Jalangi: A Tool Framework for Concolic Testing, Selective Record-Replay, and Dynamic Analysis of JavaScript

Koushik Sen*
EECS Department
UC Berkeley, CA, USA.
ksen@cs.berkeley.edu

Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs
Samsung Research America
75 West Plumeria Drive, San Jose, CA, USA
{s.kalasapur,t.brutch,s.gibbs}@sisa.samsung.com

ABSTRACT

We describe a tool framework, called JALANGI, for dynamic analysis and concolic testing of JavaScript programs. The framework is written in JavaScript and allows implementation of various heavy-weight dynamic analyses for JavaScript. JALANGI incorporates two key techniques: 1) selective record-replay, a technique which enables to record and to faithfully replay a user-selected part of the program, and 2) shadow values and shadow execution, which enables easy implementation of heavy-weight dynamic analyses such as concolic testing and taint tracking. JALANGI works through source-code instrumentation which makes it portable across platforms. JALANGI is available at `https://github.com/SRA-SiliconValley/jalangi` under Apache 2.0 license. Our evaluation of JALANGI on the SunSpider benchmark suite and on five web applications shows that JALANGI has an average slowdown of 26X during recording and 30X slowdown during replay and analysis. The slowdowns are comparable with slowdowns reported for similar tools, such as PIN and Valgrind for x86 binaries.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution, testing tools*

General Terms

Verification

Keywords

JavaScript; Dynamic Analysis; Concolic Testing

1. INTRODUCTION

JavaScript is the language of choice for writing client-side web applications and is getting increasingly popular for writing mobile applications (e.g. web apps for Tizen OS and iOS), desktop apps (e.g. apps for Windows 8 and Gnome Desktop), and server-side applications (e.g. node.js). However, there are not that many tools available for analysis, testing, and debugging of JavaScript applications. We have developed a simple yet powerful framework, called JALANGI, for writing heavy-weight dynamic analyses for JavaScript. In this paper, we briefly describe the framework and its usage scenarios. The framework provides a few useful abstractions and an API that significantly simplifies implementation of dynamic analyses for JavaScript. A detailed description of the techniques underlying JALANGI can be found in [6].

JALANGI works on any browser or `node.js`. We achieve browser independence through selective source instrumentation. JALANGI can operate even if certain source files are not instrumented. An analysis in JALANGI works in two-phases. In the first phase, an instrumented JavaScript application is executed and recorded on a user selected platform (e.g. mobile chrome running on Android). In the second phase, the recorded data is utilized to perform a user specified dynamic analysis in a desktop environment.

JALANGI allows easy implementation of a dynamic analysis through the support of *shadow values* and *shadow execution*. *Shadow values* enable us to associate a shadow value with any value used in the program. In JALANGI, we have implemented several dynamic analyses using shadow values and execution: 1) concolic testing, 2) pure symbolic execution, 3) tracking origins of `null` and `undefined` [1], 4) detecting likely type inconsistencies, 5) a simple object allocation profiler, and 6) a simple dynamic taint analysis.

2. TECHNICAL DETAILS

We provide a summary of the technical details of JALANGI. For a more detailed technical discussion, please refer to [6]. The user identifies a subset of a web application, for record and replay, which is instrumented by JALANGI. The resulting instrumented code is executed on a platform of user's choice, during the *recording* phase. Even though a subset of user-code is instrumented, the entire application is executed during the recording phase, inclusive of instrumented and uninstrumented JavaScript code, and native code. However, only the instrumented code is replayed by JALANGI during the *replay* phase. The ability of JALANGI to execute the JavaScript application in its entirety on user's platform, enables the recorded execution to be replayed on a development laptop/desktop JavaScript engine for debugging, inclusive of mobile browser, and node.js based systems, or an in-

*The work of this author was supported in full by Samsung Research America.

ESEC/FSE '13, August 18-26, 2013, Saint Petersburg, Russia

tegrated development system with an embedded JavaScript engine. This approach also enables the implementation of dynamic analyses with underlying shadow values, which support shadow executions.

Record and replay can be effectively provided, by recording every value loaded from memory during execution, and re-using them during the associated memory loads during replay. This approach, though sound, is not without its challenges, such as: (1) How to effectively record functions and objects? (2) How to provide for replay, when un-instrumented of native function(s) (for example, the JavaScript event dispather) call instrumented functions? The first issue is resolved by providing indirect recording, where a unique numerical identifier is associated with every object and function, and by recording the values of these identifiers. The second issue is addressed by explicit recording and calling of instrumented functions, which in turn are called from un-instrumented code, or are executed as a result of being dispatched by the event handler of JavaScript.

Furthermore, we observed that the value of a memory load during replay can be computed through execution of the instrumented code, without the need to record the values of all memory loads. This was used to improve the efficiency of JALANGI, by recording only the necessary memory loads. To identify if the value of a memory load must be recorded, JALANGI keeps track of shadow memory during recording phase, which is updated along side actual memory, as the instrumented code is executed. No updates to shadow memory result from the execution of native and un-instrumented code. To ensure availability of correct values during the replay phase, JALANGI stores the value of memory loads only if a difference is found in the value stored in a memory location during the recording phase (i.e. if a difference is found between the values of the actual memory being loaded and its associated value stored in the shadow memory).

In JALANGI, any value used in execution during the replay phase may be replaced by an *annotated value*, which may provide additional information for the actual value used. An example would be extra taint information needed to support taint analysis, or it may be information related to actual values in symbolic execution, which may be provided in the form of a symbolic expression. JALANGI uses objects of type `ConcolicValue`, to denote annotated values.

3. DYNAMIC ANALYSES

In JALANGI, we have implemented the following dynamic analyses:

- Concolic testing [2, 7]: Concolic testing performs symbolic execution along a concrete execution path, generates a logical formula denoting a constraint on the input values, and solves a constraint to generate new test inputs that would execute the program along previously unexplored paths. Concolic testing in JALANGI supports constraints over integer, string, and object types and *novel type constraints.* We introduced type constraints to handle the dynamic nature of JavaScript—the type of an input variable could be different for different feasible execution paths of the program.

- Pure symbolic execution: Pure symbolic execution executes the program symbolically and never restarts the program for the purpose of backtracking. It checkpoints the state before executing a branch statement, executes one branch, and later backtracks with the checkpointed state to explore the other branch. For small programs, pure symbolic execution avoids time wastage due to repeated restarts.

- Tracking origins of `null` and `undefined` [1]: This analysis records source code locations where null and undefined values originate and reports the location when an error occurs dues a null or undefined. Whenever there is an error due to such literals, such as accessing the field of a null value, the shadow value of the literal is reported to the user. Such a report helps the programmer to easily identify the origin of the null value.

- Detecting Likely Type Inconsistencies: The dynamic analysis checks if an object created at a given program location can assume multiple inconsistent type. It computes the types of object and function values created at each definition site in the program. If an object or a function value defined at a program location has been observed to assume more than one type during the execution, the analysis reports the program location along with the observed types. Sometimes these kind of type inconsistencies could point us to a potential bug in the program. We have noticed such issues in two SunSpider benchmark programs.

- Simple Object Allocation Profiler: This dynamic analysis records the number of objects created at a given allocation site and how often the object has been accessed. It reports if the objects created at a given allocation site are read-only or a constant. It also reports the maximum and average difference between the object creation time and the most recent access time of the object . If an allocation site creates too many constant objects, then it could lead to memory inefficiency. We have found such a problem in one of the web applications in our benchmark suite.

- Dynamic taint analysis [4]: A dynamic taint analysis is a form of information flow analysis which checks if information can flow from a specific set of memory locations, called sources, to another set of memory locations, called sinks. We have implemented a simple form of dynamic taint analysis in JALANGI. In the analysis, we treat read of any field of any object, which has not previously been written by the instrumented source, as a source of taint. We treat any read of a memory location that could change the control-flow of the program as a sink. We attach taint information with the shadow value of an actual value.

4. IMPLEMENTATION

JALANGI is available at `https://github.com/SRA-SiliconValley/jalangi`. We have implemented JALANGI in JavaScript.

JALANGI operates by instrumenting JavaScript code. Table 3 shows code obtained after instrumentation of the code in Table 1. During instrumentation, JALANGI inserts various callback functions from the JALANGI library. The callback functions are listed in Table 2. These functions wrap

```
var a = {x:1, y:2};

function f2 (c) {
 if (c > 5 )
   a.y = a.x + c;
 return c;
}

f1(12);
```

Table 1: Sample Code Before Instrumentation

```
J$.U(iid, op, oprnd); // wrapper for unary operations
J$.B(iid, op, left, right); // wrapper for binary operations
J$.C(iid, cond); // wrapper for conditional branches
J$.C1(iid, key); // wrapper for the key of a switch statement
J$.C2(iid, case); // wrapper for a case label of a switch
J$._(); // returns last value passed to J$.C

J$.H(iid, val); // wrapper for hash used in for-in
J$.I(val); // ignore argument
J$.G(iid, base, offset); // wrapper for getField
J$.P(iid, base, offset, val); // wrapper for putField
J$.R(iid, name, val); // wrapper for local variable read
J$.W(iid, name, val, lhs); // wrapper for local variable write
J$.Niid, name, val, isArgument); // wrapper for initialization
J$.T(iid, val, type); // wrapper for a object/function/regexp/array literal
J$.F(iid, f, isConstructor); // wrapper for a function call
J$.M(iid, base, offset, isConstructor); // wrapper for a method call
J$.A(iid, base, offset, op); // wrapper for +=, -=, ...
J$.Fe(iid, val, dis); // callback at function entry
J$.Fr(iid); // callback at function return
J$.Se(iid, val); // callback at script entry
J$.Sr(iid); // callback at script exit
J$.Rt(iid, val); // wrapper for value being returned
J$.Ra(); // callback for grabbing return value

J$.makeSymbolic(symbol, val); // make a value symbolic
J$.addAxiom(formula, branch); // adds a constraint to the path constraint
J$.endExecution(); // callback at the end of an execution
```

Table 2: Callback Functions from Instrumented Code

```
if (typeof window === 'undefined') {
  require('/user/jalangi/src/js/analysis.js');
  require('/user/jalangi/src/js/InputManager.js');
  require('/user/jalangi/src/js/instrument/esnstrument.js');
  require(process.cwd() + '/inputs.js');
}
{
  try {
    J$.Se(73, 'tests/unit/instrument-small_jalangi_.js');
    J$.N(77, 'a', a, false);
    J$.N(85, 'f2', J$.T(81, f2, 12), false);
    var a = J$.W(17, 'a', J$.T(13, {
      x: J$.T(5, 1, 22),
      y: J$.T(9, 2, 22)
    }, 11), a);
    function f2(c) {
      jalangiLabel0:
        while (true) {
          try {
            J$.Fe(49, arguments.callee, this);
            J$.N(53, 'arguments', arguments, true);
            J$.N(57, 'c', c, true);
            if (J$.C(4, J$.B(6, '>', J$.R(21, 'c', c), J$.T(25, 5, 22))))
              J$.P(45,J$.R(29,'a',a),'y',J$.B(10,'+',J$.G(37,J$.R(33,'a',a),'x'),J$.R(41,'c',c)));
            return J$.Rt(97, J$.R(49, 'c', c));
          } catch (J$e) {
            throw J$e;
          } finally {
            if (J$.Fr(93))
              continue jalangiLabel0;
            else
              return J$.Ra();
          }
        }
    }
  J$.F(69,J$.I(typeof f1==='undefined'?J$.R(61,'f1',undefined):J$.R(61,'f1',f1)),false)(J$.T(65,12,22))
  } catch (J$e) {
    throw J$e;
  } finally {
    J$.Sr(89);
  }
}
// JALANGI DO NOT INSTRUMENT

//@ sourceMappingURL=instrument-small_jalangi_.js.map
```

Table 3: After Instrumentation of Code in Table 1

Benchmark	LOC	Records	SlowR	Slowdown in Replay		
				empty	taint	track
3d-cube	339	3670	18.33	25.16	28.67	26
3d-morph	56	6	18.2	33.2	35.83	33.6
3d-raytrace	443	79791	38.17	29.05	30.5	35
b-trees	52	146048	57.8	40	42.4	42.8
fannkuch	68	246	40.6	76.4	73	80.4
nbody	170	78	19	25.8	25.67	24.16
nsieve	39	5	16.4	23.6	30	24.2
3bit-in-byte	38	1	16.6	29	31	30.2
bits-in-byte	26	1	25	25	51.4	47
bitwise-and	31	1	12.83	21.83	29.2	26.2
controlflow	25	1	20	33.2	34.6	28.33
crypto-md5	288	42	12	18	22.2	22
crypto-sha1	225	52	13.4	19.4	21	21.2
date-tofte	300	32018	92.16	92.67	92.83	95.5
date-xparb	418	95715	29.83	21	22.67	25.67
math-cordic	101	8	29.6	35.6	45.4	40.17
partial-sums	33	5	14.6	23.4	22.16	23.8
spectral-norm	51	15	19.8	25.2	29.2	29.4
regexp-dna	1714	42	2	4	3.17	3.8
string-fasta	90	56947	40.17	30.33	34.5	38.6
string-tagcloud	266	117577	51.42	50.86	44	42.8
string-unpack	67	193057	29.88	13.25	13.75	17
nsieve-bits	35	3	20	36.6	45.4	40
crypto-aes	425	23926	19	21	23.67	23
string-validate	90	60	1.5	1.5	1.4	1.5
string-base64	136	40965	25	27.2	29.6	29.2
annex	9663	87623	-	-	-	-
calculator	787	1288	-	-	-	-
go	10,039	114609	-	-	-	-
tenframe	1491	4656	-	-	-	-
shopping	5397	1144	-	-	-	-

Table 4: Results: "Records" column reports number of values recorded, "SlowR" reports slowdown during recording compared to normal execution.

the various operations in JavsScript. The selective record-replay engine of JALANGI is implemented by defining these callback functions.

JALANGI exposes the instrumentation library as a function `instrumentCode`. This enables us also to dynamically instrument any code that is created and evaluated at runtime. For example, we modify any call to `eval(s)` to `eval(instrumentCode(s))`.

5. PERFORMANCE OF JALANGI

We ran JALANGI's record-replay on 26 programs in the JavaScript SunSpider (`http://www.webkit.org/perf/sunspider/sunspider.html`) benchmark suite and on five web apps written for the Tizen OS using HTML5/JavaScript. (`https://developer.tizen.org/downloads/sample-web-applications`). Table 4 shows the overhead associated with the record phase and with the three dynamic analyses: no analysis (denoted by *empty*), tracking origins of null and undefined (denoted by *track*), and a taint analysis (denoted by *taint*). We also report the number of values we recorded for each benchmark program. The experiments were performed on a laptop with 2.3 GHz Intel Core i7 and 8 GB RAM, running the web apps on Chrome 25 and performed the replay executions on node.js 0.8.14.

We did not measure the slowdown of the web apps because these are mostly interactive applications. For the SunSpider benchmark suite, we observed an average slowdown of 26X during the recording phase with a minimum of 1.5X and a maximum of 93X. On the *empty* analysis during the replay phase, we observed an average slowdown of 30X with a minimum of 1.5X and a maximum of 93X. *Track* analysis showed an average slowdown of 32.75X with a minimum of 1.5X and a maximum of 96X.

5.1 Issues Detected by JALANGI Dynamic Analyses

JALANGI's likely type inconsistency checker found that the function `CreateP` in `3d-cube.js` of the SunSpider benchmark suite is mostly used as a constructor, but at one location it was called as a function. As result of the function invocation, the program creates an unnecessary `V` field in the global object. We believe that this call is a possible programming error.

JALANGI's object allocation profiler noticed that the method `getValue(place, _board)` in the Annex game webapp creates a constant object thousands of times. We believe that such unnecessary creation of the constant object can be avoided by hoisting the constant object outside the method.

6. RELATED WORK

To our best knowledge JALANGI is the first dynamic analysis framework for JavaScript. There are few tools that could perform record-replay of JavaScript programs. JSBench [5] uses record-replay mechanisms to create JavaScript benchmarks. Mugshot [3] is captures all events to deterministically replay executions of web applications. Ripley [8] replicates execution of a client-side JavaScript program on a server side replica.

7. CONCLUSION

JALANGI has taken care of various challenging details of JavaScript. One can easily implement a dynamic analysis in the JALANGI framework since all the worrisome corners of JavaScript are handled. We expect that JALANGI will facilitate future research on dynamic analysis of JavaScript.

8. REFERENCES

[1] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 405–422, 2007.

[2] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI'05*, June 2005.

[3] J. Mickens, J. Elson, and J. Howell. Mugshot: deterministic capture and replay for JavaScript applications. In *7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 11–11, 2010.

[4] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium*, 2005.

[5] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 677–694. ACM, 2011.

[6] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE'13*, August 2013. To appear.

[7] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE'05*, Sep 2005.

[8] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *16th ACM conference on Computer and communications security*, pages 173–186. ACM, 2009.