StatSym: Vulnerable Path Discovery through Statistics-Guided Symbolic Execution

Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Tian Lan and Guru Venkataramani

Department of Electrical and Computer Engineering
The George Washington University

DSN 2017, Denver, Colorado
Outline

- Background
- Motivation
- StatSym Design
- StatSym Evaluation
- Case Study and Sensitivity Study
- Conclusion
Background

▪ Software applications are growing rapidly
  ➤ Diversified application domains

  ➤ Tremendous complexity in code base

Many man-months                  Big SLOCs
Software vulnerabilities are common

- 5~10 bug per 1000 lines of code
- Number of Vulnerabilities increased 39% in the past 5 years[^1]

Security breaches could be costly


Software vulnerability diagnosis is critical

[^1]: FLEXERA, Vulnerability Review 2016, [http://resources.flexerasoftware.com](http://resources.flexerasoftware.com)
Vulnerable Path Identification is difficult

Often involve substantial amount of manual efforts
Vulnerable Path Discovery Techniques

- **Statistical methods (statistical debugging)**
  - E.g., Xu et al., DSN’16; Liblit et al., ICSE’09
  - Fast
  - Maybe inaccurate

- **Formal method (symbolic execution)**
  - E.g., KLEE, OSDI’08; S^2E ASPLOS’11
  - Complete and accurate
  - Non-scalable due to state explosion

- **Combining statistical and formal method**
  - Statistics-guided symbolic execution
Motivational Example

void vulnerable(int a)
{
    //...
    if (a >= 3)
    {
        vulnerability;
    }
}

void main(int x)
{
    if (x >= 1000 || x < 0)
    {
        ...
    }
    else
    {
        int i = 0;
        while (i < x)
        {
            vulnerable(i);
            i++;
        }
    }
    ...
}

Pure Symbolic Execution:
Huge space has to be explored
Motivational Example-Cont.

```c
void vulnerable(int a)
{
    //...
    if( a >=3 )
        vulnerability;
}

void main(int x) {
    if(x>=1000 || x<0){
        ...
    }
    else{
        int i = 0;
        while(i<x){
            vulnerable(i);
            i++;
        }
    }
}
```

Statistical information:
Path: 8 -> 14 -> 4
Condition: x>=3 at line 14

Symbolic execution with statistical guidance:
Search space is considerable pruned
Observations

- **Explosion of program states in symbolic execution**
  - Exhaustive search of all paths (branches, loops, functions)

- **Runtime statistics can guide symbolic execution**
  - Statistical paths trim off unnecessary sub-tree of state
  - Predicates further reduce uninteresting search iterations
StatSym: Statistics-guided Symbolic Execution

- **target program**
- Sampled logs → Predicate construction & ranking
- statistics-guided symbolic execution → Candidate Path Construction
- exact paths and constraints

**Formal**

**Statistical**
StatSym Design – Statistical Analysis

- **Predicate construction**
  - Construct predicate based on divergence between the value ranges for variable \( a \), faulty run value set \( F \), correct run value set \( C \).
  - Find a constraint \( x \) (predicate) that best separates the two sets.

- **Predicate Ranking**
  - Give each predicate a score based on how \( x \) can distinguish \( F \) and \( C \).
  - Score of predicate: \( s = |\mathbb{P}(x|C) - \mathbb{P}(x|F)|. \)
  - \( P(x|C) \) and \( P(x|F) \): probabilities of predicate.

- **Candidate Path Identification**
  - Traverse locations with highly ranked predicates.
  - Generate *candidate paths* by connecting the nodes.
  - Modeled as *graph covering problem*.
StatSym Design – Guided Symbolic Execution

Check predicate P1 for each subpath

Check predicate P2 for each subpath

Check predicate P3 for each subpath
Statistics-guided Symbolic Execution: heuristics

- High priority for paths that covers current node
- Set a threshold for number of diverted hops from current node
StatSym Implementation

Statistical Analysis Module

- Candidate Path Constructor
- Predicate Manager
- Program Monitor
  - Fjalar

Symbolic Execution Module

- KLEE Executor
- StatSym State Manager
- KLEE State Manager
- StatSym Scheduler
  - KLEE Scheduler
Experimental Setup

- **Benchmark Selection**
  - Four application from different application domains
    - *polymorph*, utility for file name conversion, *BugBench*
    - *Grep*, plain-text searching tool, from *STONESOUP*
    - *Ctree*, GNU tool for file system hierarchy display, *STONESOUP*
    - *Thttpd*, open source web server

- **Log Collection and Sampling**
  - Select a number of locations to instrument
  - Use a ranges of sampling rates

- **System Setup**
  - 4-core Intel Xeon E5405
  - 12GB DRAM
# Evaluation (StatSym vs. KLEE)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>KLEE w/ StatSym</th>
<th>Pure Sym. Exec. w/ KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#paths</td>
<td>time (sec)</td>
</tr>
<tr>
<td>polymorph</td>
<td>63</td>
<td>214.6</td>
</tr>
<tr>
<td>CTree</td>
<td>112</td>
<td>45.6</td>
</tr>
<tr>
<td>thttpd</td>
<td>5168</td>
<td>1691.0</td>
</tr>
<tr>
<td>Grep</td>
<td>11462</td>
<td>563.0</td>
</tr>
</tbody>
</table>

KLEE failed for 3 out of the four applications. For smallest application, polymorph, StatSym speeds up KLEE by 15X.
## StatSym Execution Time Breakdown

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Statistical Analysis Module</th>
<th>Symbolic Execution Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>polymorph</td>
<td>1.9</td>
<td>180.6</td>
</tr>
<tr>
<td>CTree</td>
<td>58.4</td>
<td>1.6</td>
</tr>
<tr>
<td>thttpd</td>
<td>561.2</td>
<td>247</td>
</tr>
<tr>
<td>Grep</td>
<td>661.4</td>
<td>37.7</td>
</tr>
</tbody>
</table>

**Time breakdown when sampling rate is 100%**.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Statistical Analysis Module</th>
<th>Symbolic Execution Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>polymorph</td>
<td>1.6</td>
<td>213</td>
</tr>
<tr>
<td>CTree</td>
<td>43.2</td>
<td>2.4</td>
</tr>
<tr>
<td>thttpd</td>
<td>428</td>
<td>1263</td>
</tr>
<tr>
<td>Grep</td>
<td>518.7</td>
<td>44.3</td>
</tr>
</tbody>
</table>

**Time breakdown when sampling rate is 30%**.
Case Study - Polymorph

**Instrumented Locations:**
- L1: grok_commandLine():leave
- L2: convert_fileName():enter
- L3: is_fileHidden():leave
- L4: does_nameHaveUppers():enter
- L5: does_newnameExist():leave
- L6: grok_commandLine():enter
- L7: convert_fileName():leave
- L8: main():enter
- L9: does_newnameExist():enter
- L10: main():leave
- L11: is_fileHidden():enter
- L12: does_nameHaveUppers():leave

**Instrumented Variables:**
- GLOBAL: target, wd, hidden, track, clean, init_file, hidden_file
- FUNCPARAM: argc, original, suspect

<table>
<thead>
<tr>
<th>No.</th>
<th>Predicate</th>
<th>Loc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>len(suspect_FUNCPARAM) &gt; 536.5</td>
<td>L9</td>
</tr>
<tr>
<td>P2</td>
<td>len(original_FUNCPARAM) &gt; 518.0</td>
<td>L2</td>
</tr>
<tr>
<td>P3</td>
<td>len(suspect_FUNCPARAM) &gt; 535.0</td>
<td>L12</td>
</tr>
<tr>
<td>P4</td>
<td>len(suspect_FUNCPARAM) &gt; 517.5</td>
<td>L3</td>
</tr>
<tr>
<td>P5</td>
<td>len(suspect_FUNCPARAM) &gt; 526.0</td>
<td>L2</td>
</tr>
<tr>
<td>P6</td>
<td>len(suspect_FUNCPARAM) &gt; 497.5</td>
<td>L5</td>
</tr>
<tr>
<td>P7</td>
<td>track_GLOBAL &lt; -infinity</td>
<td>L7</td>
</tr>
<tr>
<td>P8</td>
<td>wd_GLOBAL &lt; -infinity</td>
<td>L7</td>
</tr>
<tr>
<td>P9</td>
<td>track_GLOBAL &lt; -infinity</td>
<td>L10</td>
</tr>
<tr>
<td>P10</td>
<td>clean_GLOBAL &lt; -infinity</td>
<td>L10</td>
</tr>
</tbody>
</table>

*Instrumentation List and Predicate Lists*
Case Study- Polymorph

- About 2s for generating candidate paths
- About 180s for symbolic executing the first path
  - 63 paths explored by StatSym vs. 8368 paths explored by KLEE
For Polymorph, symbolic execution time dominates the total time. CTree, Statistics analysis time dominates the total time. Overall, StatSym scales well with sample rates.
Conclusion

- We propose StatSym, a framework for automatic vulnerable path discovery.

- Our approach harnesses the scalability of statistical analysis and the rigorousness of symbolic execution.

- We evaluate StatSym using real-world applications.
  - StatSym achieves about 15X speedup.
Acknowledgements

Email: albertyao@gwu.edu
Evaluation - Candidate Path Statistics

Maximum, average and minimum candidate path lengths
Full Candidate Path in thttpd
Statistics-guided Symbolic Execution: heuristics

- High priority for paths that covers current node
- Set a threshold for number of diverted hops from current node