Fuzzing

Martin Kellogg

Reading Quiz: fuzzing

Q1: the chapter uses an xkcd comic to explain which of these famous bugs that was detected by fuzzing?

A. Spectre/Meltdown
B. Y2K
C. Pentium FDIV
D. HeartBleed

Q2: **TRUE** or **FALSE**: the chapter includes another example targeting the `bc` utility

Reading Quiz: fuzzing

Q1: the chapter uses an xkcd comic to explain which of these famous bugs that was detected by fuzzing?

A. Spectre/Meltdown
B. Y2K
C. Pentium FDIV
D. HeartBleed

Q2: **TRUE** or **FALSE**: the chapter includes another example targeting the `bc` utility

Reading Quiz: fuzzing

Q1: the chapter uses an xkcd comic to explain which of these famous bugs that was detected by fuzzing?

A. Spectre/Meltdown
B. Y2K
C. Pentium FDIV
D. HeartBleed

Q2: **TRUE** or **FALSE**: the chapter includes another example targeting the `bc` utility

HW2 thoughts

HW2 thoughts

"I realized that it would be very time consuming and also difficult for me to manually collect a high coverage test suite…I **wrote a script** that would select an image **if it increases the coverage value**"

- this is an excellent approach to a problem like this!
    - always consider automation if a task is repetitive and manual
    - this student treated coverage as a **fitness function**, much like a mutational fuzzer (more details later)

Fuzzing: agenda

- **story time**
- mutational fuzzing
- grammar-based fuzzing
- fuzzing in the real world
- start symbolic execution (if there is enough time left)

Story Time

Story Time

- on a **stormy night** in Wisconsin in 1988…

Story Time

- on a **stormy night** in Wisconsin in 1988…
- a CS professor was connected **over a phoneline** to the computer in his office

Story Time

- on a **stormy night** in Wisconsin in 1988…
- a CS professor was connected **over a phoneline** to the computer in his office
- the thunderstorm outside caused "**fuzz**" on the line

Story Time

- on a **stormy night** in Wisconsin in 1988…
- a CS professor was connected **over a phoneline** to the computer in his office
- the thunderstorm outside caused “**fuzz**” on the line
  - this was a well-known problem in the days of telephones
  - on a phonecall, you’d just hear static

Story Time

- on a **stormy night** in Wisconsin in 1988…
- a CS professor was connected **over a phoneline** to the computer in his office
- the thunderstorm outside caused “**fuzz**” on the line
  - this was a well-known problem in the days of telephones
  - on a phonecall, you’d just hear static
- the fuzz caused many of the Unix utilities that the professor was using **to crash**

Story Time

- on a **stormy night** in Wisconsin in 1988…
- a CS professor was connected **over a phoneline** to the computer in his office
- the thunderstorm outside caused "**fuzz**" on the line
  - this was a well-known problem in the days of telephones
  - on a phonecall, you'd just hear static
- the fuzz caused many of the Unix utilities that the professor was using **to crash**
  - **insight**: just a few bits of random inputs are enough!

Test input generation

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?
  - Lens of **Statistics**: choose inputs "at random"

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?
  - Lens of **Statistics**: choose inputs "at random"
  - Lens of **Logic**: choose inputs that will maximize coverage

Test input generation

- As a human, often **choosing good test inputs** is the hardest part of writing a test
- For a computer, that's not true: computers can pick inputs **very fast** (given some policy)
- **Key problem**: which inputs should we pick?
  - Lens of **Statistics**: choose inputs "at random"
  - Lens of **Logic**: choose inputs that will maximize coverage

Modern fuzzers combine these two ideas.

Test data

- What are **all** the inputs to a test?

Test data

- What are all the inputs to a test?
  - Many programs (especially student programs) read from a file or stdin …

Test data

- What are **all** the inputs to a test?
    - Many programs (especially student programs) read from a file or stdin …
    - But what **else** is "read in" by a program and may influence its behavior?

Test data

- What are all the inputs to a test?
  - Many programs (especially student programs) read from a file

What else besides “input” can **influence** program behavior?

- User Input (e.g., GUI)
- Environment Variables, Command-Line Args
- Scheduler Interleavings
- Data from the Filesystem
  - User configuration, data files
- Data from the Network
  - Server and service responses

Test data: operating systems philosophy

Test data: operating systems philosophy

- “Everything is a file.”

Test data: operating systems philosophy

- "Everything is a file."
- After a few libraries and levels of indirection, reading from the user's keyboard boils down to opening a **special device file** (e.g., /dev/ttyS0) and reading from it
  - Similarly with mouse clicks, GUI commands, etc.

Test data: operating systems philosophy

- "Everything is a file."
- After a few libraries and levels of indirection, reading from the user's keyboard boils down to opening a **special device file** (e.g., /dev/ttyS0) and reading from it
    - Similarly with mouse clicks, GUI commands, etc.
- Ultimately programs can only interact with the outside world through ***system calls***
    - open, read, write, socket, fork, gettimeofday

Test data: operating systems philosophy

- “Everything is a file.”
- After a few libraries and levels of indirection, reading from the user's keyboard boils down to opening a **special device file** (e.g., /dev/ttyS0) and reading from it
  - Similarly with mouse clicks, GUI commands, etc.
- Ultimately programs can only interact with the outside world through ***system calls***
  - open, read, write, socket, fork, gettimeofday
- System calls (plus OS scheduling, etc.) are the full inputs

Test data: operating systems philosophy

- "Everything is a file"
- After a few librari … the user's keyboard b… e (e.g., /dev/ttyS0) and re…
  - Similarly with …
- Ultimately progra… rld through ***system call…***
  - open, read, write, socket, fork, gettimeofday
- System calls (plus OS scheduling, etc.) are the full inputs

1. Fully **hermetic** tests should include all these inputs
2. We want fully hermetic tests
3. 1 & 2 imply test input generation must also control the environment

Test data: operating systems philosophy

- “Everything is a file”
- After a few librari… the user's keyboard b… e (e.g., /dev/ttyS0) and re…
    - Similarly with …
- Ultimately progra… rld through *system call…*
    - open, read, write, socket, fork, gettimeofday
- System calls (plus OS scheduling, etc.) are the full inputs

1. Fully **hermetic** tests should include all these inputs
2. We want fully hermetic tests
3. 1 & 2 imply test input generation must also **control the environment**

Fuzzing: agenda

- story time
- **mutational fuzzing**
- grammar-based fuzzing
- fuzzing in the real world
- start symbolic execution (if there is enough time left)

What is fuzzing?

Key idea: provide inputs “at random” to the program and use an ***implicit oracle***

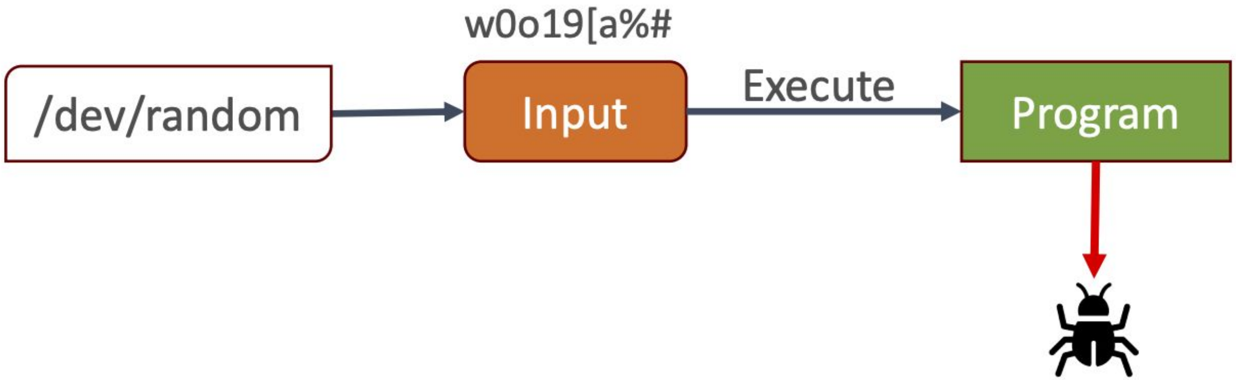
What is fuzzing?

Key idea: provide inputs "at random" to the program and use an ***implicit oracle***

An **implicit oracle** is an oracle that doesn't require an explicit spec from the programmer, such as "programs should not crash".

What is fuzzing?

Key idea: provide inputs "at random" to the program and use an ***implicit oracle***



What is fuzzing?

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

What is fuzzing?

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

- typical oracle: **crashes**

What is fuzzing?

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

- typical oracle: **crashes**
  - but any other implicit oracle will work (we'll discuss more implicit oracles in a few weeks)

What is fuzzing?

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing random or semi-random inputs to a program and monitoring for violations of an implicit oracle.

- typical oracle: **crashes**
  - but any other implicit oracle will work (we'll discuss more implicit oracles in a few weeks)
- the simplest fuzzers use **truly-random input**

What is fuzzing?

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing ran … n and monitoring for violatio …

- typical oracle: **crashes**
  - but any other implicit oracle will work (we’ll discuss more implicit oracles in a few weeks)
- the simplest fuzzers use **truly-random input**

Truly-random input example:

'!7#%"*#0=)$;%6*;>638:*>80"=</>('

What is fuzzing?

Definition: ***fuzzing*** (or ***fuzz testing***) is an automated testing technique that involves providing ran n and monitoring for violatio

Truly-random input example:

'!7#%"*#0=)$;%6*;>638:*>80"=</>(`

- typical oracle: **crashes**
  - but any other implicit oracle will work (we'll discuss more implicit oracles in a few weeks)
- the simplest fuzzers use **truly-random input**
  - but that rarely works well in practice except to test the code that **reads input** (why?)

Safely reading input

- originally, fuzzing was most effective for detecting bugs in **input-handling code**
  - that is, code that might be exposed to the outside

Safely reading input

- originally, fuzzing was most effective for detecting bugs in **input-handling code**
  - that is, code that might be exposed to the outside
  - such code shouldn't crash under any circumstances
    - even when presented with invalid input!

Safely reading input

- originally, fuzzing was most effective for detecting bugs in **input-handling code**
  - that is, code that might be exposed to the outside
  - such code shouldn't crash under any circumstances
    - even when presented with invalid input!
- however, **most code** in most programs is not input-handling code

Safely reading input

- originally, fuzzing was most effective for detecting bugs in **input-handling code**
    - that is, code that might be exposed to the outside
    - such code shouldn't crash under any circumstances
        - even when presented with invalid input!
- however, **most code** in most programs is not input-handling code
    - because most programs accept input in a **defined format**

Safely reading input

- originally, fuzzing was most effective for detecting bugs in **input-handling code**
  - that is, code that might be exposed to the outside
  - such code shouldn't crash under any circumstances
    - even when presented with invalid input!
- however, **most code** in most programs is not input-handling code
  - because most programs accept input in a **defined format**
  - **implication**: fuzzing with random input produces tests that have **low coverage**

Achieving high coverage

- As an example, consider a program that accepts a URL:

Achieving high coverage

- As an example, consider a program that accepts a URL:

```
scheme://netloc/path?query#fragment
```

Achieving high coverage

- As an example, consider a program that accepts a URL:

```
scheme://netloc/path?query#fragment
```

- `scheme` is the protocol to be used, including http, https, ftp...
- `netloc` is the host to connect to, e.g., `www.google.com`
- `path` is the path on that host
- `query` is a list of key/value pairs, such as `q=fuzzing`
- `fragment` is a marker for a location in the retrieved document

Achieving high coverage

What do you think are the odds of generating a valid URL by choosing random characters?

- As an example, consider a prog

```
scheme://netloc/path?query#fragment
```

- `scheme` is the protocol to be used, including http, https, ftp...
- `netloc` is the host to connect to, e.g., `www.google.com`
- `path` is the path on that host
- `query` is a list of key/value pairs, such as `q=fuzzing`
- `fragment` is a marker for a location in the retrieved document

Achieving high coverage

- For programs with **structured input**, random input generation is insufficient to achieve high coverage

Achieving high coverage

- For programs with **structured input**, random input generation is insufficient to achieve high coverage
    - we need a way to generate inputs that pass the program's **input validation**

Achieving high coverage

- For programs with **structured input**, random input generation is insufficient to achieve high coverage
  - we need a way to generate inputs that pass the program's **input validation**
- Most of today's lecture will be about various ways to do that:

Achieving high coverage

- For programs with **structured input**, random input generation is insufficient to achieve high coverage
  - we need a way to generate inputs that pass the program's **input validation**
- Most of today's lecture will be about various ways to do that:
  - by using **seed inputs** from the user to help

Achieving high coverage

- For programs with **structured input**, random input generation is insufficient to achieve high coverage
  - we need a way to generate inputs that pass the program's **input validation**
- Most of today's lecture will be about various ways to do that:
  - by using **seed inputs** from the user to help
  - by taking advantage of a **known grammar** for the inputs

Achieving high coverage

- For programs with **structured input**, random input generation is insufficient to achieve high coverage
  - we need a way to generate inputs that pass the program's **input validation**
- Most of today's lecture will be about various ways to do that:
  - by using **seed inputs** from the user to help
  - by taking advantage of a **known grammar** for the inputs
  - by using program analysis to find **constraints** on the input that will allow it to pass various checks

Review: genetic algorithms

Review: genetic algorithms

- ***genetic algorithms*** are a class of biology-inspired algorithms that "evolve" a solution to a problem

Review: genetic algorithms

- ***genetic algorithms*** are a class of biology-inspired algorithms that "evolve" a solution to a problem
    - maintain a fixed-size ***population*** of possible solutions

Review: genetic algorithms

- ***genetic algorithms*** are a class of biology-inspired algorithms that "evolve" a solution to a problem
    - maintain a fixed-size ***population*** of possible solutions
    - define a set of ***mutation operators*** that combine (parts of) solutions from the population to create new solutions

Review: genetic algorithms

- ***genetic algorithms*** are a class of biology-inspired algorithms that "evolve" a solution to a problem
  - maintain a fixed-size ***population*** of possible solutions
  - define a set of ***mutation operators*** that combine (parts of) solutions from the population to create new solutions
  - apply the mutation operators to the current population to a create a new "generation" of solutions

Review: genetic algorithms

- ***genetic algorithms*** are a class of biology-inspired algorithms that "evolve" a solution to a problem
    - maintain a fixed-size ***population*** of possible solutions
    - define a set of ***mutation operators*** that combine (parts of) solutions from the population to create new solutions
    - apply the mutation operators to the current population to a create a new "generation" of solutions
    - use a ***fitness function*** to prune the starting population + the new generation back down to the fixed population size

Review: genetic algorithms

- ***genetic algorithms*** are a class of biology-inspired algorithms that "evolve" a solution to a problem
    - maintain a fixed-size ***population*** of possible solutions
    - define a set of ***mutation operators*** that combine (parts of) solutions from the population to create new solutions
    - apply the mutation operators to the current population to a create a new "generation" of solutions
    - use a ***fitness function*** to prune the starting population + the new generation back down to the fixed population size
    - repeat until some stopping condition

Review: genetic algorithms: properties

- genetic algorithms are a kind of ***search algorithm***

Review: genetic algorithms: properties

- genetic algorithms are a kind of ***search algorithm***
  - typically, they work best when the space of possible solutions is very large

Review: genetic algorithms: properties

- genetic algorithms are a kind of ***search algorithm***
    - typically, they work best when the space of possible solutions is very large
- a **good fitness function** is critical to an effective genetic algorithm
    - what are some properties of a good fitness function?

Review: genetic algorithms: properties

- genetic algorithms are a kind of ***search algorithm***
  - typically, they work best when the space of possible solutions is very large
- a **good fitness function** is critical to an effective genetic algorithm
  - what are some properties of a good fitness function?
    - continuous
    - monotonic (or at least with few local optima)
    - cheap to evaluate

Review: genetic algorithms: properties

- genetic algorithms are a kind of ***search algorithm***
  - typically, they work best when the space of possible solutions is very large
- a **good fitness function** is critical to an effective genetic algorithm
  - what are some properties of a good fitness function?
    - continuous
    - monotonic (or at least with few local optima)
    - cheap to evaluate
  - what might make a good fitness function for a fuzzer?

Review: genetic algorithms: properties

- genetic algorithms are a kind of ***search algorithm***
  - typically, they work best when the space of possible solutions is very large
- a **good fitness function** is critical to an effective genetic algorithm
  - what are some properties of a good fitness function?
    - continuous
    - monotonic (or at least with few local optima)
    - cheap to evaluate
  - what might make a good fitness function for a fuzzer?
    - coverage!

Mutational fuzzing

- ***mutational fuzzing*** is the use of a genetic algorithm for generating test inputs

Mutational fuzzing

- ***mutational fuzzing*** is the use of a genetic algorithm for generating test inputs
  - fitness function is usually coverage (statement or branch)

Mutational fuzzing

- ***mutational fuzzing*** is the use of a genetic algorithm for generating test inputs
  - fitness function is usually coverage (statement or branch)
  - population is made up of test inputs

Mutational fuzzing

- *mutational fuzzing* is the use of a genetic algorithm for generating test inputs
  - fitness function is usually coverage (statement or branch)
  - population is made up of test inputs
- key questions:
  - where does the initial population come from?
  - how are the test inputs mutated?

Mutational fuzzing

- *mutational fuzzing* is the use of a genetic algorithm for generating test inputs
  - fitness function is usually coverage (statement or branch)
  - population is made up of test inputs
- key questions:
  - **where does the initial population come from?**
  - how are the test inputs mutated?

Mutational fuzzing: seed inputs

- typically, a mutational fuzzer is starts with an initial population of ***seed inputs*** provided by the user

Mutational fuzzing: seed inputs

- typically, a mutational fuzzer is starts with an initial population of ***seed inputs*** provided by the user
    - for example, in our URL parsing example, these would be URLs that we know are valid

Mutational fuzzing: seed inputs

- typically, a mutational fuzzer is starts with an initial population of ***seed inputs*** provided by the user
  - for example, in our URL parsing example, these would be URLs that we know are valid
- the **choice of seed inputs** is one of the most important inputs to the fuzzer

Mutational fuzzing: seed inputs

- typically, a mutational fuzzer is starts with an initial population of ***seed inputs*** provided by the user
  - for example, in our URL parsing example, these would be URLs that we know are valid
- the **choice of seed inputs** is one of the most important inputs to the fuzzer
  - “garbage in, garbage out” is very true for this kind of fuzzer
  - can also significantly impact performance
  - HW3 hint: choose seed images carefully

Mutational fuzzing

- ***mutational fuzzing*** is the use of a genetic algorithm for generating test inputs
    - fitness function is usually coverage (statement or branch)
    - population is made up of test inputs
- key questions:
    - where does the initial population come from?
    - **how are the test inputs mutated?**

Mutational fuzzing: mutation operators

- you might think that the choice of mutation operator would also have a big impact on performance

Mutational fuzzing: mutation operators

- you might think that the choice of mutation operator would also have a big impact on performance
  - but, surprisingly, techniques like **bit flipping** or **random string mutation** work just fine

Mutational fuzzing: mutation operators

- you might think that the choice of mutation operator would also have a big impact on performance
  - but, surprisingly, techniques like **bit flipping** or **random string mutation** work just fine
    - as long as it is **cheap to evaluate the fitness** of a particular input, we can create and discard many, many inputs!

Mutational fuzzing: mutation operators

- you might think that the choice of mutation operator would also have a big impact on performance
  - but, surprisingly, techniques like **bit flipping** or **random string mutation** work just fine
    - as long as it is **cheap to evaluate the fitness** of a particular input, we can create and discard many, many inputs!
- using low-level mutations means we must be able to combine mutants into ***higher-order mutants*** of seed inputs

Mutational fuzzing: mutation operators

- you might think that the choice of mutation operator would also have a big impact on performance
  - but, surprisingly, techniques like **bit flipping** or **random string mutation** work just fine
    - as long as it is **cheap to evaluate the fitness** of a particular input, we can create and discard many, many inputs!
- using low-level mutations means we must be able to combine mutants into ***higher-order mutants*** of seed inputs
  - our genetic algorithm lets us do this easily, because *neutral* mutations naturally accumulate in the population

Mutational fuzzing: mutation operators

- you might think that the choice of mutation operator would also have a big impact on performance
  - but, surprisingly, techniques like **bit flipping** or **random string mutation** work just fine
    - as long as it is **cheap t**
      input, we can create a
- using low-level mutations me
  mutants into ***higher-order mutants*** of seed inputs
  - our genetic algorithm lets us do this easily, because ***neutral*** mutations naturally accumulate in the population

> A ***neutral*** mutation is one that does not impact fitness. E.g., `go`**`g`**`gle.com` is also a valid URL.

Mutational fuzzing: coverage as fitness

- we can easily build a fuzzer that uses line/statement coverage as its fitness function

Mutational fuzzing: coverage as fitness

- we can easily build a fuzzer that uses line/statement coverage as its fitness function
- however, statement coverage is actually a bit **too coarse-grained** in practice

Mutational fuzzing: coverage as fitness

- we can easily build a fuzzer that uses line/statement coverage as its fitness function
- however, statement coverage is actually a bit **too coarse-grained** in practice
- practical fuzzers like AFL (used in HW3) use branch or ***path*** coverage

Mutational fuzzing: coverage as fitness

- we can easily build a fuzzer that uses line/statement coverage as its fitness function
- however, statement coverage is actually a bit **too coarse-grained** in practice
- practical fuzzers like AFL (used in HW3) use branch or ***path*** coverage
  - AFL's fitness function rewards an input for **any new path**, even if that path has the same branch coverage
    - this means e.g., that an input that causes a loop to go around **twice instead of once** is rewarded

Mutational fuzzing: power schedules

- consider a new generation of test inputs containing:
  - one input that covered a new branch or path that was created in the last round of mutation
  - *n*-1 inputs that have been in the population for at least a few generations
- **which** input should we mutate?

Mutational fuzzing: power schedules

- consider a new generation of test inputs containing:
  - one input that covered a new branch or path that was created in the last round of mutation
  - *n*-1 inputs that have been in the population for at least a few generations
- **which** input should we mutate?
  - intuitively, we know that the **new input** should be mutated more often in the next generation

Mutational fuzzing: power schedules

- consider a new generation of test inputs containing:
  - one input that covered a new branch or path that was created in the last round of mutation
  - *n*-1 inputs that have been in the population for at least a few generations
- **which** input should we mutate?
  - intuitively, we know that the **new input** should be mutated more often in the next generation
  - we implement this intuition via **power schedules**

Mutational fuzzing: power schedules

- a ***power schedule*** distributes fuzzing time among the seeds in the population

Mutational fuzzing: power schedules

- a ***power schedule*** distributes fuzzing time among the seeds in the population
  - each seed is assigned an ***energy*** value

Mutational fuzzing: power schedules

- a ***power schedule*** distributes fuzzing time among the seeds in the population
    - each seed is assigned an ***energy*** value
        - the odds of mutating a seed are proportional to its energy

Mutational fuzzing: power schedules

- a ***power schedule*** distributes fuzzing time among the seeds in the population
  - each seed is assigned an ***energy*** value
    - the odds of mutating a seed are proportional to its energy
  - the usual policy is:

Mutational fuzzing: power schedules

- a *power schedule* distributes fuzzing time among the seeds in the population
    - each seed is assigned an ***energy*** value
        - the odds of mutating a seed are proportional to its energy
    - the usual policy is:
        - newly-discovered seeds start with high energy

Mutational fuzzing: power schedules

- a *power schedule* distributes fuzzing time among the seeds in the population
  - each seed is assigned an ***energy*** value
    - the odds of mutating a seed are proportional to its energy
  - the usual policy is:
    - newly-discovered seeds start with high energy
    - when a seed is mutated to produce an input that increases fitness, its energy increases

Mutational fuzzing: power schedules

- a ***power schedule*** distributes fuzzing time among the seeds in the population
  - each seed is assigned an ***energy*** value
    - the odds of mutating a seed are proportional to its energy
  - the usual policy is:
    - newly-discovered seeds start with high energy
    - when a seed is mutated to produce an input that increases fitness, its energy increases
    - when a seed is mutated, but doesn’t produce an input that increases fitness, its energy decreases

Mutational fuzzing: power schedules

- we can use **any policy** to assign energy
- examples:

Mutational fuzzing: power schedules

- we can use **any policy** to assign energy
- examples:
  - change the power schedule so that seeds that exercise ***unusual paths*** have more energy

Mutational fuzzing: power schedules

- we can use **any policy** to assign energy
- examples:
  - change the power schedule so that seeds that exercise ***unusual paths*** have more energy
    - "unusual" paths are those rarely covered by other seeds

Mutational fuzzing: power schedules

- we can use **any policy** to assign energy
- examples:
  - change the power schedule so that seeds that exercise ***unusual paths*** have more energy
    - "unusual" paths are those rarely covered by other seeds
    - this technique can dramatically improve the fuzzer's performance

Mutational fuzzing: power schedules

- we can use **any policy** to assign energy
- examples:
  - change the power schedule so that seeds that exercise ***unusual paths*** have more energy
    - "unusual" paths are those rarely covered by other seeds
    - this technique can dramatically improve the fuzzer's performance
  - change the power schedule to assign energy based on **distance to some objective**
    - called ***directed fuzzing***

Mutational fuzzing: putting it all together

- let’s consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it

Mutational fuzzing: putting it all together

- let’s consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)

Mutational fuzzing: putting it all together

- let's consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)

Population of inputs:

```
https://www.google.com/
https://web.njit.edu/~mjk76/
https://calendar.google.com/calendar/u/0/r?cid=bWprNzZAbmppdC5lZHU
http://3.149.230.63:50000
...
```

Mutational fuzzing: putting it all together

- let’s consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy

Mutational fuzzing: putting it all together

- let’s consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy

Population of inputs (energy):

```
https://www.google.com/ (1)
https://web.njit.edu/~mjk76/ (1)
https://calendar.google.com/calendar/u/0/r?cid=bWprNzZAbmppdC5lZHU
(1)
http://3.149.230.63:50000 (1)
...
```

Mutational fuzzing: putting it all together

- let’s consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy
    - choose an input at random, weighted by energy

Mutational fuzzing: putting it all together

- let's consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy

Population of inputs (energy):

```
https://www.google.com/ (1)
https://web.njit.edu/~mjk76/ (1)
https://calendar.google.com/calendar/u/0/r?cid=bWprNzZAbmppdC5lZHU
(1)
http://3.149.230.63:50000 (1)
...
```

Mutational fuzzing: putting it all together

- let's consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy
    - choose an input at random, weighted by energy
    - mutate that input by changing a random character

Mutational fuzzing: putting it all together

- let's consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy

Population of inputs (energy):

```
https://www.google.com/ (1)
https://web.njit.edy/~mjk76/ (1)
https://calendar.google.com/calendar/u/0/r?cid=bWprNzZAbmppdC51ZHU
(1)
http://3.149.230.63:50000 (1)
...
```

Mutational fuzzing: putting it all together

- let’s consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy
    - choose an input at random, weighted by energy
    - mutate that input by changing a random character
    - evaluate whether coverage increases

Mutational fuzzing: putting it all together

- let’s consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy
    - choose an input at random, weighted by energy
    - mutate that input by changing a random character
    - **evaluate whether coverage increases**
      - suppose that it doesn’t

Mutational fuzzing: putting it all together

- let's consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy

Population of inputs (energy):

```
https://www.google.com/ (1)
https://web.njit.edu/~mjk76/ (0.75)
https://calendar.google.com/calendar/u/0/r?cid=bWprNzZAbmppdC5lZHU
(1)
http://3.149.230.63:50000 (1)
...
```

Mutational fuzzing: putting it all together

- let’s consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy
    - choose an input at random, weighted by energy
    - mutate that input by changing a random character
    - evaluate whether coverage increases
    - repeat the process…

Mutational fuzzing: putting it all together

- let's consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy
    - **choose an input at random, weighted by energy**
    - mutate that input by changing a random character
    - evaluate whether coverage increases
    - repeat the process…

Mutational fuzzing: putting it all together

- let’s consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
    - we provide a set of seed inputs (valid and invalid URLs)
    - initially, each seed has equal energy
        - choose an input at random, weighted by energy
        - **mutate that input by changing a random character**
        - evaluate whether coverage increases
        - repeat the process…

Mutational fuzzing: putting it all together

- let’s consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy
    - choose an input at random, weighted by energy
    - mutate that input by changing a random character
    - **evaluate whether coverage increases**
    - repeat the process…

Mutational fuzzing: putting it all together

- let's consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy

Population of inputs (energy):

```
https://www.google.com/ (1)
https://web.njit.edu/~mjk76/ (0.75)
https://calendar.google.com/calendar/u/0/r?cid=bWprNzZAbmppdC5lZHU
(1)
http://f.149.230.63:50000 (1)
...
```

Mutational fuzzing: putting it all together

- let’s consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
    - we provide a set of seed inputs (valid and invalid URLs)
    - initially, each seed has equal energy
        - choose an input at random, weighted by energy
        - mutate that input by changing a random character
        - **evaluate whether coverage increases**
            - suppose that it does
        - repeat the process…

Mutational fuzzing: putting it all together

- let's consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy

Population of inputs (energy):

```
https://www.google.com/ (1)
https://web.njit.edu/~mjk76/ (0.75)
https://calendar.google.com/calendar/u/0/r?cid=bWprNzZAbmppdC5lZHU
(1)
http://3.149.230.63:50000 (2)
http://f.149.230.63:50000 (2)
```

Mutational fuzzing: putting it all together

- let's consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy
    - choose an input at random, weighted by energy
    - mutate that input by changing a random character
    - evaluate whether coverage increases
    - repeat the process…
  - **create a new generation and then start over**

Mutational fuzzing: putting it all together

- let's consider the URL parsing example again and walk through how a mutational fuzzer might fuzz it
  - we provide a set of seed inputs (valid and invalid URLs)
  - initially, each seed has equal energy
  - create a new generation and then start over

Population of inputs (energy):

```
https://www.google.com/ (1)
~~https://web.njit.edu/~mjk76/ (0.75)~~
https://calendar.google.com/calendar/u/0/r?cid=bWprNzZAbmppdC5lZHU
(1)
http://3.149.230.63:50000 (2)
http://f.149.230.63:50000 (2)
```

Fuzzing: agenda

- story time
- mutational fuzzing
- **grammar-based fuzzing**
- fuzzing in the real world
- start symbolic execution (if there is enough time left)

Grammar-based fuzzing

- Mutating seed inputs is effective in practice to find inputs that are "near" the seeds
- But usually **we know a lot** more about a program's input format!

Grammar-based fuzzing

- Mutating seed inputs is effective in practice to find inputs that are “near” the seeds
- But usually **we know a lot** more about a program’s input format!

```
scheme://netloc/path?query#fragment
```

Grammar-based fuzzing

- Mutating seed inputs is effective in practice to find inputs that are “near” the seeds
- But usually **we know a lot** more about a program’s input format!

```
scheme://netloc/path?query#fragment
```

- In our previous example, the fuzzer had a **equal chance** to mutate each character in the URL

Grammar-based fuzzing

- Mutating seed inputs is effective in practice to find inputs that are "near" the seeds
- But usually **we know a lot** more about a program's input format!

```
scheme://netloc/path?query#fragment
```

- In our previous example, the fuzzer had a **equal chance** to mutate each character in the URL
- But we know a lot more about how URLs are **structured**!

Grammar-based fuzzing

- Mutating seed inputs is effective in practice to find inputs that are "near" the seeds
- But usually **we know a lot** more about a program's input format!

```
scheme://netloc/
```

- In our previous example, the each character in the URL
- But we know a lot more about how URLs are **structured**!

Key idea: provide that structure to the fuzzer, and only select inputs that are valid!

Grammar-based fuzzing: review of grammars

- A ***formal grammar*** describes which strings from an alphabet of a formal language are valid according to the language's syntax.
[Wikipedia]

Grammar-based fuzzing: review of grammars

- A ***formal grammar*** describes which strings from an alphabet of a formal language are valid according to the language's syntax.
  [Wikipedia]
- For example, here is a grammar for URLs:

  URL = S :// N / P? `scheme://netloc/path?query#fragment`

  S = http | https | ftp | …

  N = any string

  P = any string / P | P ? Q | ε

  Q = any string | Q # F

  F = any string

Grammar-based fuzzing

Definition: a ***grammar-based fuzzer*** augments the input generation part of a fuzzer with a formal grammar, which is used to produce new valid inputs to the target program

Grammar-based fuzzing

Definition: a ***grammar-based fuzzer*** augments the input generation part of a fuzzer with a formal grammar, which is used to produce new valid inputs to the target program

- i.e., the seed inputs are **replaced** with the grammar, and the population is created by sampling from the grammar.

Grammar-based fuzzing

Definition: a ***grammar-based fuzzer*** augments the input generation part of a fuzzer with a formal grammar, which is used to produce new valid inputs to the target program

- i.e., the seed inputs are **replaced** with the grammar, and the population is created by sampling from the grammar.
- mutation changes from "change a random character" or similar to "change a part of the **derivation tree** for a term"

Grammar-based fuzzing: usefulness

- grammar-based fuzzing is useful for programs with **highly-structured**, **well-defined** inputs

Grammar-based fuzzing: usefulness

- grammar-based fuzzing is useful for programs with **highly-structured**, **well-defined** inputs
    - e.g., compilers, APIs, GUI applications

Grammar-based fuzzing: usefulness

- grammar-based fuzzing is useful for programs with **highly-structured**, **well-defined** inputs
  - e.g., compilers, APIs, GUI applications
- for such programs, providing a grammar can dramatically improve fuzzing efficiency

Grammar-based fuzzing: usefulness

- grammar-based fuzzing is useful for programs with **highly-structured**, **well-defined** inputs
  - e.g., compilers, APIs, GUI applications
- for such programs, providing a grammar can dramatically improve fuzzing efficiency
  - downside: someone usually has to write the grammar
  - but this is an area of active research!

Fuzzing: agenda

- story time
- mutational fuzzing
- grammar-based fuzzing
- **fuzzing in the real world**
- start symbolic execution (if there is enough time left)

Fuzzing in practice

- Fuzzing is **common in industry**
  - AFL (most famous coverage-guided fuzzer) was built at Google
  - oss-fuzz project fuzzes many important open-source projects constantly using industry resources

Fuzzing in practice

- Fuzzing is **common in industry**
  - AFL (most famous coverage-guided fuzzer) was built at Google
  - oss-fuzz project fuzzes many important open-source projects constantly using industry resources
- Fuzzing is **machine-intensive**
  - most inputs aren’t useful (grammars can help)

Fuzzing in practice

- Fuzzing is **common in industry**
  - AFL (most famous coverage-guided fuzzer) was built at Google
  - oss-fuzz project fuzzes many important open-source projects constantly using industry resources
- Fuzzing is **machine-intensive**
  - most inputs aren’t useful (grammars can help)
- Fuzzing **finds real bugs**
  - especially useful for finding security bugs

Fuzzing in practice: security

- **Why** is fuzzing useful for finding security bugs?

Fuzzing in practice: security

- **Why** is fuzzing useful for finding security bugs?
    - most common cause of vulnerabilities: **buffer overflows**

Fuzzing in practice: security

- **Why** is fuzzing useful for finding security bugs?
  - most common cause of vulnerabilities: **buffer overflows**
- It is straightforward to augment a fuzzer to detect buffer overflows in addition to crashes

Fuzzing in practice: security

- **Why** is fuzzing useful for finding security bugs?
  - most common cause of vulnerabilities: **buffer overflows**
- It is straightforward to augment a fuzzer to detect buffer overflows in addition to crashes
  - ~doubles running time for most C programs, but fuzzing is **already resource-intensive**

Fuzzing in practice: security

- **Why** is fuzzing useful for finding security bugs?
  - most common cause of vulnerabilities: **buffer overflows**
- It is straightforward to augment a fuzzer to detect buffer overflows in addition to crashes
  - ~doubles running time for most C programs, but fuzzing is **already resource-intensive**
  - fuzzers have detected many important security issues
    - e.g., Heartbleed in OpenSSL

Fuzzing: agenda

- story time
- mutational fuzzing
- grammar-based fuzzing
- fuzzing in the real world
- **start symbolic execution** (if there is enough time left)

Symbolic Execution

- we’ve seen coverage used as a fitness function for a fuzzer

Symbolic Execution

- we've seen coverage used as a fitness function for a fuzzer
  - but what if we just try to figure out **which inputs would improve coverage** directly?

Symbolic Execution

- we've seen coverage used as a fitness function for a fuzzer
    - but what if we just try to figure out **which inputs would improve coverage** directly?
- this is the key idea behind using **symbolic execution** to generate test inputs that improve coverage

Symbolic Execution

Definition: ***symbolic execution*** abstractly runs the target program while computing a **formula** for each variable

Symbolic Execution

Definition: ***symbolic execution*** abstractly runs the target program while computing a **formula** for each variable

- effectively, use math to figure out which values of each variable will cause the program to take particular paths

Symbolic Execution

Definition: ***symbolic execution*** abstractly runs the target program while computing a **formula** for each variable

- effectively, use math to figure out which values of each variable will cause the program to take particular paths
- our plan: choose an uncovered bit of code, and then symbolically execute **backwards** from there to figure out what values the input variables would need to take on in order to cover the code

Symbolic Execution

Definition: ***symbolic execution*** abstractly runs the target program while computing a **formula** for each variable

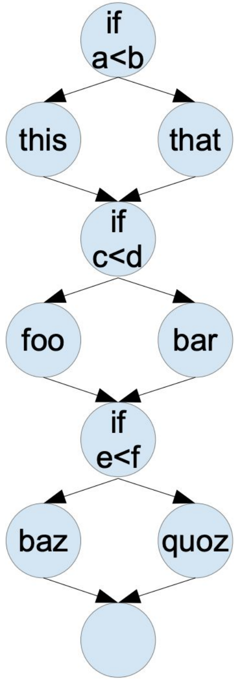
- effectively, use math to figure out which values of each variable will cause the program to take particular paths
- our plan: choose an uncovered bit of code, and then symbolically execute **backwards** from there to figure out what values the input variables would need to take on in order to cover the code
  - this is the **Lens of Logic** again, but applied in a different way

Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
   if a < b: this
   else: that
   if c < d: foo
   else: bar
   if e < f: baz
   else: quoz
```

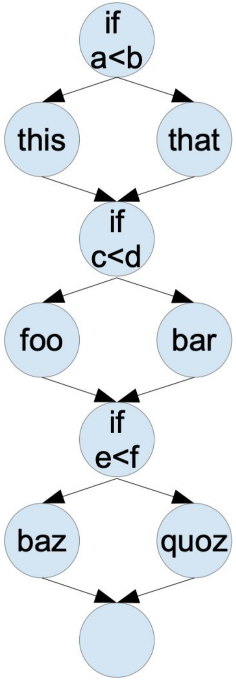
Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```



Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```

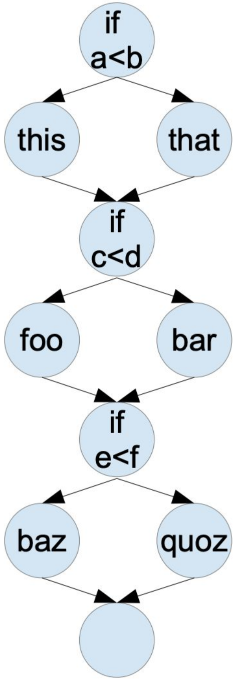


How would you choose inputs that **maximize**:

- line coverage?

Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```

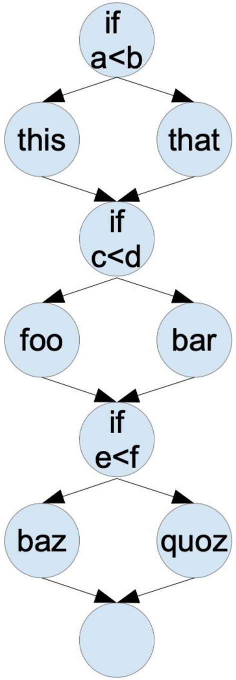


How would you choose inputs that **maximize**:

- **line** coverage?
- **branch** coverage?

Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```

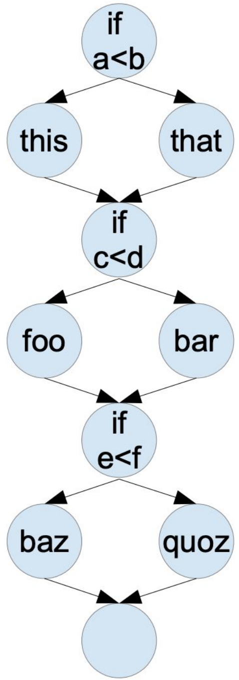


How would you choose inputs that **maximize**:

- line coverage?
- branch coverage?
- path coverage?

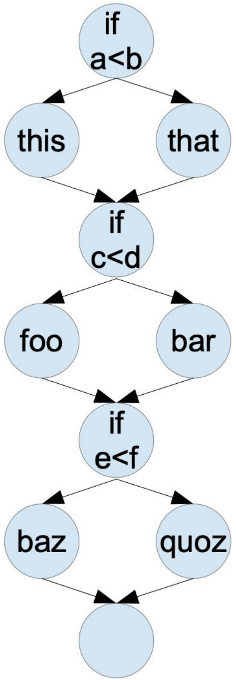
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …



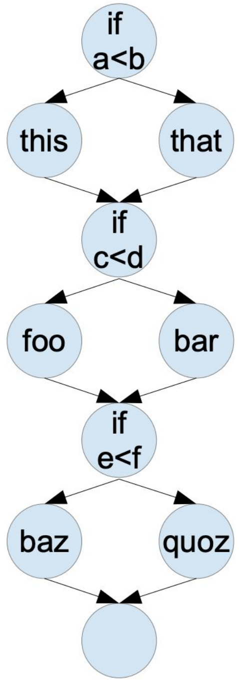
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …
- There are **2N** branch edges



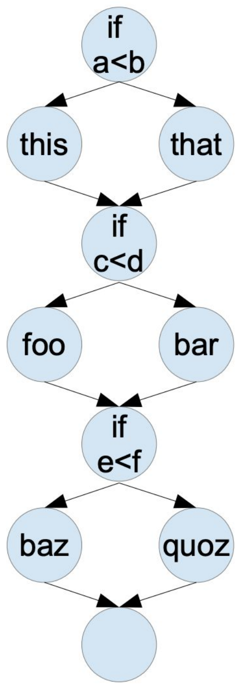
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …
- There are **2N** branch edges
    - Which you could cover in 2 tests!



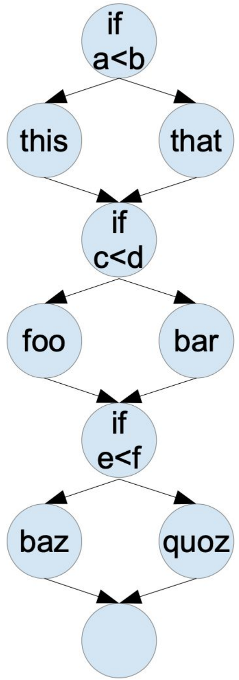
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements ...
- There are **2N** branch edges
  - Which you could cover in 2 tests!
    - One always goes left, one always right



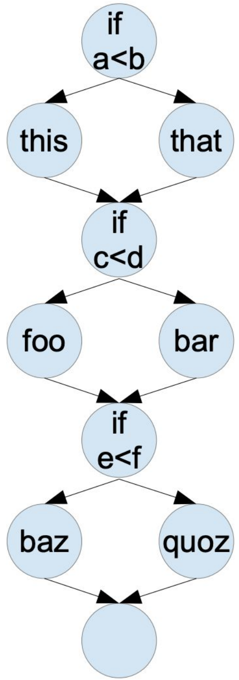
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements ...
- There are **2N** branch edges
  - Which you could cover in 2 tests!
    - One always goes left, one always right
- But there are **2^N paths**



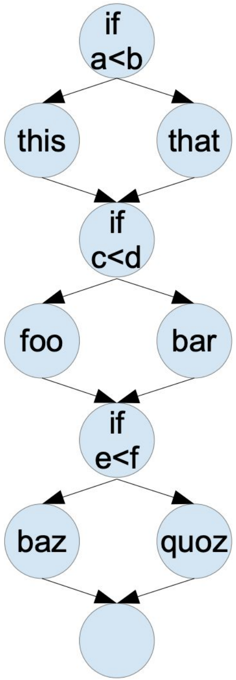
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements ...
- There are **2N** branch edges
  - Which you could cover in 2 tests!
    - One always goes left, one always right
- But there are **2^N paths**
  - You need **2^N tests** to cover them



Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …
- There are **2N** branch edges
  - Which you could cover in 2 tests!
    - One always goes left, one always right
- But there are **2^N paths**
  - You need **2^N tests** to cover them
- Path coverage **subsumes** branch coverage



Lens of Logic: maximize coverage

- Consider generating test inputs to cover a path

Lens of Logic: maximize coverage

- Consider generating test inputs to cover a path
  - If we could do that, branch/statement/etc coverage is easy

Lens of Logic: maximize coverage

- Consider generating test inputs to cover a path
  - If we could do that, branch/statement/etc coverage is easy
- **Key idea**: solve this problem with **math**

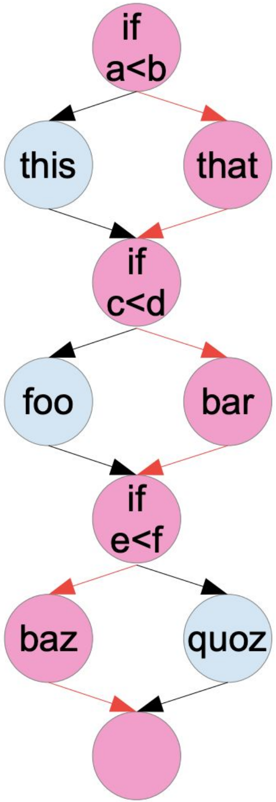
Lens of Logic: maximize coverage

- Consider generating test inputs to cover a path
  - If we could do that, branch/statement/etc coverage is easy
- **Key idea**: solve this problem with **math**

Definition: a ***path predicate*** (or ***path condition***, or ***path constraint***) is a boolean formula over program variables that is true when the program executes the given path

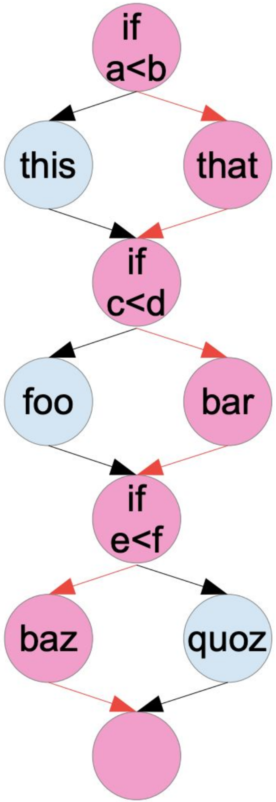
Lens of Logic: path predicate example

- Consider the highlighted (in **pink**) path
  - i.e., “false, false, true”
- What is its path predicate?



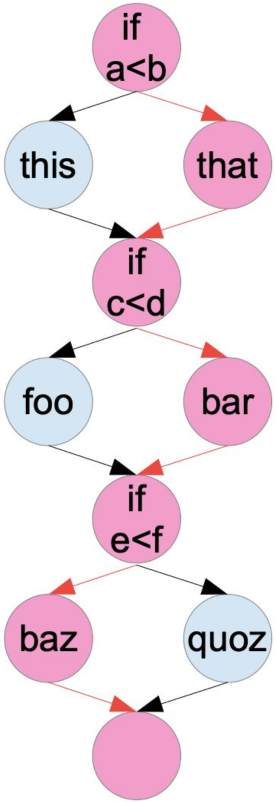
Lens of Logic: path predicate example

- Consider the highlighted (in **pink**) path
  - i.e., “false, false, true”
- What is its path predicate?
  - `a >= b && c >= d && e < f`



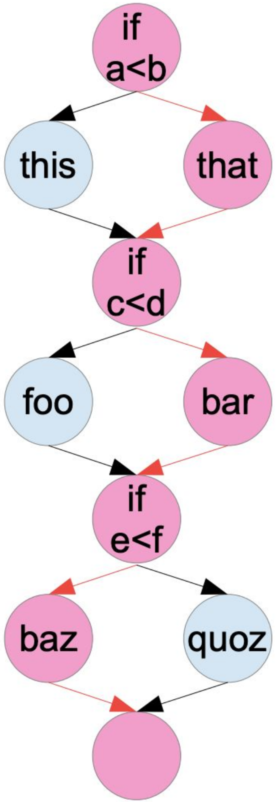
Lens of Logic: path predicate example

- Consider the highlighted (in **pink**) path
  - i.e., "false, false, true"
- What is its path predicate?
  - `a >= b && c >= d && e < f`
- When the path predicate is true, control flow will follow the given path



Lens of Logic: path predicate example

- Consider the highlighted (in **pink**) path
  - i.e., “false, false, true”
- What is its path predicate?
  - `a >= b && c >= d && e < f`
- When the path predicate is true, control flow will follow the given path
- So, given a path predicate, how do we choose a test input that covers the path?



Lens of Logic: solving path predicates

Definition: A ***satisfying assignment*** is a mapping from variables to values that makes a predicate true.

Lens of Logic: solving path predicates

Definition: A ***satisfying assignment*** is a mapping from variables to values that makes a predicate true.

- What is a satisfying assignment for
  - `a >= b && c >= d && e < f` ?

Lens of Logic: solving path predicates

Definition: A ***satisfying assignment*** is a mapping from variables to values that makes a predicate true.

- What is a satisfying assignment for
  - `a >= b && c >= d && e < f` ?
    - `a=5, b=4, c=3, d=2, e=1, f=2`
    - `a=0, b=0, c=0, d=0, e=0, f=1`
    - … many more

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?
  - Option 1: **ask humans**
    - labor-intensive, slow, expensive, etc.

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?
  - Option 1: **ask humans**
    - labor-intensive, slow, expensive, etc.
  - Option 2: repeatedly **guess randomly**
    - works surprisingly well (when answers are **not sparse**)

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?
    - Option 1: **ask humans**
        - labor-intensive, slow, expensive, etc.
    - Option 2: repeatedly **guess randomly**
        - works surprisingly well (when answers are **not sparse**)
    - Option 3: use an ***automated theorem prover***
        - cf. Wolfram Alpha, MatLab, Mathematica, Z3, etc.
        - works very well for a **restricted class of equations** (e.g., linear but not arbitrary polynomials, etc.)

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it
    - A solution is a satisfying assignment of values to input variables
      → those are your test input

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it
  - A solution is a satisfying assignment of values to input variables → those are your test input
  - None found? Dead code, tough predicate, etc.

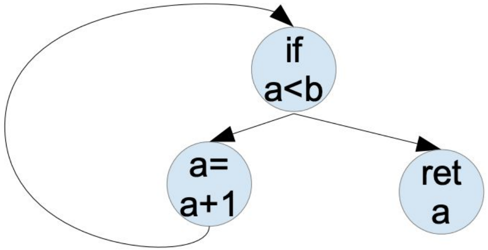
Lens of Logic: enumerating paths

- What could **go wrong** with enumerating paths in a method?

Lens of Logic: enumerating paths

- What could go wrong with enumerating paths in a method?
- There could be infinitely many!

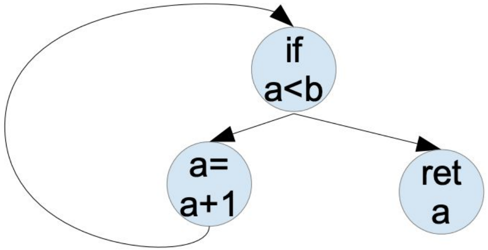
```
while a < b:
  a = a + 1
return a
```



Lens of Logic: enumerating paths

- What could **go wrong** with enumerating paths in a method?
- There could be **infinitely many**!

```
while a < b:
  a = a + 1
return a
```



- One path corresponds to executing the loop once, another to twice, another to three times, etc.

Lens of Logic: enumerating paths: approximation

- **Key idea**: don't enumerate all paths, **approximate** instead

Lens of Logic: enumerating paths: approximation

- **Key idea**: don't enumerate all paths, **approximate** instead
- Typical Approximations:

Lens of Logic: enumerating paths: approximation

- **Key idea**: don't enumerate all paths, approximate instead
- Typical Approximations:
    - Consider only acyclic paths (corresponds to taking each loop zero times or one time)

Lens of Logic: enumerating paths: approximation

- **Key idea**: don't enumerate all paths, approximate instead
- Typical Approximations:
  - Consider only **acyclic** paths (corresponds to taking each loop zero times or one time)
  - Consider only taking each loop **at most *k*** times

Lens of Logic: enumerating paths: approximation

- **Key idea**: don't enumerate all paths, **approximate** instead
- Typical Approximations:
  - Consider only **acyclic** paths (corresponds to taking each loop zero times or one time)
  - Consider only taking each loop **at most *k*** times
  - Enumerate paths breadth-first or depth-first and **stop after *k*** paths have been enumerated

Lens of Logic: enumerating paths: approximation

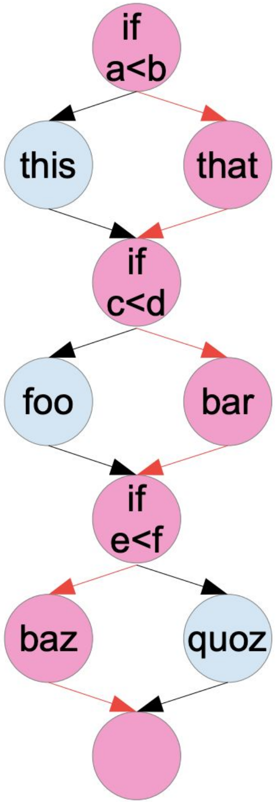
- **Key idea**: don't enumerate all paths, **approximate** instead
- Typical Approximations:
  - Consider only **acyclic** paths (corresponds to taking each loop zero times or one time)
  - Consider only taking each loop **at most *k*** times
  - Enumerate paths breadth-first or depth-first and **stop after *k*** paths have been enumerated
- For more on this topic, take a graduate-level course on static analysis or compilers

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** "all" paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it
  - A solution is a satisfying assignment of values to input variables → those are your test input
  - None found? Dead code, tough predicate, etc.

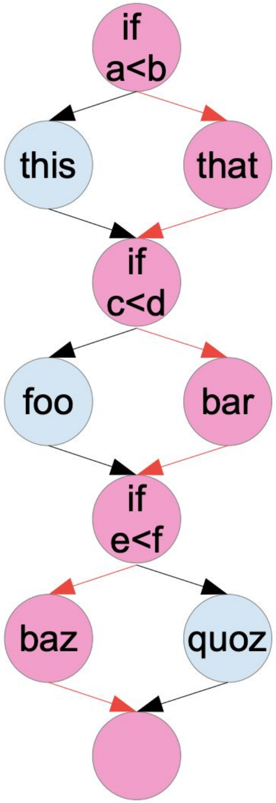
Lens of Logic: collecting path predicates

- Now we have a path through the program
- What could go wrong with **collecting** the path predicate?



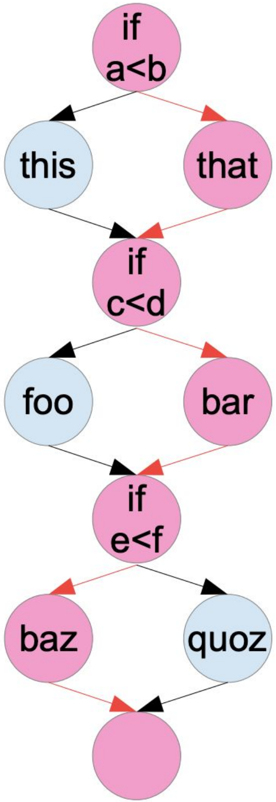
Lens of Logic: collecting path predicates

- Now we have a path through the program
- What could go wrong with **collecting** the path predicate?
  - The path predicate may not be **expressible** in terms of the inputs we control



Lens of Logic: collecting path predicates

- Now we have a path through the program
- What could go wrong with **collecting** the path predicate?
  - The path predicate may not be **expressible** in terms of the inputs we control



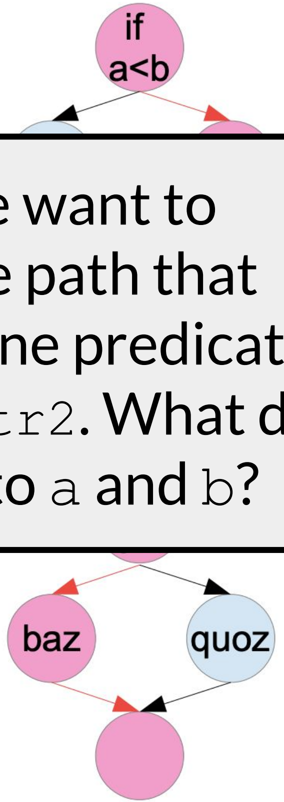

```
foo(a,b):
  str1 = read_from_url("abc.com")
  str2 = read_from_url("xyz.com")
  if (str1 == str2): bar()
```

Lens of Logic: collecting path predicates

- Now we have a path through the program
- What could go wrong with **collecting** th
  predicate?
  - The path predicate may not be **expr**
    terms of the inputs we control

Suppose we want to exercise the path that calls `bar`. One predicate is `str1==str2`. What do you assign to `a` and `b`?

```
foo(a,b):
  str1 = read_from_url("abc.com")
  str2 = read_from_url("xyz.com")
  if (str1 == str2): bar()
```



Lens of Logic: path predicate woes

- When we can’t solve for a path predicate, what can we do?

Lens of Logic: path predicate woes

- When we can’t solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don’t generate a test)

Lens of Logic: path predicate woes

- When we can’t solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don’t generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way

Lens of Logic: path predicate woes

- When we can't solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don't generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way
- So, we make a **best effort**:

Lens of Logic: path predicate woes

- When we can’t solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don’t generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way
- So, we make a **best effort**:
  - Collect the path predicates as best we can

Lens of Logic: path predicate woes

- When we can't solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don't generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way
- So, we make a **best effort**:
  - Collect the path predicates as best we can
  - Ask the solver to find a solution in terms of the input variables

Lens of Logic: path predicate woes

- When we can't solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don't generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way
- So, we make a **best effort**:
  - Collect the path predicates as best we can
  - Ask the solver to find a solution in terms of the input variables
  - If it can't (because the math is too hard, we don't control the input, etc.), we give up

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it
  - A solution is a satisfying assignment of values to input variables → those are your test input
  - None found? Dead code, tough predicate, etc.

Lens of Logic: test input generation plan

- Recall: we want to automatically generate **test cases**

Lens of Logic: test input generation plan

- Recall: we want to automatically generate **test cases**
- We have an approach that works well in practice:
  - **Enumerate** some paths
  - **Extract** their path constraints
  - **Solve** those path constraints

Today's in-class: HW3

- you'll use the AFL fuzzer to generate tests for libpng (same target as last week's homework)

Today’s in-class: HW3

- you’ll use the AFL fuzzer to generate tests for libpng (same target as last week’s homework)
- **warning**: AFL can take a long time to achieve the needed coverage (especially in a VM)

Today’s in-class: HW3

- you’ll use the AFL fuzzer to generate tests for libpng (same target as last week’s homework)
- **warning**: AFL can take a long time to achieve the needed coverage (especially in a VM)
    - good news: it can run by itself, so you can leave it overnight

Today’s in-class: HW3

- you’ll use the AFL fuzzer to generate tests for libpng (same target as last week’s homework)
- **warning**: AFL can take a long time to achieve the needed coverage (especially in a VM)
    - good news: it can run by itself, so you can leave it overnight
    - bad news: you can’t start this homework the day before it’s due

Today's in-class: HW3

- you'll use the AFL fuzzer to generate tests for libpng (same target as last week's homework)
- **warning**: AFL can take a long time to achieve the needed coverage (especially in a VM)
  - good news: it can run by itself, so you can leave it overnight
  - bad news: you can't start this homework the day before it's due
- note: there is no autograder for this assignment. You only need to turn in a written report (but to write the report, you'll need data from AFL that you can only get by running it on libpng)