Week 7B: Tackling Roadblocks

Stefan Nagy
University of Utah
Reminders

- **Lab 3: Harnessing due tonight**
  - Submit *one report* per group by 11:59PM
  - Include *all group member names*

- **Semester Team Project released today**
  - Details at end of lecture
Reminders

- No class next week (fall break)
Please complete your mid-semester course evals!

- I want your feedback!
  - Help me improve the class

- Due by **October 10th**
  - [https://scf.utah.edu](https://scf.utah.edu)
  - Please please please!
Questions?
Evaluating Harnesses
Recap: What makes a good harness?

- **Speed**
  - Avoid irrelevant, wasteful code (e.g., GUIs)

- **Coverage**
  - Execute interesting, hard-to-reach parts of code
  - Avoid leaving blindspots (hidden bugs)

- **Correctness**
  - Upholds program’s expected behavior
  - Does not incur spurious effects (e.g., FP crashes)
Pay attention to performance

- How is speed changing over time?
  - **Beginning**: usually faster
    - Working through input validity checks
    - Less code executing per input

```plaintext
now trying: interest 32/8
stage execs: 3996/34.4k (11.62%)
total execs: 27.4M
exec speed: 893 / second
```
Pay attention to performance

- How is speed changing over time?
  - **Beginning:** usually faster
    - Working through input validity checks
    - Less code executing per input
  - **Later on:** usually slower
    - Executing more code per input

Fuzzing Time

Fuzzing Speed

Fewer valid inputs

More valid inputs

now trying: interest 32/8
stage execs: 3996/34.4k (11.62%)
total execs: 27.4M
eexec speed: 893 / second
Don’t take speed at face value

- Faster may mean...
  - Successfully omitting irrelevant code
    - E.g., GUI setup routines we don’t care about
    - Especially critical for harnessing binaries
  - Erroneously overlooking necessary code
    - E.g., input parsing routines and/or checks
    - Need to understand what the API expects
Don’t take speed at face value

- **Slower** may mean...
  - More time spent iterating loops
    - Too few iterations can miss some bugs
    - Not every loop should be maximized
    - Still an open research problem
  - Your harness is covering too much
    - Focus testing on specific attack vectors
    - Many harnesses instead of a huge one
Measure and plot your code coverage

- Critical to understanding your harness
  - Changes in **edges covered**
  - Changes in edge **hit counts**
  - Source code **visualizations**

- Useful coverage tools
  - [github.com/mrash/afl-cov](https://github.com/mrash/afl-cov)
  - [github.com/gcovr/gcovr](https://github.com/gcovr/gcovr)
  - [github.com/andreafioraldi/afl-qemu-cov](https://github.com/andreafioraldi/afl-qemu-cov)
  - [github.com/eqv/aflq_fast_cov](https://github.com/eqv/aflq_fast_cov)
  - Python scripting with Matplotlib
What does your code coverage tell you?

- **Edge coverage:**
  - Strictly **increases** with time
    - Ideally increases the whole time
  - Always look at **multiple trials**
    - Studies show at least **5 trials**
  - All fuzzers eventually **plateau**
    - Random mutation only gets so far
    - **Early plateaus** indicate you are stuck
    - Potentially missing critical code

---

=SCHOOL OF COMPUTING  
UNIVERSITY OF UTAH  

Stefan Nagy
What does your code coverage tell you?

- **Hit counts:**
  - *Higher* = more cycle iterations
    - Deeper loop exploration
    - More recursion
  - Examine **relative changes**
    - E.g., comparing two harnesses

### Hit counts:

- **Relative Max Consecutive Iterations Per Loop**
  
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>4.06</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>6.77</td>
<td>1.09</td>
<td>1.0</td>
<td>1.1</td>
<td>16.12</td>
<td>1.0</td>
<td>8.9</td>
<td>3.18</td>
</tr>
<tr>
<td>2</td>
<td>1.07</td>
<td>1.0</td>
<td>3.58</td>
<td>10.75</td>
<td>3.0</td>
<td>1.38</td>
<td>1.68</td>
<td>1.05</td>
</tr>
<tr>
<td>3</td>
<td>2.28</td>
<td>4.38</td>
<td>4.25</td>
<td>10.56</td>
<td>1.1</td>
<td>8.1</td>
<td>10.56</td>
<td>1.33</td>
</tr>
<tr>
<td>4</td>
<td>10.64</td>
<td>6.98</td>
<td>1.49</td>
<td>3.76</td>
<td>2.77</td>
<td>1.07</td>
<td>1.07</td>
<td>1.5</td>
</tr>
<tr>
<td>5</td>
<td>2.27</td>
<td>1.6</td>
<td>1.0</td>
<td>1.0</td>
<td>13.04</td>
<td>2.0</td>
<td>1.0</td>
<td>2.88</td>
</tr>
<tr>
<td>6</td>
<td>1.0</td>
<td>2.25</td>
<td>10.62</td>
<td>5.0</td>
<td>2.1</td>
<td>7.46</td>
<td>1.35</td>
<td>0.53</td>
</tr>
<tr>
<td>7</td>
<td>5.59</td>
<td>3.76</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Ranges:**
  - [1]
  - [2]
  - [3]
  - [4,7]
  - [8,15]
  - [16,31]
  - [32,127]
  - [128+]
What does your code coverage tell you?

- **Source line visualization:**
  - Costs **more time** to generate reports
  - Provides you **more information**
  - Does not support binaries

| Line Coverage $\%$ | Functions $\%$
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0 %</td>
<td>0 / 1</td>
</tr>
<tr>
<td>0.0 %</td>
<td>0 / 120</td>
</tr>
<tr>
<td>0.0 %</td>
<td>0 / 22</td>
</tr>
<tr>
<td>75.3 %</td>
<td>64 / 85</td>
</tr>
<tr>
<td>100.0 %</td>
<td>102 / 102</td>
</tr>
<tr>
<td>89.5 %</td>
<td>17 / 19</td>
</tr>
<tr>
<td>100.0 %</td>
<td>31 / 31</td>
</tr>
<tr>
<td>34.4 %</td>
<td>56 / 163</td>
</tr>
<tr>
<td>100.0 %</td>
<td>51 / 51</td>
</tr>
<tr>
<td>30.4 %</td>
<td>24 / 79</td>
</tr>
<tr>
<td>100.0 %</td>
<td>24 / 24</td>
</tr>
</tbody>
</table>

```c
#include <stdio.h>

int main()
{
    int i, j, rows;

    printf("Enter number of rows: ");
    scanf("%d", &rows);

    for(i=1; i<=rows; ++i)
    {
        for(j=1; j<=i; ++j)
        {
            printf("*");
        }
        printf("\n");
    }

    printf("\n");
    return 0;
}
```
Are you fuzzing for long enough?

- **Early plateaus** can be misleading
  - Look for **sustained** plateaus

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

---

Edges Covered ➤ 2 hr ➤ 10 hr

Fuzzing Time

Stefan Nagy
Is your execution timeout large enough?

- **Timeout**: maximum duration of any execution
  - When exceeded, **terminates execution**
    - ... and marks test case as a “hang”
  - AFL’s default is **very small** (mere milliseconds)
    - AFL prefers short-running test cases
    - Too low of a timeout = excessive hangs
      - Missed code coverage
    - Need to **readjust** for your target

![Diagram](image)
Are plateaus fuzzer-dependent?

- Try different input generation techniques
  - Relying on **random mutation** is not advisable
    - Not good at solving magic bytes
  - Lots of options in the AFL universe
    - Grammars, concolic exec, etc.
    - Other code coverage metrics
    - **No single technique is the best**

Source: https://www.fuzzbench.com/reports/paper/Main%20Experiment/index.html
Evaluate your crashes

- **Replay** all fuzzer-found crashes
  - Use tools like AddressSanitizer, DrMemory, etc.
  - If a test case crashes your harness...
    - It should crash the original program too!

- **Identify** false-positive crashes
  - I.e., crashes that occur only in your harness
    - Indicates you are missing **critical code**
  - Pay attention to **what tools tell you** (e.g., ASAN)
    - Source lines (in your harness or API), etc.

== ASAN: heap-use-after-free on address 0x61900000047f at pc 0x00000040a52c bp 0x7fff9200dbf0 sp 0x7fff9200dbe0
READ size 1 at 0x61900000047f thread T0
  #0 0x40a52b in src/main.cpp:30
  #1 0x40e088 in std_function.h:297
  #2 0x40d605 in std_function.h:687
  #3 0x40b8d5 in src/main.cpp:130
  #4 0x7f9a498ff412 in libc-start.c:308
Leverage available oracles

- A library’s provided **front-end programs**
  - Often are very large applications
    - E.g., objdump for Binutils
    - E.g., bsdtar for libArchive
  - Can serve as a **ground-truth** correct API usage

- **Differential testing**
  - Compare against similar programs
    - E.g., Foxit PDF vs. Adobe Reader
  - Do they spit-out **similar messages**?
    - E.g., “this file is definitely invalid for reason X”
  - **Better yet:** do they crash too?
Harnessing is a trial-and-error art...

Don’t give up!
Collect data, investigate, and refine!
Semester Team Project
Semester Team Project

- **Objective:** uncover new bugs in a real-world program

- Team up in groups of 3 – 4

- Select an “interesting” target program of your choice; e.g.:
  - Popular applications
  - Nintendo emulators
  - Old computer games
  - MacOS Rosetta
  - The SoC’s TA Portal
  - ...
  - GET CREATIVE!

- **Figure out how to fuzz** your target, **find bugs**, and **disclose them**

- **Deliverables:** a report, disclosure of bugs, and open-source your team’s fuzzer
Semester Team Project

Objective: uncover new bugs in a real-world program

Team up in groups of 3 – 4

Select an “interesting” target program of your choice; e.g.:

- Popular applications
- Nintendo emulators
- Old computer games
- MacOS Rosetta
- The SoC’s TA Portal
- ...
- GET CREATIVE!

NOTE: Under no circumstances may you exploit or misuse any bugs that you find (e.g., zero-day vulnerabilities) for unauthorized access or other illegal activity.

Violations of this policy will be referred to Student Conduct.

Figure out how to fuzz your target, find bugs, and disclose them

Deliverables: a report, disclosure of bugs, and open-source your team’s fuzzer
Semester Team Project

- Objective: uncover new bugs in a real-world program

- Team up in groups of 3 – 4

- Select an “interesting” target program of your choice; e.g.:
  - Popular applications
  - Nintendo emulators
  - Old computer games
  - MacOS Rosetta
  - The SoC’s TA Portal
  - GET CREATIVE!

- Figure out how to fuzz your target, find bugs, and disclose them

- Deliverables: a report, disclosure of bugs, and open-source your team’s fuzzer

---

NOTE: Under no circumstances may you exploit or misuse any bugs that you find (e.g., zero-day vulnerabilities) for unauthorized access or other illegal activity.

Violations of this policy will be referred to Student Conduct.

Our goal is to help devs & users, have fun, and learn!
Schedule after Fall Break

- **Tuesday, Oct. 18th:** in-class workday
  - Welcome to bounce ideas off of me
  - Get feedback for your 5-min proposal

- **Thursday, Oct. 20th:** proposal day
  - **Instructions:** a 5-minute presentation that motivates your project
  - **Goal:** practice the art of “the pitch”
    - Get feedback from your peers
    - Follow Heilmeier’s Catechism!

- **Dec. 6th and 8th:** final presentations
  - 15–20 minute slide deck and discussion
  - What you did, and why, and what results

---

**The Heilmeier Catechism**

- What are you trying to do? Articulate objectives using absolutely no jargon.
- How is it done today, and what are the limits of current practice?
- What is new in your approach and why do you think it will be successful?
- Who cares? If you are successful, what difference will it make?
- What are the risks?
- How much will it cost?
- How long will it take?
- What are the mid-term and final “exams” to check for success?
Questions?
Alternative Project Ideas
Idea #1: fuzzing for variability bugs

- **Variability bugs**: bugs that occur only in specific build configs
  - C Preprocessor #IFDEF statements
  - Specific compilers, architectures, and operating systems, etc.

- **Open questions:**
  - How often are these missed by conventional fuzzing (e.g., OSS-Fuzz)?
  - How can fuzzing be adapted to find them (e.g., differential testing)?
  - **Interested? Contact me!**

---

Source: Variability Bugs in Highly-Configurable Systems: A Qualitative Analysis
Idea #2: building patch-tailored fuzzers

- Security patches are often **incomplete** fixes
  - **Deja vulnerabilities**: recurring, **already-patched** security bugs

- **Idea**: find ways to generate **patch-tailored** fuzzers
  - Tailor fuzzer input generation to scrutinize a specific patch
  - Interested? Get in touch with me to get my 14-bug corpus

---

Maddie Stone
@maddiestone

Déjà vu-Inerability: A Year in Review of 0-days Exploited in-the-wild in 2020 🌍🔥

googleprojectzero.blogspot.com/2021/02/deja-v...
Idea #3: fuzzing binary decompilers

- **Binary decompilation** is critical for modern-day security efforts
  - Malware analysis, static security analysis, reverse engineering, etc.

- But binary decompilers are **not great**...
  - On complex C/C++ programs
  - On Rust and Go programs

- Need ways to **automatically fuzz them**
  - **Idea**: differential testing vs. original program
  - **Obstacle**: their output is seldom recompilable
  - **Interested? See Colin and myself!**
Please complete your mid-semester course evals!
Questions?