SPEjs: A Symbolic Partial Evaluator for JavaScript

Sümeyye Süslü
Computer Science & Engineering Department
University of Texas at Arlington
Arlington, TX, USA

Christoph Csallner
Computer Science & Engineering Department
University of Texas at Arlington
Arlington, TX, USA

ABSTRACT

Partial evaluation is widely performed statically, to perform a source to source transformation on a source program that yields a specialized source program. A key observation is that current partial evaluation schemes perform fast but relatively shallow static analyses. In this paper we propose to deepen the reach of such partial evaluation schemes by selectively adding local symbolic execution. Concretely, we describe the SPEjs symbolic partial evaluator for JavaScript that is built on Babel and the SMT solver Z3. To gauge the promise of this approach we compared SPEjs with Facebook's state-of-the-art partial evaluator Prepack. Our results on a set of micro benchmarks and Prepack's test suite indicate that, within Prepack's runtime budget, SPEjs was able to simplify additional expressions and therefore remove dead code branches that Prepack failed to remove, yielding smaller residual programs.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing theory, concepts and paradigms**; • **Software and its engineering → Compilers**;

KEYWORDS

Mobile applications, partial evaluation, symbolic execution

ACM Reference Format:
Sümeyye Süslü and Christoph Csallner. 2018. SPEjs: A Symbolic Partial Evaluator for JavaScript. In *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis (A-Mobile '18), September 4, 2018, Montpellier, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3243218.3243220

1 INTRODUCTION

Partial evaluation is widely performed statically, to perform a source to source transformation that yields specialized code [10, 11]. A key observation is that current partial evaluation schemes perform fast but relatively shallow static analyses. For example, existing techniques check if all variables in a given expression are already resolved to constants. But existing techniques typically do not perform further analysis on expressions that contain non-resolved variables. The result is that current techniques leave open opportunities for more aggressive program specialization.

The size of the code shipped to users and its execution speed are important. For example, the Facebook mobile application runs daily on hundreds of millions of smart phones. It consists of a relatively small native portion, whereas the majority is written in JavaScript. Minimizing the size and execution speed of this JavaScript code is the goal of Facebook's partial evaluator Prepack [7]. Prepack mainly compacts the program's initialization code. This code calls general-purpose libraries and uses variables that are ultimately assigned to constants, opening many code size reduction opportunities for partial evaluation. In such a setting, it would make sense to extend partial evaluation with more heavy-weight program analysis techniques, to further reduce the size of the specialized program (which is then shipped to hundreds of millions of users).

Beyond the Facebook mobile app, JavaScript is one of the most widely used programming languages in general. For example, as of July 2018, 94.8% of all web sites use JavaScript [33]. In addition to web sites, JavaScript is widely used in iOS and Android applications as well as in server-side components of mobile and web apps.

It is well-known that most program analysis questions are undecidable in general, even for simple programming languages. Large and dynamic languages such as JavaScript pose many additional technical challenges. Previous partial evaluation research has made significant progress on supporting dynamic language features such as pointers [4, 16], dynamic method call dispatch [6, 24], Java reflection [27], native x86 code [28], removing object allocations and runtime type checks [1], and implementing high-performance dynamic language virtual machines [34]. While these are all important contributions, we are not aware of approaches that enrich partial evaluation with symbolic execution to further specialize code.

Fully symbolic execution can be very expensive and may not be the right fit for partial evaluation, even in a high-value setting such as the Facebook example. So a key challenge is how to selectively extend traditional partial evaluation with deeper program analysis, to get some benefits while avoiding the excessive runtime that, for example, full symbolic execution would impose.

The most closely related approach is Prepack, which is a state-of-the-art partial evaluator for JavaScript. While it uses program analysis techniques such as abstract interpretation, it currently does not deeply analyze expressions that contain abstract variables (i.e., variables that do not have a clear value assignment). Also related are Jeene and Google's Closure compiler, which both partially evaluate JavaScript code but are more limited [8, 13].

This is the first paper that enriches partial evaluation with symbolic execution for the purpose of further specializing the analyzed code. In our initial formulation, the partial evaluation traverses a program's abstract syntax tree and identifies expressions that could be resolved via constraint solving. (A more traditional treatment

A-Mobile '18, September 4, 2018, Montpellier, France

ACM ISBN 978-1-4503-5973-3/18/09…$15.00
https://doi.org/10.1145/3243218.3243220

of traversing a CFG is future work.) Our approach infers from an abstract syntax tree (AST) partial execution traces, encodes them as queries for a SMT solver, and interprets the solver's results as program values, which can allow further partial evaluation. We implement a prototype symbolic partial evaluator for JavaScript in the SPEjs tool on top of Babel and Z3 [14, 17]. We then compare SPEjs to Prepack on a set of micro benchmarks. On these benchmarks, SPEjs could specialize programs more aggressively than Prepack, leading to specialized programs of smaller size and shorter runtime. To summarize, this paper makes the following contributions.

- This paper presents the first approach to enriching traditional partial evaluation with symbolic execution.
- We evaluate this technique by implementing it in the SPEjs prototype tool for JavaScript and comparing it with the state-of-the-art tool Prepack on a set of micro benchmarks.
- In some cases, SPEjs yielded specialized programs that are both smaller and faster than Prepack-generated code. SPEjs is available as open-source software [30].

2 MOTIVATING EXAMPLE

Listing 1 shows a small code example with three nested if statements, where (at least) one branch is infeasible. However at this point we ignore the semantics of the `Date.now` function and just assume it can return any value. Regardless of the `Date.now` semantics, the second branch can never be executed, as `x` cannot be smaller than four and larger than eight at the same time.

```
var k = Date.now();
function foo(x) {
  var a;
  if (x < 0)                a = 4;
  else if (x < 4 && x > 8)  a = 6;
  else if (x > 0)           a = 8;
  return a;
}
var t = foo(k);
```

Listing 1: Motivating example: The `foo` function contains an infeasible branch ("`a=6`").

```
(function () {
  var _$1 = this;
  var _$0 = _$1.Date.now();
  var _F = _$0 < 0;
  var _B = _$0 < 4;
  var _D = _$0 > 8;
  var _A = _B && _D;
  var _7 = _$0 > 0;
  var _4 = _7 ? 8 : void 0;
  var _2 = _A ? 6 : _4;
  var _0 = _F ? 4 : _2;
  t = _0;
}).call(this);
```

Listing 2: Residual code produced by the state-of-the-art partial evaluator Prepack for the Listing 1 code.

While the Listing 1 example is simplistic, clearly a larger program could involve longer execution traces and thus expressions that are harder to reason about. However the key observation remains that variable `x` is being assigned a value that is not known statically. So a traditional (static) partial evaluator such as Prepack quickly gives up on expressions that contain such an "abstract" variable as `x`, without analyzing or partially evaluating such expressions further, yielding Listing 2, which is essentially equivalent to Listing 1, keeping all three branches.

To evaluate code such as Listing 1 further than existing partial evaluators do, SPEjs collects symbolic expressions during partial evaluation and converts them to SMT queries. In this example, SPEjs produces one query per branch condition. Since the SMT solver does not find a satisfiable assignment for the second branch condition, SPEjs infers that the expression is always `false`, which in this scenario means that an entire branch is infeasible. Listing 3 shows SPEjs's residual code, which is shorter, eliminates unnecessary branches, and empirically runs faster.

```
function foo(x) {
  var a;
  if (x < 0)       a = 4;
  else if (x > 0) a = 8;
  return a;
}
```

Listing 3: Residual code SPEjs produces for the Listing 1 code. In contrast to a traditional partial evaluator (Listing 2), SPEjs has removed the infeasible "`a=6`" branch.

3 BACKGROUND

This section provides necessary background information on partial evaluation, the Babel JavaScript source-to-source compiler, symbolic execution, and the Microsoft Z3 SMT solver.

Partial evaluation is a program transformation technique that creates a specialized version of the input program, by evaluating expressions whose variables have known values. Such a variable whose value is already known is also called *static*, whereas a variable whose value is not yet clear is called *dynamic*. Evaluating expressions in this manner allows removing dead branches, propagating constants throughout the program, inlining function calls, and unrolling loops. Partial evaluation is often used to create a version of a function that is specialized for a specific input value. A classic example is the power function, e.g., as shown in Listing 4.

```
function power(n,x) {
  var p=1;
  while (n>0) {
    if (n%2===0) {
      x = x*x;
      n = n/2;
    } else {
      p = p*x;
      n = n-1;
    }
  }
  return p;
}
```

Listing 4: The original power function takes two input parameters, `n` and `x`.

If we know that the program will be computing the cube of x several times, it may make sense to include a version of power that is specialized to n==3. With n fixed to three, a partial evaluator can then derive the specialized version shown in Listing 5.

```
function power_3(x) {
  return x*x*x;
}
```

Listing 5: A partial evaluator has specialized the Listing 4 code to this residual power function for n==3.

In the context of JavaScript applications, such a situation often arises with initialization code, which typically is written in a generic form, e.g., calling functions such as power, but often or always with the same fixed input such as n==3, where n typically is a hard-coded configuration parameter.

Generally speaking, a partial evaluator does not change execution semantics, so passing n==3 and x to the Listing 4 code always yields the same result as passing x to the Listing 5 code (for all values of x). In the notation from the literature [19]: $[[power]]_{js}\ [3, x] = [[power\_3]]_{js}\ x$.

3.1 Babel

Babel is a source-to-source compiler (or transpiler) for JavaScript [14]. It is widely used to convert JavaScript code written in ES6 to other versions of JavaScript. Figure 1 gives an overview of Babel's three main phases: parsing source code to an AST, transforming the AST, and converting the AST back to source code.

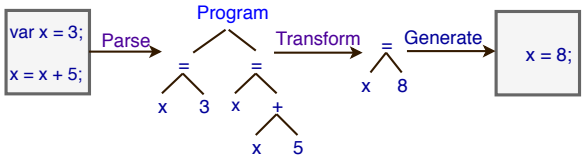

Figure 1: Overview of Babel's three main stages.

3.2 Symbolic Execution

Symbolic execution is a program analysis technique that evaluates a given program not on concrete input values but on symbolic variables [12]. At runtime, a symbolic execution engine thus represents a runtime value as a symbolic expression over the program's input variables. A symbolic execution engine iteratively traverses a program's control-flow graph and creates a symbolic execution tree. It considers all possible paths during the generation and decides path feasibility using a constraint solver.

```
  var x, y, z; // Symbolic
  var a = 0; b = 0; c = 0;
  if (x > z)        {a = 2;}
  else if (y < x)
  {
    if (z < 3)      {b = -2;}
    c = 4;
  }
}
```

Listing 6: Example JavaScript code.

As an example, Figure 2 shows the result of full symbolic execution of the Listing 6 example program. While traversing a program's control flow graph, symbolic execution adds the outcome of each taken branch decision as a conjunct to the current path condition. At each point in the traversal, the symbolic execution engine can call a constraint solver to check if the current path condition is still satisfiable.

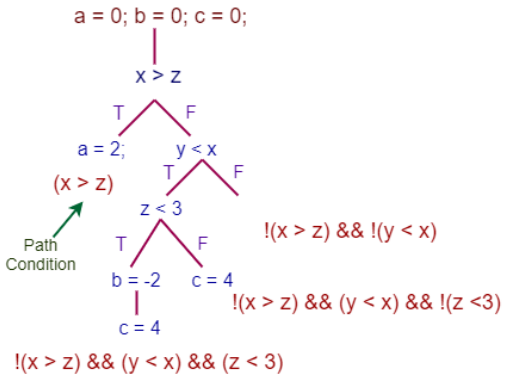

Figure 2: Symbolic execution tree of the Listing 6 code.

3.3 Analyzing JavaScript with the Microsoft Z3 SMT Solver

Constraint solvers are widely used by program analysis tools [5]. Satisfiability Modulo Theories (SMT) solvers are both expressive and powerful, since they contain theories and reasoning strategies for data structures that are commonly used in programming languages, such as bit-vectors to model integer types and their overflow behavior.

Z3 is a modern SMT solver that answers both satisfiability questions and produces sample solutions for such questions. While Z3 provides native bindings for several languages including Python and C#, there is currently no native binding to JavaScript. A common workaround is expressing queries for Z3 in the standard SMT2 language and passing such queries as text files to Z3 [18].

4 SPEJS OVERVIEW AND DESIGN

This section proposes a unified solution SPEjs for the partial evaluation problem using the design principles of partial evaluators and symbolic execution techniques. We designed SPEjs as a Babel plugin and combined it with Microsoft's Z3 SMT solver to overcome the mentioned issues.

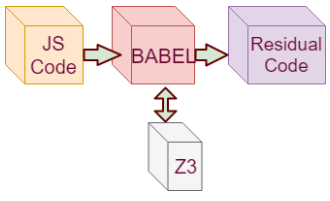

Figure 3: High-level overview of SPEjs's workflow.

Figure 3 gives an overview of SPEjs's workflow. At a high level, SPEjs uses the Babel transpiler to convert a program's JavaScript code into an AST and traverse the AST. SPEjs uses the Microsoft Z3 SMT solver to decide if branches are feasible. After SPEjs performs partial evaluation, it uses Babel to export the resulting AST to a residual JavaScript program.

4.1 Partial Evaluator Design

The design of a symbolic partial evaluator shares many aspects with the design of a corresponding traditional partial evaluator. (1) First, both types of partial evaluator traverse an abstraction of the program, which is typically a control-flow graph. (Our prototype implementation traverses an AST, but later versions will traverse a CFG.) (2) Second, both types maintain environment information on the state of the program's variables. In the traditional case, the environment keeps track of a variable's concrete value (if it is known) or the fact that the concrete value is currently unknown.

(3) Third, once a variable has been resolved to a concrete value, both evaluator types propagate this value throughout the program abstraction. (4) Finally, where an expression only contains resolved variables, both evaluator types evaluate the expression and iteratively propagate the resulting value. Both types of evaluators perform standard transformations such as expression folding, loop unrolling, function inlining, and elimination of dead code branches.

However the design of a symbolic partial evaluator diverges from its traditional counterpart, as its environment can in some scenarios selectively maintain for some program variables symbolic expressions, similar to the environment a symbolic execution engine maintains during symbolic execution. A key difference to a symbolic execution engine is that a symbolic partial evaluator can operate much more locally (and therefore be more lightweight) while still maintaining its utility. For example, for code such as Listing 1, a symbolic partial evaluator does not need to maintain any symbolic expressions for any of the program variables but can still achieve the desired effect of removing an infeasible branch, even if the code contained additional assignment statements that mutate the x variable.

A symbolic partial evaluator can then convert symbolic expressions to satisfiability queries for an SMT solver. In the motivating example of Listing 1, we can convert each branch condition to an SMT satisfiability query, to determine if the branch is infeasible and can thus be removed from the AST.

Beyond these difference, the usual caveats of static analysis apply, such as merging the updates performed in two branches of an if statement. Our prototype implementation just represents such an update in the environment with a symbolic variable.

4.2 Implementation with Babel and Z3

We developed a custom Babel plugin using a visitor pattern. SPEjs creates SMT files and passes them to a Node.js child process running Microsoft Z3. The design of the communication, processing, and resolving tasks are adapted from the design of the Leena symbolic execution engine for JavaScript [18].

Figure 4 shows two examples of SPEjs's processing. Figure 4a shows an example infeasible if-condition as it may appear in a JavaScript program. SPEjs builds its AST and converts the relevant

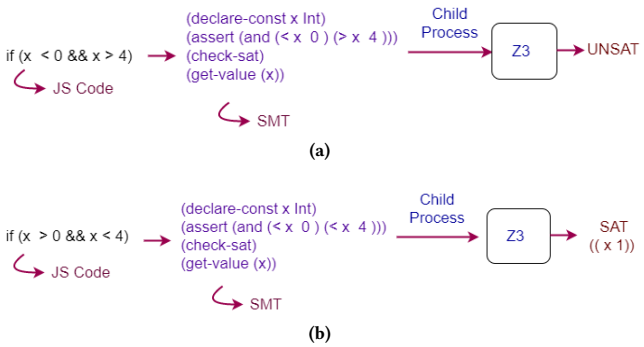

Figure 4: Stages for the resolution of path conditions using Z3.

AST sub-tree into SMT conditions. SPEjs currently only supports integral types. Support for more complex types such as strings is future work. For this example, Z3 cannot find any satisfying solution. So SPEjs proceeds to remove the corresponding if-branch from the AST. Figure 4b gives an example of a satisfiable if-condition. Since Z3 can find a satisfying assignment, SPEjs judges the branch feasible and at this point does not modify the program's AST.

5 EXPERIMENTS AND RESULTS

To evaluate symbolic partial evaluation, we would like to know both if symbolic partial evaluation (i.e., as implemented in SPEjs) is feasible with realistic resources and if the resulting residual programs are smaller and run faster than both the input programs and, more importantly, programs obtained via a traditional state-of-the-art partial evaluator (i.e., Prepack). We thus investigate the following three research questions (RQ), expectations (E), and hypotheses (H).

- RQ1: What is SPEjs's runtime compared to Prepack?
  - E1: Due to the traditionally high overhead of symbolic execution and constraint solving, we expect SPEjs to take longer than Prepack.
  - H1: If SPEjs issues SMT solver queries then SPEjs takes longer than Prepack.
- RQ2: How does the original program's runtime compare to the runtime of the programs produced by SPEjs and Prepack?
  - E2: On program constructs both SPEjs and Prepack support we expect SPEjs to perform more aggressive partial evaluation.
  - H2: On certain micro benchmarks SPEjs-produced programs should have the shortest runtime.
- RQ3: How does the original program's code size compare to the size of the code produced by SPEjs and Prepack?
  - E2: On program constructs both SPEjs and Prepack support we expect SPEjs to perform more aggressive partial evaluation.
  - H2: On certain micro benchmarks SPEjs-produced programs should have the smallest code size.

To explore these research questions, we use two sets of benchmarks. The first set consists of micro benchmarks we have handcrafted such that they only use language features supported by

both SPEjs and Prepack. These benchmarks are available on the SPEjs web site [30]. The second set is a subset of the test cases in Prepack's public code repository. We picked these tests because they have a relatively high chance that Prepack can handle them. (As we later found out, the public Prepack version we used could in fact only handle a relatively small portion of these test cases.)

We ran all experiments in the same environment, which consisted of Node.js version 9.2.0 on a 6 GB RAM virtual machine running Ubuntu 16.04 LTS. The VM's host platform had a 2.2 GHZ 64 bit i5-5200 processor, 12 GB RAM, and Windows 10 OS. We used Prepack version 0.2.19-alpha.0.

We measured tool runtimes using Unix's built-in time function. We used the *chrt* mechanism to raise the task priority to near real time and computed the average over 100 runs for each sample. For the original and residual code runtimes, we used Benchmark.js [3], which is a statistical benchmarking tool for JavaScript. To determine code size we used jsmeter [21], which counts program statements instead of code lines.

5.1 Micro Benchmark: Seven Sample Programs

The micro benchmarks combine various basic JavaScript language features that both SPEjs and Prepack are designed to handle. Listings 7 and 8 show two representative examples.

```
var z = Date.now();
var b = Date.now();
function foo (z, b) {
  var x = 5, a = 9;
  if (z>2 && z<0) { b = 5 - a;         }
  else if (x>=5)  { a = 2;             }
  else            { a = a+2; b = b-1; }
  a = a - 6;
  return a;
}
var a = foo(z, b);
```

Listing 7: Sample 6 writes to various variables.

The Listing 7 code writes to the same variable in two branches. Ruling out some of these branches as infeasible will enable a partial evaluator to reflect the correct state update in its symbolic state, which in turn allows subsequent code simplifications.

```
var a = foo(Date.now());
function foo (a) {
  var x = 0;
  while (a < 10) {
    a += 1;
    x += 1;
  }
  if (a < 5)
    var y = 2;
}
```

Listing 8: Sample 7: Infeasible if-branch after while loop.

The Listing 8 code branches on the value of a variable that a previous while loop branches on and updates in its loop body. Keeping track of the symbolic expression of this variable will enable a partial evaluator to decide the feasibility of the subsequent if-branch.

5.2 Micro Benchmark Results

Table 1 gives an overview of the micro benchmark experiments. Prepack could not process Sample 7.

Table 1: Results on micro-benchmark samples: Both SPEjs itself and SPEjs-generated programs had in most cases a lower runtime than with Prepack (PP).

	Tool [ms]		Residual [ns]			Residual [stmt]		
#	PP	SPEjs	Orig.	PP	SPEjs	Orig.	PP	SPEjs
1	1160	607	1.51	1.43	1.46	9	6	9
2	1114	652	1.55	1.34	1.30	14	5	8
3	1263	644	1.54	1.41	1.44	9	6	7
4	1213	670	1.65	1.60	1.63	9	8	9
5	1147	842	1.68	1.54	1.51	11	11	8
6	1190	721	1.56	1.54	1.46	17	9	1
7	1103	670	1.71	n/a	1.37	10	n/a	7

5.2.1 RQ1: SPEjs With Lower Runtime. Despite making relatively heavy-weight calls to the Z3 SMT solver, to our surprise SPEjs had generally a lower runtime. However this can be explained by Prepack's larger overall machinery and SPEjs's prototype status. So it likely does not generalize to a setting in which SPEjs is fully built out with a similar coverage of JavaScript language features that Prepack supports. However we did observe that SPEjs's runtime increased as expected with its number of SMT solver calls.

5.2.2 RQ2: Partially Evaluated Code Faster. Both tools produced code that was faster than the original. But the differences were small and due to the tiny code size may be misleading. Prepack emphasizes that it optimizes its residual code for runtime performance. This lead to overall similar results as SPEjs's residual code that is wholly un-optimized outside its more aggressive partial evaluation.

5.2.3 RQ3: Partially Evaluated Code Smaller. The main results are in RQ3, as for micro benchmarks code size is a good proxy for partial evaluation performance. Both tools generally reduced code size. SPEjs in some cases benefited from its more aggressive partial evaluation. A good example is Sample 6, for which Prepack produced the Listing 9 residual code, which removes only one of Listing 7's two infeasible branches.

```
(function () {
  var _$2 = this;
  var _$0 = _$2.Date.now();
  var _3 = _$0 > 2;
  var _6 = _$0 < 0;
  var _2 = _3 && _6;
  var _1 = _2 ? 9 : 2;
  var _0 = _1 - 6;
  a = _0;
}).call(this);
```

Listing 9: Prepack's residual code for Listing 7.

On the other hand, SPEjs produced Listing 10, which removed both infeasible branches, which in turn reduces the entire code to a single line. While Prepack fails to process Sample 7, SPEjs removes its infeasible branch.

```
1    a = -4;
```

Listing 10: SPEjs's residual code for Listing 7.

5.3 Prepack Test Suite Results

We repeated the experiments on the Prepack test suite. While we could only get a small subset to work with the public Prepack version, we manually injected mutations to make some of the branches infeasible. On these subjects we received results that generally matched what we observed for the micro benchmarks.

6 RELATED WORK

Researchers have implemented both static and dynamic symbolic execution systems for JavaScript and applied them to various tasks such as testing and finding and correcting security vulnerabilities [15, 23, 25, 26]. Besides Prepack, other partial evaluation schemes for JavaScript have been developed, for example, for finding vulnerabilities [31], speeding up browser-specific code [13], reducing code size [8], or pre-computing results on the server-side [20].

While previous work has combined partial evaluation with symbolic execution, it was done differently and for a different goal. In these earlier approaches, partial evaluation is used to speed up static symbolic execution, i.e., by tackling its path explosion problem [2, 9, 22, 32]. The SPEjs M.S. thesis contains additional details on SPEjs and the experiments [29].

7 CONCLUSIONS

This paper proposed to deepen the reach of current partial evaluation schemes by selectively adding local symbolic execution. Concretely, we described the SPEjs symbolic partial evaluator for JavaScript that is built on Babel and the SMT solver Z3. To gauge the promise of this approach we compared SPEjs with Facebook's state-of-the-art partial evaluator Prepack. Our results on a set of micro benchmarks and Prepack's test suite indicate that, within Prepack's runtime budget, SPEjs was able to simplify additional expressions and therefore remove dead code branches that Prepack failed to remove, yielding smaller residual programs.

REFERENCES

[1] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Allocation removal by partial evaluation in a tracing JIT. In *Proc. 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*. ACM, 43–52.

[2] Richard Bubel, Reiner Hähnle, and Ran Ji. 2010. Interleaving symbolic execution and partial evaluation. In *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO)*. Springer, 125–146.

[3] Mathias Bynens and John-David Dalton. 2010. Benchmark.js v2.1.2. https://benchmarkjs.com/. accessed July 2018.

[4] Charles Consel, Julia L. Lawall, and Anne-Françoise Le Meur. 2004. A tour of Tempo: a program specializer for the C language. *Science of Computer Programming* 52 (Aug. 2004), 341–370.

[5] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77.

[6] Jeffrey Dean, Craig Chambers, and David Grove. 1995. Selective Specialization for Object-Oriented Languages. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 93–102.

[7] Facebook. 2017. Prepack. https://prepack.io/. accessed July 2018.

[8] Google. 2015. Closure Compiler. https://developers.google.com/closure/compiler/. accessed July 2018.

[9] Ran Ji and Richard Bubel. 2012. PE-KeY: A Partial Evaluator for Java Programs. In *Proc. 9th International Conference on Integrated Formal Methods (IFM)*. Springer, 283–295.

[10] Neil D. Jones. 1996. An Introduction to Partial Evaluation. *Comput. Surveys* 28, 3 (Sept. 1996), 480–503.

[11] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

[12] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394.

[13] Karl Krukow. 2008. Jeene: An automatic partial evaluator for JavaScript. http://blog.higher-order.net/2008/09/14/jeene.html. accessed July 2018.

[14] Jamie Kyle. 2017. Babel Plugin Handbook. https://github.com/thejameskyle/babel-handbook/. accessed July 2018.

[15] Guodong Li, Esben Andreasen, and Indradeep Ghosh. 2014. SymJS: Automatic Symbolic Testing of JavaScript Web Applications. In *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 449–459.

[16] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renaud Marlet. 2001. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems* 19, 2 (May 2001), 217–251.

[17] Microsoft. 2013. Z3. https://github.com/Z3Prover/z3. accessed July 2018.

[18] Mmicu. 2016. Leena. https://github.com/mmicu/leena. accessed July 2018.

[19] Torben Mogensen and Peter Sestoft. 1997. Partial evaluation. *Encyclopedia of Computer Science and Technology* 37 (1997), 247–279.

[20] Ravi Netravali and James Mickens. 2018. Prophecy: Accelerating mobile page loads using final-state write logs. In *Proc. 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX.

[21] Noah Peters. 2016. jsmeter: JavaScript code metrics. http://jsmeter.info. accessed July 2018.

[22] José Miguel Rojas and Corina S Pasareanu. 2013. Compositional symbolic execution through program specialization. *BYTECODE'13 (ETAPS)* (2013).

[23] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for JavaScript. In *Proc. IEEE Symposium on Security and Privacy (Oakland)*. IEEE, 513–528.

[24] Ulrik Pagh Schultz, Julia L. Lawall, and Charles Consel. 2003. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 4 (July 2003), 452–499.

[25] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proc. 21th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 488–498.

[26] Koushik Sen, George C. Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path symbolic execution using value summaries. In *Proc. Foundations of Software Engineering (FSE)*. ACM, 842–853.

[27] Amin Shali and William R. Cook. 2011. Hybrid partial evaluation. In *Proc. 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 375–390.

[28] Venkatesh Srinivasan and Thomas W. Reps. 2015. Partial evaluation of machine code. In *Proc. ACM SIGPLAN International Conference on Object-Oriented Programming (OOPSLA)*. ACM, 860–879.

[29] Sümeyye Süslü. 2018. *JSSpe: A Symbolic Partial Evaluator for JavaScript*. Master's thesis. University of Texas at Arlington.

[30] Sümeyye Süslü. 2018. SPEjs: A Symbolic Partial Evaluator for JavaScript: Project home page. https://github.com/SumeyyeSuslu/SPEjs.

[31] Omer Tripp, Pietro Ferrara, and Marco Pistoia. 2014. Hybrid Security Analysis of Web JavaScript Code via Dynamic Partial Evaluation. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 49–59.

[32] Germán Vidal. 2012. Closed symbolic execution for verifying program termination. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*. IEEE, 34–43.

[33] W3Techs. 2018. Usage of JavaScript for websites. https://w3techs.com/technologies/overview/client_side_language/all. accessed July 2018.

[34] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 662–676.