

SymJS: Automatic Symbolic Testing of JavaScript Web Applications

Guodong Li
Fujitsu Labs of America,
Sunnyvale, CA, USA
gli@fla.fujitsu.com

Esben Andreasen *
Dept. of Computer Science,
Aarhus University, Denmark
esbena@cs.au.dk

Indradeep Ghosh
Fujitsu Labs of America,
Sunnyvale, CA, USA
ighosh@fla.fujitsu.com

ABSTRACT

We present SymJS, a comprehensive framework for automatic testing of client-side JavaScript Web applications. The tool contains a symbolic execution engine for JavaScript, and an automatic event explorer for Web pages. Without any user intervention, SymJS can automatically discover and explore Web events, symbolically execute the associated JavaScript code, refine the execution based on dynamic feedbacks, and produce test cases with high coverage. The symbolic engine contains a symbolic virtual machine, a string-numeric solver, and a symbolic executable DOM model. SymJS's innovations include a novel symbolic virtual machine for JavaScript Web, symbolic+dynamic feedback directed event space exploration, and dynamic taint analysis for enhancing event sequence construction. We illustrate the effectiveness of SymJS on standard JavaScript benchmarks and various real-life Web applications. On average SymJS achieves over 90% line coverage for the benchmark programs, significantly outperforming existing methods.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Reliability, Experimentation

Keywords: JavaScript, Web, Symbolic Execution, Automatic Software Testing, Event Sequence, Taint Analysis

1. INTRODUCTION

Traditionally, software quality has been assured through manual testing which is tedious, difficult, and often gives poor coverage especially when availing of random testing approaches. This has led to much recent work in the area of formal validation and testing. One such technique is symbolic execution [5] which can be used to automatically generate test inputs with high structural coverage.

Symbolic execution treats input variables to a program as unknown quantities or symbols [10]. It then creates com-

plex equations by executing all possible finite paths in the program with the symbolic variables. These equations are then solved through an SMT (Satisfiability Modulo Theories [11]) solver to obtain test cases and error scenarios, if any. Thus this technique can reason about all possible values at a symbolic input in an application. Though it is very powerful, and can find specific input values to uncover corner case bugs, it is computationally intensive.

Symbolic execution poses specific challenges to the domain of JavaScript based Web applications which are quite different from traditional programs [19]. Web applications are usually event driven, user interactive, and string intensive. Also, unlike traditional programming languages JavaScript is a dynamic, untyped and functional language.

Some symbolic execution tools have been presented for C/C++ [23, 4], Java [18], and low level code [25, 8]. It has been reported that they can find corner case bugs and produce test cases with very high coverage. SymJS targets JavaScript Web applications, which so far has received much less attention. To make such a symbolic execution tool usable in practice, the tool should be fully automatic, highly efficient, and able to handle general JavaScript Web applications. This not only requires an efficient and customizable engine with start-of-the-art solving techniques addressing JavaScript language semantics, but also a smart automatic driver builder that explores the user event sequence and input value space exhaustively during the dynamic execution and subsequent test generation phases.

Though there are a few symbolic execution tools for JavaScript such as Kudzu [21] and Jalangi [22], such comprehensive testing capabilities as described above are lacking. Kudzu uses fuzzing and symbolic string reasoning to analyze security vulnerabilities in Web applications. Jalangi instruments the source JavaScript code to collect path constraints and data for replaying. While Jalangi works for pure JavaScript programs and not general Web applications, Kudzu focuses on only one specific testing need - finding security issues. In this paper, we show that the event-driven nature of Web applications poses a bigger challenge than JavaScript itself. Unfortunately, neither Kudzu [21] nor Jalangi [22] pays much (if any) attention to this problem.

The following table shows how well random and symbolic testing work for pure JavaScript (JS) programs and Web applications. For pure JavaScript benchmarks manipulating numbers, random testing can easily achieve over 90% line coverage; the coverage drops to around 55% for string intensive programs. SymJS employs symbolic execution with a powerful string solver to achieve more than 96% average

*Contributed to this work during a summer internship program at Fujitsu Labs of America

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...\$15.00
<http://dx.doi.org/10.1145/2635868.2635913>

coverage. Covering the rest requires better drivers. This becomes more prominent for Web applications: random testing achieves 76% coverage, primitive symbolic execution gets over 88%, while symbolic execution with optimizations can cover the corner cases that are very difficult to hit. More details are presented in Section 5.

	Random	Sym.	+Opt.
Pure JS	> 90%(num.) ~55%(str.)	> 96%	–
JS Web	76%	88.7%	90.8%

In practice, JavaScript programs are used primarily in the Web context, hence the main task is to develop a symbolic engine for Web, and automatically construct good drivers while curbing space explosion. This requires novel methods to perform symbolic execution and construct event sequences. Specifically, we present how to (1) implement a symbolic virtual machine supporting both symbolic and taint analysis; (2) perform feedback-directed symbolic execution to explore the event state space; and (3) use taint analysis to optimize event sequence construction.

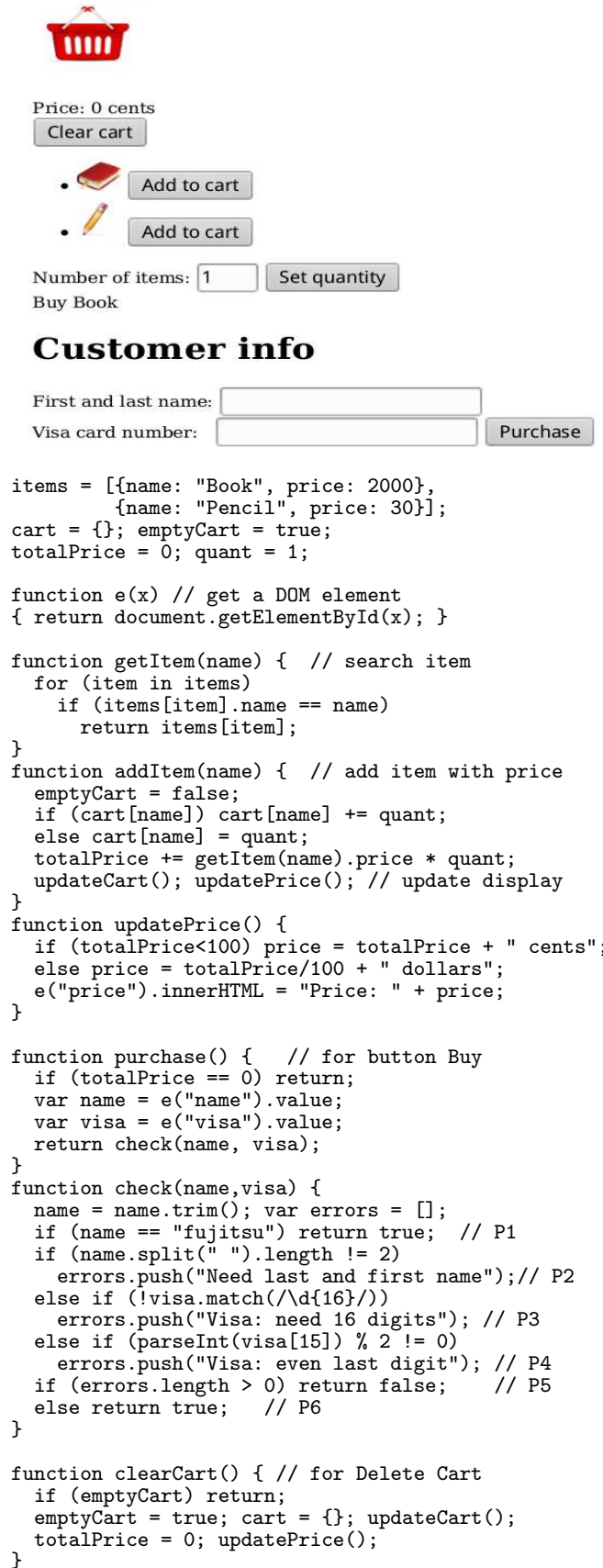
SymJS uses a symbolic virtual machine to execute JavaScript code. The source code is not instrumented but a symbolic virtual machine is implemented (similar to [4, 16] and [18]). Unlike Kudzu which separates event exploration with symbolic execution, SymJS includes a powerful event explorer that builds and refines the drivers automatically during symbolic execution. The novel features and achievements of SymJS are as follows:

- The execution engine performs symbolic execution for JavaScript within a virtual machine. It supports all language features and most built-in libraries of JavaScript, and various execution modes. It incorporates crucial optimizations such as state merging for property accesses. It is the first to (1) use a parametrized-array string-numeric solver with various optimizations, and (2) implement a symbolic DOM model with model and executable classes. (Section 3)
- For many benchmark programs the execution engine is able to achieve 100% line coverage, with over 96% on average. This will be apparent in Section 5.1.
- The automatic event explorer relieves a user from the burden of building drivers and manually controlling the execution. It uses dependency and feed-backs to explore the event state space more efficiently. (Section 4.1)
- The symbolic engine supports taint analysis through dynamic symbolic execution, which can be used to significantly enhance event sequence construction. (Section 4.2)
- For various Web applications, SymJS is able to achieve over 90% average coverage in reasonable time, which is significantly better than some prior automatic Web testing tools such as [1]. (Section 5.2)

2. MOTIVATION AND OVERVIEW

Figure 1 shows a simplified version of a shopping cart example, where the Web page contains three input text boxes (for quantity, customer name and credit card number) and five buttons for manipulating the cart: add “book”, add “pencil”, clear cart, set quantity, and purchase.

Event `purchase` invokes function `check`. Its first branch checks whether the name input equals to “fujitsu”, and the second one checks whether the name consists of a first name and a last name. The third branch uses a regular expression to ensure that the card number contains 16 digits, and



```

items = [{name: "Book", price: 2000},
          {name: "Pencil", price: 30}];
cart = {}; emptyCart = true;
totalPrice = 0; quant = 1;

function e(x) // get a DOM element
{ return document.getElementById(x); }

function getItem(name) { // search item
  for (item in items)
    if (items[item].name == name)
      return items[item];
}

function addItem(name) { // add item with price
  emptyCart = false;
  if (cart[name]) cart[name] += quant;
  else cart[name] = quant;
  totalPrice += getItem(name).price * quant;
  updateCart(); updatePrice(); // update display
}

function updatePrice() {
  if (totalPrice < 100) price = totalPrice + " cents";
  else price = totalPrice/100 + " dollars";
  e("price").innerHTML = "Price: " + price;
}

function purchase() { // for button Buy
  if (totalPrice == 0) return;
  var name = e("name").value;
  var visa = e("visa").value;
  return check(name, visa);
}

function check(name, visa) {
  name = name.trim(); var errors = [];
  if (name == "fujitsu") return true; // P1
  if (name.split(" ").length != 2)
    errors.push("Need last and first name");// P2
  else if (!visa.match(/\d{16}/))
    errors.push("Visa: need 16 digits");// P3
  else if (parseInt(visa[15]) % 2 != 0)
    errors.push("Visa: even last digit");// P4
  if (errors.length > 0) return false; // P5
  else return true; // P6
}

function clearCart() { // for Delete Cart
  if (emptyCart) return;
  emptyCart = true; cart = {}; updateCart();
  totalPrice = 0; updatePrice();
}

```

Figure 1: Motivating example “Demo Cart”.

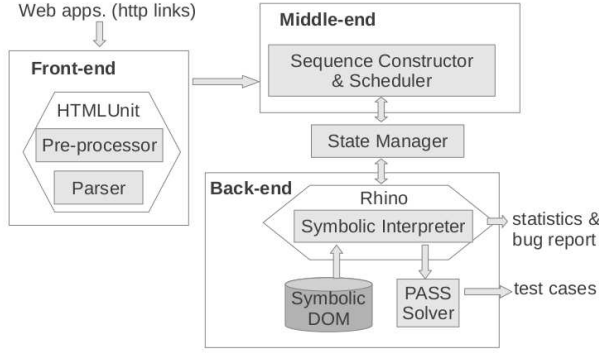


Figure 2: Main Components of SymJS.

the fourth one checks the last digit. The path conditions are shown below, where $P2' = (\text{trim}(\text{name}) \neq \text{"fujitsu"} \wedge \text{trim}(\text{name}).\text{split}(\text{" "}).\text{length} = 2)$. Test cases can be produced by solving these path conditions, *e.g.* a valid solution for P4 is: $\text{name} = \text{"a b"}$ and $\text{visa} = \text{"0000000000000001"}$. SymJS applies symbolic execution to explore all paths, and uses an efficient string-numeric solver [13] to determine path feasibility and produce test cases.

```
P1 : trim(name) = "fujitsu"
P2 : trim(name) ≠ "fujitsu" ∧ trim(name).split(" ").len ≠ 2
P3 : P2' ∧ ¬visa.match("d{16}")
P4 : P2' ∧ visa.match("d{16}") ∧ parseInt(visa[15]) % 2 ≠ 0
P5 : P2 ∨ P3 ∨ P4
```

Function check will be executed only when $\text{totalPrice} \neq 0$. Directly clicking Purchase after loading the page will not invoke check. To execute this function, we can construct and execute the following 2-event sequence.

1. click Add to Cart on ‘‘book’’ (invoke addItem);
2. click Purchase (invoke purchase)

In this sequence, function `addItem` modifies totalPrice to a value greater than 0. Then function `purchase` can pass the check on totalPrice and invoke `check`. Note that `addItem` contains a branch with condition $\text{cart}[\text{name}] = \text{undef}$. To visit the right side of this branch, we can use another sequence [AddItem; AddItem]. In general, many event sequences may be needed for exhaustive testing. SymJS constructs these sequences automatically based on dynamic analysis.

Overview. As shown in Figure 2, SymJS consists of three main components: front-end (parser), middle-end (sequence manager), and back-end (symbolic executor). The front-end obtains Web pages (*e.g.* through HTTP requests), parses them, and stores data in local format. We add pre-processing in the front-end to collect information for the middle-end. Currently the front-end uses a frame-less browser HTMLUnit¹. We extend HTMLUnit’s DOM and Browser API model to support symbolic execution.

The middle-end creates, schedules and manages event sequences. It symbolizes the inputs and executes a sequence event by event. It monitors the execution and collects feedback information to construct and refine event sequences. It contains various sequence construction schemes, one of which is our implementation of the main algorithm of a concrete testing tool Artemis [1].

¹<http://htmlunit.sourceforge.net/>

l	$:= l_1, l_2, \dots$	label
uop	$:= \text{not}, \text{neg}, \dots$	unary operation
bop	$:= \text{add}, \text{mul}, \text{rshift}, \dots$	binary operation
cop	$:= \text{gt}, \text{ge}, \text{eq}, \dots$	condition operation
instr	$:= \text{push } v$	add value into frame
	$\text{uop} \mid \text{bop}$	unary binary operation
	cop	comparison operation
	$\text{goto } l$	unconditional jump
	$\text{ifeq } l \mid \text{ifne } l$	conditional jump
	$\text{getprop} \mid \text{setprop}$	get set property
	$\text{call} \mid \text{ret}$	function call return

Push value into the frame: **push v** :

$$(l, pc, \llbracket \vec{v} \rrbracket, G) \hookrightarrow (l + 1, pc, \llbracket v; \vec{v} \rrbracket, G)$$

Unary Operation: **uop** :

$$(l, pc, \llbracket v_1; \vec{v} \rrbracket, G) \hookrightarrow (l + 1, pc, \llbracket \text{uop}(v_1); \vec{v} \rrbracket, G)$$

Binary/Comparison Operation: **op** for $\text{op} \in \{\text{bop}, \text{cop}\}$:

$$(l, pc, \llbracket v_1; v_2; \vec{v} \rrbracket, G) \hookrightarrow (l + 1, pc, \llbracket v_1 \text{ op } v_2; \vec{v} \rrbracket, G)$$

Conditional Jump: **ifeq l'** :

$$(l, pc, \llbracket v_1; \vec{v} \rrbracket, G) \hookrightarrow \{(l', pc \wedge v_1, \llbracket \vec{v} \rrbracket, G) \cup \{(l + 1, pc \wedge \neg v_1, \llbracket \vec{v} \rrbracket, G)\}$$

Property Get: **getprop** :

$$(l, pc, \llbracket S; v_1; \vec{v} \rrbracket, G) \hookrightarrow \bigcup_{i \in \text{dom}^*(S)} (l + 1, pc \wedge i = v_1, \llbracket S(i); \vec{v} \rrbracket, G) \cup \{(l + 1, pc \wedge \bigwedge_{i \in \text{dom}^*(S)} i \neq v_1, \llbracket \text{undef}; \vec{v} \rrbracket, G)\}$$

Optimized Property Get: **getprop** :

$$(l, pc, \llbracket S; v_1; \vec{v} \rrbracket, G) \hookrightarrow (l + 1, pc, \llbracket v_{\text{new}}; \vec{v} \rrbracket, G) \text{ where } v_{\text{new}} = \text{ite}(v_1 = i_1, S(i_1), \text{ite}(v_1 = i_2, S(i_2), \dots)) \text{ for } i_1, i_2, \dots \in \text{dom}^*(S)$$

Property Set: **setprop** :

$$(l, pc, \llbracket S; v_1; v_2; \vec{v} \rrbracket, G[x \mapsto S, y \mapsto S']) \hookrightarrow \bigcup_{S' \in \text{pts}(S) \wedge i \in \text{dom}(S')} (l + 1, pc \wedge i = v_1, \llbracket \vec{v} \rrbracket, G_1[y \mapsto S'[i \mapsto v_2]]) \cup \{(l + 1, pc \wedge \bigwedge_{i \in \text{dom}(S)} i \neq v_1, \llbracket \vec{v} \rrbracket, G_2[x \mapsto S[i \mapsto v_2]])\} \text{ where } G_1 = G[x \mapsto S] \text{ and } G_2 = G[y \mapsto S'].$$

Figure 3: Syntax and operational semantics of core Rhino bytecode (excerpt).

The back-end is the symbolic virtual machine for JavaScript, which extends the Rhino² JavaScript engine for symbolic execution. It interprets each Rhino Icode symbolically, forks states on feasible branches, and manages the states. It does not reuse any existing symbolic executor. The back-end and middle-end share the state management component to control the state and sequence execution.

3. SYMBOLIC EXECUTION ENGINE

Similar to [4, 18], the engine works on intermediate representations (bytecode, *i.e.* Rhino Icode) rather than source programs. This lessens the burden of handling complex source syntax, and leads to a more robust tool implementation. Figure 3 shows the bytecode’s syntax and semantics.

An execution state consists of the current label l (or instruction counter), path condition pc , frame F , and global scope G . A frame is a stack of values. It also maintains a pointer to its parent frame pertaining to the function caller. We use $\llbracket v; \vec{v} \rrbracket$ to denote a frame with v at the top and \vec{v} representing the rest values. Pushing a new value v_1 into this frame results in $\llbracket v_1; v; \vec{v} \rrbracket$, and popping v re-

²<https://developer.mozilla.org/en-US/docs/Rhino>

sults in $[\bar{v}]$. A scope maps properties to values (e.g. constants or objects), and may embed other scopes. For a scope object S , we use $S[x]$ to denote the value at address x , and $S[x \mapsto v]$ the state mapping address x to value v . Clearly, a read after a write satisfies the following property: $(S[x \mapsto v])[y] = \text{ite}(x = y, v, S[y])$, (ite means “if-then-else”).

A JavaScript object may have a prototype chain for its parent objects. A scope S can access the properties of its prototype scopes. We use $S\langle x \rangle$ to denote reading property x from scope S dynamically, and show below a formal definition. Here $\text{dom}(S)$ gives the domain (i.e. property names) of S ; $\text{prototype}(S)$ returns S ’s direct prototype. Similarly, we introduce notation $\text{pts}(S)$ for all the prototype objects of S (including S); and $\text{dom}^*(S)$ for the domain of $\text{pts}(S)$.

$$S\langle x \rangle = \begin{cases} S[x] & \text{if } x \in \text{dom}(S) \\ S'\langle x \rangle & \text{if } S' = \text{prototype}(S) \wedge S'\langle x \rangle \neq \text{undef} \\ \text{undef} & \text{otherwise} \end{cases}$$

Figure 3 shows the execution of some instructions over symbolic values. Other instructions such as object and property creation and deletion, various function call types, exception handling, scopes and literals are not included. We show how a state transits (\hookrightarrow) to a new state when executing an instruction. For example, instruction **neg** increases the label by 1, pops value v_1 from the frame, and puts v_1 ’s negation back into the frame. A comparison instruction is always followed by a conditional jump. New states may also be spawned if the condition and its negation are satisfiable under the current path condition. Here we use set operations (e.g. set union) to depict state spawning.

When a property is read from a scope S , if the property’s id is symbolic, then we match it with existing properties in $\text{dom}^*(S)$, and spawn a new state for each match $i = v_1$ (more precisely, the first match in the order of S and then S ’s prototypes). When no match exists, the path condition is updated and **undef** is returned as the result. We will discuss an optimization later. For property set, S and its prototypes are searched, and the first S' containing a matched property will be updated. If no match is found, then S is updated.

For illustration, we show below the state transitions for statement “`cart[name] += quant;`” in the motivating example. Suppose that (1) variables “name” and “quant” have symbolic values n and q respectively; and (2) in the global scope G , property “cart” maps to scope S , which in turn maps “book” to q_1 , and “pencil” to q_2 , i.e. the quantities of pencil and book are q_1 and q_2 respectively.

$(l, [\bar{n}; \dots], G[\text{“cart”} \mapsto S]) \hookrightarrow$	push “cart”
$(l + 1, [\bar{G}; \text{“cart”}; n; \dots], G[\text{“cart”} \mapsto S]) \hookrightarrow$	push G
$(l + 2, [\bar{G}; \text{“cart”}; n; \dots], G[\text{“cart”} \mapsto S]) \hookrightarrow$	getprop
$(l + 3, [\bar{S}; n; \dots], G') \hookrightarrow$	getprop
{state 1 : $(l + 4, (p_1 : n = \text{“book”}), [\bar{q}_1; \dots], G')$,	
state 2 : $(l + 4, n = \text{“pencil”}, [\bar{q}_2; \dots], G')$,	
state 3 : $(l + 4, n \notin \{\text{“pencil”}, \text{“book”}\}, [\text{undef}; \dots], G')$ }	
where $G' = G[\text{“cart”} \mapsto S]$ and $p_1 = (n = \text{“book”})$	
state 1 \hookrightarrow	push q
$(l + 5, p_1, [\bar{q}; q_1; \dots], G') \hookrightarrow$	add
$(l + 6, p_1, [\bar{q}_1 + q; \dots], G') \hookrightarrow$	push n
$(l + 7, p_1, [\bar{n}; q_1 + q; \dots], G') \hookrightarrow$	push “cart”
$(l + 8, p_1, [\bar{G}; \text{“cart”}; q_1 + q; \dots], G') \hookrightarrow$	push G'
$(l + 9, p_1, [\bar{G}; \text{“cart”}; n; q_1 + q; \dots], G') \hookrightarrow$	getprop
$(l + 10, p_1, [\bar{S}; n; q_1 + q; \dots], G') \hookrightarrow$	setprop
$(l + 11, p_1, [\dots], G[\text{“cart”} \mapsto [\text{“book”} \mapsto q_1 + q]])$	

The “property get” instruction leads to three states. In state 1, the path condition specifies that the name matches

property “book”, and value q_1 is fetched from the scope and pushed into the stack. The subsequent computation adds q into the book’s quantity, and updates scope S with the new quantity. Here type conversion will be applied to the operands according to the ECMA standard. The computations for the other two states are similar.

SymJS applies crucial optimizations to mitigate state explosion, especially for symbolic property gets and sets. SymJS uses **ite** expressions to merge the states if possible. In the above example, the 3 states produced by **getprop** can be combined to the following one. Our string solver is good at handling string-numeric **ite** expressions.

$$(l + 4, \text{true}, [q'; \dots], G') \\ \text{where } q' = \text{ite}(x = \text{“book”}, q_1, \text{ite}(x = \text{“pencil”}, q_2, \text{undef}))$$

SymJS provides two methods for spawning new states. As in [4, 18], the first method clones states for new execution paths. The second one is easier to implement but has bigger overhead: it restarts the execution from the beginning. This is a fuzzing mode similar to [23, 8] but it is performed within a symbolic stack-based virtual machine.

In the fuzzing mode, a state contains a L/R sequence recording the sides (Left or Right) taken for the feasible branches so far, which is used to “replay” the execution during a restart. For instance, function **check** alone leads to the following 6 states, where $tname$ is the short-hand for **trim(name)**. For each state, the executor first replays the execution by taking all the specified sides, then continues the usual symbolic execution from there.

State	PC	L/R Seq
1	$P_1 : tname = \text{“fujitsu”}$	l
2	$\neg P_1 \wedge (P_2 : \text{split}(tname).len \neq 2)$	r;l
3	$\neg P_1 \wedge \neg P_2 \wedge (P_3 : \neg \text{visa.match}(/d\{16\}/))$	r;r;l
4	$\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge (P_4 : \text{visa}[15]\%2 \neq 0)$	r;r;r;l
5	$\neg P_1 \wedge (P_2 \vee P_3 \vee P_4)$	r;r;r;l
6	$\neg P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4$	r;r;r;r

These path conditions are solved to produce test cases. The test cases can be replayed in the same engine. For example, the following statement introduces a symbolic string for variable s . When replaying, SymJS assigns the test value (e.g. “fujitsu”) to s , then the engine performs concrete execution over this value. We use replaying to confirm test validity and coverage. Supporting random testing is trivial: this function returns a random string, then the engine performs concrete execution.

```
// s = symb. value in sym. exec. and test value in replay.
var s = symjs_mk_sym_string('s');
```

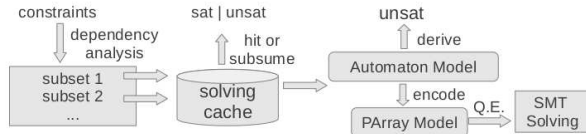
String-Numeric Solver. One key for practical symbolic execution is constraint solving since the solver may be invoked hundreds of thousands of times for a non-trivial application. It is the advances in constraint solving (SMT solving [11] became over 1000 times faster in the last decade) that make symbolic execution practical and more popular.

Symbolic numeric constraints can be solved through SMT solvers (Yices [6] in our case). However, JavaScript features extensive String operations, which are not supported by most modern SMT solvers. Thus SymJS introduces the PASS solver [13] to handle string constraints. PASS supports integers, bit-vector numbers, floating point numbers, and strings. In particular, it supports a majority of the JavaScript string operations, including string comparisons, string-numeric conversions, and regular expressions.

PASS is the first to use parameterized arrays (PArrays) as the main data structure to model strings, and converts string constraints into quantified expressions that are solved through an efficient and sound quantifier elimination algorithm. In particular, similar to [7], it can identify unsatisfiable cases quickly. For example, constraints `s.contains("ab")` \wedge `s.match("a+")` can be disproved in the automaton domain.

Consider the path condition of state 4 shown above, where `name` not equals to "fujitsu" and can be split into two parts, and variable `visa` should have 16 digits, with the last digit modulo 2 not equal to 0. PASS first introduces PArrays for `name` and `visa`, then creates an automaton from the regular expression `\d{16}` to constrain `visa`'s value. The automaton is encoded into PArray formulas, which enforces that `visa`'s length is 16. Then the quantifier eliminator instantiates the PArrays to obtain a solution, *e.g.* `visa` = "0000000000000001" and `name` = "A A".

Multiple optimizations are adopted when we incorporate PASS into SymJS. The engine utilizes dependency solving, cache solving and expression simplification to reduce the burden on the solver. They work for both numeric and string constraints. For example, in the `check` function, when examining condition `!visa.match(/\d{16}/)`, we do not need to consider the constraints associated with variable `name`. We apply a dependency analysis for this. Caching can speed up the solving too, *e.g.* implement incremental solving implicitly [5]. Similar techniques have been used in KLEE [4] and SAGE [8], but for pure numeric constraints only. The following shows the main steps in the optimized PASS.



JavaScript Library and Symbolic DOM. JavaScript has a built-in library for common data structures and operations. SymJS interprets them symbolically, and optimizes the calls so as to reduce the overhead in symbolic execution. Some optimizations are similar to those described in [14]. The processed libraries include Number, String, Array, Global, String, Math, and Regular Expression.

HTMLUnit contains a quite comprehensive model for HTML DOM and Browser APIs. HTMLUnit models Chrome, Firefox, IE 8-11, and some older browsers. Since the HTMLUnit model allows only concrete values, SymJS revises the entire model (including hundreds of model and executable classes) to support symbolic values.

4. AUTOMATIC WEB EVENT EXPLORER

The event explorer is essentially an automatic driver generator for Web applications. Since these applications are event-driven, and expect user inputs, an automatic tool is required to create scenarios that simulate user inputs and dispatch the events automatically. The main challenge is to curb state explosion: n events may produce $O(n^k)$ event sequences of length k ; the situation becomes worse when each sequence incurs multiple paths w.r.t symbolic inputs. This problem exists even for unit-testing since an event may depend on other events. For example, an event may be created or enabled only after a sequence is executed.

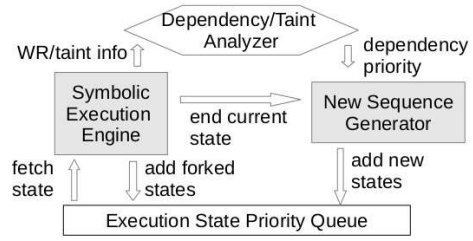


Figure 4: Main components in the event explorer.

```

executed_states  $\Sigma_{ex} = \{\}$ ;
priority_queue  $Q = \{[evt] \mid evt \text{ is enabled}\}$ ;
while  $Q \neq \{\}$  do
    // the following executes an iteration
    state  $\sigma = (seq, \mathbb{B}) = Q.remove\_head()$ ;  $\Sigma_{new} = \{\}$ ;
    load page; replay branch decisions recorded in  $\mathbb{B}$ ;
    forall the remaining instruction  $instr \in seq$  do
        execute  $instr$ , fork new states if needed;
        add new forked states into  $\Sigma_{new}$ ;
        update L/RSeq and WS info for  $\sigma$ ;
        update RS information for the current event;
    end
     $\Sigma_{ex} = \Sigma_{ex} \cup \{\sigma\}$  // mark  $\sigma$  as executed;
     $Q = Q \cup \Sigma_{new}$ ; // add all new/forked states
    // add sequences for new events
    foreach new discovered  $evt_1$  do
        add  $([seq; evt_1], \mathbb{B})$  with priority  $-1$  into  $Q$ ;
    end
    // add new sequences for existing events
    foreach enabled event  $evt$  do
        if  $([seq; evt], \mathbb{B}) \notin \Sigma_{ex}$  then
            priority  $i = \text{sizeof}([seq, \mathbb{B}] \rightsquigarrow_{wr} evt)$ ;
            add state  $([seq; evt], \mathbb{B})$  into  $Q$  with priority  $i$ ;
        end
    end
    // reorder existing sequences
    foreach  $([seq, evt], \mathbb{B}) \in Q : evt\text{'s } RS \text{ is changed}$  do
        recalculate  $([seq, evt], \mathbb{B})$ 's priority;
    end
end

```

Algorithm 1: Feedback-directed event sequence construction during symbolic execution.

Figure 4 shows the main procedure. At each iteration, the first state in the priority queue Q is executed. During the execution, the information about how the events and the sequences read and write shared variables is recorded in the dependency analyzer. The analyzer can be enhanced using taint analysis. When the current state σ forks new states, *e.g.* at feasible branches, these new states will be added into Q . When σ finishes execution, a test case is generated, and the New Sequence Generator constructs new sequences based on dependency information collected so far.

4.1 Dynamic+Symb. Sequence Construction

Consider the Demo Cart example. After the page is loaded, five events are enabled: Clear Cart, two Add to Cart, Set Quantity, Purchase. Composing these events arbitrarily may lead to a large number of sequences. We apply various analysis to identify duplicate sequences so as to reduce the sequence state space. The first one is Write-Read (WR) analysis,

which relates events according to how they write and read shared variables. For example, the WR information of these events is as following (due to space constraint we only show a portion of this information).

Event	Read Set	Write Set
clearCart	{emptyCart = {true}}	
purchase	{totalPrice = {0}}	
add("book")	{totalPrice = {0}, cart.book = {undef}, ...}	{emptyCart = false, cart.book = 1, totalPrice = 2000, ...}

In add("book")'s write set, variable *totalPrice* has value 2000, which is different from that in purchase's read set. Hence we can construct a new sequence [add("book");purchase]. Here $[evt_1; evt_2; \dots]$ denotes a sequence containing events evt_1, evt_2, \dots . In contrast, we do not build sequence [clearCart; purchase] since clearCart does not alter the value of any variable, *i.e.* this sequence is the same as [purchase].

We construct sequences and collect WR information dynamically during symbolic execution. Algorithm 1 describes the feed-back directed event explorer. The first state σ of the queue is first replayed to the last forked point according to its L/RSeq \mathbb{B} , then continues regular symbolic execution. The L/R sequence \mathbb{B} (see Section 3) records the branch sides taken so far by this sequence. We do not present \mathbb{B} if it is empty. Note that the path condition is not shown here since it will not be used explicitly. New states forked by σ are recorded in Σ_{new} and will be added into the queue. The WR information of both the current sequence *seq* and all executed events is updated during the execution. When σ finishes execution, if there are new discovered events, then they will be appended to σ with priority -1 (indicating that these events haven't been executed). Then for each enabled event, SymJS uses the WR information to infer dependency and assign priorities. The priority is the size of the shared variable set determined by \sim_{wr} . Finally, for each state $([seq, evt], \mathbb{B})$, if *evt*'s RS information is updated, then to reflect the feed-backs, the scheduler recalculate this state's priority w.r.t. the relation of *seq* and *evt*.

More formally, notations $RS(\sigma)$ and $WS(\sigma)$ return σ 's read set and write set respectively. The RS and WS record the *first read* value and *last written* value of each variable respectively. In a sequence's WS, each variable has a single value; while in an event's RS, a variable may have multiple values collected from different executions of this event.

Event Chain	:=	seq	⊂	array of events
Sequence State	:=	σ	⊂	$seq \times L/RSeq$
Sequence Write Set	:=	WS =	⊂	$[id \mapsto \mathbb{V}]$
Event Read Set	:=	RS	⊂	$[id \mapsto \text{set of } \mathbb{V}]$

When a sequence state $\sigma = (seq, \mathbb{B})$ is finished, we generate new sequences by appending the enabled events to σ . If the analyzer determines that an event *evt* is related to σ , then a new state $([seq; evt], \mathbb{B})$ will be created. Consider the following example: state σ containing sequence *seq* writes variable *a* with symbolic value *v*; events evt_1 and evt_2 read *a* with values {1} and {2, *v*} respectively. An event read may have multiple values corresponding to its different invocations. Here σ and evt_1 are related since evt_1 may read a different value from the one written by *seq*, *i.e.* constraint $v = 1$ is satisfiable. Hence we generate a new state $([seq; evt_1], l; r)$, which inherits *seq*'s L/R sequence. On the other hand, σ and evt_2 are unrelated since *v* is already in evt_2 's RS. That is, appending evt_2 to *seq* may fail to explore new state space.

$$\frac{\sigma = (seq, l; r)}{WS = \{a = v\}} \quad \frac{evt_1}{RS = \{a = \{1\}\}} \quad \frac{evt_2}{RS = \{a = \{2, v\}\}}$$

Dynamic Write-Read Analysis. We use $\sigma \sim_{wr} evt$ to measure the dependency relation between sequence state σ and event *evt* regarding shared variables. Dependency and data-flow analysis prevails in traditional program analysis; however the dynamic feature of JavaScript makes it more difficult to statically calculate dependency information. Hence we collect this information dynamically.

- **Named WR:** $\sigma \sim_{wr} evt = \{x \mid x \in WS(\sigma) \wedge x \in RS(evt)\}$, *i.e.* the variables written by σ and read by *evt*.
- **Valued WR:** $\sigma \sim_{wr} evt = \{x \mid x \in WS(\sigma) \wedge x \in RS(evt) \wedge WS(\sigma)[x] \notin RS(evt)[x]\}$, *i.e.* a variable *x* is included if σ 's write value to *x* is not in the read set of *x* in ψ . If the values are symbolic, the solver is used to determine whether the values can be different. This is the default analysis that SymJS uses.
- **Conditional Valued WR:** the path condition is also recorded for a read or write, and is used to determine inequality. However our experience indicates that this does not outperform Valued WR in practice.

An important point here is to identify shared variables at run-time. Roughly, we name a DOM element with a unique id, and a JavaScript shared object through traversing its prototype chains till the global (*e.g.* window) scope. A property or variable *x* is named "*s.x*" where *s* is the name of *x*'s scope. For example, the fifth element of an array object *A* in the top scope is named "window.A.5". We can skip the "window" when the context is clear. This scheme is not 100% accurate, but it is sufficient for WR analysis in practice.

Example. Continue with the Demo Cart example. The explorer first adds 5 initial states into the state queue, then executes them one by one with the Valued WR analyzer. Supposed that the current sequence is [add("book")]. The analyzer assigns priority 1 to sequence [add("book");purchase] based on variable *totalPrice*, and to [add("book"); clearCart] based on *emptyCart*. Similarly [add("book"); add("book")] has priority 2. If the current sequence is [clearCart], then the priority of both [clearCart; purchase] and [clearCart; add("book")] is 0 (meaning "unrelated"). All these sequences are added into the queue. We show below the WR information of two new sequences. More sequences can be constructed until a pre-defined bound is reached.

Sequence	RS	WS
[add("book"); purchase]	{totalPrice={0}, name.value={sym_v1}, cart.book = {undef}, ...}	{emptyCart=false, totalPrice=2000 item=1, ...}
[add("book"); clearCart]	{emptyCart={true}, totalPrice={0}, ...}	{emptyCart=true, totalPrice=0, ...}

The analyzer is able to avoid creating useless sequences such as [clearCart; add("pencil");] and [add("book"); clearCart; purchase]. In other words, SymJS implicitly prunes duplicate sequences using the analysis. This often leads to significant reduction of the explored state/sequence space.

4.2 Sequence Construction via Taint Analysis

The WR analysis maintains shared variables in WR sets. However, some variables may be irrelevant to the branches in subsequent computations. If line/branch coverage is the

main goal, then we can apply an optimization to rule out irrelevant variables from the WR sets, and use coverage information to guide event sequence construction.

We apply a taint analysis to calculate how shared variables flow into unvisited branches. For an event function f , the analysis builds a map from f 's branches to used variable sets. For example, in function `clearCart`, the branch involves variable `emptyCart` only, *i.e.* `emptyCart` is tainted w.r.t the branch. To cover this branch, we can consider only this variable in the WR sets.

We use symbolic execution with a shadow data-flow calculation to obtain taint information. That is, data-flow information is collected and updated during symbolic execution runs. The executor introduces symbolic values for input variables, propagates values to path conditions, and checks termination at backward control flow points. At each control flow point, set V_{sym} stores all the symbolic inputs known so far by all runs. The calculation terminates when a fix-point of V_{sym} on all points is reached. This mimics the traditional use-def calculation for C or Java, except that our method is based on dynamic symbolic execution. A traditional method to collect taint information is to start from all sinks and calculate the relevant variables by exploring the control flow backwards. Here we show a different approach. For illustration, consider the following function with a loop.

```
function f(a,b) {
1:  for (a > 0) {           // use {a}
2:    e = a - 2;           // use {a}, def {e}
3:    if (A.d > e + a) {    // branch br: use {e,a,A.d}
4:      a += c; ...;       // use {a,c}, def {a}
5:    }
6:  }
```

Initially, the V_{sym} at each control flow point is empty. Since the loop header uses variable a , we introduce a shadow symbolic value to a and stores a in V_{sym} . In the first iteration of the loop, line 2 uses a , which is already in V_{sym} . Then two execution states σ_1 and σ_2 (see below, where $V_0 = \{a, c, A.d\}$) are spawned at the “if” branch br , with $A.d$ added into V_{sym} . Here property access `access A.d` is resolved using the property get method presented in Section 3. Only the variable or property of a primitive type will be made symbolic. We dynamically instantiate the type of a shadow symbolic variable. For variable A , we use its original value when it turns out to be an object at line 3. Hence our analysis is not precise, but it is sufficient for our data-flow calculation.

state	Selected steps with executed lines and associated V_{sym}
σ_1	1 : { a }; 3 : { $a, A.d$ }; 4 : V_0 ; 5 : V_0 ; 1 : V_0 ; ...; 1 : V_0
σ_2	1 : { a }; 3 : { $a, A.d$ }; 5 : V_0 .

The branch now uses $\{a, A.d\}$. State σ_1 executes line 4 and uses variable c . Here c is used the first time, thus a shadow symbolic value is assigned to c . State σ_2 takes br 's right side and uses no variable. Their control-flows converge at line 5. Suppose σ_1 reaches this line first. It updates V_{sym} 's new value to $\{a, c, A.d\}$ from its old value, *i.e.* empty set. Since V_{sym} 's value is changed, state σ_1 continues the execution. Now, supposed σ_2 reaches line 5. The old and new values of V_{sym} are the same, hence σ_2 terminates. Next, σ_1 goes back to line 1 (another convergence point) and the second iteration begins. Here V_{sym} at line 1 is updated to $\{a, c, A.d\}$ from $\{a\}$. Then the execution goes on as shown

above. Finally, branch br uses variable $\{a, c, A.d\}$, indicating that variable b is not relevant to br .

The following shows more formally the calculation. For a state σ , we use a set $\text{def}(\sigma)$ to record the locally defined variables (so that no symbolic values are introduced for them). Assume that $\sigma = (l, pc, [S; v_1; v_2; \vec{v}], G)$. At a branch br , we fork states, and record in $\text{use}(br)$ the free symbolic variables in the path condition. When a property get or set occurs, def and V_{sym} are updated. Convergence points are recorded in \mathbb{L}_{conv} , *e.g.* the target of a jump will be added into \mathbb{L}_{conv} . An optimization is to consider only immediate post-dominators. When such a point is reached, we check whether if V_{sym} is unchanged. If yes then terminate σ .

Instruction	Taint Operation
$br : \text{ifeq } l'$	$\begin{cases} \text{use}(br) \cup = \{x \mid x \in \text{free_vars}(pc)\} \\ \mathbb{L}_{conv} \cup = \{l'\} \end{cases}$
$\text{goto } l'$	$\mathbb{L}_{conv} \cup = \{l'\}$
getprop	$\begin{cases} \text{def}(\sigma) \cup = \{S.v_1\} \wedge V_{sym} \cup = \{S.v_1\} \\ \text{if } S.v_1 \notin \text{def}(\sigma) \end{cases}$
setprop	$\text{def}(\sigma) \cup = \{S.v_1\}$
$l \in \mathbb{L}_{conv}$	$\begin{cases} \text{terminate } \sigma & \text{if } V_{sym_{old}} = V_{sym} \\ V_{sym_{old}} \cup = V_{sym} & \text{otherwise} \end{cases}$

The following shows some taint operations for state σ_1 . At line 1, \mathbb{L}_{conv} is updated to $\{l_1\}$ since the loop jumps back to l_1 , and a shadow symbolic value for a is introduced. Now V_{sym} contains a , *i.e.* the symbolic input known so far is a . At line 2, no new symbolic value is introduced since a is already in def . The execution goes on till the second iteration begins. The second loop iteration adds c into $\text{use}(br)$. When the iteration finishes, this state terminates due to $S_{sym_{old}} = V_{sym} = \{a, A.d, c\}$. Similarly other states terminate after the second iteration, leading to a fixed-point where $\text{use}(br) = \{a, c, A.d\}$ contains all the variables relevant to the target br .

line	instr.	$V_{sym_{old}}$	def	V_{sym}
1 (l_1)	<code>getprop, ifeq</code>	$\{\}$	$\{a\}$	$\{a\}$
2 (l_2)	<code>get/setprop</code>	$\{\}$	$\{a, e\}$	$\{a\}$
3 (l_3)	<code>getprop, ifeq</code>	$\{\}$	$\{a, e, A.d\}$	$\{a, A.d\}$
4 (l_4)	<code>get/setprop</code>	$\{\}$	$\{a, e, A.d, c\}$	$\{a, A.d, c\}$
1 (l_1)	$l_1 \in \mathbb{L}_{conv}$	$\{a\}$	$\{a, e, A.d, c\}$	$\{a, A.d, c\}$
...				
1 (l_1)	$l_1 \in \mathbb{L}_{conv}$	$\{a, A.d, c\}$	$\{a, e, A.d, c\}$	$\{a, A.d, c\}$

It is trivial to use taint information to optimize sequence construction: we simply exclude irrelevant variables in the WR sets when constructing new sequences. SymJS also considers only unvisited branches in the event. As shown below, the Named WR relation between sequence state σ and event evt can be extended using taint information. The definition of Taint Valued WR is analogous.

- **Taint Named WR:** $\sigma \sim_{wr}^{taint} evt = \{x \mid x \in \text{WS}(\sigma) \wedge \exists \text{unvisited } br \in evt : x \in \text{use}(br)\}$, *i.e.* the variables written by σ and relevant to at least an unvisited branch in evt .

Example. Continue with the **Demo Cart** example in Section 4.1. Supposed that sequences `[purchase]` and `[add("book")]` have been executed, hence the two sides of the branch “`totalPrice = 0`” have been covered. Consider sequence `[add("pencil")]` with the following WS. Variables `price` and `item` do not appear in any program branch. The taint analyzer (TA) makes use of this information, and can fur-

ther rule out sequences pertaining to variable *totalPrice* by analyzing the coverage.

Sequence	Write Set (WS)
[add("pencil")]	{totalPrice=30, emptyCart=false, price="30 cents", item=1, ...}

The following event `purchase` also reads variable *totalPrice*. Pure WR analysis produces a new sequence [add("pencil"); `purchase`] with priority 1, which is useless since this new sequence behaves the same as [`purchase`] in terms of branch coverage. Specifically, the two sides of "`totalPrice = 0`" have been covered, and the uncovered branches in function `check` do not involve *totalPrice*. In contrast, the TA infers that no variables in [add("pencil")]’s WS are relevant to the uncovered branches in `purchase`, hence discards this new sequence.

Event	ReadSet (RS)
<code>purchase</code>	{ totalPrice = {0, 2000}, name.value= <i>sym.v1</i> , ... }

We also apply a simple optimization that records the path condition *pc* of an uncovered branch if *pc* keeps the same during taint calculation, *e.g.* when the branch is not within a loop. For example, consider the branch in function `updatePrice`. Obviously sequence [add("book")] covers the right side, and sets *totalPrice* = 2000. To cover the left side, the TA infers that [add("book");add("book")] is useless since path condition *totalPrice* < 100 is falsified by *totalPrice* = 2000. Many other similar sequences are excluded.

5. EVALUATION

We evaluate both the execution engine and the event explorer. All experiments are performed on a machine with Intel(R) Xeon(R) CPU @ 2.67GHz and 4GB memory.

5.1 Evaluating Pure JavaScript Programs

We evaluate the SymJS executor on the Sunspider 1.0.1 benchmark suite and the V8 JavaScript benchmark suite (v7) in the latest WebKit version (<http://www.webkit.org/>). These benchmark programs use core JavaScript language only, not the DOM or other browser APIs. For each program we create a driver, where the inputs are assigned symbolic numbers, strings, or untyped values. We intentionally write simple drivers without investigating the programs.

Table 1 compares the results of random, manual, and symbolic testing. The programs in the upper half of the table are from Sunspider, and the bottom ones are from V8. For Sunspider, we analyze representative programs, *e.g.* `access-fannkuch` from 4 `access` programs. We exclude libraries when counting the lines. For instance, after excluding "`prototype.js`", `raytrace` has 918 source lines, among which 404 are executable. While each program is not large, it usually contains expensive computations, *e.g.* containing embedded loops and involving extensive bit-wise operations.

Random testing can achieve reasonable coverage but may miss corner cases. For example, for `3d-cube`, random testing achieves 96.2% line and 77% branch (taken) coverage with 100 runs combined. For random testing, we use $n = \max(100, m)$ runs where m is the number of tests produced in the symbolic case. Using the tests in the original benchmark suite results in 92.3% and 65.7% respectively. As indicated by `earley-boyer` and `regexp`, random performs much worse when string inputs are involved.

In contrast, SymJS produces 9 valid test cases in 6.1 seconds for `3d-cube`. Among this execution time, 1% (*i.e.* 0.061

second) is spent on solving. The coverage is increased to 98.1% (line) and 81% (branch taken). Although the engine explores 9 paths, all branches can be covered by only one test. SymJS is able to perform coverage-preserving test reduction to exclude duplicate tests. Consider another example — program `string-validate-input` that checks the name and zip code. Its 702 tests can be safely reduced to 5.

We investigate why SymJS does not achieve full coverage for some programs. The reasons include (1) the program contains expensive operations (EO), especially non-linear operations; (2) the driver is too restrictive (DR); and (3) state explosion (SE). We mark the reasons for non-fully-covered programs in Table 1. Specifically, SymJS’s concretizing non-linear operations may miss paths, and expensive bit-wise operations often lead to time-out of the SMT solver and therefore the termination of the path. We sometimes need to restrict the drivers to avoid SE and EO.

Program `early-boyer parser` takes 1663 paths to achieve 86.9% branch coverage. Increasing the number of tests does not help because the new paths contain expensive bit-wise operations that the solver fails to handle. The coverage for `raytrace` is relatively low because the code related to the "canvas" object is unreachable in the pure JavaScript mode.

For many programs in the Table, constraint solving contributes to most of the total execution time. On average, the solver optimizations described in Section 3 can lead to 10 – 20% improvement on solving time. The improvement is marginal since PASS [13] has used various inner optimizations, *e.g.* applying early termination and incremental solving, as well as using the automaton model lazily.

Since SymJS has achieved very high coverage for standard benchmark programs (which are typically small), and the uncovered code is due mainly to the driver problem, we do not perform coverage comparison of SymJS with other JavaScript symbolic tools such as Jalangi.

5.2 Evaluating the Tool on Web Applications

Table 2 shows the results of running SymJS on 10 Web applications, some of which are used by Artemis [1] (www.brics.dk/artemis). These applications are off-line, with all (client-side) source code stored in the local disk. We compare random testing, symbolic testing with or without WR analysis, and the extension with taint analysis. For each application, we give its number of source lines (excluding JavaScript libraries such as JQuery), the line and branch coverage, the number of iterations (#I, see Algorithm 1), the maximal sequence length L , the total execution time along with the solving time percentage.

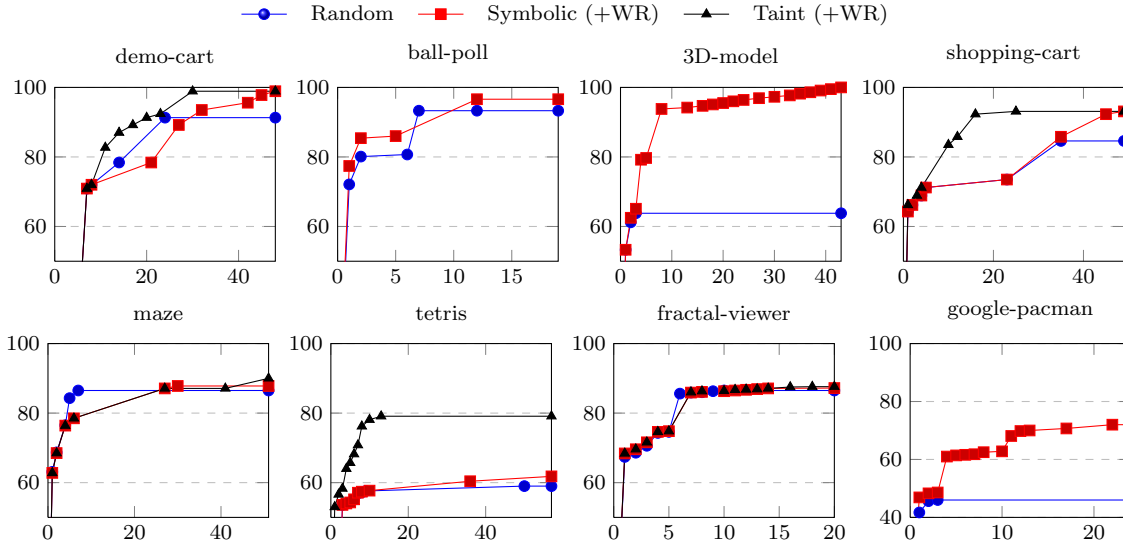
For example, application `demo-cart` contains 93 executable lines. Note that some statements such as large array declaration and initialization are not counted. With 100 iterations, random testing can achieve 91.3% line and 68.7% branch coverage with or without using WR analysis. With naive symbolic execution we can get 98.9% line and 93.7% branch coverage with 93 iterations; executing 7 more will not increase the coverage (*e.g.* many useless sequences are explored). Applying WR analysis reduces the iteration number to 48. The entire execution takes 1.9 seconds, 13% of which is spent on solving. We use a 1-second wait time for an event with timers. The maximum sequence length is 3. Taint analysis takes 0.4 second to run; it can help to reduce the iteration number to 30. For some programs, taint time "—" indicates the failure to get a valid fixed-point in 2 min-

Table 1: Evaluation Results on Sunspider 1.0.2 (Upper) and V8 (Bottom) Benchmark Suite

Program	#Lines		Random Cov.	Suite Cov.	Symbolic				Cov.
	Source	Exec.	Line/Br	Line/Br	#Tests	Line/Br Cov.	Exec. T	Sol.%	Impr.
access-fannkuch	74	44	100%*/91.1%	97.7%/91.1%	12 (1)	100%/93.3%	47.6s	12%	DR,EO
3d-cube	369	216	96.2%*/77%	92.3%/65.7%	9 (1)	98.1%/81%	6.1s	1%	EO,DR
math-cordic	85	26	100%/100%	91.8%/87.5%	30 (1)	100%/100%	0.6s	75%	—
string-validate-input	99	51	92.1%/88.8%	92.5%/92.3%	702 (5)	100%/100%	9.3s	87%	—
splay	386	151	90%/75.7%	91.2%/74.2%	48 (7)	92.7%/81.4%	1.5s	79%	DR
richards	523	230	97.3%/96.4%	95.1%/91%	16 (3)	97.3%/96.4%	9.6s	37%	DR
regex	353	206	55.8%/50%	100%/100%	185 (92)	100%/100%	5.3s	79%	—
earley-boyer(parser)	618	294	39.7%/19.1%	—	1663 (52)	91.4%/86.9%	26.9s	87%	DR,SE
deltablue	874	355	91.8%/70.2%	94%/70.8%	100 (4)	97.1%/82.2%	2.9s	9%	DR
raytrace	918	404	84.9%/76.4%	84.6%/68.8%	11 (8)	84.9%/76.4%	65s	47%	DR,EO
average			84.8%/74.5%	93.2%/82.4%		96.2%/89.8%			

Table 2: Evaluation results on offline Web applications.

Program	#Lines		Random	Symbolic (Max #I = 100)									
	Source	Exec.	#I=100	-WR			+WR			+WR+Taint			Cov. (L/B)
			Cov. (L/B)	#I	Cov. (L/B)	#I	L	Time	#I	L	Tnt T.	Cov.	
demo-cart	146	93	91.3%/68.7%	93	98.9%/93.7%	48	3	1.9s(13%)	30	3	0.4s	98.9%/93.7%	
form-validator	95	54	62.9/30%	71	100%/95%	71	1	3.6s(37%)	71	1	0.3s	100%/95%	
simp-calculator	243	138	87%/74.3%	73	90.5%/80%	73	1	2.6s(36%)	73	1	2.1s	90.5%/80%	
ball-poll	263	150	93.3%/87%	18	96%/85.1%	19	1	41s(17%)	19	1	–	96.6%/90.7%	
3D-model	4745	227	63.8%/59.5%	45	100%/97.6%	45	1	121s(1%)	45	1	69s	100%/97.6%	
shopping-cart	809	400	84.6%/78.6%	49	93.1%/82.7%	49	5	6.4s(9%)	25	3	0.2s	93.1%/82.7%	
maze	187	140	86.5%/63.8%	54	87.8%/69.4%	49	2	1.8s(4%)	51	2	3.4s	90%/77.7%	
tetris	1396	637	59%/30.2%	89	61.8%/35%	57	3	45s(1%)	13	3	3.7s	79.1%/57.6%	
fractal-viewer	1585	752	86.5%/75%	18	87.2%/76.7%	20	6	142s(1%)	18	2	32s	87.6%/77%	
google-pacman	3231	1454	46%/25.8%	24	72%/54.2%	24	2	268s(1%)	24	2	–	72%/54.2%	
average			76.1%/59.3%		88.7%/76.9%							90.8%/80.6%	


Figure 5: Coverage (y-axis) as a function of the iterations (x-axis) for some benchmarks.

utes, which may be caused by path explosion or exceptions related to introduced symbolic values.

Typically, 10 to 70 iterations are required to achieve good coverage. After that, exploring the remaining states is mostly fruitless. The sequence lengths are from 1 to 6. Particularly, 4 applications require only singleton sequences, *i.e.* the events are symbolically unit-tested.

Table 5 shows plot charts measuring the coverage over iterations. In most cases, we can get over 50% using a few iterations; and the coverage improvement becomes slower

when #I becomes larger, *e.g.*, for these applications, the latter half of the iterations bring < 30% new coverage. Here, complex event sequences may be needed to cover more, and more state space reduction will be required to eliminate useless states so as to go deeper into the state space.

From these charts, it is clear that the symbolic method can get higher coverage with fewer iterations than the random method. The major advantage of symbolic+taint is to reduce the iteration number substantially. It can also hit corner cases that a pure symbolic method fails to find.

Clearly, it is important to use WR analysis to increase the coverage, and taint analysis to reduce the number of sequences. Interestingly, pure WR without taint analysis may perform worse. Table 3 shows more information, where various versions of `demo-cart` are obtained by adding listing items. Consider “demo-cart2”, the no-WR, Named WR, Valued WR and taint methods need 87, 99, 49 and 36 iterations to achieve the same 98.9% coverage. For `shopping-cart` (-RFS), only the WR+taint method can beat the no WR one. Here RFS (Restricting Shared Attributes) is a technique that considers only a subset of attributes of a DOM element when building WR sets, *e.g.* it excludes unimportant attributes such as “window.height” so as to increase the accuracy of a WR analysis. We predefine this subset through an empirical study. This experience leads us to develop an automatic taint analyzer to identify these attributes. When this option is turn off, pure WR analysis may perform poorly since it considers too many irrelevant variables. In contrast, the taint analysis is insensitive to RSV and can always identify relevant variables. Hence, in practice, taint analysis is a *must* rather than an optional optimization.

Table 3: Comparing various analysis w.r.t. the iteration number to achieve the same target coverage.

Program	No WR	+WR		+Taint
		Named	Valued	
demo-cart1	49	93	27	20
demo-cart2	87	99	49	36
demo-cart3	>100	>100	>100	28
shopping-cart (-RSV)	49	> 100	> 100	25
shopping-cart1 (-RSV)	> 100	> 100	> 100	19
dynamic-articles	> 100	> 100	54	22

Table 4 compares SymJS with Artemis [1] on some shared benchmarks. We do not include other Artemis benchmarks which HTMLUnit cannot parse properly (*e.g.* PHP files) or SymJS cannot introduce symbolic inputs into. In general, SymJS achieves better coverage with fewer iterations. For example, SymJS uses 18 iterations to obtain 87.6% line coverage for program `fractal-view`, while Artemis obtains 75% at iteration 80 and cannot improve the coverage further. One reason is that SymJS does not need to guess input values since SymJS performs symbolic execution. We could not compare SymJS with Kudzu [21] as the end-to-end Kudzu tool is not publicly available.

Table 4: Comparing SymJS with Artemis [1] in term of the best coverage achieved.

Program	Artemis		SymJS	
	Cov.	#Iter.	Cov.	#Iter.
3D-model(+lib.)	74%	~15	78.6%	45
ball-poll	90%	~65	96.6%	18
fractal-viewer	75%	~80	87.6%	18
google-pacman	44%	~15	72%	24

The coverage can be improved using various enhancements. First of all, some browser dependent code (typically 1 – 3%) in the benchmarks can be covered by simulating other browsers. Second, currently we introduce symbolic values for specific elements such as text inputs and radio boxes. We observed that the coverage could be increased with more symbolic inputs for some applications. Finally, executing longer sequences may explore new state space. For example, the uncovered code of `tetris` is due mainly to the “gameOver” function, which will be invoked with a long sequence.

On-line Applications. We have run SymJS on various on-line Web applications including Google and Yahoo on-line services, and popular electronic commerce sites. SymJS communicates with the server to fetch the pages, and explores them dynamically. Preliminary results show that SymJS can obtain 65 – 90% line coverage. However, exceptions may occur when symbolic data are sent to the server. We leave better interactions with the server as future work.

6. DISCUSSIONS AND CONCLUSION

Additional Related Work. Feedback-directed testing has been used for API calls in Randoop [17], and for JavaScript in Artemis [1]. Kudzu [21] constructs event scenarios from random testing (in a separate phase) but does not utilize feed-backs. Unlike these tools, SymJS performs feedback-directed event construction during symbolic execution.

WR set based analysis [2] has been used to reduce state space for whole C programs (hence no drivers are needed). This technique can be adopted to improve the symbolic execution of pure JavaScript programs. The approach in [9] builds event sequences for Android applications. It also uses WR set based dependency to connect events. The sequences are constructed back-ward from a model obtained by random exploration. In contrast, our method discovers the model during symbolic execution, constructs sequences in a forward way, and uses taint analysis to connect events (we have shown that pure WR is insufficient). In another domain, [20] symbolically executes classes through a read/write analysis to track possible changes between methods.

Regarding state explosion, we may adopt advanced state merging techniques (*e.g.* [12]) and duplicate state elimination (*e.g.* [3]). For example, [3] uses dynamic slicing and symbolic reasoning to detect duplicate states. Coverage-directed search and reduction used in [16] may be adopted to handle the concurrent nature of event execution. In addition, it is possible to apply symbolic encoding and loop abstraction as in [15] to avoid explicitly enumerating all interactions between dependent units such as events.

Taint analysis techniques [26, 24] use dynamic+static analysis to find violations in Web applications. Our light-weight method reuses symbolic execution to collect data-flow information. More sophisticated taint analysis may help reduce the state/sequence space further.

Conclusion. We present SymJS, a symbolic framework for both pure JavaScript and client-side Web programs. To the best of our knowledge, this push-button tool is the first to implement a symbolic virtual machine for JavaScript and automatic driver construction with dynamic symbolic taint analysis. The approximate development size of major components in SymJS is 27,000 lines of Java code (counting only our development), with around 10%, 35% and 55% for the front-end, middle-end, and back-end respectively.

Compared to Kudzu [21] (whose focus is on string reasoning and security checking) and Jalangi [22] (whose focus is on light-weight source instrumentation), SymJS aims to provides an end-to-end, fully automatic, and comprehensive symbolic executor. We have demonstrated how to build enhancements such as taint analysis in the framework. In addition to performance improvement, we plan to enhance the interactions between the tool and Web servers. We are also interested in detecting security vulnerabilities in websites.

7. REFERENCES

- [1] ARTZI, S., DOLBY, J., JENSEN, S. H., MOLLER, A., AND TIP, F. A framework for automated testing of JavaScript Web applications. In *International Conference on Software Engineering (ICSE)* (2011).
- [2] BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. R. RWset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008).
- [3] BUGRARA, S., AND ENGLER, D. R. Redundant state detection for dynamic symbolic execution. In *USENIX Annual Technical Conference (USENIX ATC)* (2013).
- [4] CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).
- [5] CADAR, C., AND SEN, K. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.
- [6] DUTERTRE, B., AND MOURA, L. D. The Yices SMT Solver. Tech. rep., Computer Science Laboratory, SRI International, 2006.
- [7] GHOSH, I., SHAFIEI, N., LI, G., AND CHIANG, W.-F. JST: An automatic test generation tool for industrial Java applications with strings. In *International Conference on Software Engineering (ICSE)* (2013).
- [8] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Sage: Whitebox fuzzing for security testing. *Commun. ACM* 10, 1 (2012), 20.
- [9] JENSEN, C. S., PRASAD, M. R., AND MØLLER, A. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis (ISSTA)* (2013).
- [10] KING, J. Symbolic execution and program testing. *Communications of the ACM* 19, 7 (1976), 385–394.
- [11] KROENING, D., AND STRICHMAN, O. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 2008.
- [12] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient state merging in symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2012).
- [13] LI, G., AND GHOSH, I. PASS: String solving with parameterized array and interval automaton. In *Haifa Verification Conference (HVC)* (2013).
- [14] LI, G., GHOSH, I., AND RAJAN, S. P. KLOVER : A symbolic execution and automatic test generation tool for C++ programs. In *International Conference on Computer Aided Verification (CAV)* (2011).
- [15] LI, G., AND GOPALAKRISHNAN, G. Scalable SMT-based verification of GPU kernel functions. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT FSE)* (2010).
- [16] LI, G., LI, P., SAWAGA, G., GOPALAKRISHNAN, G., GHOSH, I., AND RAJAN, S. P. GKLEE: Concolic verification and test generation for GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2012).
- [17] PACHECO, C., LAHIRI, S. K., ERNST, M. D., AND BALL, T. Feedback-directed random test generation. In *International Conference on Software Engineering (ICSE)* (2007).
- [18] PĂSĂREANU, C. S., AND RUNGTA, N. Symbolic PathFinder: symbolic execution of Java bytecode. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2010).
- [19] RICHARDS, G., LEBRESNE, S., BURG, B., AND VITEK, J. An analysis of the dynamic behavior of JavaScript programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2010).
- [20] RIZZI, E. F., DWYER, M. B., AND ELBAUM, S. Safely reducing the cost of unit level symbolic execution through read/write analysis. *ACM SIGSOFT Software Engineering Notes* 39, 1 (2014).
- [21] SAXENA, P., AKHAWA, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A Symbolic Execution Framework for JavaScript. In *IEEE Symposium on Security and Privacy (Oakland)* (2010).
- [22] SEN, K., BRUTCH, T., GIBBS, S., AND KALASAPUR, S. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (SIGSOFT FSE)* (2013).
- [23] SEN, K., MARINOV, D., AND AGHA, G. CUTE: a concolic unit testing engine for C. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2005).
- [24] SRIDHARAN, M., ARTZI, S., PISTOIA, M., GUARNIERI, S., TRIPP, O., AND BERG, R. F4F: taint analysis of framework-based Web applications. In *ACM International Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)* (2011).
- [25] TILLMANN, N., AND DE HALLEUX, J. PEX: white box test generation for .net. In *International Conference on Tests and Proofs (TAP)* (2008).
- [26] TRIPP, O., PISTOIA, M., FINK, S. J., SRIDHARAN, M., AND WEISMAN, O. TAJ: effective taint analysis of Web applications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2009).