✓ Artifact evaluated by FSE

Static DOM Event Dependency Analysis for Testing Web Applications

Chungha Sung, Markus Kusano
Virginia Tech
Blacksburg, VA, USA

Nishant Sinha
IBM Research
Bangalore, India

Chao Wang
Univ. of Southern California
Los Angeles, CA, USA

ABSTRACT

The number and complexity of JavaScript-based web applications are rapidly increasing, but methods and tools for automatically testing them are lagging behind, primarily due to the difficulty in analyzing the subtle interactions between the applications and the event-driven execution environment. Although static analysis techniques have been routinely used on software written in traditional programming languages, such as Java and C++, adapting them to handle JavaScript code and the HTML DOM is difficult. In this work, we propose the first constraint-based declarative program analysis procedure for computing dependencies over program variables as well as event-handler functions of the various DOM elements, which is crucial for analyzing the behavior of a client-side web application. We implemented the method in a software tool named JSDEP and evaluated it in ARTEMIS, a platform for automated web application testing. Our experiments on a large set of web applications show the new method can significantly reduce the number of redundant test sequences and significantly increase test coverage with minimal overhead.

CCS Concepts

•**Software and its engineering** → *Automated static analysis; Software testing and debugging;*

Keywords

JavaScript, Static analysis, Automated testing, Event dependency, Partial order reduction

1. INTRODUCTION

Static analysis of client-side JavaScript web applications is difficult not only due to the language's dynamic features [19, 35] but also due to the subtle interactions between JavaScript code and the event-driven execution environment. At the center of this execution environment is the HTML Document Object Model (DOM). The DOM stores the buttons, images, text-boxes, and other visible objects on the web page, together with a large number of event-handler functions attached to these DOM objects. Prior work on statically analyzing JavaScript focused primarily on modeling the

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...
http://dx.doi.org/10.1145/2950290.2950292

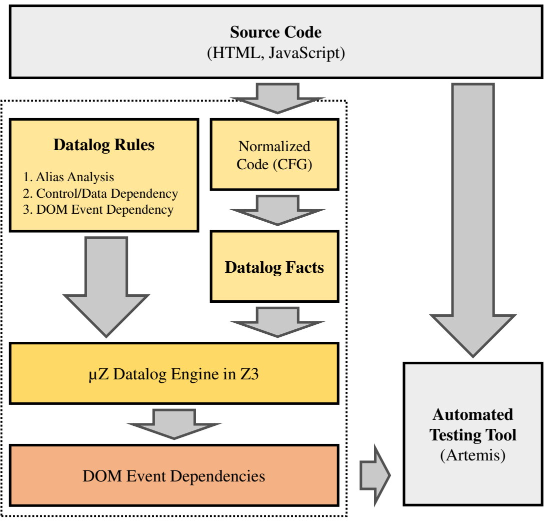


Figure 1: Overall flow of DOM-event dependency analysis.

language [3, 8, 14–16, 31, 39] as opposed to the language's interaction with the DOM. For example, existing methods do not robustly handle dependencies between DOM event handlers, e.g., the various functions responding to the user's actions, timers, AJAX requests, or their callbacks, despite that such dependencies are crucial in reasoning about client-side web applications.

We propose the first constraint-based static analysis method for computing dependencies both across event-handlers and between HTML DOM elements. Such DOM event dependencies fundamentally differ from traditional control and data dependencies over program variables because they are tied to the event-driven execution environment. Specifically, a modern JavaScript web application stores various data inside the DOM while simultaneously using JavaScript code to read and manipulate this data in response to various, often user-triggered, events such as *onclick*, *onload*, and *timeout*. If executing the handler m_A of event A causes the handler m_B of event B to be registered, triggered, or removed, we say that event B depends on event A, denoted $A \rightarrow_{DOM} B$. This differs from the traditional notion of control dependencies ($\rightarrow_{ctrl}$) and data dependencies ($\rightarrow_{data}$) over program variables. Furthermore, statically reasoning about DOM event dependencies is challenging: it requires proper handling of the aliasing between DOM elements, and modeling the effects of APIs provided by the web browser and popular frameworks such as *jQuery*.

Figure 1 shows the flow of our DOM event dependency analysis, which follows the declarative program analysis framework [25, 28, 32, 42]. Given the HTML and JavaScript source file(s) of a client-side web application, we first extract the JavaScript code and generate its control flow graph (CFG). We traverse the CFG to encode its control and data flows in a set of logical constraints called Datalog *facts*. Next, we specify our static dependency analysis in a set of Datalog inference *rules*. Finally, we use an off-the-shelf Datalog engine [17] in Z3 [10] to solve the Datalog program. Internally, the Datalog engine repeatedly applies the set of inference rules to the set of facts until they reach a fix-point. The fix-point results in a new relation $\rightarrow_{DOM}$ over DOM events. This relation allows the user to query for dependency information through Z3's API.

Our method for statically computing DOM event dependencies differs from the prior work. First, it differs from the declarative methods [6, 25, 28, 32, 42] for analyzing programs written in standard programming languages such as Java: we analyze JavaScript web applications. Additionally, the static analysis of Guarnieri and Livshits [14], while targeting JavaScript, focused on type inference as opposed to inter-event-handler dependencies in the HTML DOM. Our method also differs from the dynamic change impact analysis of Alimadadi et al. [2], which analyzed concrete executions to identify the interplay between JavaScript code changes and the content of the DOM: since it is dynamic, their analysis is valid only for the given executions; ours, based on static analysis, is valid for all executions. Madsen et al. [29, 30] proposed several static analysis methods for JavaScript, but they targeted applications using *Node.js* [30] or Windows 8 APIs [29]. The static analysis tool of Jensen et al. [20] modeled some aspects of the HTML DOM and browser APIs, but its focus was on type inference as opposed to a dependency analysis.

We implemented our new method in a static analysis tool named JSDEP, building upon ESPRIMA for parsing the JavaScript source code, JS-WALA for generating the control-flow graph, and Z3 for solving the Datalog program. We evaluated JSDEP on a large set of client-side web applications. Overall, we analyzed 21 programs totaling 18,559 lines of JavaScript code. Our experiments show that our static analysis method can quickly process the JavaScript code of these applications and compute the DOM event dependencies with reasonable accuracy.

To demonstrate our technique's usefulness, we leveraged its results to improve the performance of a popular automated web application testing tool named ARTEMIS [5]. ARTEMIS traverses the application's execution space by systematically triggering handlers of various DOM events. However, since ARTEMIS cannot statically compute DOM event dependencies, it relies on heuristics for generating sequences of event-handler executions. We show empirically that these heuristics are largely random and introduce many redundant tests. But, our DOM event dependency analysis can *provably* prune redundant test sequences and thus direct ARTEMIS to explore truly useful tests. In particular, the default ARTEMIS stuck at 67% statement coverage even after running for 3.5 hours, whereas our new method enabled ARTEMIS to quickly reach 80% coverage.

Besides ARTEMIS, our static DOM event dependency analysis may benefit other dynamic analysis or symbolic execution tools such as Kudzu [37], SymJS [26], and Jalangi [38]. A problem that is common to these tools is that they lack the capability of conducting a whole-program static analysis; in this sense, our new method is complementary. In a broader sense, our dependency analysis method is useful in many other software engineering applications, e.g., to improve program understanding, software maintenance, automated debugging, and program repair.

In summary, the main contributions of our work are:

- We propose the first constraint-based static dependency analysis for client-side web applications, taking into consideration not only traditional control and data dependencies but also the new DOM-event dependencies.
- We propose a new method for leveraging our static dependency analysis results in an automated web application testing tool, ARTEMIS, to eliminate redundant tests and improve test coverage.
- We implement these new methods and evaluate them on a large set of web applications to demonstrate the efficiency of the static analysis method and its effectiveness in improving automated testing.

The remainder of this paper is organized as follows. We first motivate the main ideas of our new methods through examples in Section 2. We establish notation in Section 3, and formalize our static dependency analysis in Section 4. We present the integration of our dependency analysis with ARTEMIS in Section 5. We evaluate our approach empirically in Section 6. Finally, we review related work and conclude in Sections 7 and 8.

2. MOTIVATION

In this section, we show what DOM event dependencies are and how they can improve the automated testing of web applications.

2.1 DOM Event Dependency

Consider the example in Figure 2. An HTML file defines the DOM elements including four buttons, and a JavaScript file defines the functions manipulating these elements. The four buttons, named `test1`–`test4`, are referenced in the JavaScript using the variables `a`, `b`, `c`, and `d`, respectively. The *onclick* event handler of `a`, i.e., the function executed if the button `test1` is clicked, registers the *onclick* event handler of `c` to the function `makeSomeNoise()`. The *onclick* event handler of `b` increments the value of `x`. Since `x` is used in `makeSomeNoise()` to control the branch conditions, the handler of `b` affects, in some sense, the behavior of the event handler of `c`. Finally, the event handler of `d` prints a message to the console. From the JavaScript code, we identify the following dependencies:

- Clicking `test1` registers an event handler to `test3`.
- Clicking `test2` increments the value of `x`, which in turn affects the same handler of `test2`.
- Clicking `test3` traverses the program paths of the handler function `makeSomeNoise()` based on the value of `x`.

We say that the *onclick* event of `test3` depends on the *onclick* event of `test1` since the handler of `test3` is registered only when the handler of `test1` is executed. Also, the *onclick* event of `test3` depends on the *onclick* event of `test2` since the handler of `test2` modifies the value of `x` read by the handler of `test3`. Similarly, the *onclick* event of `test2` depends on itself due to the reads/writes to `x`. In contrast, the handler of `test4` does not depend on any DOM event. These DOM event dependencies are shown in Figure 3.

There are two types of dependencies in Figure 3: one relying on traditional control and data dependencies, and another relying on the new DOM event dependency relation. Specifically, `test2` depends on `test2` because the read and write of variable `x` in the handler of `test2` changes the program state, which also affects the behavior of `makeSomeNoise()` for `test3`. In contrast, `test3` depends on `test1` because the handler of `test1` installs the handler of `test3` – this type of dependency arises only from the specific event-driven execution environment of the web browser; it cannot be expressed using the traditional control and data dependency relations.

To the best of our knowledge, the only work somewhat related to our new dependency analysis is the change impact analysis procedure developed by Alimadadi et al. [2]. It monitors the interplay between JavaScript code changes and their impact on the DOM. However, it relies on a trace-based dynamic analysis, and is therefore only valid for the given execution traces. Our method, in contrast, is solely static and valid over all possible executions. In addition, the modeling of dependencies between event handlers in Ali-

```
 1  <html>
 2  <head>
 3    <p Click example of three buttons </P>
 4    <script type="text/javascript" src="ex.js"></
        script>
 5  </head>
 6  <body>
 7    <div id="content"> ...  </div>
 8    <div id="buttons">
 9      <button id="test1" type="button"> b1 </button>
10      <button id="test2" type="button"> b2 </button>
11      <button id="test3" type="button"> b3 </button>
12      <button id="test4" type="button"> b4 </button>
13    </div>
14  </body>
15  </html>
```

```
 1  var a = document.getElementById('test1');
 2  var b = document.getElementById('test2');
 3  var c = document.getElementById('test3');
 4  var d = document.getElementById('test4');
 5  var x = 0;
 6  function makeSomeNoise() {
 7    if (x<2) {console.log("x is lower than 2");}
 8    else if (x<4) {console.log("x is lower than 4");}
 9    else if (x<6) {console.log("x is lower than 6");}
10    else if (x<8) {console.log("x is lower than 8");}
11    else {console.log("x is higher than 8");
12          some error codes;}
13  }
14  a.addEventListener("click", function() {
15    c.onclick = makeSomeNoise; });
16  b.addEventListener("click", function() {
17    x = x + 1; });
18  d. addEventListener("click", function() {
19    console.log("test4 is clicked!"); });
```

Figure 2: Example HTML page and associated JavaScript file.

madadi et al. [2] is not as accurate as our method. In particular, they assume that function g depends on function f (Definition 9 in [2]) if f invokes g and either (1) the signature of g indicates that it takes parameters or (2) the definition of f includes a return value. This is a much coarser definition than ours: we model the actual impact of the statements in a function during our dependency analysis.

2.2 Web Application Testing

Next we show how DOM event dependencies can help improve automated web application testing tools like ARTEMIS. Such tools generate test sequences by systematically triggering user events up to a fixed depth. The search tree of our running example (Figure 2) up to depth three can be seen in Figure 4 (a). Each edge represents the execution of an event handler, and each path represents a test sequence. The default algorithm in ARTEMIS inefficiently explores the search space since many of its randomly generated test sequences are actually redundant. For example, the *onclick* event

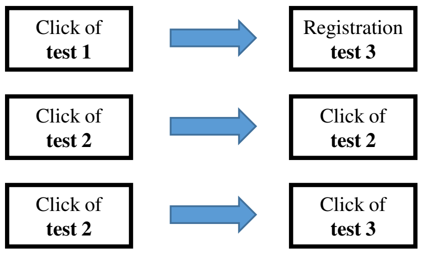
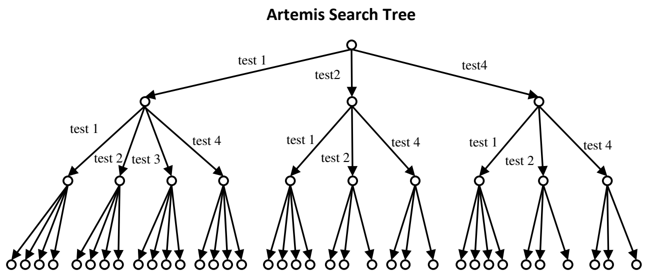
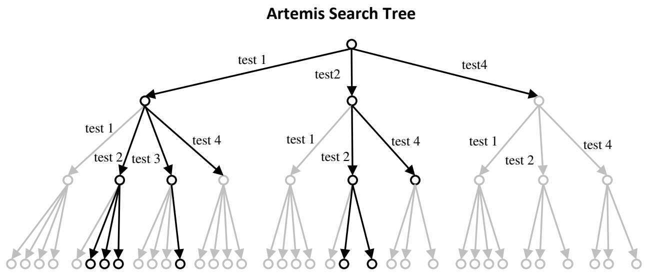


Figure 3: DOM event dependencies for the example in Figure 2.



(a) The default algorithm with no pruning.



(b) With DOM event dependency based pruning.

Figure 4: Event sequences explored by ARTEMIS for Figure 2.

of `test4` does not have a DOM event dependency with any DOM event. Any permutation of events involving `test4` is redundant, e.g.: `test1` $\rightarrow$ `test4` $\rightarrow$ `test3` leads to the same behavior as `test1` $\rightarrow$ `test3` $\rightarrow$ `test4` and therefore only one needs to be tested.

Using newly computed DOM event dependencies in ARTEMIS allows redundant test sequences to be pruned away. We will explain the detailed redundancy-pruning algorithm in Section 5, but for now, it suffices to say that permutations involving two independent event handlers can safely be ignored without affecting the exploration capability of the tool. After such reduction, the new search tree, shown in Figure 4 (b), is significantly smaller. Here, grayed-out edges are those deemed redundant and therefore are skipped. For example, the *onclick* event of `test1` is not dependent with itself as seen in the dependency relation in Figure 3. So, executing the *onclick* event of `test1` after another *onclick* event of `test1` does not alter the program's state and therefore can be skipped. Similarly, `test1` $\rightarrow$ `test4` $\rightarrow$ `test3` is skipped because an equivalent sequence, `test1` $\rightarrow$ `test3` $\rightarrow$ `test4`, has already been tested.

Also note that exploring all test sequences up to the depth 3 does not guarantee to cover all statements in this program. Indeed, only the first branch of the function `makeSomeNoise()` in Figure 2 (Line 9) can be executed; sequences of only length three are not long enough to increment `x` above 2 while also registering and executing the handler associated with `test3`. Fully covering all the statements, in this case, requires at least a sequence of length 15: That is, `test1` $\rightarrow$ `test3` $\rightarrow$ `test2` $\rightarrow$ `test2` $\rightarrow$ `test3` $\rightarrow$ `test2` $\rightarrow$ `test2` $\rightarrow$ `test3` $\rightarrow$ `test2` $\rightarrow$ `test2` $\rightarrow$ `test3` $\rightarrow$ `test2` $\rightarrow$ `test2` $\rightarrow$ `test3` $\rightarrow$ `test4`.

Since we need to test up to depth 15 for full statement coverage, the default search algorithm in ARTEMIS may explore *more* than $3^1 + \cdots + 3^{15} = 21,523,359$ sequences. In contrast, with our new pruning technique, complete statement coverage can be achieved by exploring at *most* 60 sequences. We ran ARTEMIS with our new improvement on this example and reached 100% coverage in only 0.37 seconds. The original version of ARTEMIS could not reach 100% coverage after 10 minutes.

In the remainder of this paper, we present the detailed algorithm of our new DOM event dependency analysis.

3. PRELIMINARIES

In this section, we introduce the fundamental concepts and notations for our work.

3.1 Web Applications

Client-side web applications are executed by the web browser, which loads and parses the HTML/JavaScript files, represents them as a DOM tree, and then executes the JavaScript code. Each node in the DOM tree represents an object on the web page, or a JavaScript code block to be executed immediately after parsing. Each object may also be associated with a set of events initiated either by the user or by the browser, such as *onload* and *onclick*. These events are responded to by a set of JavaScript functions called event handlers. For example, when a user clicks a button, the callback function associated with the *onclick* event will be executed. Callback functions may be registered statically inside the HTML file or dynamically inside the JavaScript code. Although the browser ensures that each callback function is executed atomically, i.e., in a single-threaded fashion, the execution of multiple callback functions may interleave; this makes the execution of the entire web application nondeterministic.

3.2 JavaScript Statements

Let St be the set of JavaScript statements. Following the notation of Guarnieri and Livshits [14], we define the syntax of each statement $st \in St$ as follows.

$$\begin{array}{rll} st ::= & \epsilon & \mid \text{[empty]} \\ & st_1; st_2 & \mid \text{[sequence]} \\ & v = \texttt{new}\ v_0(v_1, \ldots, v_n) & \mid \text{[constructor]} \\ & v_1 = v_2 & \mid \text{[assignment]} \\ & v_1 = v_2.f & \mid \text{[load]} \\ & v_1.f = v_2 & \mid \text{[store]} \\ & m = \texttt{function}\ (v_1, \ldots, v_n)\ \{st;\} & \mid \text{[functionDecl]} \\ & v = m(v_1, \ldots, v_n) & \mid \text{[functionCall]} \\ & \texttt{return}\ v & \mid \text{[return]} \end{array}$$

Each statement st is either empty, an elementary statement, or a sequence of statements of the form $st_1; st_2$. An elementary statement can be an object construction, where v_0 is a constructor and $v_1, \ldots, v_n$ are its arguments; an assignment; a load of the object field $v_2.f$; a store to the object field $v_1.f$; a definition of a function; a call to a function; or a return from a function. Other complex statements may be transformed into a sequence of equivalent statements through preprocessing prior to applying our analysis.

3.3 Points-to Analysis

Points-to analysis is the process of determining whether a reference variable $v \in V$ can point to $o \in O$, a JavaScript object or HTML DOM element. As in the literature [36], we use V to denote the set of all reference variables defined in the program, O to denote the set of objects created at the set L of allocation sites, and F to denote the set of object fields. For each site $l_i \in L$, we map all objects created at l_i to a single abstract object $o_i \in O$. The points-to relation, denoted T_{ptsTo}, consists of a set of pairs of the form (v, o_i), meaning the reference variable $v \in V$ points to the object $o_i \in O$, and of the form $(o_i.f, o_j)$, meaning the field $f \in F$ of the object $o_i \in O$ points to the object $o_j \in O$.

We define an abstract transformer for each $st \in St$ as a function $f_{ptsTo} : T_{ptsTo} \times St \rightarrow T_{ptsTo}$, which takes a points-to relation $T \subseteq T_{ptsTo}$ as input and returns a new points-to relation $T' \subseteq T_{ptsTo}$ as output. For brevity, we provide definitions only for the following statements:

- Allocation: `l = new c`
- Assignment: `l = r`
- Store: `l.f = r`
- Load: `l = r.f`

For each of the above statements, the new points-to relation T' is defined with respect to the old points-to relation T as follows:

- Allocation: $T' = T \cup \{(l, o_i)\}$
- Assignment: $T' = T \cup \{(l, o_i) \mid (r, o_i) \in T\}$
- Store: $T' = T \cup \{(o_i.f, o_j) \mid (l, o_i) \in T \text{ and } (r, o_j) \in T\}$
- Load: $T' = T \cup \{(l, o_i) \mid (r, o_j) \in T \text{ and } (o_j.f, o_i) \in T\}$

For an allocation, we add (l, o_i) to the points-to relation. For an assignment, if the pair (r, o_i) is already in the points-to relation, we add (l, o_i) as well. For a store and a load, the abstract transformers are defined similarly.

3.4 Call-graph Construction

Although many of the function calls in JavaScript code can be resolved to a unique target function at the time of the static analysis, there are cases where the resolution has to be carried out at run time. In such cases, our analysis over-approximates the set of functions that may be called. We leverage the result of our points-to analysis to determine which function may be invoked. Specifically, consider the statement `l = v`$_0$`.m(v`$_1$`, ..., v`$_n$`)`, where $v_0 \in R$ is a reference variable, $m \in F$ is the field name, and $v_1, \ldots, v_n \in V$ are the actual parameters of the function call.

Let $m_i(p_0, p_1, \ldots, p_n, ret_j)$ be a function that $v_0.m$ may point to, where p_0 refers back to the object, ret_j refers to the return value, and $p_1, \ldots, p_n$ are the formal parameters. For each object that v_0 may point to, denoted $(v_0, o_0) \in T$, and for each function that $o_0.m$ may point to, denoted $(o_0.m, m_i) \in T$, we transform the function call to the following statements:

- `p`$_1$ `= v`$_1$`, ..., p`$_n$ `= v`$_n$`;`
- executing the code in $m_i()$; and
- `l =` ret_j`.`

The abstract transformer for the function call is defined as follows: $T' = T \cup \{\ (p_0, o_0), (p_1, o_1), \ldots, (p_n, o_n), (l, o_j)\ \}$, such that $(v_1, o_1) \in T, \ldots, (v_n, o_n) \in T$, and $(ret_j, o_j) \in T$.

3.5 Dependency Relations

For each statement $st \in St$, let $V_{RD}(st)$ be the set of memory locations read by st, and $V_{WR}(st)$ be the set of memory locations written to by st. We define the traditional control and data dependency relations [12] as follows: a data dependency, $\rightarrow_{data}$, exists between $st_1, st_2 \in St$ if st_1 is a write to some variable x and st_2 is a read of x. That is, $(st_1, st_2) \in \rightarrow_{data}$ if $V_{WR}(st_1) \cap V_{RD}(st_2) \neq \emptyset$. A control dependency, $\rightarrow_{ctrl}$, exists between $st_1, st_2 \in St$ if st_1 is a branch statement, st_2 is another statement, and the evaluation of the predicate p in st_1 determines the execution of st_2.

Since each JavaScript code block is executed atomically, we are concerned with the dependency relations between code blocks as opposed to individual statements. Let m_1 and m_2 be two JavaScript functions. We say $(m_1, m_2) \in \rightarrow_{ctrl}$ if executing m_1 may affect the control flow of m_2; that is, there exists $st_1 \in m_1$ and $st_2 \in m_2$ such that $(st_1, st_2) \in \rightarrow_{ctrl}$. Similarly, we say $(m_1, m_2) \in \rightarrow_{data}$ if $(st_1, st_2) \in \rightarrow_{data}$.

The DOM event dependency relation, in contrast, is defined directly over events. Intuitively, if the execution of some callback function m_1 of the event ev_1 affects the execution of some callback function m_2 of the event ev_2, there is a DOM event dependency between ev_1 and ev_2. More so, m_1 may affect m_2 through control/data dependencies; or, m_1 may affect m_2 by registering, removing, or modifying the callback functions of event ev_2, which includes m_2. This effect is unique to the event driven environment of client-side web applications. Formally:

DEFINITION 1. *Two events $ev_1, ev_2 \in EV$ are in the DOM event dependency relation, $(ev_1, ev_2) \in \rightarrow_{DOM}$, if there exists a callback function m_1 of ev_1 and a callback function m_2 of ev_2 such that,*

```
DomA.onclick(function() {
  c = true;
});
DomB.onclick(function() {
  if (c) {
    statement1;
  } else {
    statement2;
  }
});
```

Figure 5: Example: data/control dependencies in the DOM.

- *either* $(m_1, m_2) \in (\rightarrow_{data} \cup \rightarrow_{ctrl})^*$*, or*
- *executing* m_1 *registers, removes, or modifies the handler* m_2*.*

Here, T^* denotes the transitive closure of a relation T.

Consider the code in Figure 5 as an example. There are two functions registered as the *onclick* event handlers of `DomA` and `DomB`; `c` is a global variable used in the two event handlers. Inside the handler of `DomA`, there is an assignment to `c`. The value of `c` is used as the predicate of a branch in the event handler of `DomB`. So, clicking `DomA` affects the reachability of the statements guarded by the branch `if (c)`. Thus, `DomB` is DOM-event dependent on `DomA`.

4. CONSTRAINT-BASED DEPENDENCY ANALYSIS

In this section, we present our static analysis algorithm for computing DOM event dependencies.

4.1 Datalog-based Program Analysis

We follow the declarative program analysis framework pioneered by Whaley, Livshits, and Lam [25,28,42], where the analysis is represented by a Datalog program consisting of a set of *facts* and a set of *rules*. The facts are relations that hold in the program, and the rules specify the algorithm for deriving new relations from existing relations. For example, the first two lines below show the two facts describing a graph where n_1 is the parent of n_2, and n_2 is the parent of n_3. The next two lines define the rules to infer the ancestor relation: if X is the parent of Y, X is the ancestor of Y. Or if X is the parent of Z and Z is the ancestor of Y, X is the ancestor of Y.

```
parent(n1,n2)
parent(n2,n3)
ancestor(X,Y)   ← parent(X,Y)
ancestor(X,Y)   ← parent(X,Z),ancestor(Z,Y)
```

A Datalog engine takes the program above as input and computes the `ancestor` relation. Internally, it repeatedly applies the rules over the facts until reaching a fix-point. Then, one may query the Datalog engine to check, for example, if `ancestor(n1,n3)` holds.

4.2 Generating the Datalog Facts

We first *normalize* the JavaScript code to break down complex statements into series of simpler statements by adding auxiliary variables. Figure 6 shows an example of this. Then, we traverse the control flow graph (CFG) of the simplified code and, for each statement, generate its Datalog facts. Later on, these Datalog facts are merged with a predefined set of Datalog rules that specify our dependency analysis algorithm. Finally, we use a Datalog engine to solve the program to obtain the analysis results.

The Datalog facts generated from the input program populate the relations shown in Figure 7. The domains used in these relations are as follows: V, the set of variables; St, the set of statement IDs; O, the set of objects; F, the set of object fields; and $E = \{load, mouse, keyboard, timeout, ajax, other\}$, the set of event handler types. Next, we provide examples on this process.

Chained statement	Normalized form

`var a = document.images.length;`	`var temp0 = document.images;`
	 `var temp1 = temp0.length;`
	 `var a = temp1;`

Figure 6: Example for JavaScript code normalization.

ASSIGN($v_1 : V, v_2 : V, st : St$)	Variable Assignment: $v_1 = v_2$ with ID st
LOAD($v_1 : V, v_2 : V, f : F, st : St$)	Object field load: $v_1 = v_2.f$ with ID st
STORE($v_1 : V, f : F, v_2 : V, st : St$)	Object field store: $v_1.f = v_2$ with ID st
FUNCDECL(v : V, o : O)	v assigned function o: `v =function(){...}`
FORMAL($o : O, n : N, v : V$)	v is the n^{th} formal argument of function o
ACTUAL($st : St, n : N, v : V$)	v is the n^{th} argument in call-site at st
METHODRET($o : O, v : V$)	v is the return value of function o
CALLRET($st : St, v : V$)	Return into variable v at call-site st
STMT($st : St, o : O$)	st is a statement in function o
HEAP($v : V, o : O$)	Allocation of heap object o into variable v
PTSTO($v : V, o : O$)	Variable v points-to object o
DOM($o : O$)	o is a DOM object
DOM-MODIFY($o : O, e : E, f : O, st : St$)	Attach function f to object o's event e

Figure 7: The relations defined to specify our analysis.

Largely, the relations in Figure 7 correspond to various statements in the program, e.g., assignments, loads, and stores. Each statement then, for the most part, generates a corresponding input fact. Specifically, every statement in the program is identified with a unique ID $st \in St$. The formal arguments of a function are those used within the function itself, e.g., for function `f(a, b){...}`, `a` and `b` are the formal arguments. Given this function declaration, if $a, b \in V$ represent variables `a` and `b` and $f \in O$ represents the function `f()` then FORMAL$(f, 1, a)$ and FORMAL$(f, 2, b)$. At a call-site, `v = f(a1, b2)`, `a1` and `a2` are the actual arguments, e.g., ACTUAL$(s, 1, a_1)$ where s is the statement ID of the call-site and $a_1 \in V$ represents the variable `a1`. The 0th actual argument is the function object performing the call, e.g., ACTUAL$(s, 0, f)$.

Continuing the example, assume the statement `return r` is the return statement of the function `f()`. Let $r \in V$ represent `r` then METHODRET(f, r). Each call-site similarly has its own return value from a function. Using the previous call-site and let $v \in V$ represent the variable `v`, then we have CALLRET(s, v).

Next, we present examples for generating facts about DOM elements and operations.

DOM References We model DOM elements as heap objects and operations modifying DOM elements as those on heap objects. For a DOM element $o_d \in O$, we add an implicit heap allocation and the corresponding fact DOM(o_d) indicating that o_d is a DOM object. We treat alternate methods to access the same DOM element in a unified manner, e.g., `a = document.getElementById("model")` and `a = $("#model")[0]` can be used to access the same element. Let $o_m \in O$ be the DOM element with ID `"model"`, $v_g \in V$ be an auxiliary variable representing the value returned from the `getElementById()` call at the call-site. Then, we have DOM(o_m), and PTSTO(v_g, o_m) indicating that the result of the call points-to DOM object o_m. Furthermore, let $a \in V$ be the variable storing the returned value; we have ASSIGN(a, v_g, s) where s is the statement ID of the call. The use of `$("#model")[0]` can be handled similarly with an auxiliary variable.

We also handle the various ways of accessing attributes of DOM elements in the same way as reads/writes to objects. For example, using `a.setAttribute("value",x); y=a.getAttribute("value")` results in the facts: STORE(a, value, x) and LOAD(y, a, value).

DOM Listeners Prior works on statically analyzing JavaScript often do not accurately model the dynamic registration, triggering, and removal of event handlers. For example, Jensen et al. [20] abstract away the information on where in the DOM tree an event handler is registered. Furthermore, they assumed that *load* handlers always executed before other kinds of handlers; this may not be true. In contrast, we model such information more accurately.

We distinguish the different event categories: *load, keyboard, mouse, timeout, ajax*, and *other*. These correspond to event attributes such as `onkeydown` and `onclick`. We model both the static and the dynamic methods for registering and removing event handlers. For example, `<obj id="a1" onclick="scrpt">` statically installs the callback function `scrpt` to the *onclick* event of the DOM object `a1`. In contrast, one may dynamically modify a callback function using an explicit store, `tmp.onclick = scrpt`, or using an API, `tmp.addEventListener("click", scrpt)`. In all three cases, we generate the same fact: DOM-MODIFY(o_e, *mouse*, o_s, *st*) where $o_e \in O$ is the DOM element `a1`, $mouse \in E$ the type of event, $o_s \in O$ the `scrpt` function, and *st* the ID of the statement.

Timer related DOM APIs call functions after durations of time, e.g., `setTimeOut(func, t)` calls `func` after time `t`. We model timers with a DOM element $o_t \in O$ with event type $timeout$. The previous `setTimeOut()` becomes DOM-MODIFY$(o_t, timeout, o_f, st)$ where o_f is the object representing `func`, and *st* the statement ID.

Removing DOM event handers also uses DOM-MODIFY. Consider `o.removeEventListener("click",f)`, which removes `f` from `o`'s `"click"` event. We model it as DOM-MODIFY$(o, mouse, f, s)$ where $o \in O$ is the object representing `o`, f representing `f`, and s representing the statement ID. Essentially, the act of removing an event handler `f` may effect any of the event handlers which `f` may effect; this is the same as if `f` were installed, or removed. We provide more examples shortly in the next sub-section.

As seen in the previous examples, generating the input set of Datalog facts amounts to traversing each statement in the CFG and generating its corresponding fact. Thus, it is a linear-time process.

Our modeling of global objects, and the DOM elements in particular, is analogous to using a single `global` object and then modeling all reads/writes to JavaScript globals as loads and stores of fields of this global object.

DOM Aliasing There are three ways of handling DOM node aliasing: over-approximation, under-approximation, and precise modeling. Since precise modeling is expensive, we omit it from the discussion. Below is a summary of the other two approaches:

- To over-approximate aliasing, the simplest approach is to treat all elements in the DOM as a single abstract object [14, 15]. That is, reading from or writing to one DOM element will be regarded as potentially reading from or writing to any DOM element. A more accurate over-approximation is to group all DOM elements of the same type as a single abstract object [20]. That is, reading from or writing to an integer variable will be regarded as potentially reading from or writing to any integer variable. However, it will be distinguished from a non-integer variable.
- To under-approximate aliasing, the simplest approach is to assume that each access (read/write) is on a separate object [2]. That is, one can pretend that dependencies through the DOM don't exist.

These approaches represent two extreme cases, and therefore may not be accurate enough, but have the advantage of being scalable in practice. In this work, we focus on a conservative static analysis that uses the over-approximation.

4.3 Generating the Datalog Rules

Next, we introduce the rules that specify our DOM dependency analysis. They use the existing facts to infer new relationships. For ease of understanding, we divide these rules into two subsets. The first subset is for the points-to analysis, and the second subset is for the dependency analysis. These analyses are interleaved in our implementation.

4.3.1 Rules for the Points-To Analysis

Our rules for the points-to analysis are shown in Figure 8. They implement a flow-insensitive and context-insensitive analysis following Guarnieri and Livshits [14]. The main difference is that we encode the locations of assign, store, and load operations, which will be crucial in computing the DOM event dependencies.

PTSTO(v_1, o_1)	←	HEAP(v_1, o_1)
PTSTO(v_1, o_1)	←	FUNCDECL(v_1, o_1)
PTSTO(v_1, o_1)	←	PTSTO(v_2, o_1), ASSIGN(v_1, v_2, st)
HEAPPTSTO(o_1, f, o_2)	←	STORE(v_1, f, v_2, st), PTSTO(v_1, o_1), PTSTO(v_2, o_2)
PTSTO(v_1, o_1)	←	LOAD(v_1, v_2, f, st), PTSTO(v_2, o_2), HEAPPTSTO(o_2, f, o_1)
CALLS(o, st)	←	ACTUAL(st, 0, v_1), PTSTO(v_1, o)
ASSIGN(v_1, v_2, st_1)	←	CALLS(o_1, st_1), FORMAL(o_1, n_1, v_1), ACTUAL(st_1, n_1, v_2)
ASSIGN(v_1, v_2, st_1)	←	CALLS(o_1, st_1), METHODRET(o_1, v_2), CALLRET(st_1, v_1)

Figure 8: Datalog rules for specifying the points-to analysis.

Based on the points-to analysis, we can proceed to compute the dependency relations between operations on the DOM elements, including DOM reference, DOM read, and DOM write operations.

4.3.2 Rules for the Dependency Analysis

First, we compute the traditional data dependency relation as in Figure 9. Here, $v_1, v_2 \in V$ are reference variables, $st_1, st_2 \in St$ are statement IDs, and $o_1, o_2, o_3, f \in O$ are heap objects.

WRITE1(v_1, st_1)	←	ASSIGN(v_1, v_2, st_1)
WRITE1(v_1, st_1)	←	LOAD(v_1, v_2, f, st_1)
WRITE2(v_1, f, st_1)	←	STORE(v_1, f, v_2, st_1)
READ1(v_2, st_1)	←	ASSIGN(v_1, v_2, st_1)
READ1(v_2, st_1)	←	STORE(v_1, f, v_2, st_1)
READ2(v_2, f, st_1)	←	LOAD(v_1, v_2, f, st_1)
DATA-DEP(st_1, st_2)	←	READ1(v_1, st_2), WRITE1(v_1, st_1)
DATA-DEP(st_1, st_2)	←	READ2(v_1, f, st_2), WRITE1(v_1, st_1)
DATA-DEP(st_1, st_2)	←	READ1(v_1, st_2), WRITE2(v_2, f, st_1)
		PTSTO(v_1, o_1), PTSTO(v_2, o_1)
DATA-DEP(st_1, st_2)	←	READ2(v_1, f, st_2), WRITE2(v_2, f, st_1)
		PTSTO(v_1, o_1), PTSTO(v_2, o_1)
DATA-DEP(st_1, st_3)	←	DATA-DEP(st_1, st_2), DATA-DEP(st_2, st_3)
CALL-EDGE(o_2, o_1)	←	CALLS(o_1, st_1), STMT(st_1, o_2)
CALL-EDGE(o_1, o_3)	←	CALL-EDGE(o_1, o_2), CALL-EDGE(o_2, o_3)

Figure 9: Datalog rules for the data dependency analysis.

To model the DATA-DEP relation, we use auxiliary relations WRITE1, WRITE2, READ1, READ2 to represent the writes and reads of variables/fields of objects. They correspond to the first six rules of Figure 9. Given the auxiliary read and write relations, we consider two statements to be data dependent, DATA-DEP(st_1, st_2), if there is a read at st_2 and a write to the same variable(s) at st_1. The first two rules are for data dependencies through variables; the next two rules are for data-dependencies through aliasing objects; and the fifth rule is for the transitivity of data-dependencies.

For two functions $o_1, o_2 \in O$, CALL-EDGE(o_2, o_1) if the function o_1 is called in function o_2; this is specified in the second to last rule in Figure 9. The last rule says that the relation is transitive – it represents edges in the call-graph.

To compute the control dependencies, we implement the algorithm of Cytron et al. [9] on the JavaScript CFG to generate the CTRL-DEP(st_1, st_2) relation, meaning that st_2 is control-dependent on st_1. The corresponding Datalog rules are omitted for brevity.

Figure 10 shows the rules for computing the DOM event dependency relation. Here, $m_1, m_2, m_3 \in O$ are function objects, $o_1, o_2 \in O$ are DOM objects, $st_1, st_2 \in St$ are statement IDs, and $e_1, e_2 \in E$ are DOM event types. First, we create the program-dependence relation [12], PROG-DEP, i.e., the transitive closure of the control- and data-dependencies. Then, we leverage PROG-DEP to create the FUNC-DEP relation representing dependencies across

PROG-DEP(st_1, st_2)	$\leftarrow$	DATA-DEP(st_1, st_2)
PROG-DEP(st_1, st_2)	$\leftarrow$	CTRL-DEP(st_1, st_2)
PROG-DEP(st_1, st_3)	$\leftarrow$	PROG-DEP(st_1, st_2)
		PROG-DEP(st_2, st_3)
FUNC-DEP(m_1, m_2)	$\leftarrow$	PROG-DEP(st_1, st_2)
		STMT(st_1, m_1)
		STMT(st_2, m_2)
DOM-PROG-DEP(o_1, e_1, o_2, e_2)	$\leftarrow$	DOM-MODIFY(o_1, e_1, m_1, st_1)
		DOM-MODIFY(o_2, e_2, m_2, st_2)
		FUNC-DEP(m_1, m_2)
DOM-MODIFY-DEP(o_1, e_1, o_2, e_2)	$\leftarrow$	DOM-MODIFY(o_1, e_1, m_1, st_1)
		DOM-MODIFY(o_2, e_2, m_2, st_2)
		STMT(st_2, m_1)
DOM-MODIFY-DEP(o_1, e_1, o_2, e_2)	$\leftarrow$	DOM-MODIFY(o_1, e_1, m_1, st_1)
		CALL-EDGE(m_1, m_3)
		DOM-MODIFY(o_2, e_2, m_2, st_2)
		STMT(st_2, m_3)
DOM-DEP(o_1, e_1, o_2, e_2)	$\leftarrow$	DOM-MODIFY-DEP(o_1, e_1, o_2, e_2)
DOM-DEP(o_1, e_1, o_2, e_2)	$\leftarrow$	DOM-PROG-DEP(o_1, e_1, o_2, e_2)

Figure 10: Datalog rules for the DOM dependency analysis.

functions. Finally, we consider the two cases for DOM event dependencies: those through program dependencies, and those involving event handler modifications.

The relation DOM-PROG-DEP captures the first case, where a program-dependency exists between two functions called from the DOM event handlers. Specifically, let two DOM objects $o_1, o_2 \in O$ have event handlers m_1 and m_2 attached to their events of type e_1 and e_2, respectively. If m_2 is dependent on m_1, we say that DOM-PROG-DEP(o_1, e_1, o_2, e_1), i.e., there is a DOM event dependency between o_2's handler of type e_2 and o_1's handler of type e_1.

The relation DOM-MODIFY-DEP captures the second case, when the event handler of one DOM object installs/removes/modifies the event handler of another DOM object. The first DOM-MODIFY-DEP rule captures the simplest case: there is a function m_1 which is an associated event handler of DOM object o_1's event of type e_1. Also, there is a DOM event handler add/remove/modification at statement st_2 where st_2 is in function m_1. Because there is a DOM modification of o_2's event e_2 in m_1 (at statement st_2) we say that o_2's event e_2 is dependent on o_1's event e_1: DOM-MODIFY-DEP(o_1, e_1, o_2, e_2).

The next DOM-MODIFY-DEP rule is similar but captures the case where a DOM event handler calls a function that modifies a DOM object's event handler. Specifically, there is a function m_1 registered to DOM object o_1's event e_1, and there is a call from m_1 to some function m_3, which has a DOM modification of object o_2's event e_2 at statement st_2. Since m_1 transitively affects DOM object o_2's event e_2 through m_3, we say: DOM-MODIFY-DEP(o_1, e_1, o_2, e_2). Recall that CALL-EDGE is defined as the transitive closure of function calls; it captures some DOM event-handler calling an arbitrary sequence of function calls leading to a DOM modification.

Finally, the DOM event dependency relation, DOM-DEP, is the combination of DOM-PROG-DEP and DOM-MODIFY-DEP.

Since we focus on the over-approximated analysis, we deal with event propagations (capturing and bubbling) and AJAX callbacks conservatively. Recall that how the web browser propagate events through the HTML DOM tree may affect the control flow of the JavaScript code in a web application. When *capturing* is enabled, the parent element captures the event first and then passes it down to the children. In contrast, when *bubbling* is enabled, the target element captures the event first before passing it up to the parent elements. For efficiency reasons, distinguishing these two cases in a static analysis is difficult. Therefore, we conservatively assume that all JavaScript functions in the application may be executed in any order. This approximation also works for modeling the execution of asynchronous callbacks of the AJAX requests.

5. IMPROVING AUTOMATED TESTING

In this section, we leverage the static DOM event dependency analysis to improve ARTEMIS [5], a popular automated tester of client-side web applications. Since ARTEMIS generates event sequences randomly (like RANDOOP [34]), it often lacks the ability to reach high statement coverage. During our experiments, for example, the default algorithm in ARTEMIS could not reach more than 65% coverage even after 500 iterations. In contrast, leveraging our static DOM event dependency analysis enabled ARTEMIS to quickly reach 80% coverage.

5.1 The Default Algorithm of Artemis

Algorithm 1 shows the default test input generation procedure in ARTEMIS. It takes the initial test $\langle u_0, S_0, \rho_0 \rangle$ as input and returns a set *Results* of explored tests as output. Here, a test input is defined as a tuple $\langle u, S_0, \rho \rangle$ where u is the URL of the web page, S_0 is the initial state of the application, and $\rho = ev_1 \dots ev_n$ is a sequence of events. An *event* $ev = \langle param, state, env \rangle$ captures not only activities performed by the user, but also timer responses and AJAX callbacks. Here, *param* denotes the values of the event parameters, *state* denotes the values of the HTML form fields, and *env* denotes the values of environment parameters, such as the window size and time of day. Line 18 shows our new pruning method: leveraging the DOM event dependencies to skip redundant sequences.

Algorithm 1 Test sequence generation algorithm in ARTEMIS.

```
   Initially: $Worklist := \{\ \}$; run ARTEMIS($u_0$, $S_0$, $\rho_0$).

 1: ARTEMIS(URL $u_0$, State $S_0$, Sequence $\rho_0$) {
 2:     $Results := \{\ \}$;
 3:     $Worklist := \{\langle u_0, S_0, \rho_0 \rangle\}$;
 4:     while ($Worklist \neq \emptyset$ and $\neg timeout$ and $\neg maxruns$)
 5:         $c = \langle u, S, \rho \rangle = Worklist.removeNext()$;
 6:         $S'$ := EXECUTEAPPLICATION($c$);
 7:         $Results := Results \cup \{(c, S')\}$;
 8:         //make test inputs by modifying the last event in $\rho$
 9:         foreach (variant $ev'_n$ of $ev_n$ in $\rho = ev_1 \dots ev_n$) {
10:             $\rho' := ev_1 \dots ev_{n-1} \cdot ev'_n$;
11:             $Worklist := Worklist \cup \{\langle u, S, \rho' \rangle\}$
12:         }
13:         //make test inputs by extending $\rho$ with a new event
14:         if ($S' \notin VisitedStates$) {
15:             $VisitedStates.add(S')$;
16:             foreach ($ev'_{n+1}$ enabled at $S'$) {
17:                 $\rho'' := \rho \cdot ev'_{n+1}$;
18:                 if ($\neg$ ISREDUNDANT($u$, $S$, $\rho''$))
19:                     $Worklist := Worklist \cup \{\langle u, S, \rho'' \rangle\}$
20:             }
21:         }
22:     }
23:     return $Results$;
24: }
```

ARTEMIS starts with an empty set *Results* of tests and a worklist consisting of only $\langle u_0, S_0, \rho_0 \rangle$. Then, it loads the web page from u_0 with initial state S_0 and executes the sequence ρ_0 of events. Let S' be the application state after applying these events. Next, it generates new event sequences using one of the following methods. The first method is to generate a variant ev'_n of the last event ev_n in the sequence $\rho = ev_1 \dots ev_n$; this creates a new sequence $\rho' = ev_1 \dots ev_{n-1} ev'_n$ (Lines 8–12). In this case, $ev'_n = \langle param'$, $state', env' \rangle$ will have the same event type as ev_n but different values for the event parameters, form fields, and environmental parameters; meaning ρ' may lead to a different program state.

The second method for generating a new event sequence is to append a new event ev_{n+1} to the end of ρ to create the new sequence ρ'' (Lines 13–21). In this case, the main problem is that the default algorithm in ARTEMIS never checks whether ρ'' is redundant, i.e., whether ρ'' is equivalent to some event sequence(s) that have already been explored. In contrast, our new method will perform such a check. As shown in Line 18, if ρ'' is proved to be redundant by this newly added check, it will not be added to $Worklist$.

5.2 Pruning Redundant Event Sequences

Algorithm 2 shows the pseudocode of IsREDUNDANT() used at Line 18 in Algorithm 1. Inside IsREDUNDANT(), the theoretical foundation for deciding whether ρ'' is redundant is partial order reduction (POR [13,23,43]). We say that two sequences ρ_1 and ρ_2 are equivalent if we can transform one sequence into the other by repeatedly swapping *adjacent* and *independent* events. Two events ev_1 and ev_2 are adjacent if they occur consecutively. They are *dependent* if the two events access the same object and at least one of them is a write (modifying the content of the object); we say they are *independent* if the two events are not dependent on each other.

Algorithm 2 Checking if the sequence ρ is redundant.

```
1: IsRedundant(URL $u$, State $S$, Sequence $\rho$) { //Default in Artemis
2:     return false;
3: }
4: IsRedundant(URL $u$, State $S$, Sequence $\rho$) { //Our New Method
5:     Let $\rho = ev_1 \ldots ev_n \cdot ev_{n+1}$;
6:     if $(ev_n \not\to_{DOM} ev_{n+1} \wedge ev_{n+1} \not\to_{DOM} ev_n \wedge ev_n \not<_{lex} ev_{n+1})$
7:         return true;
8:     return false;
9: }
```

Consider $\rho_1 = ev_1 \ldots ev_i ev_j \ldots ev_n$, where ev_i and ev_j are independent. Since swapping the order of ev_i and ev_j does not change the behavior of the application (they are commutative), we know $\rho_2 = ev_1 \ldots ev_j ev_i \ldots ev_n$ triggers the same behavior as ρ_1. Therefore, ρ_1 and ρ_2 are equivalent. During testing, if ARTEMIS has already explored ρ_1, then we can safely skip ρ_2, since it suffices to test one representative from each equivalence class of sequences.

In Algorithm 2, the pruning of equivalent sequences is implemented using a form of the sleep-set based reduction [21,40]. Toward this end, we assign the events of the application a lexical order, $<_{lex}$. When two adjacent events ev_n and ev_{n+1} satisfy the following conditions:

- (1) $(ev_n \not\to_{DOM} ev_{n+1}) \wedge (ev_{n+1} \not\to_{DOM} ev_n)$, meaning they are independent with each other, and
- (2) $ev_n <_{lex} ev_{n+1}$,

we choose to explore the sequence $\ldots ev_{n+1} ev_n \ldots$ while skipping the sequence $\ldots ev_n ev_{n+1} \ldots$. As shown in Line 6 of Algorithm 2, we use the result of our DOM event dependency analysis to check whether the two events are independent from each other.

5.3 The Running Example

Consider the application in Figure 2, whose DOM event dependencies are shown in Figure 3. Since the *click* event of `test1` is independent with itself, we skip the test sequence `test1` $\to$ `test1` ... as shown by the gray path on the left side of Figure 4 (b). Also, since the *click* event of `test1` is independent with the *click* of `test2`, we explore `test1` $\to$ `test2` but skip `test2` $\to$ `test1`; we also skip the subsequence `test2` $\to$ `test1` $\to$ `test2`. Similarly, we skip all the other gray sequences in Figure 4 (b). Therefore, up to depth 3, our new method can reduce the total number of test sequences generated by ARTEMIS from 49 down to 14.

6. EXPERIMENTS

We implemented the new dependency analysis in a software tool named JSDEP. It uses ESPRIMA for parsing and normalizing the JavaScript code, JS-WALA for constructing the control flow graph, and the μZ fix-point engine [17] in Z3 [10] for solving the Datalog

Table 1: Results of the static DOM-event dependency analysis.

Name	LOC	Total Deps.	Calculated Deps.	Constraints	Time (s)
case1	59	16	2	166	0.11
case2	72	16	3	187	0.11
case3	165	36	6	517	0.15
case4	196	64	8	618	0.16
frog	567	361	264	2,398	4.34
cosmos	363	169	144	1,000	0.20
hanoi	246	576	324	1,026	0.23
flipflop	525	36	25	2,445	0.34
sokoban	3,056	361	256	2,116	0.35
wormy	570	81	64	3,683	0.42
chinabox	338	49	16	1,281	0.63
3dmodel	5,414	25	19	3,813	13.83
cubuild	1,014	36	25	5,684	6.83
pearlski	960	144	100	4,129	7.17
speedyeater	784	361	64	4,170	0.61
gallony	300	196	72	1,372	0.25
fullhouse	528	64	49	1,007	0.20
ball_pool	1,745	81	30	1,709	0.28
lady	820	121	81	4,564	7.88
harehound	468	529	168	1,976	1.53
match	369	576	400	6,385	4.49
Total	18,559	3,898	2,120	50,246	50.11

program. To demonstrate the usefulness of the analysis we applied it to improve the performance of ARTEMIS [5], a state-of-the-art web application testing tool.

Our experiments were designed to answer two questions:

- Can JSDEP compute the DOM event dependency relations with reasonable accuracy at negligible run-time cost?
- Can JSDEP help ARTEMIS reach a higher testing coverage than the default algorithm?

We evaluated JSDEP on a number of client-side web applications. Our benchmarks fall into two groups. The first are four variants of Figure 2, case1 to case4, with four to eight buttons. The second are seventeen real web applications, ranging from hundreds to thousands of lines of code. Two are from ARTEMIS's benchmarks [5] (ball_pool and 3dmodel). The rest are JavaScript-based games [1]. In total, there are 21 benchmark applications with 18,559 lines of code total. We ran all experiments on a computer with an Intel Quad-Core i5-4440 3.10 GHz CPU with 12 GB of RAM.

6.1 Results: Dependency Analysis

Table 1 shows the result of the static DOM event dependency analysis. Columns 1–2 show the name of the benchmark program and the number of lines of code. Column 3 shows the maximum number of possible DOM event dependencies, i.e., N^2 where N is the number of DOM events in the application. Conceptually, this is the dependency relation used by default in ARTEMIS: every DOM event is dependent on every DOM event. Column 4, in contrast, shows the number of DOM event dependencies found by our analysis. Columns 5–6 show the statistics of our analysis: the size of the Datalog program, and the time spent on the analysis. The time includes parsing, normalizing, and transforming the code, generating the Datalog program, and calling μZ to solve the program.

Overall, our analysis can very quickly generate DOM event dependency results: the time ranges from 0.5 to 13 seconds. The total time spent on analyzing the 21 benchmarks is less than 1 minute. Also, the results are much better than the theoretical worst case. Next, we show our analysis results are useful: they can significantly improve the performance of ARTEMIS.

6.2 Results: Improving Artemis

Table 2 shows the results of running ARTEMIS with and without leveraging JSDEP. Column 1 shows the benchmark's name. Columns 2–4 show statement coverage after running ARTEMIS and

ARTEMIS+JSDEP for 100 iterations. Columns 5–7, 8–10, 11–13, and 14–16 show the statement coverages achieved by running them up to 200, 300, 400, and 500 iterations, respectively. ARTEMIS is the algorithm as described by Artzi et al. [5] with their best prioritization technique enabled. Each iteration executes one test sequence, i.e., an iteration of the loop in Figure 1.

Overall, the statement coverage of ARTEMIS+JSDEP is 10–16% higher than ARTEMIS. For case1 and case2, in particular, the default ARTEMIS algorithm cannot reach 100% coverage even after 500 iterations. ARTEMIS+JSDEP can reach 100% coverage within only 100 iterations. Furthermore, as the number of iterations increases the average coverage of ARTEMIS remains stuck at 65%. But, the average coverage of ARTEMIS+JSDEP keeps increasing. This is because the default algorithm of ARTEMIS explores many redundant test sequences. Our static analysis results are accurate enough to skip many of these sequences and focus on useful tests.

There are some cases where ARTEMIS+JSDEP temporarily had lower coverage than ARTEMIS (e.g., *match* at 100 iterations). We believe this is mainly due to the inherent randomness of selecting items from the worklist. However, as the number of iterations increases, ARTEMIS+JSDEP becomes much better.

6.3 Results: Redundancy Removal

Next, we investigated how many sequences ARTEMIS+JSDEP deemed redundant. Table 3 summarizes the results. We ran each benchmark for 500 iterations and counted both the number of sequences generated (Column 3) and the number of sequences we found redundant (Column 4). Without using our method, all the redundant sequences would have been added to the worklist. Overall, we reduce the number of sequences added to the worklist by 36% on average. Examining the dependency results (Table 1), we can see our analysis actually finds 46% of the DOM events independent on average. The difference in the number of reduced sequences and the actual number of independent DOM events comes from the sleep-set approach of Algorithm 2: it does not guarantee to test *only* one sequences from each equivalence class. This is a limitation of our POR implementation and not the static analysis. Note: Column 3 counts the total number of sequences added to the worklist; only 500 of these were actually executed.

In addition to running ARTEMIS and ARTEMIS+JSDEP for a fixed number of iterations we also ran them for a fixed amount of time. Table 4 shows the result. Here, Columns 1–2 show the benchmark name and execution time. Columns 3–4 show the number of iterations and statement coverage obtained by ARTEMIS. Columns 5–6 show the number of iterations and statement coverage obtained by ARTEMIS+JSDEP.

The runtime here, and in all tests, includes the static analysis overhead and the overhead in ARTEMIS to perform the dependency check. So, ARTEMIS+JSDEP explores slightly fewer (92%) iterations on average within the 10 minute bound. Also, the number of iterations explored within the bound depends on the length of the tested sequences; this depends on the length of the sequences skipped. But, we still see a significant increase in the average of the statement coverage: from 67% achieved by ARTEMIS to 80% achieved by ARTEMIS+JSDEP. Overall, this indicates the default ARTEMIS search strategy spends much time on redundant tests.

Threat to Validity. Although our static dependency analysis is designed to be sound in the absence of JavaScript's reflexive features, there is no theoretical guarantee of its soundness. However, this is consistent with the norm of the research field. As the authors of [19, 35] have argued, due to the impact of JavaScript's dynamic features, it is impossible to develop a truly sound and, at the same time, practically useful static analysis framework. Thus, in practice, software tools strive for achieving *soundiness* [27] as opposed to achieving *soundness*. The goal is being as sound as possible without significantly compromising precision and scalability. Since we focus on improving the test coverage of ARTEMIS, as opposed to proving properties, achieving *soundiness* is sufficient.

7. RELATED WORK

Datalog-based program analysis was pioneered by Whaley and Lam [42]. They introduced a framework for implementing points-to analyses as database queries [25]. Livshits and Lam [28] and Naik et al. [32] used similar techniques in detecting security errors and data-races. Bravenboer and Smaragdakis [6] also formulated a points-to analysis as database queries. Kusano and Wang [24] used Datalog engines to analyze interference in multithreaded programs. However, these existing methods were all designed for analyzing programs written in more static languages such as Java and C++.

Although there are works on applying Datalog-based program analysis to JavaScript, none of them can handle DOM event dependencies that are crucial for client-side web applications. Specifically, Guarnieri and Livshits [14] implemented a Datalog-based analysis procedure in the GATEKEEPER tool; but, the goal was statically enforcing security policies. Zheng et al. [44] developed a method for modeling AJAX APIs to check possible bugs when there are asynchronous requests. Meyerovich and Livshits [31] also developed a method for enforcing security policies in the browser. Jensen et al. [20] proposed a type inference algorithm for JavaScript based web applications, which tracks DOM elements and browser APIs based on their IDs and types but did not compute the dependency relations. Feldthaus et al. [11] proposed a method for constructing approximate call graphs but completely ignored their interactions with the DOM. Madsen et al. [29] proposed a static analysis procedure that can infer the behavior of framework APIs but it targeted JavaScript-based applications in Windows 8 only. Madsen et al. [30] developed a static analysis procedure for the event-driven *Node.js* applications but they were server-side applications as opposed to client-side applications.

The methods proposed by Arlt et al. [4] and Cheng et al. [7] for testing Java-based GUI applications are significantly different in that they do not model the registration, modification, and removal of event handlers (the focus of our work). Instead, they assumes that all event handlers are pre-installed, and thus focuses on analyzing only data dependencies between these handlers. This assumption may be reasonable for some Java-based GUI frameworks, but is not valid for JavaScript-based web applications.

There is also a large body of work on pointer analysis, flow analysis, and type inference for JavaScript, which are not based on Datalog and are not designed specifically for analyzing interactions with the HTML DOM. For example, Chugh et al. [8] proposed a staged information flow analysis for JavaScript to detect certain security violations in client-side code. Sridharan et al. [39] proposed a technique called correlation tracking to improve points-to analysis. Guha et al. [15, 16] proposed a static flow analysis for detecting AJAX intrusions, and typing local control and state. Wei and Ryder [41] developed a set of blended analysis tools, using both dynamic and static analyses to improve the points-to analysis. Andreasen and Møller [3] extended the *TAJS* analysis framework by adding a static dataflow analysis to infer and exploit determinacy information; this improves the type inference and call-graph construction for JavaScript programs using *jQuery*. *TAJS* itself builds upon the classic monotone framework of Kam and Ullman [22] using a specialized analysis lattice structure. Alimadadi et al. [2] developed a change impact analysis capturing the interplay between the JavaScript code changes and the HTML DOM, but their method is valid only for the given dynamic execution, whereas our method is static and therefore valid for all possible executions.

Table 2: Results of comparing ARTEMIS and ARTEMIS+JSDEP with a fixed number of iterations.

		Artemis	Art.+JSdep		Artemis	Artemis+JSdep		Artemis	Art.+JSdep		Artemis	Art.+JSdep		Artemis	Art.+JSdep
Name	Iter.	Cov.(%)	Cov.(%)	Iter.	Cov.(%)	Cov.(%)	Iter.	Cov.(%)	Cov.(%)	Iter.	Cov.(%)	Cov.(%)	Iter.	Cov.(%)	Cov.(%)
case1	100	70.59	100	200	70.59	100	300	70.59	100	400	70.59	100	500	70.59	100
case2	100	43.90	100	200	43.90	100	300	43.90	100	400	43.90	100	500	43.90	100
case3	100	38.10	62.86	200	38.10	74.29	300	38.10	74.29	400	38.10	95.24	500	38.10	95.24
case4	100	47.12	59.62	200	47.12	72.12	300	47.12	72.12	400	47.12	72.12	500	47.12	72.12
frog	100	86.79	89.29	200	88.93	96.43	300	88.93	96.43	400	88.93	96.79	500	88.93	97.86
cosmos	100	57.48	72.44	200	57.48	77.17	300	57.48	77.17	400	57.48	77.95	500	57.48	77.95
hanoi	100	77.19	76.32	200	77.19	76.32	300	77.19	82.46	400	77.19	82.46	500	77.19	82.46
flipflop	100	97.05	95.94	200	97.05	97.05	300	97.05	97.05	400	97.05	97.05	500	97.05	97.05
sokoban	100	73.09	76.46	200	73.09	76.46	300	73.09	76.46	400	73.09	76.46	500	73.09	76.46
wormy	100	39.76	40.95	200	39.76	40.95	300	39.76	40.95	400	39.76	40.95	500	39.76	40.95
chinabox	100	79.88	82.32	200	79.88	83.54	300	79.88	84.15	400	79.88	84.15	500	79.88	84.15
3dmodel	100	64.01	71.50	200	64.01	71.50	300	64.01	71.50	400	64.01	71.98	500	64.01	71.98
cubuild	100	61.30	68.15	200	61.30	73.46	300	61.30	78.42	400	61.30	85.79	500	61.30	85.79
pearlski	100	52.52	52.72	200	52.52	53.72	300	52.52	53.72	400	52.52	53.92	500	52.52	56.54
speedyeater	100	45.93	46.41	200	45.93	53.11	300	45.93	53.35	400	45.93	53.35	500	45.93	54.78
gallony	100	69.86	93.15	200	69.86	94.52	300	69.86	94.52	400	69.86	94.52	500	69.86	94.52
fullhouse	100	77.38	83.33	200	77.38	83.33	300	77.38	83.33	400	77.38	87.50	500	77.38	87.50
ball_pool	100	71.43	89.75	200	73.16	91.24	300	73.16	93.09	400	73.16	93.43	500	73.16	93.43
lady	100	76.13	77.25	200	76.13	79.50	300	76.13	79.50	400	76.13	79.50	500	76.13	79.50
harehound	100	80.28	88.07	200	80.28	91.28	300	80.28	91.28	400	80.28	92.20	500	80.28	92.20
match	100	61.45	50.28	200	61.45	62.01	300	61.45	73.18	400	61.45	73.18	500	61.45	73.18
Average	100	65.29	75.08	200	65.48	78.47	300	65.48	79.66	400	65.48	81.35	500	65.48	81.60

Table 3: Results of blocked sequence ratio (step 500).

Name	Iter.	Redundancy Checked	Redundancy Found	Ratio (%)
case1	500	1,001	499	49.85
case2	500	1,832	1,326	72.38
case3	500	4,436	3,232	72.86
case4	500	4,009	2,976	74.23
frog	500	9,501	1,895	19.95
cosmos	500	6,501	500	7.69
hanoi	500	11,501	2,015	17.52
flipflop	500	9,001	8,145	90.49
sokoban	500	9,501	1,830	19.26
wormy	500	3,033	443	14.61
chinabox	500	3,501	1,709	48.81
3dmodel	500	2,949	708	24.01
cubuild	500	3,001	768	25.59
pearlski	500	6,001	749	12.48
speedyeater	500	12,501	1,848	14.78
gallony	500	7,001	3,393	48.46
fullhouse	500	14,001	499	3.56
ball_pool	500	4,501	2,206	49.01
lady	500	5,001	290	5.80
harehound	500	11,501	4,771	41.48
match	500	12,384	5,502	44.43
Average		6,793	2,157	36.05

Table 4: Comparing ARTEMIS with ARTEMIS+JSDEP.

		Artemis		Artemis+JSdep	
Name	Time (s)	Iter.	Coverage (%)	Iter.	Coverage (%)
case1	600	5,819	70.59	1,972	100
case2	600	5,018	43.90	4,208	100
case3	600	4,292	38.10	7,090	100
case4	600	3,995	47.12	4,532	72.12
frog	600	1,656	88.21	96	84.64
cosmos	600	1,663	57.48	1,123	78.74
hanoi	600	2,782	77.19	1,884	82.46
flipflop	600	771	97.05	459	97.05
sokoban	600	1,225	73.09	264	76.68
wormy	600	1,179	52.23	538	40.95
chinabox	600	736	79.88	174	84.15
3dmodel	600	137	64.01	132	71.98
cubuild	600	661	61.30	242	75.51
pearlski	600	1,257	53.32	322	53.72
speedyeater	600	2,688	77.27	2,735	78.47
gallony	600	3,756	69.86	4,596	94.52
fullhouse	600	2,372	77.38	1,107	88.10
ball_pool	600	36	71.43	34	74.19
lady	600	64	75.90	55	76.58
harehound	600	2,383	80.28	2,305	94.50
match	600	2,462	62.01	7,444	73.18
Average	600	2,140	67.50	1,967	80.83

In addition to improving the performance of ARTEMIS, the result of our static DOM event dependency analysis may be used to improve a wide range of dynamic analysis tools. For example, the SYMJS tool of Li et al. [26] relies on symbolic execution to generate test inputs for JavaScript based web applications, but does not leverage the result of any static dependency analysis procedure. The KUDZU tool of Saxena et al. [37] uses a virtual machine based symbolic execution procedure to analyze client-side JavaScript code injection. The JALANGI tool developed by Sen et al. [38] provides a generic framework for implementing dynamic analysis techniques for JavaScript, e.g., concolic testing, but lacks the capability of conducting static program analysis. Nguyen et al. [33] proposed a delta-debugging based method for reducing the redundant parts of a test case generated by a symbolic execution tool. Jensen et al. [18] developed a stateless model checking tool for systematic testing of event-driven applications with a fixed data input. However, these methods focus on dynamic analysis, whereas our work focuses on static analysis and therefore is complementary.

8. CONCLUSIONS

We have presented a constraint-based method for statically computing DOM event dependencies in client-side web applications by formulating the static analysis as a Datalog program. We have also presented a method for leveraging the result of our dependency analysis to improve the performance of a popular web application testing tool named ARTEMIS. We have implemented our methods and evaluated them on real web applications. Our experiments show that the new methods can compute DOM event dependencies with reasonable accuracy and at a negligible cost. Furthermore, they allow ARTEMIS to significantly reduce test sequence redundancies and therefore improve the test coverage.

9. ACKNOWLEDGMENTS

This work was primarily supported by the NSF under grants CCF-1149454, CCF-1405697, and CCF-1500024. Partial support was provided by the ONR under grant N00014-13-1-0527.

10. ARTIFACT DESCRIPTION

Our artifact includes two parts: JSDEP, the new tool for static DOM event dependency analysis, and the modified ARTEMIS for automated testing of client-side web applications.

- The performance of JSDEP can be evaluated by applying it to the JavaScript benchmarks included in the artifact. The results have been shown in Table 1.
- The performance of ARTEMIS, with and without the performance improvement provided by JSDEP, can be evaluated on the web application benchmarks included in the artifact. The results have been shown in Tables 2, 3 and 4.
- The artifact includes scripts for running the individual analysis procedures provided by JSDEP and ARTEMIS, as well as generating the experimental result tables.
- The artifact also includes the raw data of experiments that we conducted to produce the result tables used in this paper.

The overall structure of the artifact is shown in Figure 11. Here, the block labeled *DOM Analysis* refers to the static analysis performed by JSDEP, and the block labeled *Modified Artemis* refers to the web application testing tool.

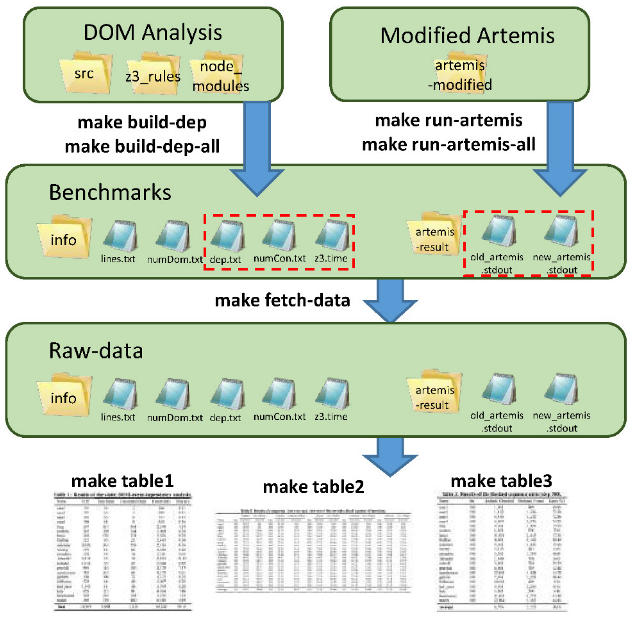


Figure 11: Structure of the artifact.

The tools, benchmarks, and experimental data are publicly available. The URL is `https://github.com/sch8906/JSdep`.

10.1 Installation

JSDEP has been developed and tested on a popular Linux platform: Ubuntu 12.04 Desktop 64-bit. Internally, JSDEP builds upon *Node.js* and the *Z3* SMT solver [10]. Therefore, prior to running JSDEP, both *Node.js* and *Z3* must be installed.

- To install *Node.js* in Ubuntu Linux, run `sudo apt-get install nodejs`.
- To install the *Z3* SMT solver, visit its official website and follow the directions from there.

To install our modified version of ARTEMIS, please

- install all dependencies that ARTEMIS needs, based on the `README` file provided, and then
- follow the directions provided in the `INSTALL` file under the `artemis-modified` directory.

10.2 Evaluating the DOM Analysis

The JavaScript benchmarks included in the artifact are stored in a set of subdirectories, one benchmark program per subdirectory. To evaluate the DOM event dependency analysis on all benchmarks, run the following script:

```
$ make build-dep-all
```

Alternatively, the dependency analysis may be evaluated on each individual benchmark. Assume that the name of the benchmark is `prog`; you may run the following script:

```
$ make build-dep prog
```

The result of our new dependency analysis tool, JSDEP, will be stored in three files under the `info` subdirectory for each benchmark:

- `dep.txt`, which stores the dependency relations;
- `numConstraints.txt`, which stores the total number of constraints, and
- `z3.time`, which stores the execution time.

10.3 Evaluating the Modified Artemis

To evaluate the modified ARTEMIS tool on all benchmarks, run the following script:

```
$ make-run-artemis-all
```

Alternatively, ARTEMIS may be evaluated on each individual benchmark. Assume that the name of the benchmark is `prog`; you may run the following script:

```
$ make-run-artemis prog
```

Internally, ARTEMIS uses the `dep.txt` file computed by JSDEP for each benchmark to improve the testing performance. It stores the final results in two plaintext files under the `artemis-result` directory for each benchmark:

- `old_artemis.stdout`: output of the original ARTEMIS, and
- `new_artemis.stdout`: output of the modified ARTEMIS.

10.4 Experimental Data

Our raw experimental data are stored in the directory named `raw-data`. The data may be used to generate the result tables used in this paper, by running the following three commands:

```
$ make table1
$ make table2
$ make table3
```

The above commands print the result tables in the LATEX format.

After obtaining new experimental data by re-running our tools, the command `make fetch-data` may be used to move the newly created experimental data from the current benchmark directory to the `raw-data` directory.

After that, the three aforementioned commands, `make table1`, `make table2` and `make table3`, may be used to process the updated data and generate the new result tables.

11. REFERENCES

[1] 100 Online JavaScript Games. URL: http://www.lutanho.net/stroke/online.html.

[2] S. Alimadadi, A. Mesbah, and K. Pattabiraman. Hybrid dom-sensitive change impact analysis for javascript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 321–345, 2015.

[3] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 17–31, 2014.

[4] S. Arlt, A. Podelski, and M. Wehrle. Reducing GUI test suites via program slicing. In *International Symposium on Software Testing and Analysis*, pages 270–281, 2014.

[5] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, and F. Tip. A framework for automated testing of JavaScript web applications. In *International Conference on Software Engineering*, pages 571–580, 2011.

[6] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 243–262, 2009.

[7] L. Cheng, J. Chang, Z. Yang, and C. Wang. GUICat: GUI testing as a service. In *IEEE/ACM International Conference On Automated Software Engineering*, 2016.

[8] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–62, 2009.

[9] 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. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[10] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *International Conference on Software Engineering*, pages 752–761, 2013.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[13] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Springer, 1996.

[14] S. Guarnieri and V. B. Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, pages 151–168, 2009.

[15] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *International Conference on World Wide Web*, pages 561–570, 2009.

[16] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming*, pages 256–275, 2011.

[17] K. Hoder, N. Bjørner, and L. de Moura. muZ - an efficient engine for fixed points with constraints. In *International Conference on Computer Aided Verification*, pages 457–462, 2011.

[18] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. T. Vechev. Stateless model checking of event-driven applications. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 57–73, 2015.

[19] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the *eval* that men do. In *International Symposium on Software Testing and Analysis*, pages 34–44, 2012.

[20] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of javascript web applications. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 59–69, 2011.

[21] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *International Conference on Computer Aided Verification*, pages 398–413, 2009.

[22] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.

[23] M. Kusano and C. Wang. Assertion guided abstraction: a cooperative optimization for dynamic partial order reduction. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 175–186, 2014.

[24] M. Kusano and C. Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2016.

[25] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–12, 2005.

[26] G. Li, E. Andreasen, and I. Ghosh. SymJS: automatic symbolic testing of JavaScript web applications. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 449–459, 2014.

[27] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015.

[28] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, 2005.

[29] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 499–509, 2013.

[30] M. Madsen, F. Tip, and O. Lhoták. Static analysis of event-driven Node.js JavaScript applications. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 505–519, 2015.

[31] L. A. Meyerovich and V. B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy*, pages 481–496, 2010.

[32] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[33] C. Nguyen, H. Yoshida, M. R. Prasad, I. Ghosh, and K. Sen. Generating succinct test cases using don't care analysis. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 1–10, 2015.

[34] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *ACM SIGPLAN Conference on*

Object Oriented Programming, Systems, Languages, and Applications, pages 815–816, 2007.

[35] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - A large-scale study of the use of eval in javascript applications. In *European Conference on Object-Oriented Programming*, pages 52–78, 2011.

[36] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 43–55, 2001.

[37] 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.

[38] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 488–498, 2013.

[39] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of JavaScript. In *European Conference on Object-Oriented Programming*, pages 435–458, 2012.

[40] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 382–396, 2008.

[41] S. Wei and B. G. Ryder. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *European Conference on Object-Oriented Programming*, pages 1–26, 2014.

[42] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, 2004.

[43] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 250–259, 2015.

[44] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *International Conference on World Wide Web*, pages 805–814, 2011.