Week 7: Lecture B

Tackling Roadblocks

Wednesday, February 21, 2024

Recap: Key Dates

- **Feb. 14** — **Final Project released**
- **Feb. 19** — No class (President's Day)
- **Feb. 26** — **Sign up final project team**
- **Feb. 28** — **Lab 3 due**
- **Feb. 28** — **5-minute project proposals**
- **Mar. 04 & 06** — No class (Spring Break)
- **Apr. 17 & 22** — **Final project presentations**

cs.utah.edu/~snagy/courses/cs5963/schedule

Feb. 12 **Harnessing I** (slides) ▶ Readings: **Harnessing Lab released**	Feb. 14 **Harnessing II** (slides) ▶ Readings: **Final Project released** **Triage Lab due by 11:59pm**
Feb. 19 **No Class (President's Day)**	Feb. 21 **Tackling Roadblocks** ▶ Readings:
Feb. 26 **Fuzzing Science** ▶ Readings: **Sign up your final project team by 11:59pm**	Feb. 28 **Proposal Presentations** **Harnessing Lab due by 11:59pm**
Mar. 04 **No Class (Spring Break)**	Mar. 06 **No Class (Spring Break)**

Recap: Lab 3 Overview

- **Assignment:** write your own **AFL-friendly** harness for libArchive
  - Read its documentation in: https://linux.die.net/man/3/libarchive
  - https://github.com/google/oss-fuzz/blob/master/projects/libarchive/libarchive_fuzzer.cc
- **Create a harness that reads data from files**
  - What functions did you try?
  - What worked and what didn't?
- **Deliverable:** a **1–3 page report** detailing your findings
  - Feel free to make it your own (e.g., pictures, text, etc.)
  - Submit your harness code in your report
  - **Free to team up (max 3 students per group)**
  - **Submit one report per group**
- **Linux environments are recommended**
  - Use a VM if you don't have one!

Recap: Lab 3 Tips

- **Read libArchive's documentation and get inspiration from others' code**
  - Understand the libArchive manpages
  - Look at how others (e.g., non-fuzzing projects) use its API
- **Validate your results**
  - Measure code coverage of the libArchive codebase
  - Look for increasing code coverage over time
- **Deadline:** Wednesday, February 28th by 11:59PM
  - Group assignment (**up to 3 members**)
  - Look for teammates in-class and on Piazza
  - See cs.utah.edu/~snagy/courses/cs5963/assignments.html

Questions?

Recap: Semester Final Project

- Objective: **uncover new bugs in a real-world program**
- Team up in groups of **1 – 4**
- Select an “interesting” target program of your choice; e.g.:
  - Popular applications
  - Nintendo emulators
  - Old computer games
  - MacOS Rosetta
  - **GET CREATIVE!**
- **Figure out how to fuzz** your target, **find bugs**, and **responsibly disclose them**
- **Deliverables:** a report, disclosure of bugs, and open-source your team’s fuzzer

Recap: Semester Final Project

- Objective: u
- Team up in
- Select an "i
  - Popular
  - Nintend
  - Old computer games
  - MacOS Rosetta
  - GET CRE
- **Figure out h** ... **e them**
- **Deliverables** ... n's fuzzer

5-minute project **proposal** on Feb. 28

Final presentations at semester's end

You have full creative liberty—get creative and **fuzz something fun**!

Recap: Semester Final Project

- Details also now available on course website **Assignments** page:

Final Project (collected via Canvas)

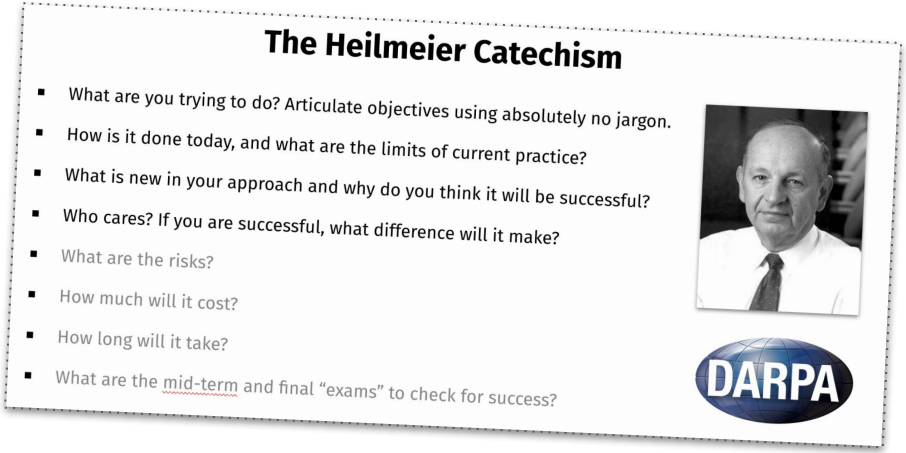
Instructions: Using your skills from Labs 1–3, team up in groups of **no more than four students** to hunt down bugs in a **real-world application** of your choice! Upon selecting a target application, your team will need to figure out how to (1) harness it, (2) fuzz it, and (3) triage any discovered bugs. You may select any target you like (e.g., software APIs, video games, emulators), provided that it has *not* been fuzzed before—or has demonstrably not yet been fuzzed *effectively*.

Halfway through the semester, your team will present a **5-minute project proposal** to the class outlining your chosen target, your proposed approach, and the significance of your work. At the semester's end, you will prepare and deliver a **15-minute final presentation** alongside a **final report** outlining your ultimate approach, findings, and any discovered bugs.

Heilmeier's Catechism will serve as the high-level rubric for your proposal, presentation, and report—so be ready to explain *why* your project idea matters! But most importantly, **get creative and have fun**, and report any bugs you find along the way!

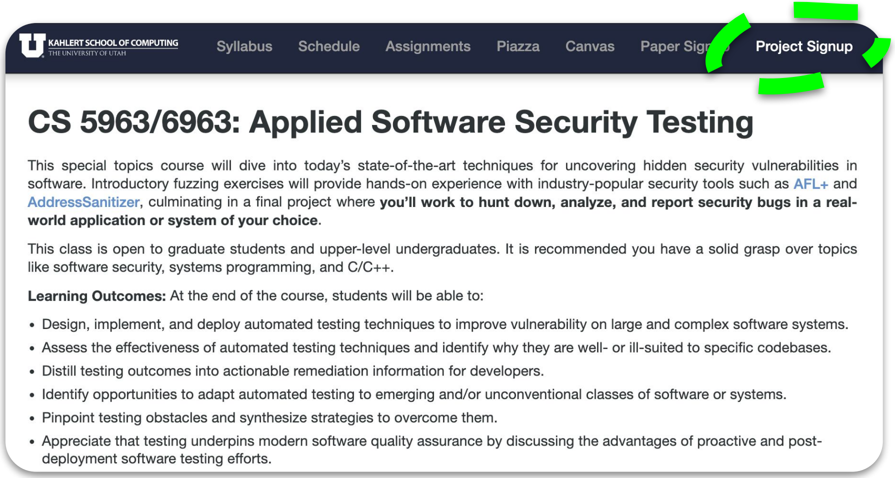
Recap: Project Schedule

- **Monday, Feb 26th:** team signup due
- **Wednesday, Feb. 28th:** 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!**
- **Mar. 27th:** in-class project workday
- **Apr. 17th & 22nd:** final presentations
  - 15–20 minute slide deck and discussion
  - What you did, and why, and what results



Project Team Signup

- **Signup sheet** available on course website (must use **UofU gcloud** account)
  - Fill-in your **project title** and **teammate names** by **11:59PM on Monday, February 26th**



KAHLERT SCHOOL OF COMPUTING
THE UNIVERSITY OF UTAH

Syllabus | Schedule | Assignments | Piazza | Canvas | Paper Signup | Project Signup

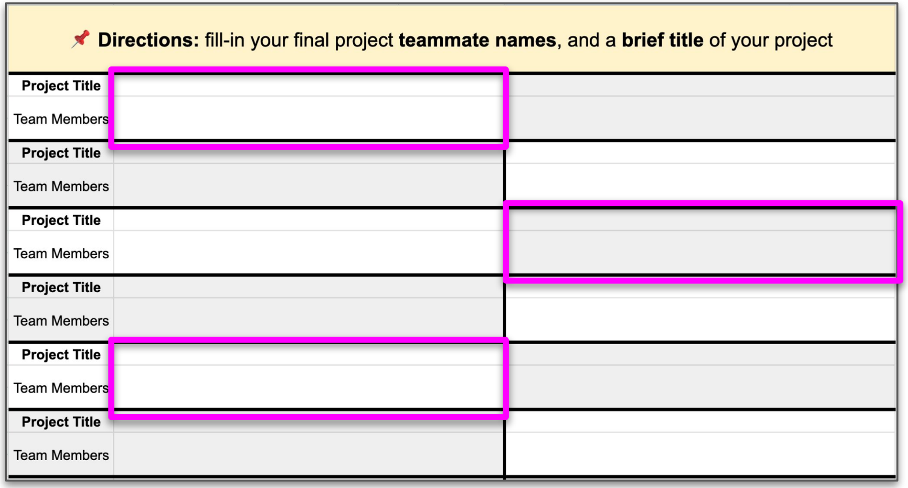
CS 5963/6963: Applied Software Security Testing

This special topics course will dive into today's state-of-the-art techniques for uncovering hidden security vulnerabilities in software. Introductory fuzzing exercises will provide hands-on experience with industry-popular security tools such as AFL+ and AddressSanitizer, culminating in a final project where **you'll work to hunt down, analyze, and report security bugs in a real-world application or system of your choice**.

This class is open to graduate students and upper-level undergraduates. It is recommended you have a solid grasp over topics like software security, systems programming, and C/C++.

Learning Outcomes: At the end of the course, students will be able to:

- Design, implement, and deploy automated testing techniques to improve vulnerability on large and complex software systems.
- Assess the effectiveness of automated testing techniques and identify why they are well- or ill-suited to specific codebases.
- Distill testing outcomes into actionable remediation information for developers.
- Identify opportunities to adapt automated testing to emerging and/or unconventional classes of software or systems.
- Pinpoint testing obstacles and synthesize strategies to overcome them.
- Appreciate that testing underpins modern software quality assurance by discussing the advantages of proactive and post-deployment software testing efforts.



📌 **Directions:** fill-in your final project **teammate names**, and a **brief title** of your project

Project Title		
Team Members		
Project Title		
Team Members		
Project Title		
Team Members		
Project Title		
Team Members		
Project Title		
Team Members		
Project Title		
Team Members		

Questions?

Evaluating Harnesses

Recap: What makes a good harness?

- ???

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)

Line	Branch	Exec	Source
1			`// example.cpp`
2			
3		1	`int foo(int param)`
4			`{`
5	✗✓	1	`    if (param)`
6			`    {`
7			`        return 1;`
8			`    }`
9			`    else`
10			`    {`
11		1	`        return 0;`
12			`    }`
13			`}`

Pay attention to performance...

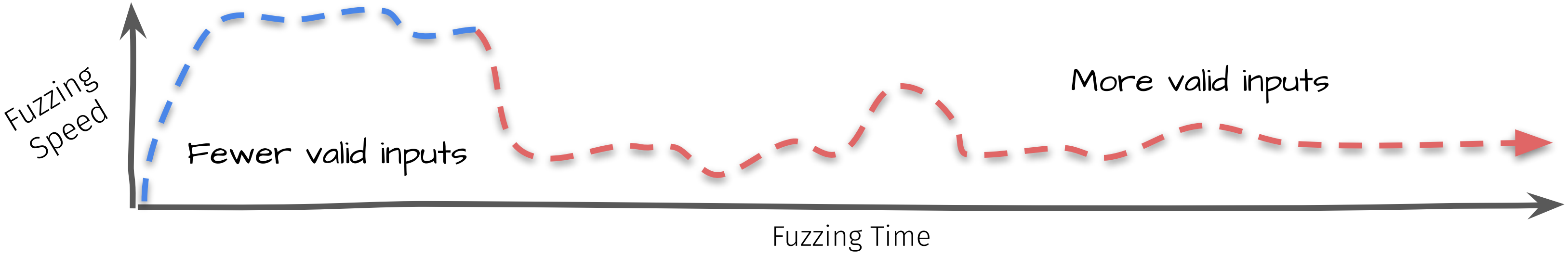
- How is speed changing over time?
  - **Beginning:** usually faster
    - Working through input validity checks
    - Less code executing per input

now trying :	interest 32/8
stage execs :	3996/34.4k (11.62%)
total execs :	27.4M
exec speed :	893 / second

Pay attention to performance...

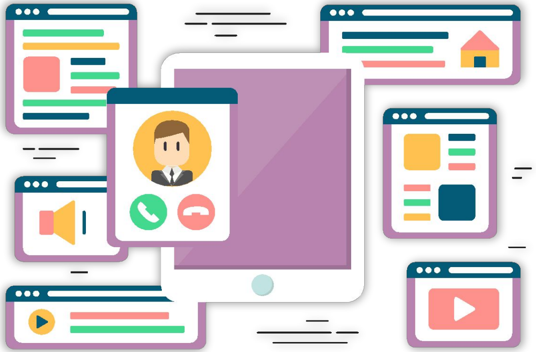
- How is speed changing over time?
  - **Beginning:** usually faster
    - Working through input validity checks
    - Less code executing per input
  - **Later on:** usually slower
    - Executing more code per input

now trying :	interest 32/8
stage execs :	3996/34.4k (11.62%)
total execs :	27.4M
exec speed :	893 / second



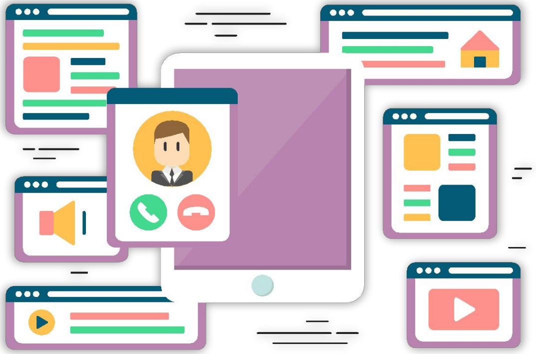
Don’t take speed at face value!

- **Faster** may mean…
  - **???**



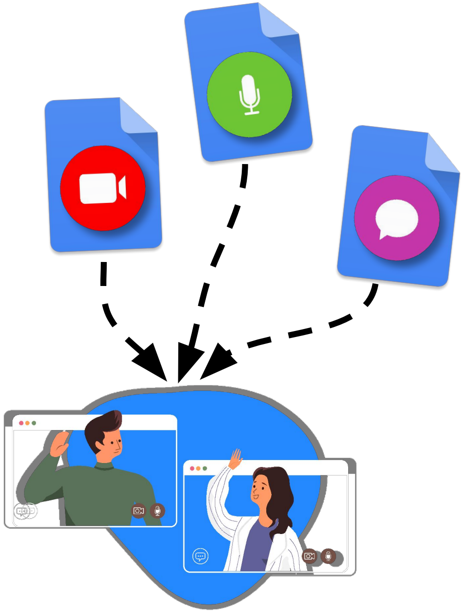
Don't take speed at face value!

- **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



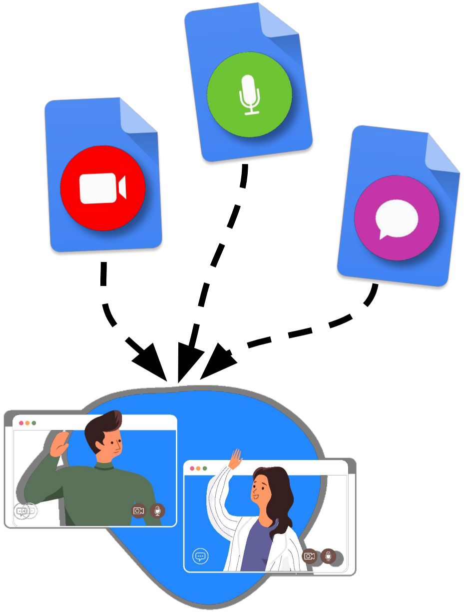
Don’t take speed at face value!

- **Slower** may mean...
  - **???**



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



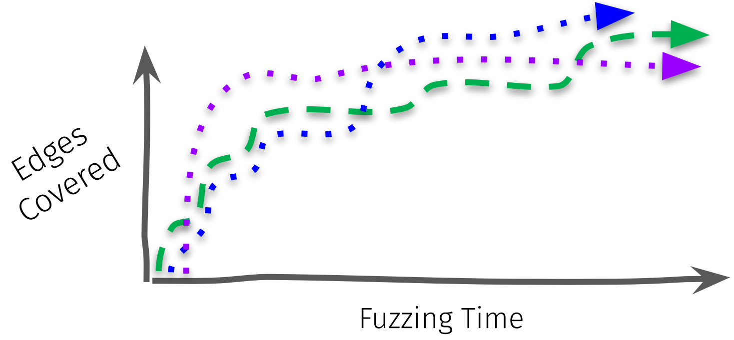
Measure and plot your code coverage!

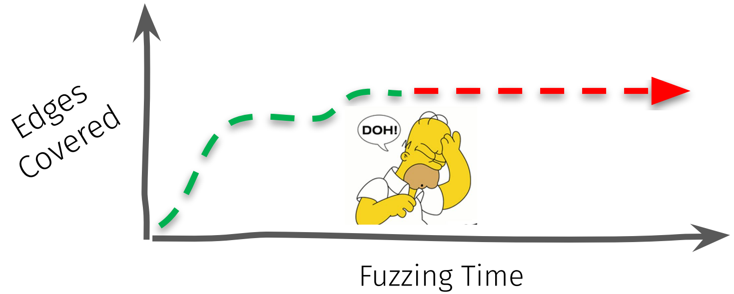
- Critical to understanding your harness
  - Changes in **edges covered**
  - Changes in edge **hit counts**
  - Source code **visualizations**

- Useful coverage tools
  - github.com/mrash/afl-cov
  - github.com/gcovr/gcovr
  - github.com/andreafioraldi/afl-qemu-cov
  - 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

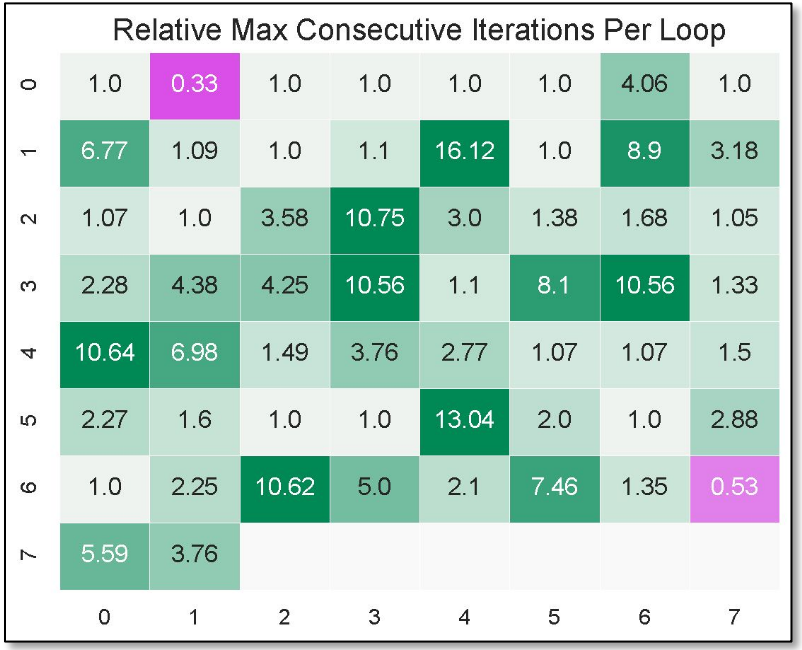




What does your code coverage tell you?

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

[1]	[2]	[3]	[4,7]
[8,15]	[16,31]	[32,127]	[128+]



What does your code coverage tell you?

- **Source line coverage** (e.g., gcov)
  - Costs **more time** to generate reports
  - Provides you **more information**
  - Does not support binaries

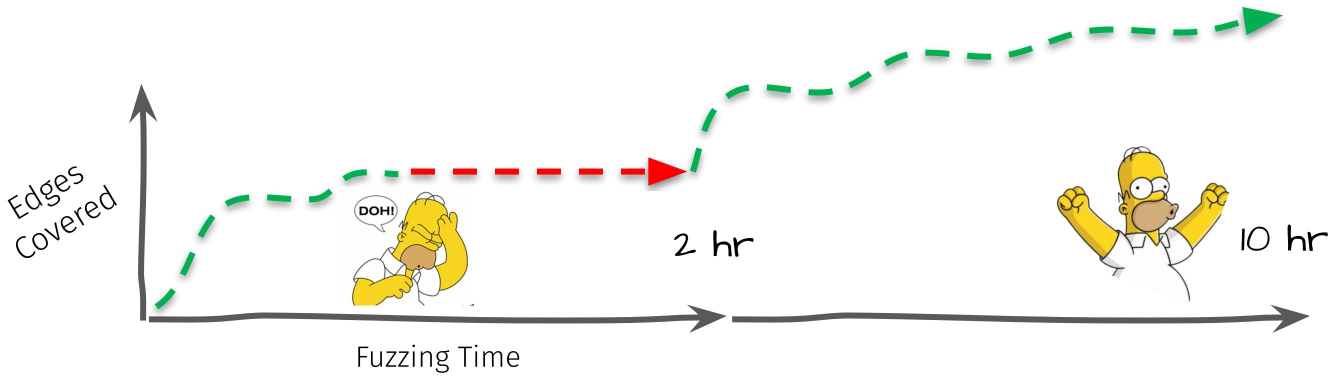
Line Coverage			Functions	
	0.0 %	0 / 11	0.0 %	0 / 1
	0.0 %	0 / 129	0.0 %	0 / 12
	0.0 %	0 / 22	0.0 %	0 / 2
	75.3 %	64 / 85	69.2 %	9 / 13
	100.0 %	102 / 102	88.9 %	8 / 9
	89.5 %	17 / 19	57.1 %	4 / 7
	100.0 %	31 / 31	88.9 %	8 / 9
	34.4 %	56 / 163	50.0 %	5 / 10
	100.0 %	51 / 51	88.9 %	8 / 9
	30.4 %	24 / 79	40.0 %	4 / 10
	100.0 %	24 / 24	88.9 %	8 / 9

```
Line data    Source code
          : #include <stdio.h>
        1 : int main()
          : {
          :     int i, j, rows;
          :
        1 :     printf("Enter number of rows: ");
        1 :     scanf("%d",&rows);
          :
        9 :     for(i=1; i<=rows; ++i)
          :     {
       44 :         for(j=1; j<=i; ++j)
          :         {
          :
          : //// LCOV_EXCL_LINE
       36 :             printf("* ");
          :         }
          :
          : //// LCOV_EXCL_LINE
        8 :         printf("\n");
          :     }
        1 :     return 0;
          : }
```

Are you fuzzing for long enough?

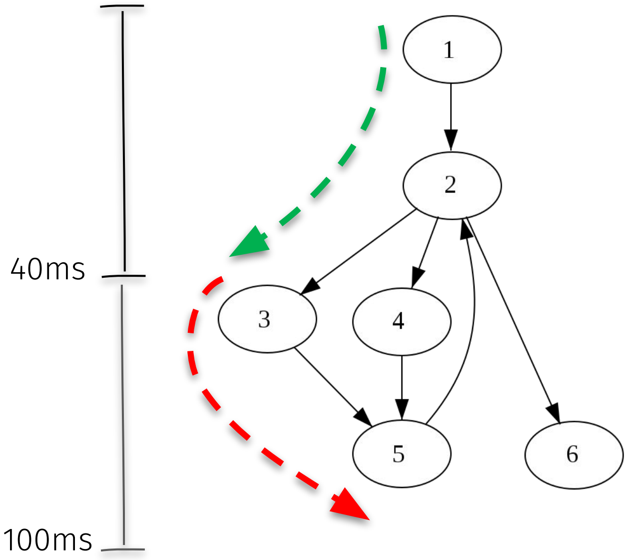
- **Early plateaus** can be misleading
  - Look for **sustained** plateaus

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



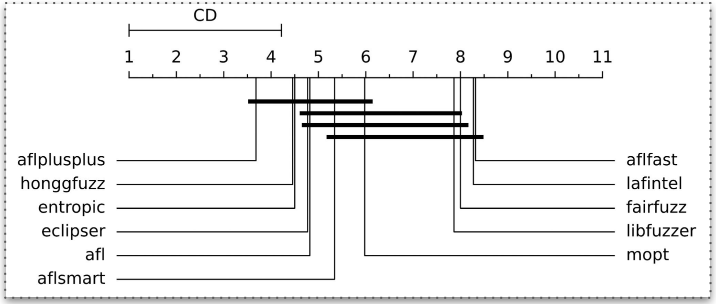
Is your execution timeout large enough?

- **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



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!**

Questions?