Fuzzing

Overview of security testing and fuzz testing:
Trying to use testing to detect bugs or vulnerabilities in a program. Similar to static analysis.
Made up of test case generation + bug oracle.
You find some way of the creating test cases -> run the program under tests -> bug oracle (tells you if the output/state is good or bad. Crashes are easy to detect and clear sign of vulnerability).

Bug Oracle:
- Does it crash? (most common)
- Any runtime monitor (bounds checking). i.e. Valgrind.

Fuzz testing: random test cast generation.
- Earliest type: random input. Usually works poorly, because there are practically infinite combinations of random bytes and odds are you won't find the appropriate sequence to trigger some behavior.
- Random mutation: take some input and then randomly mutate it. Usually looks at the output to determine if a bug was found. Decent.
- AFL (grey-box): Rather than modifying inputs randomly, do it in a smarter way. It focuses on the execution path and knows how to emphasis mutating "interesting" test cases. Game changer. Helped finding many safety bugs.
  - Minimization: avoids running the same path even when given different input files (if they include the same execution paths)

Magic Bytes:
A png file, for example, can contain validation chunks that the program checks before executing. Fuzzing is unlikely to fuzz a file to contain those.

Say we're given the two following chunk check-up schemes:
If (a[0] == 0xAE) {
  If (a[1] == 0x3C) {
    If (a[2] == 0x79) {
      f() .....}}}
===> ~4*2^8 test cases until f() is triggered

If (a[0...3] == 0xAE3C7920) {
  f() {
  ===> ~2^32 test cases until f() is triggered

Could a compiler do anything to make this easier? Yes. For example, a compiler could convert method 2 to method 1.
Or, you could have coverage: anytime you make a comparison and it fails, you can count the longest prefix that has been matched. Coverage (line of code, how many bits matched)
Checksums:
Example code:
If (checksum(a[0...n-3]) != a[n-4...n-1])
    error()
Somewhat similar to magic bytes, but could be changed dynamically.

Differential testing:
- bug oracle checks if f(x) == g(x) == h(x).
- Parse(unparse(parse(s))) == parse(s)

Symbolic execution:
Similar to fuzz testing, but the test case generation is smarter:
How to do it:
- Given a path through the code: is there any input that follows that path and causes a bug?
  - z = 1/x → assert (x != 0); z = 1/x
- Given a path through the code: is there any input that follows a different path?
  - x = inputs; v = variables; v == f(x) & PC(x,v).
  - Notice: PC is a path constraint. Some sort of SAT, for example.
    - v == f(x) & PC(x,v) & v == 0

Discussion: How can it detect buffer overruns? Like array out of bounds:
- Bounds checking problem. We can use any bounds checking scheme.
- To check any vulnerability, add whatever checking scheme as assert statements, then run the symbolic executor.
Quick discussion on the over-constraint problem and QSYM’s solution

if $x > 0$
  $x = x + 20$
else
  $y = f(x)$
if $x \leq -10$
  $y = \frac{x}{(x + 876329)}$

Q1 Is there a divide-by-zero error in the code?
Q2 How would a symbolic execution-like tool find it?
  a. How would such an engine deal with $f$? ($f$ is unknown/heap/binary/...)

* 5 minute symbolically executing the example including discussion of what to do with $f$
  - treat as UF
  - try to model (write a spec or summarize)
  - symbolically execute inside (non-modular)
  - concretize and call (and add concretization to path constraint)

Q3 If concretize, then say we concretize $x$ to -1 just before calling $f$.
    What happens now?

QSYM combines fuzzing with symbolic execution to catch these corner cases—but that wouldn’t
help here. Their goal is to explore as much of the target program as fast as possible.

Q4 How can we fix this overconstrained issue?

Q5 QSYM will look at a subset (specifically the last constraint). Will this work here?
How to deal with external procedure calls?
What is the overconstrained problem?
Can we select subset of constraints to solve?
Which ones and why?
What happens if we're wrong?
- Our model is wrong
- We say unmatch again but it actually is set

Their contributions:
- Instruction level symbolic execution
- Optimistic constraint solving
- Basic block pruning

**Over constrained!**

\[
f(x) \text{ is complicated}
\]

So keep corectible
\[x \text{ to be } -1\]
\[f(-1) = 0\]
\[x \times -10\]
\[x \times -10\]