Type-Aware Concolic Testing of JavaScript Programs

Monika Dhok  
Indian Institute of Science  
monika.dhok@csa.iisc.ernet.in

Murali Krishna Ramanathan  
Indian Institute of Science  
muralikrishna@csa.iisc.ernet.in

Nishant Sinha  
IBM Research, India  
nishant.sinha@in.ibm.com

1. INTRODUCTION

Dynamic typing [11] in JavaScript (JS) enables modifications to the object properties during runtime and helps reduce development time [6]. However, the lack of compile time warnings increases the possibility of dormant bugs resulting in many type errors in the field [12]. Hence, effective testing frameworks are essential to reduce the maintenance cost for JS based applications [36, 25, 13, 37, 28, 18]. These approaches are targeted towards testing various facets of JS applications – event driven and string intensive web applications can employ SymJS [25], security vulnerabilities can be detected using Kudzu [36], web specific features pertaining to manipulation of DOM and AJAX can be tested using ARTEMIS [13], etc. Unfortunately, there is limited support to generate high coverage test cases for pure JS programs e.g., nodejs programs, where input types are complex objects instead of events or strings.

In practice, for programs based on statically typed languages, concolic testing [20, 38, 19, 31, 24] helps achieve high coverage. Broadly, this involves a combination of concrete and symbolic execution which generates a collection of symbolic constraints on execution paths [30, 15]. Different paths in a program are covered by negating collected branch constraints appropriately and solving the resultant collection. When constraints cannot be solved symbolically, concrete values are provided to ensure completion of the process. The approach is designed to avoid redundant exploration of paths.

When conventional concolic testing is applied to JS (as in [37]), we observe that two kinds of constraints are generated. Constraints that correspond to program branch conditions, labeled branch constraints (e.g., z = 1); these constraints may be flipped [20, 38] to execute hitherto unexplored paths. Constraints that restrict variable types are called type constraints, (e.g., typeof(z) = number). Testing different combinations of branch constraints help explore different paths in the program. In contrast, type constraints ensure that the variables are well-typed, which in turn provides the basis for assigning correct values to them.

When applying concolic testing to JS, we encounter an unexpected stumbling block to scalability: highly-redundant input generation. Redundant inputs are inputs that do not achieve new code coverage with respect to existing tests. We observe that many of these tests predominantly contain variables that have not been assigned any value. Such variables are recognized as undefined in JS. If undefinedness is expected explicitly, execution of a program in the presence of undefined values can help achieve more coverage. Otherwise, this leads to generation of inputs which are a few orders of magnitude more than the actual number of normally ter-
2. MOTIVATION

In this section, we motivate the need for our approach by using a real example from hapi-v8.1.0 [8], a popular library with 99k downloads in the month of July 2015, that provides a rich framework for building JS applications and services. It is also under active development with 4400 commits, 188 releases and 125 contributors.

Figure 1 presents a simplified implementation of two functions _marshal_ and _streamify_ from response.js in the library. The number of intra-procedural control flow paths are 2 and 7 respectively.\(^1\) The function _marshal_ takes one parameter, invokes _streamify_ conditionally depending upon the presence of the field, _processors_ in the invoking object, which should not contain the property _marshal_ (lines 3 - 5). Meanwhile, the function _streamify_ checks the type of the first parameter, and considers the fields present in the passed object to perform various operations (lines 19 - 26). Overall, an invocation of _marshal_ on a malformed object can result in a failure at any one of lines 3, 20 and 21 in the presented code.

```
1  ol._marshal = function (next) {
2      var self = this;
3      if ((this._processors.marshal) {
4          return this._streamify(this._source, next);
5      } )
6      this._processors.marshal(this, function (err, source) {
7          ...
13    });
14 );
17  ol._streamify = function (source, next) {
19      if (source instanceof Stream) {
20          var stream = (source.socket || source ) ;
21          if (stream._readableState.objectMode) {
22              return next(Boom.badImplementation('error'));
23          };
24          this._payload = source;
25          return next();
26      };
55  });
```

Figure 1 Motivating example.

Initially, when concolic testing is applied to _marshal_, the receiver (this) does not have field _processors_ causing a crash at line 3. This results in the inference that it has a field _processors_ which can be dereferenced and added appropriately. When re-executed, the absence of _marshal_ in _processors_ results in the invocation of _streamify_ at line 4. The check at line 19 will fail because the receiver does not have _source_. Consequently, a new input is generated where the receiver has _source_ which is an instance of Stream. The absence of _socket_ in _source_ causes _this.source_ to be assigned to _stream_ at line 20. Then, the execution crashes (also referred to as exceptions) again at line 21 due to the absence of _readableState_. With a re-execution where _this.source_ has _readableState_, the absence of _objectMode_ results in condition at line 21 to be false and execution of lines 24 and 25. This completes the execution of an entire path. The inference due to the execution will cause _objectMode_ to be added as a field (default value is zero) causing lines 24 and 25 to be re-executed. The value of _objectMode_ is made non-zero to cover the other branch.

\(^1\) We consider paths that do not trigger runtime exception in a function as normally terminating paths.

\(^2\) Line 20 is treated as a conditional.
There are two problems even when an execution completes (e.g., return at line 25) and is free of crashes. **Firstly**, all objects are not well-formed (e.g., `this.source.socket` and `this._processors.marshal` are undefined). **Secondly**, these undefined variables can introduce redundancy to the entire process. More specifically, when the approach backtracks to fix the undefined `this.source.socket` (at line 20) and assigns a zero to it, the constraints pertaining to `_readableState` and `objectMode` (due to line 21) are eliminated. Also, the assignment leads the execution along the previously explored path. This causes the re-executions to **repeat** previously seen crashes (and inferences) in **streamify**. The redundancy is magnified for inter-procedural testing. This causes the re-executions to **repeat** previously seen crashes (and inferences) in **streamify**. In summary, the problems with existing concolic testing for JS are:

- Execution of program in the presence of undefined variables does not help achieve more coverage, unless undefinedness is expected explicitly in a conditional or the variable type cannot be inferred.
- Defining the undefined variables incrementally results in repeated exploration of the same paths.
- The redundancy is magnified for inter-procedural testing.

When we test `marshal` in conjunction with `streamify`, concolic testing generated more than 1000 inputs in 1 hour. A variant of the testing which employs a depth-first version to explore paths (implemented in JALANGI) generates 72 inputs. This difference in the number of generated tests is mainly because of the way branches are explored. In contrast, we design an approach based on type-aware concolic testing which generates only 8 inputs without loss of coverage. These inputs are also sufficient to generate effective type preconditions for the tested functions.

There are multiple benefits that accrue due to function preconditions. Apart from their use in inter-procedural testing, these can help avoid potential crashes due to invocations with malformed objects. For example, the derived function precondition of `streamify` specifies that `this._sources.socket` should be present when `this._processors` is an instance of `Stream` to avoid the crash at line 21. This need not always be followed (e.g., see issue 2368 in hapi [5]). The derived preconditions can be used to monitor the type of `this._sources` and `_readableState` can be inserted, whenever absent, during an invocation to avoid potential runtime crashes [33].

### 3. Design

We now present the design of type-aware concolic testing of JS programs. Initially, we present the definitions in Section 3.1 that are used in the rest of the Section. We explain the extension of conventional concolic testing to JS using Algorithm 1 which uses (a) Algorithm 2 and 3 to add constraints on the execution stack, and (b) Algorithm 4 to flip the constraints on the stack for new input generation [20].

In Section 3.3, we explain type-aware intra-procedural concolic testing. We design a modified flip procedure (Algorithm 5) to resolve type and path constraints separately. Subsequently, we discuss type-aware inter-procedural testing procedure in Algorithm 6.

### 3.1 Preliminaries

For ease of presentation, we consider a subset of JS, JS-LITE, over the following grammar:

\[
\begin{align*}
ap &\in accesspath \; := \; v \mid v.f \mid e \mid \text{eval}(\text{ap}, S) \mid \text{eval}(e, S) \\
e &\in expr \; := \; ap \mid \text{call ap} \mid \text{upop e1 e2} \mid e1 \bowtie e2 \\
s &\in stmt \; := \; v := e \mid \text{if}(e) \; \text{goto} \; \ell \;
\end{align*}
\]

where

\[
v \in \text{Var}, \; \text{bop} \in \{+, -, \leq, \geq, \ldots\}, \; \text{upop} \in \{!, - , +, \ldots\}
\]

Access paths include variables (v) and one-level dereferences to the variables (v.f). An access path v.f.g can be represented as a composition of multiple access paths. The variables are from a set of symbolic variables (Var). Expressions include the access paths, invocations of functions, expressions involving unary (upop) and binary (bop) operations. Statements correspond to assignments or conditional branches.

We maintain the state (σ) as a five element tuple:

\[
\sigma = (\lambda, T, C, S, stack),
\]

where \( \lambda \) is the function name, \( T \) maps variables to concrete type values, \( C \) and \( S \) map variables to concrete and symbolic values respectively, and stack maintains an ordered list of constraints. We consider the following concrete types in the language: `{number, string, object, function, null, undefined}`.

### 3.2 Concolic Testing of JavaScript

**Algorithm 1** RUNCONCOLIC

**Input:** \( \{\lambda, T, C, S, stack\} \)

1. `stackPos ← 0;`
2. `while (\langle I ← \text{nextInstruction}(\lambda) \rangle \notin \{\text{exception}, \text{halt}\})` do
3. \`
4. case (ap := e) \n5. ap′ ← \text{eval}(ap, S); \; e′ ← \text{eval}(e, S) \n6. \text{AddC}(ap′, T, stack); \text{AddC}(e′, T, stack); \text{update}(ap′, e′, T, C, S) \n7. case (if(b) then goto \ell) \n8. \text{AddC}(b, T, stack) \n9. if (eval(b, C)) then e ← eval(b, S) \n10. else c ← \text{neg}(eval(b, S)) \n11. if (stackPos ≥ stack.length) then \n12. \text{stack.push}((\text{constraint} = \text{in_branch} = 1, \text{flipped} = 0)) \n13. Increment stackPos by one; \n14. if (stack.length > 0) then \n15. FLIP(stack); \n16. if ((T, C′, S′) = \text{solve}(stack)) then \n17. RUNCONCOLIC(\lambda, T′, C′, S′, stack) \n\`

The procedure RUNCONCOLIC in Algorithm 1 presents concolic testing algorithm for JS. It takes as input a five-tuple state σ, where \( T, C \) and \( S \) are initialized to value undefined for all access paths, and the stack is empty. In RUNCONCOLIC, nextInstruction executes the instructions in the function \( \lambda \) until it encounters a halt instruction or a crash due to an exception. We describe RUNCONCOLIC for the intra-procedural case where we ignore all the other function invocations inside that procedure. The return values of this function is considered as seed input for better coverage.

RUNCONCOLIC makes use of two standard auxiliary functions eval and update for expression evaluation and heap update, respectively, whose descriptions we omit for brevity. Given a program term \( t \) and a mapping \( \Sigma \) from access paths to values, eval(\( t, \Sigma \)) evaluates the term \( t \) in \( \Sigma \) to a value. If \( \Sigma \) is concrete, i.e., contains only constant values, then eval
returns a constant value; else a symbolic term is returned. The update \((l, v, T, C, S)\) updates the heap reference corresponding to expression \(l\) with the value of expression \(v\) in both \(C\) and \(S\) and also the concrete type of \(l\) in \(T\).

To evaluate expressions in assignments (line 4), RunConcolic creates type constraints to make sure that expressions are well-typed, using ADDC (Algorithm 2). ADDC derives the type of a given expression \(e\) as either object (line 2), function (lines 3-4) or the operator type (lines 5-9), and computes a type constraint \(c\), using the uninterpreted function \(ty\). Based on the current concrete type of expression \(e\), computed using the map \(T\), the ADDTC (Algorithm 3) adds either \(c\) or \(\neg c\) (lines 2 - 4) to the stack. It also records that the constraint is not a branch constraint. We write type constraints of form \(ty(x) = \text{num}\) as \(x \equiv \text{num}\).

**Algorithm 2** ADDC

**Input:** expression \(e\), concrete type map \(T\), stack \(stack\)

1: swap \(e\) do
2: case \((e, f)\) : ADDTC\((T(v) \neq \text{"object"}, ty(v) \neq \text{"object"}, stack)\)
3: case \((\text{call ap})\) :
4: ADDTC\((T(ap) \neq \text{"function"}, ty(ap) \neq \text{"function"}, stack)\)
5: case \((\text{lop c1})\) : /*lop is unary operator*/
6: \(t \leftarrow \text{getNonUnaryType}(lop)\)
7: ADDTC\((T(c1) \neq \text{t}, ty(c1) \neq \text{t}, stack)\)
8: case \((\text{e1 lop e2})\) : /*lop is binary operator*/
9: /*handled similar to unary operators*/

**Algorithm 3** ADDTC

**Input:** Boolean value \((b)\), constraint \((c)\), stack \((stack)\)

1: if \((\text{stackPos} \geq \text{stack length})\) then
2: if \((\neg b)\) then \(c \equiv \neg c\)
3: stack.push\((\text{constraint} = \text{c}, \text{isBranch} = 0, \text{flipped} = 0)\);
4: Increment stackPos by one;

For the conditional statement in Algorithm 1 at line 8, it evaluates the condition \(b\) in concrete value map \(C\) and depending upon the result, it adds a branch constraint \(c\) (corresponding to \(b\) evaluated in symbolic map \(S\)) to the stack (using lines 12-14). If the current stack pointer is not beyond the top of the stack, then the constraint already exists in the stack (applicable for subsequent runs of the algorithm) and need not be added. Otherwise, the constraint \(c\) is pushed on the stack along with the annotation flipped = 0, which tracks whether \(c\) has been flipped or not. We also add an annotation denoting \(c\) is a branch constraint.

After collecting all the constraints, Algorithm 1 calls FLIP procedure at line 16. This procedure is used to flip i.e., negate the constraint on the top of the stack to explore a new path. This procedure (Algorithm 4) describes the conventional [20] flipping procedure: first, it pops out all constraints that are already flipped from the stack. Then, the unflipped constraint at the top of the stack is negated. The constraints remaining on the stack are solved using a constraint solver (the function \text{solve}) at line 17. This results in updated concrete maps \(T'\) and \(C'\); RunConcolic is invoked on the state consisting of the updated maps and the stack.

**Algorithm 4** FLIP

**Input:** stack \((stack)\)

1: while \((c \leftarrow \text{stack.top}() \land c \text{.flipped})\) do stack.pop();
2: if \((\text{stack.length} \geq 0)\) then
3: \(c\text{.constraint} \leftarrow \neg c\text{.constraint} ;\)
4: \(e\text{.flipped} \leftarrow 1\)

### Table 1 Inputs and constraints for the example.

<table>
<thead>
<tr>
<th>ID</th>
<th>(I/P) ((x, y, z, o))</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((x, 0, z, z))</td>
<td>(x \equiv N, y \neq N, z \equiv N, z \equiv 1)</td>
</tr>
<tr>
<td>2</td>
<td>((x, 0, 0, \bot))</td>
<td>(x \equiv N, y \neq N, z \equiv N, z \equiv 1)</td>
</tr>
<tr>
<td>3</td>
<td>((x, 1, 1, \bot))</td>
<td>(x \equiv N, y \neq N, z \equiv N, x \equiv 1, o \equiv O, o\text{.bar} \equiv F)</td>
</tr>
<tr>
<td>4</td>
<td>((x, 1, \bot, \bot))</td>
<td>(x \equiv N, y \neq N, z \equiv N, x \equiv 1, o \equiv O, o\text{.bar} \equiv F)</td>
</tr>
<tr>
<td>5</td>
<td>((x, 0, 1, \bot))</td>
<td>(x \equiv N, y \equiv N, z \equiv N, x \equiv 1)</td>
</tr>
<tr>
<td>6</td>
<td>((x, 0, 1, 0))</td>
<td>(x \equiv N, y \equiv N, z \equiv N, x \equiv 1, o \equiv O, o\text{.bar} \equiv F)</td>
</tr>
<tr>
<td>7</td>
<td>((x, 0, 0, \bot))</td>
<td>(x \equiv N, y \equiv N, z \equiv N, x \equiv 1)</td>
</tr>
<tr>
<td>8</td>
<td>((x, 1, 1, \bot))</td>
<td>(x \equiv N, y \equiv N, z \equiv N, x \equiv 1, o \equiv O, o\text{.bar} \equiv F)</td>
</tr>
<tr>
<td>9</td>
<td>((x, 1, 1, 0))</td>
<td>(x \equiv N, y \equiv N, z \equiv N, x \equiv 1, o \equiv O, o\text{.bar} \equiv F)</td>
</tr>
<tr>
<td>10</td>
<td>((x, 0, 1, 0))</td>
<td>(x \equiv N, y \equiv N, z \equiv N, x \equiv 1)</td>
</tr>
<tr>
<td>11</td>
<td>((x, 1, 1, 1))</td>
<td>(x \equiv N, y \equiv N, z \equiv N, x \equiv 1, o \equiv O, o\text{.bar} \equiv F)</td>
</tr>
<tr>
<td>12</td>
<td>((x, 1, 1, 0))</td>
<td>(x \equiv N, y \equiv N, z \equiv N, x \equiv 1)</td>
</tr>
<tr>
<td>13</td>
<td>((x, 1, 1, 1))</td>
<td>(x \equiv N, y \equiv N, z \equiv N, x \equiv 1)</td>
</tr>
</tbody>
</table>

#### 3.2.1 Redundant input generation

Algorithm 4 does not differentiate between a type constraint and a branch constraint. However, in the context of dynamically-typed JS programs, this results in the generation of a large number of redundant inputs, thus drastically diminishing the scalability of the testing algorithm. We demonstrate the drawback using the following example:

```javascript
function foo(x, y, z, o) {
    var d = x - y;
    if(z == 1) o.bar(d);
}
```

There are four inputs to function \text{foo} and applying Algorithm 1 generates more than 20 inputs to cover the two paths in the function. Table 1 presents the inputs generated and the constraints generated for the corresponding execution. For ease of presentation, the stack is shown as a sequence. Initially, all inputs are undefined (⊥). Here we refer to types number, object and function by N, O and F respectively.

When the first input is executed, the assignment \(d = x - y\) is considered at line 4 of Algorithm 1. Applying ADDC on the RHS results in addition of the type constraints \(x \neq \text{num}\) and \(y \neq \text{num}\). Subsequently, when \(z == 1\) is evaluated, the constraint \(z \neq \text{num}\) is also added and because \(z\) is undefined currently, the conditional in \text{foo} is not satisfied and the execution terminates. A state is added to the precondition and because the stack length is greater than 0 (currently 3), the constraint at the top of the stack is flipped. This causes \(z \neq \text{num}\) being negated and the constraints on the stack are solved using \text{solve} at line 17. Consequently, the input \((1, 1, 0, 1)\) is generated to satisfy the constraints and RunConcolic is invoked at line 18. The remaining rows in the table are computed similarly.

For the constraints generated due to execution of \text{foo} with the fifth input, the last four constraints are removed from the corresponding negations are already solved (inputs 4, 3, 2 and 1 respectively). The resultant last constraint \(y \neq \text{num}\) is negated to generate the sixth input. Interestingly, not only does it lose the knowledge on inferred types of \(z, o\) and \text{o.bar} being \text{number, object} and \text{function} respectively, but also the generated constraints do not expose any new fact. The exploration \text{repeats} up to the tenth input, without exploring a newer execution path. The process repeats all over again for \(x = 0\). Not differentiating the type and branch constraints results in this unnecessary generation of inputs.

#### 3.3 Type-aware Concolic Testing

We address this fundamental drawback by proposing an approach that differentiates between the branch and type
This is achieved by flipping the set of type constraints simultaneously. For this purpose, we propose a modified version of the flipping algorithm in Algorithm 5 that can be used instead of Algorithm 4.

**Algorithm 5** Flip2

```plaintext
Input: stack (stack)
1: while (e ← stack.top() ∧ e.flipped) do stack.pop();
2: if (stack.length ≥ 0 ∧ e.is_branch) then
3: e.flipped ← 1; e.constraint ← ¬e.constraint;
4: else
5: pos ← stack.length - 1;
6: while (pos ≥ 0 ∧ stack[pos].is_branch) do
7: c ← stack[pos]; c.flipped ← 1;
8: c.constraint ← ¬c.constraint;
9: pos ← pos - 1;
```

Initially, Flip2 eliminates all constraints that are already flipped (line 1). Subsequently, if it finds a branch constraint at the top of the stack, it negates the constraint and returns (lines 2-3). This is because of the invariant that all the type constraints beneath the branch constraint are already flipped. Otherwise, it identifies all the non-branch constraints (using is_branch), negates them and marks them as flipped (lines 5-8). When Algorithm 1 is executed after this, the type constraints that are seen previously are guaranteed to be flipped and hence need not be solved incrementally.

Figure ??fig:intria presents the inputs and constraints generated by incorporating the new Flip2 algorithm in RunConcolic at line 16. The first input and the associated constraints are similar to the existing concolic testing algorithm. However, unlike the previous algorithm, we flip the type constraints simultaneously and solve them. This results in the types of x, y and z to be num and a default value of 0 is assigned to them resulting in the second input. When executed, the condition z := 1 is not satisfied and a branch constraint is added accordingly. Subsequently, the constraint at the top of the stack is flipped and the third input is constructed with z equals 1. This results in a crash due to a dereference of o as it is still undefined. The dereference causes it to be considered as an object resulting in the generation of the constraint and consequently, the fourth input. The example demonstrates that solving the type constraints simultaneously ensures significant reduction in the number of inputs generated and is closer to the number of normally terminating paths in the program. Our approach is able to completely cover all the paths with just five inputs as compared to more than 20 inputs by existing concolic testing.

By applying this technique, we avoid generation of the same pattern of malformed inputs by fixing the inputs after the execution crashes. Executing the program with the fixed input avoids the crash at the same point. For instance, in Table 1, for inputs 3 and 8, program crashes at the same point. However, input 8 doesn’t add any additional information. As shown in Figure ??fig:intria, our approach exposes the crash once with input 3 reducing the redundancy.

### 3.3.1 Differentiating branch and type constraints

Our algorithm relies on how we carefully define and distinguish between branch and type constraints. Branch constraints correspond directly to the branch condition expression: type constraints are implicitly generated at multiple places, including assignment, call and branch statements. For example, the statement, if (x == 3), generates a type constraint (x ≤ num) and a branch constraint (x == 3). This distinction between the constraints allows us to apply our type-aware exploration to programs where the branch constraint may restrict the type of the variable but is not collated with the type constraints.

In the above code, we treat the constraints generated on x as branch constraints. Thus, we can explore both paths without losing coverage. Otherwise, if these are considered as type constraints, then flipping them together and solving will lead to an infeasible execution and hence loss of coverage. Note that for buggy programs, a programmer may erroneously assume inconsistent types for a variable x along some path of execution p. In such cases, we detect that type constraints are unsatisfiable together on p and abort testing.

### 3.4 Inter-procedural Testing

Extending RunConcolic (Algorithm 1) to an inter-procedural setting involves a number of challenges to avoid path explosion. For example, if a function f’ calls f in the program under test, the inputs to f may be undefined or ill-typed, causing the execution to crash in f. To handle the crashes, we need to call f with well-typed values after detecting each abort. Repeating this exercise for each call to f and execution path inside f can lead to a severe blowup in the number of input tests generated, thus diminishing the scalability. We propose to address this problem by generating input **preconditions** for f during testing. We use these preconditions to avoid invoking f with ill-typed values.

Function preconditions are generated during the process of performing concolic testing of a function λ. It is achieved by updating RunConcolic (Algorithm 1) between lines 14 and 15 with following:

\[
\text{If}(I = \text{halt}) \quad \xi[\lambda] \leftarrow \xi[\lambda] \cup \{\text{stack}\}
\]
In other words, when an execution halts in \( \lambda \), we add the stack from the termination state to the set of preconditions \( \{\lambda\} \) for \( \lambda \). The preconditions generated for the method \( \text{foo} \) from the illustrative example is shown in Table 2 (constraints are represented as a comma-separated list). It has two entries that corresponds to the two normally executing execution paths due to condition \( x = 1 \). Each precondition contains the relevant type constraints as well as branch constraints on inputs of \( \text{foo} \). Because the number of preconditions may be quite large (in worst case, equals the number of paths executed in \( \lambda \)), we represent the set compactly using a trie data structure, ordered by formal parameter names, for efficient lookup and reuse (see below).

### Table 2 Preconditions of \( \text{foo} \).

<table>
<thead>
<tr>
<th>ID</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( x = N, y = N )</td>
</tr>
<tr>
<td>2</td>
<td>( x = N, y = N, z = 1, o = O, a _\text{bar} = F )</td>
</tr>
</tbody>
</table>

#### 3.4.1 Employing function preconditions

Algorithm 6 shows how to handle function call statements \((\text{call } ap)\) in Algorithm 2 by using previously generated preconditions. Here, let \( ap \) evaluate to a callee function \( \lambda' \) and as before, the caller function is \( \lambda \).

**Algorithm 6 CALL**

**Input:** \((\{\text{caller } \lambda, \text{callee } \lambda', \text{C, S, stack}\})\)

1: if all type constraints in stack are not flipped then
2: for each unflipped type constraint \( c \) in stack do
3: \( c\_\text{constraint} \leftarrow \neg\text{glob}(c\_\text{constraint}); c\_\text{flipped} \leftarrow 1 \)
4: \( \sigma' \leftarrow \text{solve(stack)} \)
5: runConcolic(\( \sigma' \)); abort
6: if ((\( \{\lambda'\} \) does not exist)) then
7: \( \rho \leftarrow \text{formal parameters of } \lambda' \)
8: \( T' = C' = \text{undefined}, \forall p \in \rho; S'[p] = p, \forall p \in \rho \)
9: runConcolic(\( \{\lambda, T', C', S', []\} \))
10: \( \sigma \leftarrow (\lambda, T, C, S, \text{stack}) \)
11: for (each \( \pi \) in \( \{\lambda'\} \)) do
12: \( \pi' = \text{Unify(} \pi, \sigma); \) doAbort \leftarrow false
13: if \( \pi' \neq [] \) then
14: doAbort \leftarrow true; stack' \leftarrow stack
15: for each \( \pi \) in \( \pi' \) do \( c\_\text{flipped} \leftarrow 1; \) stack' \text{push}(c)
16: \( (T'', C'', S'') = \text{solve(stack'); runConcolic(}\{\lambda, T'', C'', S'', stack'\};
17: if doAbort then abort

The procedure first flips all type constraints on the stack before calling \( \lambda' \) and re-run \text{runConcolic} (lines 1-5 in Algorithm 6) while aborting the current execution. This ensures that the variables passed to \( \lambda' \) are restricted by the types in the current execution prefix. To avoid exploring redundant inputs, CALL first computes (lines 6-9) the preconditions for \( \lambda' \) (if it doesn’t exist) and then reuse one or more precondition entries of \( \lambda' \) (lines 10-17). To compute the preconditions, we define a new set of type, concrete and symbolic maps \((T', C', S')\) for the formal parameters of \( \lambda' \) and apply \text{runConcolic} to \( \lambda' \). The completion of this step ensures the generation of the precondition for \( \lambda' \).

In order to reuse pre-computed preconditions, CALL uses an auxiliary procedure \text{Unify}, which tries to \text{unify} the current execution state \( \sigma \) in \( \lambda \) with some precondition entry \( \pi \) (recall that each entry \( \pi \) is a list of type and path constraints). Unification is based on the following rules: if \( \pi \) constrains a variable \( x \) with value \( v \), then \( \sigma.C(x) = v \). Similarly, if \( \pi \) constrains the type of \( x \) to \( t \), then either \( \sigma.T(x) \) is undefined or \( t \). If unification succeeds, it returns set of constraints \( \pi' \) (possibly empty) to update \( \sigma \) with. CALL adds these constraints to \( \sigma.\text{stack} \), and solves for new inputs which satisfy these constraints; the current execution is aborted.

We will illustrate the use of preconditions in inter-procedural concolic testing (Algorithm 6) using the example that invokes the method \( \text{foo} \) discussed in Section 3.3.

```java
function baz(x,y,z,o) {
    if(x == 1) foo(x,y,z,o);
    else foo(x,y,z,o);
    o.z = z;
}
```

Figure 2B presents the inputs and constraints generated with the use of preconditions from Table 2. Initially, the inputs are undefined resulting in the constraint \( z \neq \text{number} \). Because \( z \neq 1 \), \( \text{foo} \) in the else branch is called. At this point, because the type constraint in the caller is not flipped, we negate the constraint, solve it and generate the second input such that \( z \) is a number and assign it the default value (zero). When \( \text{foo} \) is invoked in the else branch, the stack contains a path constraint \( z \neq 1 \). The precondition entry for \( \text{foo} \) that is feasible on this path is the first entry in Table 2. The constraints are placed on the stack and flipped value of these constraints is set to one as specified in Algorithm 6 (line 15). This ensures that newly generated input will follow the path corresponding to the chosen precondition.

The merging of the two stacks is shown next to the second input. When the third input is generated to satisfy the constraints, the execution crashes at \( o.z \) and \( o \neq \text{obj} \) is added. The constraint is flipped to generate the fourth input with which execution halts. All the constraints that are flipped on the stack are eliminated and the branch constraint \( z \neq 1 \) is flipped. When \( \text{foo} \) is invoked in the if branch, the constraints that correspond to the feasible path are added (see Table 2). The constraints are solved to generate the sixth input which when executed reaches end of path. Unlike the else branch, executing the if branch will ensure that \( o.z \) does not crash in the first attempt as the precondition of \( \text{foo} \) ensures that \( o \) is an object.

### 4. IMPLEMENTATION

The concolic testing approaches can be classified as type-agnostic and type-aware approaches. For each classification, there are two variations of exploring the paths in the program – DFS and BFS. The only difference between these strategies is the order in which constraints are solved. Overall, we consider the following variants:

1. **BFS**: Existing concolic testing implementation in [37],
2. **DFS**: Implementation of Alg. 1 that uses Alg. 4,
3. **TA-BFS**: Type-aware variant of BFS, and
4. **TA-DFS**: Implementation of Alg. 1 that uses Alg. 5.

**Test Driver**: To perform concolic testing, it is essential that a proper environment be setup with initial seed inputs and variables, and the function under consideration is invoked. We statically parse the programs and setup the environment to perform concolic testing on each function. We consider global variables detected by JSHINT [3] as seed inputs. The seed inputs are initialized with undefined values to enable coverage of all possible types.

**Path-sensitivity**: The existing implementation of (type-agnostic) BFS is path-insensitive and generates inputs considering properties on objects accessed on irrelevant paths.
Because, this unnecessarily increases the redundancy of generated inputs, we made it path-sensitive. The DFS variants are also implemented to be path-sensitive. Consequently, the number of entries in the preconditions is few reducing the lookup time during inter-procedural testing.

**Handling function calls:** Constructing call graph, required for inter-procedural testing, for dynamically type languages is difficult [17] and the absence of type information of input objects makes the task more difficult. We address this by generating the preconditions of the callee functions on demand. These preconditions are derived in a context-insensitive manner. For example, if a function `foo` invokes `bar`, it is discovered at runtime. We stall the execution of `foo` and compute the precondition of `bar`. For subsequent invocations of `bar`, we reuse the computed preconditions.

**k-level inter-procedural challenges:** In the case of inter-procedural testing, we save the state of the current function (i.e., symbolic and concrete values, and stack) when the preconditions for the callee are being derived. Since the preconditions are generated on demand, the process can be performed in a nested manner until no more functions are invoked. Because saving the state at each level is complex, we parameterize the approach to handle k-levels of nesting.

**Arrays, loops and recursions:** The existing implementation of BFS testing does not handle arrays which can affect coverage. Therefore, for an array access, we ensure that the type of the index is set before setting the type of the array object. For loops, we discard the constraints generated beyond a specified limit. We arbitrarily break recursions based on the order of function invocations.

## 5. EXPERIMENTAL RESULTS

We evaluate the benefits of type-aware concolic testing across various dimensions. We use a 64 bit, 8 core Ubuntu 13.10 desktop, equipped with an Intel core i7 processor and 16GB RAM to run experiments. We applied our approach on various open-source JS programs (and libraries). We consider the following benchmarks – SunSpider [7] programs used to measure performance of JavaScript, box2d programs that are part of the octane benchmark suite [4], V8 programs used for tuning the performance of V8 [1], and hapí [8] library classes that form a framework for building applications and services.

Table 3 presents the benchmarks, their version, lines of code and number of functions in the programs/classes analyzed. We associate an identifier with each benchmark (B1 to B10) and henceforth will use them for reference. The LoC ranges from 113 for B5 to 904 for B7 and the function count ranges from 10 to 25. We designed our experiments to answer the following research questions:

1. **RQ1:** Is type-aware intra-procedural concolic testing more beneficial than existing type-unaware approaches?
2. **RQ2:** Does type-aware intra-procedural concolic testing help achieve more coverage?
3. **RQ3:** Do preconditions reduce the number of generated inputs for inter-procedural testing?
4. **RQ4:** Can preconditions be used to handle type-related crashes?

### 5.1 RQ1: Benefits of Type-aware Testing

Initially, we study the benefits of type-aware intra-procedural concolic testing. We analyze each function in an isolated manner without providing the implementation of its callees. The goal is to demonstrate the ability of the generated tests to achieve high coverage. If we consider the callee implementation, achieved coverage will be limited by the existing program structure. For each function, we analyze it with four variants of concolic testing as described in Section 4. The statistics pertaining to the number of inputs generated by TA-BFS and TA-DFS is given in Table 3 including the minimum and maximum number of inputs generated across all functions in a benchmark. For example, when the 15 functions in B9 are tested, the number of inputs generated for the different functions in it using TA-BFS range from 3 to 30. The total number of inputs required to cover the paths in all the functions is also presented. The overall number of inputs required to cover the paths in all the functions analyzed across all the benchmarks ranges from 42 to 119 using TA-BFS and 36 to 130 for TA-DFS.

Figure 3 presents the improvements due to type-aware concolic testing. More specifically, we compute the percentage of number of inputs generated using type-aware testing compared to that generated by the existing approach. Since the existing approach can generate large number of inputs, we bound the input generation to 1000 per function. Out of 152 functions that are tested across all benchmarks, BFS and DFS approaches reach the upper bound of 1000 inputs for 60 (40%) and 16 (11%) functions respectively. In other words, concolic testing these functions can be significantly longer due to redundant exploration of paths.

Figure 3a presents the comparison between TA-BFS and BFS. In the figure, “<5%” segment shows the percentage of functions for which type-aware testing generated less than 5% of the inputs as compared to the original, “5-10%” shows the percentage of functions for which type-aware testing generated 5 to 10% of the inputs generated by the original approach and so on. We observe that for a significant percentage of functions, TA-BFS generates less than 5% inputs as compared to BFS. For example, B2 has approximately 80% of the functions that is present in this category. More-
over, of the 25 functions analyzed for B2, 14 functions timed out (reached the upper bound) using BFS. Even for the remaining functions that did not go beyond 1K inputs, the percentage of inputs generated with TA-BFS is quite low. There are a few functions across benchmarks for which input generation is not significantly affected due to type-awareness (e.g., 10% of the functions in B1, B2 and B3). Our manual inspection shows fewer property accesses in these functions. Consequently, they do not expose the potential problem with type-agnostic approaches. Even when there are a significant number of such functions (e.g., B4 and B5 have approximately 70%), few functions (30%) exist for which input generation reaches the upper bound. In these scenarios, TA-BFS outperforms BFS and reduces the number of inputs generated.

Figure 3b presents the corresponding improvement for the DFS variants of concolic testing. Here, the overall number of functions that reach the upper bound is lower (as compared to BFS). This is because of the difference in the way constraints are solved in both the approaches. Even with the reduction, the number of inputs generated by DFS is non-trivial. By employing TA-DFS, for 30% of the functions, we reduce the number of inputs generated by 95% (on average).

In Figure 4, we present the ratio of number of exceptions (e.g., an exception will happen when o.x is executed and o is undefined) during an execution when type-aware testing is employed to the exceptions in the original version. Based on Figure 3, we expect the number of exceptions to be quite low (here, less than 0.03) when executing with TA-BFS. Similarly, the ratio of the number of exceptions is also low with TA-DFS. This is a consequence of reducing the redundancy.

We tabulated the ratio of the representative inputs compared to total inputs that are generated for the type-aware variants. We define representative inputs as the inputs that do not encounter exceptions during an execution, and therefore can contribute to the function precondition. Figure 5 presents the ratios for TA-BFS and TA-DFS and are comparable for both the variants. Broadly, they range from 0.2 for B4 to 0.6 for B2. This shows that the functions that are considered have many properties that cause executions on many inputs generated by type-aware approaches to have exceptions. For B4 and B5, we verified that even though there are many functions with few inputs, the representative inputs is still lower than the overall inputs because of the exceptions.

Figure 6 presents the normalized execution time of the type-aware approaches with respect to the original approaches. The time taken, which ranges from 1.5 to three seconds per function, is significantly lower than the original approaches. Interestingly, even though the exceptions for B9 are relatively less, the time ratio of TA-DFS is nevertheless high. This is because of the actual time taken to execute the different functions in it is higher.

5.2 RQ2: Coverage of Type-aware Testing

In this section, we show that coverage is uncompromised due to type-awareness. We use Istanbul [9], a JS code coverage tool to track statements, branches and functions executed. Table 3 shows the coverage data for the benchmarks.
For example, in 3d-cube program, 97.5% statement coverage is obtained after testing 12 functions.

As shown in Table 3, for a majority of the benchmarks, the achieved coverage is closer to 100%. However, for B2, B9, B10, following are the reasons for poor coverage – presence (or return) of a function definition within a function body, and presence of constraints involving strings, array properties and complex expressions. Since the statements in the function definition cannot be counted as part of the encompassing function which is tested, we obtain code coverage closer to 100% after discarding these statements as shown in the parenthesis in Table 3. The problem with unsolvable constraints even for existing type-agnostic approaches is orthogonal to the proposed approach.

Though it is hard to get coverage numbers for BFS and DFS in many cases, we believe that the coverage remains unaffected since the search strategy remains the same. In addition, to demonstrate that the ability of concolic testing to find bugs is not diminished in the proposed approach, we inserted 20 defects randomly across all the benchmarks. After performing concolic testing, we observe that, both existing and type-aware concolic testing detected 17 defects.

For the remaining three defects, the defect was introduced at unreachable source locations (e.g., inside a condition with unsolvable constraint). As mentioned earlier, this is an orthogonal to the problem addressed in this paper.

Figure 7 Average number of inputs generated.

The number represents the average coverage achieved using intra-procedural type-aware concolic testing for that benchmark. For example, in 3d-cube program, 97.5% statement coverage is obtained after testing 12 functions.

As shown in Table 3, for a majority of the benchmarks, the achieved coverage is closer to 100%. However, for B2, B9, B10, following are the reasons for poor coverage – presence (or return) of a function definition within a function body, and presence of constraints involving strings, array properties and complex expressions. Since the statements in the function definition cannot be counted as part of the encompassing function which is tested, we obtain code coverage closer to 100% after discarding these statements as shown in the parenthesis in Table 3. The problem with unsolvable constraints even for existing type-agnostic approaches is orthogonal to the proposed approach.

Though it is hard to get coverage numbers for BFS and DFS in many cases, we believe that the coverage remains unaffected since the search strategy remains the same. In addition, to demonstrate that the ability of concolic testing to find bugs is not diminished in the proposed approach, we inserted 20 defects randomly across all the benchmarks. After performing concolic testing, we observe that, both existing and type-aware concolic testing detected 17 defects. For the remaining three defects, the defect was introduced at unreachable source locations (e.g., inside a condition with unsolvable constraint). As mentioned earlier, this is an orthogonal to the problem addressed in this paper.

Figure 7 presents the comparison of the average number of inputs generated across the three dimensions for all the benchmarks using TA-DFS. The average number of inputs generated when the callee implementations are provided without preconditions is approximately twice the number of inputs generated when implementations are not provided. Moreover, for most of the benchmarks, using the preconditions of the callees reduces the number of inputs generated by half as compared to when the preconditions are not used. As expected, these numbers are slightly more than the scenario when no callee implementations are provided.

We observe an interesting aspect pertaining to the results of B7 and B10. The number of inputs generated is lower while using preconditions as compared to when the implementation is not provided. Upon closer examination, we identify that having the callee implementation ensures that types of objects are determined and testing does not result in an exception in other parts of the caller function. However, when the implementation is not provided, these types are unavailable resulting in more exceptions in the caller.

Another interesting data point is the result on B3 in Figure 7. The number of inputs generated when the implementations are provided is independent of whether preconditions are used. This is because the normally terminating paths in the function is equal to the number of entries for the callee precondition. Therefore, employing preconditions does not reduce the number of inputs generated in that scenario.

Figure 8 presents the average number of exceptions and provides a comparison under the three scenarios. In general, the results validate our expectations in that the use of callee preconditions reduce the exception count. For B8, when the implementation is not provided, an exception happens in the caller because the callee is not recognized. On the other hand when preconditions are provided, even this exception disappears resulting in the absence of any exceptions with the use of preconditions for it.

The number of exceptions is also a function of the number of entries in the function preconditions. For example, for B4, we observe that the number of exceptions is more with preconditions than without the callee implementation. This is because of multiple entries corresponding to the same function in the preconditions. On the contrary, when multiple invocations are made without the callee implementations, each invocation will result in an exception.

5.3 RQ3: Inter-procedural testing

We also study the benefits of employing function preconditions in reducing the overall number of test inputs. For each benchmark, we pick three functions randomly such that it has at least one invocation to some other function. We perform concolic testing on each function using TA-DFS. This is because the experimental results in Section 5.1 demonstrate that the DFS variant of concolic testing is comparably better than DFS and BFS. We perform the testing as follows:

- Test the function with the implementation of its callees but without using preconditions, and
- Test the function in isolation without providing the implementation of its callees,
- Test the function with the implementation of its callees and use the preconditions for the callees in Section 5.1.

The derivation of function preconditions reduces the number of inputs generated for inter-procedural testing.
5.4 RQ4: Handling type-related crashes

Apart from using function preconditions in the context of inter-procedural concolic testing, the generated conditions can also be employed in automatically fixing ill-typed objects being passed to methods during a method invocation. We now demonstrate the use of the preconditions in automatically fixing malformed inputs.

Broadly, the idea is similar to failure oblivious computing [33] where the the crash is avoided by manufacturing values when a failure is expected. [33] uses this strategy to enhance server availability whereas we propose to use this strategy to avoid crashes in JS applications. Consider a function foo defined in Section 3.3, and corresponding preconditions in Table 2. When foo(5, 2, 1, {}) is invoked, we can avoid the crash by using the second precondition in Table 2. This is achieved by introducing a dummy bar function in the object o (i.e., last parameter) during runtime.

<table>
<thead>
<tr>
<th>Library</th>
<th>File</th>
<th>#Issue</th>
<th>Missing Property</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>hapi</td>
<td>response.js</td>
<td>2368</td>
<td>readableState</td>
<td>1</td>
</tr>
<tr>
<td>hapi</td>
<td>handler.js</td>
<td>2242</td>
<td>isBoom</td>
<td>2</td>
</tr>
<tr>
<td>pm2</td>
<td>child_process.js</td>
<td>808</td>
<td>close</td>
<td>12</td>
</tr>
<tr>
<td>pm2</td>
<td>monit.js</td>
<td>2</td>
<td>cpu</td>
<td>3</td>
</tr>
<tr>
<td>mocha</td>
<td>mocha.js</td>
<td>1496</td>
<td>callback</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4 Real crashes in JS programs. E: number of entries in the precondition.

Table 4 presents a sample of real crashes reported on different JS programs. Typically, these crashes happen when the method is invoked with malformed objects. Therefore, a dereference on such an object for a specific property results in the crash. For the purpose of detecting the correctness of the objects passed to a method, it is essential that function preconditions are available. As our experimental results demonstrate, deploying existing concolic testing approaches to derive preconditions can be infeasible due to the number of inputs generated. On the other hand, our type-aware approach generates the preconditions effectively. Moreover, because the preconditions are path-sensitive, we fix malformed objects along feasible paths.

We are able to generate preconditions for all the problematic functions reported in Table 4. The associated number of entries in the preconditions is also shown. For example, for class child_process, the number of entries in the precondition for function onexit is 12. In this case, when an input with an object that does not have the property close is used to invoke the method, we lookup the entries in the precondition and detect that the feasible path for the corresponding path requires the receiver, to contain close. Applying a variant of the procedure described in Algorithm 6 to fix the receiver ensures that the execution does not crash.

5.4 RQ4: Handling type-related crashes

Apart from using function preconditions in the context of inter-procedural concolic testing, the generated conditions can also be employed in automatically fixing ill-typed objects being passed to methods during a method invocation. We now demonstrate the use of the preconditions in automatically fixing malformed inputs.

Broadly, the idea is similar to failure oblivious computing [33] where the the crash is avoided by manufacturing values when a failure is expected. [33] uses this strategy to enhance server availability whereas we propose to use this strategy to avoid crashes in JS applications. Consider a function foo defined in Section 3.3, and corresponding preconditions in Table 2. When foo(5, 2, 1, {}) is invoked, we can avoid the crash by using the second precondition in Table 2. This is achieved by introducing a dummy bar function in the object o (i.e., last parameter) during runtime.

<table>
<thead>
<tr>
<th>Library</th>
<th>File</th>
<th>#Issue</th>
<th>Missing Property</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>hapi</td>
<td>response.js</td>
<td>2368</td>
<td>readableState</td>
<td>1</td>
</tr>
<tr>
<td>hapi</td>
<td>handler.js</td>
<td>2242</td>
<td>isBoom</td>
<td>2</td>
</tr>
<tr>
<td>pm2</td>
<td>child_process.js</td>
<td>808</td>
<td>close</td>
<td>12</td>
</tr>
<tr>
<td>pm2</td>
<td>monit.js</td>
<td>2</td>
<td>cpu</td>
<td>3</td>
</tr>
<tr>
<td>mocha</td>
<td>mocha.js</td>
<td>1496</td>
<td>callback</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4 Real crashes in JS programs. E: number of entries in the precondition.

Table 4 presents a sample of real crashes reported on different JS programs. Typically, these crashes happen when the method is invoked with malformed objects. Therefore, a dereference on such an object for a specific property results in the crash. For the purpose of detecting the correctness of the objects passed to a method, it is essential that function preconditions are available. As our experimental results demonstrate, deploying existing concolic testing approaches to derive preconditions can be infeasible due to the number of inputs generated. On the other hand, our type-aware approach generates the preconditions effectively. Moreover, because the preconditions are path-sensitive, we fix malformed objects along feasible paths.

We are able to generate preconditions for all the problematic functions reported in Table 4. The associated number of entries in the preconditions is also shown. For example, for class child_process, the number of entries in the precondition for function onexit is 12. In this case, when an input with an object that does not have the property close is used to invoke the method, we lookup the entries in the precondition and detect that the feasible path for the corresponding path requires the receiver, to contain close. Applying a variant of the procedure described in Algorithm 6 to fix the receiver ensures that the execution does not crash.

The function preconditions can help avoid crashes due to malformed objects.

6. RELATED WORK

Several approaches for automated test generation have been proposed [20, 38, 39, 35, 34]. Sen et al. [37] implement a “multi-path” symbolic execution [39] and variant of concolic testing [20, 38] to generate test inputs for JS programs. We show that [37] generates redundant inputs even for small programs. In this paper, we propose an approach that builds on top of Jalangi to reduce the redundancy by distinguishing type constraints and branch constraints.

Automatic testing of such applications [25, 36, 13, 27] has been studied extensively. SymJS [25] contains a symbolic engine for JavaScript and an automatic event explorer for web pages using taint analysis. Mishokraie et al. [28] present a technique to automatically generate test cases for individual JavaScript functions and event sequences to maximize function coverage. Wassermann et al. [40] use finite state transducers to solve string constraints and use real values when constraint resolution is incomplete. While the focus of these approaches is to generate proper event sequences or inputs for complex strings, we are generating minimum number of test inputs to cover JS programs.

Various type inference techniques [10, 21, 29] are proposed to generate optimized code or to detect type inconsistencies. Hackett and Guo [21] generate fast JS code by using derived type information. Ahn et al. [10] propose inline caching mechanisms for faster property look ups. Pradel et al. [29] use type inference to detect inconsistent types to expose optimization opportunities. These techniques perform type inference during JIT compilation. Our aim is to derive the appropriate types of variables starting from executions with undefined types. Also, due to imprecision, existing static type checkers (e.g., Flow [2]) are not suitable for our approach.

TypeScript [14] extends JavaScript with type annotations to prevent type inconsistencies. Feldthaus et al. [16] propose an approach to automatically detect errors in the TypeScript declaration files for JavaScript libraries. Rastogi et al. [32] improve soundness of TypeScript by enforcing stricter type checks to minimize performance overhead of runtime checks. Our work differs from these approaches as our technique does not require any programmer interference.

Several static analysis [22, 23, 26] tools are proposed to improve the reliability of JavaScript programs. Jensen et al. [22] propose a static analysis infrastructure that can infer detailed and sound type information using abstract interpretation. Kashyap et al. [23] developed an efficient abstract interpreter that takes care of several features including configurable sensitivity. Previous approaches either ignore or make assumptions about dynamic behavior. Since we employ testing, we do not suffer from these drawbacks.

7. CONCLUSIONS

We observe that conventional concolic testing approaches are not scalable for testing JS programs. This is a consequence of treating branch and type constraints, equivalently. We address the problem by proposing an effective approach that introduces type-awareness to reduce generation of redundant inputs. We extend the approach for inter-procedural testing by incorporating preconditions appropriately. We perform elaborate experimentation and demonstrate the scalability of type-aware concolic testing approaches over existing approaches. We also show the usefulness of the derived preconditions in avoiding crashes.

Acknowledgments

We thank the anonymous reviewers for their useful feedback. We are grateful to Malavika Samak and Rashmi Mudduluru for providing comments on an initial draft of the paper. The project is partially funded by a Google Faculty Research award and a Microsoft Research India travel grant.
References


