Practical Dynamic Symbolic Execution of Standalone JavaScript

Johannes Kinder

Royal Holloway, University of London

Joint work with Blake Loring and Duncan Mitchell

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, invalidates common patterns
  - Static analysis becomes very hard, many sources of precision loss
- Embrace it and go for dynamic approach

- 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
- Constraint solver
  - Decides path feasibility
  - Generates test cases

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

Run 1: `f(0)`:

```
PC: true
x ↦ X
```

```
PC: true
x ↦ X
y ↦ X + 2
```

```
PC: X + 2 ≤ 10
x ↦ X
y ↦ X + 2
```

Query: `X + 2 > 10`

Run 2: `f(9)`

High-Level Language Semantics

- Classic DSE focuses on C / x86
  - Straightforward encoding to bitvector SMT
- High-level languages are richer
  - 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	%
Total Packages	415,487	100.00%
Packages with regular expression	145,100	34.92%
Packages with captures	84,972	20.45%
Packages with no source files	33,757	8.10%
Packages with backreferences	15,968	3.84%
Packages with quantified backreferences	503	0.12%

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

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

Regular Expressions in Practice

- Regular expressions in most programming languages aren't regular!
- Not supported by solvers

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

Regular Expressions in Practice

- Regular expressions in most programming languages aren't regular!
- Not supported by solvers

lazy quantifier

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

capture group

backreference

Regular Expressions in Practice

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

- There's more than just testing membership
- 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

- Idea: split expression and use concatenation constraints

$$t \in \mathscr{L}(\texttt{/<(a+)>.*?<\textbackslash/\textbackslash 1>/})$$

$$\exists\, s_1, s_2 : (\, s_1 \in \mathscr{L}(\texttt{/a+/}) \wedge s_2 \in \mathscr{L}(\texttt{/>.*<\textbackslash//}) \wedge t = \texttt{"<"} + s_1 + s_2 + s_1 + \texttt{">"})$$

- Works for membership

- Correct language membership doesn't guarantee correct capture values!

$$t \in \mathscr{L}(\texttt{/<(a+)>.*?<\textbackslash/\textbackslash 1>/})$$

⬇

$$\exists\, s_1, s_2 : (\, s_1 \in \mathscr{L}(\texttt{/a+/}) \wedge s_2 \in \mathscr{L}(\texttt{/>.*<\textbackslash//}) \wedge t = \texttt{"<"} + s_1 + s_2 + s_1 + \texttt{">"})$$

- SAT: $s_1 = \texttt{"a"}$; $s_2 = \texttt{"></"}$; therefore $t = \texttt{"<a>"}$

Too permissive! Over-approximating matching precedence (greediness)

✗

$$\exists\, s_1, s_2 : (\, s_1 \in \mathscr{L}(\texttt{/a+/}) \wedge s_2 \in \mathscr{L}(\texttt{/>.*<\textbackslash//}) \wedge t = \texttt{"<"} + s_1 + s_2 + s_1 + \texttt{">"})$$

- SAT: $s_1 = \texttt{"a"}$; $s_2 = \texttt{"></"}$; therefore $t = \texttt{"<a>"}$
- Execute $\texttt{"<a>".match(/<(a+)>.*?<\textbackslash/\textbackslash 1>/)}$ and compare
- Conflicting captures: generate blocking clause from concrete result

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

- SAT, model $s_1 = \texttt{"aa"}$; $s_2 = \texttt{"></"}$; therefore $t = \texttt{"<a>"}$

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

I didn't mention...

- Implicit wildcards
  - Regex is implicitly surrounded with `.*?`
- Statefulness
  - Affected by flags
- Nesting
  - Capture groups, alternation, updatable backreferences

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

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

ExpoSE

- Dynamic symbolic execution engine (prototype) [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, regular expressions
  - 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

Coverage Increase

Feature	Improved	%	Coverage	Tests/m
No Support	0	0%	+0%	11.46
Regular	528	46.68%	+3.09%	10.14
Encoding	194	17.15%	+2.3%	9.42
Refinement	63	5.57%	+2.18%	8.7
Overall	617	54.55%	+3.39%	

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

Conclusion

- Symbolic execution of code with ECMAScript regex
  - Encode to classic regular expressions and string constraints
  - CEGAR scheme to address matching precedence / greediness
- Robust implementation in ExpoSE
  - Automatic test generation - test oracles currently offloaded to developers
  - Full support for ES5 node.js, including async, eval, regex

https://github.com/ExpoSEJS