Saving the World Wide Web from Vulnerable JavaScript

Salvatore Guarnieri
IBM Watson Research Center
and University of Washington
sguarni@us.ibm.com

Marco Pistoia
IBM Watson Research Center
pistoia@us.ibm.com

Omer Tripp
IBM Software Group
and Tel Aviv University
omert@il.ibm.com

Julian Dolby
IBM Watson Research Center
dolby@us.ibm.com

Stephen Teilhet
IBM Software Group
steilhet@us.ibm.com

Ryan Berg
IBM Software Group
ryan.berg@us.ibm.com

ABSTRACT

JavaScript is the most popular client-side scripting language for Web applications. Exploitable JavaScript code exposes end users to integrity and confidentiality violations. Client-side vulnerabilities can cost an enterprise money and reputation, and cause serious damage to innocent users of the Web application. In spite of all this, recent research in the area of information-flow security has focused more on other languages that are more suitable for server-side programming, such as Java.

Static analysis of JavaScript code is very challenging due to the dynamic nature of the language. This paper presents ACTARUS, a novel, product-quality static taint analysis for JavaScript that scales to large programs and soundly models all the JavaScript constructs with the exception of reflective calls. This paper discusses the experimental results obtained by running ACTARUS on a collection of 9,726 Web pages obtained by crawling the 50 most visited Web sites worldwide as well as 19 other popular Web sites. The results expose 526 vulnerabilities in 11 sites. Those vulnerabilities, if exploited, can allow malicious JavaScript code execution.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Static Analysis, Security, Abstract Interpretation

Keywords

Information Flow, Abstract Interpretation, JavaScript

1. INTRODUCTION

In today's world of Rich Internet Applications (RIAs), mashups, and Asynchronous JavaScript And XML (AJAX) applications, the one common denominator is JavaScript—an object-oriented language that allows Web pages to be augmented with executable code. The use of JavaScript has become a standard among all Web developers due to its ease of use, flexibility and power. In fact, unlike Java applets, programs written in JavaScript do not need to be pre-compiled, do not require the installation of a plugin, can be tested quickly, and allow for easy programmatic manipulation of HTML Document Object Model (DOM) elements. For these reasons, JavaScript has effectively supplanted Java on the client side, to the point that 98 out of the 100 most visited Web sites use JavaScript for client-side programming according to Alexa.[1] As a consequence, JavaScript has gained the attention of researchers and developers worldwide.

The success of JavaScript has been fueled by the migration of application code from the server to the client. Older Web applications implemented very little logic on the client side. With the newer Web 2.0 style of applications, sources of tainted data can originate from a myriad of locations, such as Really Simple Syndication (RSS) feeds, Web services, local or remote databases and files, and AJAX callbacks. This is mainly due to the flexibility and power of the `XMLHttpRequest` object, on which all AJAX code is based. The downside of placing more code on the client is that this provides attackers with a much deeper view of the application's internals, and exposes a larger surface that attackers can exploit.

Since the inception of JavaScript, its security implications have been a concern. The first problem to address was to ensure that JavaScript programs obeyed the *same-origin policy*, which allows scripts originating from the same Web site to access each other's properties and functions, but prevents such access to scripts originating from different sites. Attackers, however, have been able to bypass the same-origin policy by injecting specially crafted malicious scripts into otherwise legitimate Web pages. Once injected, a malicious script can perform any number of exploits; essentially, attackers have the full power of JavaScript at their disposal. The consequences for the Web site can be very serious, and may include Web-site defacement, breakage of integrity and confidentiality, and complete identity theft. Thus, it has become imperative for enterprises to guarantee that their Web applications are secured against attacks not only on the server side, but on the client side as well.

In the past few years, several research directions in the area of JavaScript security have been pursued, including static analysis [6, 17], dynamic enforcement [36, 41], a combination of both static and dynamic analysis [36], and code rewriting [21]. Sound and scalable static analysis is an especially attractive solution for find-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA'11, July 17–21, 2011, Toronto, ON, Canada
Copyright 2011 ACM 978-1-4503-0562-4/11/05 ...$10.00

[1] `http://alexa.com`.

ing security vulnerabilities, particularly for developers, deployers, auditors and end users, who want to be reassured that the code they write, install, inspect and execute, respectively, is free of security problems. Unfortunately, many of the published approaches do not immediately apply to the security requirements of industrial Web applications because they are not sufficiently precise and have not been shown to scale to the size of production-level programs. Furthermore, static analysis is much more challenging for JavaScript than for more traditional languages, such as Java and .NET [35, 20]. Characteristics of JavaScript that contribute to this difficulty include prototype-chain property lookups, lexical-scoping rules for variable resolution, reflective property accesses, function pointers, and the fact that the properties and prototype chain of any object can be modified [27].

This paper presents ACTARUS, the first JavaScript static security analysis that addresses all the challenges listed above. Except for the fact that it does not model reflective calls (such as `eval`) and the `with` construct, ACTARUS is sound; it does not miss any of the security problems in the applications it analyzes. Furthermore, ACTARUS scales to large programs and automatically detects the following security vulnerabilities, which the Open Web Application Security Project (OWASP)[2] considers to be among the top ten security vulnerabilities in today's Web applications:

- *Injection*, the most common vulnerability: It occurs any time an attacker succeeds in sending untrusted data to an interpreter as part of a command or query, causing the interpreter to execute unintended commands.
- *Cross-Site Scripting* (XSS), the second most frequent vulnerability: If an attacker injects a malicious script into some text that is supposed to be displayed on other people's browsers, that script will not be *rendered* as regular plaintext, but *executed*. The consequences of this attack can be very serious. Its variant, DOM-based XSS, exploits the ability of JavaScript code to access the DOM. In DOM-based XSS, the attacker injects malicious code into the DOM causing that code to be executed on the victim's browser. DOM-based XSS completely circumvents the server side and bypasses any validation routine deployed on the server.
- *Unvalidated redirects and forwards*, the tenth most widespread form of attack: They exploit unchecked Uniform Resource Locators (URLs) to redirect the user to unintended Web sites, perform unauthorized AJAX requests, and connect to servers using ports or protocols that are different from the expected ones.

Each of these vulnerabilities can be cast as a problem in which tainted information from an untrusted *source* propagates, through data and/or control flow, to a high-integrity *sink* without being properly *endorsed* (*i.e.*, corrected or validated) by a *sanitizer*. ACTARUS is equipped with a rich set of "rules", each *rule* being a triplet of the form (Src, Snk, Snt), where Src, Snk and Snt are sets of related sources, sinks and sanitizers, respectively. ACTARUS statically verifies that there is no flow from any source to any sink that has not been intercepted by a sanitizer in the same rule.

1.1 Research Contributions

This paper makes the following contributions:

1. **A novel taint-analysis algorithm.** ACTARUS employs a novel combination of various static-analysis abstractions for modeling taint propagation. The result is a new taint-propagation algorithm that is infinitely context-sensitive up to recursion.
2. **Sound model of JavaScript language constructs.** Being targeted for industrial use, ACTARUS soundly models constructs of the JavaScript language that have been often eluded in previous work, such as prototype-chain property lookups and reflective property accesses.
3. **A complete set of rules.** Unlike other languages, where sources, sinks and sanitizers are just *methods* of objects of well-known types [35, 20], in JavaScript sources and sinks can also be *fields*. To complicate things, statically locating those members is difficult because the types of the objects holding them are not statically known. ACTARUS solves this problem by having its rules spell out the precise programmatic paths of retrieval that lead to objects containing those members, and by statically matching those access paths during the analysis.
4. **Implementation and evaluation.** We have implemented ACTARUS on top of the T. J. Watson Libraries for Analysis (WALA) open-source framework.[3] The core algorithm of ACTARUS is currently used in a commercial product.[4] In this paper, we present implementation details of ACTARUS and the results of ACTARUS on 9,726 Web pages obtained by crawling the top 50 most visited Web sites according to Alexa as well as 19 other popular sites, revealing 526 vulnerabilities on 11 sites. Those vulnerabilities can allow malicious execution of JavaScript code.
5. **Suite of microbenchmarks.** We have also implemented an exhaustive suite of microbenchmarks—analogous to the SecuriBench Micro suite[5] for Java taint analysis—that expose all the challenges that a static taint analysis for JavaScript should address. These include lexically scoped accesses, prototype lookups, callback functions, arguments array usage, string operations, interprocedural aliasing relations, sanitization, and DOM manipulation. ACTARUS successfully passes all these benchmarks. We are making the suite available to the scientific community to advance further research in this area.[6]

1.2 Paper Organization

This paper is organized as follows: Section 2 provides sample code illustrating the challenges lying in static taint analysis of JavaScript programs. Section 3 details the static-analysis framework employed by ACTARUS. Section 4 explains the ACTARUS core taint-analysis algorithm. Section 5 presents the experimental results obtained by analyzing 9,726 popular Web pages. Section 6 compares ACTARUS with previous work. Finally, Section 7 concludes this paper.

2. MOTIVATING EXAMPLES

To illustrate the security issues that ACTARUS can detect, we present two sample programs that demonstrate them. To make clear how the code works, we first introduce briefly the *framework* for client-side Web code; that is, how the code integrates into Web content and how it interacts with that content. After that, we present the sample programs themselves.

2.1 Web Programming Model

At the most basic level, code is included on Web pages using the `<SCRIPT>` tag; all text within such a tag is interpreted as executable code by the browser. In addition, the tag can have a `src` attribute, the value of which is a URL indicating a file of code to be loaded. In general, the tag can also have a `language` attribute

[2] `http://owasp.org`.

[3] `http://wala.sourceforge.net`.

[4] IBM Rational AppScan Source Edition, `http://ibm.com/software/rational/products/appscan/source/`.

[5] `http://suif.stanford.edu/~livshits/work/securibench-micro/`.

[6] `https://researcher.ibm.com/researcher/view_page.php?id=1598`.

denoting the language in which the code is written; however, the only widely supported such language is JavaScript, so we focus on that. Code included within <SCRIPT> tags is executed when the page is loaded; this typically involves defining functions for later use and initializing data structures. The functions defined are later invoked in two ways:

1. In *callbacks* for a range of HTML elements, such as `onclick` and `onmouseover`
2. In `href` elements that denote not a URL as usual, but rather code specified as "`javascript: ...`"

The code in such fragments and the script code within <SCRIPT> tags are all in the same scope, so the fragments can make use of, and change, the definitions in the script code.

The code on a Web page can interact with the page itself via the DOM, which is accessible as the variable `document`. The DOM provides Application Programming Interfaces (APIs) for finding, creating and changing DOM elements, so the code can manipulate and query the page in arbitrary ways. For instance, a common use of the APIs is to extract form values entered by the user. However, making large-scale changes to the page by altering HTML elements individually can be cumbersome, and so the document also provides two APIs for wholesale modification:

1. `innerText`, which can be used to set the content of a given DOM node to be the literal text provided.
2. `innerHTML`, which can be used to set the content of a given DOM node to be the DOM subtree obtained by parsing the text provided as HTML.

Both these APIs, when setting the content of a given DOM node, have also the effect of replacing any children the node might currently have.

2.2 Sample Programs

The code samples in this section bring together several of the most common and crucial hurdles faced by static taint-analysis engines for JavaScript. Designed for expository purposes, they illustrate the challenge of tracking reflective property accesses, aliasing relations, prototype-chain property lookups, lexical scoping rules for variable resolution and function pointers, while maintaining a high signal-to-noise ratio.

The HTML page of Figure 1 contains two `div` elements, with IDs `d1` and `d2`, respectively. The scenario depicted in this example assumes that the text in the `div` element with ID `d2` comes from an untrusted source. In fact, it appears that an attacker has tried to inject inside the `div` element with ID `d2` the JavaScript code shown in Figure 2. However, while forming the HTML page and sending it to the client, the server must have correctly sanitized that text with server-side encoding. We can infer this by noticing, for example, that characters < and > have been replaced with their HTML encodings `<` and `>`, respectively, thereby transforming the injected JavaScript code in the `div` element with ID `d2` into non-executable plaintext. The script inside the page, however, starts executing as the page is loading, and as this example is going to show, server-side sanitization may not be sufficient when code executes on the client side. At line 11, the script gets the `div` element with ID `d1` and assigns it to variable `el1`. It then calls function `foo` at line 23. Inside this function, the script gets the `div` element with ID `d2` at line 13 and assigns it to variable `el2`. Next, it calls function `bar` at line 20. Inside `bar`, the script constructs a new `Element` and assigns it to variable `el3` at line 15. At this point, `el1`, `el2` and `el3` are all in scope. At line 16, the value held by `el2.innerText`, which is a tainted source, gets sanitized with a call to `encodeURIComponent`. Therefore, the call to sink method `document.write` at line 17 is safe. Conversely,

```
1:  <html><head><title>Welcome!</title></head><body>
2:  <div id="d1"></div>
3:  <div id="d2">
4:    &lt;b onMouseOver='alert(1);'&gt;
5:        Attack Me&lt;/b&gt;
6:  </div>
7:  <script>
8:    function Element() {
9:      this.innerText = "http://somesite.com";
10:   }
11:   var el1 = document.getElementById("d1");
12:   function foo() {
13:     var el2 = document.getElementById("d2");
14:     function bar() {
15:       var el3 = new Element();
16:       var s = encodeURIComponent(el2.innerText);
17:       document.write(s); // sanitized
18:       el1.innerHTML = el2.innerText; // violation
19:       document.location = el3.innerText; // benign
20:     }
21:     bar();
22:   }
23:   foo();
24:   function baz(a, b) {
25:     a.f = document.URL;
26:     document.write(b.f); // violation
27:   }
28:   var x = new Object();
29:   baz(x, x);
30: </script>
31: <h1>Welcome to our system</h1></body></html>
```

Figure 1: `example_01.html` Program

```
<b onMouseOver='alert(1);'>Attack Me</b>
```

Figure 2: Code Injected in HTML Page of Figure 1

the assignment of tainted source `el2.innerText` to sink field `el1.innerHTML` at line 18 is unsafe because no sanitization has taken place. The problem here is that `el1.innerHTML` gets assigned the value held by `el2.innerText`, which is the text displayed by the browser in the `div` element with ID `d2`. This is exactly the text shown in Figure 2. Therefore, the malicious JavaScript code that the server had initially transformed into plaintext is now reinterpreted as JavaScript executable code because it is being assigned to the `innerHTML` field of an HTML element.

Interestingly, the assignment of `el3.innerText` to sink field `document.location` at line 19 is safe because `el3` is an object implementing function `Element`, defined at lines 8-10. Such function, coincidentally, contains an `innerText` field, but this is not a tainted source since the object that holds it, unlike `el1` and `el2`, is not an HTML element. This example has demonstrated that the challenges that an analysis must face when scanning JavaScript applications include resolving lexically scoped variables, accounting for sanitization, and recognizing the programmatic paths of retrieval leading to security-sensitive fields of actual HTML elements, while disambiguating such fields from identically named fields in other functions.

A separate vulnerability exposed by the code in Figure 1 appears at line 26, where a call to sink method `document.write` is performed with parameter `b.f`. At line 25, the value of source field `document.URL` is assigned to `a.f`. Static analyzers that do not account for inter-procedural alias relations may fail to recognize that `a.f` and `b.f` actually point to the same object via the `baz(x, x)` call at line 29 and, as a consequence, the DOM-based XSS vulnerability at lines 25-26 may remain undetected. Though somewhat contrived, the code sample of Figure 1 exposes salient features of modern JavaScript Web applications. In particular, JavaScript client code often gives rise to non-trivial aliasing relations, which require complex inter-procedural alias analysis to be resolved.

```
1:    <html><head><title>Welcome!</title></head><body>
2:    <script>
3:      function A() {
4:        this.f = new Object();
5:      }
6:      var a = new A();
7:      a.f = document.URL;
8:      function B() { }
9:      B.prototype = a;
10:     var b = new B();
11:     var doc = document;
12:     var w = "wr" + "ite";
13:     doc[w](b.f); // violation
14:   </script>
15:   <h1>Welcome to our system</h1></body></html>
```

Figure 3: `example_02.html` Program

The HTML code of Figure 3 shows a different set of challenges, which include modeling prototype-chain property lookups and reflective property accesses that involves string manipulations. At lines 3-5, function `A` is defined. Object `a` is created at line 6. At line 8, function `B` is defined, and object `b` is instantiated al line 9 with a call to the constructor of `B`. Such an object does not have a field `f` explicitly declared. Therefore, referencing `b.f` as at line 13 causes a prototype-chain property lookup, which gets resolved as `a.f` since `a` was declared to be `B`'s prototype at line 9. Object `a` does have a field `f`, which points to the tainted value `document.URL` as per the instruction at line 7. A static security analysis, at this point, should detect that the instruction at line 13 is a vulnerability because variable `doc` is an alias of `document` as per line 11, and reflective property access `doc[w]` gets resolved into a reference to sink method `document.write` as per line 12. Resolving this property access requires also tracking string constants and modeling string concatenations.

Sound and precise analysis of such codes should not only consider all these challenges, but also be sensitive to security concerns. This includes awareness of the type of sanitization performed on the input *vis-a-vis* the sinks using it. A solid solution to the challenges in these motivating examples is key for effective analysis of real-world JavaScript Web applications.

3. CALL GRAPH / POINTER ANALYSIS

Before performing taint analysis on a program, ACTARUS builds a static representation of the program, consisting of a call graph and a pointer analysis [15]. This section describes how the foundation of this static-analysis infrastructure and how the basic challenges were overcome. Section 4 will explain how ACTARUS lifts this infrastructure to perform an effective taint analysis of JavaScript Web applications.

ACTARUS is based on the assumption that taint analysis is a demand-driven problem. Ideally, the analysis could simply track flows originating from tainted sources, and there would not be a need for a complete call graph, but in order to identify statically the sources reachable from the program entry points and the interprocedural alias relations between tainted variables, a complete call graph and its associated pointer analysis are needed.

Since JavaScript uses function pointers and lacks a class hierarchy, inexpensive call-graph-construction algorithms based on Class Hierarchy Analysis (CHA) [8] or Rapid Type Analysis (RTA) [3] are not viable solutions. Therefore, ACTARUS uses Andersen's analysis [1] to combine pointer-analysis and call-graph construction and to abstract run-time objects into allocation sites. The very dynamic semantics of JavaScript, however, requires some adaptation of standard techniques. Some aspects of JavaScript, such as first-class functions, are handled very naturally; indeed, much of the pioneering work on this combined analysis was done for Scheme [32]. Nevertheless, some aspects of JavaScript are hard to model directly in a static way, and require some special treatment, as explained in the remainder of this section.

3.1 Prototypes

JavaScript uses *prototypes* rather than object-oriented inheritance. In this mechanism, an object will traverse its prototype pointer to look for properties that it itself does not have. We model this explicitly in the Intermediate Representation (IR) for JavaScript. More specifically, each such property access, `e.g., b = a.f` gets translated through an Abstract Syntax Tree (AST) rewrite into a loop using non-standard prototype notation that ACTARUS understands, before generating the actual IR, as shown in Figure 4.

```
1:    var x = a;
2:    do {
3:      b = x.f;
4:      x = x.__proto__;
5:    } while (!defined(b));
```

Figure 4: Encoding of Prototype-chain Property Lookup

On the other hand, assignments of properties never traverse the prototype pointers; if a value is assigned to a property that does not exist, a new property is created, and no prototype-chain property lookup is performed. Thus, assignment of properties are modeled directly as property-write IR nodes.

3.2 Object Creations

Although a `new` instruction in JavaScript looks syntactically like a `new` in C++ or Java, its semantics is very dynamic, since the argument is a variable rather than a constant. Therefore, the effect of `new X` can depend upon what value `X` takes on during execution. For example, Figure 5 shows code that creates different kinds of objects depending on a given condition `b`.

```
1:    if (b) {
2:      X = Object;
3:    } else {
4:      X = Array;
5:    }
6:    y = new X(7);
```

Figure 5: Dynamic Semantics of `new` Instructions

To handle such dynamism, we model `new` as a special function property on objects, and so a `new` expression becomes a first-class function call on its argument.

Furthermore, even for a specific object, the object-allocation semantics of JavaScript can be very dynamic. For instance, the meaning of the allocation of an `Array` depends upon how many and what types of arguments are provided. If there is just one argument of `Number` type, then an array of the specified size is created; in all other cases, the arguments are the initial contents of the array.

Thus, we rely on custom dispatch to direct `new` expressions to stub methods that implement the right semantics based on the target object, and the number and types of arguments. Since this is a static analysis, the system may end up conservatively adding calls to multiple methods in cases like the one in Figure 5, where the target is ambiguous.

One complication of this approach is that now, any actual object-creation is a function-call away from the syntactic creation site in the original code. Thus, in order to get per-creation-site object names, we use a level of calling-context sensitivity (1-CFA [15]) when analyzing these special constructor functions, and a level of call strings for object names (1-1-CFA [15]).

3.3 Reflective Property Accesses

An issue that occurs in other languages as well, but is very prominent in JavaScript, is the use of *first-class property names*, which allow for expressions of the form `a[b] = c` and `c = a[b]`, where `b` is a variable. In many idioms, `b` refers to either a single or a small number of string constants. In ACTARUS, we use a precise model of string constants in which a different abstract object is allocated for each specific constant. We also model string concatenation (the + operator at the source level) by creating new string constants when appropriate.

3.4 Lexical Scoping

JavaScript features *lexical scoping*, allowing variables from a function to be accessed and modified by functions declared inside it. For instance, in the code of Figure 1, `bar` refers to variable `el2`, which was declared in `foo`.

ACTARUS represents the variable of a program in Static Single Assignment (SSA) form [7]. SSA takes a program and transforms it into an intermediate representation in which every single variable is assigned only once. This is achieved by taking the existing variables in the program and splitting them into *versions*. In order to preserve SSA form for lexically scoped variables, ACTARUS converts them to SSA form as usual, but dependency information is recorded so that the SSA representation can be updated if actual assignments occur in called functions. We are going to see this in an example.

```
1:      function foo(a) {
2:        var x = a;
3:        function bar() {
4:          var y = x + 5;
5:          function baz() {
6:            x = y + 1;
7:          }
8:          baz();
9:          return x;
10:       }
11:       var t = bar();
12:       return x + t;
13:     }
14:     var z = foo(3);
```

Figure 6: Program Performing Lexically Scoped Accesses

Figure 6 contains a program snippet that makes use of lexically scoped accesses to variables `x` and `y`, which are declared in `foo` and `bar`, respectively; `x` is read in `bar` and written in `baz`, while variable `y` is read in `baz`. These lexical accesses are shown in the code of Figure 7 with `LexicalRead` and `LexicalWrite` statements, with each variable, in SSA form, labeled by its name and defining function. During call-graph construction, when the call to `bar` is encountered, the analysis records that the current value of `x` is `x_1`, and the `LexicalRead` in `bar` is connected to that value. Similarly, the call to `baz` links its `LexicalRead` to `y_1` and its `LexicalWrite` to `x_3`. When the `LexicalWrite` is encountered, callers in the call stack get their SSA form updated to indicate that there is a new definition of `x`, which generates `x_3` in `baz` and `x_4` in `bar`.

4. TAINT ANALYSIS

Recall that taint analysis comprises searching for flows of data from untrusted points of input (the *sources*) to sensitive consumers (the *sinks*). These flows are potential security issues unless each flows passes through an operation that renders the data safe (a *sanitizer*).

Given a call graph G, constructed as discussed in Section 3, ACTARUS' taint-analysis algorithm comprises of two stages:

```
1:      function foo(a) {
2:        var x_1 = a;
3:        function bar() {
4:          x_2 = LexicalRead(x, foo);
5:          var y_1 = x_2 + 5;
6:          function baz() {
7:            y_2 = LexicalRead(y, bar);
8:            LexicalWrite(x, foo, y_2 + 1);
9:          }
10:         baz() [reads:  y_1, writes:  x_3];
11:         return x_3;
12:       }
13:       var t_1 = bar() [reads:  x_1, writes:  x_4];
14:       return x_4 + t_1;
15:     }
16:     var z_1 = foo(3);
```

Figure 7: Code of Figure 6 with Lexical-scoping Annotations

1. G is traversed to find sources, sinks and sanitizers in the code:
   - *Sources* are either values obtained by reading certain fields or values returned from calls to certain methods.
   - *Sinks* can be either fields of certain objects or parameters of given methods.
   - *Sanitizers* are methods that transform the data they receive as input into innocuous data that can be safely passed to sinks without causing a security vulnerability.

   Sources, sinks and sanitizers are partitioned into *rules* based on the vulnerabilities they relate to. ACTARUS comes with a standard set of rules. Custom rules (containing, for example, user-defined sanitizers) could be easily added to the default specification if necessary.
2. An inter-procedural data-flow analysis is performed starting at the sources to determine if there are tainted flows that reach sinks without having been intercepted by sanitizers. The analysis is seeded at the variables defined by source constructs.

As for Step 1 above, it should be observed that in JavaScript there are no specific types. While in Java, for example, it is possible to specify that the return value of any invocation of `getParameter` on any object of *type* `HttpServletRequest` is a source, the absence of specific types in JavaScript makes this sort of security configuration impossible. To address this problem, the rules in ACTARUS specify the complete path of retrieval for sources, sinks and sanitizers. For example, a rule specifies that field `innerHTML` in any object returned by a call to `document.getElementById` is a sink, since writes to that field get parsed as HTML and placed in the page. Thus, ACTARUS locates all the aliases of global object `document`, detects all the calls to `getElementById` on those aliases, and collects all the objects returned by such calls. It then looks for all field-write instructions on the `innerHTML` field of those objects. The variables used in those field-write instructions are marked as sinks by the analysis. A similar procedure is followed to locate other sinks, as well as sources and sanitizers.

Another characteristic of JavaScript that makes JavaScript code more vulnerable to attacks than code written in other languages is that in JavaScript, variables can point to functions. This feature allows variables pointing to security-sensitive functions (including sanitizers and sink methods) to be reassigned. If a variable pointing to a sanitizer gets assigned a different value, untrusted input may no longer be sanitized as intended. A novel security contribution of ACTARUS is its automatic detection of field-write instructions that can lead to assigning new values to variables pointing to such security-sensitive functions.

4.1 Tracking Taint via Access Paths

To uncover vulnerable information-flow paths, ACTARUS maintains the set of all heap locations that store untrusted values. A

naïve way of doing this is to explicitly model the entire heap, including all benign locations, and then track—at each point during the analysis—which portions of the heap are tainted. This solution is, in general, prohibitively expensive, and thus also unscalable, as demonstrated, *e.g.*, in [35]. Our solution uses a *storeless* view of the heap [11], which—instead of representing the heap explicitly—tracks only information relevant for taint propagation; namely, which sequences of local variable and field dereferences may lead to untrusted data.

We assume a standard concrete semantics where a program state and an evaluation of an expression in a program state are defined. The following semantic domains are used:

$$\begin{array}{ll} L & \in \mathit{objects} \\ v & \in \mathit{Val} = \mathit{objects} \cup \{\mathit{null}\} \\ \rho & \in \mathit{Env} = \mathit{VarId} \rightarrow \mathit{Val} \\ h & \in \mathit{Heap} = \mathit{objects} \times \mathit{FieldId} \rightarrow \mathit{Val} \\ \sigma = \langle L, \rho, h \rangle & \in \mathit{States} = 2^{\mathit{objects}} \times \mathit{Env} \times \mathit{Heap} \end{array}$$

where *objects* is an unbounded set of dynamically allocated objects, and *VarId* and *FieldId* are sets of local variables and field identifiers, respectively. A *program state* σ thus maintains the set L of allocated objects, an environment ρ mapping local variables to values, and a mapping h from fields of allocated objects to values.

As motivated above, the data-flow analysis carried out by ACTARUS is based on the notion of an "access path" [12]. Formally, an *access path* is a pair, $\langle \mathtt{v}, \langle \mathtt{f1}, \ldots, \mathtt{fn} \rangle \rangle$, where v is a local variable, and $\mathtt{f1}, \ldots, \mathtt{fn}$ are field identifiers. The evaluation of access path $\langle \mathtt{v}, \langle \mathtt{f1}, \ldots, \mathtt{fn} \rangle \rangle$ in a concrete state σ with an environment ρ and a heap h yields the unique heap-allocated object o satisfying the following condition:

$$\exists o_1, \ldots, o_n . o_1 = \rho(\mathtt{v}) \wedge o_2 = h(o_1, \mathtt{f1}) \wedge \ldots \wedge o = h(o_n, \mathtt{fn})$$

such that $o, o_1, \ldots, o_n \in L$, where L is the set of allocated objects in σ. If no such object o exists, then the result of the evaluation is a failure, $\perp$.

The set of all access paths evaluating to object o in state σ is a sound representation of o, in that aliasing between access paths is made explicit, and so flows through the heap can be treated in a sound manner. Unfortunately, this set is, in general, not guaranteed to be finite even in the concrete setting due to cycles in the heap (caused by recursive structures and back pointers). Even if the set is finite, deeply nested objects can produce very long chains.

This mandates a bound, k, on the length of tracked access paths for the static analysis to be tractable. An access path of length greater than k is then soundly approximated (or *widened*) by replacing its suffix, beyond the first k field identifiers, by a special symbol, $*$. The evaluation of widened access path $\langle \mathtt{v}, \langle \mathtt{f1}, \ldots, \mathtt{fk}, * \rangle \rangle$ in concrete state σ yields all the objects in L that are reachable via (zero or more) heap edges from the object $\langle \mathtt{v}, \langle \mathtt{f1}, \ldots, \mathtt{fk} \rangle \rangle$ evaluates to in σ. In practice, we have found that setting $k = 5$ works well, and have used this value for our experiments, which we describe in Section 5.

```
1:  var p = document.URL;
2:  var q = { }
3:  var r = p;
4:  q.f = r;
5:  document.location = q.f;
```

Figure 8: Tracking of Tainted Access Paths

Access paths are a natural way of representing taint flows. Consider, for example, the program in Figure 8. The source statement at line 1 produces the seed access path $\langle \mathtt{p}, \epsilon \rangle$, where ϵ denotes an empty sequence of field identifiers. Next, the assignment at line 3 results in another tainted access path, $\langle \mathtt{r}, \epsilon \rangle$. The statement at line 4, which writes field f, leads to the emergence of a third access path, $\langle \mathtt{q}, \langle \mathtt{f} \rangle \rangle$, which reaches the assignment to sink field location at line 5, and causes a vulnerability to be flagged.

4.2 Taint-analysis Algorithm

To propagate tainted access paths, ACTARUS employs a novel extension of the Reps-Horwitz-Sagiv (RHS) algorithm [26]. RHS provides a highly precise static-analysis framework to transform numerous data-flow problems into graph-reachability problems. ACTARUS seeds taint propagation at sources. Every time a tainted access path is used in an instruction, ACTARUS accordingly taints the access paths that are defined in that instruction.

The taint-propagation process is demand driven in the sense that access paths are instantiated only when taint reaches them, which makes this algorithm very efficient. Another important characteristic of this algorithm is that it is context-sensitive: Each method may assume multiple taint behaviors depending on the context in which it is invoked—a key requirement for precision. We discuss this point further in Sections 4.2.1 and 4.2.2.

Furthermore, ACTARUS enhances RHS since it handles issues that involve aliasing relations in the heap (*i.e.*, multiple local names for the same object). This characteristic is not there in the original RHS algorithm, which does not lend itself to modeling problems that involve aliasing relations established in different procedures.

In the remainder of this section, we first illustrate how the algorithm works in the simple case without aliasing issues, and then discuss how the heap is handled.

4.2.1 Basic Taint-analysis Algorithm

In the absence of any issues with heap aliasing, taint analysis is straightforward: a precise meet-over-all-feasible-paths solution can be computed using a standard Reps-Horwitz-Sagiv (RHS) solver [26]. We illustrate how this works on the example in Figure 9.

```
1:  function id(x) {
2:    return x;
3:  }
4:  function set(y, z) {
5:    z.f = y;
6:  }
7:  var p = document.URL;
8:  var q = { }
9:  var r = id(p);
10:  set(r, q);
```

Figure 9: Code Snippet without Heap Issues

The read of document.URL at line 7 is a source of taint, which generates tainted access path $\langle \mathtt{p}, \epsilon \rangle$. The value of p flows to the invocation of id at line 9. The analysis of id, lines 1–3, reveals that it simply propagates taint from its parameter to its return value, so the *relational summary* $\langle \mathtt{x}, \epsilon \rangle \rightarrow \langle \mathit{ret}, \epsilon \rangle$ is established for the id function (where *ret* is a privileged symbol denoting the return value of the method) and propagated to the callers of id. Applying this summary to the main method at line 9 generates the fact that $\langle \mathtt{r}, \epsilon \rangle$ is tainted.

The summary $\langle \mathtt{x}, \epsilon \rangle \rightarrow \langle \mathit{ret}, \epsilon \rangle$ is said to be *relational* because if there is another invocation of id in the program such that the argument passed to id is *not* tainted, the return value will correctly *not* be tainted in that case. Therefore, while a summary is generated at a callee and propagated to its callers, it is applied to a caller only when the relevant precondition holds in that caller. In this sense, ACTARUS' taint analysis is infinitely context-sensitive: Taint propagation is based on the calling context. Recursion is handled

by the underlying RHS solver. An example with two invocations to the `id` method—one with a tainted argument and the other with a non-tainted argument—will be presented in Section 4.2.2.

Continuing with the example of Figure 9, the value of `r` is passed to `set` at line 10. The `set` function (lines 4–6) contains a field-write instruction, which propagates taint from its first argument to the `f` field of its second argument, thereby creating a non-empty access path. In this case, the function's relational summary is $\langle \mathtt{y}, \epsilon \rangle \rightarrow \langle \mathtt{z}, \langle \mathtt{f} \rangle \rangle$. Applying this summary to the caller of `set` adds the fact that $\langle \mathtt{q}, \langle \mathtt{f} \rangle \rangle$ is tainted. When the analysis terminates, we will have learned that $\langle \mathtt{p}, \epsilon \rangle$, $\langle \mathtt{r}, \epsilon \rangle$ and $\langle \mathtt{q}, \langle \mathtt{f} \rangle \rangle$ are tainted—a precise result.

4.2.2 *Full Taint-analysis Algorithm*

The program in Figure 9 is an *aliasing-free* program; there are never multiple names for the same heap location. That is, $\langle \mathtt{q}, \langle \mathtt{f} \rangle \rangle$ is the only name for the given location. Suppose, however, that some other variable pointed to the same location; in that case, the rules above might cause us to miss the fact that that variable's `f` field is also tainted. For example, consider the variables `q` and `s` in the very similar program in Figure 10; both `q` and `s` refer to the same object, and so any taint that results from one of them must carry over to the other. Hence, our taint-analysis algorithm extends the RHS algorithm to also account for heap aliasing based on the pointer-analysis solution computed during call-graph construction.

```
1:  function id(x) {
2:    return x;
3:  }
4:  function set(y, z) {
5:    var x = z.g;
6:    x.f = y;
7:  }
8:  var p = document.URL;
9:  var q = { g:  { } }
10: var r = id(p);
11: var s = id(q);
12: set(r, s);
```

Figure 10: Code Sample Snippet with Heap Issues

To resolve aliasing relations, we construct an abstraction of the pointer-analysis solution, the "heap graph". ACTARUS' *heap graph* is a bipartite graph, $\mathfrak{H} = \langle \mathfrak{P} \cup \mathfrak{O}, \mathfrak{E} \rangle$, where $\mathfrak{P}$ is the set of environment and heap pointers in the program—that is, local variables and fields that reference objects—and $\mathfrak{O}$ is the set of object abstractions participating in the pointer-analysis solution. Edge $p \rightarrow o$ from pointer p to abstract object o denotes that o may be pointed-to by p. Edge $o \rightarrow p$ from abstract object o to field p denotes that o owns field pointer p. Constructing such a heap-graph is a fairly standard procedure (see, for example, [12]), and our abstraction supports it because the analysis is *field-sensitive*, meaning that it distinguishes the fields of an abstract object from each other as well as fields of different abstract objects [28]. The heap graph for Figure 10 is shown in Figure 11.

In the program of Figure 10, there are two changes compared to Figure 9. The first is an additional call to `id` (line 11), which defines `s`, and the second is the more complex `set` function (lines 4–7). Analysis at first proceeds as above, finding taint for $\langle \mathtt{p}, \epsilon \rangle$ and $\langle \mathtt{r}, \epsilon \rangle$.

The call to `set` illustrates the heap issues. The assignment `x.f = y` establishes the summary $\langle \mathtt{y}, \epsilon \rangle \rightarrow \langle \mathtt{x}, \langle \mathtt{f} \rangle \rangle$, but that is clearly not sufficient: `x.f` refers to the same location as `z.g.f`, and our analysis needs to capture such taint. Sound reasoning about the effect of a field-write statement requires finding (a conservative approximation of) the set of access paths that are aliased with $\langle \mathtt{x}, \langle \mathtt{f} \rangle \rangle$ in the lexical scope of `set`. This is not handled by the original

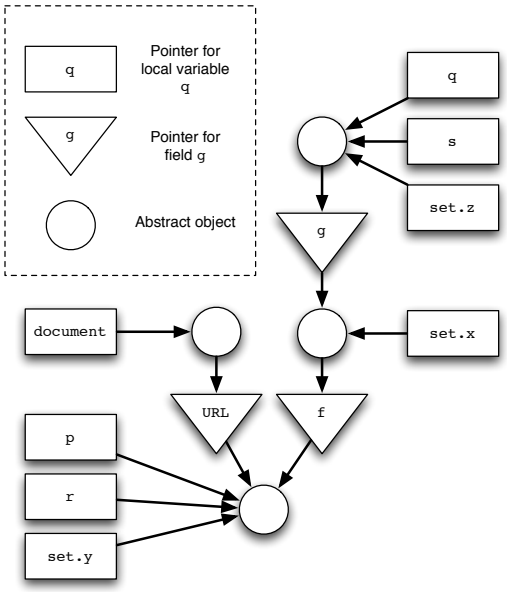

Figure 11: Heap Graph for Figure 10

RHS algorithm [26]. Our analysis, on the other hand, uncovers the aliases of $\langle \mathtt{x}, \langle \mathtt{f} \rangle \rangle$ based on the definitions in Figure 12.

Figure 12 defines function $Aliases$, which, for an access path rooted at a local variable, returns all the access paths that satisfy the following conditions:

1. They are rooted at local variables. This is equivalent to testing that $isLocal(\mathtt{v})$ evaluates to *true*.
2. Those local variables belong to the same method. This is equivalent to testing $methodOf(\mathtt{v}) = methodOf(\mathtt{w})$.
3. Alias the given access path. This is equivalent to testing that the sets of abstract objects obtained with the two calls to $PathTo$ have a non-empty intersection.

This function uses some very intuitive auxiliary functions, including $Truncate$, which limits the length of an access path to a given bound, and $PathTo$, which computes the sets of abstract objects that are reachable through a given access path. Of particular interest is function $FieldName$, which, given a field pointer in $\mathfrak{P}$, returns the corresponding field identifier in $FieldId$. This is necessary because, as we explained above, ACTARUS is a field-sensitive algorithm, and as such distinguishes field pointer keys of different abstract objects even when such field pointer keys represent identically named fields. Every time taint flows into an access path, ACTARUS executes function $Aliases$ to determine local aliases, and then proceeds with the RHS-based taint propagation.

In this example, `z` is indeed a local variable of the same method, and the global pointer analysis records that the `g` field of an object it can name may point to the same object as `x`; hence the heap path $\langle \mathtt{g}, \mathtt{f} \rangle$ for `z` resolves to the same abstract object as $\langle \mathtt{f} \rangle$ for `x`. The path is short enough that there is no need to truncate it. This is illustrated in Figure 11, where the path from `set.x` through `f` leads to the same abstract object as the path from `set.z` through `g` and `f`. The analysis, therefore, computes the relational summary $\langle \mathtt{y}, \epsilon \rangle \rightarrow \langle \mathtt{z}, \langle \mathtt{g}, \mathtt{f} \rangle \rangle$ for `set`. Applying this summary at the caller site (line 12) adds access path $\langle \mathtt{s}, \langle \mathtt{g}, \mathtt{f} \rangle \rangle$. Once again, this is incomplete, since `s` and `q` denote the same object, and hence, the definition in Figure 12 uncovers the additional access paths.

Note that the additional call to `id` (line 11) adds no taint information, since `q` is not tainted to begin with. ACTARUS' context-

Limiting access paths to length k

$$Truncate(\langle \mathtt{f1}, \ldots, \mathtt{fi} \rangle) \leftarrow \begin{cases} \langle \mathtt{f1}, \ldots, \mathtt{fi} \rangle & i \leq k \\ \langle \mathtt{f1}, \ldots, \mathtt{fk}, * \rangle & i > k \end{cases}$$

Abstract objects reachable as `v.f1. ... .fi`

$$PathTo(\langle \mathtt{v}, \langle \mathtt{f1}, \ldots, \mathtt{fi} \rangle \rangle) \leftarrow \left\{ o \;\middle|\; \exists o_1, \ldots, o_{i+1} \in \mathfrak{O}, f_1, \ldots, f_i \in \mathfrak{P} \,.\, \left(\begin{array}{l} (\mathtt{v} \rightarrow o) \in \mathfrak{E} \wedge \\ \quad \forall j = 1, \ldots, i \left(\begin{array}{l} isObject(o_j) \wedge \\ isField(f_j) \wedge \\ FieldName(f_j) = \mathtt{fj} \wedge \\ (o_j \rightarrow f_j) \in \mathfrak{E} \wedge \\ (f_j \rightarrow o_{j+1}) \in \mathfrak{E} \end{array} \right) \wedge \\ o_{i+1} = o \end{array} \right) \right\}$$

Local aliases of `w.g1. ... .gj`

$$Aliases(\langle \mathtt{w}, \langle \mathtt{g1}, \ldots, \mathtt{gj} \rangle \rangle) \leftarrow \left\{ \langle \mathtt{v}, \langle \mathtt{f1}, \ldots, \mathtt{fi} \rangle \rangle \;\middle|\; \exists \mathtt{h1}, \ldots, \mathtt{ht} \in FieldId \,.\, \left(\begin{array}{l} isLocal(\mathtt{v}) \wedge \\ methodOf(\mathtt{v}) = methodOf(\mathtt{w}) \wedge \\ \langle \mathtt{f1}, \ldots, \mathtt{fi} \rangle = Truncate(\langle \mathtt{h1}, \ldots, \mathtt{ht} \rangle) \wedge \\ PathTo(\langle \mathtt{v}, \langle \mathtt{h1}, \ldots, \mathtt{ht} \rangle \rangle) \cap \\ \quad PathTo(\langle \mathtt{w}, \langle \mathtt{g1}, \ldots, \mathtt{gj} \rangle \rangle) \neq \emptyset \end{array} \right) \right\}$$

Figure 12: Computation of Local Aliases on Heap Graph $\mathfrak{H} = \langle \mathfrak{O} \cup \mathfrak{P}, \mathfrak{E} \rangle$ with Access-path-length Bound k

sensitive taint propagation, discussed in Section 4.2.1, saves us from adding $\langle \mathtt{s}, \epsilon \rangle$ to the set of tainted access paths, which would have made the analysis very coarse and generated numerous false positives. In fact, the tainted access path $\langle \mathtt{s}, \langle \mathtt{g}, \mathtt{f} \rangle \rangle$ that we have computed for `s` is much more precise than just $\langle \mathtt{s}, \epsilon \rangle$.

This algorithm for resolving aliasing relations is *flow-insensitive* with respect to fields, meaning that it does not account for *strong updates* on fields; *i.e.*, if field `f` of object o is assigned values v and w at two different program points, the analysis conservatively considers field `f` to point to the set of values $\{v, w\}$. Conversely, a *flow-sensitive* analysis attempts to determine which write instruction is performed first and which last, and based on that information it reports that `f` points to *either* v *or* w.

While flow-sensitive analyses may appear to be more precise, they are not always sound. This is particularly true for JavaScript. In fact, since in JavaScript the execution of programs is often event-driven—based, for example, on the click of a button or the interaction with a User Interface (UI) gadget—the order of execution of certain routines cannot always be established. Attempting to assert an order of execution can lead to unsound results. Therefore, we chose to conservatively make the analysis flow-insensitive with respect to fields.

Inside a procedure, however, ACTARUS is flow-sensitive with respect to local variables because it accounts for strong updates on them. In fact, for every access path $\langle \mathtt{v}, \langle \mathtt{f1}, \ldots, \mathtt{fn} \rangle \rangle$ ACTARUS instantiates, variable `v` is in SSA form. As described in Section 3.4, SSA creates variable versions to make sure that each variable gets assigned only once, which indirectly provides a measure of intra-procedural flow-sensitivity.

4.3 Handling First-class Fields

It should be observed that, while traversing the heap graph backwards, special field pointer keys may be encountered that correspond to reflective property accesses, where the properties could not be fully disambiguated through string-constant propagation or string concatenation, as explained in Section 3.3. In such cases, the sound solution adopted by ACTARUS is to consider all the possible properties of an object upon which a reflective property access has been performed, compatible with the type of the property that is being accessed reflectively. For example, even though a property cannot be disambiguated, it may be possible to establish statically that its type is `Number`. All the properties of type `Number` should then be considered for this step in the access path, whereas all the others can be safely excluded.

```
1:  function id(x) {
2:    return x;
3:  }
4:  function set(y, z, f) {
5:    var x = z.g;
6:    x[f] = y;
7:  }
8:  var p = document.URL;
9:  var q = { g:  { } }
10: q.g.k = "safe";
11: var r = id(p);
12: var s = id(q);
13: set(r, s, (...  ?  "f" :  "g"));
14: document.write(q.g.k);
```

Figure 13: Sample Code Snippet with Ambiguous Field Use

Disambiguation of property names also allows for fairly precise analysis results. Consider the code snippet in Figure 13—a slightly modified version of the example of Figure 10. The `set` function (lines 4–7) now takes an additional parameter `f`, establishing which of the properties of `x` will be set, and the property-write instruction at line 6 uses `f` as a variable instead of the literal `f`. Furthermore, `q.g` has a field `k` that is set to constant string `"safe"` at line 10, and there is now a sink method call at line 14. The call to `set` (line 13) passes either `"f"` or `"g"` as the name of the property to set, based on the evaluation of a boolean condition (indicated, for brevity, as `"..."`). In this case, the analysis will assume that either property `f` or property `g` has been written, and the taint analysis will conservatively record that both access paths $\langle \mathtt{q}, \langle \mathtt{g}, \mathtt{f} \rangle \rangle$ and $\langle \mathtt{q}, \langle \mathtt{g}, \mathtt{g} \rangle \rangle$ are potentially tainted. This assumption, while conservative, is still quite precise because it does not indiscriminately taint all the access paths rooted at `q`. In particular, $\langle \mathtt{q}, \langle \mathtt{g}, \mathtt{k} \rangle \rangle$ is not

marked as tainted. Therefore, the sink method call at line 14 is not flagged as a vulnerability, which is a precise result.

5. EXPERIMENTAL RESULTS

ACTARUS was tested on three sets of benchmarks to examine its ability to discover taint violations. The first set of benchmarks was a test suite designed by the authors of this paper, and now made available to the scientific community[6] to advance further research in this area. The suite was comprised of over 140 micro benchmarks to test the precision and soundness of ACTARUS. The second set was a collection of Web pages obtained from crawling the top 50 sites according to Alexa[1]. The third set was a collection of Web pages obtained by crawling an additional 19 large, well known Web sites.

The first set of benchmarks, the ACTARUS test suite designed by the authors, was used primarily for testing. The test suite included tests that ranged from basic tests that any taint analysis should pass, to complex tests that could only be passed with an advanced taint-analysis technique. Some of the advanced tests included tainting lexically scoped variables, tainting variables that are interproceduraly aliased, overwriting sanitizers, accessing variables through the arguments array, and accessing tainted data through the prototype chain. All these microbenchmarks test patterns that are used in real JavaScript applications. ACTARUS correctly finds all exploits in these microbenchmarks.

The second and third set of benchmarks comprise real Web pages that were downloaded off the Internet. For the second set of benchmarks, the top 50 sites according to Alexa were crawled to a level of 3 links away from the home page. The top 50 sites were chosen due to time limitations; however, many seemingly popular or important sites are not in the top 50. To account for this, the third set of benchmarks is a collection Web sites that the authors thought were popular or important. Both sets of Web sites were crawled to a limited depth, again due to time limitations.

The Web pages from the second and third benchmarks were combined and ACTARUS ran on a sample of them. This was done because the crawls produced over 30,000 pages, which was a prohibitive number for the purposes of this evaluation. The sampling was done by picking various places to start in our alphabetically ordered list of Web addresses. In total, ACTARUS was run on 12,473 Web pages and successfully analyzed 9,726 pages. ACTARUS failed due to a timeout of 4 minutes or an exception originated from parsing illegal JavaScript code in approximately 22% of the runs. Since ACTARUS analyzes each Web page individually, it would be trivial to increase analysis throughput by running many copies of ACTARUS on many computers. This would also lessen the need for a timeout. For this paper, however, ACTARUS was run on only 4 disparate computers at the same time.

Site	Unique True Positives	Total True Positives
A	7	80
B	4	12
C	4	91
D	7	13
E	2	4
F	1	200
G	1	1
H	1	114
I	3	7
J	1	3
K	1	1

Table 1: True Positives for a Sample of the Crawled Web Sites

To save bandwidth and reduce download time, JavaScript files are typically *minified*, which includes reducing variable names to single characters and eliminating new-line characters. While the resulting JavaScript code is (almost always) still legal, reporting issues on JavaScript programs that are only one line long is very problematic since the developer cannot easily detect the tainted flows. For this reason, ACTARUS is integrated with JavaScript Unpacker and Beautifier[7], a tool that preprocesses and nicely formats all the Web pages before passing them to ACTARUS for the taint analysis. The 4 minutes granted to ACTARUS to complete the analysis of a Web page and its embedded JavaScript code included the time necessary to beautify the JavaScript code.

Table 1 shows the results from our study of the issues reported by ACTARUS.[8] For a given Web site, the table shows the numbers of "total true positives" and "unique true positives" for a Web site. A *true positive* is a feasible data-flow path from an untrusted source to a security-sensitive sink, not intercepted by a sanitizer; the source, sink and sanitizer belong to the same security rule. In total, ACTARUS found true positives on 11 different sites. However, due to the way Web sites are designed—which includes the fact that Web pages in the same site often point to code templates and libraries, or even include identical copies of certain JavaScript programs—the same true positive may be present in multiple pages on the same Web site. Therefore, we considered it important to count the *total true positives* as the total number of true positives found on that site. If the same code was reported as a true positive multiple times, we counted it once as a *unique true positive*. Comparing unique true positives with total true positives for a given Web site gives an indication of the impact that individual issues can have on a Web site if an attacker exploits them.

The results in Table 1 indicate that a true positive may be present in many different Web pages. This is very useful information for developers. If their Web site is based around automatic code reuse via templates or server-side code inclusion, the developer will know how to fix the problem at the source. If their Web site was designed by copying and pasting pages, with little or no automatic code reuse, these findings tell them all the pages that may contain a vulnerability.

ACTARUS conservatively considers the entire DOM to be tainted. This is necessary because untrusted user-provided data may appear anywhere on a Web page. One example is when user data is read from a database and output as part of a Web page. This leads to some issues being reported for assigning one part of the DOM to another part of the DOM. These are true positives because untrusted data is potentially being used in a new context which may be dangerous. For instance, one part of the DOM could be inserted into a script segment in a different part of the DOM. What was previous benign HTML will now be interpreted as a script.

With the help of pretty printing the results, a useful user interface to display the results, and detailed taint-flow reporting (all implemented as part of the ACTARUS tool), it took a total of 6 person days to manually separate the true positives from the false positives for the 9,726 pages ACTARUS analyzed. ACTARUS's total-true-positive rate was slightly over 40%, meaning that out of every 5 issues reported, 2 were true positives and 3 were false positives. Since false positives are non-issues, distinguishing between unique and total false positives would not generate helpful information. Rather than delving into this expensive classification, we preferred to study the most recurrent characteristics of the false positives, which we used to further refine ACTARUS. A recurrent source of

[7] `http://jsbeautifier.org`.

[8] The names of the Web sites have been anonymized.

false positives consisted of tainted flows reaching the parameters in a URL. While tainting the hostname and/or path in a URL is a serious vulnerability, which can lead to redirection attacks, tainting the parameters is not a vulnerability; parameters are always tainted. We observed that refining the analysis by rejecting flows that reach the parameters in a URL would dramatically cut the false-positive rate. As part of future work, we will integrate ACTARUS with an analysis that removes many of these false positives.

6. RELATED WORK

In this section, we survey research work in the space of JavaScript security and static taint analysis. Various approaches have been proposed for JavaScript security bug finding. Chugh, *et al.* [6] present a staged approach for handling JavaScript's dynamic nature. First, static analysis is applied to as much of the information flow as possible based on the known code. Then, at the browser, the residual checking is performed whenever new code is dynamically loaded. Contrary to [6], ACTARUS does not require a staged approach, and relies solely on static analysis of the JavaScript application. ACTARUS also employs more precise abstractions and accounts for more JavaScript constructs.

Guha, *et al.* [17] use static analysis to extract a model of expected client behavior, as seen from the server, for JavaScript programs. This model is then used to build an intrusion-prevention proxy for the server. To avoid mimicry attacks, random asynchronous requests are inserted. In a related study, Vogt, *et al.* [36] propose a system that stops XSS attacks already on the client side by tracking the flow of sensitive information inside the Web browser. If sensitive information is about to be transferred to a third party, then the decision whether to permit the transfer is delegated to the user. The system is integrated into the Firefox browser.

Maffeis, *et al.* [21] study three techniques that are effective in protecting sensitive properties of honest code against an attacker that supplies code to be executed in the same JavaScript environment: *filtering*, *rewriting* and *wrapping*. Filtering is a static analysis that takes place once, before the untrusted code is loaded, to judge whether the code conforms to certain criteria; if not, then the code is rejected. Rewriting inserts run-time checks to inhibit undesirable actions by the untrusted code. Finally, wrapping is used to protect sensitive resources of the trusted environment by enclosing them inside functions that perform run-time checks to verify that these resources are not used maliciously by untrusted code.

Yu, *et al.* [41] use another form of rewriting to eliminate security attacks due to JavaScript. Their algorithm takes a JavaScript application, identifies relevant operations, modifies suspicious behaviors, and prompts the user on how to proceed when appropriate.

Gatekeeper, by Guarnieri and Livshits [16], detects security and reliability problems in JavaScript widgets. It uses a static pointer analysis for program understanding, and issues queries against the pointer analysis to detect dangerous behavior, even in the presence of malicious obfuscation. ACTARUS also is based on a pointer analysis, but it is designed to operate on large applications, and contains advanced rules and special handling for JavaScript constructs.

KUDZU, by Saxena, *et al.* [29], uses symbolic execution to explore the execution space of JavaScript applications. It assumes as input the URL for a Web application, and outputs a high-coverage test suite to systematically explore it. It uses a specialized constraint solver for the theory of strings, which expresses the semantics of JavaScript operations. It has been applied to find client-side code-injection vulnerabilities in Web applications. In another study [30], Saxena, *et al.* describe FLAX, a taint-enhanced black-box fuzzer that finds validation bugs in JavaScript programs. FLAX avoids false alarms and demonstrates the presence of candidate vulnerabilities by generating test cases via random fuzz testing.

VEX, by Bandhakavi, *et al.* [4], computes tainted flows for every source/sink pair, regardless of whether those sources and sinks belong to the same security rules, and without precomputing reachable sources and sinks. After this expensive computation is performed, results are post-processed. In contrast, ACTARUS only computes flows starting at a *reachable* source, and stops as soon as those flows reach a sanitizer or sink *from the same rule*. This is a potentially huge performance improvement. Furthermore, unlike VEX, ACTARUS conservatively handles prototype-chain lookups. Like VEX, ACTARUS handles property accesses conservatively. Unlike VEX, whenever a property is statically known through constant propagation or constant-string concatenation analysis, ACTARUS replaces the property access with a load or store for the specific field that has been computed.

Typically, the data manipulated by a program can be tagged with security levels [10], which naturally assume the structure of a poset. Under certain conditions, this poset is a lattice [9]. Given a program, the principle of *non-interference* dictates that low-security behavior of the program be not affected by any high-security data, unless that high-security data has been previously downgraded [14]. The taint-analysis problem described in this paper is an information-flow problem in which high data is the untrusted output of a source, low-security operations are those performed by sinks, and untrusted data is downgraded by sanitizers.

Volpano, *et al.* [37] have shown a type-based algorithm that certifies implicit and explicit flows and guarantees non-interference. Shankar, *et al.* present a taint analysis for C using a constraint-based type-inference engine based on `cqual` [31]. To find format string bugs, `cqual` uses a type-qualifier system [13] with two qualifiers: *tainted* and *untainted*. The variables whose values can be controlled by an untrusted adversary are qualified as tainted, all the others as untainted. Similar to the propagation graph built by ACTARUS, a constraint graph is constructed for a `cqual` program. If there is a path from a tainted node to an untainted node in the graph, an error is flagged. Myers' Java Information Flow (JIF) [23] uses type-based static analysis to track information flow. JIF considers all memory as a channel of information, which requires that every variable, field, and parameter used in the program be statically labeled. Labels can be either declared or inferred.

Ashcraft and Engler's taint analysis [2] provides user-defined sanity checks to untaint potentially tainted variables. Snelting, *et al.* [33] and Hammer, *et al.* [18] propose a flow-sensitive program-slicing-based taint analysis for Java.

Livshits and Lam [20] present a taint analysis for Java that is engineered to track taint flowing through heap-allocated objects. Their analysis requires prior computation of Whaley and Lam's flow-insensitive, context-sensitive may-points-to analysis, which is based on Binary Decision Diagrams (BDDs) [40], and requires the presence of programmer-supplied descriptors for sources, sinks, and library methods that handle objects through which taint may flow. An important distinction is that ACTARUS automatically generates the summaries it consumes, and requires no user input. It is also unclear whether BDD-based static analysis can scale to large applications when using object sensitivity [19]. A more recent study by Tripp, *et al.* [35] presents Taint Analysis for Java (TAJ), which is effective at analyzing real-world Web applications. The main limitation of TAJ is that it trades soundness for scalability and accuracy. This contributes to the usability of TAJ, but leaves the security status of the subject application unknown.

Wassermann and Su extend Minamide's string analysis [22] to syntactically isolate tainted substrings from untainted substrings in PHP. They label non-terminals in a Context-Free Grammar (CFG)

with annotations reflecting taintedness and untaintedness. Their expensive, yet elegant, mechanism is applied to detect both injection [38] and XSS [39] vulnerabilities.

7. CONCLUSION

This paper has presented ACTARUS, a novel, product-quality taint-analysis algorithm for JavaScript. ACTARUS soundly models most of the characteristics of JavaScript that have traditionally made static analysis of JavaScript programs a challenging problem. This paper has also discussed the experimental results obtained by running ACTARUS on 9,726 Web pages, obtained by crawling the 50 most visited Web sites worldwide as well as other popular Web sites. Such results expose 526 vulnerabilities on 11 sites. Those vulnerabilities can cause malicious JavaScript code execution.

8. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, Copenhagen, Denmark, 1994.

[2] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *S&P*, 2002.

[3] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA*, 1996.

[4] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting Browser Extensions for Security Vulnerabilities. In *USENIX Security*, 2010.

[5] W. Chang, B. Streiff, and C. Lin. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. In *CCS*, 2008.

[6] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged Information Flow for JavaScript. In *PLDI*, 2009.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *TOPLAS*, 13(4), 1991.

[8] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP*, 1995.

[9] D. E. Denning. A Lattice Model of Secure Information Flow. *CACM*, 19(5), 1976.

[10] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *CACM*, 20(7), 1977.

[11] A. Deutsch. A Storeless Model of Aliasing and Its Abstractions Using Finite Representations of Right-regular Equivalence Relations. In *ICCL*, 1992.

[12] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective Typestate Verification in the Presence of Aliasing. In *ISSTA*, 2006.

[13] J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *PLDI*, 2002.

[14] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *S&P*, 1982.

[15] D. Grove and C. Chambers. A Framework for Call Graph Construction Algorithms. *TOPLAS*, 23(6), 2001.

[16] S. Guarnieri and B. Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for Javascript Code. In *USENIX Security*, 2009.

[17] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *WWW*, 2009.

[18] C. Hammer, J. Krinke, and G. Snelting. Information Flow Control for Java Based on Path Conditions in Dependence Graphs. In *S&P*, 2006.

[19] O. Lhoták and L. J. Hendren. Context-Sensitive Points-to Analysis: Is It Worth It? In *CC*, 2006.

[20] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security*, 2005.

[21] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with Filters, Rewriting and Wrappers. In *ESORICS*, 2009.

[22] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *WWW*, 2005.

[23] A. C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *POPL*, 1999.

[24] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *SOSP*, 1997.

[25] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.

[26] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, 1995.

[27] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *PLDI*, 2010.

[28] B. G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Languages. In *CC*, 2003. Invited Paper.

[29] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

[30] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*, 2010.

[31] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *USENIX Security*, 2001.

[32] O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

[33] G. Snelting, T. Robschink, and J. Krinke. Efficent Path Conditions in Dependence Graphs for Software Safety Analysis. *TOSEM*, 15(4), 2006.

[34] M. Sridharan, S. J. Fink, and R. Bodík. Thin Slicing. In *PLDI*, 2007.

[35] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. In *PLDI*, 2009.

[36] P. Vogt, F. Nentwich, N. Jovanovich, E. Kirda, C. Kruegel, and G. Vigna. Cross-site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, 2007.

[37] D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *JCS*, 4(2-3), 1996.

[38] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *PLDI*, 2007.

[39] G. Wassermann and Z. Su. Static Detection of Cross-site Scripting Vulnerabilities. In *ICSE 2008*, 2008.

[40] J. Whaley and M. S. Lam. Cloning Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI*, 2004.

[41] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *POPL*, 2007.