Random Testing (Fuzzing)

https://www.youtube.com/playlist?list=PLF3-CvSRq2SaApl3Lnu6Tu_ecsBr94543

- Feed random inputs to a program
- Observe whether it behaves “correctly”
  - Execution satisfies given specification
  - Or just doesn’t crash
    - A simple specification
- Special case of mutation analysis
  - **Fuzzing** can be viewed as a technique that randomly perturbs a specific aspect of the program. Namely, its input from the environment, such as the user or the network. Mutation analysis, on the other hand, randomly perturbs arbitrary aspects of the program.

The infinite monkey theorem

- “A monkey hitting keys at random on typewritter keyboard will produce any given text, such as the complete works of Shakespeare, with probability approaching 1 as time increases.”
- The monkey is a metaphor for a device that produces an endless random sequence of keys. Translated into our setting of random testing, the monkey is the first testing tool. And typing a given text is analogous to the monkey finding an input that exposes a bug in the program being tested.

Random testing: case studies

- UNIX utilities: Univ. of Wisconsin’s fuzz study
- Mobile apps: Google’s Monkey tool for Android
- Concurrent programs: Cuzz tool from Microsoft

A popular fuzzing study

- Conducted by Barton Miller @ Univ. of Wisconsin
- 1990: Command-line fuzzer, testing reliability of UNIX programs
  - Bombards utililities with random data
- 1995: Expanded to GUI-based programs (X Windows), network protocols, and system library APIs
- Later: Command-line and GUI-based Windows and OS X apps

Fuzzing UNIX Utilities: aftermath

- 1990: Caused 25-33% of UNIX utility programs to crash (dump state) or hang (loop indefinitely)
- 1995: Systems got better … but not by much!
- “Even wors is that many of the same bugs that we reported in 1990 are still present in the code releases of 1995.”

A silver lining: security bugs

- gets() function in C has no parameter limiting input length
  - Programmer must make assumptions about structure of input
- Causes reliability issues and security breaches
  - Second most common cause of errors in 1995 study
- Solution: Use fgets(), which includes an argument limiting the maximum length of input data

Fuzz testing for mobile apps

- One domain in which fuzz testing has proved useful is that of mobile applications. A popular fuzz testing tool for mobile applications is the Monkey tool on the Android platform.

```
class MainActivity extends Activity implements OnClickListener {
   void onCreate(Bundle bundle) {
      Button buttons = new Button[] { play, stop, ... };
      for (Button b : buttons) b.setOnClickListener(this);
   }
   void onClick(View target) {
      switch (target) {
      case play:
         startService(new Intent(ACTION_PLAY));
         break;
      case stop:
         startService(new Intent(ACTION_STOP));
         break;
      ...
   }
}
```

TOUCH(136,351)

TOUCH(136,493)

TOUCH(x, y) where x, y are randomly generated:
x in [0..480], y in [0..800]

Generating multiple-input events

- Generating a single event is not enough to test realistic mobile apps. Typically a sequence of such events is needed to sufficiently test the app's functionality. Therefore the Monkey tool is typically used to generate a sequence of touch events, separated by a set amount of delay.

Generating gestures

- A common kind of input to mobile apps is gestures. By generating a sequence of touch events random testing can generate arbitrary gestures.

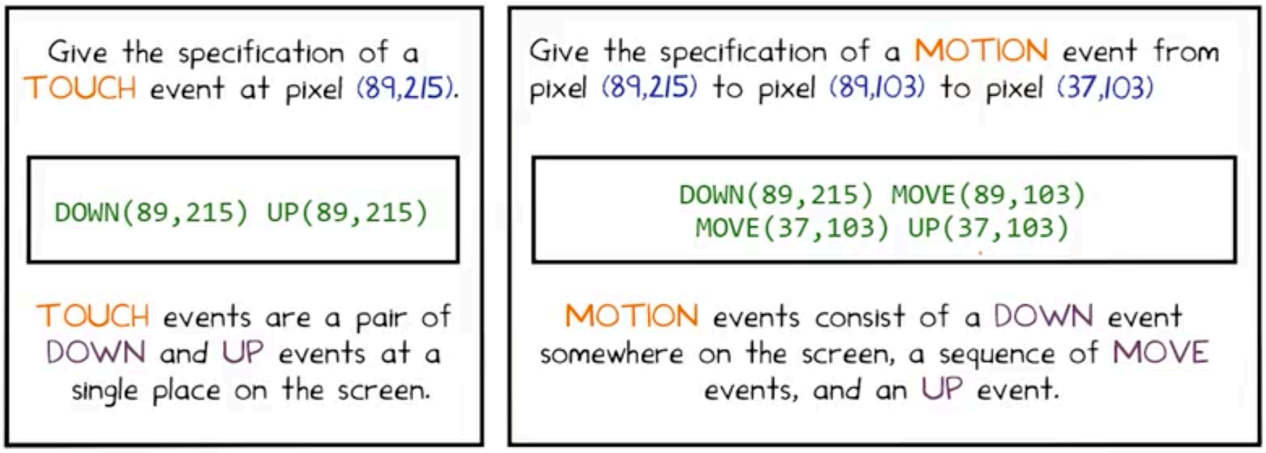
Grammer of Monkey events

- A grammar that systematically characterizes the possible inputs that the Monkey tool can generate. Each test case, or input, is a sequence of some number of events.

```
test_case := event*
event := action(x, y)
action := DOWN | MOVE | UP
x := 0 | 1 | ... | x_limit
y := 0 | 1 | ... | y_limit
```

Monkey events example

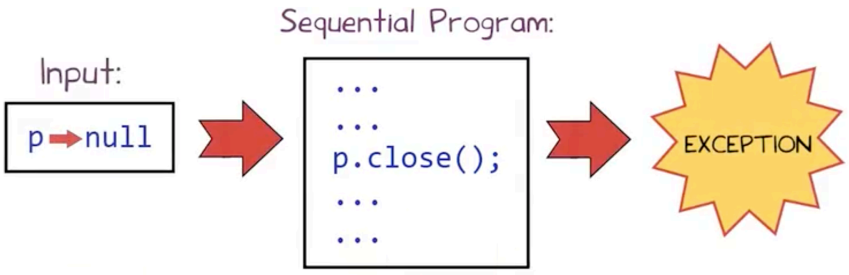
- In Monkey's grammer using UP, MOVE, and DOWN statements:



Testing concurrent programs

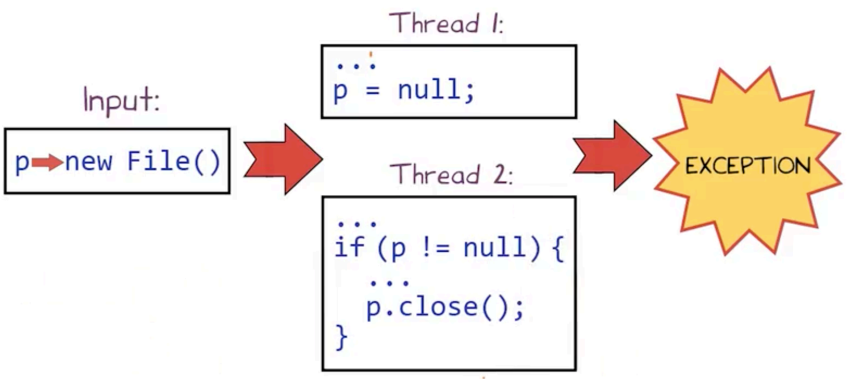
- Another important domain in which random testing is exceedingly useful is the testing of concurrent programs.

  In a sequential program, a bug is triggered under a specific program input, and testing sequential programs is primarily concerned with techniques to discover such an input.



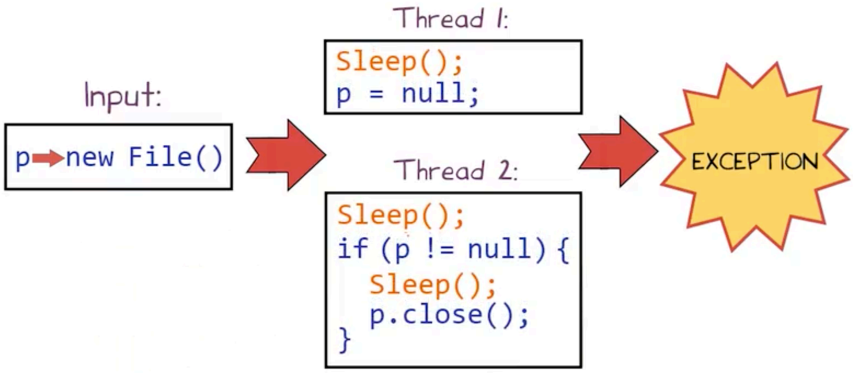
- Unlike a sequential program, which consists of a single computation, a concurrent program consists of multiple threads of computation that are executing simultaneously and potentially interacting with each other.

  In a concurrent program, a bug is triggered not only under a specific program input, but also under a specific thread schedule which may be viewed as the order in which the computation of different threads is executed.



Concurrency testing in practice

- The predominant approach to testing concurrent programs today is to introduce random delays indicated by the calls to a system function called Sleep.



- Note however, that unlike in the case of the Unix fuzzing experiment, where we fuzzed program inputs, here we are fuzzing the threads scheduler. This is the key underlying the concurrency fuzzing tool from Microsoft called **Cuzz**.

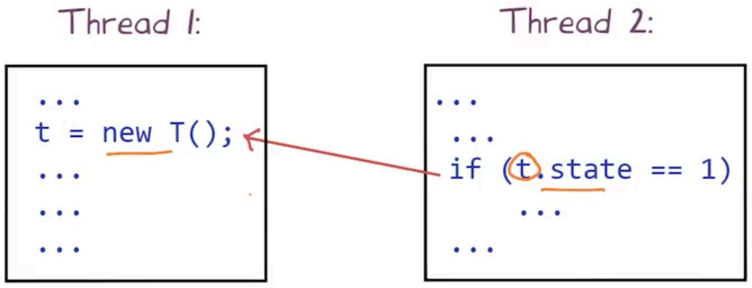
Cuzz: fuzzing thread schedules

- Introduces Sleep() calls
  - Automatically (instead of manually)
  - Systematically before each statement (instead of those chosen by tester)
    - Less tedious, less error-prone
- Gives worst-case probabilistic guarantee on finding bugs

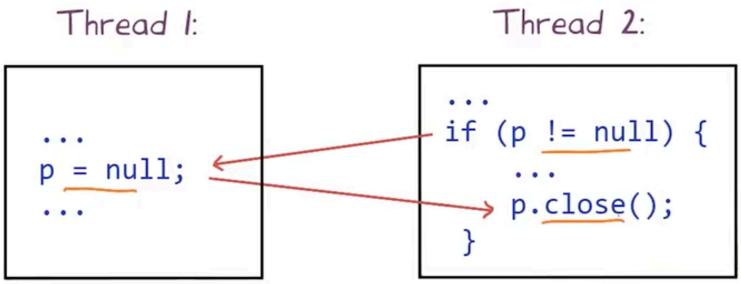
Depth of a concurrency bug

- Bug depth = the number of **ordering constraints** a schedule has to satisfy to find the bug

  An ordering constraint is a requirement on the ordering between 2 statements in different threads.
- Example 1 - the depth of the concurrency bug is one.



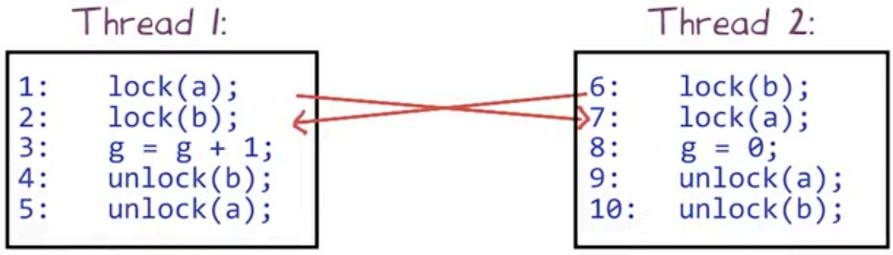
- Example 2 - the depth of the concurrency bug is two.



- Note that ordering constraints within a thread, don't count towards the bug depth, because a thread's control flow implicitly defines the constraints on the order in which statements are executed within a thread.
  Bug depth therefore only counts order dependencies across different threads.
- Observation exploited by Cuzz: many typical bugs have small depth

Concurrency bug depth

- The depth of the concurrency bug in the following example is 2.
  All ordering constraints needed to trigger the bug are (1,7) (6,2), where the notation (x,y) means statement x comes before statement y and multiple constraints are separated by a space.



Cuzz algorithm

```
Input:
  int n;          // # of threads
  int k;          // no. of steps  - guessed from previous runs
  int d;          // target bug depth - randomly chosen
State:
  int pri[] = new int[n];         // thread priorities
  int change[] = new int[d-1];    // when to change priorities
  int stepCnt;                    // current step count
```

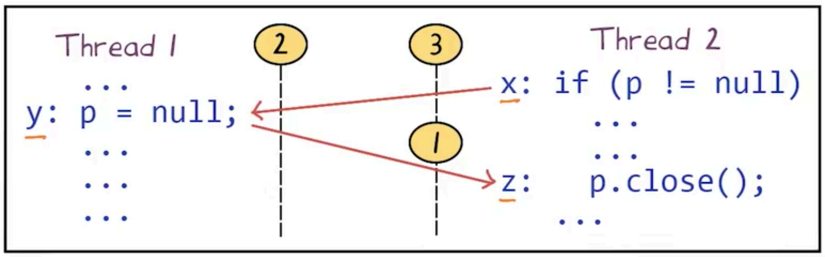
```
Initialize() {
  stepCnt = 0;
  a = random_permutation(1,n);
  for (int tid = 0; tid < n; tid++)
    pri[tid] = a[tid] + d;
  for (int i = 0; i < d-1; i++)
    change[i] = rand(1,k);
}
```

```
Sleep(tid) {
  stepCnt ++;
  if stepCnt == change[i] for some i
    pri[tid] = i;
  while (tid is not highest priority
         enabled thread)
    spin;
}
```

Probabilistic guarantee

- Given a program with:
  - n threads (~ tens)
  - k steps (~ millions)
  - Bug of depth d (1 or 2)
- Cuzz will find the bug with a probability of at least 1/(nk^(d-1)) in each run

Proof of guarantee (sketch)



- Probability (choose correct initial thread priorities) >= 1/n
- Probability (choose correct step to switch thread priorities) >= 1/k
- Probability (triggering bug) >= 1/nk
- Intuitively, for a bug of depth d, thread priorities are changed d-1 times.

  The probability of picking the right set of d-1 statements for changing priorities is at least 1/k^(d-1), so the probability of triggering a bug of depth d ought to be 1/nk^(d-1).

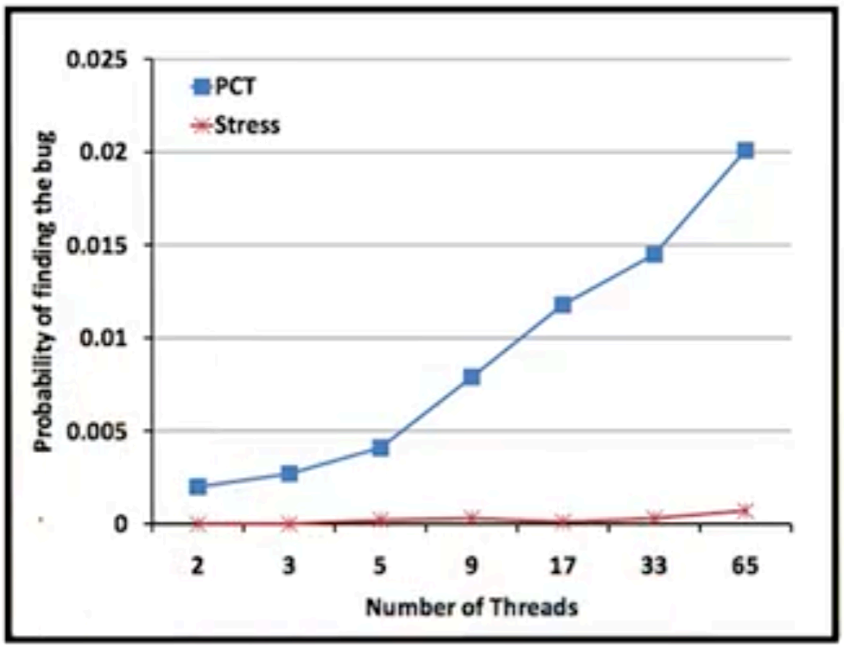
Measured vs. worst-case probability

- Worst-case guarantee is for **hardest-to-find** bug of given depth
- If bugs can be found in multiple ways, probabilities add up!
- Increasing number of threads help
  - Leads to more ways of triggering a bug
- Below is a plot showing the probability of finding a concurrency bug in a work stealing cube program using causes algorithm, called **PCT**, versus **stress testing**
  - requires the system beyond the maximum designed load
  - overloading tests the way the system “falls”
    - systems must not fail catastrophically
    - stress testing verifies unacceptable data loss or functionality

  as the number of threads in the program is increased.

  The interesting thing to note is that the probability of detecting the bug with stress testing is low and nondeterministic.

For example, with 2 threads, the worst case probability is 0.0003 whereas the measured is 0.002 which is an order of magnitude better than the worst case probability.



Cuzz case study

- Measure bug-finding probability of stress testing vs. Cuzz
  - Without Cuzz: 1 fail in 238 820 runs

    ratio = 0.000004817
  - With Cuzz: 12 fails in 320 runs

    ratio = 0.0375
- 1 day of stress testing = 11 seconds of Cuzz testing!

Cuzz: key takeaways

- Bug depth: useful metric for concurrency testing efforts
- Systematic randomization improves concurrency testing
- Whatever stress testing can do, Cuzz can do better
  - Effective in flushing out bugs with existing tests
  - Scales to large number of threads, long-running tests
  - Low adoption barrier

Random testing: pros and cons

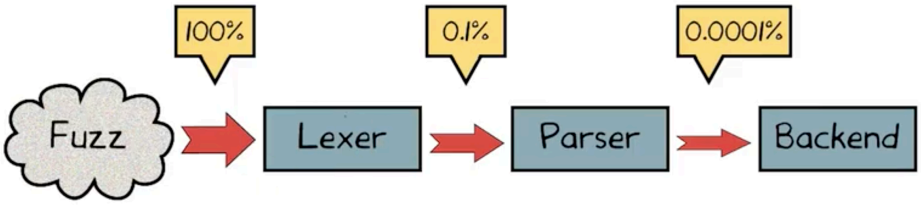
- Pros:
  - Easy to implement
  - Provably good coverage given enough tests
  - Can work with programs in any format

  - Appealing for finding security vulnerabilities
- Cons:
  - Inefficient test suite
  - Might find bugs that are unimportant
  - Poor coverage

Coverage of random testing

- Consider a compiler for, say, the Java programing language.

  Let's see what would happen if we were to test such a compiler program by feeding it random inputs. The Lexer will see all of these inputs and will hopefully reject almost all of them as invalid Java programs.



- The Lexer is very heavily tested by random inputs
- But testing of later stages is much less efficient

What have we learned?

- Random testing:
  - Is effective for testing security, mobile apps, and concurrency
  - **Should complement not replace systematic, formal testing**
  - Must generate test inputs from a reasonable distribution to be effective
  - May be less effective for systems with multiple layers (e.g. compilers)

Notă[1]