Random Testing

Introduction

1. Random testing
   - Theory behind random testing
   - Historical attempts
   - Demonstrate applications of random testing in the emerging domains of multithreaded apps and mobile apps

Random Testing (Fuzzing)

1. Random Testing: 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 randomly perturbs a specific aspect of the program (its input from the environment)
     - Mutation analysis randomly perturbs arbitrary aspects of the program

The Infinite Monkey Theorem

1. A monkey hitting keys at random on a typewriter keyboard will produce any given text, such as the complete works of Shakespeare, with probability approaching 1 as time increases
   - Aristotle

Random Testing: Case Studies

1. Random testing is a paradigm, as opposed to a technique that would work out of the box on any given program
   - For random testing to be effective, the inputs must be generated from a reasonable distribution
2. Case studies
   - UNIX utilities: U. of Wisconsin's Fuzz study
   - Mobile apps: Google's Monkey tool for Android
   - Concurrent programs: Cuzz tool from Microsoft

A Popular Fuzzing Study

1. Conducted by Barton Miller at University of Wisconsin
   - 1990: Command-line fuzzer, testing reliability of UNIX programs
     - Bombards the utilities 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

1. 1990: Caused 25-33% of UNIX utility programs to crash (dump state) or hang (loop indefinitely)
2. 1995: Systems got better… but not by much!
   - "Even worse 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

1. gets() funciton in C has no parameter limiting input length

- Programmer must make assumptions about structure of input
  - Becomes easy to trigger a buffer overflow by inputting a large amount of data
- 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
- Fuzzing can be effective at scouting memory corruption errors in C and C++ programs

Fuzz Testing for Mobile Apps

1. Monkey: Popular fuzzing tool for Android applications
   - Most indivisible and routine kind of input to a mobile app is a GUI event, such as a touch event at a certain pixel on the mobile device's display
     - Touch event results in the execution of the onClick function according to which pixel was touched
   - Monkey generates touch inputs at random locations on the mobile device's screen
     - Limited to the resolution of the display
     - Can simulate more complex events as well, such as an incoming phone call or a change in the user's GPS location

Generating Multiple Input Events

1. A single event is not sufficient for testing a mobile app
   - Typically, a sequence of events is needed
   - Monkey tool is used to generate a sequence of touch events separated by some amount of delay
     - Allow us to ensure that the app correctly handles any sequence of touch events it might receive
     - Also checks that a different amount of delay doesn't cause bugs

Generating Gestures

1. Gestures
   - Down, Move, Up simulates touching, dragging, and releasing
     - Greatly increases the number of tests we can run
     - Can test unlock screen or password

Monkey Events

1. Grammar of Monkey Events

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

Testing Concurrent Programs

1. Give the specification of a TOUCH event at pixel (89,215)
   - DOWN(89,215)
   - UP(89,215)
2. Give the specification of a MOTION event from pixel (89,215) to pixel (89,103) to pixel (37, 103)
   - DOWN(89,215)
   - MOVE(89,103)
   - MOVE(37,103)
   - UP(37,103)
3. Can adapt the testing paradigm to a domain to bias it towards generating common inputs

Testing Concurrent Programs

1. Testing sequential programs is considered with finding inputs that cause bugs
   - Concurrent programs have multiple threads executing simultaneously
   - Thread schedule also influences program result
     - Thread schedule is determined by OS and is not deterministic across different runs of the same program
   - Need to test different inputs and thread schedules
     - Dominant approach is to introduce random delays using sleep()
     - This is a form of fuzzing, but on the thread scheduler instead of program inputs

Cuzz: Fuzzing Thread Schedules

1. Cuzz 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 1

1. Bug Depth: The number of ordering constraints a schedule has to satisfy to find the bug
   - Thread 1: if(p != null) { p.close(); }
   - Thread 2: p = null
   - These threads have a bug depth of 2

Depth of a Concurrency Bug 2

1. The greater the bug depth, the more constraints on program execution must be satisfied in order to find the bug
   - Observation exploited by Cuzz: many typical bugs have small depth
     - Small test case hypothesis: If there is a bug, there will be some small input that will trigger the bug
   - Restrict search space by only looking for small bug counts

Concurrency Bug Depth

1. Thread 1:

```
lock(a);
lock(b);
g = g + 1;
unlock(b);
unlock(a);
```

2. Thread 2:

```
lock(b);
lock(a);
g = 0;
unlock(a);
unlock(b);
```

3. Specify the depth of the concurrency bug in the above example
   - 2
4. Specify all ordering constraints needed to trigger the bug
   - (1,7) (6,2)

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;
}
```

Cuzz Algorithm

Probabilistic Guarantee

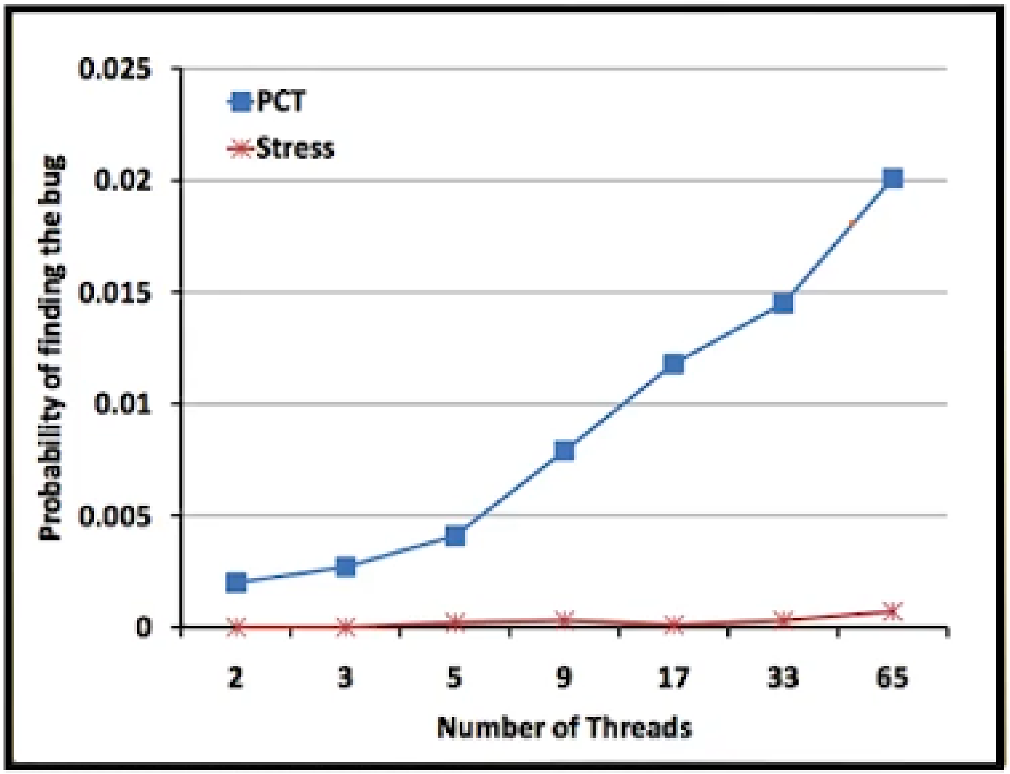
1. Given a program with:
   - n threads (tens)
   - k steps (millions)
   - bug of depth d (1 or 2)
2. Cuzz will find the bug with a probability of at least 1/(n * k^(d-1)) in each run
   - Worst case guarantee; Cuzz performs better in practice

Proof of Guarantee (Sketch)

1. Probability(choose correct initial thread priorities) >= 1/n
2. Probability(choose correct step to switch thread priorities) >= 1/k
3. Probability(triggering bug) >= 1/nk
4. Probability of triggering a bug of depth d: 1/k^(d-1)
5. Probability: 1/(nk^(d-1))

Measured vs. Worst-Case Probability

1. Worst-case guarantee is for hardest-to-find bug of given depth
   - Exactly one thread schedule to trigger bug
2. If bugs can be found in multiple ways, probabilities add up!
3. Increasing number of threads helps
   - Leads to more ways of triggering a bug
   - PCT is Cuzz's algorithm
     - Deterministic if given the same random seed



Probability of Triggering a Bug

Cuzz Case Study

1. 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

1. Bug depth: Useful metric for concurrency testing efforts
2. Systematic randomization improves concurrency testing
3. 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

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

- Appealing for finding security vulnerabilities

2. Cons:
    - Inefficient test suite
    - Might find bugs that are unimportant
    - Poor coverage

Coverage of Random Testing

1. Fuzz -> Lexer -> Parser -> Backend
    - Lexer sees all of the inputs and rejects invalid programs
    - 0.1% will reach the parser
    - 0.0001% will reach the backend
    - Lexer is very heavily tested by random inputs
    - Testing of later stages is much less efficient

Conclusion

1. 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 distribute to be effective
    - May be less effective for systems with multiple layers (e.g., compilers)