Random Testing

CS 6340

{HEADSHOT}

In the lesson on introduction to testing, we learned about the virtues of automated testing: it helps find bugs quickly, and it does not require writing or maintaining tests.

In this lesson, we will learn about one specific paradigm for automated testing: random testing. We'll see some of the theory behind why random testing works.
We'll also see some of the historical attempts at using random testing, where they went wrong, and the lessons we can learn from these attempts.

Most importantly, we will demonstrate applications of random testing in the emerging domains of mobile apps and multi-threaded programs. We will look at random testing in action in two different tools:

- the Monkey tool from Google for testing Android apps,
- and the Cuzz tool from Microsoft for testing multi-threaded programs.

Let's begin with formulating precisely what we mean by "random testing."

Random Testing (Fuzzing)

- 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

Random testing (also called "fuzzing," a terminology we'll use throughout the lesson) is a simple yet powerful testing paradigm.

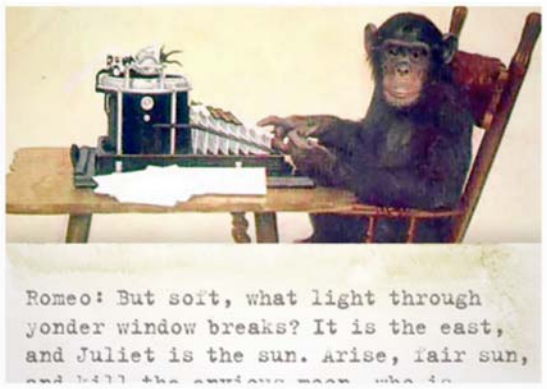
The idea is straightforward: we feed a program a set of random inputs, and we observe whether the program behaves "correctly" on each such input.

Correctness can be defined in various ways. For example, if a specification such as a pre- and post-condition exists, then we can check whether the execution satisfies the specification. In the absence of such a specification, we can simply check that the execution does not crash.

Note that the concept of fuzzing can be viewed as a special case of mutation analysis in the following sense. 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 a typewriter keyboard will produce any given text, such as the complete works of Shakespeare, with probability approaching 1 as time increases."



The motivation for random testing can be seen in the Infinite Monkey Theorem, which can be traced back to Aristotle. This theorem states that "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."

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 fuzz testing tool, and typing a given text is analogous to the monkey finding an input that exposes a bug in the program being tested.

You can learn more about the Infinite Monkey Theorem by following the link in the instructor notes.

[https://en.wikipedia.org/wiki/Infinite_monkey_theorem]

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

Random testing is a paradigm as opposed to a technique that will work out-of-the-box on any given program. In particular, for random testing to be effective, the test inputs must be generated from a reasonable distribution, which in turn is specific to the given program or class of programs.

We will look at three case studies next that highlight the effectiveness of random testing on three important classes of programs.

The first class of programs is UNIX utility programs that take command-line textual inputs. A famous case study applying random testing to such programs was conducted by the University of Wisconsin, which also coined the term "fuzzing".

The second class of programs is mobile apps. In particular, we will look at Google's Monkey tool for fuzz testing Android apps.

The third class of programs is concurrent programs -- programs that run multiple threads concurrently for higher performance on multi-core machines that are commonplace today. In particular, we will look at Microsoft's Cuzz tool for testing such programs.

A Popular Fuzzing Study

- Conducted by Barton Miller @ Univ of Wisconsin
- 1990: Command-line fuzzer, testing reliability of UNIX programs
  - Bombards 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

The first popular fuzzing experiment was conducted by Barton Miller at the Univ of Wisconsin. In the year 1990, his team developed a command-line fuzzer to test the reliability of UNIX utility programs by bombarding them with random data. These programs covered a substantial part of those that were commonly used at the time, such as the mail program, screen editors, compilers, and document formatting packages. This study focused only on fuzz testing command-line programs.

In the year 1995, his team expanded the scope of the experiment to also include GUI-based programs, notably those built on the windowing system X-Windows, as well as networking protocols and system library APIs.

In an even later study, the scope of the experiment was expanded further to include both command-line and GUI-based apps on operating systems besides UNIX that had begun gaining increasing prominence: Windows and Mac OS X.

The diversity of these applications alone highlights the potential of the random testing paradigm.

Follow the links in the instructor notes to read more about these studies.

[Main webpage: pages.cs.wisc.edu/~bart/fuzz/fuzz.html]

[The 1990 study: "An Empirical Study of the Reliability of UNIX Utilities" ftp://ftp.cs.wisc.edu/par-distr-sys/technical_papers/fuzz.pdf]

[The 1995 study: "Fuzz revisited: A re-examination of the reliability of UNIX utilities and services" ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-revisited.pdf]

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 worse is that many of the same bugs that we reported in 1990 are still present in the code releases of 1995."

Let's look at the aftermath of these studies.

In the 1990 study, a total of 88 utility programs were tested on 7 different versions of UNIX, with most utility programs being tested on each of the 7 systems. Two kinds of errors were discovered in 25-33% of the tested programs: crashes (which dump state, commonly called core dumps in UNIX lingo) and hangs (which involve looping indefinitely). These errors were reported to the developers of the programs.

In the 1995 study, it was discovered that the reliability of many of these systems had improved noticeably since the 1990 study, but perhaps surprisingly, many of the exact original bugs were still present despite being reported years earlier.

There is an important takeaway message here: many of the errors in the 1990 study were pertaining to input sanitization; developers have more pressing things to focus on than fixing input sanitization issues, such as adding new features, or fixing bugs that occur on correct inputs.

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

The UNIX fuzzing experiment did have a silver lining. Security attacks such as buffer overruns were becoming increasingly destructive. The 1995 study highlighted a security vulnerability that was at the heart of many of these attacks.

This vulnerability lies in using the gets() function in the C programming language, which reads a line from the standard input and stores it in an array of characters. However, the gets() function does not include any parameter that limits the length of the input that will be read. As a result, the programmer must make an implicit assumption about the structure of the input it will receive: for example, that it won't be any longer than the space allocated to the array.

Because C doesn't check array bounds, it becomes easy to trigger a buffer overflow by entering a large amount of data into the input. This can affect software reliability and security. In fact, in the 1995 fuzzing study, it was the second most common cause of crashes of the UNIX utility programs.

The solution was to deprecate usage of gets() in favor of the function fgets(), which has the same functionality but requires a parameter to limit the maximum length of the data that is read from stdin.

The main lesson here is that fuzzing can be effective at scouting memory corruption errors in C and C++ programs, such as the above buffer overflow. A human tester could then follow up on such errors to determine whether they can compromise security.

Fuzz Testing for Mobile Apps

```
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;
      ...
   }
}
```

One domain in which fuzz testing has proved useful is that of mobile applications -- programs that run on mobile devices such as smartphones and tablets. A popular fuzz testing tool for mobile applications is the Monkey tool on the Android platform.

To understand how the Monkey tool works, consider an example music player app on the Android platform. The code shown is only the app's code, written by the developer of the music player app, but it interacts with a large underlying Android framework that defines classes such as Activity and interfaces such as OnClickListener.

Whenever the user taps on one of the 6 buttons, the onClick() function is called by the Android framework. The function has an argument called 'target' that indicates which of the 6 buttons was clicked. An action corresponding to the button's functionality is taken, such as playing music, stopping music, and so on.

Let's see how fuzzing can be used to test this app.

Generating Single-Input Events

```
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]

The 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.

A TOUCH event results in the execution of the onClick() function according to which pixel is touched. For example, a TOUCH event at the pixel whose x-coordinate is 136 and y-coordinate is 351 results in a Play action, and a TOUCH event at the pixel whose x-coordinate is 136 and y-coordinate is 493 results in a Stop action.

The Monkey tool generates TOUCH events at random pixels on the mobile device's display, choosing the x- and y-coordinates within ranges appropriate to the mobile device being tested. For instance, on a device with a 480x800 pixel display, the x-coordinate is chosen in the range 0 to 480, and the y-coordinate is chosen in the range 0 to 800.

The Monkey tool is capable of generating many other kinds of input events which we shall not illustrate here, such as a key press on the device's keyboard, an input from the device's trackball, and so on. More generally, one can simulate even more sophisticated input events such as an incoming phone call or a change in the user's GPS location.

Black-Box vs. White-Box Testing

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.

Here is a sequence of three such events that tests important functionality of our music player app. The widgets clicked by these events are highlighted.

- The first TOUCH event clicks the eject button on the main screen, which pops up a dialog box where the user can either enter the location of an audio file to play, or use the default one shown.
- The second TOUCH event clicks the play button of the dialog box, which causes the app to return to the main screen and start playing the audio file.
- The third TOUCH event clicks the stop button on the main screen, which stops playing the audio file.

In summary, such multiple-input events allow us to ensure that the app correctly handles any sequence of touch events that it might receive. It further lets us ensure that the app continues to react correctly even with different amounts of delay between the events.

Generating Gestures

DOWN(x1,y1) MOVE(x2,y2) UP(x2,y2)

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

A simple gesture consists of a DOWN event at a pixel (x1,y1) (to simulate putting one's finger down on the display), then a MOVE event from (x1,y1) to a second pixel (x2,y2) (to simulate dragging one's finger across the display), followed by an UP event at pixel (x2,y2) (to simulate removing one's finger from the display).

The ability to generate gestures greatly expands the space of possible tests we can run on mobile apps. For example, we can test the drag-to-unlock functionality of an iPhone or the password entry feature of an Android phone.

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

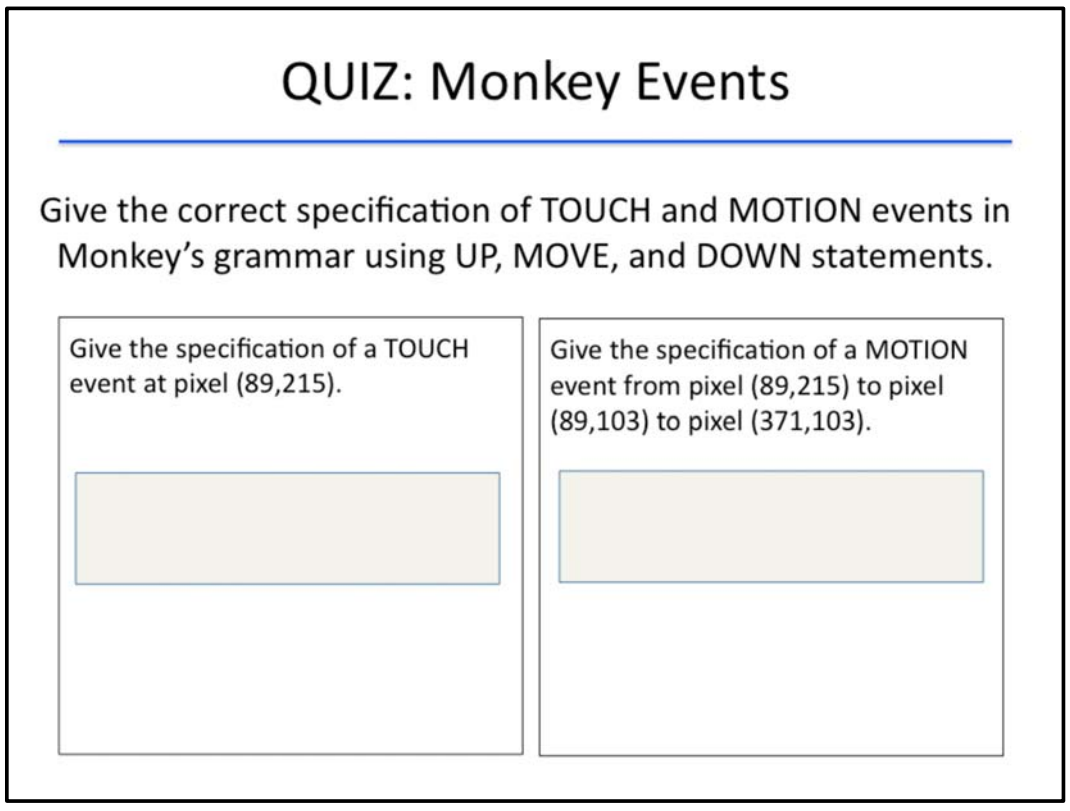
Having seen some example inputs that the Monkey random testing tool can generate, let's outline a grammar that systematically characterizes the possible inputs that the Monkey tool can generate.

Each test case, or input, is a sequence of some number of events. One kind of event that we covered is an action followed by x and y coordinates, which are picked randomly from predefined ranges corresponding to the dimensions of the display. Finally, each action is randomly chosen to be a DOWN event, a MOVE event, or an UP event.

Visit the link in the instructor notes to learn more about the Monkey tool, such as the other kinds of events it can generate.

http://developer.android.com/tools/help/monkey.html

Next, let's do a quiz to understand how individual touch events and sequences of touch events that we discussed earlier are covered by this grammar.



{QUIZ SLIDE}

Using the grammar we just defined for Monkey, for this quiz you will provide the specification for TOUCH and MOTION events on a mobile device.

In the first box, write down the specification of a TOUCH event at the pixel (89,215) using a sequence of UP, MOVE, and/or DOWN statements.

In the second box, do the same for a MOTION gesture which starts at (89,215), moves up to (89,103), and then moves left to (37,103).

QUIZ: Monkey Events

Give the correct specification of TOUCH and MOTION events in Monkey's grammar using UP, MOVE, and DOWN statements.

Give the specification of a TOUCH event at pixel (89,215).

```
DOWN(89,215) UP(89,215)
```

TOUCH events are a pair of DOWN and UP events at a single place on the screen.

Give the specification of a MOTION event from pixel (89,215) to pixel (89,103) to pixel (371,103).

```
DOWN(89,215) MOVE(89,103)
MOVE(37,103) UP(37,103)
```

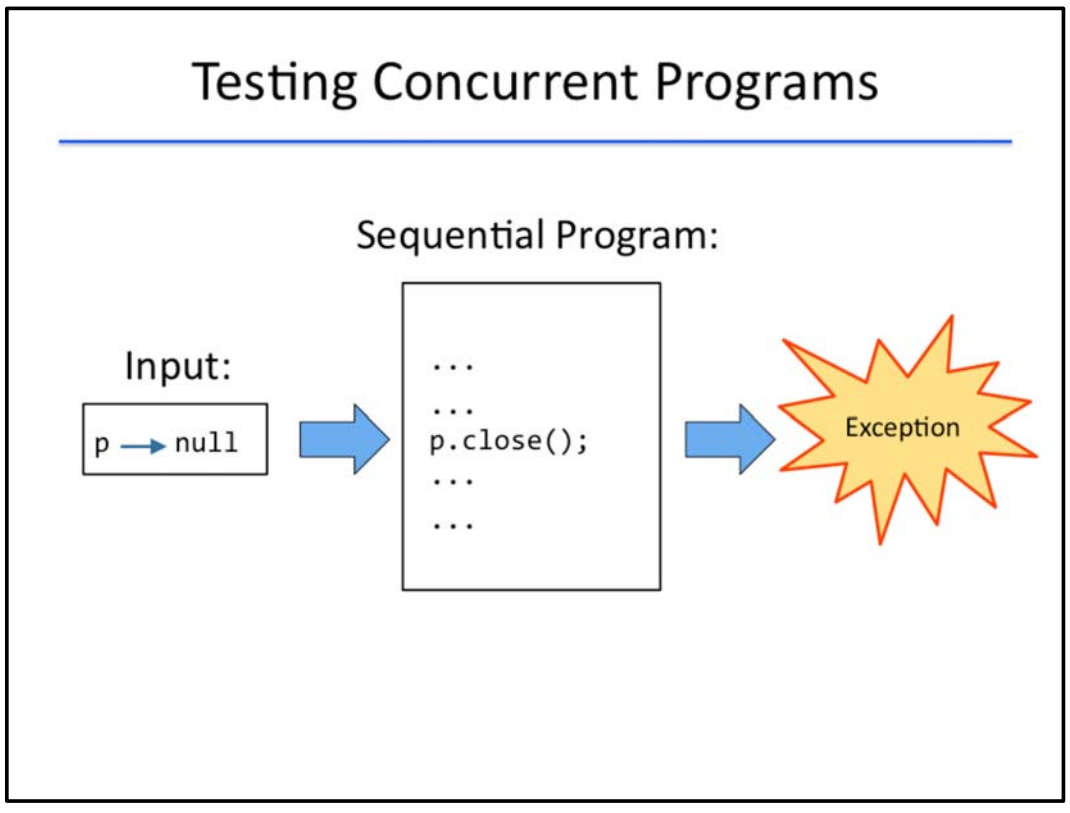
MOTION events consist of a DOWN event somewhere on the screen, a sequence of MOVE events, and an UP event.

{SOLUTION SLIDE}

A TOUCH event at a single pixel will be just a pair of DOWN and UP events at that pixel. So the answer to the first question is DOWN(89,215) UP(89,215).

A MOTION event consists of a DOWN event at the start pixel, a sequence of MOVE events to each intermediate pixel along the path of motion, followed by an UP event at the last pixel that we moved to. In this case, the answer to the second question is DOWN(89,215) MOVE(89,103) MOVE(37,103) UP(37,103).

Because TOUCH and MOTION events are far more useful in practice than arbitrary DOWN, MOVE, and UP events, the Monkey tool directly generates TOUCH and MOTION events as opposed to individual DOWN, MOVE, and UP events. This is a simple example of how the random testing paradigm can be adapted to a domain to bias it towards generating common inputs.



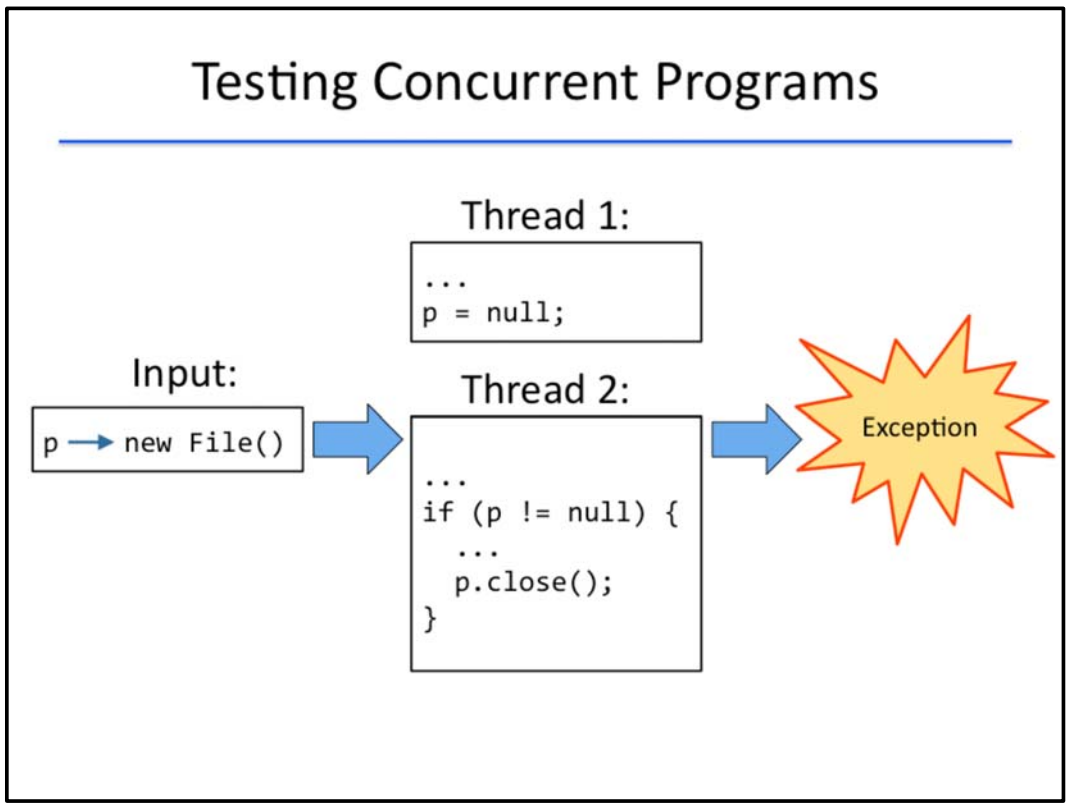
Another important domain in which random testing is exceedingly useful is the testing of concurrent programs.

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.

For instance, consider the following sequential Java program that takes as input a File handle p and calls function p.close().

An input under which this program would crash is a null File handle.

We will learn about techniques that automatically discover such inputs later in the course.



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.

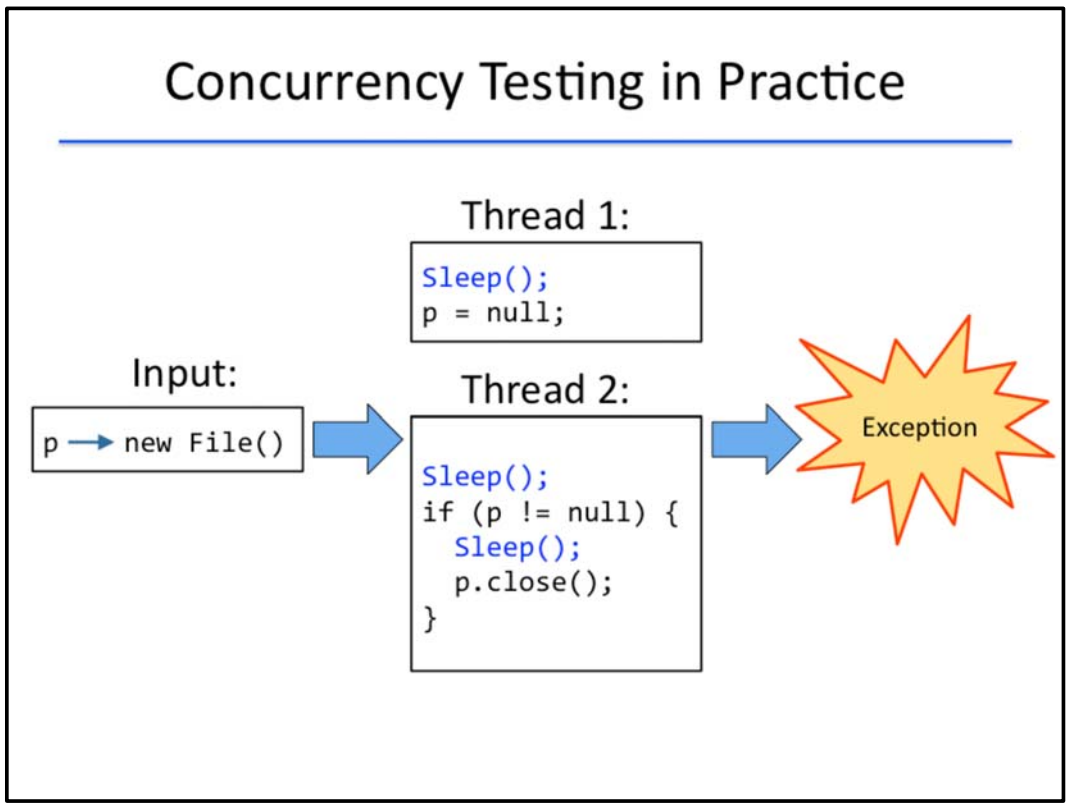
In a concurrent program, a bug is triggered not only under a specific program input, but also under a specific thread schedule, which may be viewed as the order in which the computation of different threads is executed. The thread schedule is typically dictated by the scheduler of the underlying operating system, and is non-deterministic across different runs of the concurrent program even on the same input. Therefore, although a particular run of a concurrent program on a given input succeeds, another run of the program on the same input might crash, because of a different thread schedule used by the underlying scheduler.

To be more concrete, consider this concurrent program that consists of two threads and takes as input a File handle p. Suppose we wish to test this program using a non-null File handle as input. The resulting execution may succeed or crash depending upon the thread schedule.

If the non-null check in Thread 2 is executed first, followed by the assignment of null to p in Thread 1, followed by the p.close() statement in Thread 2, then the program will throw a null pointer exception at this statement.

However, if the thread schedule were different (for example, if the entirety of Thread 2 finished execution before p were assigned null by Thread 1, or if p were assigned null by Thread 1 before Thread 2 executed), then the bug would not be triggered.

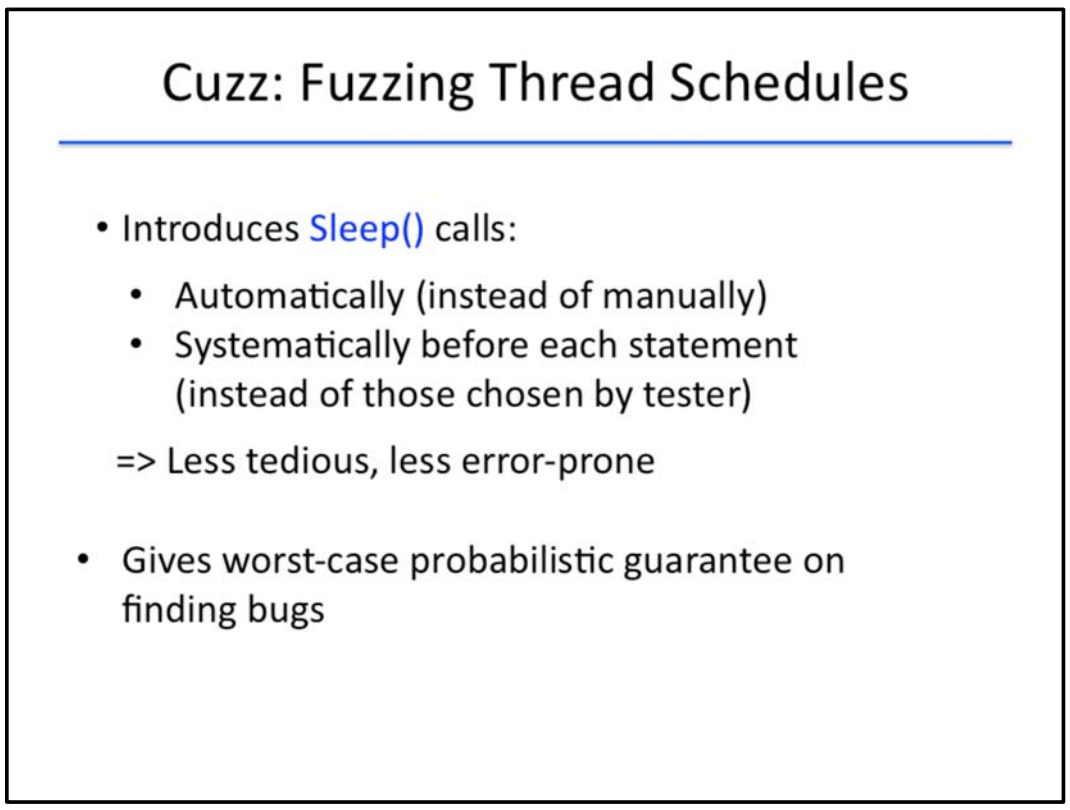
In summary, uncovering bugs in concurrent programs requires not only discovering specific program inputs, but also specific thread schedules. In this section, we will focus on techniques for finding thread schedules that trigger bugs on a given input.



The predominant approach to testing concurrent programs today is to introduce random delays, indicated by the calls to a system function Sleep() in our example program. These delays serve to perturb the thread schedule: a Sleep() call has the effect of lowering the priority of the current thread, causing the underlying thread scheduler to schedule a different thread.

Making these delays random has the effect of attempting different thread schedules in the hope of finding one that triggers any lurking concurrency bug.

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



The idea behind Cuzz is to automate the approach of introducing calls to Sleep() in order to find concurrency bugs more effectively.

In a realistic program, there is a large number of possible places at which to introduce Sleep() calls.

Using Cuzz, the calls to Sleep() are introduced automatically instead of manually by a human tester, and they are introduced systematically before each statement in the program instead of only those chosen by a human tester.

The resulting process is therefore less tedious and less prone to mistakes.

More significantly, Cuzz even provides a good probabilistic guarantee on finding concurrency bugs through its simple approach of fuzzing thread schedules.

Next, we'll examine the basics behind the algorithm Cuzz uses to systematize scheduler fuzzing.

You can find more details about Cuzz in the resources listed in the instructor notes on this page.

http://research.microsoft.com/en-us/projects/cuzz/

Depth of a Concurrency Bug

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

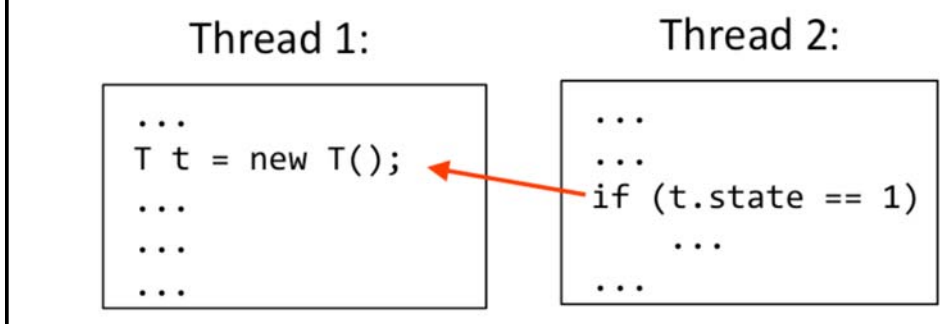
First, let's introduce some terminology.

The *depth* of a concurrency bug is the number of *ordering constraints* that a thread schedule has to satisfy in order for the bug to be triggered.

An ordering constraint is a requirement on the ordering between two statements in different threads.

Bug Depth: Example 1

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



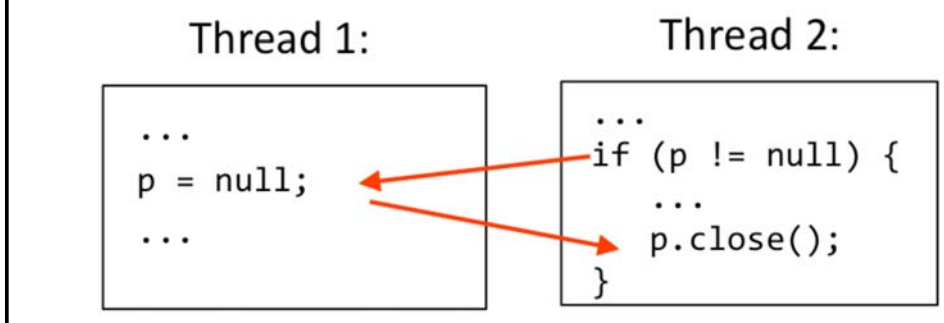
For example, let's look at the following concurrent program.

If this line in Thread 2 is executed before this line in Thread 1, then an exception will be thrown because Thread 2 will be attempting to dereference an undefined variable t.

Since there is one constraint on the ordering of statements across threads, we say the depth of this concurrency bug is 1.

Bug Depth: Example 2

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



Let's look at another example. Here's the concurrent program we looked at earlier in the lesson. The concurrency bug we found in it has a depth of 2: triggering the bug requires the non-null check in Thread 2 to be executed before the null assignment in Thread 1, and it requires this null assignment to be executed before the call to close() in Thread 2.

Note that ordering constraints within a thread don't count towards the bug depth, because a thread's control flow implicitly defines constraints on the order in which statements are executed within a thread.

Bug depth therefore only counts order dependencies across different threads.

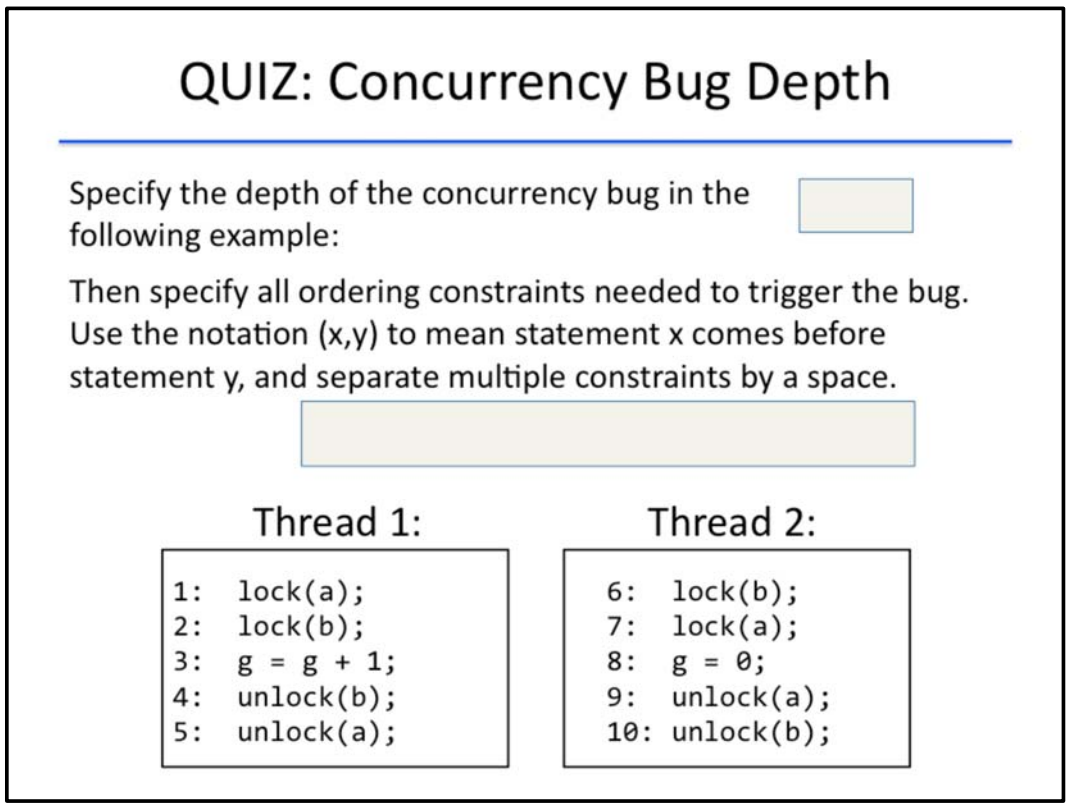
Depth of a Concurrency Bug

- Bug Depth = the number of ordering constraints a schedule has to satisfy to find the bug
- Observation exploited by Cuzz: bugs typically have small depth

The greater the bug depth, the more constraints on program execution need to be satisfied in order to find the bug. This in turn means that more things have to happen "just right" for the bug to trigger.

The observation exploited by Cuzz is that concurrency bugs typically have a small depth. In other words, most concurrency bugs will not have a large number of prerequisites on the thread schedule in order to occur.

This is a form of the "small test case" hypothesis that we will see throughout the course: if there is a bug, there will be some small input that will trigger the bug. Therefore, when we run Cuzz, we'll restrict our search space by only looking for bugs of small depth. This will give us a good chance to find all the bugs without needing to run too many test cases.



{QUIZ SLIDE}

To check your understanding about bug depth, please do the following quiz. In the code displayed here, there is a concurrency bug.

- First, enter the depth of the concurrency bug in the box at the top of the slide.
- Then, enter the ordering constraints needed to trigger the concurrency bug. Use the notation open-parenthesis x comma y close-parenthesis to denote the fact that statement x must be executed before statement y. If you need to enter multiple order constraints, separate them by a space.

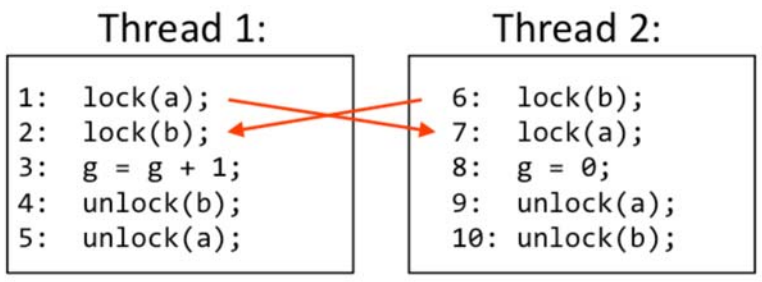
Note that the lock() method acquires a lock on the specified variable while the unlock() method releases the lock on the specified variable. Locks are a means of enforcing mutual exclusion between threads: at most one thread can hold a lock on a given variable at any instant. A thread that attempts to acquire a lock that is held by another thread blocks and cannot execute any statements until the other thread releases the lock.

QUIZ: Concurrency Bug Depth

Specify the depth of the concurrency bug in the following example: 2

Then specify all ordering constraints needed to trigger the bug. Use the notation (x,y) to mean statement x comes before statement y, and separate multiple constraints by a space.

(1,7) (6,2)



{SOLUTION SLIDE}

Let's look at the solution. Whenever two threads running in parallel are allowed to hold multiple locks, there's a potential for both threads to block indefinitely, if the threads acquire the locks in different order. This classic concurrency bug is called a deadlock: a situation in which neither thread can execute any more statements because the other thread is holding a lock needed to make progress.

The concurrency bug in this program is a deadlock of depth two. It is triggered if:

- Statement 1 in Thread 1 is executed before Statement 7 in Thread 2, and
- Statement 6 in Thread 2 is executed before Statement 2 in Thread 1.

Any thread schedule that satisfies these two ordering constraints will prevent either thread from progressing on beyond the second statement in each thread, resulting in the program hanging.

Cuzz Algorithm

```
Input:
  int n;             // # of threads
  int k;          // # 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)
    ;
}
```

Let's look at the algorithm underlying the Cuzz tool to find concurrency bugs such as these in an automated fashion.

Let *n* be the number of threads that the program creates on a given input, let *k* be an approximation of the number of steps or statements that the program executes on that input, and suppose we randomly set our bug depth parameter to be *d*.

The algorithm calls the Initialize() function once at the start of the program, and the Sleep() function before executing each instruction in each thread.

The Initialize() function randomly assigns each of d+1 through d+n as the priority value of one of the n threads. We will see why it does not use lower priority values 1 through d momentarily.

Triggering a bug of depth d requires d-1 changes in thread priorities over the entire execution. So the Initialize() function picks d-1 random priority change points $k_1, \ldots, k_{d-1}$ in the range [1, k]. Each such priority change point k_i has an associated priority value of i. This is where the lower priority values 1 through d get used.

The underlying thread scheduler schedules the threads by honoring their assigned priorities in the array pri[]. When a thread reaches the i-th change point (that is, when it executes the k_i-th step of the execution), its priority is changed, that is, lowered, to i. This is done in the call to the Sleep() method before each instruction in each thread.

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

$$\frac{1}{n\,k^{d-1}}$$ in each run

We can now state the probabilistic guarantee that Cuzz provides on finding concurrency bugs through its simple approach of fuzzing thread schedules.

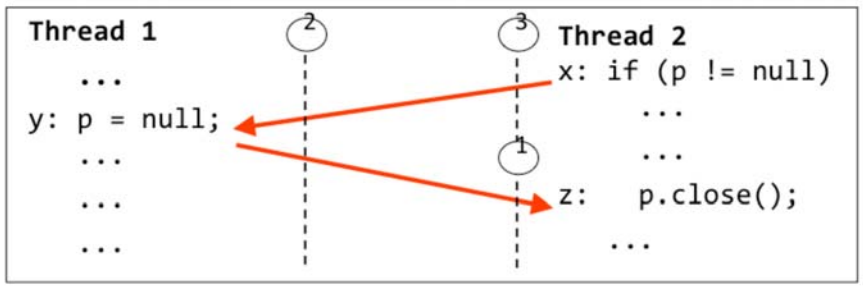
Suppose there is a concurrency bug of depth d in a program with n threads and taking k steps. (Typically n will be on the order of tens and k will be on the order of millions while d will a small number like 1 or 2.)

Then Cuzz will find the bug with a probability of at least 1/(n * k^(d-1)) per run. In other words, we expect to find the bug once after n*k^(d-1) runs, which is a tractable number of runs for n and k in these ranges.

More significantly, this is a worst-case guarantee, and as we shall see shortly, Cuzz does even better in practice, in that it finds concurrency bugs with far fewer runs than what is predicted by this guarantee.

First let's look at a sketch of the proof of this probabilistic guarantee.

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)

Let's use this program again as an example to demonstrate why the probabilistic guarantee holds.

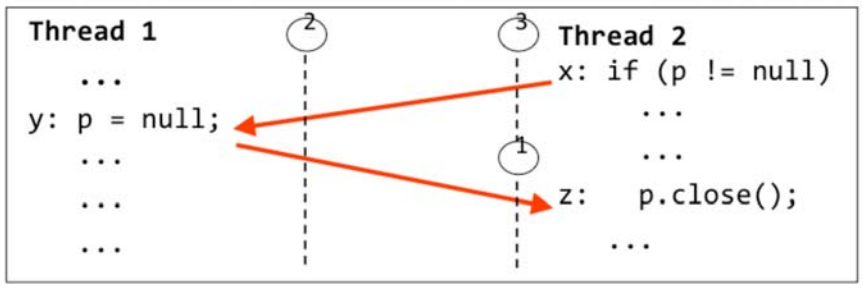
To trigger the bug here, Statement X must execute before Statement Y, and Statement Y must execute before Statement Z.

This order is possible if Thread 1 starts with a lower priority than Thread 2, ensuring that Statement X executes before Statement Y. (For example, if Thread 1 starts with a priority of 2 and Thread 2 starts with a priority of 3.)

Because Cuzz randomly assigns initial thread priorities, the probability that Thread 1 has a lower priority than Thread 2 is one-half.

However, in general, if the above example had n threads, Statement X would only be guaranteed to execute before Statement Y if Thread 1 is assigned the lowest priority initially. (Even if Thread 2 had a higher priority than Thread 1, another thread could block Thread 2's progress by locking p, for example, allowing Thread 1 to execute before Thread 2 could execute the if-statement.) The probability that Thread 1 has the lowest priority initially is 1/n.

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)

Next, to ensure that Statement Y executes before Statement Z, the priority of Thread 2 should become lower than Thread 1 after statement X is executed. This can be achieved if the thread priorities are changed after Statement X is executed. For example, before executing Statement Z, thread 2 is assigned a lower priority of 1.

As Cuzz picks the statements where the thread priorities are changed uniformly over all statements, the probability of picking somewhere between Statement X and Statement Z to change the priorities is at least 1/k (recall that k is the number of statements executed by the program).

Because these random choices were made independently of one another, the overall probability of triggering a bug is therefore 1/n * 1/k = 1/nk.

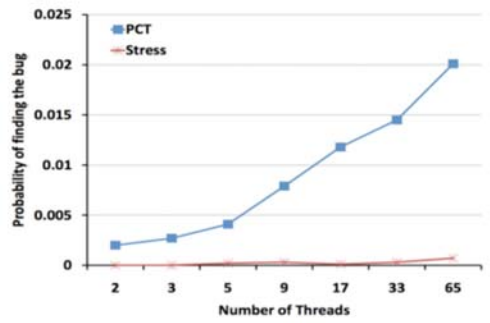
Intuitively, for a bug of depth d, thread priorities are changed (d-1) times; that is, Cuzz needs to pick (d-1) statements in the program. The probability of picking the right set of (d-1) statements for changing priorities is at least $1/k^{(d-1)}$, so the probability of triggering a bug of depth d ought to be $1/nk^{(d-1)}$.

This proof sketch does not account for the possibility of multiple priority changes along with arbitrary synchronization and control flow statements. You can see the full proof in Section 3 of the paper linked in the instructor notes.

http://research.microsoft.com/pubs/118655/asplos277-pct.pdf

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 helps
  - Leads to more ways of triggering a bug



Even with this guaranteed lower bound on probability, Cuzz often finds bugs even more commonly in practice. There are several reasons why this is the case.

- The theoretical lower bound is only for the hardest-to-find bug of a given depth; that is, a bug that has exactly one thread scheduling that causes it to trigger.
- If a bug can be found via multiple thread schedules, then the probability of finding that bug is the sum of the probabilities that each of those schedules is chosen.
- And, while the theoretical lower bound decreases as the number of threads increases, in practice we see that the probability of finding a bug *increases* as the number of threads increases.
- This is because having more threads typically means there are more ways to trigger a bug.

Let's look at some real measurements that depict this phenomenon.

Here is a plot showing the probability of finding a concurrency bug in a work-stealing queue program using Cuzz's algorithm, denoted PCT, versus stress testing 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 is nondeterministic.

On the other hand, for any given number of threads, Cuzz has a higher probability of detecting the bug, and it is also deterministic when given the same random seed, which helps with debugging and bug-fixing efforts once the bug is detected. Furthermore, as the number of threads increases, the probability with which Cuzz finds the bug increases. Finally, for any given number of threads, this measured probability is much better than the worst-case probability. For example, with 2 threads, the worst-case probability is 0.0003 whereas the measured is 0.002, an order of magnitude better.

Cuzz Case Study

Measure bug-finding probability of stress testing vs. Cuzz

- Without Cuzz: 1 Fail in 238,820 runs
  - ratio = 0.000004187
- With Cuzz: 12 Fails in 320 runs
  - ratio = 0.0375

1 day of stress testing = 11 seconds of Cuzz testing!

To better appreciate the amount of resources needed to find concurrency bugs using Cuzz vs. stress testing, here is a case study that the developers of Cuzz conducted to find a concurrency bug in a certain program.

Without Cuzz, that is, using stress testing, the bug was triggered only once in over 238,000 runs, giving a mere probability of 0.000004187 for finding this bug using stress testing.

On the other hand, using Cuzz, the bug is triggered 12 times in just 320 runs, giving a dramatically higher probability of 0.0375. It took an entire day to execute the 238,000 runs using stress testing compared to a mere 11 seconds using Cuzz!

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

Let's review the key points you should take away from this section on concurrency testing.

Bug depth, which is the number of statement ordering constraints required to trigger a concurrency bug, is a useful metric for concurrency testing efforts. In particular, focusing on bugs with very small depths is likely to be enough to cover most of a program's concurrency errors.

Systematic randomization improves concurrency testing. Fuzzing thread scheduling, as Cuzz does, gives us a guaranteed probability of finding a bug of a given depth (should one exist).

Finally, whatever traditional stress-testing can do, the Cuzz concurrency testing tool can do better. It is effective in flushing out concurrency bugs using existing tests: it simply needs to fuzz the thread schedule when running each of those tests. It can scale easily to a large number of threads and long-running tests. And it has a low barrier to adoption as it is fully automated: it neither requires users to provide any specifications nor make any modifications to the program.

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

While randomization is a highly effective paradigm for testing, it has its own set of tradeoffs that must be considered when choosing whether to apply it for testing a given program. You'll notice that some of these tradeoffs are similar to those we described for black-box testing in the lesson on introduction to testing.

Random testing is easy to implement, and as the number of tests increases, the probability that some test case covers a given input approaches 1.

Random testing also can be used with programs in any format: unmodifiable ones, as well as programs in managed code, native code, or binary code.

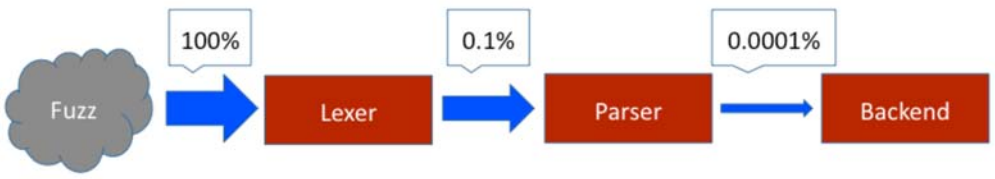
And random testing enhances software security and software safety because it often finds odd oversights and defects which human testers might fail to find and even careful human test designers might fail to create tests for.

On the other hand, random testing might result in a bloated test suite with inputs that redundantly test the same piece of code.

Additionally, as we saw with Unix utility case study, the bugs that fuzzing catches might be unimportant bugs: ones that are rarely triggered or have benign side-effects in the program's practical use.

And, despite the fact that any given input will be tested with probability approaching 1 given enough tests, in practice random testing can have poor coverage. Let's take a look at an example of this behavior.

Coverage of Random Testing



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

Consider a compiler for say the Java programming 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. So perhaps only one thousandth of these inputs will pass the lexer and reach the parser. And perhaps only one thousandth of the inputs reaching the parser will pass through to the backend of the compiler.

Thus, while random testing heavily tests the lexer, it is much less efficient in testing the later stages of the compiler.

In the next lesson, you will be introduced to different ways of generating test inputs that would be more appropriate for testing different parts of complex systems like this compiler.

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)

Before we conclude, let's summarize some of the key points about random testing you should take away from this lesson.

Random testing is a powerful technique in certain domains, including testing mobile apps and programs running in parallel.

However, random testing should be used to complement rather than replace systematic and formal testing. As we have seen, random testing can cover many cases very quickly, but it might not cover cases that are more interesting to developers. Therefore, we cannot solely use fuzzing to test our software.

Additionally, in order for random testing to be effective, the test inputs must be generated from a reasonable distribution. While a uniform distribution of strings might cover a wide range of program paths for a string utility program, they would likely only test a very limited subset of the code for a parser in a compiler for Java programs. It's much harder to come up with a reasonable distribution of test inputs that will effectively test the parser's code paths.

In the next lesson, you'll learn more techniques for automated test generation that are more directed and systematic than random testing.