Week 6: Lecture B

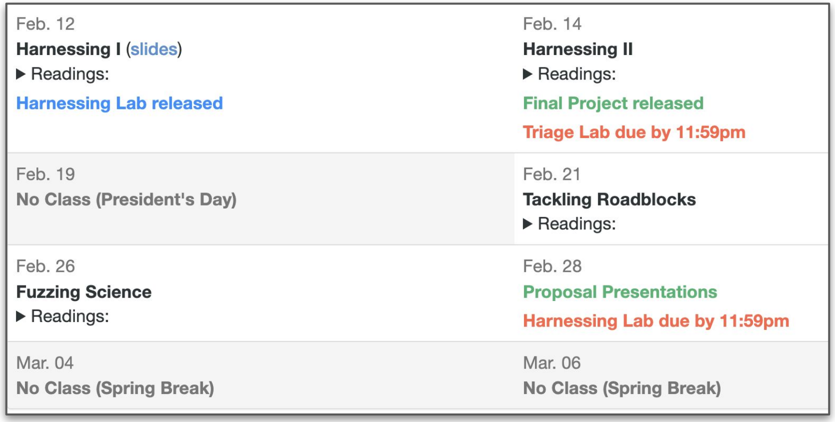
Harnessing II

Wednesday, February 14, 2024

Recap: Key Dates

Feb. 14	Lab 2 due
Feb. 14	Final Project released
Feb. 19	No class (President's Day)
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** ▶ 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:	Feb. 28 **Proposal Presentations** **Harnessing Lab due by 11:59pm**
Mar. 04 **No Class (Spring Break)**	Mar. 06 **No Class (Spring Break)**

Recap: Lab 2 Overview

- **Assignment:** learn how to use AddressSanitizer (ASAN)
  - Read its documentation in https://clang.llvm.org/docs/AddressSanitizer.html
- **Replay the crashes you found in Lab 1 on an ASAN-instrumented binary**
  - Collect information on each crash
  - What do you observe?
- **Deliverable:** a **1–3 page report** detailing your findings
  - Feel free to make it your own (e.g., pictures, text, etc.)
- **Linux environments are recommended**
  - Use a VM if you don't have one!

Recap: Lab 2 Tips

- **Re-run crashes on the ASAN instrumented binary**
  - Use Python to script collection of ASAN outputs
  - Do string post-processing to collect error types, crashing source line, etc.
  - Group and deduplicate crashes as you see fit

- **Didn't find any crashes in Lab 1?**
  - Try fuzzing fuzzgoat from **https://github.com/fuzzstati0n/fuzzgoat**
  - Should yield **lots** of crashes quickly

Lab 3: Harnessing

- **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!

Lab 3: Harnessing

- **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

- **No class this coming Monday**, February 19th

Lab 3 Tips

- **Read its 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

Questions?

Semester Final Project

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

Semester Final Project

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

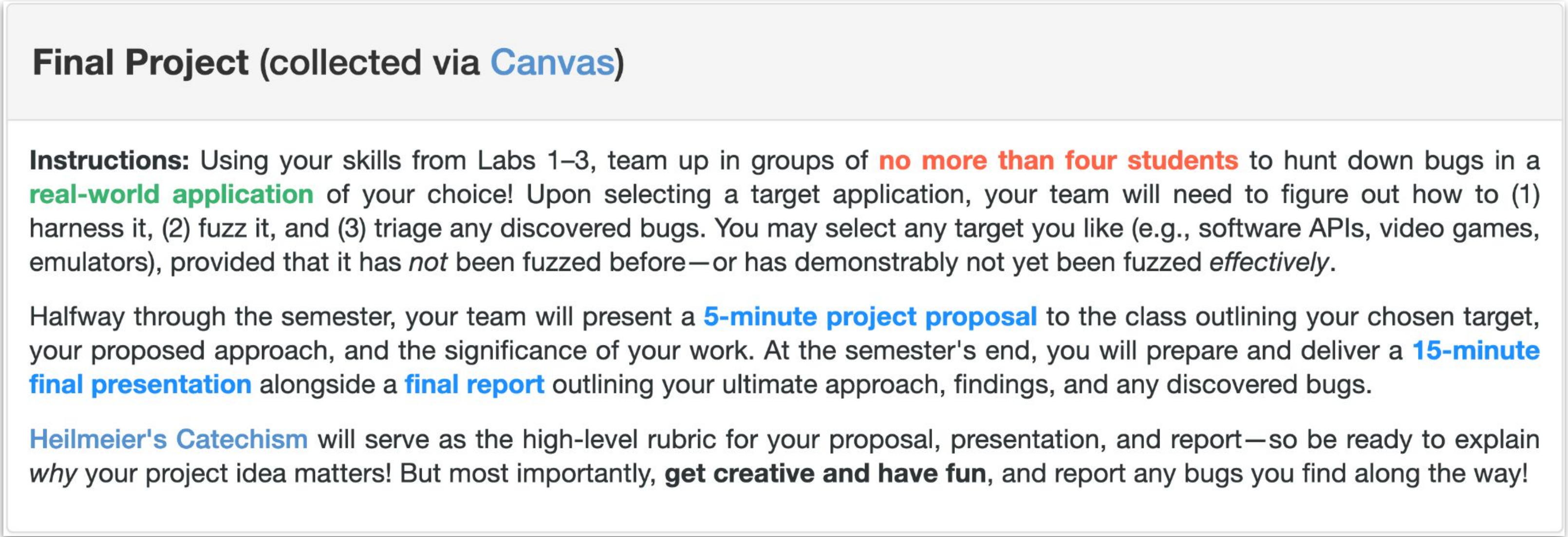
5-minute project **proposal** on Feb. 28

Final presentations at semester’s end

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

Semester Final Project

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



Final Project (collected via Canvas)

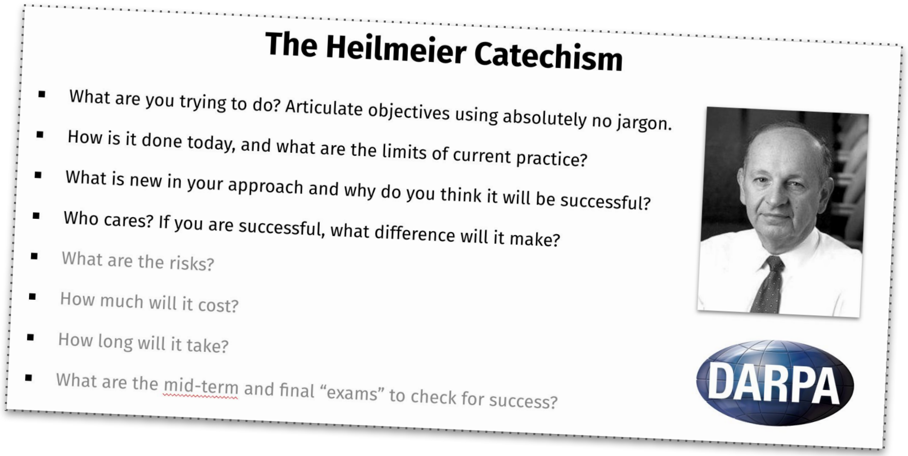
Instructions: Using your skills from Labs 1–3, team up in groups of **no more than four students** to hunt down bugs in a **real-world application** of your choice! Upon selecting a target application, your team will need to figure out how to (1) harness it, (2) fuzz it, and (3) triage any discovered 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!

Project Schedule

- **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



The Heilmeier Catechism

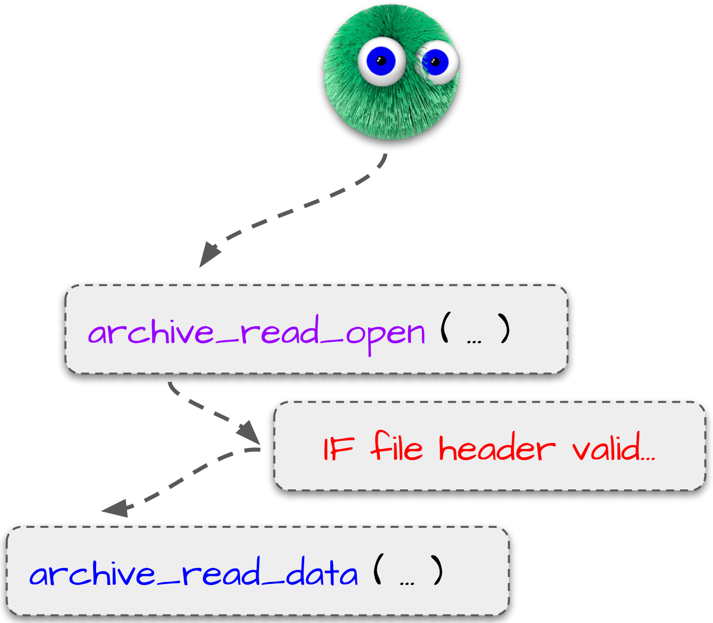
- What are you trying to do? Articulate objectives using absolutely no jargon.
- How is it done today, and what are the limits of current practice?
- What is new in your approach and why do you think it will be successful?
- Who cares? If you are successful, what difference will it make?
- What are the risks?
- How much will it cost?
- How long will it take?
- What are the mid-term and final "exams" to check for success?

Questions?

Harnessing Recap

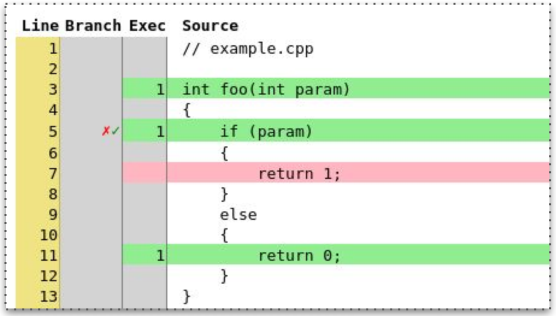
Harnessing

- **Definition:** making a program **fuzzable**
  - Pass input data to a program's core logic
  - Skip functionality we don't care about
  - Drop-in integration with fuzzers (e.g., AFL)
- **A critical (and difficult) part of fuzzing**
  - Lots of domain expertise
  - Automating still an open problem



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)

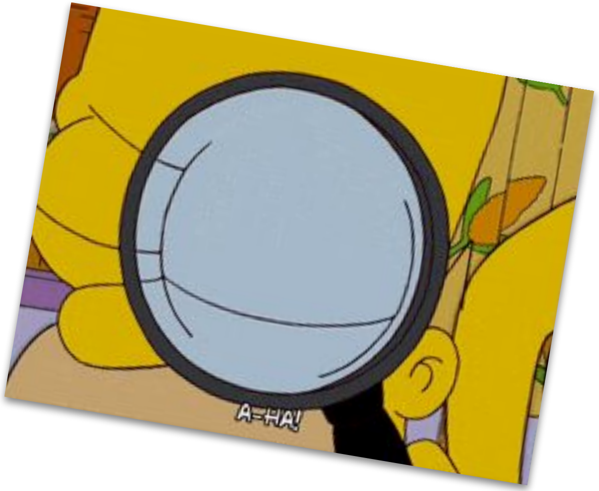


Line	Branch	Exec	Source
1			`// example.cpp`
2			
3		1	`int foo(int param)`
4			`{`
5	✗✓	1	`    if (param)`
6			`    {`
7			`        return 1;`
8			`    }`
9			`    else`
10			`    {`
11		1	`        return 0;`
12			`    }`
13			`}`

Identifying suitable targets

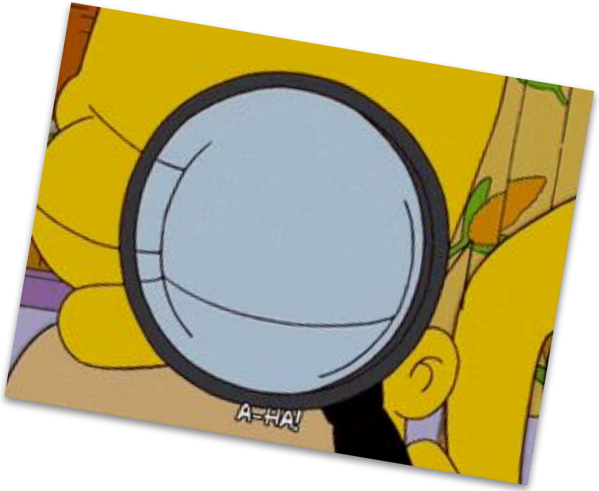
- **For libraries** (e.g., libJPEG):
  - Find API's "consumer" functions
  - Review documentation and figure out how to call it

- **For applications** (e.g., Adobe Reader):
  - Find what directly loads data (e.g., calls **`fopen()`**)
  - Skip-over irrelevant setup code



Identifying suitable targets

- **For libraries** (e.g., libJPEG):
  - Find API's "consumer" functions
  - Review documentation and figure out how to call it
- **For applications** (e.g., Adobe Reader):
  - Find what directly loads data (e.g., calls **`fopen()`**)
  - Skip-over irrelevant setup code
- **Can harnessing be automated?**



Automated Harnessing

Current Automated Harnessers

- **Single-function harnessing:**
  - Fuzzable: github.com/ex0dus-0x/fuzzable
  - Choose targets with highest **cyclomatic complexity**

- **Harness many functions:**
  - **Open-source:**
    - FUDGE: research.google/pubs/pub48314/
    - FuzzGen: github.com/HexHive/FuzzGen
  - **Closed-source:**
    - Winnie (Windows): github.com/sslab-gatech/winnie
    - APICraft (Mac APIs): github.com/occia/apicraft
  - Find targets, and **"stitch" together preceding control and data flow**

General Workflow

- **Step 1:** choose your targets
  - Heuristic driven
- **Step 2:** graph construction
  - Control flow graph, call graph
- **Step 4:** type recovery
  - Function arguments
  - Return values
- **Step 4:** stitch it all together
  - Call functions in logical order
  - Match return values with arguments

Step 1: Target Discovery

- **Functions that take inputs as…**
  - File names or paths
  - File descriptors
  - Buffered data
  - Pointers

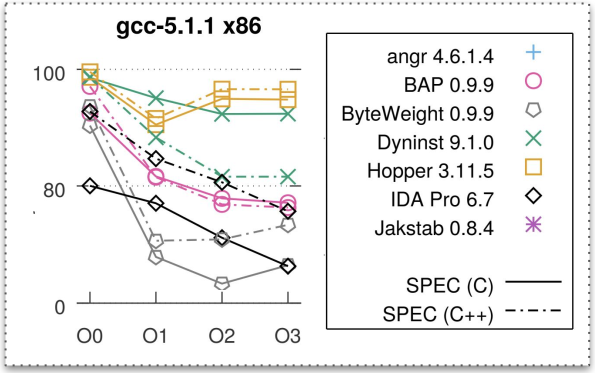
```
FILE *fp = fopen("filename", "rb");

int *arr = (int*) calloc(4096, sizeof(int));

int ret = foo (arr, &callback, &fp);
```

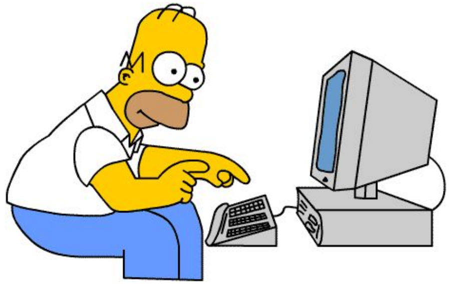
- **Approaches:**
  - **Static identification:** scan the program and mine interesting patterns
  - **Dynamic tracing:** look for patterns when tracing program on some inputs

Source: WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning

Challenges

- **Static identification:**
  - **Open-source:** slower as program size grows
  - **Closed-source:** can be derailed by errors
- **Dynamic tracing:**
  - Largely a press-and-click **manual task**
  - Only as good as **what you capture**
    - Not every input invokes interesting functions

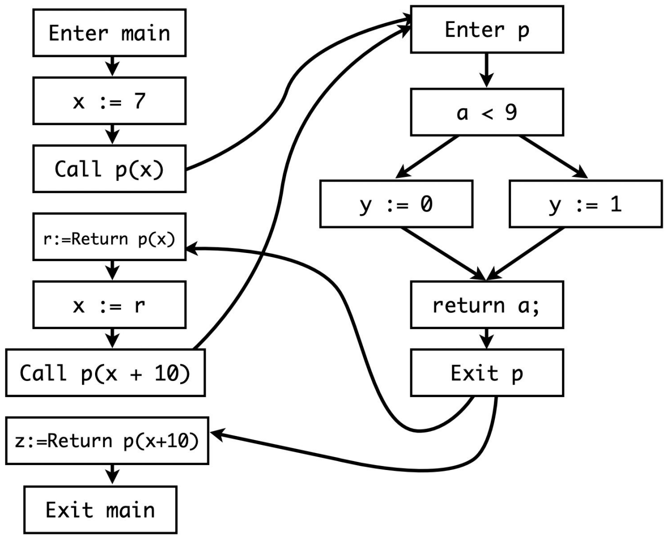




Source: An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries

Step 2: Control-flow Graph

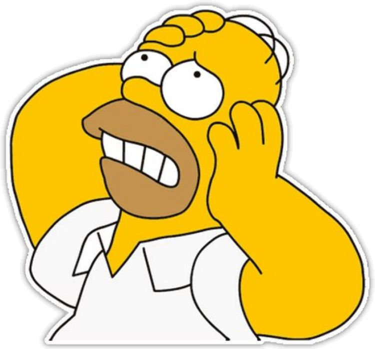
- **Collect intra- and inter-procedural CFGs:**
  - **Intraprocedural**: control flow in **one function**
    - Jumps
    - Local gotos
    - Loops
  - **Interprocedural**: transfers **between functions**
    - Calls
    - Non-local gotos
    - Returns
  - **Merge across all application components**



Source: https://groups.seas.harvard.edu/courses/cs252/2011sp/slides/Lec05-Interprocedural.pdf

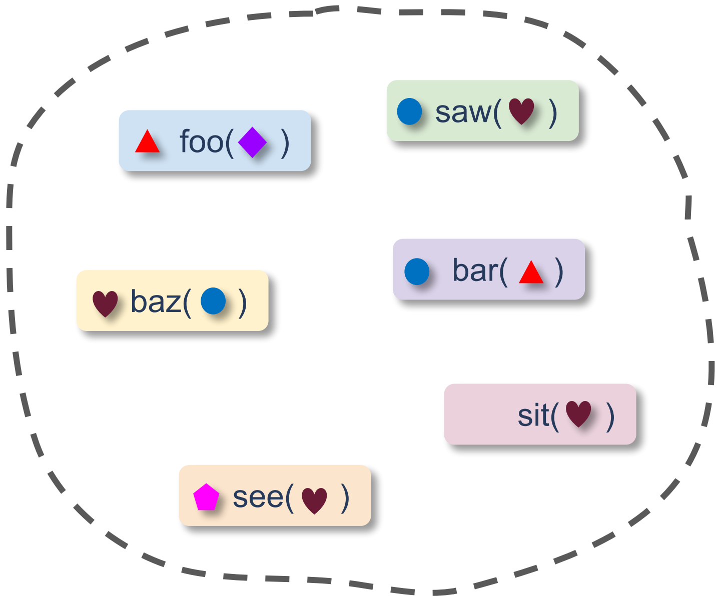
Challenges

- Recovery of **indirect edges**
  - Indirect jumps and calls, returns
  - **Undecidable problem** (even for open-source)
    - Best guess: **over-approximated** set based on some level of alias analysis
- The usual **closed-source hurdles**
  - Function boundary recovery
  - Instruction recovery
  - …



Step 3: Type Recovery

- Recover **function prototypes**
  - Arguments
  - Return types
- Self-encoded in source code
  - Meticulously extracted from binaries



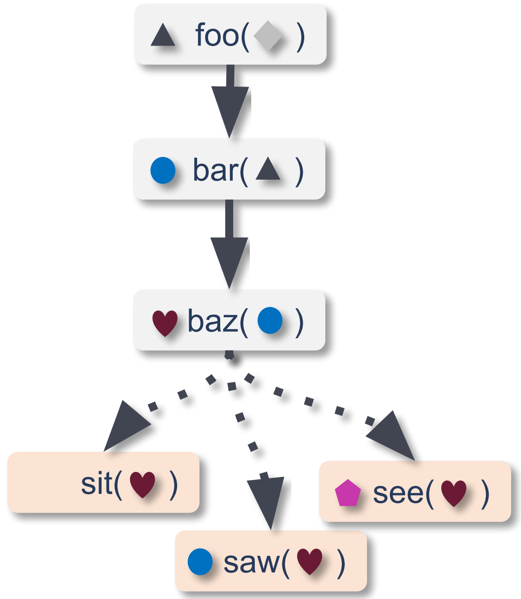
Challenges

- **Type recovery is hard for closed-source**
  - **Best case:** have symbol metadata intact
    Worst case: metadata completely stripped
    - Task becomes really **analysis intensive**
    - Generally reliant on lots of heuristics

- **Pointers are hard to analyze**
  - Pointers to functions vs. data
    - Meticulous memory inspection
  - Current tools see mixed results

Step 4: Stitching

- **Connect the dots** to form the harness
  - Call functions in a logical order
  - Must reach the function you care about
  - Match return values and function arguments



Challenges

- **Some arguments shouldn't be fuzzed**
  - Must preserve argument **dependencies**
  - Requires intensive data-flow analysis

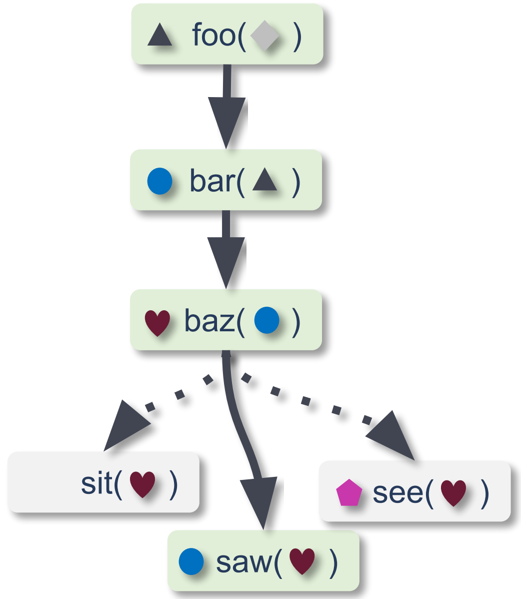
- **Must recover input validity checks**
  - Allowable ranges, status values
  - Conditional API dependencies
  - "Callback" functions passed as args
  - **Default back to human expert**

```
void *memcpy(void *dest,

             const void *src,

             size_t n);
```

```
if (v == ARC_FATAL) { exit(); }
```

Evaluating Candidate Harnesses

- Potentially **large solution space**
  - E.g., over-approximated indirect edges
  - Need to whittle-down to final candidates
- Apply **differential testing**
  - On basic fuzzing input seeds
  - Developer-provided smoke tests
  - Coverage over time
- **Make human expert have final say**



Questions?