Vulnerability Finding
First: questions so far?
Today: finding software vulnerabilities

- George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei and Michael Hicks, “Evaluating Fuzz Testing”, ACM CCS ’18

Today: finding software vulnerabilities

- George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei and Michael Hicks, “Evaluating Fuzz Testing”, ACM CCS ’18

Fuzz Testing

- What is it?
- Testing on “random” inputs
- Testing for what?

- Make sure that certain “bad things” don’t happen
  - Crashes, exceptions, infinite loops, etc...
  - Evidence of potential vulnerability
  - Why do fuzzers predominantly use crashes as their oracle of badness?

- Key: LARGE numbers of tests
Seeds and input mutation

- Purely random input can lead to poor coverage
  - Why?

- Instead, can impose more structure
  - Mutate valid (seed) input to create new test cases
  - Use grammar to generate random test cases
Visibility into program state (coverage)

- Black box
- Grey box
- White box
Why this paper now?

- What kind of paper is this?

- What is key point of paper?
Meta-paper, paper about research

- How can we (the research community) do better?

- Can we establish “rules” to ensure that published papers capture true “advances” in the state of the art
Today: finding software vulnerabilities

- George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei and Michael Hicks, “Evaluating Fuzz Testing”, ACM CCS ’18

Symbolic evaluation

- Basic idea:
  - Need lots of tests to get coverage w/fuzzing because each concrete test covers one particular path (and you’re trying at random!)
  - Instead, allow program to take ANY feasible path and let code itself identify what input values trigger which code paths
    - Use code to construct its own input!
  - On a branch: (e.g., if $i > 10$)
    - If conditional variable is concrete (e.g., $i$ set to 5 earlier) this is easy
    - If its not (i.e., its based on input) then we call it “symbolic”
      - fork() program to execute both paths
      - On each path add the constraint of that branch (e.g., $i > 10$ or $i <= 10$)
      - Resulting constraints (called the path condition) define the input requirements to cover distinct program states
  - On exit or error solve for constraints (how did I get here?)
A toy example

- Initial state: x unconstrained
- Code will return 3 times.
- Solve constraints at each return = 3 test cases.
EXE: Why this paper now?

- What kind of paper is this?
- What is key points of paper?
Issues

- Which paths to try?
  - DFS blows up on loops bounded by symbolic variables
  - Heuristic scheduler
    - Run process/child executing line of code that has run the fewest number of times
    - Still can have exponential blow up in paths

- Don’t encode certain kinds of expressions in constraint system
  - Double de-reference pointers, function pointers

- What constraints can it solve?
  - Limits of STP
  - Performance for non-linear operations (i.e., div/mod)
Evaluation

- How does this paper evaluate its approach?
Finding software vulnerabilities

- What do you think is most effective approach?
For those interested here

- Check out KLEE (successor to EXE)
  - Also angr, Qsym, SymCC

- A variety of work that has tried to combine fuzzing and symbolic methods (i.e., symbolic-driven mutation/evaluation)

- Lots of potential projects in this space
  - Lots of tools and lots of open-source code (with version histories and bug databases)
Next time

- Please send me your tentative groups!

- Two papers on software defenses
  - CFI and Nozzle