Program Analysis, Understanding, and Synthesis with Symbolic Execution

Jeff Foster
University of Maryland, College Park

Joint work with Elnatan Reisner, Kin-Keung Ma, Charles Song, Adam Porter, Avik Chaudhuri, Yit Phang Khoo, Bor-Yuh Evan Chang, Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri

Introduction

- Static analysis is great
  - Lots of interesting ideas and tools
  - Commercial companies sell, use static analysis
  - It all looks good on paper, and in papers
- But can developers use it?
  - Our experience: Not easily
  - Results in papers describe use by static analysis experts
  - Commercial tools have a huge code mass to deal with developer confusion, false positives, warning management, etc

One Issue: Abstraction

- Abstraction lets us scale and model all possible runs
  - But it also introduces conservatism
  - *-sensitivities attempt to deal with this
    - * = flow-, context-, path-, field-, etc
  - But they are never enough
- Static analysis abstraction ≠ developer abstraction
  - Because the developer didn’t have them in mind

Symbolic Execution [King, CACM 1976]

- Testing works
  - But, each test only explores one possible execution
    - assert(f(3) == 5)
  - We hope test cases generalize, but no guarantees
- Symbolic execution generalizes testing
  - Allows unknown symbolic variables in evaluation
    - y = α; assert(f(y) == 2^y-1);
  - If execution path depends on unknown, conceptually fork symbolic executor
    - int f(int x) { if (x > 0) then return 2^x - 1; else return 10; }


Symbolic Execution Example

1. int a = α, b = β, c = γ;
2.   // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c) { y = 1; }
9.   z = 2;
10. }
11. assert(x+y+z!=3)

Why Is This Possible?

- There are very powerful SMT/SAT solvers today
  - SMT = Satisfiability Modulo Theories = SAT++
  - Can solve very large instances, very quickly
    - Lets us check assertions, prune infeasible paths
  - We’ve used Z3, STP, and Yices
- Recent success: bug finding
  - Heuristic search through space of possible executions
  - Find really interesting bugs

Sym Exec Can Do Much More

- I think symbolic execution can be used in many other interesting ways
- Next: Symbolic execution as
  - Empirical studies tool
  - Web application security checker
  - Enhancement to abstraction-based static analysis
  - Program synthesis tool
- All of these take advantage of sym exec strengths, and try to avoid drawbacks

Rubyx: Symbolic Execution for Rails
**Programmable Specifications**

- Rubyx provides four special operations
  - `fresh(name)` — returns a fresh symbolic variable
  - `assume(p)` — adds `p` to the path condition
  - `assert(p)` — checks that path condition implies `p`
  - `def invariant() p end` — maintains `p` as an invariant for all objects of the class

- In above, `p` can be any Ruby expression
  - As in Ruby, `false` and `nil` are false, everything else true

- Writing specs just like writing Ruby tests
  - And testing is heavily used in Ruby community

**Generic Specifications**

```
# No XSS: output sent to trusted users has been sanitized
assert (output.trust?) unless (Prin.receiver == Lattice.bot)

# Secrecy: output secrecy level is at most level of receiver
assert (Lattice.leq (output.secrecy?, Prin.receiver))

# No CSRF: messages must be sent from higher to low trust
# levels, and requests that change state must be POST requests
assert (Lattice.leq (Prin.receiver, Prin.sender)) if params[:post]
assert (Prin.sender == Lattice.bot)
assert (params[:post] if (Session.modifier? || Db.modified?))
assert (Lattice.leq (Prin.sender, Prin.receiver)) if params[:post]
assert (Lattice.leq (Prin.id, Prin.sender)) if params[:post]

# Authentication: the sender and receiver must be at least as
# trusted as the logged-in user
assert (Lattice.leq (session[Prin.ld], Prin.sender))
assert (Lattice.leq (session[Prin.id], Prin.sender)) if params[:post]
```

**Example Specification**

Only admin can modify database

```ruby
# send login request
response = Browser.exec(UserController, :login, fresh(:PARAMS))

# assume that login is successful
assume Browser.session[:id]

# send request to update user information
response = Browser.exec(UserController, :update, fresh(:PARAMS))

# assert that logged in user is admin
assert(User.admin?(Prin.sender)) if Db.modified?(User)
```

**Results**

- Analyzed 7 apps, found security vulnerabilities in all
- Common lack of understanding of CSRF, session replay
- Most vulnerabilities could be easily fixed

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pubmgr</td>
<td>√</td>
<td>×(1)</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>(2)</td>
</tr>
<tr>
<td>coffee</td>
<td></td>
<td>×(1)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>depot</td>
<td></td>
<td>×(10)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>chuckslist</td>
<td>✔</td>
<td>×(1)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>(−)</td>
</tr>
<tr>
<td>boxroom</td>
<td>✔</td>
<td>×(5)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>mystic</td>
<td></td>
<td>×(8)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>rtiplan</td>
<td>✔</td>
<td>×(16)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

√ = no vuln. found  × = vuln. found  ? = potential vuln. found
(−) = did not fix  r = replay attack

- No CSRF: messages must be sent from higher to low trust
- No XSS: output sent to trusted users has been sanitized
- Authentication: the sender and receiver must be at least as
  trusted as the logged-in user

Symbolic Execution for Web Apps

• Web applications are “broad” and “shallow”
  ▪ Program designed to quiesce at request-response boundaries
  ▪ Many possible requests and responses, but each request-response short

• Scripting languages are highly dynamic
  ▪ Challenging to analyze with traditional static analysis
  ▪ But symbolic execution just runs the code to see what it does

• Specs are written as programs
  ▪ Should be easy for programmers to write, understand.