```java
public class Puzzle17 {

    public static void main(String[] args) {
        run(1.0 / 0.0);
    }

    public static void run(double i) {
        while(i == i + 1) {
            System.out.println(i);
        }
    }

}
```

Dynamic Analysis

- How are program inputs generated?
- What is monitored in the program?
- What is the analysis searching for?
- What is executed in the program?

How are inputs generated?

Blind (Traditional/Dumb) Fuzzing

1A. Wordlist enumeration or bruteforce search of input space

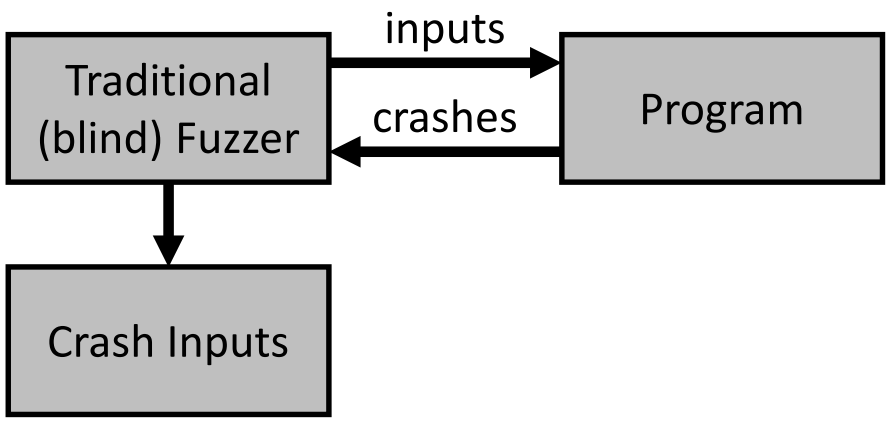
- Optionally seed or prioritize inputs with statically recovered artifacts (ex: string literals)

1B. Start with a test corpus of well formed program inputs

- Apply random or systematic mutations to program inputs

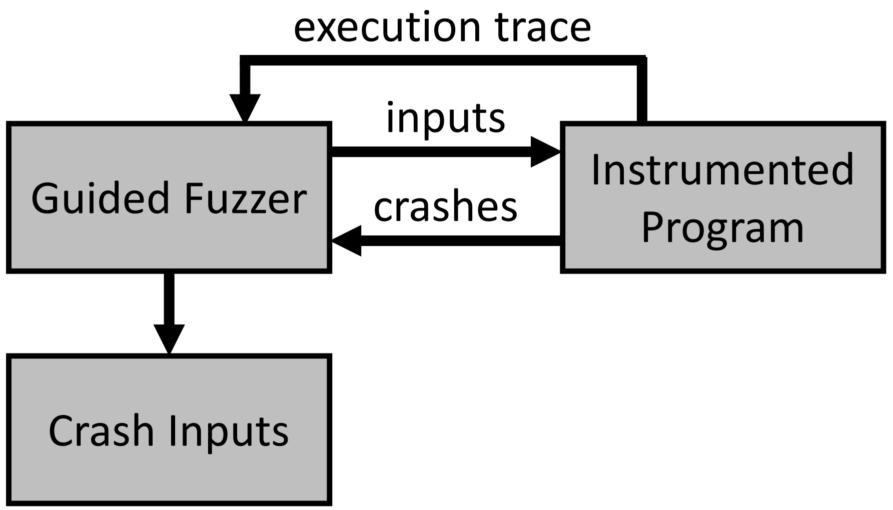
2. Run program with inputs and observe whether or not the program crashes

3. Repeat until the program crashes



Guided/Directed Fuzzing I: Feedback Driven Input Generation

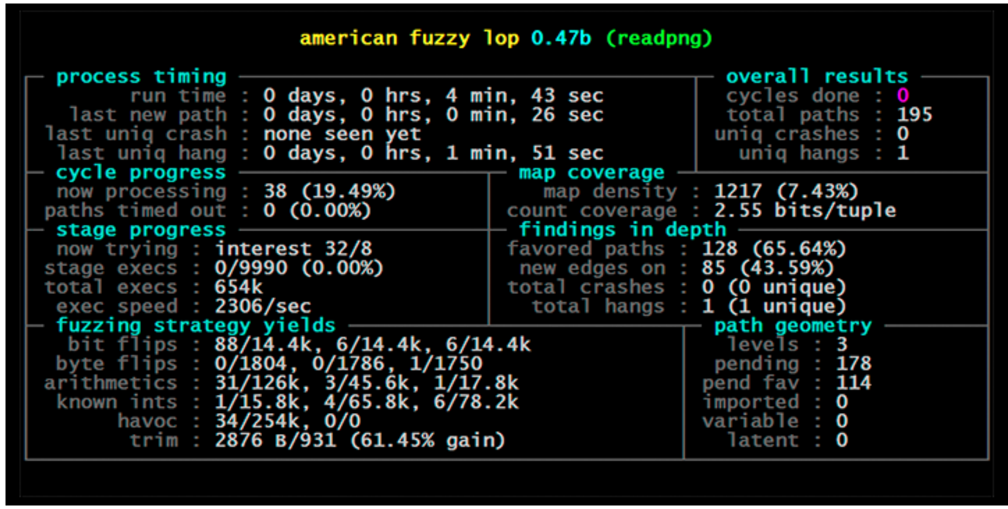
- Start with a test corpus of well formed program inputs
- Apply random or systematic mutations to program inputs
- Instrument the program branch points
- Run the instrumented program with mutated inputs and 1) observe whether or not the program crashes and 2) record the program execution path coverage
- If the input results in new program paths being explored then prioritize mutations of the tested input
- Repeat until the program crashes



Heuristics guide genetic algorithm to generate program inputs that push the fuzzer deeper into the program control flow, avoiding the common pitfalls of fuzzers to only test "shallow" code regions.

AFL (American Fuzzy Lop) Fuzzer

- Recognized as the current state of art implementation of guided fuzzing
  - Effective mutation strategy to generate new inputs from initial test corpus
  - Lightweight instrumentation at branch points
  - Genetic algorithm promotes mutations of inputs that discover new branch edges
    - Aims to explore all code paths
  - Huge trophy case of bugs found in wild
    - 371+ reported bugs in 161 different programs as of March 2018
    - http://lcamtuf.coredump.cx/afl/
- A game of economics. AFL tends to “guess” the correct input faster than a smart tool “computes” the correct input.



```
                    american fuzzy lop 0.47b (readpng)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 4 min, 43 sec       │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 26 sec       │  total paths : 195    │
│ last uniq crash : none seen yet                      │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 1 min, 51 sec       │   uniq hangs : 1      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 38 (19.49%)       │    map density : 1217 (7.43%)          │
│ paths timed out : 0 (0.00%)         │ count coverage : 2.55 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : interest 32/8         │ favored paths : 128 (65.64%)           │
│ stage execs : 0/9990 (0.00%)        │  new edges on : 85 (43.59%)            │
│ total execs : 654k                  │ total crashes : 0 (0 unique)           │
│  exec speed : 2306/sec              │   total hangs : 1 (1 unique)           │
├─ fuzzing strategy yields ───────────┴─────────────┬─ path geometry ──────────┤
│   bit flips : 88/14.4k, 6/14.4k, 6/14.4k          │    levels : 3            │
│  byte flips : 0/1804, 0/1786, 1/1750              │   pending : 178          │
│ arithmetics : 31/126k, 3/45.6k, 1/17.8k           │  pend fav : 114          │
│  known ints : 1/15.8k, 4/65.8k, 6/78.2k           │  imported : 0            │
│       havoc : 34/254k, 0/0                        │  variable : 0            │
│        trim : 2876 B/931 (61.45% gain)            │    latent : 0            │
└───────────────────────────────────────────────────┴──────────────────────────┘
```

Test Harness

- A main method (program entry point) is required to execute a program
  - A library typically does not have a main method, so one must be provided to make a complete executable program
- Practically needed to translate fuzzer generated inputs to expected program input format
  - Example: AFL generates a file as input. To fuzz a DNS service the harness must translate the generated input file to a DNS packet or data structure a function takes as a parameter

What is executed in the program?

Targeted Fuzzing

- Fuzzing of whole program or subset of program?
- Most techniques necessitate manually developing a harness, which is a natural opportunity to "target" fuzzing on a subset of the program
- Some functions are more natural to fuzz then others (ex: library APIs)
    - Helper functions may depend on state of global variables or complex data structures as parameters
- To generically fuzz a function or set of functions the dependencies must be mocked
    - Fuzzing internal program states (mocked dependencies) may ignore a practical constraint on program state enforced at runtime
    - Human reasoning could be used to add fuzzer input generation constraints

What is the analysis searching for?

What is the analysis searching for?

- Crash vs. no crash
- Expected vs. unexpected output
- Tainted vs. untainted output
  - Example: Web app fuzzers provide XSS code as input and monitor for XSS execution
- Harness can translate a domain specific problem to a standard detection output
  - Example: *if(some program state) { crash(); }*

What is monitored in the program?

What is monitored in the program?

- Blackbox
  - No knowledge of program internals or state
  - Only monitoring inputs / output values or behaviors
- Whitebox/Graybox
  - AFL monitors execution path for feedback driven input generation
  - Daikon monitors variable values at selected program points and reports a catalog of invariants that held during execution
    - Arguably this is fuzzing without a specification of how to do input generation although input generation is still required for the analysis