Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript

Blake Loring
Information Security Group
Royal Holloway, University of London
United Kingdom
blake.loring.2015@rhul.ac.uk

Duncan Mitchell
Department of Computer Science
Royal Holloway, University of London
United Kingdom
duncan.mitchell.2015@rhul.ac.uk

Johannes Kinder
Research Institute CODE
Bundeswehr University Munich
Germany
johannes.kinder@uni-bw.de

Abstract
Support for regular expressions in symbolic execution-based tools for test generation and bug finding is insufficient. Common aspects of mainstream regular expression engines, such as backreferences or greedy matching, are ignored or imprecisely approximated, leading to poor test coverage or missed bugs. In this paper, we present a model for the complete regular expression language of ECMAScript 2015 (ES6), which is sound for dynamic symbolic execution of the test and exec functions. We model regular expression operations using string constraints and classical regular expressions and use a refinement scheme to address the problem of matching precedence and greediness. We implemented our model in ExpoSE, a dynamic symbolic execution engine for JavaScript, and evaluated it on over 1,000 Node.js packages containing regular expressions, demonstrating that the strategy is effective and can significantly increase the number of successful regular expression queries and therefore boost coverage.

CCS Concepts · Software and its engineering → Software verification and validation: Dynamic analysis; Theory of computation → Regular languages.

Keywords Dynamic symbolic execution, JavaScript, regular expressions, SMT

ACM Reference Format:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI ’19, June 22–26, 2019, Phoenix, AZ, USA © 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6712-7/19/06... $15.00
https://doi.org/10.1145/3314221.3314645

1 Introduction
Regular expressions are popular with developers for matching and substituting strings and are supported by many programming languages. For instance, in JavaScript, one can write /goo+d/.test(s) to test whether the string value of s contains "go", followed by one or more occurrences of "o" and a final "d". Similarly, s.replace(/goo+d/,"better") evaluates to a new string where the first such occurrence in s is replaced with the string "better".

Several testing and verification tools include some degree of support for regular expressions because they are so common [24, 27, 29, 34, 37]. In particular, SMT (satisfiability modulo theory) solvers now often support theories for strings and classical regular expressions [1, 2, 6, 15, 25, 26, 34, 38–40], which allow expressing constraints such as s ∈ L(goo+d) for the test example above. Although any general theory of strings is undecidable [7], many string constraints are efficiently solved by modern SMT solvers.

SMT solvers support regular expressions in the language-theoretical sense, but "regular expressions" in programming languages like Perl or JavaScript—often called regex—a term we also adopt in the remainder of this paper—are not limited to representing regular languages [3]. For instance, the expression /<\w+>/.test(s) parses any pair of matching XML tags, which is a context-sensitive language (because the tag is an arbitrary string that must appear twice). Problematic features that prevent a translation of regexes to the word problem in regular languages include capture groups (the parentheses around \w+ in the example above), backreferences (the \1 referring to the capture group), and greedy/non-greedy matching precedence of subexpressions (the . as non-greedy). In addition, any such expression could also be included in a lookahead (?=), which effectively encodes intersection of context sensitive languages. In tools reasoning about string-manipulating programs, these features are usually ignored or imprecisely approximated. This is a problem, because they are widely used, as we demonstrate in §7.1.

In the context of dynamic symbolic execution (DSE) for test generation, this lack of support can lead to loss of coverage or missed bugs where constraints would have to include membership in non-regular languages. The difficulty arises from the typical mixing of constraints in path conditions—simply generating a matching word for a standalone regex is
We review the ES6 regex specification, focusing on differences to classical regular expressions. We begin with the regex API and its matching behavior (§2.1) and then explain capture groups (§2.2), backreferences (§2.3), and operator precedence (§2.4). ES6 regexes are comparable to those of other languages but lack Perl’s recursion and lookbehind and do not require POSIX-like longest matches.

2 ECMAScript Regex

We review the ES6 regex specification, focusing on differences to classical regular expressions. We begin with the regex API and its matching behavior (§2.1) and then explain capture groups (§2.2), backreferences (§2.3), and operator precedence (§2.4). ES6 regexes are comparable to those of other languages but lack Perl’s recursion and lookbehind and do not require POSIX-like longest matches.

2.1 Methods, Anchors, Flags

ES6 regexes are RegExp objects, created from literals or the RegExp constructor. RegExp objects have two methods, test and exec, which expect a string argument; String objects offer the match, split, search and replace methods that expect a RegExp argument.

A regex accepts a string if any portion of the string matches the expression, i.e., it is implicitly surrounded by wildcards. The relative position in the string can be controlled with anchors, with ^ and $ matching the start and end, respectively. Flags in regexes can modify the behavior of matching operations. The ignore case flag i ignores character cases when matching. The multiline flag m redefines anchor characters to match either the start and end of input or newline characters. The unicode flag u changes how unicode literals are escaped within an expression. The sticky flag y forces matching to start at RegExp.lastIndex, which is updated with the index of the previous match. Therefore, RegExp objects become stateful as seen in the following example:

```
let r = /goo+d/y;
if (r.test("good")); // true; r.lastIndex = 6
if (r.test("good")); // false; r.lastIndex = 0
```

The meaning of the global flag g varies. It extends the effects of match and replace to include all matches on the string and it is equivalent to the sticky flag for the test and exec methods of RegExp.

2.2 Capture Groups

Parentheses in regexes not only change operator precedence (e.g., (ab)* matches any number of repetitions of the string "ab" while ab* matches the character "a" followed by any number of repetitions of the character "b") but also create capture groups. Capture groups are implicitly numbered from left to right by order of the opening parenthesis. For example, /a|((b)*c)*d/ is numbered as /a|((b)*c)*d/. Where only bracketing is required, a non-capturing group can be created by using the syntax (?: ...).

For regexes, capture groups are important because the regex engine will record the most recent substring matched against each capture group. Capture groups can be referred to from within the expression using backreferences (see §2.3). The last matched substring for each capture group is also returned by some of the API methods. In JavaScript, the return values of match and exec are arrays, with the whole match at index 0 (the implicit capture group 0), and the last matched instance of the jth capture group at index i. In the example above, "bbbcbcd".match(/a|((b)*c)*d/) will evaluate to the array ["bbbcbcd", "bc", "b"].

2.3 Backreferences

A backreference in a regex refers to a numbered capture group and will match whatever the engine last matched the capture group against. In general, the addition of backreferences to regexes makes the accepted languages non-regular [3].

Inside quantifiers (Kleene star, Kleene plus, and other repetition operators), the string matched by the backreference can change across multiple matches. For example, the regex /\((a|b)\\1\)+/ can match the string "aabb", with the backreference \1 being matched twice: the first time, the capture...
Table 1. Regular expression operators, separated by classes of precedence.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Rewriting</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r)</td>
<td>Capturing parentheses</td>
<td></td>
</tr>
<tr>
<td>\n</td>
<td>Backreference</td>
<td></td>
</tr>
<tr>
<td>(?r)</td>
<td>Non-capturing parentheses</td>
<td></td>
</tr>
<tr>
<td>(?=r)</td>
<td>Positive lookahead</td>
<td></td>
</tr>
<tr>
<td>(!r)</td>
<td>Negative lookahead</td>
<td></td>
</tr>
<tr>
<td>\b</td>
<td>Word boundary</td>
<td></td>
</tr>
<tr>
<td>\B</td>
<td>Non-word boundary</td>
<td></td>
</tr>
<tr>
<td>r*</td>
<td>Kleene star</td>
<td></td>
</tr>
<tr>
<td>r*?</td>
<td>Lazy Kleene star</td>
<td></td>
</tr>
<tr>
<td>r+</td>
<td>Kleene plus</td>
<td>r*r</td>
</tr>
<tr>
<td>r+?</td>
<td>Lazy Kleene plus</td>
<td>r+?r</td>
</tr>
<tr>
<td>r(m,n)</td>
<td>Repetition</td>
<td>r^n</td>
</tr>
<tr>
<td>r(m,n)?</td>
<td>Lazy repetition</td>
<td>r^m</td>
</tr>
<tr>
<td>r?</td>
<td>Optional</td>
<td>r</td>
</tr>
<tr>
<td>r??</td>
<td>Lazy optional</td>
<td>ε</td>
</tr>
<tr>
<td>r1r2</td>
<td>Concatenation</td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td>r2</td>
<td>Alternation</td>
</tr>
</tbody>
</table>

group contains "a", the second time it contains "b". This logic applies recursively, and it is possible for backreferences to in turn be part of outer capture groups.

2.4 Operator Evaluation

We explain the operators of interest for this paper in Table 1; the implementation described in §6 supports the full ES6 syntax [14]. Some operators can be rewritten into semantically equivalent expressions to reduce the number of cases to handle (shown in the Rewriting column).

Regexes distinguish between greedy and lazy evaluation. Greedy operators consume as many characters as possible such that the entire regular expression still matches; lazy operators consume as few characters as possible. This distinction—called matching precedence—is unnecessary for classical regular languages, but does affect the assignment of capture groups and therefore backreferences.

Zero-length assertions or lookarounds do not consume any characters but still restrict the accepted word, enforcing a language intersection. Positive or negative lookaheads can contain any regex, including capture groups and backreferences. In ES6, lookbehind is only available through \b (word boundary), and \B (non-word boundary), which are commonly used to only (or never) match whole words in a string.

3 Overview

In an overview of our approach, we now define the word problem for regex (§3.1) and how it arises in DSE (§3.2). We introduce our model for regex by example (§3.3) and explain how to eliminate spurious solutions by refinement (§3.4).

3.1 The Word Problem and Capturing Languages

For any given classical regular expression \( r \), we write \( w \in \mathcal{L}(r) \) whenever \( w \) is a word within the (regular) language generated by \( r \). For a regex \( R \), we also need to record values of capture groups within the regex. To this end, we introduce the following notion:

Definition 1 (Capturing Language). The capturing language of a regex \( R \), denoted \( \mathcal{L}_c(R) \), is the set of tuples \((w, c_0, \ldots, c_n)\) such that \( w \) is a word of the language of \( R \) and each \( c_0, \ldots, c_n \) is the substring of \( w \) matched by the corresponding numbered capture group in \( R \).

A word \( w \) is therefore matched by a regex \( R \) if and only if \( \exists c_0, \ldots, c_n : (w, c_0, \ldots, c_n) \in \mathcal{L}_c(R) \). It is not matched if and only if \( \forall c_0, \ldots, c_n : (w, c_0, \ldots, c_n) \notin \mathcal{L}_c(R) \). For readability, we will usually omit quantifiers for capture variables where they are clear from the context.

3.2 Regex In Dynamic Symbolic Execution

The code in Listing 1 parses numeric arguments between XML tags from its input variable args, an array of strings. The regex in line 4 breaks each argument into two capture groups, the tag and the numeric value (parts[0] is the entire match). When the tag is "timeout", it sets the timeout value accordingly (lines 6–7). On line 12, a runtime assertion checks that the timeout value is truly numeric after the arguments have been processed. The assertion can fail because the program contains a bug: the regex in line 4 uses a Kleene star and therefore also admits the empty string as the number to set, and JavaScript’s dynamic type system will allow setting timeout to the empty string.

DSE finds such bugs by systematically enumerating paths, including the failure branches of assertions [17]. Starting from a concrete run with input, say, args[0] = "foo", the DSE engine will attempt to build a path condition that encodes the branching decisions in terms of the input values. It then attempts to systematically flip clauses in the path condition and query an SMT solver to obtain input assignments.
covering different paths. This process repeats forever or until all paths are covered (this program has an unbounded number of paths as it is looping over an input string).

Without support for regex, the DSE engine will concretize arg on the call to exec, assigning the concrete result to parts. With all subsequent decisions therefore concrete, the path condition becomes $pc = true$ and the engine will be unable to cover more paths and find the bug.

Implementing regex support ensures that parts is symbolic, i.e., its elements are represented as formulas during symbolic execution. The path condition for the initial path thus becomes $pc = (args[0], C_0, C_1, C_2) \not\in \mathcal{L}_c(R)$ where $R = <(\backslash w)^*([0-9]*\backslash <\\backslash 1>)$. Negating the only clause and solving yields, e.g., $args[0] = "<\a\n</\a>"$. DSE then uses this input assignment to cover a second path with $pc = (args[0], C_0, C_1, C_2) \in \mathcal{L}_c(R) \land C_1 = "timeout"$. Negating the last clause yields, e.g., "<timeout>0</timeout>", entering line 7 and making timeout and therefore the assertion symbolic. This leads to $pc = (args[0], C_0, C_1, C_2) \in \mathcal{L}_c(R) \land C_1 = "timeout" \land (C_2, C_3) \in \mathcal{L}_c([0-9]*\backslash $), which, after negating the last clause, triggers the bug with the input "<timeout></timeout>".

### 3.3 Modeling Capturing Language Membership

Capturing language membership constraints in the path condition cannot be directly expressed in SMT. We model these in terms of classical regular language membership and string constraints. For a given ES6 regex $R$, we first rewrite $R$ (see Table 1) in atomic terms only, i.e., $|, *, capture groups, back-references, lookaheads, and anchors. For consistency with the JavaScript API, we also introduce the outer capture group $C_0$. Consider the regex $R = (?: a | (b)) \backslash 1$. After preprocessing, the capturing language membership problem becomes

$$(w, C_0, C_1) \in \mathcal{L}_c((?: \cdot | \cdot) | (?: a | (b)) \backslash 1) | (?: \cdot | \cdot) | ?)$$

a generic rewriting that allows for characters to precede and follow the match in the absence of anchors.

We recursively reduce capturing language membership to regular membership. To begin, we translate the purely regular Kleene stars and the outer capture group to obtain

$$(w, C_0, C_1) \in \mathcal{L}_c(R) \implies w = w_1 + w_2 + w_3$$

$$(w_1 \in \mathcal{L}((?: \cdot | \cdot) | ?)$$

$$(w_2, C_1) \in \mathcal{L}_c((?: a | (b)) \backslash 1) \land C_0 = w_2$$

$$(w_3 \in \mathcal{L}((?: \cdot | \cdot) | ?))$$

where $+$ is string concatenation. We continue by decomposing the regex until there are only purely regular terms or standard string constraints. Next, we translate the nested capturing language constraint

$$(w_2, C_1) \in \mathcal{L}_c((?: a | (b)) \backslash 1) \implies w_2 = w_1 + w_2 \land (w_1, C_1) \in \mathcal{L}_c(a | (b)) \land (w_2) \in \mathcal{L}_c(\backslash 1).$$

When treating the alternation, either the left is satisfied and the capture group becomes undefined (which we denote as $\emptyset$), or the right is satisfied and the capture is locked to the match, which we model as

$$(w_1, C_1) \in \mathcal{L}(a) \land C_1 = \emptyset \lor (w_1, C_1) \in \mathcal{L}(b) \land C_1 = w_2.$$  

Finally we model the backreference, which is case dependent on whether the capture group it refers to is defined or not:

$$(C_1 = \emptyset \implies w_2 = \epsilon) \land (C_1 \neq \emptyset \implies w_2 = C_1).$$

Putting this together, we obtain a model for $R$:

$$(w, C_0, C_1) \in \mathcal{L}_c(R) \implies w = w_1 + w_2 + w_2 + w_3$$

$$\land C_0 = w_1 + w_2$$

$$\land ((w_1, C_1) \in \mathcal{L}(a) \land C_1 = \emptyset) \lor (w_1, C_1) \in \mathcal{L}(b) \land C_1 = w_2)$$

$$\land (C_1 = \emptyset \implies w_2 = \epsilon) \land (C_1 \neq \emptyset \implies w_2 = C_1)$$

$$\land w_1 \in \mathcal{L}((?: \cdot | \cdot) | ?) \land w_3 \in \mathcal{L}((?: \cdot | \cdot) | ?).$$

### 3.4 Refinement

Because of matching precedence (greediness), these models permit assignments to capture groups that are impossible in real executions. For example, we model $/a*\(a)\$/ as

$$(w, C_0, C_1) \in \mathcal{L}_c(/a*\(a)\$/) \implies w = w_1 + w_2$$

$$\land w_1 \in \mathcal{L}(a*) \land w_2 \in \mathcal{L}(a) \land C_0 = w \land C_1 = w_2.$$  

This allows $C_1$ to be either a or the empty string $\epsilon$, i.e., the tuple ("aa","aa","a") would be a spurious member of the capturing language under our model. Because $a*$ is greedy, it will always consume both characters in the string "aa"; therefore, (a)? can only match $\epsilon$. This problem posed by greedy and lazy operator semantics remains unaddressed by previous work [27, 29, 30, 34]. To address this, we use a counterexample-guided abstraction refinement scheme that validates candidate assignments with an ES6-compliant matcher. Continuing the example, the candidate element ("aa","aa","a") is validated by running a concrete matcher on the string "aa", which contradicts the candidate captures with $C_0 = "aa"$ and $C_1 = \epsilon$. The model is refined with the counter-example to the following:

$$w = w_1 + w_2$$

$$\land w_1 \in \mathcal{L}(a*) \land w_2 \in \mathcal{L}(a) \land C_0 = w \land C_1 = w_2$$

$$\land (w \not= "aa" \implies (C_0 = "aa" \land C_1 = \epsilon)).$$

We then generate and validate a new candidate $(w, C_0, C_1)$ and repeat the refinement until a satisfying assignment passes the concrete matcher.

### 4 Modeling ES6 Regex

We now detail the process of modeling capturing languages. After preprocessing a given ES6 regex $R$ to $R'$ (§4.1), we model constraints $(w, C_0, \ldots, C_n) \in \mathcal{L}_c(R')$ by recursively translating terms in the abstract syntax tree (AST) of $R'$
to classical regular language membership and string constraints (§4.2–4.3). Finally, we show how to model negated constraints \((w, C_0, \ldots, C_n) \notin L_c(R')\) (§4.4).

### 4.1 Preprocessing

For illustrative purposes, we make the concatenation \(R_1 R_2\) of terms \(R_1, R_2\) explicit as the binary operator \(R_1 \cdot R_2\). Any regex can then be split into combinations of atomic elements, capture groups and backreferences (referred to collectively as \(C\) terms, in line with the ES6 specification [14]), joined by explicit operators. Using the rules in Table 1, we rewrite any \(R\) to an equivalent regex \(R'\) containing only alternation, concatenation, Kleene star, capture groups, non-capturing parentheses, lookarounds, and backreferences. We rewrite any remaining lazy quantifiers to their greedy equivalents, any remaining explicit operators. Using the rules in −n (for repetition, if \(1 \leq n\), \(1 \leq k\), etc.)

\[
\forall i \in \{0, \ldots, n - m\} \quad C_i = C_{i,m} + \ldots + C_{i,m+k-1}
\]

\[
\exists x \in \{0, \ldots, n - m\} \quad (w, C_0, C_1, \ldots, C_n) \in L_c(S\{m, n\})
\]

For repetition, if \(S\{m, n\}\) has \(K\) capture groups, then \(S' = S^\{m, n\}\) has \(K\) capture groups.

### 4.2 Operators and Capture Groups

Let \(t\) be the next term to process in the AST of \(R'\). If \(t\) is capture-free and purely regular, there is nothing to do in this step. If \(t\) is non-regular, it contains \(k + 1\) capture groups (with \(k \geq 1\)) numbered \(t\) through \(t + k\). At each recursive step, we express membership of the capturing language \((w, C_0, \ldots, C_{i+k}) \in L_c(t)\) through a model consisting of string and regular language membership constraints, and a set of remaining capturing language membership constraints for subterms of \(t\). Note that we record the locations of capture groups within the regex in the preprocessing step. When splitting \(t\) into subterms \(t_1\) and \(t_2\), capture groups \(C_0, \ldots, C_{i+k}\) are contained in \(t_1\) and \(C_{i+j} = C_{i+j+1}, \ldots, C_{i+k}\) are contained in \(t_2\) for some \(j\). The models for individual operations are given in Table 2; we discuss specifics of the rules below.

When matching an alternation \(\mid\), capture groups on the non-matching side will be undefined, denoted by \(\varnothing\), which is distinct from the empty string \(\epsilon\).

When modeling quantification \(t = t_1 +\), we assume \(t_1\) does not contain backreferences (we address this case in §4.3). In this instance, we model \(t\) via the expression \(t_1 + t_1 \mid \epsilon\), where \(t_1\) is a regular expression corresponding to \(t_1\), except each set of capturing parentheses is rewritten as a set of non-capturing parentheses. In this way, \(t_1\) is regular (it is backreference-free by assumption). However, \(t_1 + t_1 \mid \epsilon\) is not semantically equivalent to \(t\) if possible, capturing groups must be satisfied, so \(t_1\) cannot consume all matches of the expression. We encode this constraint with the implication that \(t_1\) must match the empty string whenever \(t_1 \mid \epsilon\) does.

Lookahead constrains the word to be a member of the languages of both the assertion expression and \(t_2\). The word boundary \(\backslash\) is effectively a single-character lookaround for word and non-word characters. Because the boundary can occur both ways, the model uses disjunction for the end of \(w_1\) and the start of \(w_2\) being word and non-word, or non-word and word characters, respectively. The non-word boundary \(\backslash\) is defined as the dual of \(\backslash\).

For capture groups, we bind the next capture \(C_i\) to the string matched by \(t_i\). The \(i^{\text{th}}\) capture group must be the outer capture and the remaining captures \(C_{i+1}, \ldots, C_{i+k}\) must therefore be contained within \(t_i\). There is nothing to be done for non-capturing groups and recursion continues on the contained subexpression.

Anchors assert the start (\(\ast\)) and end (\$(\) of input; we represent the beginning and end of a word via the meta-characters ( and ) respectively. In most instances when handling these operations, \(t_1\) will be \(\epsilon\); this is because it is rare to have regex operators prior to those marking the start of input (or after marking the end of input, respectively). In both these cases, we assert that the language defines the start or end of input—and that as a result of this, the language of \(t_1\) must be an empty word, though the capture groups may be defined (say through \(t_1\) containing assertions with nested captures). We
give separate rules for matching a regular expression with the multiline flag set, which modify the behavior of anchors to accept either our meta-characters or a line break.

4.3 Backreferences

Table 3 lists our models for different cases of backreferences in the AST of regex R; \( k \) is a backreference to the \( k \)th capture group of R. Intuitively, each instance of a backreference is a variable that refers to a capture group and has a type that depends on the structure of R.

We call a backreference immutable if it can only evaluate to a single value when matching; it is mutable if it can take on multiple values, which is a rare but particularly tricky case. For example, consider /((a|b)\2)*\1/. Here, the backreference \( \1 \) and the second instance of \( \2 \) are immutable. However, the first instance of \( \2 \) is mutable: each repetition of the outer capture group under the Kleene plus can change the value of the second (inner) capture group, in turn changing the value of the backreference inside this quantification. For example, the string “aabbaabbaa” satisfies this regex, but “aabbaaabbaa” does not. To fully characterize these distinctions, we introduce the following definition:

**Definition 2 (Backreference Type).** Let \( t \) be the \( k \)th capture group of a regex R. Then

1. \( \backslash k \) is empty if either \( k \) is greater than the number of capture groups in R, or \( \backslash k \) is encountered before \( t \) in a post-order traversal of the AST of R;
2. \( \backslash k \) is mutable if \( \backslash k \) is not empty, and both \( t \) and \( \backslash k \) are subterms of some quantified term Q in R;
3. otherwise, \( \backslash k \) is immutable.

When a backreference is empty, it is defined as \( \epsilon \), because it refers to a capture group that either is a superterm, e.g., /\( \{a\}\)*/, or appears later in the term, e.g., /\( \{a\}\)*/. There are two cases for immutable backreferences. In the first case, the backreference is not quantified. In our model for R, \( \mathcal{C}_k \) has already been modeled with an equality constraint, so we can bind the backreference to it. In the second case, the backreference occurs within a quantification; here, the backreference evaluates to \( \epsilon \) or \( \bot \) due to alternation or an empty Kleene star. Following the ES6 standard, the backreference evaluates to \( \epsilon \) in this case.

Mutable backreferences appear in the form (\ldots t_1 \ldots \backslash k \ldots)*, where \( t_1 \) is the \( k \)th capture group; ES6 does not support forward referencing of backreferences, so in (\ldots \backslash k \ldots t_1 \ldots)*, \( \backslash k \) is empty. For illustration purposes, the fourth entry of Table 3 describes the simplest case for mutable backreferences, other patterns are straightforward generalizations. In this case, we
assume \( t_i \) is the \( k \)-th capture group but is otherwise capture group-free. We can treat the entirety of this term at once: as such, any word in the language is either \( e \), or for some number of iterations, we have the concatenation of a word in the language of \( t_i \) followed by a copy of it. We introduce new variables \( C_{k,i} \) referring to the values of the capture group in each iteration, which encodes the repeated matching on the string until settling on the final value for \( C_{k,i} \). In this instance, we need not deal with the possibility that any \( C_{k,i} \) is \( \varnothing \), since the quantification ends as soon as \( t_i \) does not match.

Unfortunately, constraints generated from this model are hard to solve and not feasible for current SMT solvers, because they require “guessing” a partition of the matched string variable into individual and varying components. To make solving such queries practical, we introduce an alternative to the previous rule where we treat quantified backreferences as immutable. The resulting model is shown in the last row of Table 3. E.g., returning to \( (\{a\}\{b\}\{2\})^+\{1\} \backslash \{2\} / \), we accept “aaaaaaa”, “aaaaaaa”, “aaaa”, “a”), but not (“aabbabba”, “aabbababba”, “aabb”, “b”). We discuss the soundness implications in §5.4. Quantified backreferences are rare (see §7.1), so the effect is limited in practice.

### 4.4 Modeling Non-Membership

The model described so far overapproximates membership of a capturing language. We define an analogous model for non-membership of the form \( \forall c_0, \ldots, c_n : (w, c_0, \ldots, c_n) \notin L_c(R) \). Intuitively, non-membership models assert that for all capture group assignments there exists some partition of the word such that one of the individual constraints is violated. Most models are simply negated. In concatenation and quantification, only language and emptiness constraints are negated, so the models take the form

\[ w = w_1 \cdots w_k \]

\[ \land (\ldots \notin L_c(\ldots) \lor \ldots \notin L_c(\ldots) \lor (w_k = e \land \neg (w_1 = e \cdots)) \].

In the same manner, the model for capture groups is

\[ (w, c_{i+1}, \ldots, c_{i+k}) \notin L_c(t_i) \land c_i = w. \]

Returning to the example of §3.3, the negated model for \( \forall c_0, c_1 : (w, c_0, c_1) \notin L_c(\{?:a\{b\}\{1\}) \) becomes

\[ \forall c_0, c_1 : w = w_1 \cdots w_k \land (c_0 = e) \land (c_1 = e) \land (w_k = e) \land (w_1 = e) \lor (w_2 = e) \lor (w_3 = e). \]

### 5 Matching Precedence Refinement

We now explain the issue of matching precedence (§5.1) and introduce a counterexample-guided abstraction refinement scheme (§5.2) to address it. We discuss termination (§5.3) and the overall soundness of our approach (§5.4).

#### 5.1 Matching Precedence

The model in Tables 2 and 3 does not account for matching precedence (see §3.4). A standards-compliant ES6 regex matcher will derive a unique set of capture group assignments when matching a string \( w \), because matching precedence dictates that greedy (non-greedy) expressions match as many (as few) characters as possible before moving on to the next [14]. These requirements are not part of our model, as encoding them directly into SMT would require nesting of quantifiers for each operator, making them impractical for automated solving.

#### 5.2 CEGAR for ES6 Regular Expression Models

We eliminate infeasible elements of the capturing language admitted by our model through counter example-guided abstraction refinement (CEGAR).

Algorithm 1 is a CEGAR-based satisfiability checker for constraints modeled from ES6 regexes, which relies on an external SMT solver with classical regular expression and string support and an ES6-compliant regex matcher. The algorithm takes an SMT problem \( P \) (derived from the DSE path condition) as a conjunction of constraints, some of
which model the m ≥ 0 original capturing language membership constraints. We number the original capturing language constraints 0 ≤ j < m so that we can refer to them as (w_j, C_{0,j}, ..., C_{n,j}) ∩ L(R_j), where ⊆ ∈ {∈,∈}. The algorithm returns null if P is unsatisfiable, or a satisfying assignment with correct matching precedence.

In a loop, we first pass the problem P to an external SMT solver. The solver returns a satisfying assignment M or null if the problem is unsatisfiable, in which case we are done (lines 4–6). If M is not null, the algorithm uses a concrete regular expression matcher (e.g., Node.js’s built-in matcher) to populate concrete capture variables C_{i,j} corresponding to the words w_j in M.

Lines 8–22 describe how the assignments of capture groups are checked for each regular expression R_j in the original problem P. We first check whether the concrete matcher returned a list of valid capture group assignments, i.e., whether the word M[w_j] from the satisfying assignment matches concretely. If it did, then w_j is a member of the language generated by R_j. If ⊆ = ∈, i.e., the membership constraint was positive, then we must check if the capture group assignments are consistent with those from M (line 13). If they are, we move on to the next regex, otherwise we refine the constraint problem by fixing capture group assignments to their concrete values for the matched word (line 15). Dually, if a modeled non-membership constraint was satisfiable but the word from the current satisfying assignment M[w_j] did match concretely, we refine the problem by asserting that w must not equal that word (line 18). We do the same if M[w_j] did not match concretely but came from a satisfied positive membership constraint (line 22).

If no refinement was necessary we have confirmed the overall assignment satisfies P and return M (line 24). Otherwise, the loop continues with solving the refined problem.

5.3 Termination

Unsurprisingly, CEGAR may require arbitrarily many refinements on pathological formulas and never terminate. This is unavoidable due to undecidability [7]. In practice, we therefore impose a limit on the number of refinements, leading to unknown as a possible third result. SMT solvers already may timeout or report unknown for complex string formulas, so this does not lead to additional problems in practice.

5.4 Soundness

When constructing the rules in Tables 2 and 3, we followed the semantics of regular expressions as laid out in the ES6 standards document [14]. The ES6 standard is written in a semi-formal fashion, so we are confident that our translation into logic is accurate, but cannot have formal proof. Existing attempts to encode ECMAScript semantics into logic such as JSIL [8] or KJS [28] do not include regexes.

With the exception of the optimized rule for mutable backreferences, our models are overapproximate, because they ignore matching precedence. When the CEGAR loop terminates, any spurious solutions from overapproximation are eliminated. As a result, we have an exact procedure to decide (non)-membership for capturing languages of ES6 regexes without quantified backreferences.

In the presence of quantified backreferences, the model after CEGAR termination becomes underapproximate. Since DSE itself is an underapproximate program analysis (due to concretization, solver timeouts, and partial exploration), our model and refinement strategy are sound for DSE.

6 Implementation

We now describe an implementation of our approach in the DSE engine ExpoSE\footnote{ExpoSE is available at https://github.com/ExpoSEJS/ExpoSE.} [27]. We explain how to model the regex API with capturing language membership (§6.1) and give a brief overview of ExpoSE (§6.2).
6.1 Modeling the Regex API

The ES6 standard specifies several methods that evaluate regexes [14]. We follow its specified pseudocode for RegExp.exec(s) to implement matching and capture group assignment in terms of capturing language membership in Algorithm 2. Notably, our algorithm implements support for all flags and operators specified for ES6.

Algorithm 2: RegExp.exec(input)

```
input′ := (′ + input + ′);
if sticky or global then
  offset := lastIndex > 0 ? lastIndex + 1 : 0;
  input′ := input′.substring(offset);
source′ := (′(?·.|\n)*?′ + source + ′)(?:.\n)*?;
if caseIgnore then
  source′ := rewriteForIgnoreCase(source′);
if (input′, C₀,...,Cₙ) ∈ L_c(source′) then
  Remove (and ) from (input′, C₀,..., Cₙ);
  lastIndex := lastIndex + C₀.startIndex + C₀.length;
  result := [C₀,...,Cₙ];
  result.input := input;
  result.index := C₀.startIndex;
  return result;
else
  lastIndex := 0;
  return undefined;
```

6.2 ExpoSE

ExpoSE is a DSE engine that uses the Jalangi2 [19] framework to instrument a piece of JavaScript software in order to create a program trace. As the program terminates, ExpoSE calls the SMT solver Z3 [13] to identify all feasible alternate test-cases from the trace. These new test cases are then queued and the next test case is selected for execution, in the manner of generational search [18]. The ExpoSE framework allows for the parallel execution of individual test cases, aggregating coverage and alternative path information as each test case terminates. This parallelization is achieved by executing each test case as a unique process allocated to a dedicated single core; as such the analysis is highly scalable.

Our strategy for test case selection is similar to the CUPA strategy proposed by Bucur et al. [9]. We use program fork points to prioritize unexplored code: each expression is given a unique identifier and scheduled test cases are sorted into buckets based upon which expression was being executed when they were created. We select the next test case by choosing a random test case from the bucket that has been accessed least during the analysis; this prioritizes test cases triggered by less common expressions.

7 Evaluation

We now empirically answer the following research questions:

(RQ1) Are non-classical regexes an important problem in JavaScript?

(RQ2) Does accurate modeling of ES6 regexes make DSE-based test generation more effective?

(RQ3) Does the performance of the model and the refinement strategy enable practical analysis?

We answer the first question with a survey of regex usage in the wild (§7.1). We address RQ2 by comparing our approach against an existing partial implementation of regex support in ExpoSE [27] on a set of widely used libraries (§7.2). We then measure the contribution of each aspect of our approach on over 1,000 JavaScript packages (§7.3). We answer RQ3 by analyzing solver and refinement statistics per query (§7.4).

7.1 Surveying Regex Usage

We focus on code written for Node.js, a popular framework for standalone JavaScript. Node.js is used for both server and desktop applications, including popular tools Slack and Skype. We analyzed 415,487 packages from the NPM repository, the primary software repository for open source Node.js code. Nearly 35% of NPM packages contain a regex, 20% contain a capture group and 4% contain a backreference.
Results We found regex usage in JavaScript to be widespread, with 145,100 packages containing at least one regex out of a total 415,487 scanned packages. Table 4 lists the number of NPM packages containing regexes, capture groups, backreferences, and backreferences appearing within quantification. Note that a significant number of packages make use of capture groups and backreferences, confirming the importance of supporting them.

Table 5 reports statistics for all 9M regexes collected, giving for each feature the fraction of expressions including it. Many regexes in NPM packages are not unique; this appears to be due to repeated inclusion of the same literal (instead of introduction of a constant), the use of online solutions to common problems, and the inclusion of dependencies (foregoing proper dependency management). To adjust for this, we provide data for both all expressions encountered and for just unique expressions. In both cases, there are significant numbers of capture groups, backreferences, and other non-classical features. As the occurrence rate of quantified backreferences is low, we do not differentiate between mutable and immutable backreferences.

Conclusions Our findings confirm that regexes are widely used and often contain complex features. Of particular importance is a faithful treatment of capture groups, which appear in 20.45% of the packages examined. On the flip side, since quantified backreferences make up just 0.01% of regexes, the optimization introduced in §4.3 will rarely lead to additional underapproximation during DSE.

7.2 Improvement Over State of the Art

We compare our approach against the original ExpoSE [27], which is, to our knowledge, the only available and functional implementation of regex support in JavaScript.

Methodology We evaluated statement coverage achieved by both versions of ExpoSE on a set of libraries, which we chose for their popularity (with up to 20M weekly downloads) and use of regex. This includes the three libraries minimist, semver, and validator, which the first version of ExpoSE was evaluated on [27]. To fairly compare original ExpoSE against our extension, we use the original automated library harness for both. Therefore we do not take advantage of other improvements for test generation, such as symbolic array support, which we have added in the course of our work. We re-executed each package six times for one hour each on both versions, using 32-core machines with 256GB of RAM, and averaged the results. We limited the refinement scheme to 20 iterations, which we identified as effective in preliminary testing (see §7.4).

Results Table 6 contains the results of our comparison. To provide an indication of program size, we use the number of lines of code loaded at runtime (JavaScript’s dynamic method of loading dependencies makes it hard to determine a meaningful LOC count statically).

The results demonstrate that ExpoSE extended with our model and refinement strategy can improve coverage more than tenfold on our sample of widely-used libraries. In the cases of moment, query-string, and yn, the lack of ES6 support in the original ExpoSE prohibited meaningful analysis, leading to 0% coverage. In the case of semver, we see a decrease in coverage if stopped after one hour. This is due to the modeling of regex increasing solving time (see also §7.4). The coverage deficit disappears when executing both versions of ExpoSE with a timeout of two hours.
Table 6. Statement coverage with our approach (New) vs. [27] (Old) and the relative increase (+) on popular NPM packages (Weekly downloads). LOC are lines loaded and RegEx are regular expression symbols executed.

<table>
<thead>
<tr>
<th>Library</th>
<th>Weekly LOC</th>
<th>New LOC</th>
<th>RegEx Old(%) New(%) + (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>babel-eslint</td>
<td>2,500k</td>
<td>23,047</td>
<td>902 21.0 26.8 27.6</td>
</tr>
<tr>
<td>fast-xml-parser</td>
<td>20k</td>
<td>706</td>
<td>562 3.1 44.6 1,338.7</td>
</tr>
<tr>
<td>js-yaml</td>
<td>8,000k</td>
<td>6,768</td>
<td>78 4.4 23.7 438.6</td>
</tr>
<tr>
<td>minimalist</td>
<td>20,000k</td>
<td>229</td>
<td>72,530 65.9 66.4 0.8</td>
</tr>
<tr>
<td>moment</td>
<td>4,500k</td>
<td>2,572</td>
<td>21 0.0 52.6 ∞</td>
</tr>
<tr>
<td>query-string</td>
<td>3,000k</td>
<td>303</td>
<td>50 0.0 42.6 ∞</td>
</tr>
<tr>
<td>semver</td>
<td>1,800k</td>
<td>757</td>
<td>616 51.7 46.2 -10.6</td>
</tr>
<tr>
<td>url-parse</td>
<td>1,400k</td>
<td>322</td>
<td>448 60.9 71.8 17.9</td>
</tr>
<tr>
<td>validator</td>
<td>1,400k</td>
<td>2,155</td>
<td>94 67.5 72.2 7.0</td>
</tr>
<tr>
<td>xml</td>
<td>500k</td>
<td>276</td>
<td>1,022 60.2 77.5 28.7</td>
</tr>
<tr>
<td>yn</td>
<td>700k</td>
<td>157</td>
<td>260 0.0 54.0 ∞</td>
</tr>
</tbody>
</table>

Conclusions We find that our modifications to ExpoSE make test generation more effective in widely used libraries using regex. This suggests that the new method of solving regex queries presented in this paper has a substantial impact on practical problems in DSE. We also see that other improvements to ExpoSE, such as ES6 support, have affected coverage. Therefore, we continue with an evaluation of the individual aspects of our model.

7.3 Breakdown of Contributions

We now drill down into how the individual improvements in regex support are contributing to increases in coverage.

Methodology From the packages with regexes from our survey §7.1, we developed a test suite of 1,131 NPM libraries for which ExpoSE is able to automatically generate a meaningful test harness. In each of the libraries selected, ExpoSE executed at least one regex operation on a symbolic string, which ensures that the library contains some behavior relevant to the scope of this paper. The test suite constructed in this manner contains numerous libraries where dependencies of packages widely used in industry, including Express and Lodash.²

Automatic test generation typically requires a bespoke test harness or set of parameterized unit tests [33] to achieve high coverage in code that does not have a simple command line interface, including libraries. ExpoSE’s harness explores libraries fully automatically by executing all exported methods with symbolic arguments for the supported types string, boolean, number, null and undefined. Returned objects or functions are also subsequently explored in the same manner.

We executed each package for one hour, which typically allowed to reach a (potentially initial) coverage plateau, at which additional test cases do not increase coverage further. We break down our regex support into four levels and measure the contribution and cost of each one to line coverage and test execution rate (Table 7). As baseline, we first execute all regex methods concretely, concretizing the arguments and results. In the second configuration, we add the model for ES6 regex and their methods, including support for word boundaries and lookaheads, but remove capture groups and concretize any accesses to them, including backreferences. Third, we also enable full support for capture groups and backreferences. Fourth, we finally also add the refinement scheme to address overapproximation.

Results Table 7 shows, for each level of support, the number and percentage of target packages where coverage improved; the geometric mean of the relative increase in coverage; and the mean test execution rate. The final row shows the effect of enabling full support compared to the baseline. Note that the number of packages improved is less than the sum of the rows above, since the coverage of a package can be improved by multiple features.

In a dataset of this size that includes many libraries that make only little use of regex, average coverage increases are expected to be small. Nevertheless, we see that dedicated support improves the coverage of more than half of packages that symbolically executed at least one regex function. As expected, the biggest improvement comes from supporting basic symbolic execution of regular expressions, even without capture groups or regard for matching precedence. However, we see further improvements when adding capture groups, which shows that they indeed affect program semantics. Refinement affects fewer packages, although it significantly contributes to coverage where it is required. This is because a lucky solver may generate correct inputs on the first attempt, even in ambiguous settings.

On some libraries in the dataset, the approach is highly effective. For example, in the manifest parser n4mf-parser, full support improves coverage by 29% over concrete; in the format conversion library sbxml2json, by 14%; and in the

Table 7. Breakdown of how different components contribute to testing 1,131 NPM packages, showing number (#) and fraction (%) of packages with coverage improvements, the geometric mean of the relative coverage increase from the feature (Cov), and test execution rate.

<table>
<thead>
<tr>
<th>Regex Support Level</th>
<th>Improved # %</th>
<th>Cov +%(%)</th>
<th>Tests min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concrete Regular Expressions</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>+ Modeling RegEx</td>
<td>528 46.68%</td>
<td>+6.16%</td>
<td>10.14</td>
</tr>
<tr>
<td>+ Captures &amp; Backreferences</td>
<td>194 17.15%</td>
<td>+4.18%</td>
<td>9.42</td>
</tr>
<tr>
<td>+ Refinement</td>
<td>63 5.57%</td>
<td>+4.17%</td>
<td>8.70</td>
</tr>
<tr>
<td>All Features vs. Concrete</td>
<td>617 54.55%</td>
<td>+6.74%</td>
<td></td>
</tr>
</tbody>
</table>
Table 8. Solver times per package and query.

<table>
<thead>
<tr>
<th>Packages/Queries</th>
<th>Constraint Solver Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
</tr>
<tr>
<td>All packages</td>
<td>0.04s</td>
</tr>
<tr>
<td>With capture groups</td>
<td>0.20s</td>
</tr>
<tr>
<td>With refinement</td>
<td>0.46s</td>
</tr>
<tr>
<td>Where refinement limit is hit</td>
<td>3.49s</td>
</tr>
<tr>
<td>All queries</td>
<td>0.001s</td>
</tr>
<tr>
<td>With capture groups</td>
<td>0.001s</td>
</tr>
<tr>
<td>With refinement</td>
<td>0.005s</td>
</tr>
<tr>
<td>Where refinement limit is hit</td>
<td>0.120s</td>
</tr>
</tbody>
</table>

browser detection library *mario*, by 16%. In each of these packages the refinement scheme contributed to the improvement in coverage. In general, the largest increases are seen in packages that include regular expression-based parsers.

Each additional feature causes a small decrease in average test execution rate. Although a small fraction (~1%) of queries can take longer than 300s to solve, concurrent test execution prevents DSE from stalling on a single query.

**Conclusions** Full support for ES6 regex improves performance of DSE of JavaScript in practice at a cost of a 16% increase in execution time (RQ2). An increase in coverage at lower execution rate in a fixed time window suggests that full regular expression support increases the quality of individual test cases.

### 7.4 Effectiveness on Real-World Queries

We now investigate the performance of the model and refinement scheme to answer RQ3. Finally, we also discuss the refinement limit and how it affects analysis.

**Methodology** We collected data on queries during the NPM experiments (§7.3) to provide details on SMT query success rates and execution times, as well as on the usage of the refinement scheme.

**Results** We found that 753 (66%) of the 1,131 packages tested executed at least one query containing a capture group or backreference. Of these packages, 653 (58%) contained at least one query to the SMT solver requiring refinement, and 134 (12%) contained a query that reached the refinement limit.

In total, our experiments executed 58,390,184 SMT queries to generate test cases. As expected, the majority do not involve regexes, but they form a significant part: 4,489,581 (7.6%) queries modeled a regex, 645,295 (1.1%) modeled a capture group or backreference, 74,076 (0.1%) required use of the refinement scheme and 2,079 (0.003%) hit the refinement limit. The refinement scheme was overwhelmingly effective: only 2.8% of queries with at least one refinement also reached the refinement limit (0.003% of all queries where a capture group was modeled). Of the refined SMT queries, the mean number of refinements required to produce a valid satisfying assignment was 2.9; the majority of queries required only a single refinement.

Table 8 details time spent processing SMT problems per-package and per-query. We provide the data over the four key aspects of the problem: we report the time spent in the constraint solver both per package and per query in total, as well as the time in the constraint solver for the particularly challenging parts of our strategy. We found that the use of refinements increased the average per-query solving time by a factor of four; however, this is dominated by SMT queries that hit the refinement limit, which took ten times longer to run on average. The low minimum time spent in the solver in some packages can be attributed to packages where a regular expression was encountered early in execution but limitations in the test harness or function models (unrelated to regular expressions) prevented further exploration.

**Conclusions** We find the refinement scheme is highly effective, as it is able to solve 97.2% of encountered constraint problems containing regexes. It is also necessary, as 10% of queries containing a capture group had led to a spurious satisfying assignment and required refinement.

Usually, only a small number of refinements are required to produce a correct satisfying assignment. Therefore, even refinement limits of five or fewer are feasible and may improve performance with low impact on coverage.

### 7.5 Threats to Validity

We now look at potential issues affecting the validity of our results, in particular soundness, package selection, and scalability.

**Soundness** In addition to soundness of the model (see §5.4), one must consider the soundness of the implementation. In the absence of a mechanized specification for ES6 regex, our code cannot be proven correct, so we use an extensive test suite for validation. However, assuming the concrete matcher is specification-compliant, Algorithm 1 will, if it terminates, return a specification-compliant model of the constraint formula even if the implementation of §4 contains bugs. In the worst case, the algorithm would not terminate, leading to timeouts and loss of coverage. Bugs could therefore only have lowered the reported coverage improvements.

**Package Selection and Harness** In §7.3, we chose packages identified in our survey (§7.1) where our generic harness encountered a regular expression within one hour of DSE. This allowed us to focus the evaluation on regex support as opposed to evaluating the quality of the harness (and having to deal with unreachable code in packages). Use of this harness may have limited package selection to simpler, unrepresentative libraries. However, we found that simple APIs do not imply simple code: the final dataset contains several
complex packages, such as language parsers, and the types of regexes encountered were in line with the survey results. On simple code we found that ExpoSE would often reach 100% coverage; failure to do so was either due to the complexity of the code or the lack of support for language features unrelated to regex and APIs that would require additional modeling (e.g., the file system).

**Scalability** Scalability is a challenge for DSE in general, and is not specific to our model for regex. Empirically, execution time for a single test (instrumentation, execution, and constraint generation) grows linearly with program size, as does the average size of solver queries. The impact of query length on solving time varies, but does not appear to be exacerbated by our regex model. In principle, our model is compatible with compositional approaches [4, 16] and state merging [5, 21], which can help DSE scale to large programs.

The scalability of our approach suffices for Node.js, however. JavaScript has smaller LOC counts than, e.g., C++, and code on NPM is very modular. For instance, among the top 25 most depended-upon NPM libraries, the largest is 30 KLOC (but contains no regex). Several packages selected for our evaluation, such as babel-eslint, had between 20-30 KLOC and were meaningfully explored with the generic harness.

### 8 Related Work

In prior work, we introduced ExpoSE and partial support for encoding JavaScript regex in terms of classical regular language membership and string constraints [27]. This initial take on the problem was lacking support for several problematic features such as lookaheads, word boundaries, and anchors. Matching precedence was presented as an open problem, which we have now addressed through our refinement scheme.

In theory, regex engines can be symbolically executed themselves through the interpreter [9]. While this removes the need for modeling, in practice the symbolic execution of the entire interpreter and regex engine quickly becomes infeasible due to path explosion.

There have been several other approaches for symbolic execution of JavaScript; most include some limited support for classical regular expressions. Li et al. [24] presented an automated test generation scheme for programs with regular expressions by on-line generation of a matching function for each regular expression encountered, exacerbating path explosion. Saxena et al. [29] proposed the first scheme to encode capture groups through string constraints. Sen et al. [31] presented Jalangi, a tool based on program instrumentation and concolic values. Li and Ghosh [23] and Li et al. [22] describe a custom browser and symbolic execution engine for JavaScript and the browser DOM, and a string constraint solver PASS with support for most JavaScript string operations. Although all of these approaches feature some support for ECMAScript regex (such as limited support for capture groups), they ignore matching precedence and do not support backreferences or lookaheads.

Thomé et al. [32] propose a heuristic approach for solving constraints involving unsupported string operations. We choose to model operations unsupported by the solver and employ a CEGAR scheme to ensure correctness. Abdulla et al. [2] propose the use of a refinement scheme to solve complex constraint problems, including support for context-free languages. The language of regular expressions with backreferences is not context-free [10] and, as such, their scheme does not suffice for encoding all regexes; however, their approach could serve as richer base theory than classic regular expressions. Scott et al. [30] suggest backreferences can be eliminated via concatenation constraints, however they do not present a method for doing so.

Further innovations from the string solving community, such as work on the decidability of string constraints involving complex functions [12, 20] or support for recursive string operations [35, 36], are likely to improve the performance of our approach in future. We incorporate our techniques at the level of the DSE engine rather than the constraint solver, which allows our tool to leverage advances in string solving techniques; at the same time, we can take advantage of the native regular expression matcher and can avoid having to integrate implementation language-specific details for regular expressions into the solver.

A previous survey of regex usage across 4,000 Python applications [11] also provides a strong motivation for modeling regex. Our survey extends this work to JavaScript on a significantly larger sample size.

### 9 Conclusion

In this paper we presented a model for the complete regex language of ES6, which is sound for the dynamic symbolic execution of the test and exec functions. We model regex membership constraints in terms of string constraints and classical regular language membership. We introduced a novel CEGAR scheme to address the challenge of matching precedence, which so far had been largely ignored in related work. To the best of our knowledge, ours is the first comprehensive solution for ES6. We demonstrated that regexes—and specifically their non-regular features—are extensively used in JavaScript and that existing DSE-based analyses would therefore suffer coverage loss from concretization. In a large scale evaluation of over 1,000 Node.js programs, our novel solution outperforms existing partial approaches to the problem and demonstrates the viability of our model for improving the analysis of string-manipulating JavaScript programs.

### Acknowledgments

Blake Loring was supported by the EPSRC Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/K035584/1).
References


