Week 10A: Optimization 1

Stefan Nagy
University of Utah
Announcements

- Coming in **Spring 2023**...
  - The Utah Cybersecurity Club
    - [https://softsec.cs.utah.edu/cyber](https://softsec.cs.utah.edu/cyber)
    - Fill out our recruitment survey!
      - Bonus points if you solve the hidden CTF challenge ;)

---

Stefan Nagy
Reminders

- **Semester Projects are underway!**
  - Harness, fuzz, and triage

- **Final presentations on December 6th**
  - Show off your cool results!

- **Need advice or feedback? Come see me**
  - Best-case outcome: a *top-tier paper* submission
    - Happy to help you work toward this!
Questions?
Fuzzing Faster
Recap: Fuzzing

Trace code coverage, Monitor execution

- New coverage
- Triggers bugs
- No new coverage

Inputs

Program
Recap: Fuzzing

- **Total execs**: 3202
- **Exec speed**: 10.7/sec (slow!)
What affects fuzzing speed?

- **Process execution**
  - Performed on every input

- **Runtime instrumentation**
  - Code coverage tracing

- **Information post-processing**
  - Data structure writing/reading
  - Other essential computation
Why is speed so important?

- **Need to find the bugs before attackers do**
  - Time is money; bug-finders limited by time/resource budgets
  - Race to find and fix before monthly “Patch Tuesday”

- **People’s privacy (and lives) at stake**
  - Nation-state attackers have unlimited budgets
  - They’re in it to win it just as much
Complexity adds overhead

- **Fancy/slow is often less effective than crude/fast**
  - E.g., taint tracking-based fuzzing vs. good ol’ AFL
  - *Academically interesting is not always practical*

<table>
<thead>
<tr>
<th>Applications</th>
<th>Version</th>
<th>AFL</th>
<th>CollAFL- br</th>
<th>Honggfuzz</th>
<th>VUzzer</th>
</tr>
</thead>
<tbody>
<tr>
<td>libbson</td>
<td>1.8.0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>libsndfile</td>
<td>1.0.28</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>libconfuse</td>
<td>3.2.2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libwebm</td>
<td>1.0.0.27</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libxslt</td>
<td>2.4</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>libcapa</td>
<td>0.99beta19</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>liblas</td>
<td>2.4</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libssl</td>
<td>20180901</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libxml</td>
<td>v1.8.2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>libxml2</td>
<td>release-1.10</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>-</td>
<td>21</td>
<td>34</td>
<td>18</td>
<td><strong>6</strong></td>
</tr>
</tbody>
</table>

Table 1: Number of vulnerabilities (accumulated in 5 runs)

Source: GREYONE: Data Flow Sensitive Fuzzing
Pre-execution Optimization
Test Case Minimization

- Test cases get larger as fuzzing continues
  - More program execution = more overhead
  - Need to **cut-out unnecessary execution**

- **Delta debugging**: change, then check
  - Iteratively remove input bytes
  - Check if code coverage changes
    - If coverage changes, undo
    - Like one big jenga game
Corpus Minimization

- **Test case corpus grows as fuzzing continues**
  - Lots of test cases reach new edge, hit count coverage
  - Many test cases have overlapping code coverage
  - Fuzzer will struggle to pick the “best” one

- **Corpus minimization**: condense your corpus
  - I.e., smallest set that covers all edges seen so far
  - **AFL**: also minimize file size and execution time

Trade-offs

- Complicated for **highly-structured inputs**
  - E.g., JPEG images versus ELF executables
  - Byte-level changes won’t work on the latter
  - Grammar-level mutations require more machinery

- Complicated by **code coverage granularity**
  - E.g., edges versus hit counts
  - Finer-grained info is harder to condense
  - **Still an unsolved research problem**
Post-execution Optimization
Storing Information

- Must store information in data structures
  - E.g., bitmaps for code coverage traces
  - E.g., ASTs for dynamically-learned grammars

- Data structure design affects fuzzing speed
  - Memory footprint
  - Cost of reads/writes
Trade-offs

- **Best case:** small enough to fit in L2 cache
  - But, smaller size sacrifices information storage

Source: BigMap: Future-proofing Fuzzers with Efficient Large Maps
Intra-execution Optimization
Program Instrumentation

Compiler

Binary (static)

Binary (dynamic)
Program Instrumentation

- **Open-source: compiler instrumentation**
  - Bake-in instrumentation code at compile-time
  - *Efficient* and *correct*

- **Closed-source: dynamic binary translation**
  - Instrument program as it is executing
  - Generally *correct* but *inefficient*

- **Closed-source: static binary rewriting**
  - Instrument program before it executes
  - Generally *incorrect* but *efficient*
Faster Binary Instrumentation

- **Dynamic binary translation**
  - Primary expense comes from **translation**
    - Performed on **every** piece of code
    - **Re-translate** already seen code
  
  - **Solution**: make already-seen code **cached**
    - Avoid re-translating as much as possible
Faster Binary Instrumentation

- **Our solution (ZAFL):** design static rewriters to match compilers
  - Achieves **compiler-level speeds** for closed-source targets
Faster Binary Instrumentation

- **Our solution (ZAFL):** design static rewriters to match compilers
  - Achieves **compiler-level speeds** for closed-source targets
  - Finds vulnerabilities faster than other binary tracers

<table>
<thead>
<tr>
<th>Vulnerability Type</th>
<th>Executable</th>
<th>Dyninst</th>
<th>QEMU</th>
<th>ZAFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap Overflow</td>
<td>nconvert</td>
<td>Can’t find</td>
<td>18.3 hrs</td>
<td>12.7 hrs</td>
</tr>
<tr>
<td>Heap Overflow</td>
<td>unrar</td>
<td>Can’t find</td>
<td>12.3 hrs</td>
<td>9.04 hrs</td>
</tr>
<tr>
<td>Use-After-Free</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 Overflow</td>
<td>IDA Pro</td>
<td>23.7 hrs</td>
<td>Can’t find</td>
<td>2.30 hrs</td>
</tr>
</tbody>
</table>

ZAFL’s Mean Relative Decrease: -660% -113%
Hardware-assisted Tracing

- **Collect coverage via fast CPU mechanisms**
  - E.g., Intel Processor Trace, ARM Coresight
  - An emerging feature used in binary fuzzing

- **Trade-offs:**
  - Attains speeds similar to compiler instrumentation
  - Only usable (and effective) on specific hardware
    - ARM Coresight is way slower than Intel PT
  - Cannot instrument programs to do other things
    - E.g., sub-instruction profiling
Instrumentation Culling

- **Save overhead by instrumenting less of the program**
  - **Crude approach:** instrument code at random
  - **Smart approach:** instrument leaf nodes of dominator tree
    - *A dominates B iff every path to B first intersects A*
    - Cuts down about 30–50% of basic blocks
**Instrumentation Optimization**

- **Downgrade** from edge to block-based instrumentation
  - Save a few instructions (i.e., from computing edge hashes)
  - Saved for basic blocks with **single predecessors**

```c
cur_location = <COMPILE_TIME_RANDOM>;
Shared_mem [cur_location ⊕ prev_location]++;
prev_location = cur_location >> 1;
```

```c
Shared_mem [PreDeterminedID]++;
```
Why trace every single test case?

- Equivalent to checking **each** straw to find **one** needle
  - Cost adds up from instrumentation’s *instruction footprint*
    - 3–5 additional instructions per basic block
    - More instructions from post-processing coverage
Why trace every single test case?

- **Less than 1%** of all inputs reach new code coverage
  - The other 99.9% are **discarded** right after tracing
  - **Wasted resources!**
Coverage-guided Tracing

- **Idea:** restrict tracing to only when new coverage is guaranteed
  - Guaranteed how? By using interrupts!
Coverage-guided Tracing

- **Idea:** restrict tracing to **only when new coverage is guaranteed**
  - Guaranteed how? By using **interrupts**!

---

**Diagram:**

1. **Basic block interrupt**
   - Hit interrupt: perform full trace & remove all interrupts

2. **Interrupts cleared**
   - A → B → C → D → E → F → G

---

SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

Stefan Nagy

30
Coverage-guided Tracing

- **Idea:** restrict tracing to only when new coverage is **guaranteed**
  - Guaranteed how? By using **interrupts**!

![Diagram showing the concept of coverage-guided tracing with nodes A, B, C, D, E, F, G, and arrows indicating the flow of execution.](image)

- Hit interrupt: perform full trace & remove all interrupts
- Basic block interrupt
- Interrupts cleared
- No interrupt hit = no new coverage
Coverage-guided Tracing

- **Implementation: UnTracer**
  - Averages just **0.3%** overhead
  - Coverage-guided fuzzing at the speed of **black-box** fuzzing
Questions?