Week 4: Lecture A

Input Generation

Monday, January 29, 2024

Recap: Lab 1

- **Lab 1: Beginner Fuzzing (due 2/07 by 11:59PM)**
  - Familiarize yourself with AFL++ and its features
  - Check out its documentation in **docs/**
- **Pick three features, evaluate them, and discuss your findings**
  - E.g., impacts on code coverage, speed, crash discovery
  - What insights do you have?
  - Why did one feature work better than another?
- **Deliverable:** a **1–3 page report** detailing your findings
  - Feel free to make it your own (e.g., pictures, text, etc.)
- Need a Linux environment
  - Use the **CS 4440 VM** if you don't have one!

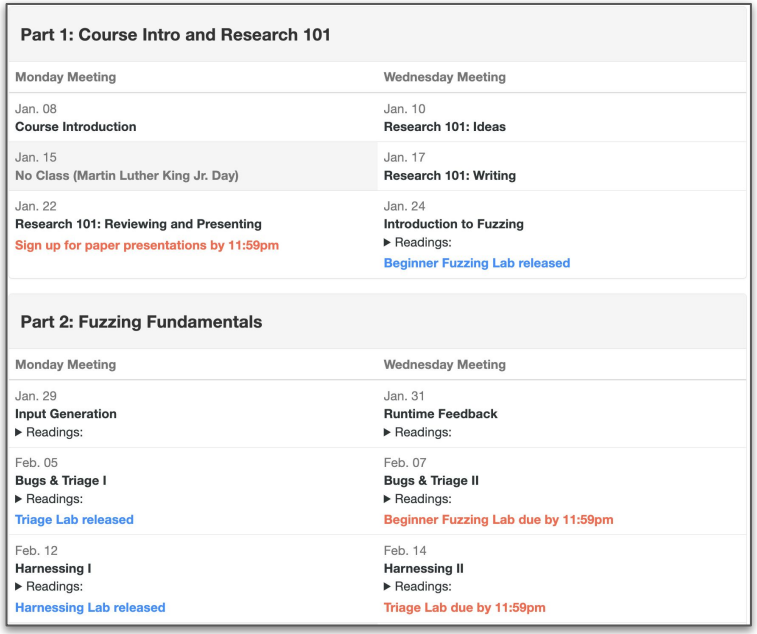
Recap: Lab 1

- Pick any **target program** you like, e.g.:
  - FuzzGoat fuzzing benchmark
  - FoRTE-FuzzBench
  - HexHive's Magma
- Skills you'll learn along the way:
  - **Compiling** a C/C++ program
  - Inserting AFL++'s **instrumentation**
  - Initiating **fuzzing** with AFL++
  - Interpreting AFL++'s **results**

Recap: Key Dates

- **Jan. 24** — **Lab 1 released**
- **Feb. 07** — **Lab 1 due**
- **Feb. 14** — Lab 2 due
- **Feb. 19** — No class (President's Day)
- **Feb. 28** — Lab 3 due
- **Feb. 28** — **5-minute project proposals**
- **Mar. 04 & 06** — No class (Spring Break)
- **Apr. 17 & 22** — **Final project presentations**

cs.utah.edu/~snagy/courses/cs5963/schedule



Part 1: Course Intro and Research 101

Monday Meeting	Wednesday Meeting
Jan. 08 **Course Introduction**	Jan. 10 **Research 101: Ideas**
Jan. 15 **No Class (Martin Luther King Jr. Day)**	Jan. 17 **Research 101: Writing**
Jan. 22 **Research 101: Reviewing and Presenting** Sign up for paper presentations by 11:59pm	Jan. 24 **Introduction to Fuzzing** ▸ Readings: Beginner Fuzzing Lab released

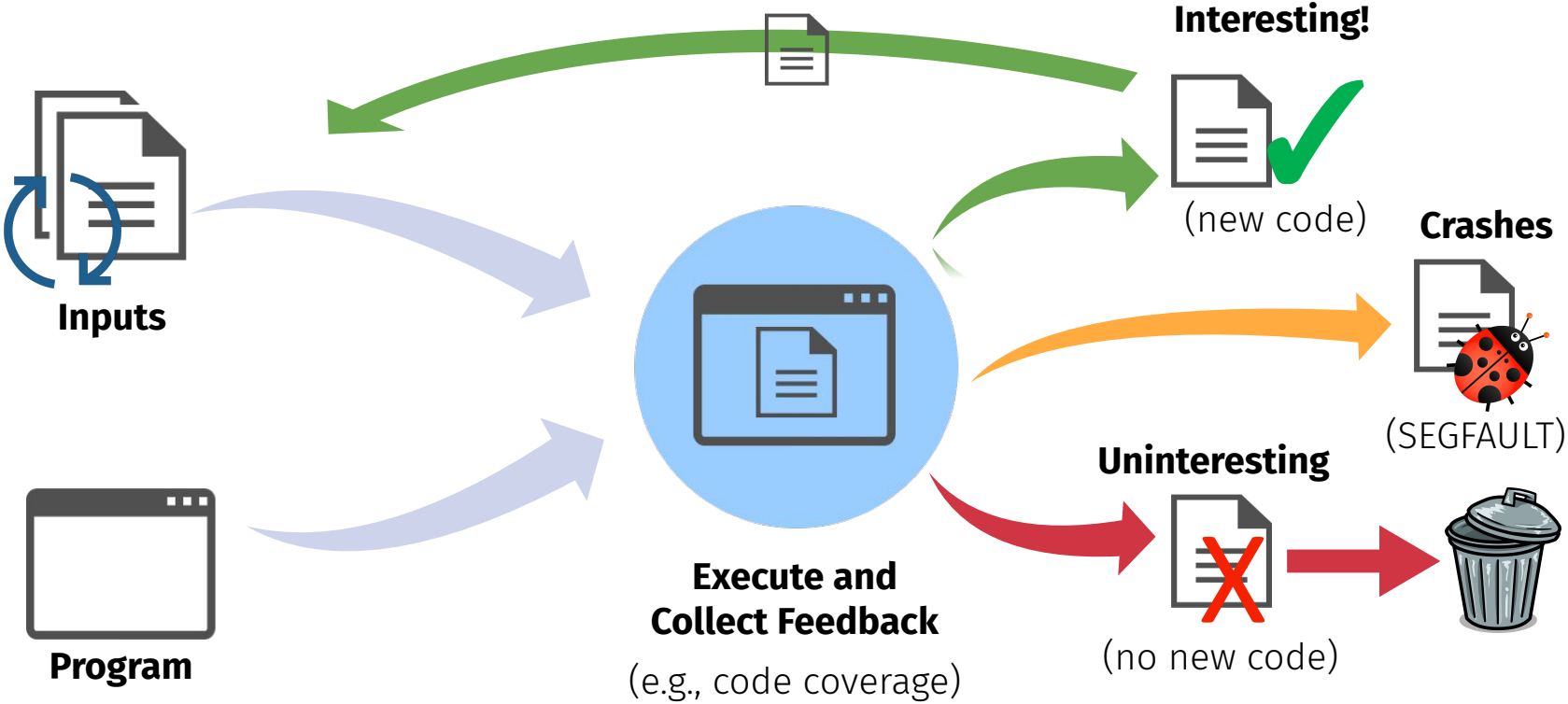
Part 2: Fuzzing Fundamentals

Monday Meeting	Wednesday Meeting
Jan. 29 **Input Generation** ▸ Readings:	Jan. 31 **Runtime Feedback** ▸ Readings:
Feb. 05 **Bugs & Triage I** ▸ Readings: Triage Lab released	Feb. 07 **Bugs & Triage II** ▸ Readings: Beginner Fuzzing Lab due by 11:59pm
Feb. 12 **Harnessing I** ▸ Readings: Harnessing Lab released	Feb. 14 **Harnessing II** ▸ Readings: Triage Lab due by 11:59pm

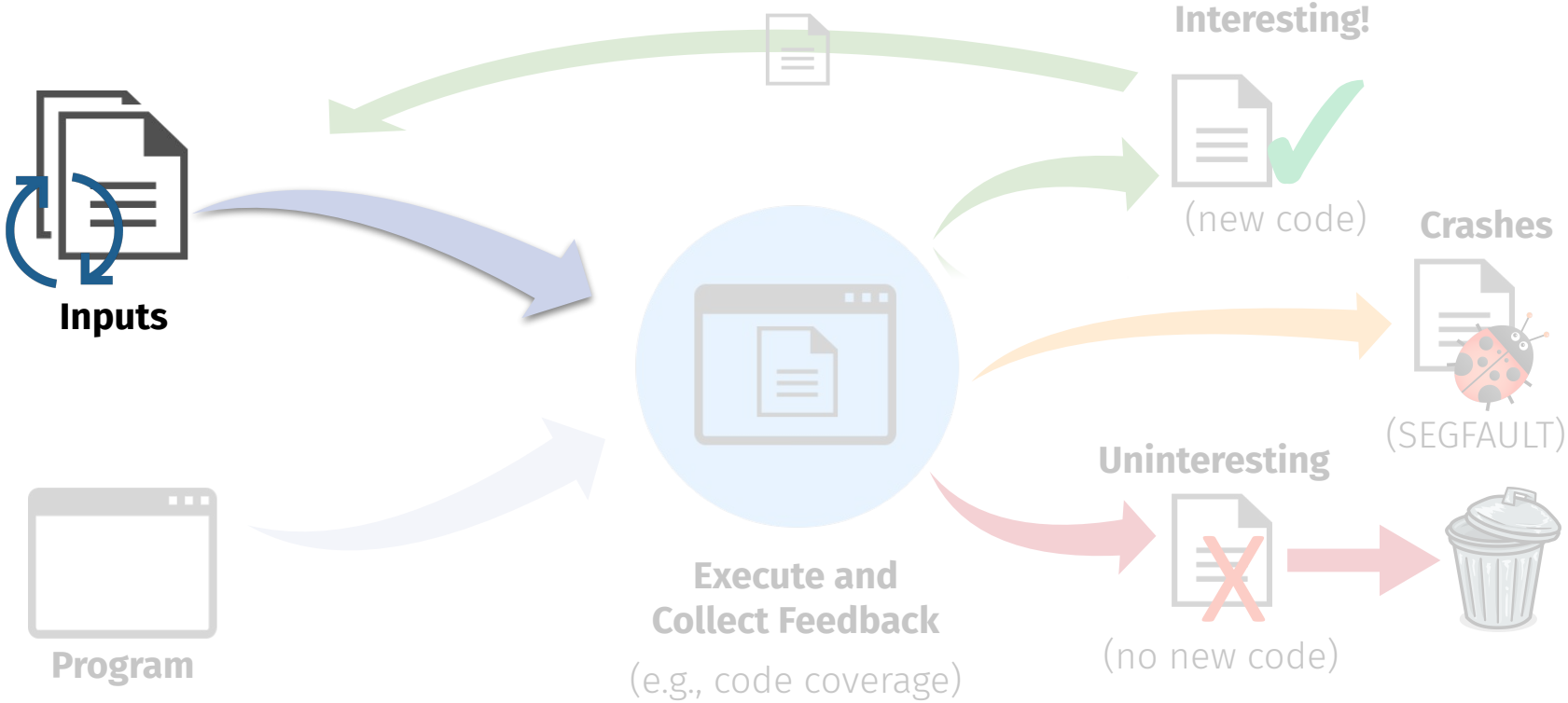
Questions?

Input Generation

Recap: Coverage-guided Fuzzing



Recap: Coverage-guided Fuzzing



Types of Input Generation

- **Model-agnostic: brute-force** your way to valid inputs
  - Random insertions, deletions, and splicing
- **Model-guided:** follow a **pre-defined** input specification
  - Follow “rules” to create highly-structured inputs
- **White-box approaches:**
  - **Symbolic execution:** solve branches as **symbolic** expressions
  - **Concolic execution:** solve branches as **concrete** values
  - **Taint tracking:** infer critical input **“parts”** and mutate those

Source: The Art, Science, and Engineering of Fuzzing: A Survey

Before you start: choose your seeds!

- **Seeds:** the **starting inputs** from which to mutate from

Before you start: choose your seeds!

- **Seeds:** the **starting inputs** from which to mutate from
- Small seeds
  - E.g., the smallest-possible PDF file
  - E.g., an empty file

small.pdf

```
 0 25504446 2D312E30 0A312030 206F626A 3C3C2F54 7970652F 43617461 6C6F672F  %PDF-1.0 1 0 obj<</Type/Catalog/
32 50616765 73203220 3020523E 3E656E64 6F626A20 32203020 6F626A3C 3C2F5479  Pages 2 0 R>>endobj 2 0 obj<</Ty
64 70652F50 61676573 2F4B6964 735B3320 3020525D 2F436F75 6E742031 3E3E656E  pe/Pages/Kids[3 0 R]/Count 1>>en
96 646F626A 20332030 206F626A 3C3C2F54 7970652F 50616765 2F4D6564 6961426F  dobj 3 0 obj<</Type/Page/MediaBo
128 785B3020 30203320 335D3E3E 656E646F 626A2074 7261696C 65723C3C 2F53697A  x[0 0 3 3]>>endobj trailer<</Siz
160 6520342F 526F6F74 20312030 20523E3E                                      e 4/Root 1 0 R>>
```

```
steve@stefansacbookm1 Downloads % file small.pdf
small.pdf: PDF document, version 1.0, 1 pages
```

Before you start: choose your seeds!

- **Seeds:** the **starting inputs** from which to mutate from
- Small seeds
  - E.g., the smallest-possible PDF file
  - E.g., an empty file

small.pdf

```
  0 25504446 2D312E30 0A312030 206F626A 3C3C2F54 7970652F 43617461 6C6F672F  %PDF-1.0 1 0 obj<</Type/Catalog/
 32 50616765 73203220 3020523E 3E656E64 6F626A20 32203020 6F626A3C 3C2F5479  Pages 2 0 R>>endobj 2 0 obj<</Ty
 64 70652F50 61676573 2F4B6964 735B3320 3020525D 2F436F75 6E742031 3E3E656E  pe/Pages/Kids[3 0 R]/Count 1>>en
 96 646F626A 20332030 206F626A 3C3C2F54 7970652F 50616765 2F4D6564 6961426F  dobj 3 0 obj<</Type/Page/MediaBo
128 785B3020 30203320 335D3E3E 656E646F 626A2074 7261696C 65723C3C 2F53697A  x[0 0 3 3]>>endobj trailer<</Siz
160 6520342F 526F6F74 20312030 20523E3E                                      e 4/Root 1 0 R>>
```

Provides a fuzzer the “ingredients” to **pass** the program’s initial **input-parsing logic**!

```
if self.token[1][0] == '%':
elif self.token[1] == '/':
elif self.token[1] == 'trailer':
if self.token[1] == 'endobj':
```

Before you start: choose your seeds!

- **Seeds:** the **starting inputs** from which to mutate from
- Small seeds
  - E.g., the smallest-possible PDF file
  - E.g., an empty file
- Large seeds
  - E.g., crawl web for every PDF ever created
  - E.g., **243,246** SSL/TLS certificates

Before you start: choose your seeds!

- **Seeds:** the **starting inputs** from which to mutate from
  - Small seeds
    - E.g., the smallest-possible PDF file
    - E.g., an empty file
  - Large seeds
    - E.g., crawl web for every PDF ever created
    - E.g., **243,246** SSL/TLS certificates

- **No right answer—it is target-dependent!**
  - **Smaller seeds** = cover **earlier** code, but struggle to reach **deeper** code
  - **Larger seeds** = cover **deeper** code to start, but are **slower** to execute

Before you start: choose your seeds!

- **Seeds:** the **starting inputs** from which to mutate from

- Publicly-available seed corpora:
  - AFLplusplus/testcases directory
    - A few basic file formats
    - Images, PDF, MP4, etc.
  - My own fuzzing-seeds repo
    - Lots of seed corpora
    - Many file formats

Model-agnostic Generation

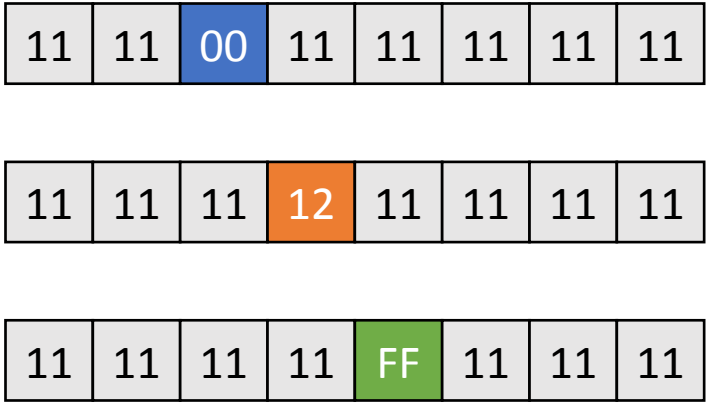
Model-agnostic Generation

- Brute-force your way to valid inputs
  - Bit and byte “flipping”
  - Addition and subtraction
  - Inserting random chunks
  - Inserting dictionary “tokens”

- **The good:** super fast
  - Incorporating feedback like coverage enables you to **synthesize valid inputs** (eventually)

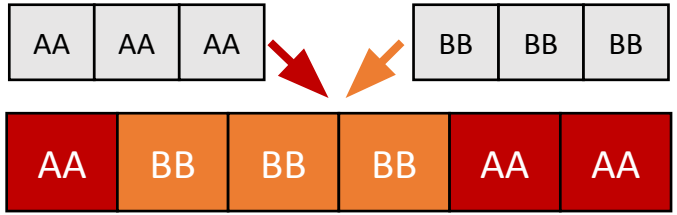
AFL's Model-agnostic Mutators

- **Deterministic mutation**
  - Bit and byte flips
    - Single, two, or four bits in a row
  - Arithmetic operators
    - Additions/subtractions of both endians
  - Inject "fun" values (-1, 256, 1024, etc.)
    - Values that often cause weird behavior



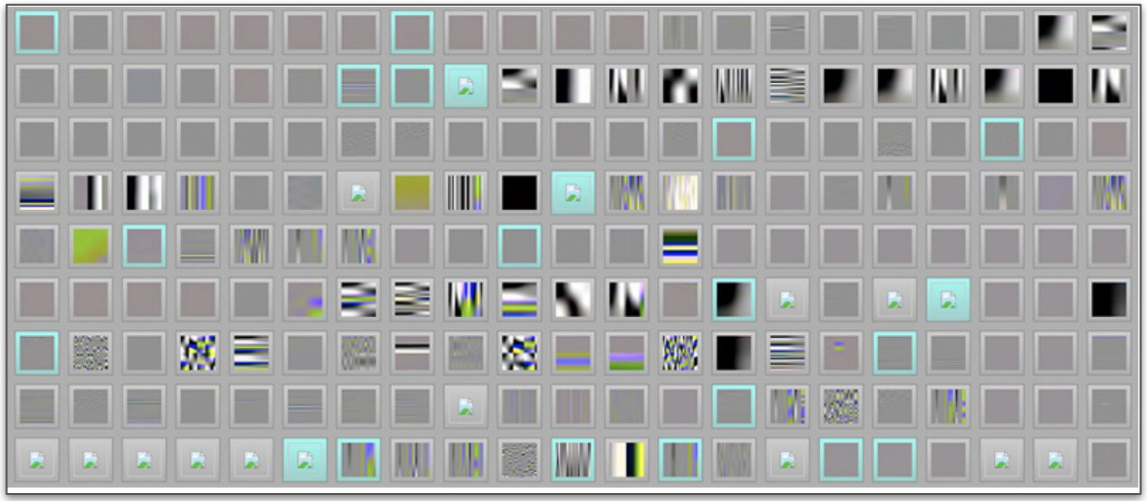
AFL's Model-agnostic Mutators (cont.)

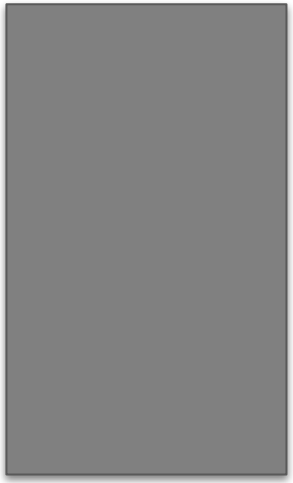
- **Non-deterministic mutation**
  - Performed on each input after deterministic mutations is exhausted or skipped entirely
  - Stacked tweaks
    - Randomly apply multiple det. Mutations
    - Clone / remove parts of the input
  - Test case splicing
    - Cuts two distinct inputs at random split points and fuses them



Trade-offs

- **Surprisingly effective:** valid inputs appear out of thin air





Trade-offs

- **Need a lot of luck** to solve magic bytes checks and nested checksums

```
if(u64(input)==u64("MAGICHDR"))
  bug(1);
```

Listing 2: Fuzzing problem (1): finding valid input to bypass magic bytes.

```
if(u64(input)==sum(input+8, len-8))
  if(u64(input+8)==sum(input+16, len-16))
    if(input[16]=='R' && input[17]=='Q')
      bug(2);
```

Listing 3: Fuzzing problem (2): finding valid input to bypass checksums.

Dictionary Tokens

- **Other “fun” values**
  - Program-specific magic bytes
    - cmp operands
    - `strcmp` operands
  - Input-specific magic bytes
    - Headers
    - Common attributes
- **Useful… but often noisy**
  - Not every cmp is relevant to an input’s structure

```
if (strcmp(header.magic_password, "h4ck3d by p1gZ")) goto terminate_now;
...or:
if (header.magic_value == 0x12345678) goto terminate_now;
```

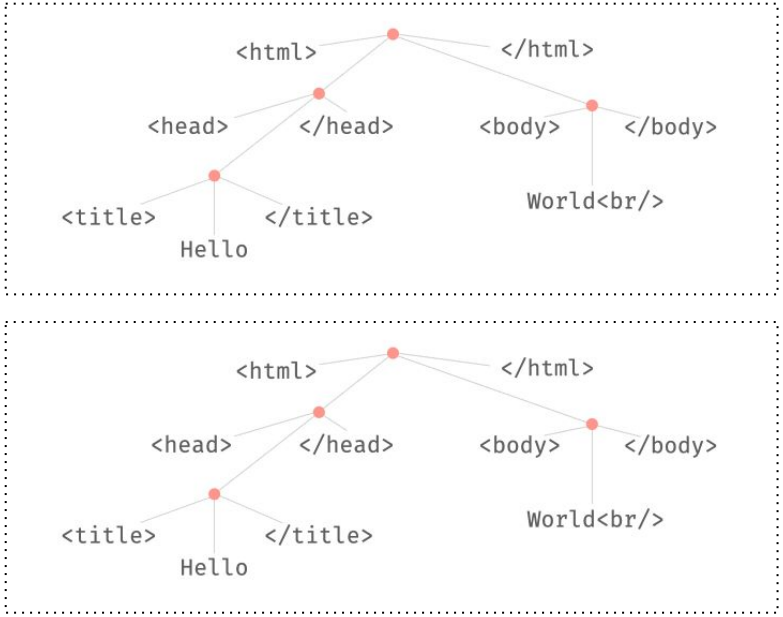
```
<html><header><title>Hello</title></header>
<body>World<br/></body></html>
```

```
<a>  <a/>
</a> ='a'
```

Model-guided Generation

Model-guided Generation

- Follow a **pre-defined** input specification
  - Pre-defined input grammars
  - Dynamically-learned grammars
  - Domain-specific generators
- **The good:** many more valid inputs
  - Model-agnostic inputs are often discarded because they fail basic input sanity checks
  - Valid inputs = **higher code coverage**



Pre-defined Models

- **Input grammars**
  - Usually handwritten
    - Domain expert
  - Many grammars already
    - ANTLR format
    - Kaitai structs

```
XML_GRAMMAR: Grammar = {
    "<start>": ["<xml-tree>"],
    "<xml-tree>": ["<text>",
                   "<xml-open-tag><xml-tree><xml-close-tag>",
                   "<xml-openclose-tag>",
                   "<xml-tree><xml-tree>"],
    "<xml-open-tag>":      ["<<id>>", "<<id> <xml-attribute>>"],
    "<xml-openclose-tag>": ["<<id>/>", "<<id> <xml-attribute>/>"],
    "<xml-close-tag>":     ["</<id>>"],
    "<xml-attribute>":     ["<id>=<id>", "<xml-attribute> <xml-attribute>"],
    "<id>":                ["<letter>", "<id><letter>"],
    "<text>":              ["<text><letter_space>", "<letter_space>"],
    "<letter>":            srange(string.ascii_letters + string.digits +
                                  "\"" + "'" + "."),
    "<letter_space>":      srange(string.ascii_letters + string.digits +
                                  "\"" + "'" + " " + "\t"),
}
```

Dynamically-learned Models

- **Infer grammars on-the-fly**
  - Learn before fuzzing starts
    - Scan program for useful data
    - Piece together grammar
  - Learn during fuzzing
    - Build state machine
    - Parse inputs accordingly
    - Refine on each iteration

Domain-specific Generators

- **Hand-written tools to spit-out conforming inputs**

- Famous examples
  - `CSmith`: C programs
  - `JSFunFuzz`: Javascript
  - `DOMFuzz`: DOM interface

- **Frameworks for writing your own**
  - `XSmith`
  - `FormatFuzzer`
  - `FuzzFactory`

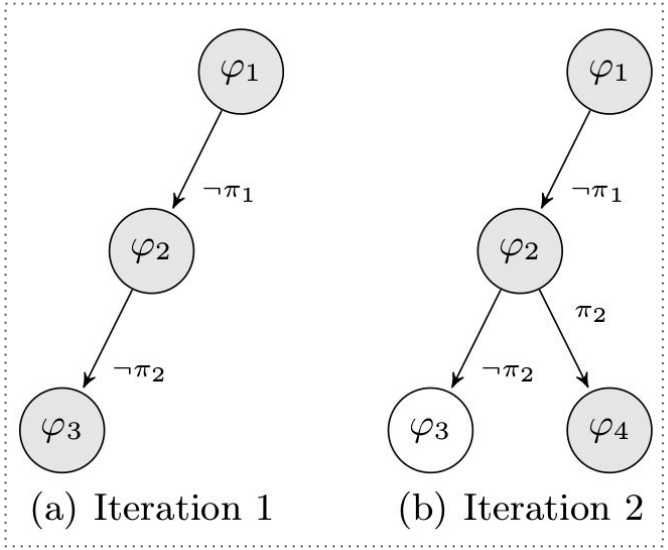
Trade-offs

- **Writing or learning specifications is hard**
  - E.g., `CSmith` written in **40,000+** LoC
  - Domain expertise is critical
- **Seemingly impossible for many inputs**
  - For example, no grammar for x86 binaries
- **Deeper coverage is not always better**
  - Likely to miss bugs hidden in shallow code (e.g., input validity checks)

White-box Input Generation

Symbolic and Concolic Execution

- Model paths as **symbolic expressions**
  - Construct a system of boolean equations
  - Pass this off to an SMT solver
  - Attempt to find all satisfiable assignments
  - **Concolic execution:** test *one* concrete path
- Many solvers available today
  - E.g., Z3, Yices, CVC4
- **The good:** great for many branches
  - Cuts through magic bytes without much trouble



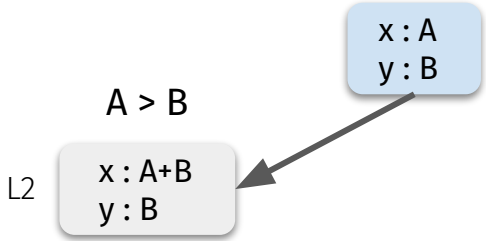
Symbolic Execution Example

```
0. def f (x, y):
1.   if (x > y):
2.     x = x + y
3.     y = x - y
4.     x = x - y
5.     if (x - y > 0):
6.       assert false
7.   return (x, y)
```

```
x:A
y:B
```

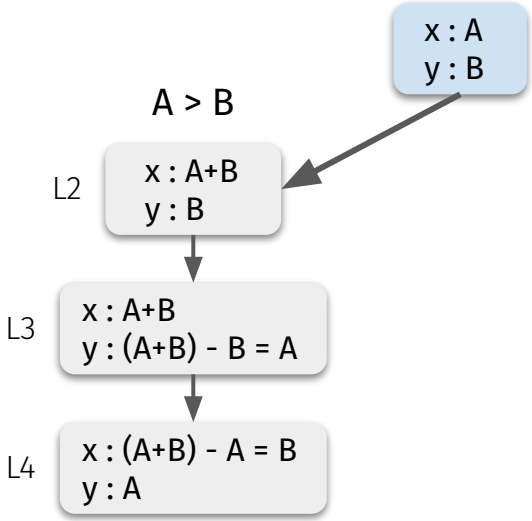
Symbolic Execution Example

```
0. def f (x, y):
1.   if (x > y):
2.     x = x + y
3.     y = x - y
4.     x = x - y
5.     if (x - y > 0):
6.       assert false
7.   return (x, y)
```



Symbolic Execution Example

```
0. def f (x, y):
1.   if (x > y):
2.     x = x + y
3.     y = x - y
4.     x = x - y
5.     if (x - y > 0):
6.       assert false
7.   return (x, y)
```

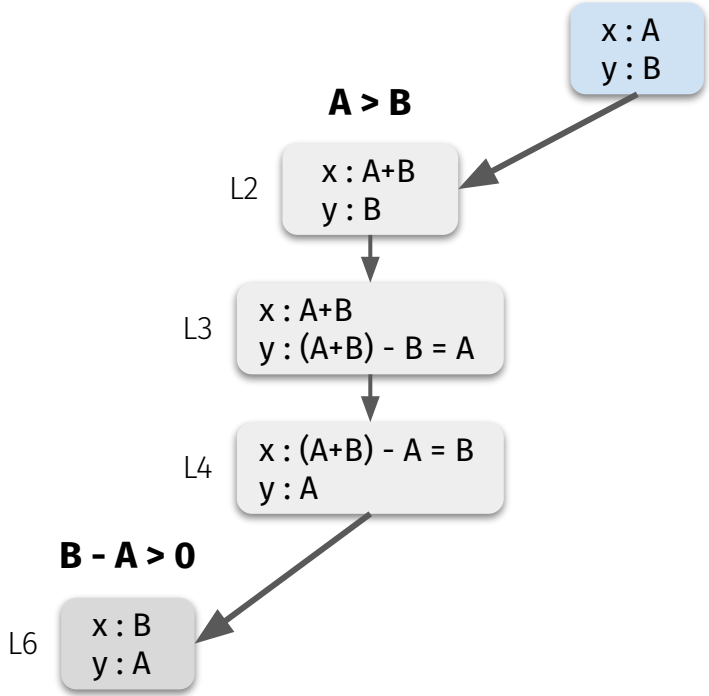


Symbolic Execution Example

```
0. def f (x, y):
1.   if (x > y):
2.     x = x + y
3.     y = x - y
4.     x = x - y
5.     if (x - y > 0):
6.       assert false
7.   return (x, y)
```

Possible path constraints:

- (A > B) and (B-A > 0) = **satisfiable?**

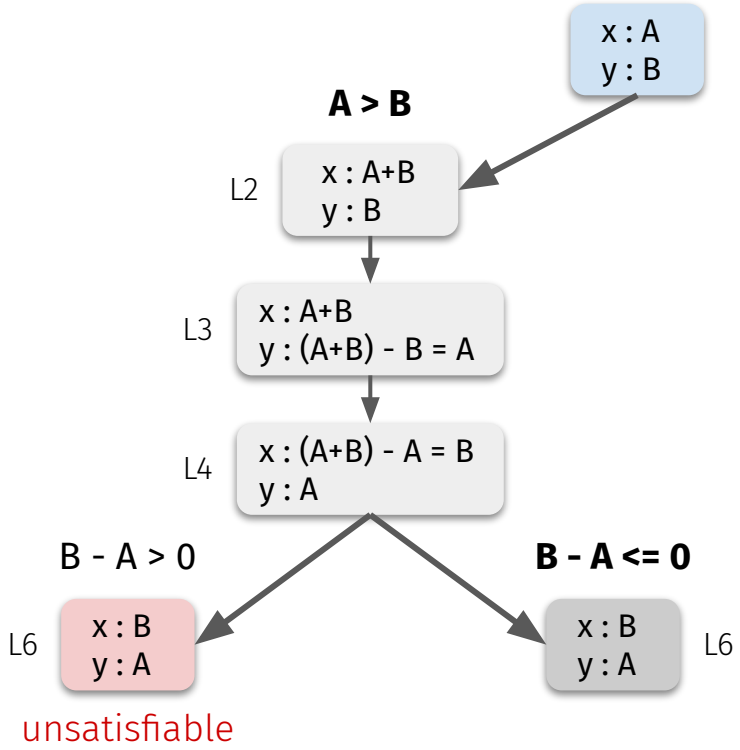


Symbolic Execution Example

```
0. def f (x, y):
1.   if (x > y):
2.     x = x + y
3.     y = x - y
4.     x = x - y
5.     if (x - y > 0):
6.       assert false
7.   return (x, y)
```

Possible path constraints:

- (A > B) and (B-A > 0) = unsatisfiable
- (A > B) and (B-A <= 0) = **satisfiable?**

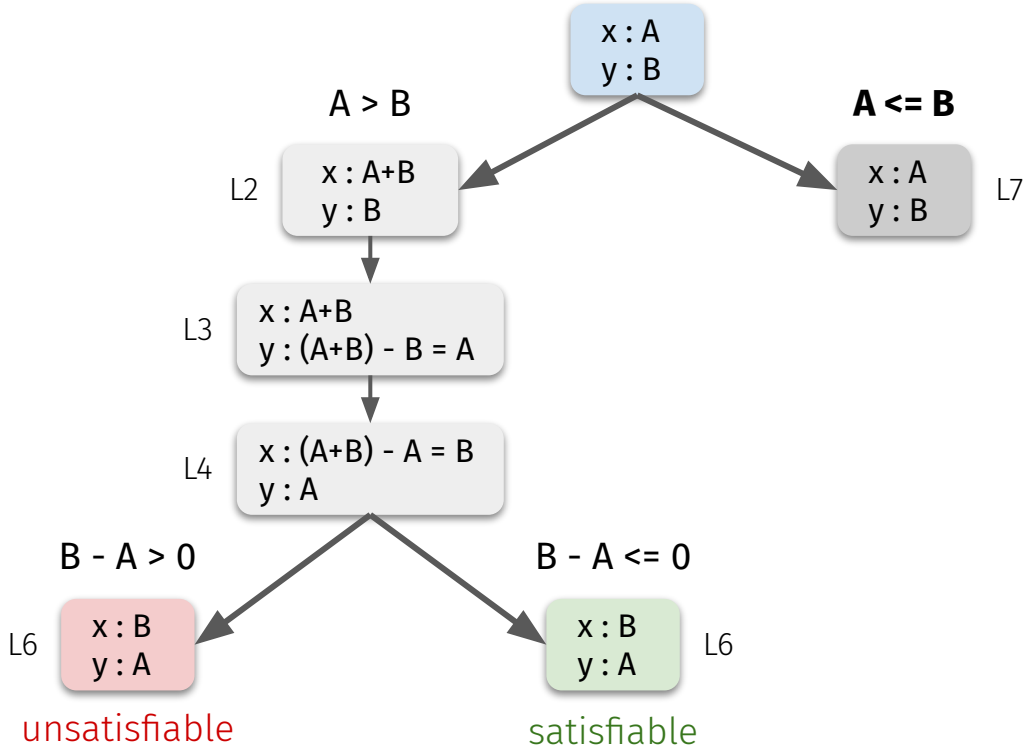


Symbolic Execution Example

```
0. def f (x, y):
1.   if (x > y):
2.     x = x + y
3.     y = x - y
4.     x = x - y
5.     if (x - y > 0):
6.       assert false
7.   return (x, y)
```

Possible path constraints:

- (A > B) and (B-A > 0) = unsatisfiable
- (A > B) and (B-A <= 0) = satisfiable
- (A <= B) = **satisfiable?**

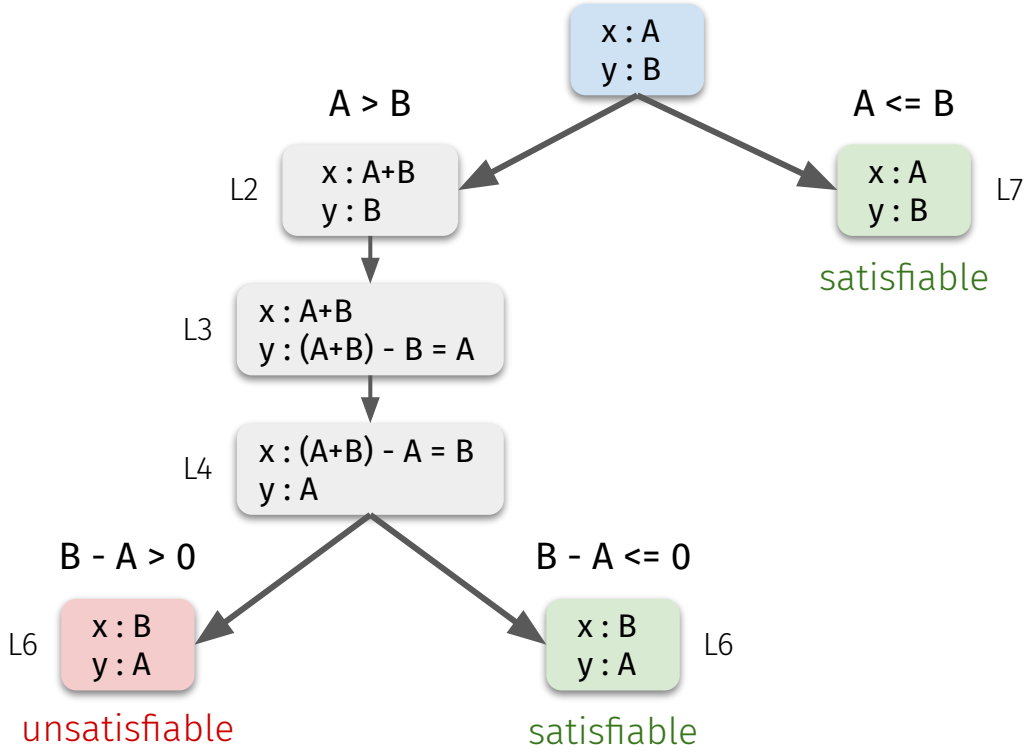


Symbolic Execution Example

```
0. def f (x, y):
1.   if (x > y):
2.     x = x + y
3.     y = x - y
4.     x = x - y
5.     if (x - y > 0):
6.       assert false
7.   return (x, y)
```

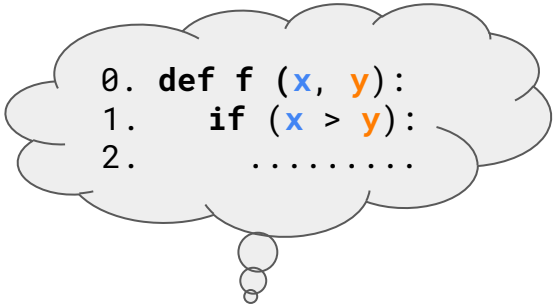
Possible path constraints:

- (A > B) and (B-A > 0) = unsatisfiable
- (A > B) and (B-A <= 0) = satisfiable
- (A <= B) = satisfiable



Taint Tracking

- Track input bytes' flow throughout program
  - Identify input "chunks" that affect program state
    - Chunks that affect branches
    - Chunks that flow to function calls
  - **Mutate these chunks**
    - Random mutation
    - Insert fun or useful tokens

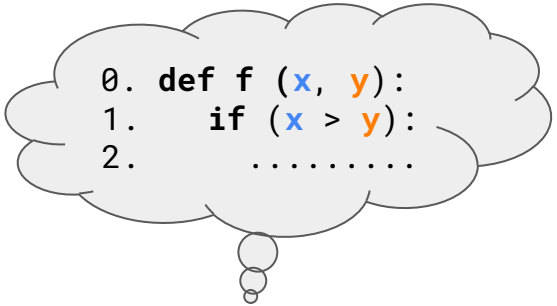


11	11	11	11	11	11	11	11
11	11	11	11	11	11	11	11

- **The good:** finding vulnerable buffers, solving branches

Taint Tracking

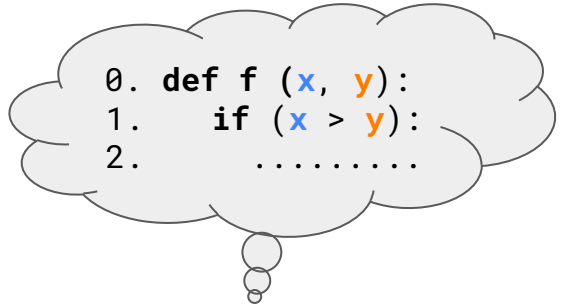
- Track input bytes' flow throughout program
  - Identify input "chunks" that affect program state
    - Chunks that affect branches
    - Chunks that flow to function calls
  - **Mutate these chunks**
    - Random mutation
    - Insert fun or useful tokens

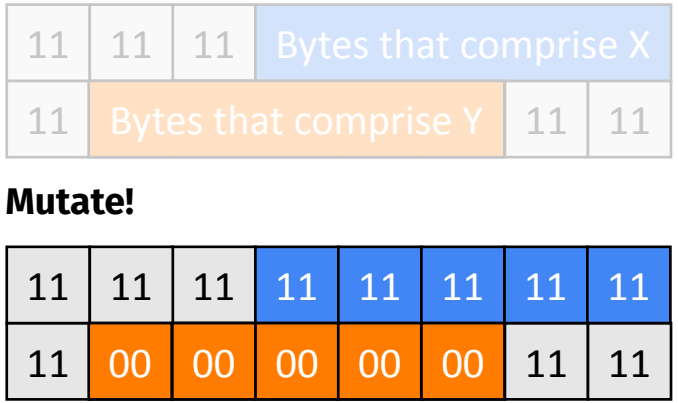


- **The good:** finding vulnerable buffers, solving branches

Taint Tracking

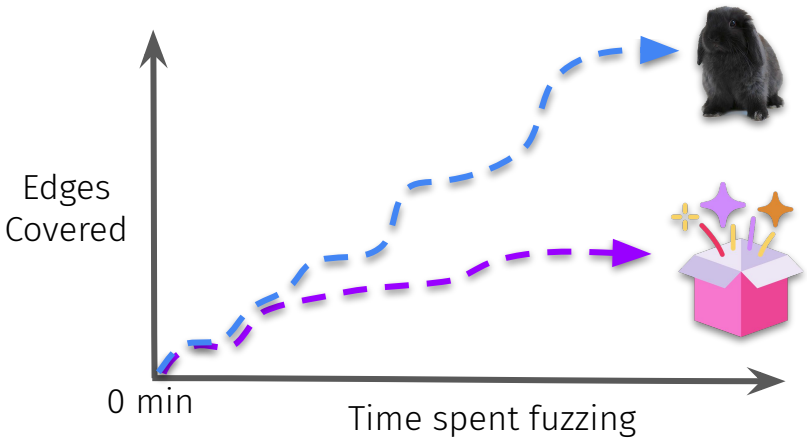
- Track input bytes' flow throughout program
  - Identify input "chunks" that affect program state
    - Chunks that affect branches
    - Chunks that flow to function calls
  - **Mutate these chunks**
    - Random mutation
    - Insert fun or useful tokens
- **The good:** finding vulnerable buffers, solving branches





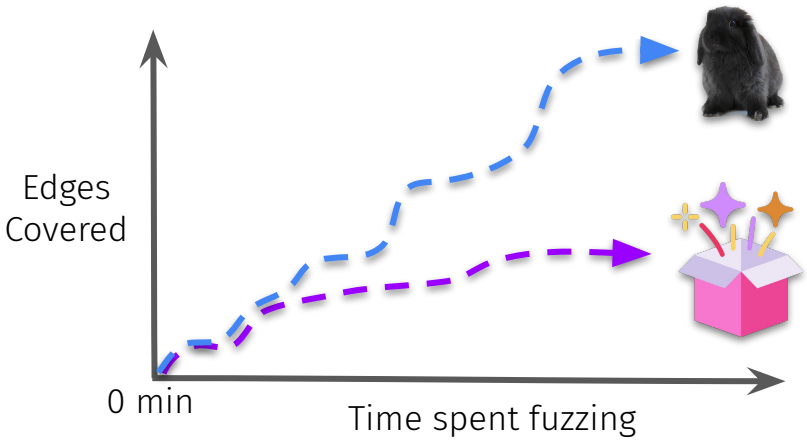
White-box Generation Trade-offs

- **All of these techniques are heavyweight**
  - Too slow to deploy for every input, branch, etc.
  - Must decide *which* problems to feed it
    - **Scheduling** problem
- **Generally limited to simple software**
  - Good luck doing taint tracking on MS Office…



White-box Generation Trade-offs

- **All of these techniques are heavyweight**
  - Too slow to deploy for every input, branch, etc.
  - Must decide *which* problems to feed it
    - **Scheduling** problem
- **Generally limited to simple software**
  - Good luck doing taint tracking on MS Office…
- **Emerging techniques give us hope!**
  - Fast “poor man’s” taint tracking: RedQueen
  - Fast source-level concolic exec: SymCC



Recap: Types of Input Generation

- **Model-agnostic:** great on simple, easy-to-solve branches
  - Need a lot of luck to solve **multi-byte conditionals** and **checksums**
- **Model-guided:** more valid inputs leads to higher coverage
  - Out of luck if specification is **not defined** or **hard-to-define**
- **White-box approaches:**
  - **Symbolic / concolic exec:** precise solving of multi-byte conditionals
  - **Taint tracking:** easily identifies key data objects, branch constraints
  - Far too **heavyweight** to deploy on *every single* generated input

Source: The Art, Science, and Engineering of Fuzzing: A Survey

Questions?