Automatic Software Error Finding: Approaches and Tradeoffs

Vaggelis Atlidakis
Software errors: Find them before they find you!

Reviewed approaches

- Test input generation
- Statistical error detection
Papers from three areas

Symbolic Execution & Fuzzing

Transfer Learning & Multitask Learning

Learning-based Program Analysis

24
2
3
5
4
Symbolic execution & fuzzing

Test input generation

Symbolic execution

Fuzzing
Symbolic execution 101

Introduced in EFFIGY[1](1976)

- Execute on symbolic inputs
- Summarize classes of inputs
Classic symbolic vs concolic execution

Symbolic execution

Test input generation

Fuzzing

Classic symbolic execution

Concolic execution
Classic symbolic execution

- Maintain symbolic state
- Fork symbolic execution on branches
- Use solver
  - Branch feasibility

Implemented in EXE [2](2006)

Concolic execution

- Maintain **concrete** & **symbolic** state
- Run concrete execution on taken branches
- Use solver
  - Cover not-taken branches

Implemented in DART [12](2005)
Example

<table>
<thead>
<tr>
<th>void test_me(int x, int y)</th>
<th>Symbolic state</th>
<th>Path constraint</th>
<th>Concrete state</th>
</tr>
</thead>
<tbody>
<tr>
<td>//naughty programmer</td>
<td>Create symbolic variables: $x=a$, $y=b$</td>
<td>$z = a^3$</td>
<td>Random concrete values: $x=1$, $y=1$</td>
</tr>
<tr>
<td>$z = x^2$</td>
<td>$z = a^3$</td>
<td>$a^3 \neq b + 1$</td>
<td></td>
</tr>
<tr>
<td>if ($z == y + 1$)</td>
<td>else</td>
<td>exit(0);</td>
<td></td>
</tr>
<tr>
<td>abort();</td>
<td>exit(0);</td>
<td></td>
<td>$z = 1$</td>
</tr>
<tr>
<td>else exit(0);</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

```c
void test_me(int x, int y){
    //naughty programmer
    z = x*x*x;

    if (z == y + 1)
        abort();
    else
        exit(0);
}
```

<table>
<thead>
<tr>
<th>Symbolic state</th>
<th>Path constraint</th>
<th>Concrete state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create symbolic variables: x=a, y=b</td>
<td>z=a<em>a</em>a != b + 1</td>
<td>Random concrete values: x=1, y=1</td>
</tr>
</tbody>
</table>

Path constraint: (a*a*a != b + 1)
Cannot solve
Example

void test_me(int x, int y) {
    //naughty programmer
    z = x*x*x;
    if (z == y + 1)
        abort();
    else
        exit(0);
}

Symbolic state
Create symbolic variables: x=a, y=b
Path constraint
Random concrete values: x=1, y=1
Concrete state

Path constraint: (a*a*a != b + 1)
Cannot solve
Simplify: a = 1 ↔ 1 != b + 1
**Example**

```c
void test_me(int x, int y){
    //naughty programmer
    z = x*x*x;
    if (z == y + 1)
        abort();
    else
        exit(0);
}
```

<table>
<thead>
<tr>
<th>Symbolic state</th>
<th>Path constraint</th>
<th>Concrete state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create symbolic variables: x=a, y=b</td>
<td>z=a<em>a</em>a</td>
<td>New concrete values: x=1, y=0</td>
</tr>
<tr>
<td>Path constraint: (a<em>a</em>a != b + 1)</td>
<td>Cannot solve</td>
<td></td>
</tr>
<tr>
<td>Simplify: a = 1 ↔ 1 != b + 1</td>
<td>Negate &amp; solve: b = 0</td>
<td></td>
</tr>
</tbody>
</table>

Path constraint: (a*a*a != b + 1)
Cannot solve
Simplify: a = 1 ↔ 1 != b + 1
Negate & solve: b = 0
Example

```
void test_me(int x, int y){
    //naughty programmer
    z = x*x*x;
    if (z == y + 1)
        abort();
    else
        exit(0);
}
```

<table>
<thead>
<tr>
<th>Symbolic state</th>
<th>Path constraint</th>
<th>Concrete state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create symbolic variables: x=a, y=b</td>
<td>z=a<em>a</em>a</td>
<td>New concrete variables: x=1, y=0</td>
</tr>
<tr>
<td>a<em>a</em>a == b + 1</td>
<td>z=1</td>
<td></td>
</tr>
</tbody>
</table>
void test_me(int x, int y) {
    //naughty programmer
    z = syscall(x)
    if (z == y + 1)
        abort();
    else
        exit(0);
}

Symbolic state
Create symbolic variables: x=a, y=b
Path constraint
z=a*a*a
Concrete state
New concrete variables: x=1, y=0
a*a*a == b + 1
z=1
Classic symbolic execution

General limitations

● Handling complex constraints
● Environment problem

➢ Path explosion

Getting stuck here

Entire computation tree
Concolic execution

What do we gain?

● Executions run to completion
➢ Path explosion still a problem
Comparative view

<table>
<thead>
<tr>
<th>Coverage</th>
<th>Number of COREUTILS tools</th>
<th>Avg. #ELOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>16</td>
<td>3307</td>
</tr>
<tr>
<td>90-100%</td>
<td>38</td>
<td>3958</td>
</tr>
<tr>
<td>80-90%</td>
<td>22</td>
<td>5013</td>
</tr>
</tbody>
</table>

COREUTILS tools statement coverage KLEE [3]

<table>
<thead>
<tr>
<th>App</th>
<th>Mean number of instructions</th>
<th>#Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Media</td>
<td>54M</td>
<td>2,266</td>
</tr>
<tr>
<td>Office</td>
<td>923M</td>
<td>3,008</td>
</tr>
</tbody>
</table>

Statistics from SAGE[19]
Classic symbolic vs concolic execution

<table>
<thead>
<tr>
<th>System</th>
<th>Type</th>
<th>What’s new?</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXE[2]</td>
<td>Symbolic</td>
<td>Pioneer symbolic execution engine</td>
</tr>
<tr>
<td>UC-KLEE[9]</td>
<td>Symbolic</td>
<td>Checks individual functions</td>
</tr>
<tr>
<td>CLOUD9[7]</td>
<td>Symbolic</td>
<td>Parallelization of symbolic execution</td>
</tr>
<tr>
<td>DART[12]</td>
<td>Concolic</td>
<td>Pioneer concolic execution engine</td>
</tr>
<tr>
<td>CUTE[13]</td>
<td>Concolic</td>
<td>Adds symbolic with pointers</td>
</tr>
<tr>
<td>PEX[5]</td>
<td>Concolic</td>
<td>Concolic execution in .NET</td>
</tr>
<tr>
<td>SAGE[19]</td>
<td>Concolic</td>
<td>Generational search on deep paths</td>
</tr>
<tr>
<td>CREST[15]</td>
<td>Hybrid</td>
<td>Concolic exec. &amp; random testing</td>
</tr>
<tr>
<td>VERISOFT[8]</td>
<td>Hybrid</td>
<td>Concolic exec. &amp; state merging</td>
</tr>
<tr>
<td>S2E[6]</td>
<td>Hybrid</td>
<td>Symbolic exec. w/ virtualization</td>
</tr>
</tbody>
</table>
Fuzzing 101

Test input generation

Fuzzing

Symbolic execution

Concolic execution

Classic symbolic execution
Fuzzing 101

Test input generation

Fuzzing

Symbolic execution

Classic symbolic execution

Concolic execution

Reliability of UNIX utilities[18](1990)

- Feed random inputs and monitor for errors
- Easy to implement:
  ```
  $> while true; \
  > do head -n 10 /dev/urandom | a.out; \
  > done
  ```
- Inputs that trigger incorrect behaviour are small fraction
Fuzzing: mutation-based vs grammar-based

Test input generation

Fuzzing

Mutation-based

Grammar-based

Symbolic execution

Classic symbolic execution

Concolic execution
Mutation-based fuzzing

➢ American Fuzzy Lop (AFL)[23]

Key points

- Coverage-guided search
- No assumptions for particular input format
- Hard branches (e.g., magic numbers)
Grammar-based fuzzing

- SPIKE grammar-based fuzzer [24]

Key points

- Use grammar to describe input formats
- Good for structured input formats
- Writing grammar is labour-intensive, manual process
Learning-based fuzzing

- Test input generation
- Fuzzing
  - Symbolic execution
  - Concolic execution
- Learning-based
  - Classic symbolic execution
  - Mutation-based
  - Grammar-based
Learning-based fuzzing

- GLADE: Synthesizing program input grammars [25](2017)

Key points
- Start with an input sample
- Construct increasingly general regular expressions
- Translate to Context Free Grammar
- Learning is slow
Learning-based fuzzing

- **NEUZZ**: Fuzzing with Neural Program Learning [29](2018)

**Key points**
- Feed input samples and monitor taken/non-taken branches
- Use training data X,Y learn model for branching behavior
- Use model to perform gradient-guided mutations
- Unclear generalization to “never-taken” paths
## Learning-based fuzzing

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Learns to model</td>
<td>Valid input format</td>
<td>Taken/non-taken branches</td>
<td>Valid input format</td>
<td>High reward mutation policy</td>
<td>Valid input format</td>
</tr>
<tr>
<td>Mutations</td>
<td>Use grammar</td>
<td>Use model's gradients</td>
<td>Use grammar and AFL</td>
<td>Use learnt policy</td>
<td>Use model's predictions</td>
</tr>
<tr>
<td>Strength</td>
<td>Fully blackbox</td>
<td>Gradient-guided mutations</td>
<td>Semantic validity of test cases</td>
<td>End-to-end RL formulation</td>
<td>Location-specific mutation probabilities</td>
</tr>
<tr>
<td>Weakness</td>
<td>Learning realistic grammars slow</td>
<td>Unclear generalization to unseen behaviors</td>
<td>Used a huge collection of input samples</td>
<td>Unclear quality of RL policy</td>
<td>Unclear benefit (production-optimized initial seeds)</td>
</tr>
</tbody>
</table>
Dynamic program analysis

- Test input generation
- Symbolic execution
  - Classic symbolic execution
  - Concolic execution
- Fuzzing
  - Mutation-based
  - Grammar-based
- Learning-based
Statistical error detection

- Correctness patterns
- Quality metrics
Correctness patterns

Statistical error detection

Quality metrics
Correctness patterns

DeepBugs [31](2017)

```javascript
var p = new Promise();
if (promises == null || promises.len == 0) {
    p.done(error, result) ?
} else {
    promises[0](error, result).then(function(res, err))
    p.done(res, err); ?
}
```

Key points

- Inconsistent but...which is correct?
- Most code is (hopefully) correct
- Perform transformations to create incorrect samples

- Need to come up with language-specific checkers
Quality metrics

- Statistical error detection
- Correctness patterns
- Quality metrics
Quality metrics

➢ Transfer Defect Learning [34](2013)

**Key points**

- Code quality metrics of known defects
- Predict if new files look defective
- General metrics ↔ reusable across new targets

➢ File-level reports
## Comparative view

<table>
<thead>
<tr>
<th>System</th>
<th>Proxy for error detection</th>
<th>Source-target</th>
<th>Transfer learning type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeepBugs [31]</td>
<td>Correctness patterns</td>
<td>Same</td>
<td>N/A</td>
</tr>
<tr>
<td>Bugs as Deviant Behaviour [30]</td>
<td>Correctness patterns</td>
<td>Same</td>
<td>N/A</td>
</tr>
<tr>
<td>Naturalness [32]</td>
<td>Quality metrics</td>
<td>Same</td>
<td>N/A</td>
</tr>
<tr>
<td>TCA+ [34]</td>
<td>Quality metrics</td>
<td>Different</td>
<td>Domain adaptation</td>
</tr>
<tr>
<td>Semi-supervised Defect Prediction [33]</td>
<td>Quality metrics</td>
<td>Different</td>
<td>Inductive transfer learning</td>
</tr>
</tbody>
</table>
Statistical error detection

Automatic software error finding

Test input generation

Symbolic execution

Concolic execution

Mutation-based

Grammar-based

Learning-based

Fuzzing

Correctness patterns

Quality metrics

Statistical error detection
Automatic software error finding

Test input generation
- Symbolic execution
  - Classic symbolic execution
  - Concolic execution
- Concolic execution
  - Mutation-based
  - Grammar-based
- Fuzzing
- Learning-based
- Statistical error detection
  - Correctness patterns
  - Quality metrics
Test Input Generation Using Symbolic Execution & Fuzzing

Test Input Generation using Symbolic Execution & Fuzzing AND Learning-based Fuzzing (dynamic)


Learning-based Program Analysis (static)


Learning-based Program Analysis (static) AND Transfer Learning


Transfer Learning


Software Model Checking (not discussed in this presentation)