ExpoSE: Practical Symbolic Execution of Standalone JavaScript

Blake Loring, Duncan Mitchell, and Johannes Kinder
Department of Computer Science
Royal Holloway, University of London

ABSTRACT

JavaScript has evolved into a versatile ecosystem for not just the web, but also a wide range of server-side and client-side applications. With this increased scope, the potential impact of bugs increases. We introduce ExpoSE, a dynamic symbolic execution engine for Node.js JavaScript applications that allows to automatically generate thorough test suites and find bugs automatically. We discuss the specific challenges for symbolic execution arising from the widespread use of regular expressions in such applications. In particular, we make explicit the issues of capture groups, backreferences, and greediness in JavaScript's flavor of regular expressions, and our models improve over previous work that only partially addressed these. We evaluate ExpoSE on three popular JavaScript libraries that make heavy use of regular expressions, and we report a previously unknown bug in the `Minimist` library.

1 INTRODUCTION

Since its inception as a scripting language for dynamic web elements, JavaScript has seen its popularity balloon and has become a versatile and widely used application platform. Browserless runtimes, and in particular Node.js, allow developers to build server-side[1] and client-side[2] applications in pure JavaScript. With its growing importance for the infrastructure of today's systems, there is an increased need for helping developers find bugs early.

A successful technique for automatically finding bugs in real world software is dynamic symbolic execution (DSE). Traditionally, DSE engines mostly targeted C, Java, or binary code [3, 7]. However, early implementations of DSE for JavaScript have shown promising results for testing browser-based JavaScript code [5, 8].

In DSE, some inputs to the program under test are made *symbolic* while the rest are fixed. Starting with an initial concrete assignment to the symbolic inputs, the DSE engine executes the program both concretely and symbolically and maintains a *symbolic state* that maps program variables to expressions over the symbolic inputs. Whenever the symbolic execution encounters a conditional operation, the symbolic state's evaluation of the condition or its negation are added to the *path condition*, depending on the concrete result of the operation. Once the execution finishes, the path condition uniquely characterizes the executed control flow path. By negating the last constraint of the path condition or of one of its prefixes, the DSE engine generates a constraint for a different path. It then calls a constraint solver to check feasibility of that path and to obtain a satisfying assignment for the symbolic input variables that drives the next execution down that path.

The joint symbolic and concrete execution is one of the advantages of DSE: when an external function cannot be analyzed or an operation lies outside the constraint solver's theory, the DSE engine can concretize parts of the symbolic state without sacrificing soundness, at the cost of reducing the search space. Nevertheless, we must avoid excessive concretization and provide symbolic semantics for as much of the target language as possible to allow effective test generation. This is one of the reasons why low-level or byte-code languages are popular DSE targets, while keyword-rich languages with large standard libraries are supported less frequently [1]. The ECMA standard for JavaScript specifies regular expressions as part of the language. Therefore, a DSE engine for JavaScript must support regular expressions to be effective.

Modern constraint solvers support strings and regular expressions via encodings as finite automata [6, 11, 12]. However, ECMA regular expressions are strictly more expressive than regular languages [2]. They include the notion of *capture groups* and *backreferences* in what is often referred to as "perl-style regular expressions"; we will refer to these languages as *regex* for short. For example, in the regex `/([a-z]+)\1/`, the parentheses denote a *capture group*, and the `\1` denotes a *backreference*, which specifies that whatever was matched inside the parentheses should be repeated at that point in the string. We need to model capture groups within a regex to avoid concretization. Consider the following program:

```
let capturedMatches = symbolic_string().match(/([a-z])/)
if (capturedMatches[1] == 'c') {
  do_something();
}
```

Without symbolic semantics for `match`, the contents of the capture group (stored at index 1 of the returned array) would be concretized, making it impossible to explore the path entering the conditional statement. In this paper, we present our design for a symbolic execution engine for JavaScript and address the practical issues that need to be overcome when trying to support real world language features, in particular regular expressions. We make the following contributions:

- We highlight the challenges in implementing a DSE engine for JavaScript, including the handling of asynchronous events and ECMA regular expressions.
- We present an encoding of JavaScript's regular expressions into a combination of classical regular expressions and SMT, including what is to our knowledge the first explicit support for both capture groups and backreferences.

2 EXPOSE

We now present the architecture and design of ExpoSE, our framework for dynamic symbolic execution of JavaScript applications.

2.1 Architecture

ExpoSE takes a JavaScript program and a symbolic unit test (the test harness) and generates test cases until it has explored all feasible paths or exceeds a time bound. ExpoSE consists of two main

[1] https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/
[2] https://blog.atom.io/2014/02/26/the-nucleus-of-atom.html

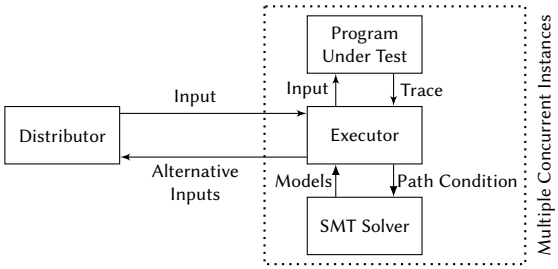

Figure 1: The ExpoSE architecture.

components, the *test executor* and the *test distributor*, as shown in the overview in Figure 1. The distributor manages the global state of the exploration, aggregates statistics, and schedules test cases for symbolic execution. Multiple test cases can be run concurrently in separate test executor processes to take advantage of parallelism.

The test case executor instruments the program under test to perform symbolic execution and detect any bugs during execution. We use the Jalangi2 framework for instrumentation, which is itself written in JavaScript. It inserts callbacks for all JavaScript syntax, including in code dynamically created by `eval` and `require`. Each instance of the executor runs a test case, symbolically executes the induced trace, and uses the Z3 SMT solver (via custom JavaScript bindings) to generate new test inputs that are passed back to the Distributor. JavaScript uses floating point representation for all its numbers, which we approximate using real arithmetic. For supporting string operations, we rely on the recently added theory of strings in Z3, together with our custom encoding of regex into regular expressions and string constraints (see §3).

2.2 Test Isolation

JavaScript programs execute in a single thread, but rely on asynchronous operations to achieve a form of parallelism and avoid blocking. Callbacks from completed asynchronous operations are scheduled whenever the execution of the current execution frame finishes. To avoid starvation of asynchronous events and timeouts in a continuously running test executor, we create a separate process for each test case. Additionally, this prevents spill-over effects from one execution to the next where the program affects the global state or dynamically modifies parts of the standard library (which is legal in JavaScript). As a side effect, the overhead of applying instrumentation is repeated for every test case; we are investigating adding caching mechanisms to Jalangi2 to avoid redundant steps.

3 REGULAR EXPRESSIONS

We now present our encoding from regex into regular expressions and string constraints. This extends prior work by including a formalization of nested capture groups and backreferences [6, 8, 9].

3.1 Capture Groups

Let R be any regex without backreferences, $\omega \in \mathcal{L}(R)$ a word of the language defined by R, and C a capture group within R. Then the capture of C is the last substring of ω "matched" by C, i.e., consumed by the transitions corresponding to C within R's equivalent non-deterministic finite automaton (NFA).

In JavaScript, `match` and related operations return capture groups in an array, with the entire matched string at position 0 and the capture of the n-th capture group in position n. Captures may be the empty string if that capture group was not matched, which can occur with alternation or quantification (* or ?). If there is no match, the entire array is empty. We model the semantics of `match` using formulas over the theory of strings and regular expressions. Each capture string is expressed as the concatenation of the capture groups and expressions within it.

Consider evaluating `str.match(/(a)b/` in a symbolic state where `str` evaluates to Ω. The result is either **`null`** (no match) or an array containing the entire match (ω) and the last match for the capture group (ω_1). During DSE, we treat the two cases as two separate paths. If the concrete evaluation of `match` succeeds, we set the result to $[\omega, \omega_1]$ and add to the path constraint the expression $\omega \ll \Omega \wedge \omega = \omega_1 \mathbin{+\!\!+} \omega_2 \wedge \omega_1 \in \mathcal{L}(\mathtt{a}) \wedge \omega_2 \in \mathcal{L}(\mathtt{b})$, where $\ll$ is the substring relation and $+\!\!+$ is string concatenation. If the concrete match fails, we add the negation of the constraint and set the result to **`null`**.

General Encoding. We now show how we construct such constraints in general, using a recursive encoding strategy. To allow referring to individual captures in constraints, we break down a regex R into its constituent capture groups and literals (which include character classes and ranges, and for the purpose of simplifying the presentation, also capture-free concatenations of literals such as `abc`). We can then describe words $\omega \in \mathcal{L}(R)$ in terms of concatenation constraints over literals and words in the languages of the capture groups, which are broken down further in a recursive manner. We write any regex R as

$$R = r_0 \circ_0 r_1 \circ_1 \ldots \circ_{n-1} r_n, \tag{1}$$

where each odd-indexed r_i is a (possibly empty) capture group (which contains, recursively, the same structure as R), each even-indexed r_i a (possibly empty) literal, and each $\circ_i$ either concatenation, alternation, or a Kleene star (+ and ? are rewritten with their direct equivalences). Note that allowing empty r_i also permits to have no capture groups at all or multiple consecutive operators (e.g., Kleene star and concatenation).

The alternation operator $r_1 \mid r_2$ matches either r_1 or r_2. Following the decomposition of R in Equation 1 above, we can, for the first i such that $\circ_i = |$, write $\omega \in \mathcal{L}(R) \iff \omega \in \mathcal{L}(r_0 \circ_0 \ldots \circ_{i-1} r_i) \vee \omega \in \mathcal{L}(r_{i+1} \circ_{i+1} \ldots \circ_{n-1} r_n)$. Applied repeatedly on each resulting subexpression, we can eliminate alternation over capture groups.

The concatenation operator is implicit: words in $\mathcal{L}(r_1 r_2)$ are the concatenation of words in $\mathcal{L}(r_1)$ and $\mathcal{L}(r_2)$. Suppose we have eliminated alternation in the regex via the constraints above, then considering Equation 1, for the first i such that $\circ_i$ is concatenation, we obtain $\omega \in \mathcal{L}(R) \iff \omega_1 \in \mathcal{L}(r_0 \circ_0 \ldots \circ_{i-1} r_i) \wedge \omega_2 \in \mathcal{L}(r_{i+1} \circ_{i+1} \ldots \circ_{n-1} r_n) \wedge \omega = \omega_1 \mathbin{+\!\!+} \omega_2$. As before, this allows to iteratively eliminate concatenation between capture groups.

Once we have eliminated alternation and concatenation at the top level, we need only describe the constraints induced by Kleene Star operations on a single subexpression r_i. Capture groups and literals can be encoded in the same manner, via

$$\omega \in \mathcal{L}(r_i*) \iff \Big(\exists m \geq 0 \wedge \omega = \mathbin{+\!\!+}_{j=0}^{m} \omega_j \wedge \forall j \in \{0, \ldots, m\}, \omega_j \in \mathcal{L}(r_i)\Big) \vee \omega = \epsilon \tag{2}$$

When r_i is a capture group, we note that the result of the capture group refers to the last matching string satisfying it, ω_m. We can then proceed recursively on each capture group as above to fully describe the generated language.

3.2 Backreferences

A backreference specifies that any match for it must match the last instance of a labeled, closed capture group. They can be used both outside and inside quantification (*, +, ?), matching the last string matched to the specified capture group.

Suppose that, in a regex R, there are m non-null capture groups (including top-level and nested captures), and that any literal r_i may additionally be a backreference $\backslash k$, where $1 \leq k \leq m$ and the k^{th} capture group, C_k, is closed at the point of the backreference. We can then define a regex $R' = r'_0 \circ_0 r'_1 \circ_1 \ldots \circ_{n-1} r'_n$, where $r'_i = C_k$ if $r_i = \backslash k$ and $r'_i = r_i$ otherwise. It is immediate that $\mathcal{L}(R) \subseteq \mathcal{L}(R')$. Since R' is backreference-free it can be encoded by the process in §3.1; we now extend this encoding to R.

Consider the case where a regex R has no quantification. At the top level, the constraint on R' generated from eliminating alternations is of the form $\omega \in \mathcal{L}(R') \iff P_1 \vee \ldots \vee P_N$. For some M and languages $\mathcal{L}_{i,j}$, each $P_i = (\omega = \omega_{i,1} +\!+ \ldots +\!+ \omega_{i,M}) \wedge (\forall j, \omega_{i,j} \in \mathcal{L}_{i,j})$, due to the elimination of concatenation. Suppose R contains a backreference to C_k at the index i. Suppose that ω_{i_1,j_1} is the string matching the capture group C_k, and ω_{i_2,j_2} is the string matching r'_i in R'. Then translate our constraint for $\mathcal{L}(R')$ into one for $\mathcal{L}(R)$ by adding the constraint $\omega_{i_1,j_1} = \omega_{i_2,j_2}$. Note this only makes sense if $i_1 = i_2$, but this follows the semantics of backreferences.

Now consider a backreference to some C_k which is not contained within a quantified capture group, either $R_1 = \ldots C_k \ldots \backslash k* \ldots$ or $R_2 = \ldots C_k* \ldots \backslash k \ldots$. For the former case, as before, suppose ω_{i_1,j_1} is the string matching C_k, and ω_{i_2,j_2} is the string matching C_k* in the constraints describing $\mathcal{L}(R'_1)$. From Equation 2, we know that either $\omega_{i_2,j_2} = \omega_0 +\!+ \ldots +\!+ \omega_m$ for some m, or $\omega_{i_2,j_2} = \epsilon$. We add an extra constraint only in the former case, which ensures each ω_i must be the same as the match of the capture group referenced: $\forall i : 0 \leq i \leq m, \omega_i = \omega_{i_1,j_1}$. In the latter case, suppose ω_{i_1,j_1} matches C_k*, then $\omega_{i_1,j_1} = \omega_0 +\!+ \ldots +\!+ \omega_m$ for some m or $\omega_{i_1,j_1} = \epsilon$. Suppose further that ω_{i_2,j_2} is the string matching the replaced backreference in R'_2. Recalling that in $\mathcal{L}(R_2)$, ω_{i_2,j_2} must match the last match of C_k, we need to add to our constraints for $\mathcal{L}(R'_2)$ that $\omega_{i_2,j_2} = \epsilon$ if $\omega_{i_1,j_1} = \epsilon$ and $\omega_{i_2,j_2} = \omega_m$ otherwise.

Finally, we consider nested backreferences with capture groups. Without loss of generality, consider $R_3 = \ldots (\ldots C_k \ldots \backslash k \ldots)* \ldots$ where C_k and k are contained within the top-level capture group C_l. Considering R'_3, and supposing that C'_l (the related regex to C_l) does not contain any alternation operators, we obtain (omitting any non-pertinent constraints):

$$\omega \in \mathcal{L}(C'_l) \iff \omega = \ldots +\!+ \omega_i +\!+ \ldots +\!+ \ldots \omega_j +\!+ \ldots \\ \wedge \ldots \wedge \omega_i \in \mathcal{L}(C_k) \wedge \ldots \wedge \omega_j \in \mathcal{L}(C_k) \wedge \ldots$$

For this i and j, we have $\omega \in \mathcal{L}(C_l) \iff \omega \in \mathcal{L}(C'_l) \wedge \omega_i = \omega_j$. Considering this additional constraint and Equation 2, we note this describes accurately the nested backreference. Recursing through the entire regex R, we can describe the entire language R in terms of these constraints.

	Minimist	Semver	Validator
Lines of Code	300	1200	1500
Path Count	52	248	168
Total Execution	902.00s	500.00s	170.00s
Mean Test Case	18.15s	7.44s	8.09s
Median Test Case	1.31s	2.52s	3.00s
Shortest Test Case	0.73s	1.09s	1.67s
Longest Test Case	900.00s	495.10s	64.47s

Figure 2: Statistics and runtimes for testing targets.

3.3 Remaining Challenges

Nested Backreferences. In practice, the encoding for nested backreferences presented above leads to sets of constraints that are hard to solve for state of the art SMT solvers. In our current implementation in ExpoSE, we use a more restrictive encoding that limits any quantified, nested backreference to have only equal matches. For example, given the regular expression `((a|b)\2)+`, ExpoSE would consider the strings `aaaa` and `bbbb` to be part of the language but not `aabb`. With string support in SMT solvers still a relatively new addition, we hope to loosen this restriction in the future.

Greediness. By default, regexes are *greedy*, which means that as the regex is matched from left to right, only the largest possible match of each subexpression is considered. Greediness makes a difference for the contents of capture groups: in `/a*(a*)/`, the capture will always be empty since all `a`'s will be consumed by the greedy initial `a*`. Our encoding disregards greediness when a regex uses unbounded quantification. In fact, we are not aware of prior work that even considered this interplay of greediness and capture groups as a limitation.

Including full support for greediness would require a combination of existential quantifiers and maximality constraints to construct encodings which force the result of matching any subexpression to be the largest string such that the remaining constraints are still satisfiable. Unfortunately, a direct encoding would likely cause many constraints to become infeasible for automatic solving.

4 EVALUATION

We evaluate ExpoSE by testing the JavaScript libraries `Minimist`, `Semver` and `Validator`[3]. These popular libraries—ranging between 1.5m and 30m downloads a month—all rely on both string operations and regular expression matching and each contain between 300 and 1500 lines of code. ExpoSE is able to cover between 56-80% and found a previously undiscovered crash bug in `Minimist`.

4.1 Methodology

Code Coverage. It is not straightforward to determine the total amount of code that could potentially be covered in a JavaScript program. The statements `eval` and `require` can dynamically add new code to be executed and thus may lead to different baselines for different test cases. In addition to all code in the main file under test, we count source code that is required (imported) on demand or evaluated (generated) during execution. We measure code coverage as the fraction of unique nodes in the program's abstract syntax tree encountered during execution. For JavaScript code, this metric

[3] https://www.npmjs.com/

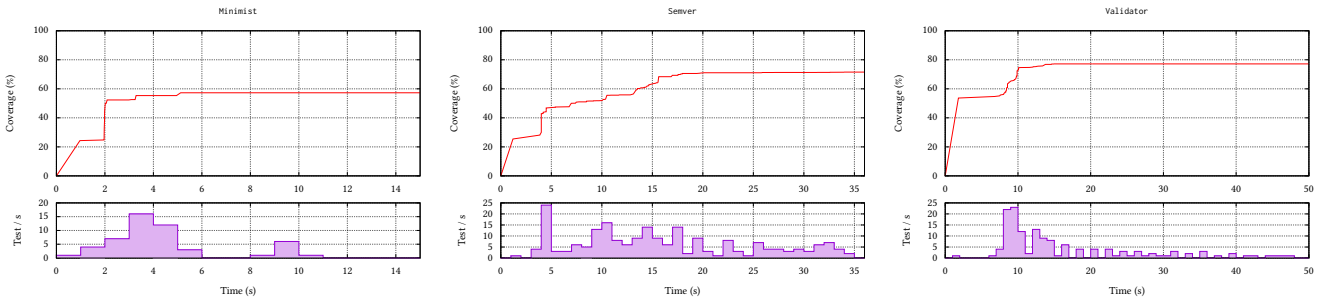

Figure 3: ExpoSE performance statistics for three popular Node.js libraries.

allows to distinguish multiple blocks in a single line of code, which are common in functional design patterns and could otherwise suggest an inflated coverage.

Test Harness. We built a generic test harness to systematically exercise all public methods in a given library with symbolic arguments. The symbolic arguments range over all our supported theories (Strings, Booleans, Reals, Undefined, Null). We ignore any spurious type exceptions due to functions expecting values of a specific type. The public methods of all three libraries expected only values of base types; for object types, we will consider a generational or lazy approach as part of future work [4].

Test Environment. Each test case was executed on a machine with a Intel Xeon E5-2640 CPU and 128GB of RAM. Each library was tested with up to 128 concurrent test cases.

4.2 Results

Figure 2 lists the runtime statistics for testing each library and number of explored paths. The graphs in Figure 3 show code coverage and test execution rate over time. Most test cases execute within 2s, including instantiation costs. However, in each library some test cases generated queries requiring several minutes to solve, depending on the random seed given to the solver.

All targets show an initial peak in execution rate, which is due to concurrently executing a large number of test cases, of which many finish quickly. The generational search in ExpoSE generates alternatives for all conditions already from the first run [3]. This rate slows down over time as the remaining cases finish, with new spawns having a smaller but recurring effect on the rate.

Typical for test generation, coverage approaches a plateau in all cases. `Minimist` plateaus below 60%, due to concretization in functions we do not model explicitly, including `split`. `Semver` and `Validator` both display a more stable test execution rate and coverage gains, due to both libraries having multiple disconnected public methods and shorter query times.

ExpoSE identified a new bug within `Minimist`, which occurs when it is passed any argument in the form `'--=...='`, due to an overly permissive regular expression. Symbolic modeling of `test` and `match` allowed for the generation for the failing test case.

5 RELATED WORK

Previous DSE engines for JavaScript [5, 8, 10] have demonstrated success in finding bugs in JavaScript on the web. Our initial results show that we can generate comparable numbers of paths for our target applications. Jalangi [10] is the predecessor to Jalangi2, and included support for DSE (Jalangi2 does not). However, it is no longer maintained and has only limited support for asynchronous events and regular expressions.

Existing string solvers do not consider perl-style regular expressions [6, 12]. The string solver Kaluza [8] does not support backreferencing but includes a form of capture groups, however it is unclear whether their encodings are faithful to the ECMA specification, as noted by Liang et al. [6]. The elimination of backreferences via concatenation constraints is described in the work of Scott et al [9], although they do not describe systematic means of reducing the backreferences into SMT.

6 CONCLUSION

In this paper, we discussed the challenges faced and design choices made when building ExpoSE, a DSE engine for standalone JavaScript applications. We presented an encoding of JavaScript regular expressions into SMT, including to our knowledge the first explicit treatment of both capture groups and backreferences. We demonstrated that ExpoSE is effective at generating tests for unmodified, string-heavy JavaScript code and we were able to report a new bug in one of three libraries tested. In future work, we plan to extend ExpoSE's support for backreferences to our full encoding and to faithfully model greediness for capture groups.

REFERENCES

[1] S. Bucur, J. Kinder, and G. Candea. Prototyping symbolic execution engines for interpreted languages. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[2] C. Câmpeanu, K. Salomaa, and S. Yu. A formal study of practical regular expressions. *Int. J. Foundations of Computer Science*, 14(06):1007–1018, 2003.

[3] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp. (NDSS)*, 2008.

[4] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 553–568, 2003.

[5] G. Li, E. Andreasen, and I. Ghosh. Symjs: automatic symbolic testing of JavaScript web applications. In *Int. Symp. Foundations of Software Engineering (FSE)*, 2014.

[6] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Int. Conf. Computer Aided Verification (CAV)*, 2014.

[7] C. S. Pasareanu and N. Rungta. Symbolic pathfinder: symbolic execution of Java bytecode. In *Int. Conf. on Automated Software Eng. (ASE)*, pages 179–180, 2010.

[8] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *IEEE Symp. Sec. and Privacy (S&P)*, 2010.

[9] J. D. Scott, P. Flener, and J. Pearson. Constraint solving on bounded string variables. In *Integration of AI and OR Tech. in Constraint Prog. (CPAIOR)*, 2015.

[10] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Int. Symp. Foundations of Software Engineering (FSE)*, 2013.

[11] M. Trinh, D. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Conf. Computer and Commun. Sec. (CCS)*, 2014.

[12] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *Int. Symp. Foundations of Software Eng. (FSE)*, 2013.