Output: A slice $S$;
$S := \emptyset$;
currFrame := lastFrame := NULL;
LiveSet := $\emptyset$;
stack := a new empty stack;
$I$ := instruction instance at the last position in T;
while true do
    inSlice := false;
    Uses := memory addresses and property set used by $I$;
    Defs := memory addresses and property set defined by $I$;
    inSlice := $I \in C$ ;                /* add all instructions in $C$ into $S$ */
    if $I$ is a return instruction then
        push a new frame on stack;
    else if $I$ is an interpreted function call then
        lastFrame := pop(stack);
    else
        lastFrame = NULL;
    end
    currFrame := top frame on stack;
    // inter-function dependence: ignore dependency due to eval
    if $I$ is an interpreted function call $\wedge$ $I$ is not eval then
        inSlice := inSlice $\vee$ lastFrame is not empty;
    else if $I$ is a control transfer instruction then
        // intra-function control dependency
        for each instruction $J$ in currFrame s.t. $J$ is control-dependent on $I$ do
            inSlice := true;
            remove $J$ from currFrame;
        end
    end
    inSlice := inSlice $\vee$ (LiveSet $\cap$ Defs $\neq \emptyset$) ;          // data dependency
    LiveSet := LiveSet $-$ Defs;
    if inSlice then                                  // add $I$ into the slice
        add $I$ into S;
        add $I$ into currFrame;
        LiveSet := LiveSet $\cup$ Uses;
    end
    if $I$ is not the first instruction instance in $T$ then
        $I$ := previous instruction instance in T;
    else
        break;
    end
end
```

Algorithm 1: d-slicing

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 is shown in Algorithm 1.

3.5 Decompilation and Code Transformation

The slicing step described in Section 3.4 identifies instructions in the dynamic trace that directly or indirectly affect arguments to native function calls, which includes functions that invoke system calls. This slice is used to transform the control flow graph of the program to an abstract syntax tree (AST) representation. We do this in two stages. In the first stage, we construct an AST that may sometimes contain explicit **goto** nodes that capture lower-level control flow. Such **goto** nodes are created in two situations: (*i*) at the end of basic blocks that do not end with a brach instruction, and (*ii*) when an explicit 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. Loops in the control flow graph are identified using dominator analysis [2] and represented in the AST as an indefinite loop structure of the form **while (1)** $\{\dots\}$, with branches out of the loop body represented using explicit **goto**s that reflect the control flow behavior of the low-level code. In the second stage, this AST is transformed using semantics-preserving **goto**-eliminating code transformations that generate valid JavaScript soure code, as described below.

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 [9]. We adapt this algorithm to handle JavaScript and the instruction set used by the SpiderMonkey JavaScript engine [15]. 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. Space constraints preclude a detailed description of our transformation rules; interested readers are referred to the full version of the paper [12]. The fact that SpiderMonkey always generates byte-code with reducible CFGs (due to the 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.

After this transformation step, 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 then traversed one last time and, for each node labeled by the decompiler described above, the corresponding source code is printed out.

```
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. 1. The test programs P_1 and P_2

4 Experimental Results

We evaluated our ideas using a prototype implementation based on Mozilla's open source JavaScript engine SpiderMonkey [15]. 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 (which is important given the space constraints of this paper) and familiar (which makes it easy to assess the quality of deobfuscation). The first of these, P_1, is shown in Figure 1(a); this program 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 1(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 by Howard [8]; and an online obfuscator [1]. Figures 2 and 3 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 4. Figure 4(a) shows the deobfuscated code for all three versions of program P_1 (the original code, shown in Figure 1(a), as well as the two obfuscated versions shown in Figure 2). Figure 4(b) shows the deobfuscated code for all three versions of the program P_2 (the original, shown in Figure 1(b), as well as the obfuscated versions

```
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. 2. Obfuscated versions of the program P_1

shown in Figure 3). 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 4(b)), we could see that our approach handles those code intented to be "hidden" by **eval** correctly.

5 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 [13, 17, 22]. There are also approaches, such as Caffeine Monkey [6], 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 [18]. The disadvantage of such approaches is that they show all the code that is executed and do not

```
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. 3. 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. 4. Deobfuscator outputs for programs P_1 and P_2

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 automatic analysis of obfuscated and/or malicious JavaScript code. Cova *at al.* [3] and Curtsinger *et al.* [5] 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. A problem with such approaches is that, given that obfuscation can also be found in benign code and really is simply an indicative of a desire to protect the code against casual inspection, classfiers that rely on obfuscation-oriented features may not be reliable indicators of malicious intent. Our technique of automatic deobfuscation can potentially increase the accuracy of such machine learning techniques by exposing the actual logic of the code. Saxena *et al.* discuss dynamic symbolic execution of JavaScript code using constraint-solving over strings [20]. 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 [7]. 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 conventional native-code malware executables [14, 19, 4, 10]. Much of this work focuses on detecting the fact of unpacking and identifying the unpacked code; because of the nature of the code involved, the techniques used are necessarily low-level, typically relying on detecting the execution of a previously-modified memory locations (or pages). 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.

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 to 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.* Addison-Wesley, Reading, Mass., 1985.
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. K. Coogan, S. Debray, T. Kaochar, and G. Townsend. Automatic static unpacking of malware binaries. In *Proc. 16th. IEEE Working Conference on Reverse Engineering*, pages 167–176, October 2009.
5. C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser JavaScript malware detection. In *USENIX Security Symposium*, 2011.
6. B. Feinstein, D. Peck, and I. SecureWorks. Caffeine monkey: Automated collection, detection and analysis of malicious JavaScript. *Black Hat USA*, 2007, 2007.
7. O. Hallaraker and G. Vigna. Detecting malicious JavaScript code in mozilla. In *Proc. 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 85–94, june 2005.
8. F. Howard. Malware with your mocha: Obfuscation and antiemulation tricks in-malicious JavaScript, 2010.
9. E. Joelsson. Decompilation for visualization of code optimizations. 2003.
10. 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)*, November 2007.
11. 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.
12. Gen Lu, Kevin Coogan, and Saumya Debray. Automatic simplification of obfuscated JavaScript code. Technical report, Dept. of Computer Science, The University of Arizona, October 2011. `http://www.cs.arizona.edu/~debray/Publications/js-deobf-full.pdf`.
13. P. Markowski. ISC's four methods of decoding Javascript + 1, March 2010. http://blog.vodun.org/2010/03/iscs-four-methods-of-decoding.html.
14. L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *Proc. 21st Annual Computer Security Applications Conference*, December 2007.
15. Mozilla. Spidermonkey JavaScript engine. https://developer.mozilla.org/en/SpiderMonkey.
16. Steven S. Muchnick. *Advanced compiler design and implementation.* 1997.
17. J. Nazario. Reverse engineering malicious Javascript. CanSecWest 2007, http://cansecwest.com/csw07/csw07-nazario.pdf.
18. W. Palant. JavaScript deobfuscator 1.5.7. https://addons.mozilla.org/en-US/firefox/addon/javascript-deobfuscator/.
19. P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 289–300, 2006.
20. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proc. IEEE Symposium on Security and Privacy*, pages 513–528, 2010.
21. T. Wang and A. Roychoudhury. Dynamic slicing on java bytecode traces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):10, 2008.
22. D. Wesemann. Advanced obfuscated JavaScript analysis, April 2008. http://isc.sans.org/diary.html?storyid=4246.