Week 13A: Fuzzing Science

Stefan Nagy
University of Utah
How are semester projects going?

Making progress?  Stuck?
## Schedule

### Part 4: New Frontiers in Fuzzing

<table>
<thead>
<tr>
<th>Tuesday Meeting</th>
<th>Thursday Meeting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nov. 15</td>
<td>Nov. 17</td>
</tr>
<tr>
<td><strong>Fuzzing Science</strong></td>
<td><strong>Fun Targets: Hardware</strong></td>
</tr>
<tr>
<td>- Reading: <em>FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing</em></td>
<td>- Reading: <em>Fuzzing Hardware Like Software</em></td>
</tr>
<tr>
<td>Nov. 22</td>
<td>Nov. 24</td>
</tr>
<tr>
<td>No Class (Thanksgiving break)</td>
<td>No Class (Thanksgiving break)</td>
</tr>
<tr>
<td>Nov. 29</td>
<td>Dec. 1</td>
</tr>
<tr>
<td><strong>Fun Targets: Compilers</strong> (guest lecture by John Regehr)</td>
<td><strong>Real-world Fuzzing</strong></td>
</tr>
<tr>
<td>- Reading: <em>Finding and Understanding Bugs in C Compilers</em></td>
<td>- Reading: <em>An Empirical Study of OSS-Fuzz Bugs</em></td>
</tr>
<tr>
<td>Dec. 6</td>
<td>Dec. 8</td>
</tr>
<tr>
<td><strong>Team Project presentations</strong></td>
<td><strong>Team Project presentations</strong></td>
</tr>
</tbody>
</table>
Questions?
Fuzzing Science
Why evaluate fuzzers?

- **Advance science**
  - “I must publish to graduate”

- **Validate your technique**
  - “My fix really does work!”

- **Convince others your fuzzer is best**
  - “I made the best fuzzer for Microswat Superclick!”
How should fuzzers be evaluated?

- Pick a few benchmarks
- **Compare** against AFL
- Run a few trials
- Compute **average coverage**
How should fuzzers be evaluated?

- Pick a few **benchmarks**
- **Compare** against AFL
- Run a few **trials**
- Compute **average coverage**
Fuzzer evaluations must be scientific

Evaluating Fuzz Testing

George Klees, Andrew Ruef, Benji Cooper
University of Texas at Dallas

Shiyi Wei
University of Maryland

Michael Hicks
University of Maryland

ABSTRACT
Fuzz testing has enjoyed great success at discovering security critical bugs in real software. Recently, researchers have devoted significant effort to devising new fuzzing techniques, strategies, and algorithms. Such new ideas are primarily evaluated experimentally so an important question is: What experimental setup is needed to produce trustworthy results? We surveyed the recent research literature and assessed the experimental evaluations carried out by 32 fuzzing papers. We found problems in every evaluation we considered. We then performed our own extensive experimental evaluation using an existing fuzzer. Our results showed that the general problems we found in existing experimental evaluations can indeed translate to actual wrong or misleading assessments. We conclude with some guidelines that we hope will help improve experimental evaluations of fuzz testing algorithms, making reported results more robust.

Why do we think fuzzers work? While inspiration for new ideas may be drawn from mathematical analysis, fuzzers are primarily evaluated experimentally. When a researcher develops a new fuzzer algorithm (call it A), they must empirically demonstrate that it provides an advantage over the status quo. To do this, they must choose:

- a compelling baseline fuzzer B to compare against;
- a sample of target programs—the benchmark suite;
- a performance metric to measure when A and B are run on the benchmark suite; ideally, this is the number of (possibly exploitable) bugs identified by crashing inputs;
- a meaningful set of configuration parameters, e.g., the seed file (or files) to start fuzzing with, and the timeout (i.e., the duration) of a fuzzing run.

An evaluation should also account for the fundamentally random
Fuzzer evaluations must be scientific

Evaluating Fuzz Testing

George Klees, Andrew Ruef,
Benji Cooper
University of Texas at Dallas

Shiyi Wei
University of Texas at Dallas

Michael Hicks
University of Maryland

ABSTRACT
Fuzz testing has enjoyed great success at discovering security critical bugs in real software. Recently, researchers have devoted significant effort to devising new fuzzing techniques, strategies, and algorithms. Such new ideas are primarily evaluated experimentally so an important question is: What experimental setup is needed to produce trustworthy results? We surveyed the recent research literature and assessed the experimental evaluations carried out by 32 fuzzing papers. We found problems in every evaluation we considered. We then performed our own extensive experimental evaluation using an existing fuzzer. Our results showed that the general problems we found in existing experimental evaluations can indeed translate to actual wrong or misleading assessments. We conclude with some guidelines that we hope will help improve experimental evaluations of fuzz testing algorithms, making reported results more robust.

An evaluation should also account for the fundamentally random nature of fuzzing, which means that the order of events matters.

Why do we think fuzzers work? While inspiration for new ideas may be drawn from mathematical analysis, fuzzers are primarily evaluated experimentally. When a researcher develops a new fuzzing algorithm (call it A), they must empirically demonstrate that it provides an advantage over the status quo. To do this, they must choose:

- a compelling baseline fuzzer B to compare against;
- a sample of target programs—the benchmark suite;
- a performance metric to measure when A and B are run against the benchmark suite; ideally, this is the number of (post-crash exploitable) bugs identified by crashing inputs;
- a meaningful set of configuration parameters, e.g., the number of file (or files) to start fuzzing with, and the timeout (i.e., duration) of a fuzzing run.

Hicks Wins NSA’s Best Scientific Cybersecurity Paper Award

NSA · cybersecurity · faculty · research · awards

Published October 2, 2019

Michael Hicks, a professor of computer science, helped lead a team of researchers to victory in the National Security Agency’s (NSA) 7th Annual Best Scientific Cybersecurity Paper Competition.
Benchmark Selection
Benchmark Selection

- **Size matters**
  - File size
    - Megabytes
  - Complexity
    - Basic blocks
    - Proxy for # of paths
Benchmark Selection

- **Size matters**
  - File size
    - Megabytes
  - Complexity
    - Basic blocks
    - Proxy for # of paths

---

Does execution mechanism speed always matter?

- Profile average time spent on **target program** vs. **execution mechanism**

<table>
<thead>
<tr>
<th>Avg. Target Time / input</th>
<th>Avg. Execution Time / input</th>
<th>Prop. spent on Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 ms</td>
<td>1–10 ms</td>
<td>33.3–83.3%</td>
</tr>
<tr>
<td>300 ms</td>
<td>1–10 ms</td>
<td>0.0–3.2%</td>
</tr>
</tbody>
</table>

- **Short-running** test cases = execution speed matters **more**
- **Long-running** test cases = execution matters **less** (and **coverage tracing** matters more)

- As usual, this phenomena is **target-dependent**
Benchmark Selection

- **Size matters**
  - File size
    - Megabytes
  - Complexity
    - Basic blocks
    - Proxy for # of paths

- **Results tell all**
  - Ideally good on all sizes

---

**Does execution mechanism speed always matter?**

- Profile average time spent on **target program** vs. **execution mechanism**

<table>
<thead>
<tr>
<th>Avg. Target Time / input</th>
<th>Avg. Execution Time / input</th>
<th>Prop. spent on Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 ms</td>
<td>1—10 ms</td>
<td>33.3—83.3%</td>
</tr>
<tr>
<td>300 ms</td>
<td>1—10 ms</td>
<td>0.0—3.2%</td>
</tr>
</tbody>
</table>

- **Short-running** test cases = execution speed matters more
- **Long-running** test cases = execution matters less (and coverage tracing matters more)

- As usual, this phenomena is **target-dependent**
Benchmark Selection

- Maximize variety
  - Program type
    - Image parser
    - Document reader
    - Audio file converter
  - Program input format
    - JPEG, GIF, EXIF
    - PDF, DOC, XML
    - MP3, WAV, OGG
  - Parent library / application
    - ImageMagick
    - Binutils
    - RARLab
Benchmark Selection

- Cardinal sins of benchmark selection
  - Fuzzing programs of a single type, format
    - E.g., PDF parsers
  - Fuzzing programs from a single package
    - E.g., Binutils, Coreutils
    - Happens far too often

- Results should be generalizable
  - If not, then explain why
  - If not justified, then reject

- GNU Binutil Command Examples
  - readelf
  - strings
  - nm
  - ar
  - objdump
  - strip
  - objcopy
  - addr2line
  - size
Other Benchmark Selection Sins

- Developing a **binary-only** approach
  - But only evaluating open-source programs
  - Finding closed-source benchmarks is hard!
Other Benchmark Selection Sins

- Developing a **binary-only** approach
  - But only evaluating open-source programs
  - Finding closed-source benchmarks is hard!

<table>
<thead>
<tr>
<th>Application</th>
<th>OS</th>
<th>Binary</th>
<th>Size</th>
<th>Blocks</th>
<th>P*C</th>
<th>Sym</th>
<th>Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1FreeArchiver</td>
<td>L</td>
<td>b1</td>
<td>4.1M</td>
<td>150,138</td>
<td>D</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>B1FreeArchiver</td>
<td>L</td>
<td>b1manager</td>
<td>19.3M</td>
<td>290,628</td>
<td>D</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BinaryNinja</td>
<td></td>
<td>binaryninja</td>
<td>34.4M</td>
<td>998,630</td>
<td>D</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BurnInTest</td>
<td>L</td>
<td>bit_cmd_line</td>
<td>2.6M</td>
<td>73,229</td>
<td>D</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BurnInTest</td>
<td>L</td>
<td>bit_gui</td>
<td>3.4M</td>
<td>107,897</td>
<td>D</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Coherent PDF</td>
<td></td>
<td>smpdf</td>
<td>3.9M</td>
<td>61,204</td>
<td>D</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>IDA Free</td>
<td>L</td>
<td>ida64</td>
<td>4.5M</td>
<td>173,551</td>
<td>I</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>IDA Pro</td>
<td>L</td>
<td>idat64</td>
<td>1.8M</td>
<td>82,869</td>
<td>I</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>LzTurbo</td>
<td>L</td>
<td>lztturbo</td>
<td>314K</td>
<td>13,361</td>
<td>D</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>NConvert</td>
<td></td>
<td>nconvert</td>
<td>2.6M</td>
<td>111,652</td>
<td>D</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>NVIDIA CUDA</td>
<td>L</td>
<td>nvidisasm</td>
<td>19M</td>
<td>46,190</td>
<td>D</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Object2VR</td>
<td>L</td>
<td>object2vr</td>
<td>8.1M</td>
<td>239,089</td>
<td>D</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PNGOUT</td>
<td>L</td>
<td>pngout</td>
<td>89K</td>
<td>4,017</td>
<td>D</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>RARLab</td>
<td>L</td>
<td>rar</td>
<td>566K</td>
<td>25,287</td>
<td>D</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>RARLab</td>
<td>L</td>
<td>unrar</td>
<td>311K</td>
<td>13,384</td>
<td>D</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>RealVNC</td>
<td>L</td>
<td>VNC-Viewer</td>
<td>7.9M</td>
<td>338,581</td>
<td>D</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>VivaDesigner</td>
<td>L</td>
<td>VivaDesigner</td>
<td>28.9M</td>
<td>1,097,993</td>
<td>D</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>VueScan</td>
<td>L</td>
<td>vueiscan</td>
<td>15.4M</td>
<td>396,555</td>
<td>D</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Everything</td>
<td>W</td>
<td>Everything</td>
<td>2.2M</td>
<td>115,980</td>
<td>D</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Imagine</td>
<td>W</td>
<td>Imagine64</td>
<td>15K</td>
<td>99</td>
<td>D</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>NirSoft</td>
<td>W</td>
<td>AppNetworkCounter</td>
<td>122K</td>
<td>4,091</td>
<td>D</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Ocenaudio</td>
<td>W</td>
<td>ocenaudio</td>
<td>6.1M</td>
<td>178,339</td>
<td>D</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>USBDDevice</td>
<td>W</td>
<td>USBDevice</td>
<td>185K</td>
<td>7,367</td>
<td>D</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
Other Benchmark Selection Sins

- Developing a **binary-only** approach
  - But only evaluating open-source programs
  - Finding closed-source benchmarks is hard!

<table>
<thead>
<tr>
<th>Application</th>
<th>OS</th>
<th>Binary</th>
<th>Size</th>
<th>Blocks</th>
<th>P*C</th>
<th>Sym</th>
<th>Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1FreeArchiver</td>
<td>L</td>
<td>b1</td>
<td>4.1M</td>
<td>150,138</td>
<td>D</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>B1FreeArchiver</td>
<td>L</td>
<td>b1manager</td>
<td>19.3M</td>
<td>290,628</td>
<td>D</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>BinaryNinja</td>
<td>L</td>
<td>binaryninja</td>
<td>34.4M</td>
<td>998,630</td>
<td>D</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>BurnInTest</td>
<td>L</td>
<td>bit_cmd_line</td>
<td>2.6M</td>
<td>73,229</td>
<td>D</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>BurnInTest</td>
<td>L</td>
<td>bit_gui</td>
<td>3.4M</td>
<td>107,897</td>
<td>D</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>Coherent PDF</td>
<td>L</td>
<td>smpdf</td>
<td>3.9M</td>
<td>61,204</td>
<td>D</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>IDA Free</td>
<td>L</td>
<td>ida64</td>
<td>4.5M</td>
<td>173,551</td>
<td>I</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>IDA Pro</td>
<td>L</td>
<td>idat64</td>
<td>1.8M</td>
<td>82,869</td>
<td>I</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>L2Turo</td>
<td>L</td>
<td>lzturbo</td>
<td>314K</td>
<td>13,361</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NConvert</td>
<td>L</td>
<td>nconvert</td>
<td>2.6M</td>
<td>111,652</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVIDIA CUDA</td>
<td>L</td>
<td>nvdisasm</td>
<td>19M</td>
<td>46,190</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object2VR</td>
<td>L</td>
<td>object2vr</td>
<td>8.1M</td>
<td>239,089</td>
<td>D</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>PNGOUT</td>
<td>L</td>
<td>pngout</td>
<td>89K</td>
<td>4,017</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RARLab</td>
<td>L</td>
<td>rar</td>
<td>566K</td>
<td>25,287</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RARLab</td>
<td>L</td>
<td>unrar</td>
<td>311K</td>
<td>13,384</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RealNVC</td>
<td>L</td>
<td>VNC-Viewer</td>
<td>7.9M</td>
<td>338,581</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VivaDesigner</td>
<td>L</td>
<td>VivaDesigner</td>
<td>28.9M</td>
<td>1,097,993</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VueScan</td>
<td>L</td>
<td>vuescan</td>
<td>15.4M</td>
<td>396,555</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Everything</td>
<td>W</td>
<td>Everything</td>
<td>2.2M</td>
<td>115,980</td>
<td>D</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Imagine</td>
<td>W</td>
<td>Imagine64</td>
<td>15K</td>
<td>99</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NirSoft</td>
<td>W</td>
<td>AppNetworkCounter</td>
<td>122K</td>
<td>4,091</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenAudio</td>
<td>W</td>
<td>octenaudio</td>
<td>6.1M</td>
<td>178,339</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USBDevView</td>
<td>W</td>
<td>USBDevview</td>
<td>185K</td>
<td>7,367</td>
<td>D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>ZAFL vs. AFL-Dyninst</th>
<th>ZAFL vs. AFL-QEMU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rel. crash</td>
<td>rel. total</td>
</tr>
<tr>
<td>idat64</td>
<td>1.000</td>
<td>0.789</td>
</tr>
<tr>
<td>nconvert</td>
<td>3.538</td>
<td>0.708</td>
</tr>
<tr>
<td>nvdisasm</td>
<td>1.111</td>
<td>0.757</td>
</tr>
<tr>
<td>pngout</td>
<td>0.147</td>
<td>5.842</td>
</tr>
<tr>
<td>unrar</td>
<td>0.838</td>
<td>6.112</td>
</tr>
</tbody>
</table>

| Mean Rel. Increase | +55% | +16% | +326% | +38% | +52% | +20% |
| Mean MWU Score    | 0.036 | 0.041 | 0.009 | 0.082 | 0.021 | 0.045 |
Relying on **synthetic benchmark corpora**
- E.g., SPEC2000, LAVA-M
- Often limited in their semantics
  - LAVA-M: only magic-byte bugs
- Many reviewers hate this
  - I am more forgiving
  - Best served as a “preliminary” data point
Competitor Selection
Choosing worthy competitors

■ Many different fuzzers today
  ○ Random fuzzing
  ○ Grammar fuzzing
  ○ Token-level fuzzing
  ○ Rare branch targeting
  ○ Invariant-guided fuzzing
  ○ Sub-instruction profiling
  ○ ...
  ○ Which should you choose?
Choose the state of the art!

- Pick the best **conventional** fuzzers
  - E.g., AFL, AFL++, libFuzzer

- Include the **latest and greatest** fuzzers
  - Are you building a better grammar fuzzer?
    - Compare to other grammar fuzzers!
    - E.g., Gramatron, Nautilus
  - Are you building a fast binary instrumenter?
    - Compare other binary instrumenters!
    - E.g., ZAFL, AFL-QEMU, AFL-Dyninst
  - **Up to you to stay up to date on the literature**
Implementation differences matter

- **Build your fuzzer off a common platform**
  - AFL is today’s most popular platform
    - Most fuzzers derived from AFL
  - Every change matters
    - E.g., speed, queue strategy, mutation

- **Leave core fuzzer design as a control**
Ablation Studies

- Did you implement a ton of new features?
  - Lots of levers to pull, knobs to twist
  - E.g., coverage granularity, execution timeout

- Compare results with & without each
  - Ablation studies make for better science
    - Is an idea the sum of its parts?
    - Or is one feature most critical?
  - Better yet: publish one key idea at a time
Cardinal sins of competitor selection

- **Choosing old, obsolete fuzzers**
  - Contribution sold as better than it is
  - Automatic reject!

- **Omitting relevant state-of-the-art**
  - Usually a major revision
  - Reevaluate with what reviewers want
  - Reviewers need to know what to suggest

- **Throwing five things at the wall**
  - Many of these papers get accepted as-is
  - Bad science; we need ablation studies!
  - Paper must be carefully read and dissected
Experiment Setup
Seed Selection Matters

Source: Evaluating Fuzz Testing
Seed Selection Matters

Source: Evaluating Fuzz Testing
- **Early plateaus** can be misleading
  - Look for **sustained** plateaus

- Likewise, **high coverage early on** can be misleading
  - Want to see **sustained growth** over time
Trial Duration

Figure 4: \(n_m\) with three sampled seeds. At 6 hours: AFLFast is superior to AFL with \(p < 10^{-13}\). At 24 hours: AFL is superior to AFLFast with \(p = 0.000105\).

Source: Evaluating Fuzz Testing
Recommended setup

- **Seeds of varying contents**
  - E.g., empty, well-formed, etc.

- **Trial length of 24+ hours**
  - The bare minimum
  - Longer is better

- **At least 5 trials per benchmark**
  - One trial is not representative
Ensuring fair comparisons

- Maintain same setup across all fuzzers
  - Same seeds, number of trials, duration, etc.
  - If a trial fails, **re-run until all 5 trials** completed

- Begin fuzzers at same starting time

![Diagram showing edges covered over time](image)
Experiment Procedure
Results Processing

What metrics do we value most?
- **Code coverage**
  - Easy to measure
- **Bugs and vulnerabilities** found
  - Hard to measure
- **Zero-day vulnerabilities** found
  - A long time to produce
  - Bad reviewers ask for this

Project-specific metrics
- Results that prove a point or back up a claim
- E.g., queue size, time spent on execution, etc.
Bugs and vulnerabilities

- Finding brand-new bugs is challenging
  - Many common fuzzing targets are well-fuzzed
  - Looks bad to pick random, unknown programs

- Synthetic bug benchmark corpora
  - E.g., Magma, LAVA-M
    - Various caveats (e.g., realism)

- Known bugs in older program versions
  - E.g., fuzzing TCPDump 4.9.1

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Category</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2011-4517</td>
<td>heap overflow</td>
<td>jasper</td>
</tr>
<tr>
<td>GitHub issue #58-1</td>
<td>stack overflow</td>
<td>mjs</td>
</tr>
<tr>
<td>GitHub issue #58-2</td>
<td>stack overflow</td>
<td>mjs</td>
</tr>
<tr>
<td>GitHub issue #58-3</td>
<td>stack overflow</td>
<td>mjs</td>
</tr>
<tr>
<td>GitHub issue #58-4</td>
<td>stack overflow</td>
<td>mjs</td>
</tr>
<tr>
<td>GitHub issue #136</td>
<td>stack overflow</td>
<td>mjs</td>
</tr>
<tr>
<td>Bugzilla #3392519</td>
<td>null pointer deref</td>
<td>nasm</td>
</tr>
<tr>
<td>CVE-2018-8881</td>
<td>heap overflow</td>
<td>nasm</td>
</tr>
<tr>
<td>CVE-2017-17814</td>
<td>use-after-free</td>
<td>nasm</td>
</tr>
<tr>
<td>CVE-2017-10686</td>
<td>use-after-free</td>
<td>nasm</td>
</tr>
<tr>
<td>Bugzilla #3392423</td>
<td>illegal address</td>
<td>nasm</td>
</tr>
<tr>
<td>CVE-2008-5824</td>
<td>heap overflow</td>
<td>sfconvert</td>
</tr>
</tbody>
</table>
Bug-finding metrics

- **Number of bugs found**
  - Proxy for general bug-finding ability
  - Don’t just report AFL’s “unique crashes”—you must deduplicate them!

- **Time-to-exposure on known bugs**
  - Helpful—especially if your focus is on accelerating fuzzing speed

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Location</th>
<th>AFL-Dyninst</th>
<th>AFL-QEMU</th>
<th>ZAFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>heap overflow</td>
<td>nconvert</td>
<td>X</td>
<td>18.3 hrs</td>
<td>12.7 hrs</td>
</tr>
<tr>
<td>stack overflow</td>
<td>unrar</td>
<td>X</td>
<td>12.3 hrs</td>
<td>9.04 hrs</td>
</tr>
<tr>
<td>heap overflow</td>
<td>pngout</td>
<td>12.6 hrs</td>
<td>6.26 hrs</td>
<td>1.93 hrs</td>
</tr>
<tr>
<td>use-after-free</td>
<td>pngout</td>
<td>9.35 hrs</td>
<td>4.67 hrs</td>
<td>1.44 hrs</td>
</tr>
<tr>
<td>heap overread</td>
<td>libida64.so</td>
<td>23.7 hrs</td>
<td>X</td>
<td>2.30 hrs</td>
</tr>
</tbody>
</table>

| ZAFL Mean Rel. Decrease | -660% | -113% |

Table 7: Mean time-to-discovery of closed-source binary bugs found for AFL-Dyninst, AFL-QEMU, and ZAFL over 5×24-hour fuzzing trials. X = bug is not reached in any trials for that instrumenter configuration.
Zero-day Vulnerabilities

- **Requires you to triage and report bugs**
  - You must fuzz the program’s latest version
  - Follow responsible disclosure practices
  - Let developer request a CVE identifier
  - See “Bugs & Triage II” lecture from class

- **“You didn’t find new bugs... REJECT!”**
  - A terrible trend in academic fuzzing
  - Happening less (from what I can tell)
Summary Statistics

- **Are your results statistically significant?**
  - Arithmetic mean doesn’t tell the story
    - Too coarse-grained of a comparison

- **The Mann-Whitney U test**
  - $p$-value above 0.05 = not statistically significant
    - Your 2x improvement doesn’t matter
  - $p$-value less than 0.05 = statistically significant
    - Great job!
  - The gold standard of fuzzing evaluations today
  - Other: Vargha and Delaney’s A-12 test
    - “Magnitude” of an improvement
Statistical Significance

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>2.58e-26</td>
<td>0.0022</td>
<td>6.96e-5</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5.72e-23</td>
<td>0.194</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>5.61e-22</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Statistical Significance

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>2.58e-26</td>
<td>0.0022</td>
<td>6.96e-5</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5.72e-23</td>
<td></td>
<td>0.194</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>5.61e-22</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Questions?