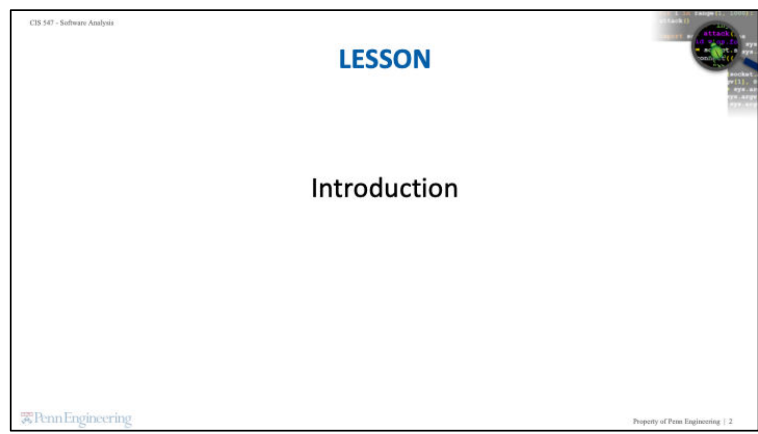
In the module on software specifications, we learned about the virtues of automated testing: it helps find bugs quickly, and it does not require writing or maintaining tests.

In this module, we will learn about one specific paradigm for automated testing: random testing, also known as fuzzing. 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 that were learnt from these attempts, culminating into powerful, modern day general-purpose fuzzers.

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.



SEGMENT

Motivation and Background

Random Testing (Fuzzing)

- Idea: Feed random inputs to a program
- Observe whether it behaves "correctly"
  - Execution satisfies given specification
  - Or simply doesn't crash
    - A simple specification
- Special case of mutation analysis

Random testing (also called "fuzzing," a terminology we'll use throughout the module) 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."

Romeo: But soft, what light through yonder window breaks? It is the east, and Juliet is the sun. Arise, fair sun,

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:

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

SEGMENT

The First Fuzzing Study

CIS 547 - Software Analysis

The First 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

PennEngineering

Property of Penn Engineering | 7

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 lecture handout to read more about these studies.

[Main webpage: http://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]

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.

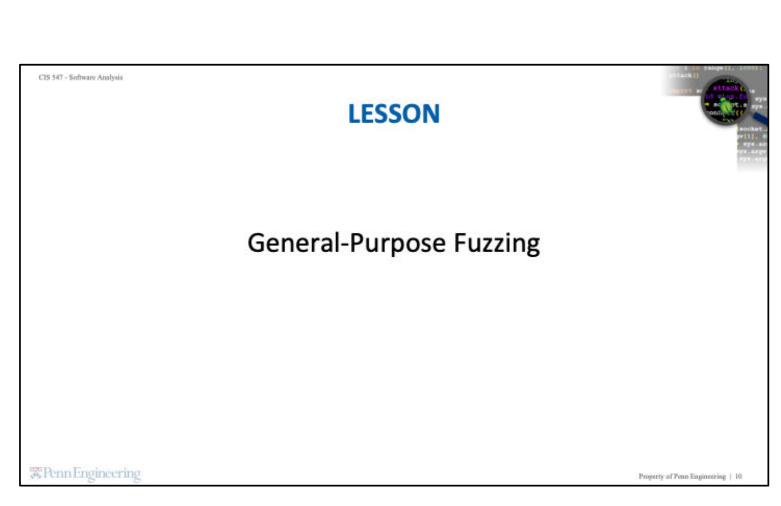
The UNIX fuzzing experiment did have an important silver lining. Security attacks such as buffer overruns were becoming increasingly destructive. The 1995 study highlighted a security vulnerability that was at the heart of many of these attacks.

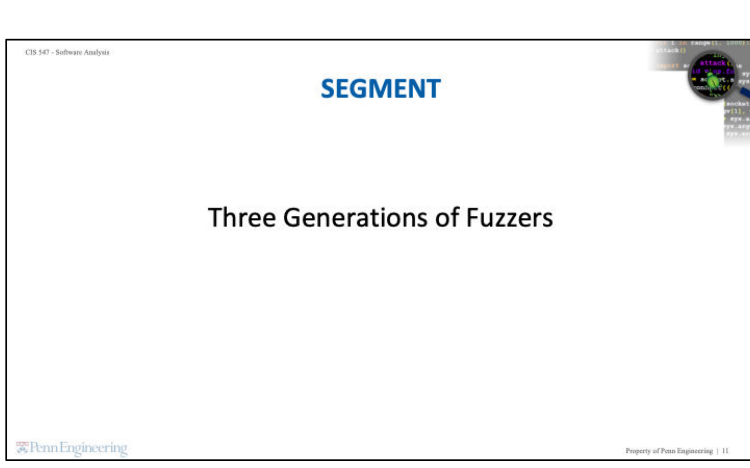
This vulnerability lies in using the gets() function in the C programming language, which reads a line from the standard input and stores it in an array of characters. However, the gets() function does not include any parameter that limits the length of the input that will be read. As a result, the programmer 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.

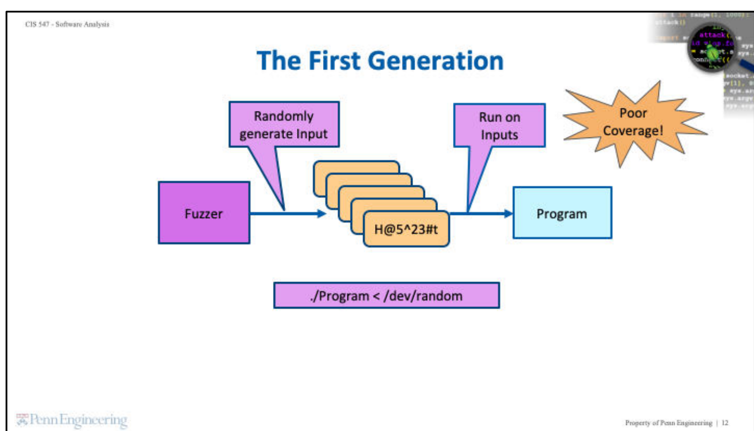
Because C doesn't check array bounds, it becomes easy to trigger a buffer overflow by entering a large amount of data into the input. This can affect software reliability and security. In fact, in the 1995 fuzzing study, it was the second most common cause of crashes of the UNIX utility programs.

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/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.


CIS 547 - Software Analysis
LESSON
General-Purpose Fuzzing
Penn Engineering
Property of Penn Engineering | 10

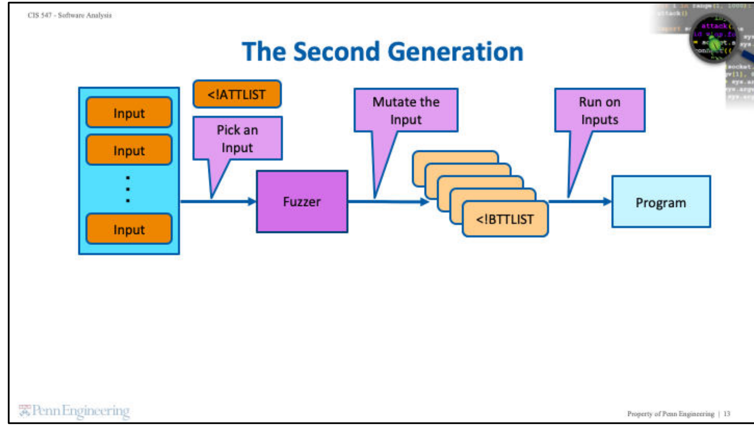

CIS 547 - Software Analysis
SEGMENT
Three Generations of Fuzzers
Penn Engineering
Property of Penn Engineering | 11



Since the original fuzzing study by Barton Miller, many fuzzers have come and gone. They can broadly be classified into three generations. Each generation built upon and further improved the previous generation, culminating in the sophisticated and robust fuzzers that are widely used today.

The first generation, exemplified here, is the simplest. It encompasses Miller's UNIX fuzzing study. Inputs are generated in an entirely random fashion and fed to the target program. This early generation of fuzzers suffered from poor performance.

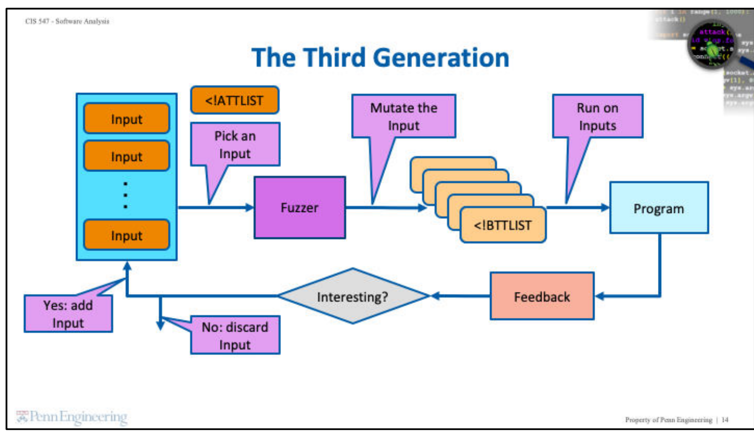
Performance, here, refers to the time between starting the fuzzer and finding a crashing input. Because the inputs are entirely random, there is no guarantee that they conform to the format expected by the target program. Most applications include input sanitization which throw away any invalid inputs before executing the majority of the program that we are interested in fuzzing. Furthermore, even if an input happens to conform to the input specifications, it is not guaranteed to explore a new or interesting path through the program. The same path could be taken millions of times by millions of unique inputs. This leads to poor code coverage, especially in large codebases with many possible paths of execution.



In order to combat the drawbacks of the previous generation, a second generation of fuzzers was developed.

The main insight of this generation of fuzzers was to begin with a set of normal inputs, called seed inputs. This provides a starting template for the fuzzer to mutate, creating random inputs, which are then fed to the program similar to in the first generation. Because most inputs are a valid format and are not thrown away immediately by input sanitization, there is a significant improvement in performance over the first generation. However, we still have the problem of poor code coverage – there is nothing preventing the fuzzer from repeating the same path many times with many different inputs.

This technique was still not sufficient in fuzzing large, complex code bases, due to the many possible execution paths.



In order to combat poor code coverage, the third generation of fuzzers selectively grows the set of seed inputs to explore new execution paths. This technique uses a feedback loop to "learn" from the input that we just executed and prevents the fuzzer from repeating mistakes.

Like the previous generation, we start with a set of seed inputs that are randomly mutated to trigger an error. However, rather than keeping our set of inputs static, we add new seeds based on the execution of the target program on previous inputs.

All modern fuzzers use this architecture to selectively grow the set of seed inputs based on a feedback loop. This is considered an "evolutionary algorithm" as it uses mechanisms inspired by biological evolution. The basic idea is to evaluate the "fitness" of each seed in our input set, and select the "fittest" inputs for mutation to create new inputs. Here, the term "fitness" refers to how "interesting" the input is in some sense. Interesting can be defined as a run that uncovers new paths of execution, longer execution traces, or simply an input that is valid according to the target specifications.

For example, suppose we define "interesting" as an input that achieves new code coverage compared to all previous attempted inputs. This indicates that it likely opened up a new part of the program's state space for exploration and we should latch on to it for further mutation.

Two of the most popular fuzzers, AFL and Libfuzzer, can be viewed as instances of this framework. We will cover their advantages and differences in this module.

Although this is the last generation of fuzzers we will cover, the problem of efficient and robust random testing is far from solved. Fuzzing remains an important tool in industry and thus is under realistic time and coverage constraints. Because of this, random testing remains an active area of research. Some exciting research directions have included humans or AI to help make difficult decisions such as which input to pick, how to mutate it, how many mutants to generate, what kind of feedback to use, or how to decide whether a run is interesting.

What Kinds of Bugs can Fuzzing Find?

- Memory errors
  - Spatial (e.g. out-of-bounds access) and temporal (e.g. use-after-free)
- Other undefined behaviors
  - Integer overflow, divide-by-zero, null dereference, uninitialized read, …
- Assertion violations
- Infinite loops (using timeout)
- Concurrency bugs
  - Data races, deadlocks, …

Now let's take a moment to look at the kinds of bugs that fuzzers can find. Most obviously, fuzzing can be used to detect violations of general program safety properties such as buffer overflows, use after free, or out of bounds accesses. This class of bugs has a significant impact on security of the application and may lead to dangerous vulnerabilities.

Additionally, random testing can be used to uncover bugs caused by undefined behavior including uninitialized variables, integer overflow, divide-by-zero etc. These bugs, however, may not cause a program to explicitly crash. A common solution to this is to tie in runtime monitors, called sanitizers, that will exit immediately upon a memory or address error.

However, fuzzing is not limited to general program safety properties. It can find bugs specific to program's functionality specified via assertions. The assertion serves as a program specification which random testing attempts to violate. Fuzzers can even find violations of liveness properties like infinite loops by using a timeout mechanism.

Lastly, random testing can find aggravating and notoriously difficult to reproduce bugs in concurrent programs. The popular fuzzing tool Cuzz was created to find reproducible concurrency bugs in large real world programs. We will discuss Cuzz more deeply later in this module.

SEGMENT

Pros and Cons of Random Testing

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 code 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 module on software specifications.

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.

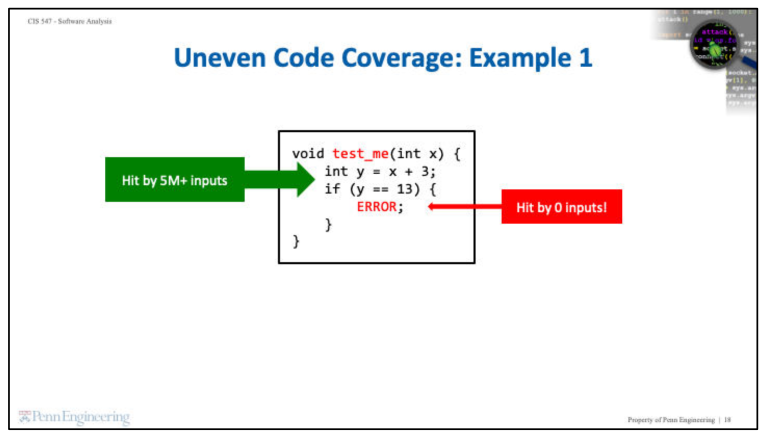
And random testing enhances software security and software safety because it often finds odd oversights and defects which human testers might fail to find and even careful human test designers might fail to create tests for.

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 the 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 code coverage. Let's take a look at two examples of this behavior next.

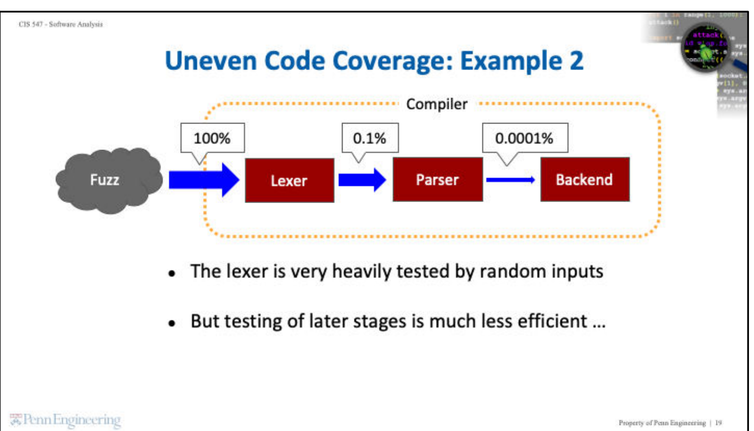


Consider the following program which adds the constant 3 to an input, x, and saves it to the variable y. An error state can be reached if y is equal to 13. It is clear to a white-box tester that an input of x = 10 will cause the program to enter the conditional and reach the error state. However, random testing is similar to black-box testing in that we do not consider the program contents when generating inputs. Because the number of inputs that trigger the error is much smaller than the total number of possible inputs, we may never find the input that causes this bug using a black-box approach.

More concretely, assuming an integer is 32 bits, there is only a 1 in 2^32 (or, one in over 4 billion) chance of generating an input that exercises the then branch to reach the error state.

In order to empirically validate this claim, you can run a state-of-the-art fuzzer called AFL on this segment of code. We will discuss AFL, its uses, and underlying techniques later in this module. In our sample run, the AFL fuzzer generated over 5 million inputs that did not satisfy the condition (x + 3 = 13) and thus, the bug was not found.

For many real world programs, the space of inputs is too large to ensure code coverage over all paths with random black-box fuzzing. Even feedback-directed fuzzing, which is grey-box in that it monitors executions, is unable to do any better. In fact, recall that AFL is a feedback-directed fuzzer belonging to the third generation of fuzzers we just discussed.

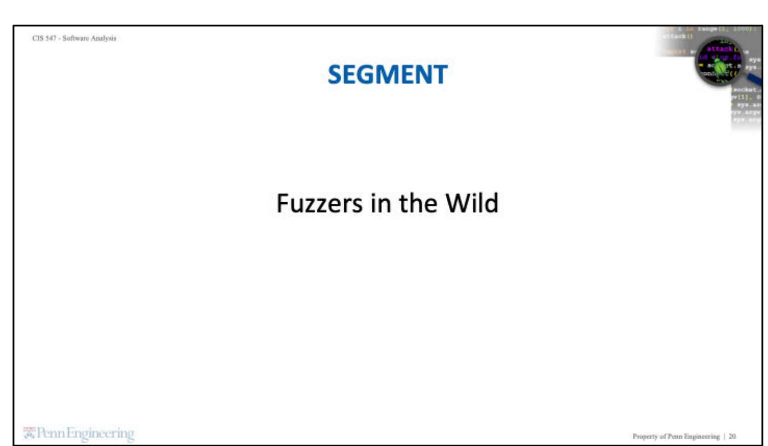


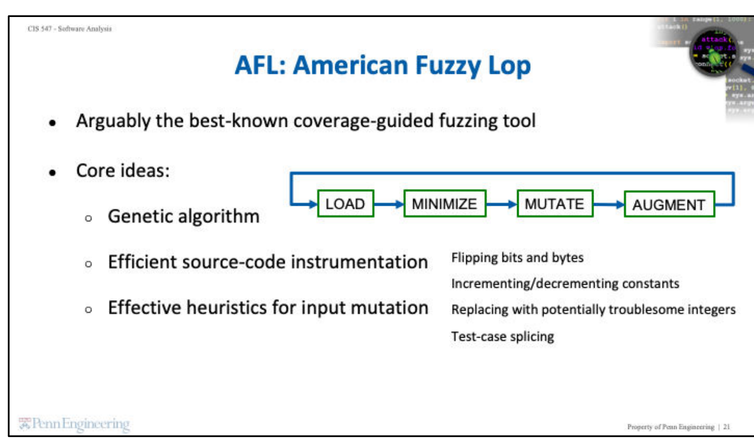
Now, let us consider a compiler for say the Java programming language. Let's see what would happen if we were to test such a compiler program by feeding it random inputs.

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.

Later in this module, you will be introduced to the concept of fuzz targets wherein the programmer can manually provide different entry points for a fuzzer to generate test inputs to exercise different parts of complex systems like this compiler.





AFL is encompassed by the "third generation" of fuzzers we discussed previously – it relies on a feedback-directed approach that begins with seed inputs. Its name is a reference to a breed of rabbit, the American Fuzzy Lop.

Developers of many popular and widely used applications have used AFL to find notable vulnerabilities and other interesting bugs. Projects such as Firefox, LibraOffice, ImageMagick, OpenCV, and the IOS Kernel have reported faults that have been uncovered by AFL. (More projects that have successfully employed AFL can be found in the link in the lecture handout: http://lcamtuf.coredump.cx/afl/)

The main ideas behind AFL include the use of a genetic algorithm, efficient source-level instrumentation, and effective input mutation heuristics.

The overall algorithm can be summed up as a loop over the following operations:
1) LOAD, 2) MINIMIZE, 3) MUTATE, and 4) AUGMENT.

In the load stage, an input is loaded. In the first iteration, the input will be loaded from the initial set of seeds. In following iterations, the input may have been added following feedback.

Next we enter the MINIMIZE stage in which AFL attempts to trim the test case to the smallest size that doesn't alter the behavior of the program. This is quite useful to the developers that are tasked with fixing the bug. A minimal crashing input is easier to

communicate and debug than an unnecessarily large crashing input.

Now we enter the MUTATE stage and repeatedly mutate the minimized input using a variety of traditional heuristic-based fuzzing strategies

Next we employ feedback to AUGMENT the set of seeds if any of the generated mutations resulted in a new state transition, and continue the process.

Mutation heuristics are arguably the most important component of a successful fuzzer. A mutation strategy that makes very small conservative changes will suffer from poor code coverage while an aggressive mutation strategy will make too many changes to the input seed such that it no longer conforms to the input specifications of the target program and will be discarded during input sanitization. Mutation strategies must be carefully engineered to exist in a sweet spot between changes that are too small and changes that are too large. The following are a set of mutation strategies used by AFL:

-> flipping bits and bytes – this mutation strategy is performed first. It is expensive in that for every bit / byte in the input, a mutated input can be generated. Additionally, returns diminish rapidly. After only a few passes, AFL switches to a less expensive mutation strategy.

-> incrementing / decrementing constants – this mutation strategy involves modifying integer value constants in the input. The range of modification has been carefully chosen to the range of –35 to + 35. This heuristic is chosen as fuzzing yields drop dramatically when further decrementing or incrementing past this range. Rather than trying every integer value, AFL switches to a different mutation strategy.

-> replacing integers – in this mutation strategy, AFL again replaces integer values from a carefully selected set of hardcoded values. These values were chosen for their demonstrably elevated likelihood of triggering edge conditions in typical code. For example, -1, 256, 1024, MAX_INT-1, or MAX_INT.

-> test case splitting - this is a last-resort strategy that involves taking two distinct input files from the queue that differ in at least two locations and splicing them at a random location. The separate inputs are then modified by randomly selecting strategies previous discussed.

LibFuzzer

- Motivation: enable to fuzz libraries (i.e. program components) instead of whole programs
- User provides fuzzing entry points called fuzz targets
- Intuition: if program has X lines of code and Y fuzz targets, then fuzzer only has to cover X / Y lines of code on average per target

AFL's well balanced and carefully engineered mutation strategies have proven to uncover bugs in large complex software with almost no setup. However, as we have just seen, it may be difficult for AFL to generate even coverage over all parts of the codebase. Recall the example of the Java compiler in which the lexer was heavily fuzzed, while only a small percentage of inputs reached the backend code.

LibFuzzer was motivated by the idea that many programs exist as libraries instead of whole programs. Even whole programs are typically built atop libraries. A fuzzer can achieve higher code coverage if the user provides entry points directly into libraries. We call such entry points as fuzz targets.

Here's the intuition. Consider the codebase of a large software organization (ex. LLVM) which comprises of say, 10 million lines of code. With a single entry point, the fuzzer has the incredibly arduous task of covering 10 million lines of code from a single entry point, in this case, the entry point of LLVM. LibFuzzer provides a framework to lessen the burden on the fuzzer by allowing the programmer to manually provide a multitude of different entry points or "fuzz targets". With say, 10 targets, the fuzzer has to do a tenth of the work on average, informally speaking: it has to reach only 1 million lines of code from each entry point. By adding additional entry points, the fuzzing engine has less responsibility of creating incredibly complex mutation strategies. Moreover, the fuzzers for different entry points can proceed in parallel.

LibFuzzer is also encompassed by the ”third generation" framework of fuzzers as it employs a coverage-guided, evolutionary fuzzing engine. The choice of LibFuzzer vs AFL should be chosen based on the codebase size and complexity in addition to constraints on setup time and manual developer labor. However, a developer need not choose a single fuzzer. In many cases, it makes sense to run both AFL and LibFuzzer in order to achieve maximum coverage.

LibFuzzer – Fuzz Target

- Fuzz Target: a function that takes as input an array of bytes and performs something interesting with the bytes using the API under test
- Fuzz target is executed by the fuzzer multiple times with different data
- Fuzz Target Skeleton:

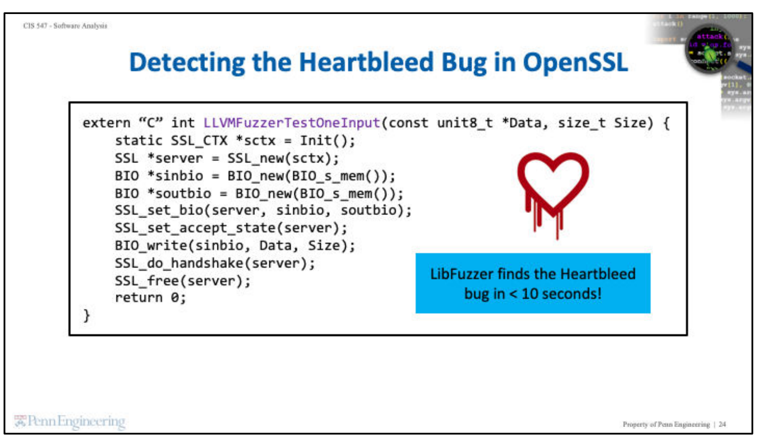
```
extern "C" int LLVMFuzzerTestOneInput(const unit8_t *Data, size_t Size) {
    PerformInterestingStuff(Data, Size)
    return 0;
}
```


Now, let's dive into the details of how a developer would setup LibFuzzer to fuzz new entry points of their application.

The fuzz target is implemented as a function that takes the input data as an array of bytes. The target should then do something "interesting" with the mutated data provided by the fuzzer. This usually involves calling the library or segment of code that we wish to test.

This fuzzing engine will execute the fuzz target many times with different inputs in the same process.

Here is the general skeleton of a fuzz target expected by LibFuzzer. It is intended for use by any program that can be compiled using the LLVM compiler infrastructure. Note that this fuzz target is not specific to LibFuzzer; it can be used by other fuzzing engines as well, such as AFL and Radamsa. The “PerformInterestingStuff” function is a placeholder which should be replaced with a normal sequence of function calls in the API of the library whose implementation we wish to fuzz. Let’s look at a concrete example next.



The infamous Heartbleed bug is a severe security vulnerability that was discovered in OpenSSL in 2014. OpenSSL is an open-source software library widely used by applications to secure communication over computer networks. The Heartbleed bug enabled eavesdropping on information protected by the SSL/TLS encryption protocol through the heartbeat protocol.

The heartbeat protocol is used to verify that the server is alive. The server sends back an exact copy of the data sent as a positive acknowledgement.

The heartbleed vulnurability is triggered when the user specifies the payload size to be larger than the actual length of the message it is sending. The server then responds with the exact message copy PLUS any additional data in the buffer. In this way, users can eavesdrop on additional data.

After writing the fuzz target as depicted here, LibFuzzer can find the heartbleed bug in under 10 seconds. The data generated by the fuzzer is a sequence of bytes saved in Data that specify the heartbeat request payload and size. Because the size and payload are randomly generated, there is no gaurantee that the size is equal to the actual size of the payload, triggering the heartbleed vulnerability in the form of a buffer overread which can be detected by a memory bounds checking sanitizer.

The sequence of function calls shown in this fuzz target are the typical sequence of function calls that are used for exercising the heartbeat functionality in OpenSSL by a

client application. The fuzz target writer need not employ any information regarding the internals of OpenSSL. In the next module, we will learn about a technique to automatically generate valid sequences of function calls of a given API, but they will rely on stronger static type systems such as Java's. In the lack of richer type information of function arguments and return results, as in the case of languages like C with weaker static type systems, or dynamic scripting languages like Javascript, one has to resort to requiring the user provide such a sequence.

The link to the complete fuzz target for heartbleed bug detection is included in the lecture handout.

https://github.com/google/fuzzer-test-suite/blob/master/openssl-1.0.1f/target.cc

Fuzz Target Guidelines

- Must tolerate different input kinds (e.g. malformed, null, huge, etc.)
- Should be fast (e.g. avoid long-running computation, logging I/O, etc.)
- Should not consume too much memory
- Narrower the better (e.g. write different fuzz targets if library handles different data formats)

LibFuzzer provides a set of guidelines for implementing fuzz targets.

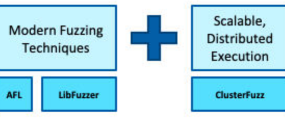
First, the target must tolerate a diverse set of inputs. The target willl be executed many times with many different mutated inputs. A fuzz target should support any type of input the fuzzer may generate.

Additionally, the target should be efficient in time and memory. Again, this target will be executed many times. A high complexity target would slow down the fuzzer on each input, causing an enormous overall slow down.

Lastly, as a general rule, the narrower the target the better. The greatest advantage of LibFuzzer is splitting the fuzzing burden between many entry points. The more we exploit that property, the more efficient our fuzzer becomes.

OSS-Fuzz

- Continuous fuzzing infrastructure hosted on the Google Cloud Platform

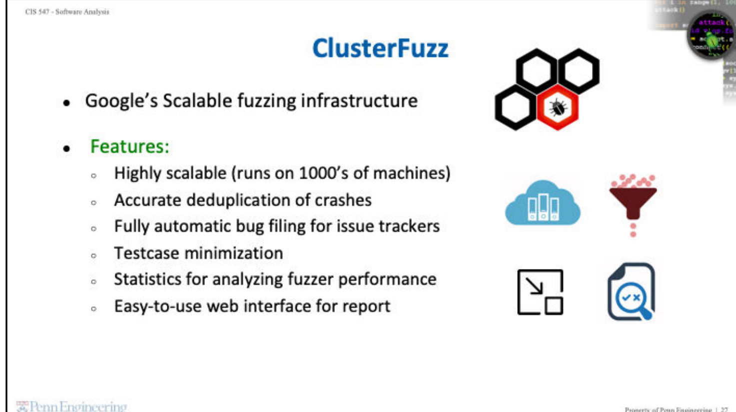


- OSS-Fuzz has discovered over 17,400 bugs from 2016 to 2019 in many large projects (e.g. openssl, llvm, postgresql, git, firefox)

Fuzzing, at its core, is a brute force technique. Although modern fuzzers employ tricks to drastically cut the search space and target inputs most likely to cause a crash, in many cases, fuzzers may run for months on end before finding a bug. For small companies, academics, or individual developers, the amount of computing power required to run fuzzers continuously is often unrealistic.

By combining modern fuzzing techniques with a scalable distributed execution, Google provides a platform for continous fuzzing, called OSS-Fuzz. OSS-Fuzz aims to make common open source software more secure and stable. The infrastructure is highly scalable as it runs on over 25,000 machines. It supports LibFuzzer and AFL in combination with sanitizers.

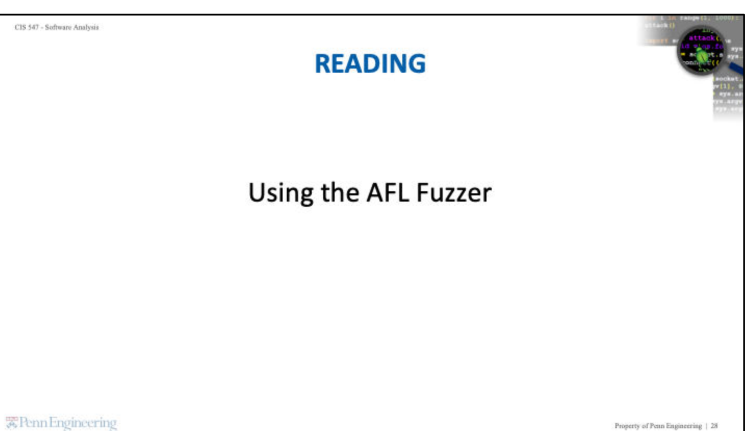
OSS-Fuzz has gained traction in the open source community and is responsible for fuzzing many high profile projects such as OpenSSL, LLVM, PostgreSQL, GIT, and Firefox.

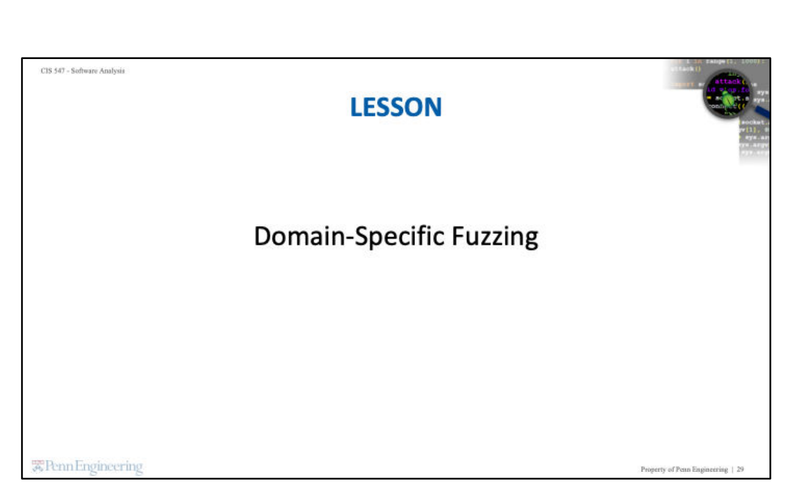


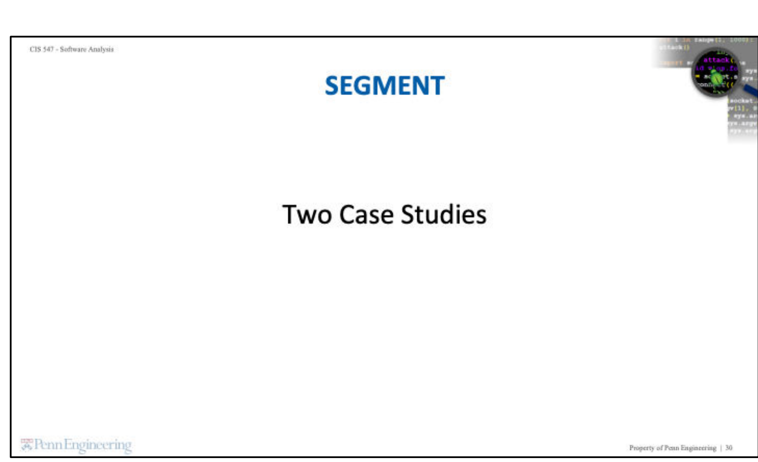
ClusterFuzz, in addition to providing the backend to OSS-Fuzz is used by Google to fuzz the Chrome Browser. As of January 2019, it has found ~16,000 bugs in Chrome and ~11,000 bugs in over 160 open source projects integrated with OSS-Fuzz.

ClusterFuzz also provides a convenient framework for automatically tracking bugs found by the underlying fuzzers. The issue tracking is hosted publicly by Monorail, a Google service originally created for tracking bugs in Chromium projects.

Additionally, ClusterFuzz logs various statistics to analyze fuzzer performance. Performance can be measured in a number of metrics such as crash rate, code coverage, and other properties necessary to ensure that a fuzzer is useful on real world programs.




CIS 547 - Software Analysis
LESSON
Domain-Specific Fuzzing
Penn Engineering
Property of Penn Engineering | 29


CIS 547 - Software Analysis
SEGMENT
Two Case Studies
Penn Engineering
Property of Penn Engineering | 30

CIS 547 - Software Analysis

Two Case Studies

- Mobile apps: Google's Monkey tool for Android
- Concurrent programs: Microsoft's Cuzz tool

Penn Engineering

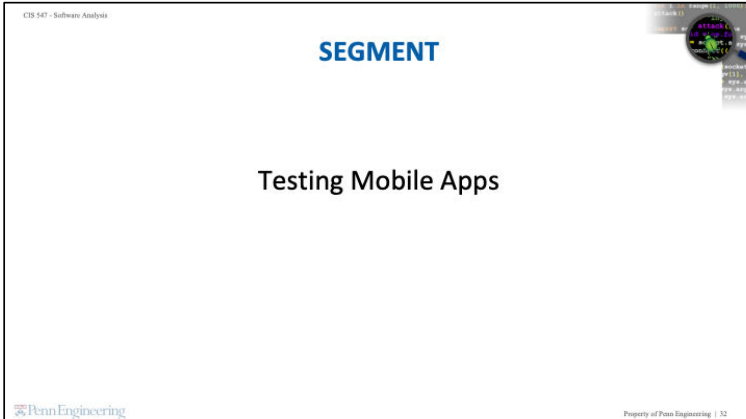
Property of Penn Engineering | 31

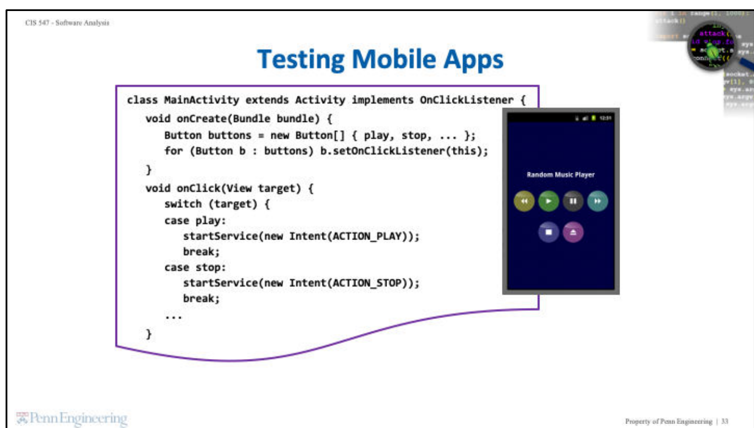
Random testing is a paradigm as opposed to a technique that will work out-of-the-box on any given program. In particular, for random testing to be effective, the test inputs must be generated from a reasonable distribution, which in turn is specific to the given program or class of programs.

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

The first class of programs is mobile apps. In particular, we will look at Google's Monkey tool for fuzz testing Android apps. While this class of programs is important in its own right, we will use this case study to also illustrate a useful technique in fuzzing, called grammar-based fuzzing.

The second 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. We will also use this case study to show how fuzzing can provide a probabilistic guarantee on finding bugs.



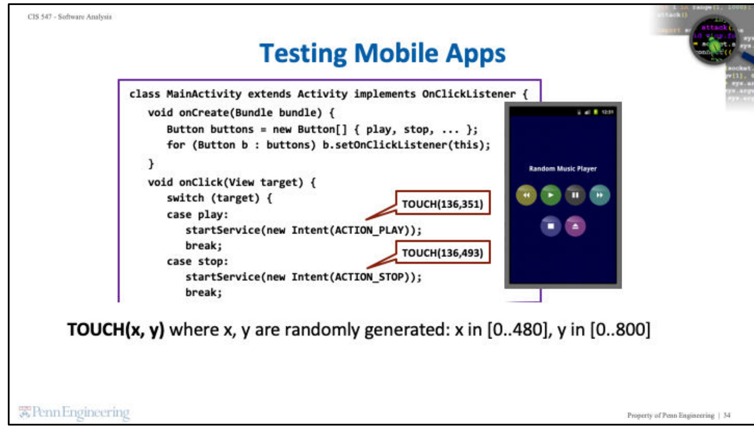


One domain in which fuzz testing has proved useful is that of mobile applications -- programs that run on mobile devices such as smartphones and tablets. A popular fuzz testing tool for mobile applications is the Monkey tool on the Android platform.

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.

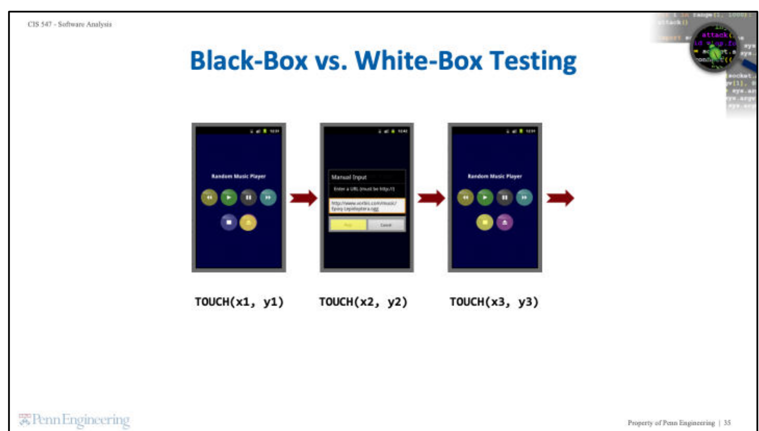


The most indivisible and routine kind of input to a mobile app is a GUI event, such as a TOUCH event at a certain pixel on the mobile device's display.

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.



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.

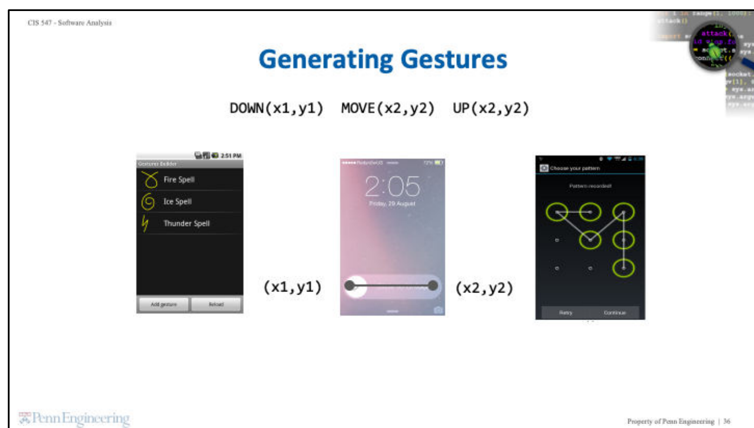
Here is a sequence of three such events that tests important functionality of our music player app. The widgets clicked by these events are highlighted.

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.



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

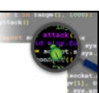
A simple gesture consists of a DOWN event at a pixel (x1,y1) (to simulate putting one's finger down on the display), then a MOVE event from (x1,y1) to a second pixel (x2,y2) (to simulate dragging one's finger across the display), followed by an UP event at pixel (x2,y2) (to simulate removing one's finger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

Next, we will look at a mechanism that is commonly used in fuzzing to specify a wide range of input formats, which in the case of mobile apps is a sequence of basic actions such as DOWN, MOVE, and UP.

SEGMENT

Grammar-Based Fuzzing



Grammar-Based Fuzzing

There are different ways to formally describe a language:

- Regular Expressions: simplest class of languages, e.g., `[a-z]*` denotes a (possibly empty) sequence of lowercase letter
- (Context-Free) Grammars: They can express a wide range of properties of an input language, e.g., the syntactical structure of an input format
- Universal Grammars: denote languages that are Turing-complete , e.g., they can specify Python programs

The mechanism we will use for specifying the input format of a program to be fuzzed is a context-free grammar, and the resulting fuzzing technique is called grammar-based fuzzing. A grammar is a set of production rules that describe all possible strings in a given language. There are different ways to formally describe a language. Here, we will discuss three well-known possibilities.

Regular Expressions are the simplest and least expressive class of languages. Limitations of Regular Expressions can be shown by the Pumping Lemma. Informally, the Pumping Lemma states that if string x is in the regular language L and a substring v is 'pumped', i.e., inserted any number of times, the resultant string still remains in the language L. The Pumping Lemma can be used as a test of irregularity. That is, if the pumping lemma does not hold for some language L, then the language is not regular. In particular, languages that require equality in number of specific characters are irregular. A classic example is the language of matching open and closed parentheses.

Context Free Grammars (CFGs) are strictly more expressive than regular expressions – Any language generated by a regular expression can be generated by a CFG. This class of grammars encompasses grammars in which any rule can be applied regardless of context. This excludes grammars that have multiple non-terminals on the left hand side of the production rules.

Universal Grammars are the most expressive class of languages. There are no

constraints on the nature of the production rules. This class encompasses Turing Complete languages including programming languages like C and Python.

The Chomsky heirarchy, developed by famous linguist and Penn alumni Noam Chomsky, is a containment heirarchy of classes of formal grammars. Each level of the hierarchy contains the levels below it. For example, universal languages encompass context free languages which encompass regular languages.

Context-free grammars provide a practical middle ground for use by grammar-based fuzzing.

Grammar-Based Fuzzing

Advantages of specifying program inputs using (context-free) grammars:

- Systematic and efficient test generation
- Expressive enough to handle complex input formats (e.g., XML or JSON)
- Provides the basis for fuzzing a wide range of software components: configurations, APIs, GUIs, network protocols, …

In this approach, the user provides a context-free grammar that specifies the form that inputs should take. A grammar based fuzzer will generate inputs from the underlying language. There are many advantages to this approach.

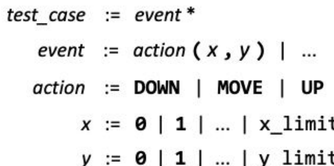
First, the user-provided information about the form of inputs is extremely useful in improving the fuzzer's performance and code coverage. The grammar is used to guide fuzzing toward specific inputs that would likely never be found using random black box fuzzers.

Second, context-free grammars are powerful enough to handle complex input formats such as XML or JSON.

Third, grammar-based fuzzing provides the basis for testing a wide range of software components such as configurations, APIs, GUIs, and network protocols as they require specific and complex input formats.

CIS 547 - Software Analysis

Grammar of Monkey Events



Penn Engineering

Property of Penn Engineering | 40

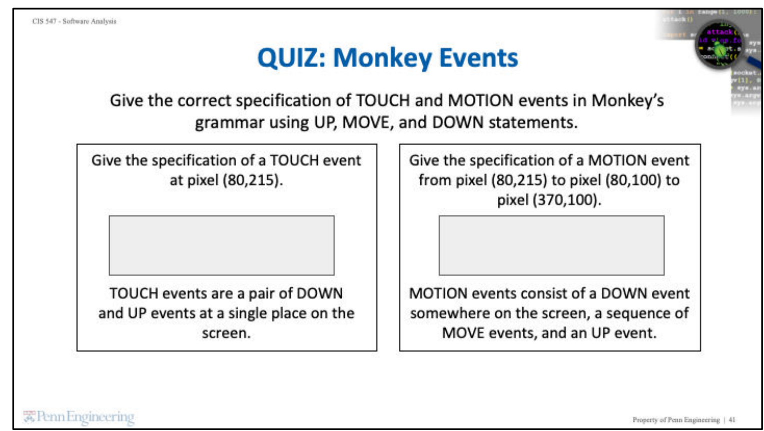
Having seen some example inputs that the Monkey random testing tool can generate, let's outline a grammar that systematically characterizes the possible inputs that the Monkey tool can generate.

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 following link 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.

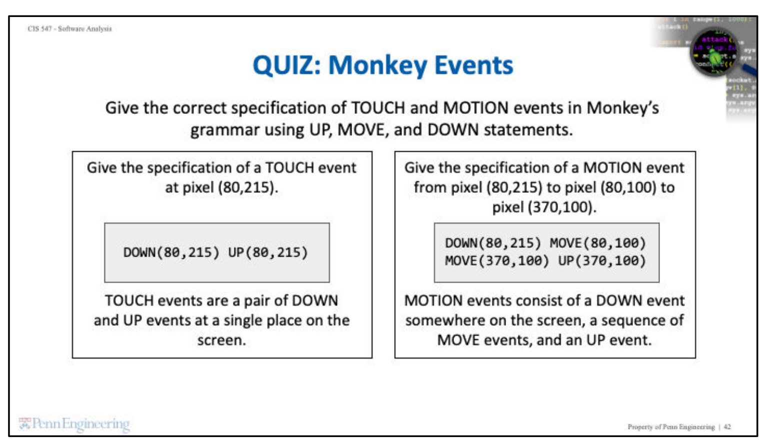


{QUIZ SLIDE}

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).

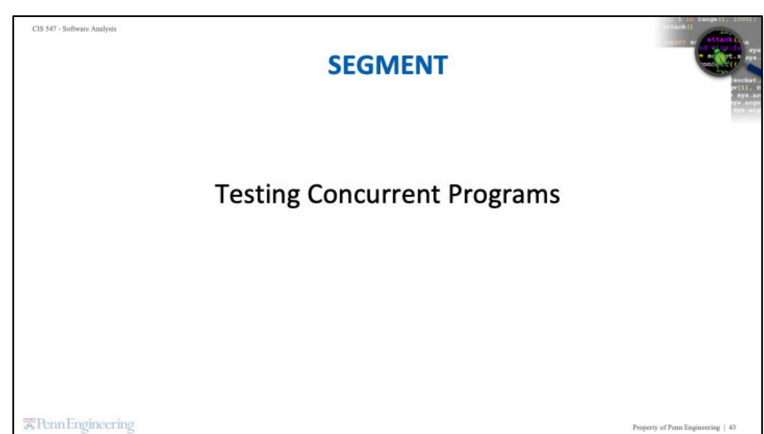


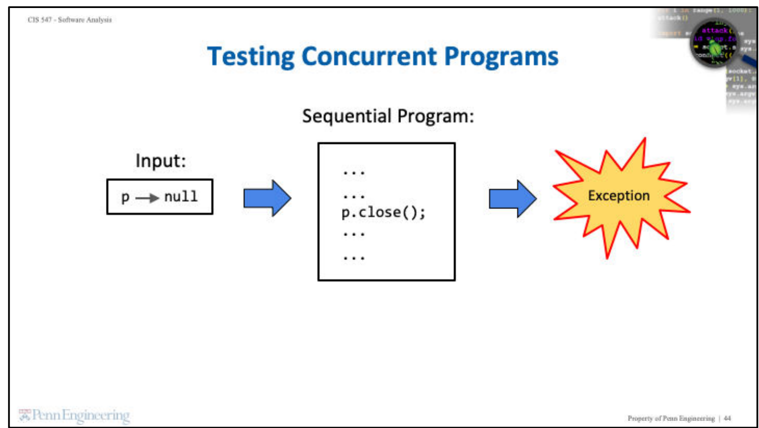
{SOLUTION SLIDE}

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.





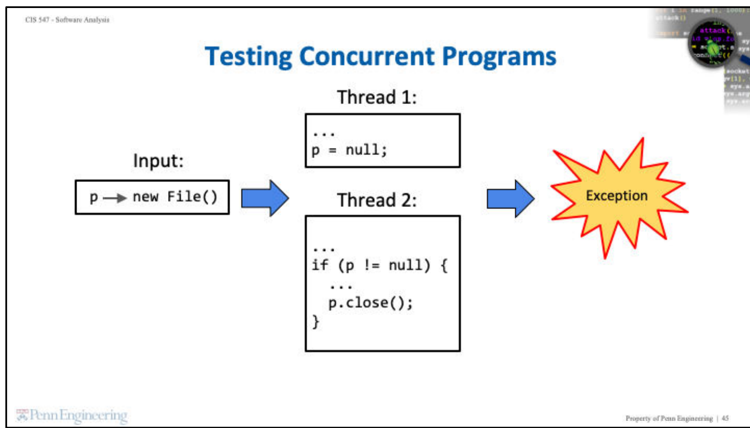
Another important domain in which random testing is exceedingly useful is the testing of concurrent programs.

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.

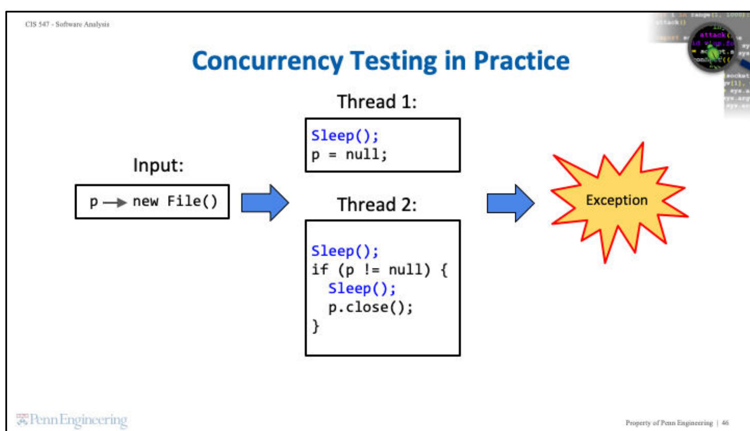
To be more concrete, consider this concurrent program that consists of two threads and takes as input a File handle p. Suppose we wish to test this program using a non-null File handle as input. The resulting execution may succeed or crash depending upon the thread schedule.

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.

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

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.

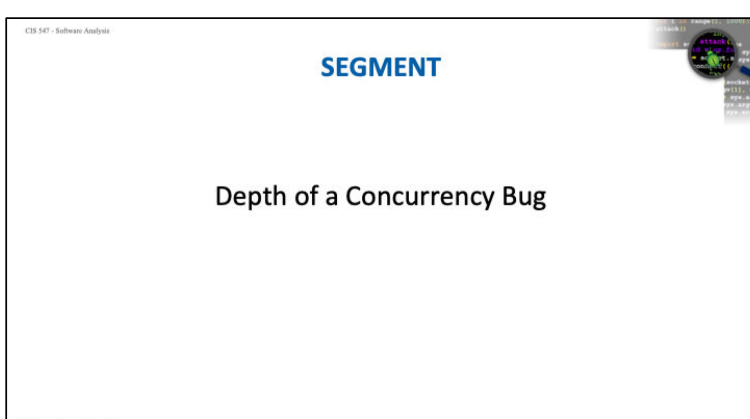
The resulting process is therefore less tedious and less prone to mistakes.

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 lecture handout.

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



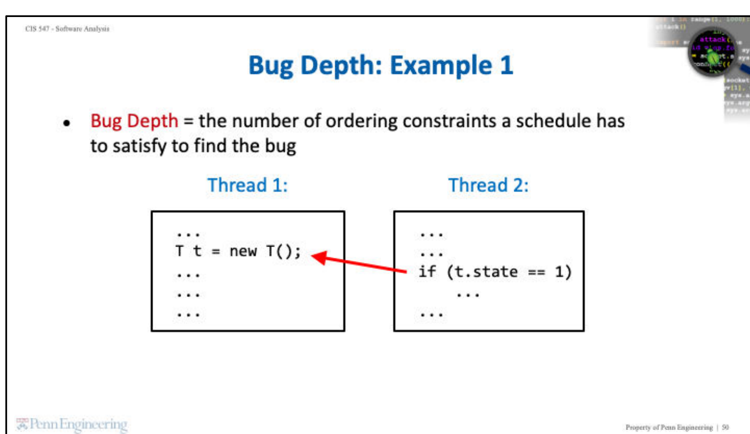
Depth of a Concurrency Bug

- 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.

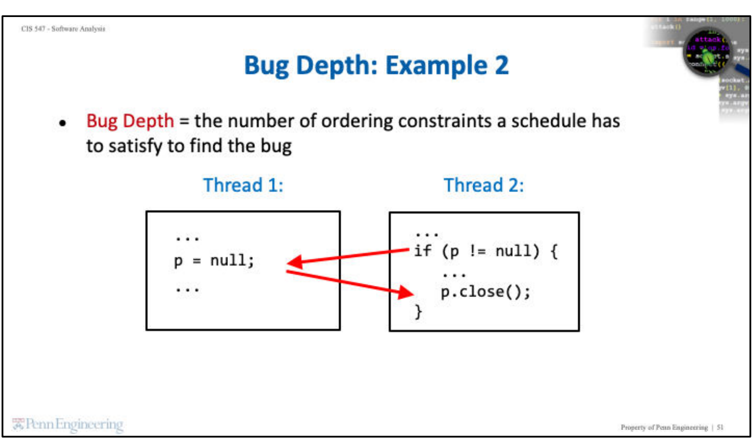
An ordering constraint is a requirement on the ordering between two statements in different threads.



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.



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.

CIS 547 - Software Analysis

Depth of a Concurrency Bug

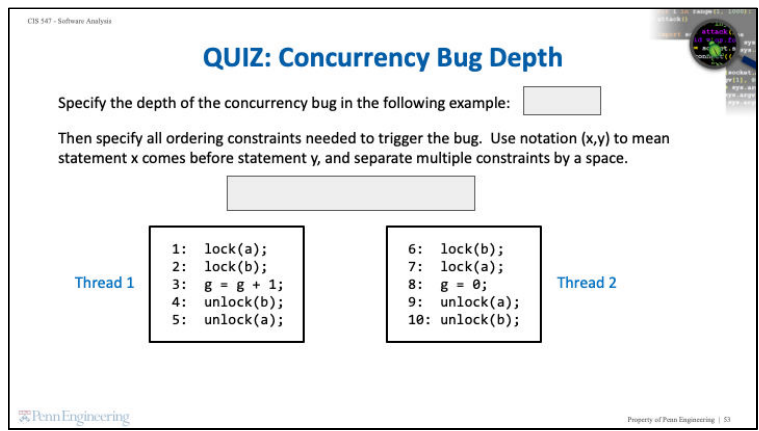
- Bug Depth = the number of ordering constraints a schedule has to satisfy to find the bug
- Observation exploited by Cuzz: bugs typically have small depth

Penn Engineering Property of Penn Engineering | 52

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.



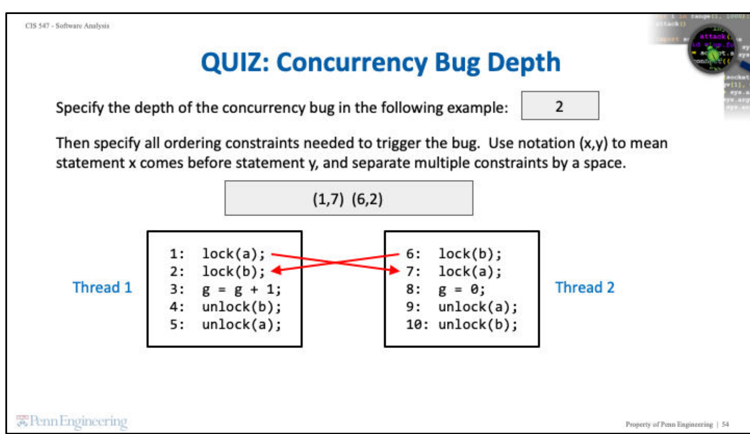
{QUIZ SLIDE}

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.



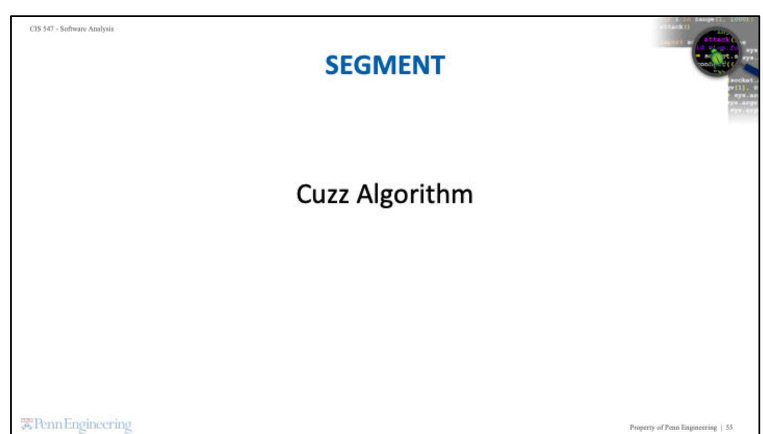
{SOLUTION SLIDE}

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;
  int a[] = random_permutation(1,n);
  for each tid in [1..n]:
    pri[tid] = d - 1 + a[tid];
  for each i in [1..d-1]:
    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;
}
```


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.

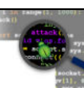
The Initialize() function randomly assigns each of d through d+n-1 as the priority value of one of the n threads. We will see why it does not use lower priority values 1 through d-1 momentarily.

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 - 1 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.

CIS 547 - Software Analysis

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

PennEngineering   Property of Penn Engineering | 57

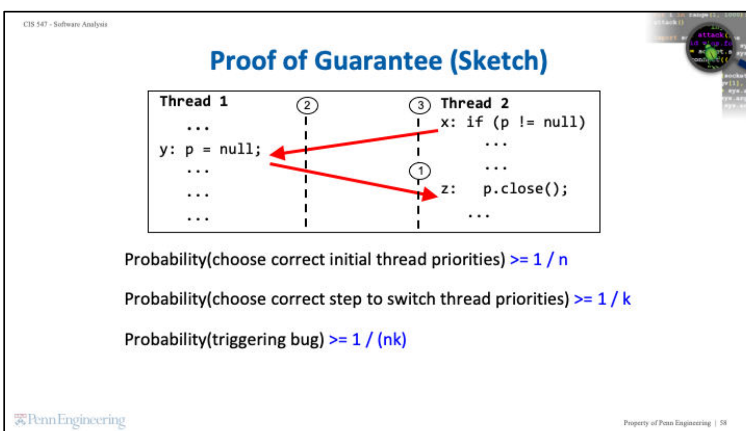
We can now state the probabilistic guarantee that Cuzz provides on finding concurrency bugs through its simple approach of fuzzing thread schedules.

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.



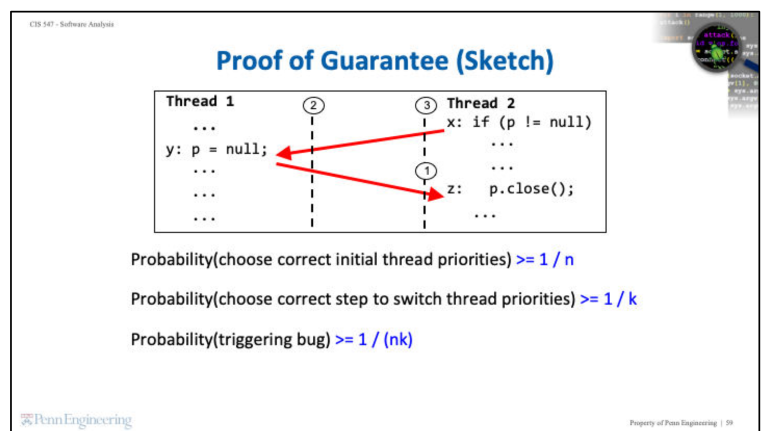
Let's use this program again as an example to demonstrate why the probabilistic guarantee holds.

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.



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.

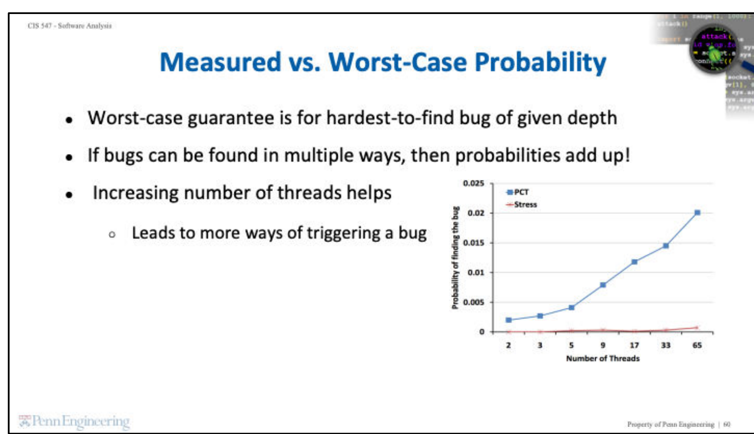
As Cuzz picks the statements where the thread priorities are changed uniformly over all statements, the probability of picking somewhere between Statement X and Statement Z to change the priorities is at least 1/k (recall that k is the number of statements executed by the program).

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 lecture handout.

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



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.

1) 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. 20 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. 3) 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.

SEGMENT

Cuzz Case Study & Key Takeaways

CIS 547 - Software Analysis

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 equals 11 seconds of Cuzz testing!

Penn Engineering

Property of Penn Engineering | 62

To better appreciate the amount of resources needed to find concurrency bugs using stress testing vs. Cuzz, 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!

CIS 547 - Software Analysis

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

Penn Engineering

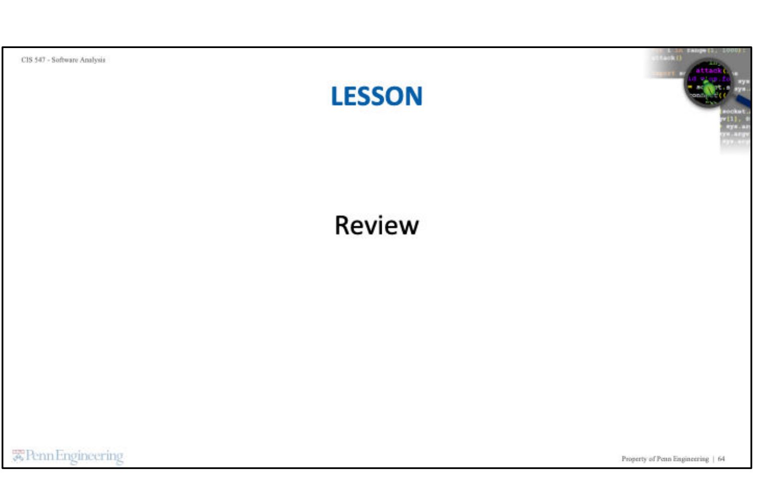
Property of Penn Engineering | 63

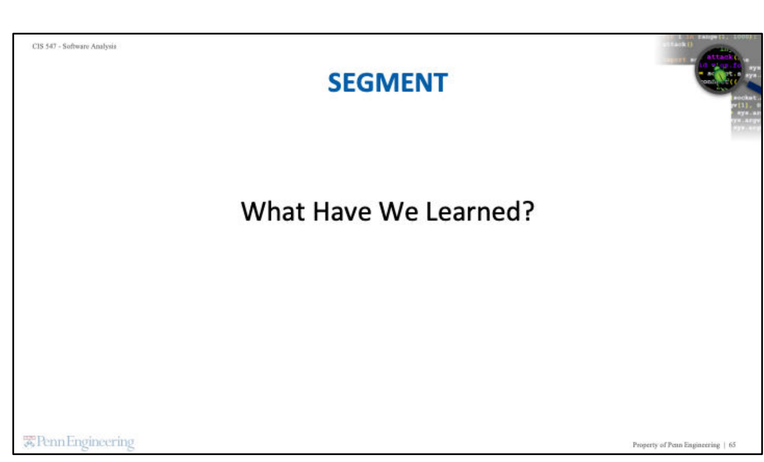
Let's review the key points you should take away from this section on concurrency testing.

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.


CIS 547 - Software Analysis
LESSON
Review
Penn Engineering
Property of Penn Engineering | 64


CIS 547 - Software Analysis
SEGMENT
What Have We Learned?
Penn Engineering
Property of Penn Engineering | 65

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 in order 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 module.

Random testing is a powerful technique in certain domains, including testing for security bugs, 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 module, you'll learn more techniques for automated test generation that are more directed and systematic than random testing.