Forschungsinstitut
Cyber Defence
Universität der Bundeswehr München

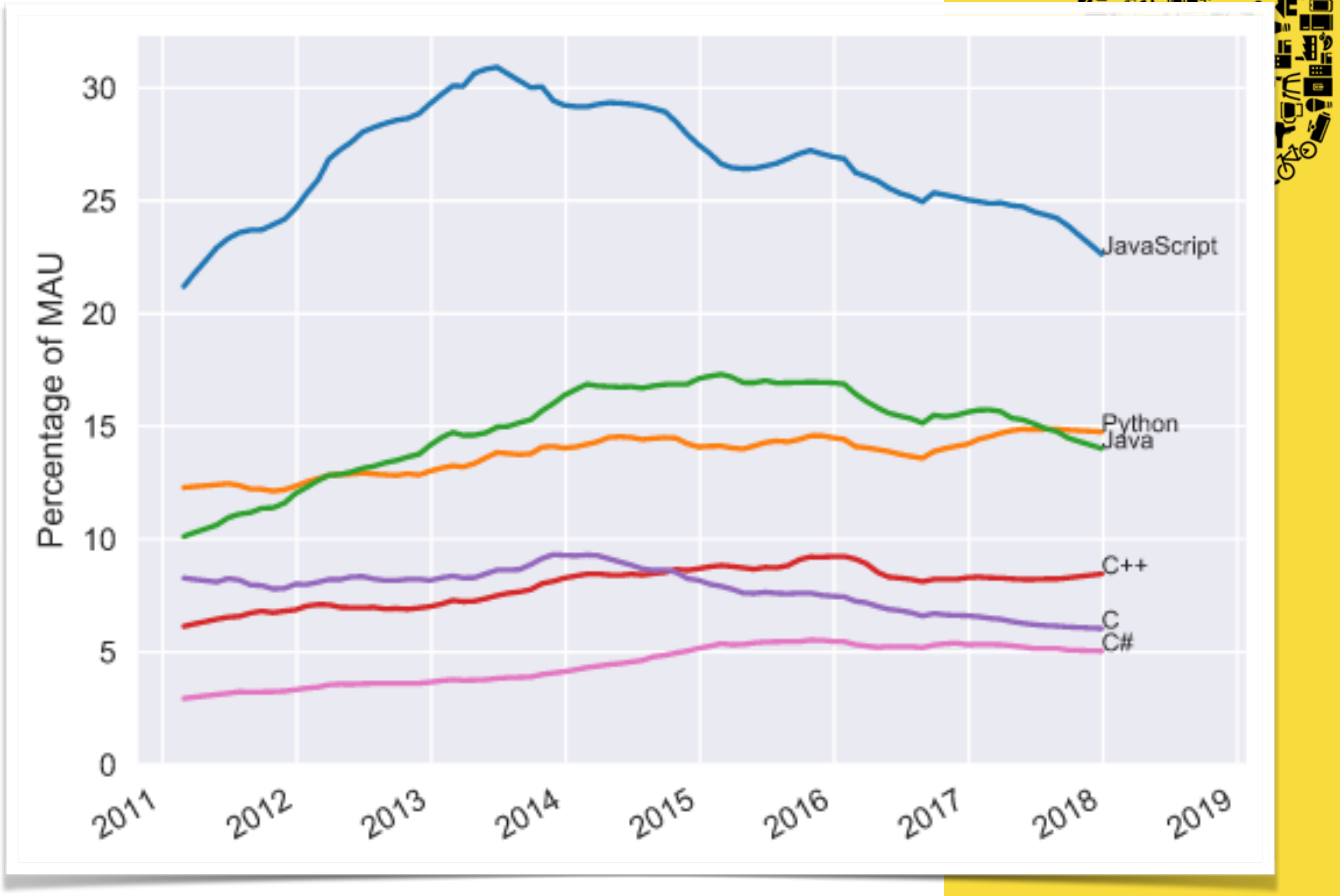
Sound DSE Semantics for JavaScript Regular Expressions

Johannes Kinder, *Research Institute CODE, Bundeswehr University Munich*

joint work with
Blake Loring and Duncan Mitchell, *Royal Holloway, University of London*

JavaScript

- The language of the web
- Increasingly popular as server-side (Node.js) and client side (Electron) solution.
- Top 10 language (Github)



Mission Statement

- Help find bugs in Node.js applications and libraries
- JavaScript is a dynamic language
  - Don't force it into a static type system
  - Static analysis becomes very hard
- Embrace it and go for dynamic approach
  - Re-use existing interpreters where possible

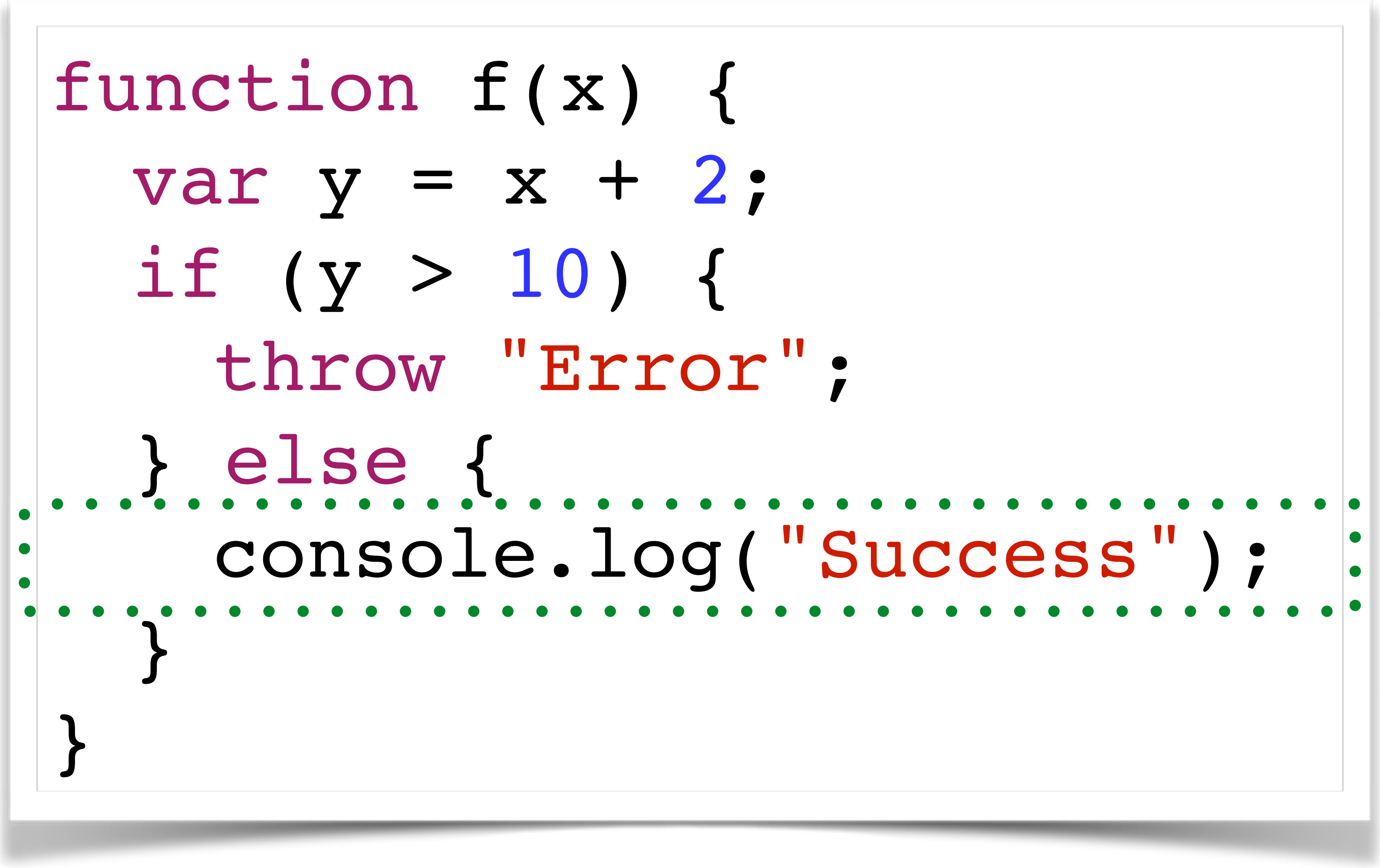
Dynamic Verification

- Similar issues as in x86 binary code
  - No types, self-modifying code
- Most successful methods for binaries are dynamic
  - Fuzz testing
  - Dynamic symbolic execution
- No safety proofs, but proofs of vulnerabilities

```
55      pushq   %rbp
48 89 e5        movq    %rsp, %rbp
48 83 ec 20     subq    $32, %rsp
48 8d 3d 77 00 00 00   leaq    119(%
48 8d 45 f8     leaq    -8(%rbp), %ra
48 8d 4d fc     leaq    -4(%rbp), %rc
c7 45 fc 90 00 00 00   movl    $144,
c7 45 f8 e8 03 00 00   movl    $1000
48 89 4d f0     movq    %rcx, -16(%rb
48 89 45 e8     movq    %rax, -24(%rb
48 8b 45 e8     movq    -24(%rbp), %r
8b 10   movl    (%rax), %edx
48 8b 45 f0     movq    -16(%rbp), %r
89 10   movl    %edx, (%rax)
8b 75 fc        movl    -4(%rbp), %es
b0 00   movb    $0, %al
e8 21 00 00 00 callq   33
48 8d 3d 3c 00 00 00   leaq    60(%r
8b 75 f8        movl    -8(%rbp), %es
89 45 e4        movl    %eax, -28(%rb
b0 00   movb    $0, %al
e8 0d 00 00 00 callq   13
31 d2   xorl    %edx, %edx
89 45 e0        movl    %eax, -32(%rb
89 d0   movl    %edx, %eax
48 83 c4 20     addq    $32, %rsp
5d      popq    %rbp
c3      retq
55      pushq   %rbp
48 89 e5        movq    %rsp, %rbp
48 83 ec 20     subq    $32, %rsp
48 8d 3d 77 00 00 00   leaq    119(%
48 8d 45 f8     leaq    -8(%rbp), %ra
48 8d 4d fc     leaq    -4(%rbp), %rc
c7 45 fc 90 00 00 00   movl    $144,
c7 45 f8 e8 03 00 00   movl    $1000
48 89 4d f0     movq    %rcx, -16(%rb
48 89 45 e8     movq    %rax, -24(%rb
48 8b 45 e8     movq    -24(%rbp), %r
8b 10   movl    (%rax), %edx
48 8b 45 f0     movq    -16(%rbp), %r
89 10   movl    %edx, (%rax)
8b 75 fc        movl    -4(%rbp), %es
b0 00   movb    $0, %al
e8 21 00 00 00 callq   33
48 8d 3d 3c 00 00 00   leaq    60(%r
8b 75 f8        movl    -8(%rbp), %es
89 45 e4        movl    %eax, -28(%rb
b0 00   movb    $0, %al
e8 0d 00 00 00 callq   13
31 d2   xorl    %edx, %edx
89 45 e0        movl    %eax, -32(%rb
89 d0   movl    %edx, %eax
48 83 c4 20     addq    $32, %rsp
5d      popq    %rbp
c3      retq
ff 25 86 00 00 00      jmpq    *134(
4c 8d 1d 75 00 00 00   leaq    117(%
41 53   pushq   %r11
ff 25 65 00 00 00      jmpq    *101(
90      nop
68 00 00 00 00 pushq   $0
e9 e6 ff ff ff jmp     -26 <__stub_
```

Dynamic Symbolic Execution

- Automatically explore paths
  - Replay tested path with “symbolic” input values
  - Record branching conditions in "path condition"
  - Spawn off new executions from branches



```
function f(x) {
  var y = x + 2;
  if (y > 10) {
    throw "Error";
  } else {
    console.log("Success");
  }
}
```

- Constraint solver
  - Decides path feasibility
  - Generates test cases

Run 1: `f(0)`:

PC: true
$x \mapsto X$

PC: true
$x \mapsto X$
$y \mapsto X + 2$

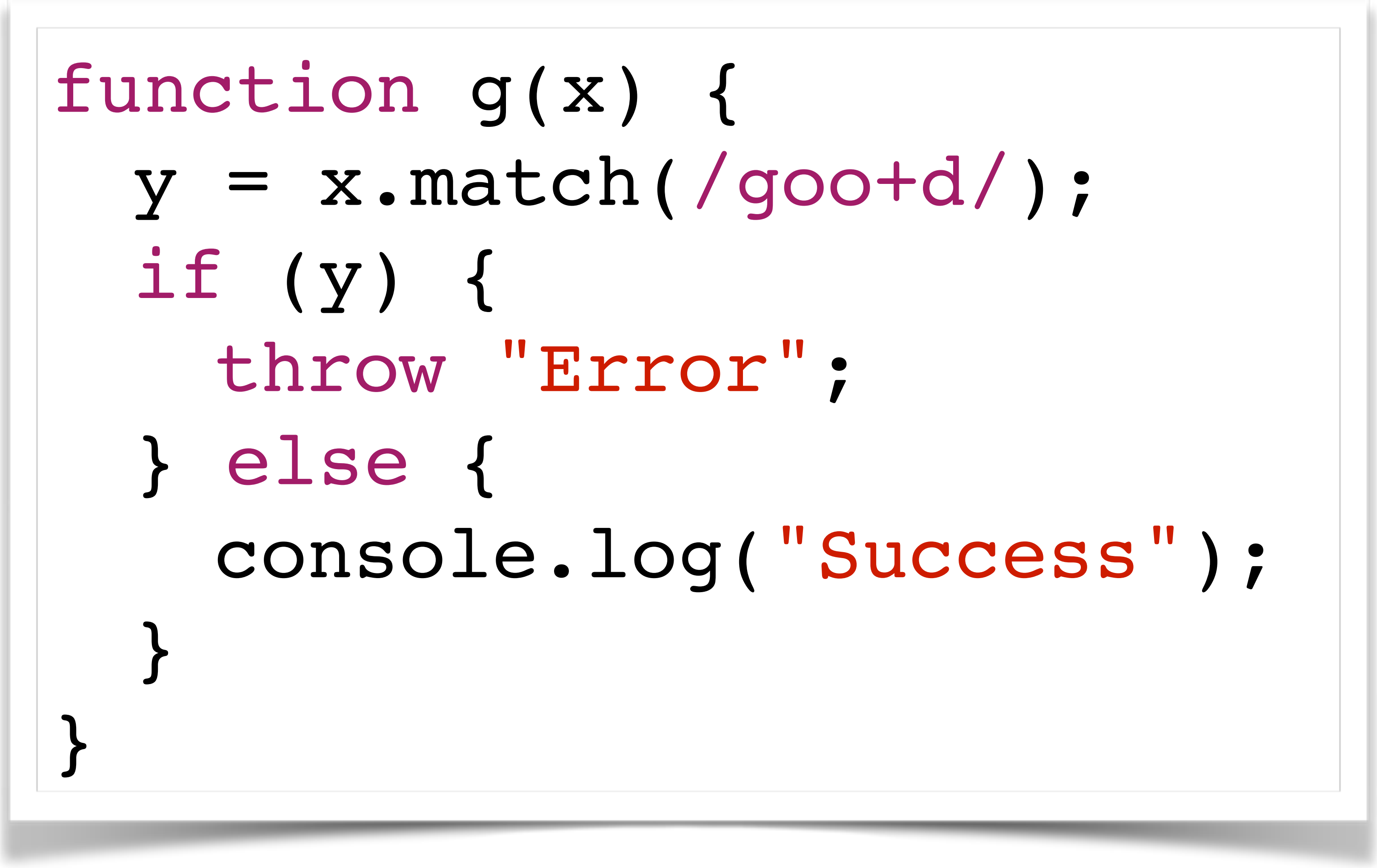
PC: $X + 2 \leq 10$
$x \mapsto X$
$y \mapsto X + 2$

Query: $X + 2 > 10$

Run 2: `f(9)`

High-Level Language Semantics

- Classic DSE focuses on C / x86 / Java bytecode
  - Straightforward encoding to bitvector SMT
  - Library functions effectively inlined
- JavaScript / Python etc. have rich builtins
  - Do more with fewer lines of code
  - Strings, regular expressions



```
function g(x) {
  y = x.match(/goo+d/);
  if (y) {
    throw "Error";
  } else {
    console.log("Success");
  }
}
```

Node.js Package Manager

Feature	**Count**	%
Packages on NPM	415,487	100.0%
. . . with source files	381,730	91.9%
. . . with regular expressions	145,100	34.9%
. . . with capture groups	84,972	20.5%
. . . with backreferences	15,968	3.8%
. . . with quantified backreferences	503	0.1%

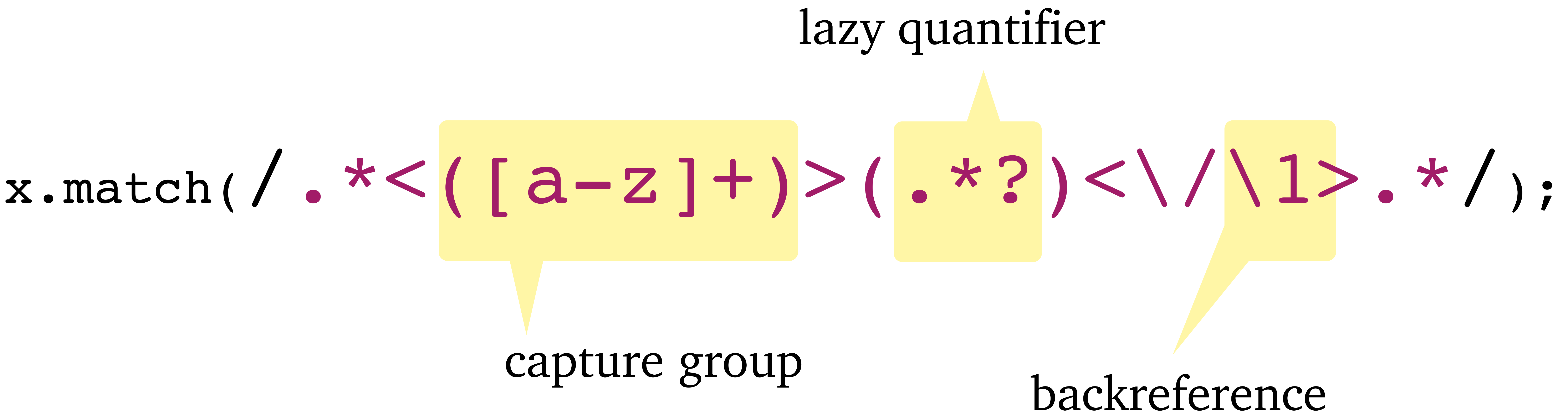
Regular Expressions

- What's the problem?
  - First year undergrad material
  - Supported by SMT solvers: strings + regex in Z3, CVC4
- SMT formulae can include regular language membership

$$(x = \text{"foo"} + s) \wedge (len(x) < 5) \wedge (x \in \mathscr{L}(\text{goo+d}))$$

Regular Expressions in Practice

- Regular expressions in most programming languages (Regex) aren't regular!



lazy quantifier

```
x.match(/.*<([a-z]+)>(.*?)<\/\1>.*/);
```

capture group

backreference

- Not supported by solvers

Regular Expressions in Practice

- There's more than just testing membership

```
x.match(/.*<([a-z]+)>(.*?)<\/\1>.*/);
```

- Capture group contents are extracted and processed

```
function f(x, maxLen) {
 var s = x.match(/.*<([a-z]+)>(.*?)<\/\1>.*/);
 if (s) {
  if (s[2].length <= 0) {
   console.log("*** Element missing ***");
  } else if (s[2].length > maxLen) {
   console.log("*** Eleme
  } else {
   console.log("*** Succe
  }
 } else {
  console.log("*** Malformed XML ***");
 }
}
```

match returns array with matched contents

[0]	Entire matched string
[1]	Capture group 1
[2]	Capture group 2
[n]	Capture group n

Capturing Languages

- Need to include capture values in the word problem

- Capturing language membership

$$(w, s_1, s_2) \in \mathscr{L}(\texttt{.*<(a+)>.*?<\textbackslash/\textbackslash1>.*})$$

- Capturing language: tuples of words and capture group values
  - Given a word and a regex, the capture values are uniquely defined by the regex matching semantics

Encoding Regex

- Idea: split expression and use concatenation constraints

$$(w, s_1, s_2) \in \mathscr{L}(\texttt{.*<}\boxed{\texttt{(a+)}}\texttt{>.*?<\textbackslash/\textbackslash1>.*})$$

$$\Downarrow$$

$$s_1 \in \mathscr{L}(\boxed{\texttt{a+}})$$

Encoding Regex

- Idea: split expression and use concatenation constraints

$$(w,\ s_1,\ s_2) \in \mathscr{L}(\texttt{.*<(a+)>}\boxed{\texttt{.*?}}\texttt{<\textbackslash/\textbackslash1>.*})$$

$$\Downarrow$$

$$s_1 \in \mathscr{L}(\texttt{a+}) \wedge s_2 \in \mathscr{L}(\boxed{\texttt{.*}})$$

Encoding Regex

- Idea: split expression and use concatenation constraints

$$(w, s_1, s_2) \in \mathscr{L}(\mathtt{.*<(a+)>.*?<\backslash/\backslash 1>.*})$$

$$\Downarrow$$

$$s_1 \in \mathscr{L}(\mathtt{a+}) \land s_2 \in \mathscr{L}(\mathtt{.*}) \land w = t_1 + \mathtt{"<"} + s_1 + \mathtt{">"} + s_2 + \mathtt{"<\backslash/"} + s_1 + \mathtt{">"} + t_2$$

- Addresses backreferences successfully

Greediness vs. Captures

- Doesn't guarantee correct capture values!

$$(w, s_1, s_2) \in \mathscr{L}(\texttt{.*<(a+)>.*?<\textbackslash/\textbackslash1>.*})$$

$$\Downarrow$$

$$s_1 \in \mathscr{L}(\texttt{a+}) \land s_2 \in \mathscr{L}(\texttt{.*}) \land w = t_1 + \texttt{"<"} + s_1 + \texttt{">"} + s_2 + \texttt{"<\textbackslash/"} + s_1 + \texttt{">"} + t_2$$

- SAT: $s_1 = \texttt{"a"}$; $s_2 = \texttt{""}$, with $w = \texttt{"<a>"}$

✗ **Too permissive! Over-approximating matching precedence (greediness)**

Greediness vs. Captures

$$s_1 \in \mathscr{L}(\texttt{a+}) \wedge s_2 \in \mathscr{L}(\texttt{.*}) \wedge w = t_1 + \texttt{"<"} + s_1 + \texttt{">"} + s_2 + \texttt{"<\textbackslash/"} + s_1 + \texttt{">"} + t_2$$

- SAT: $s_1 = \texttt{"a"}$; $s_2 = \texttt{""}$, with $w = \texttt{"<a>"}$
- Execute `"<a></a></a>".match(/.*<(a+)>.*?<\/\1>.*/)` & compare
- Conflicting captures: generate refinement clause from concrete result

$$\wedge\ (w = \texttt{"<a>"} \rightarrow s_1 = \texttt{"a"} \ \wedge\ s_2 = \texttt{""})$$

- SAT, model $s_1 = \texttt{"a"}$; $s_2 = \texttt{""}$

Counter Example-Guided Abstraction Refinement

Greediness vs. Captures

$$s_1 \in \mathscr{L}(\texttt{a+}) \land s_2 \in \mathscr{L}(\texttt{.*}) \land w = t_1 + \texttt{"<"} + s_1 + \texttt{">"} + s_2 + \texttt{"<\textbackslash/"} + s_1 + \texttt{">"} + t_2$$

- SAT: $s_1 = \texttt{"a"}$; $s_2 = \texttt{""}$, with $w = \texttt{"<a>"}$
- Execute `"<a></a></a>".match(/.*<(a+)>.*?<\/\1>.*/)` & compare
- Conflicting captures: generate refinement clause from concrete result

$$\land\ (w = \texttt{"<a>"} \rightarrow s_1 = \texttt{"a"}\ \land\ s_2 = \texttt{""})$$

- SAT, model $s_1 = \texttt{"a"}$; $s_2 = \texttt{""}$

Refinement scheme with four cases
(positive - negative, match - no match) ✔

I didn't mention...

- Implicit wildcards: regex matches anywhere in text
    - Anchors ^ and $ control positioning

`/^start$/`

- Lookarounds specify language constraints

`/^start(?!.*end$)middle/`

- Statefulness
    - Affected by flags

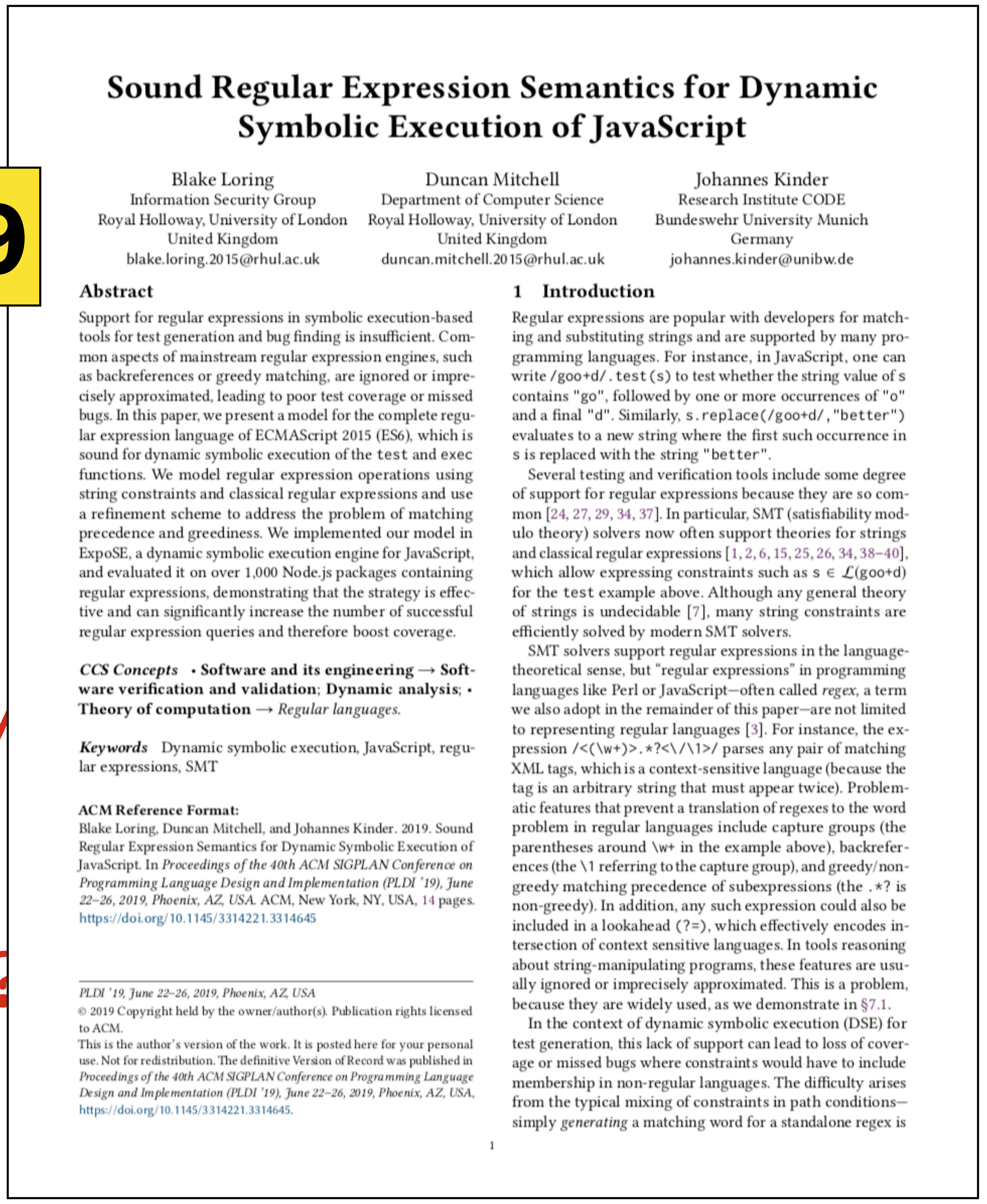
```
r = /goo+d/g;
r.test("goood"); // true
r.test("goood"); // false
r.test("goood"); // true
```

- Nesting
    - Capture groups, alternation, updatable backreferences

`/((a|b)\2)+/`

I didn't mention…

PLDI'19



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@unibw.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:
Blake Loring, Duncan Mitchell, and Johannes Kinder. 2019. Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. 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 $s \in \mathcal{L}(\texttt{goo+d})$ 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+)>.*?<\/\1>/` 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 `.*?` is 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

---

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

The definitive Version of Record was published in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA,* https://doi.org/10.1145/3314221.3314645.

- Implicit wildcards: regex matches anywhere in text
    - Anchors ^ and $ control positioning

```
/^start$/
```

- Lookarounds specify language constraints

```
/^start(?
```

- Statefulness
    - Affected by flags

```
r = /goo+d/g;
r.test("goood"); // true
r.test("goood"); // false
r.test("goood"); // true
```

- Nesting
    - Capture groups, alternation, updatable backreferences

```
/((a|b)\2)+/
```

ExpoSE

- Dynamic symbolic execution engine for ES6 [SPIN'17]
  - Built in JavaScript (node.js) using Jalangi 2 and Z3
  - SAGE-style generational search (complete path first, then fork all)
- Symbolic semantics
  - Pairs of concrete and symbolic values
  - Symbolic reals (instead of floats), Booleans, strings, regex
  - Implement JavaScript operations on symbolic values

Evaluation

- Effectiveness for test generation
  - Generic library harness exercises exported functions: successfully encountered regex on 1,131 NPM packages
- How much can we increase coverage through full regex support?
  - Gradually enable encoding and refinement, measure increase in coverage

Performance

Library	Weekly	LOC	Regex	Coverage
babel-eslint	2,500k	23,047	902	26.8%
fast-xml-parser	20k	706	562	44.6%
js-yaml	8,000k	6,768	78	23.7%
minimist	20,000k	229	72,530	66.4%
moment	4,500k	2,572	21	52.6%
query-string	3,000k	303	50	42.6%
semver	1,800k	757	616	46.2%
url-parse	1,400k	322	448	71.8%
validator	1,400k	2,155	94	72.2%
xml	500k	276	1,022	77.5%
yn	700k	157	260	54.0%

Coverage Improvement Breakdown

	Improved		Coverage	Speed
Regex Support Level	**#**	**%**	**+%**	**Tests/min**
Concrete Regular Expressions	-	-	-	11.46
+Modeling Regex	528	46.68%	+ 6.16%	10.14
+Captures and Backreferences	194	17.15%	+ 4.18%	9.42
+Refinement	63	5.57%	+ 4.17%	8.70
All Features vs. Concrete	617	54.55%	+ 6.74%	

On 1,131 NPM packages where a regex was encountered on a path

Conclusion

We're hiring!

- Supporting real-world regex
  - Defined capturing languages for regex
  - Capture values affected by greedy / lazy matching
- Model JS regex for Dynamic Symbolic Execution
  - Encode to classic regular expressions and string constraints
  - CEGAR scheme to address matching precedence / greediness

https://github.com/ExpoSEJS

https://unibw.de/patch
✉ johannes.kinder@unibw.de
@johannes_kinder

CODE FI
Forschungsinstitut
Cyber Defence
Universität der Bundeswehr München