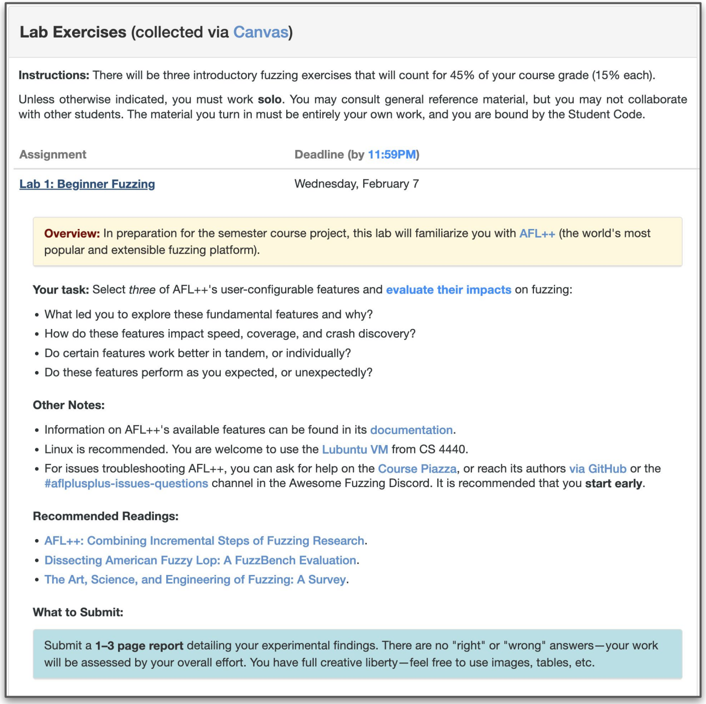
Week 3: Lecture B

Introduction to Fuzzing

Wednesday, January 22, 2025

Recap: Lab 1: Beginner Fuzzing

- See **Assignments** tab on course website
  - Click the drop-down link for Lab 1
- Deadline: **Wednesday, January 29**
  - Submit on **Canvas** by **11:59 PM MST**
  - Late assignments **are not accepted**



Lab Exercises (collected via Canvas)

Instructions: There will be three introductory fuzzing exercises that will count for 45% of your course grade (15% each).

Unless otherwise indicated, you must work **solo**. You may consult general reference material, but you may not collaborate with other students. The material you turn in must be entirely your own work, and you are bound by the Student Code.

Assignment	Deadline (by 11:59PM)
Lab 1: Beginner Fuzzing	Wednesday, February 7

Overview: In preparation for the semester course project, this lab will familiarize you with AFL++ (the world's most popular and extensible fuzzing platform).

Your task: Select *three* of AFL++'s user-configurable features and **evaluate their impacts** on fuzzing:

- What led you to explore these fundamental features and why?
- How do these features impact speed, coverage, and crash discovery?
- Do certain features work better in tandem, or individually?
- Do these features perform as you expected, or unexpectedly?

Other Notes:

- Information on AFL++'s available features can be found in its documentation.
- Linux is recommended. You are welcome to use the Lubuntu VM from CS 4440.
- For issues troubleshooting AFL++, you can ask for help on the Course Piazza, or reach its authors via GitHub or the #aflplusplus-issues-questions channel in the Awesome Fuzzing Discord. It is recommended that you **start early**.

Recommended Readings:

- AFL++: Combining Incremental Steps of Fuzzing Research.
- Dissecting American Fuzzy Lop: A FuzzBench Evaluation.
- The Art, Science, and Engineering of Fuzzing: A Survey.

What to Submit:

Submit a **1–3 page report** detailing your experimental findings. There are no "right" or "wrong" answers—your work will be assessed by your overall effort. You have full creative liberty—feel free to use images, tables, etc.

Recap: Lab 1: Beginner Fuzzing

- **Assignment:** familiarize yourself with AFL++
  - Read its documentation in **docs/**

- **Pick three features, try them out, 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 Lubuntu VM** if you don't have one

Recap: Lab 1: Beginner Fuzzing

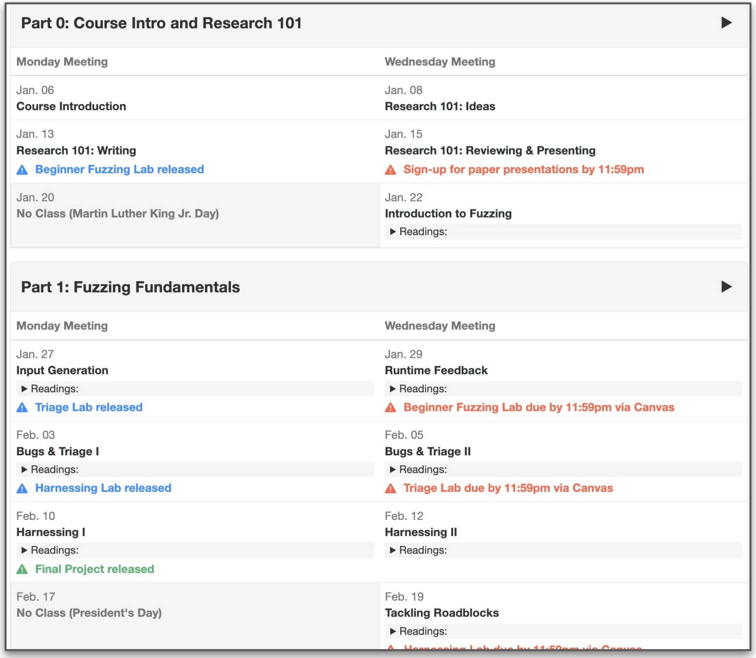
- Primary goal: **prepare you for the semester project**

- Other goals:
  - Give you experience with industry-standard tools
  - Put you in the "research" mindset
  - Improve your debugging skills

Recap: Key Dates

Jan. 15	Select one paper to present
Jan. 20	No class (MLK Jr. Day)
Jan. 29	Lab 1 due
Feb. 05	Lab 2 due
Feb. 17	No class (President's Day)
Feb. 19	Lab 3 due
Feb. 26	**5-minute project pitches**
Mar. 10 & 12	No class (Spring Break)
Apr. 16 & 21	**Final project presentations**

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



Part 0: Course Intro and Research 101 ▶

Monday Meeting	Wednesday Meeting
Jan. 06 **Course Introduction**	Jan. 08 **Research 101: Ideas**
Jan. 13 **Research 101: Writing** ⚠ Beginner Fuzzing Lab released	Jan. 15 **Research 101: Reviewing & Presenting** ⚠ Sign-up for paper presentations by 11:59pm
Jan. 20 **No Class (Martin Luther King Jr. Day)**	Jan. 22 **Introduction to Fuzzing** ▶ Readings:

Part 1: Fuzzing Fundamentals ▶

Monday Meeting	Wednesday Meeting
Jan. 27 **Input Generation** ▶ Readings: ⚠ Triage Lab released	Jan. 29 **Runtime Feedback** ▶ Readings: ⚠ Beginner Fuzzing Lab due by 11:59pm via Canvas
Feb. 03 **Bugs & Triage I** ▶ Readings: ⚠ Harnessing Lab released	Feb. 05 **Bugs & Triage II** ▶ Readings: ⚠ Triage Lab due by 11:59pm via Canvas
Feb. 10 **Harnessing I** ▶ Readings: ⚠ Final Project released	Feb. 12 **Harnessing II** ▶ Readings:
Feb. 17 **No Class (President's Day)**	Feb. 19 **Tackling Roadblocks** ▶ Readings: ⚠ Harnessing Lab due by 11:59pm via Canvas

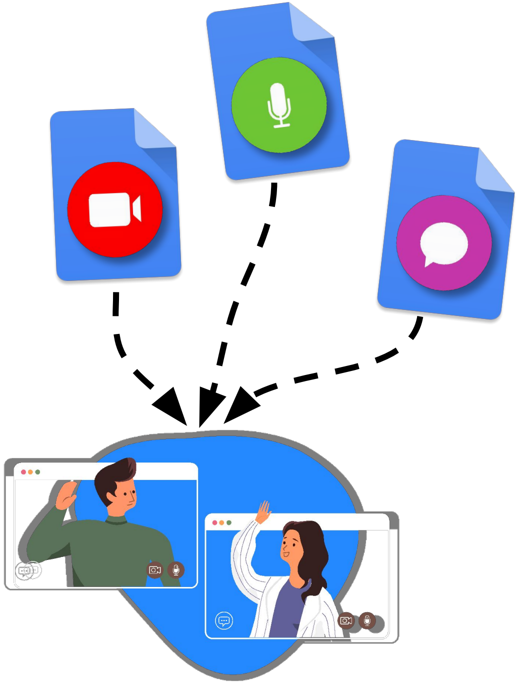
Questions?

Background

Programs and Inputs

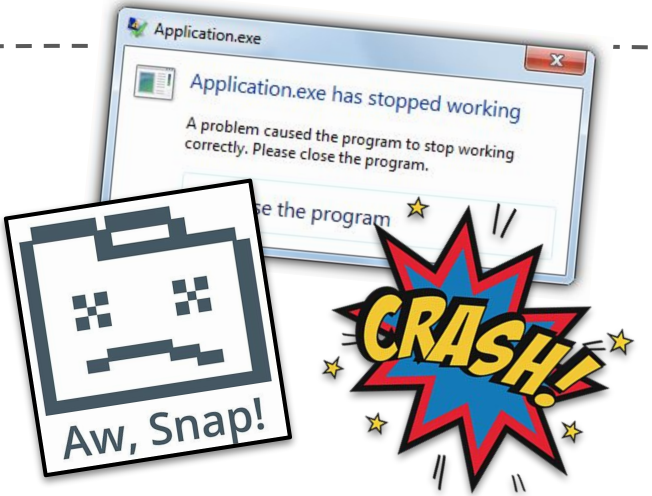
- Modern applications accept many sources of input:
  - **Files**
  - **Arguments**
  - **Environment variables**
  - **Network packets**
  - …

- Nowadays: multiple sources of inputs



Software Bugs

Software Bugs



Application.exe

Application.exe has stopped working

A problem caused the program to stop working correctly. Please close the program.

When bugs go bad

- Improper input validation leads to **security vulnerabilities**
  - Bugs that violate the system’s confidentiality, integrity, or availability

- **Exploitation**: leveraging a vulnerability to perform unauthorized actions

Exploitation

Common Vulnerabilities

- Missed initialization check
- Free'd pointers not NULL'd
- Unchecked memory writes

Consequences

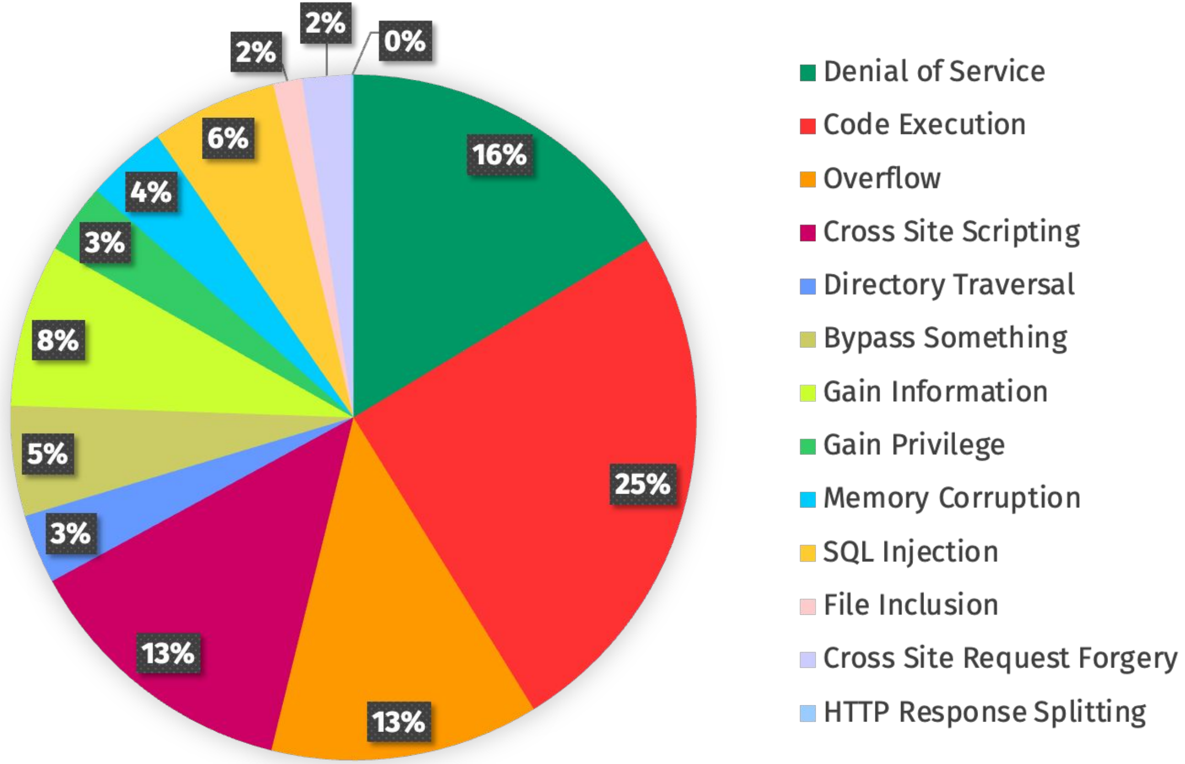
- Use uninitialized memory
- Use non-owned memory
- Overflowing a data buffer

Attacker Exploitation

- Software denial of service
- Leak sensitive information
- Inject & run arbitrary code

Race against time to **find & fix** vulnerabilities
before they are **exploited**

With so many vulnerabilities today…



Source: cvedetails.com

… exploits are getting more and more sophisticated

1972
First known overflows

1996
Stack overflows

1997
Ret-2-Libc attacks

1997
Function ptr hijacking

1998
Heap overflows

1998
StackGuard bypasses

1999
Format strings

2002
Integer overflows

2002
ASLR bypasses

2005
Hardware DEP bypasses

2005
Ret oriented programming

2007
Heap grooming

2007
Null pointer dereference

2007
Double frees

2009
Heap spraying

2010
JIT spraying

2011
Jmp oriented programming

2014
Call oriented programming

2016
Data oriented programming

2021
Zero-click exploits

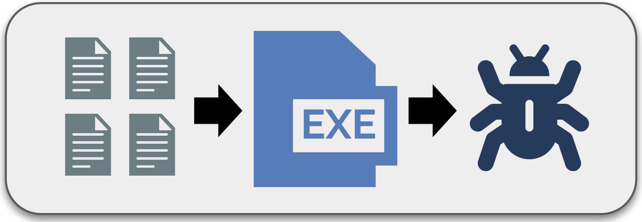
What's next?

Proactive Vulnerability Discovery

Static Analysis:

- Analyze program **without running it**
- Accuracy a major concern
  - **False negatives** (vulnerabilities missed)
  - **False positives** (results are unusable)
- As code size grows, **speed drops**

Dynamic Testing:



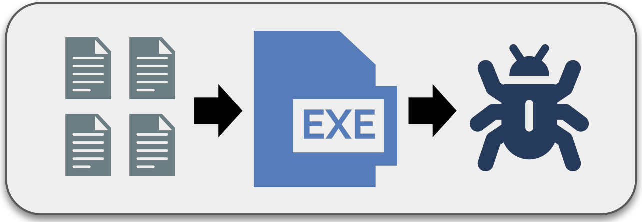
- Analyze program **by executing it**
- Better accuracy: **no false positives**
  - Execution reveals only what exists
  - Program crashed? You found a bug!
- Capable of very **high throughput**

Proactive Vulnerability Discovery

Static Analysis:

- Analyze program **without running it**
- Accuracy a major concern
  - **False negatives** (vulnerabilities missed)
  - **False positives** (results are unusable)
- As code size grows, **speed drops**

Dynamic Testing:

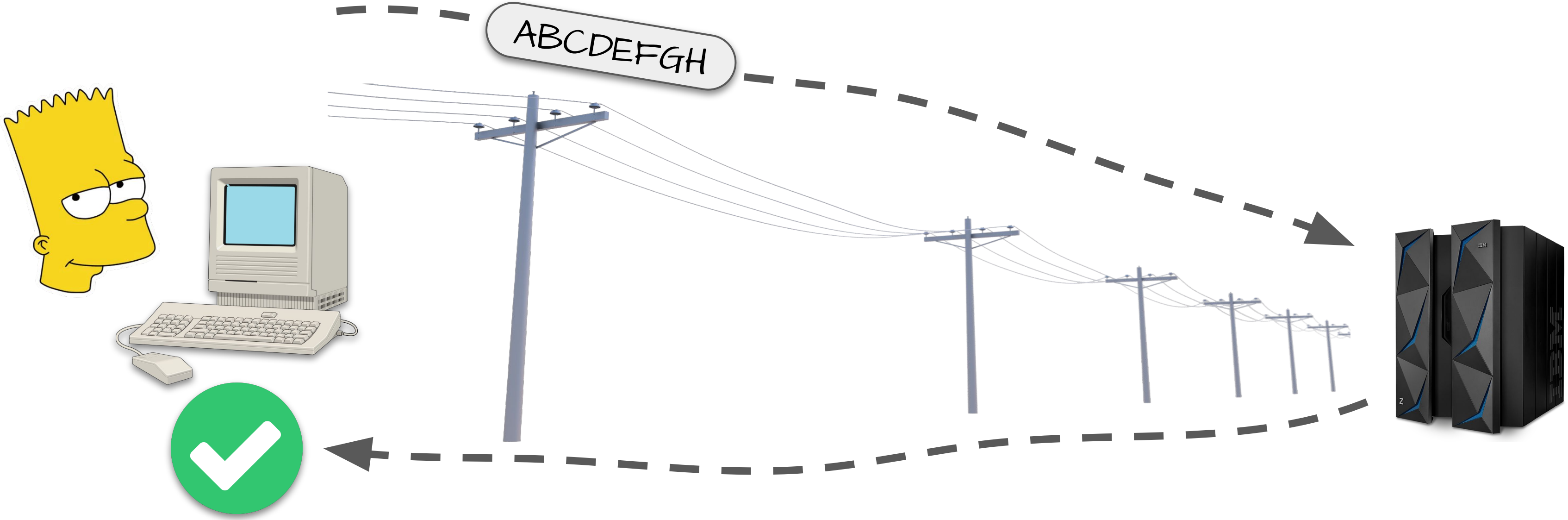


- Analyze program **by executing it**
- Better accuracy: **no false positives**
  - Execution reveals only what exists
  - Program crashed? You found a bug!
- Capable of very **high throughput**

Fuzzing

One dark and stormy night…

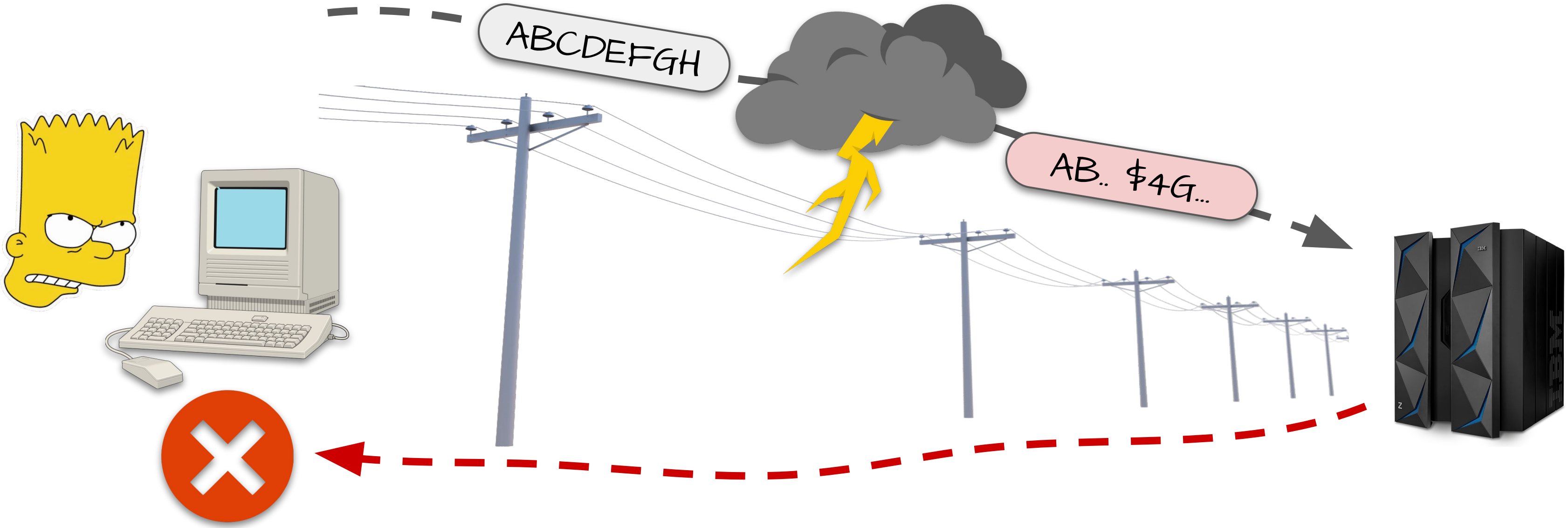
in the era of dial-up internet



Source: https://www.linux-magazine.com/Issues/2022/255/Fuzz-Testing

One dark and stormy night…

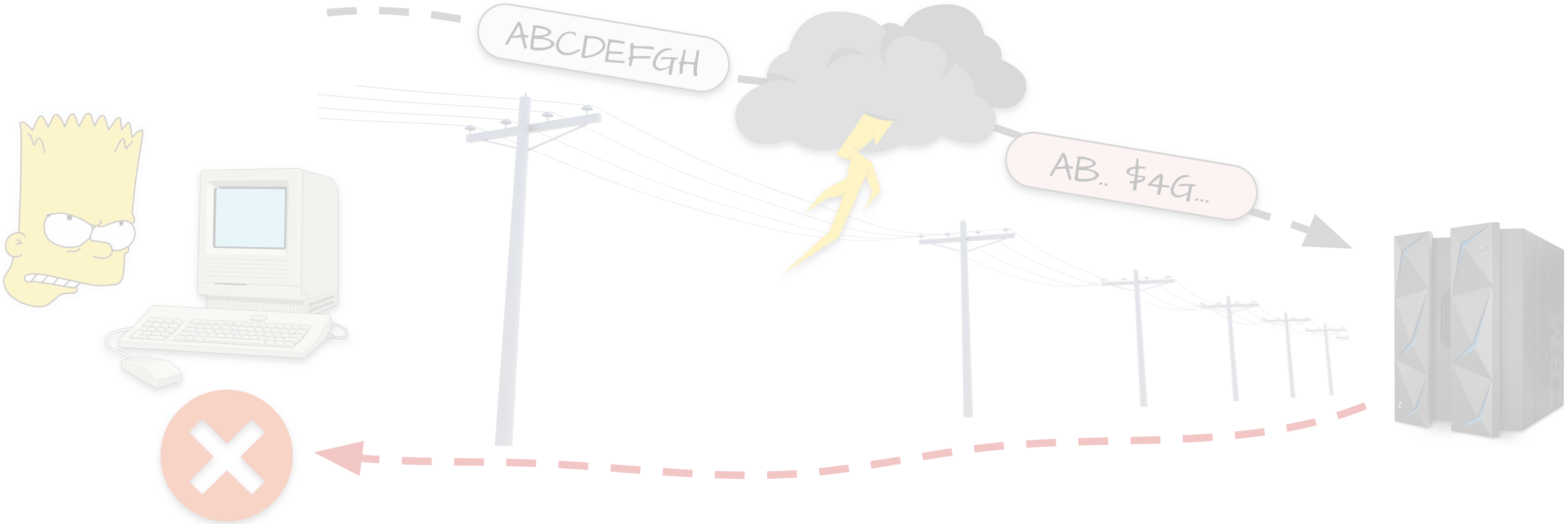
in the era of dial-up internet



Source: https://www.linux-magazine.com/Issues/2022/255/Fuzz-Testing

One dark and stormy night…

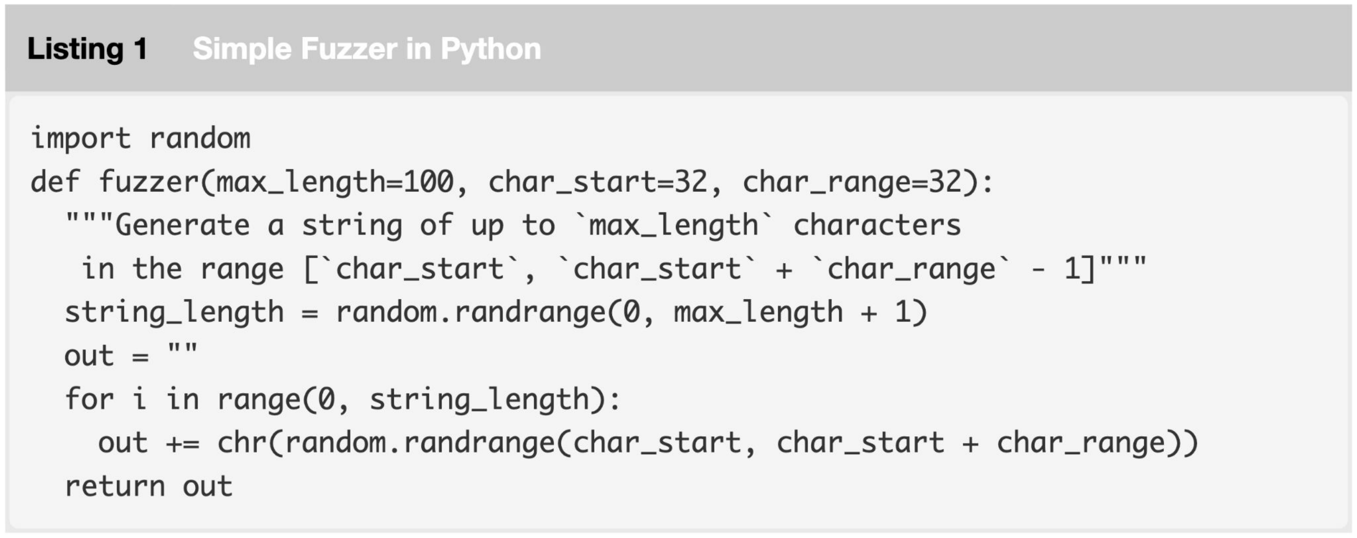
in the era of dial-up internet



- Shouldn’t programs do much better with **glitched or invalid input**?

Source: https://www.linux-magazine.com/Issues/2022/255/Fuzz-Testing

Bart's idea: test programs on *random* inputs!



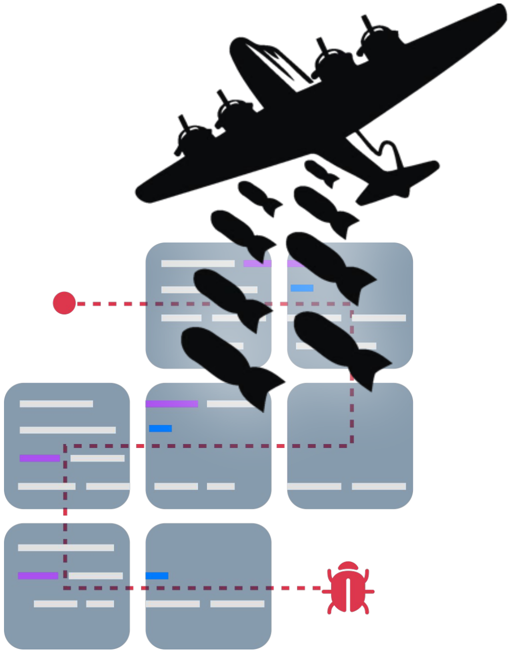
Listing 1 Simple Fuzzer in Python

```
import random
def fuzzer(max_length=100, char_start=32, char_range=32):
  """Generate a string of up to `max_length` characters
   in the range [`char_start`, `char_start` + `char_range` - 1]"""
  string_length = random.randrange(0, max_length + 1)
  out = ""
  for i in range(0, string_length):
    out += chr(random.randrange(char_start, char_start + char_range))
  return out
```

```
!7#%"*#0=)$;%6*;>638:*>80"=</>(/*
:-(2<4 !:5*6856&?""11<7+%<%7,4.8+
```

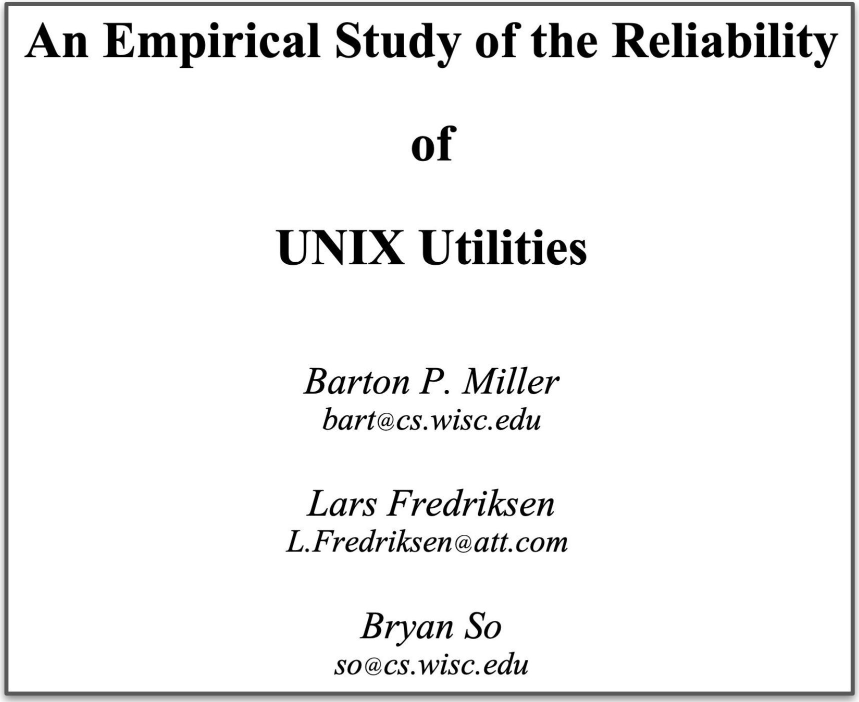
Bart's idea: test programs on *random* inputs!

- Quickly generate lots and lots of **random inputs**
- Execute each on the target program
- **See what happens**
  - Crash
  - Hang
  - Nothing at all



Random inputs work!

- Crash or hang **25–33%** of utility programs in **seven** UNIX variants
- Results reveal several common mistakes made by programmers
- They called this *fuzz* testing
  - Known today as **fuzzing**



An Empirical Study of the Reliability

of

UNIX Utilities

Barton P. Miller
bart@cs.wisc.edu

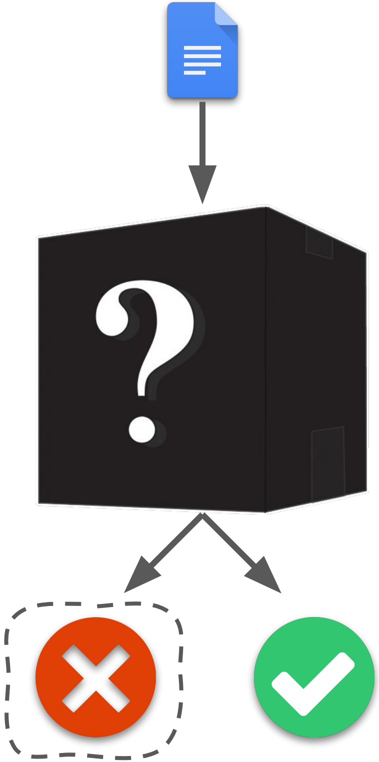
Lars Fredriksen
L.Fredriksen@att.com

Bryan So
so@cs.wisc.edu

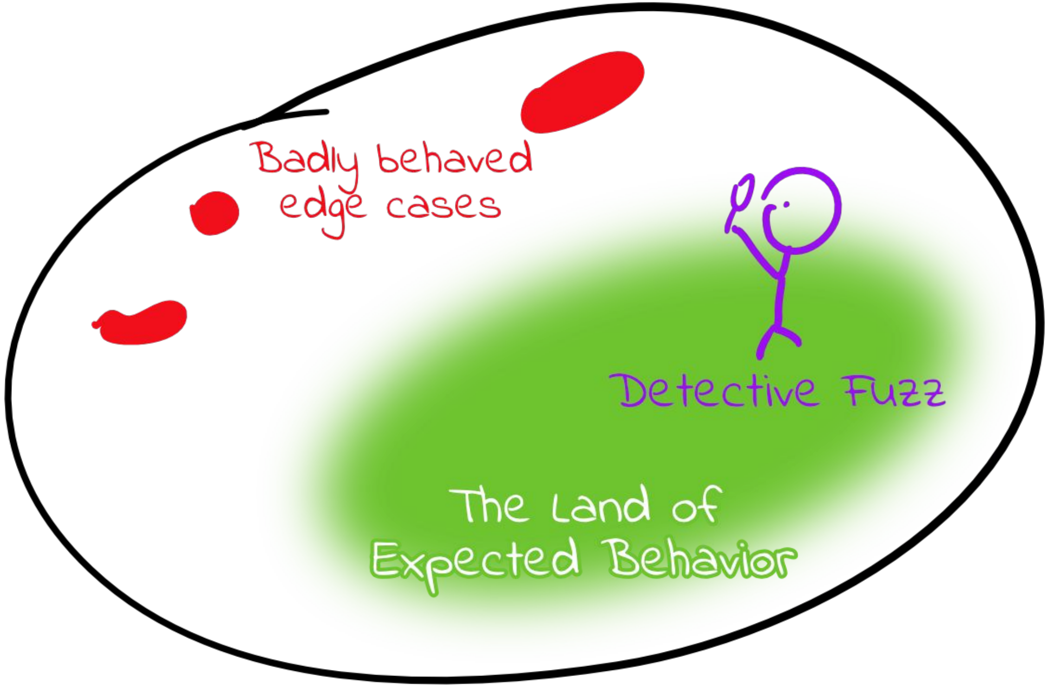
The Evolution of Fuzzing

Fuzzing like it's 1989

- Random inputs
- **Black-box:** only check program's **end result**
  - Signals
  - Return values
  - Program-specific output
- Save inputs that trigger **weird behavior**
  - `SIGSEGV`, `SIGFPE`, `SIGILL`, etc.
  - Assertion failures
  - Other reported errors



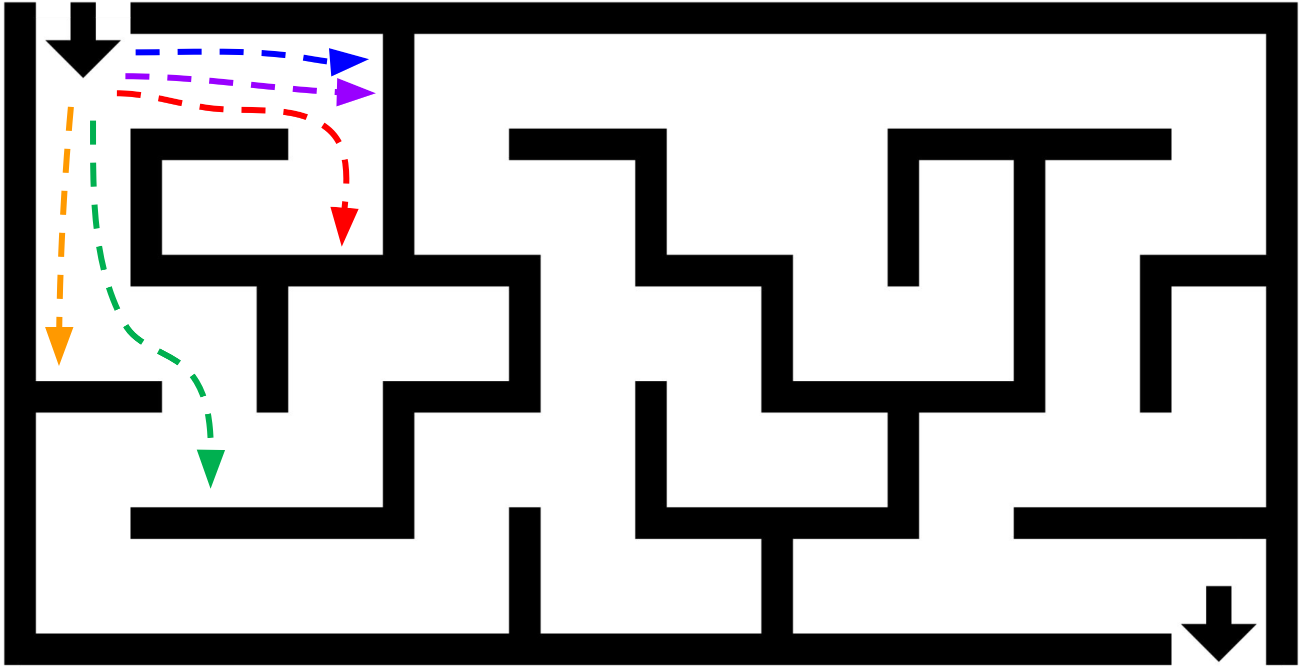
Finding Bugs with Fuzzing



The space of possible program behaviors

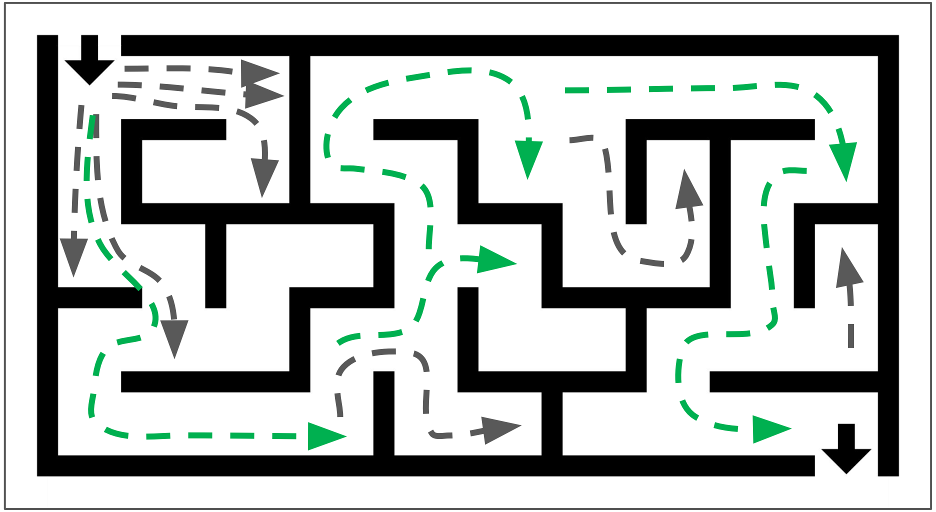
Source: https://blog.trailofbits.com/2020/10/22/lets-build-a-high-performance-fuzzer-with-gpus/

Black-box fuzzing only gets you so far...



How can fuzzing exploration be guided?

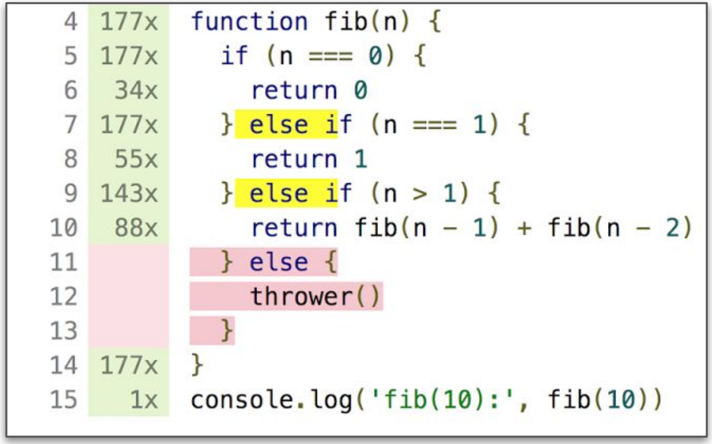
- Idea: track some measure of exploration “progress”
  - Coverage of program code
  - Stack traces
  - Memory accesses
- Pinpoint inputs that further progress over the others
- **Mutate only those inputs**



Code Coverage

- **Code coverage:** program regions reached by each test case

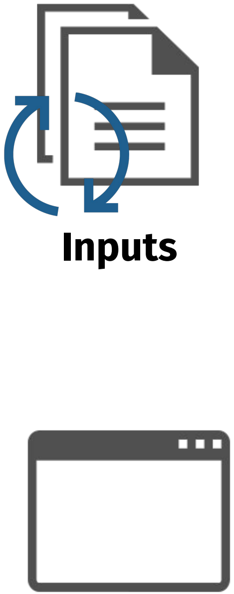
- Horse racing analogy: breed only the winning inputs
  - New coverage? **Keep the input**
  - Old coverage? **Discard it**



```
177x  function fib(n) {
177x    if (n === 0) {
 34x      return 0
177x    } else if (n === 1) {
 55x      return 1
143x    } else if (n > 1) {
 88x      return fib(n - 1) + fib(n - 2)
        } else {
          thrower()
        }
177x  }
  1x  console.log('fib(10):', fib(10))
```

Maximize code coverage

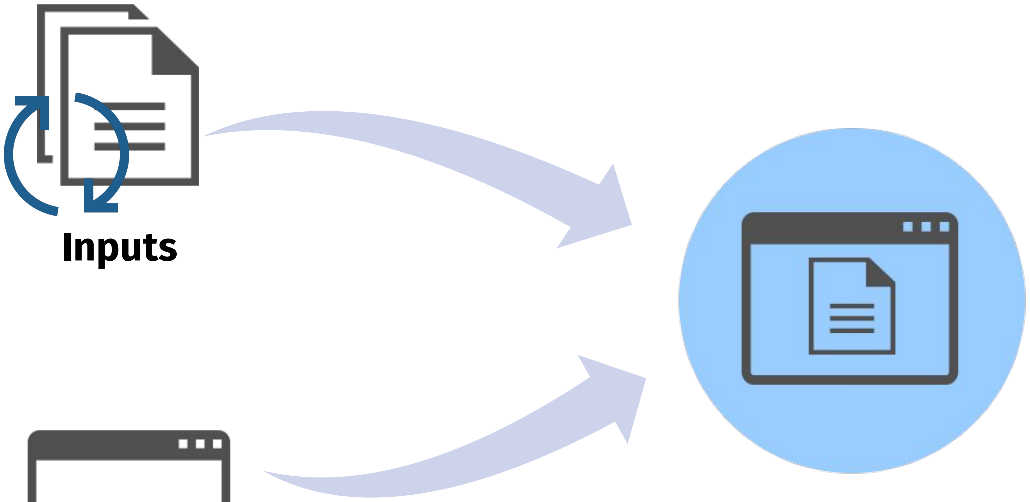
Coverage-guided Fuzzing



Inputs

Program

Coverage-guided Fuzzing



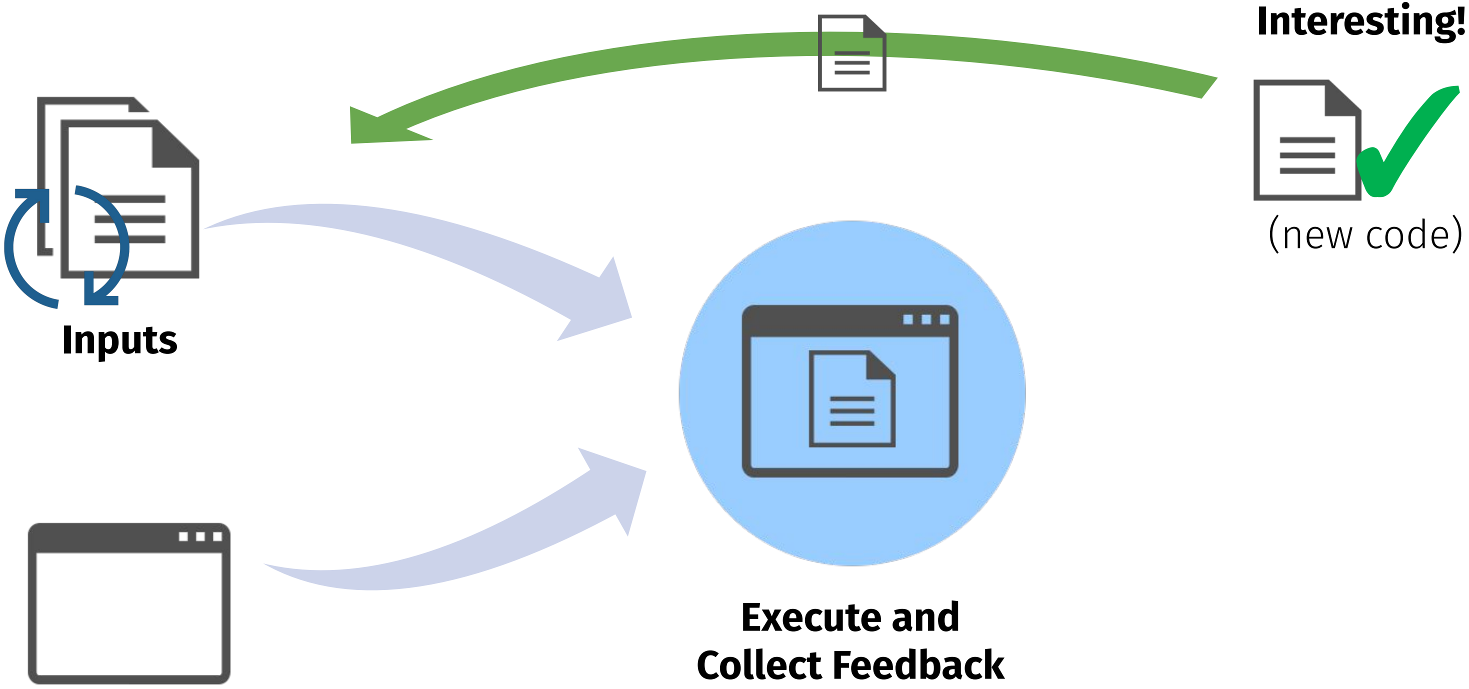
Inputs

Program

Execute and Collect Feedback

(e.g., code coverage)

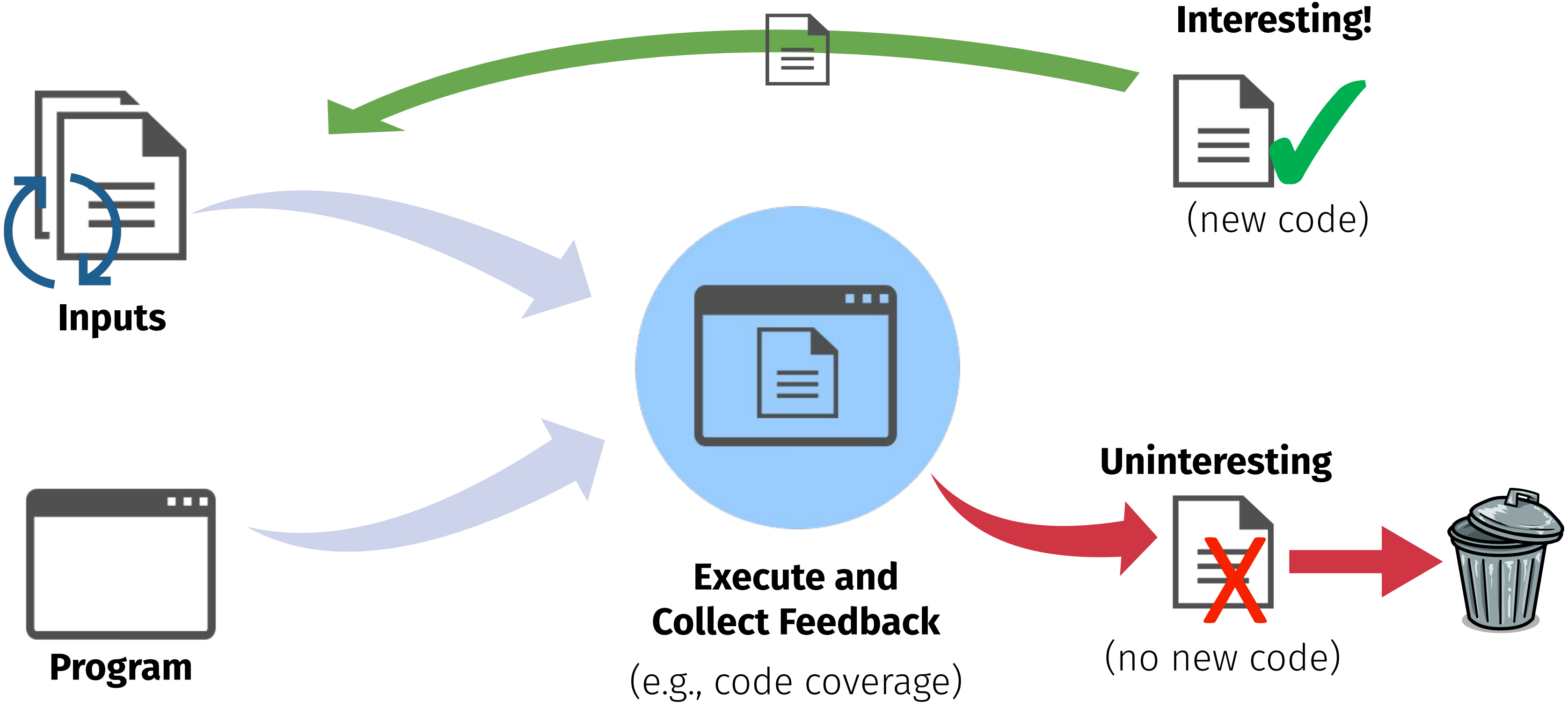
Coverage-guided Fuzzing



Execute and Collect Feedback

(e.g., code coverage)

Coverage-guided Fuzzing



Interesting!

(new code)

Inputs

Program

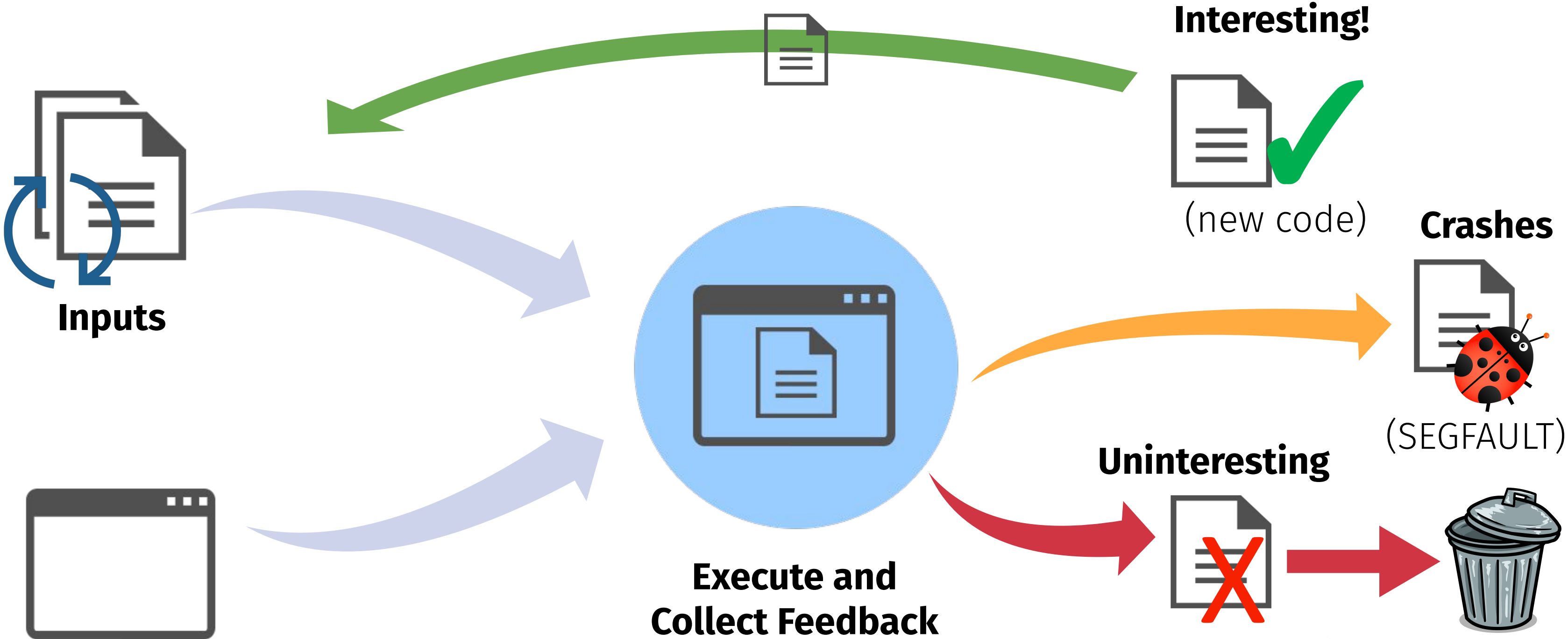
Execute and Collect Feedback

(e.g., code coverage)

Uninteresting

(no new code)

Coverage-guided Fuzzing



Interesting!

(new code)

Crashes

(SEGFAULT)

Inputs

Program

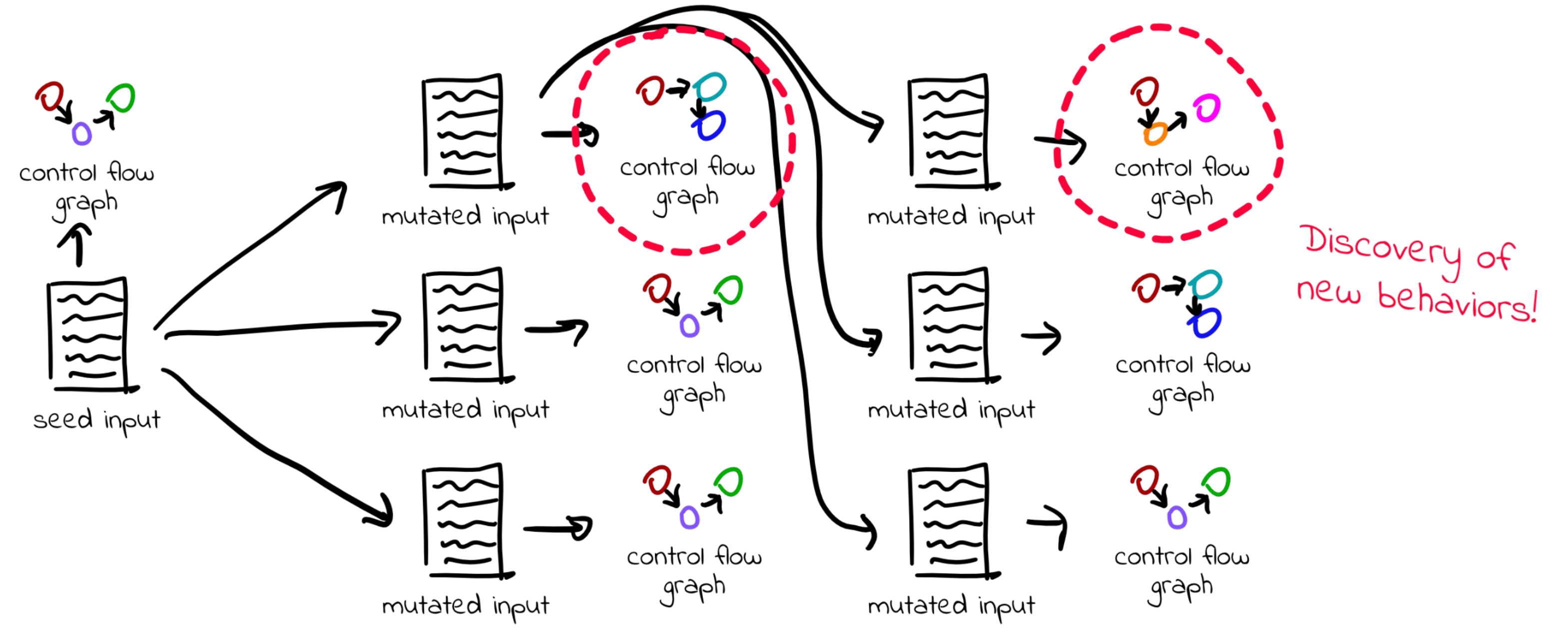
Execute and Collect Feedback

(e.g., code coverage)

Uninteresting

(no new code)

Coverage-guided Fuzzing



1) Run the seed input through the program to produce a CFG

2) Mutate the input, test the new inputs, and look for changes in the CFG

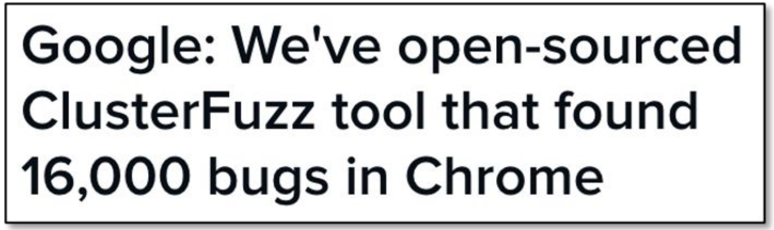
3) Rinse and repeat!

Source: https://blog.trailofbits.com/2020/10/22/lets-build-a-high-performance-fuzzer-with-gpus/

Modern Fuzzing

Fuzzing in the Industry

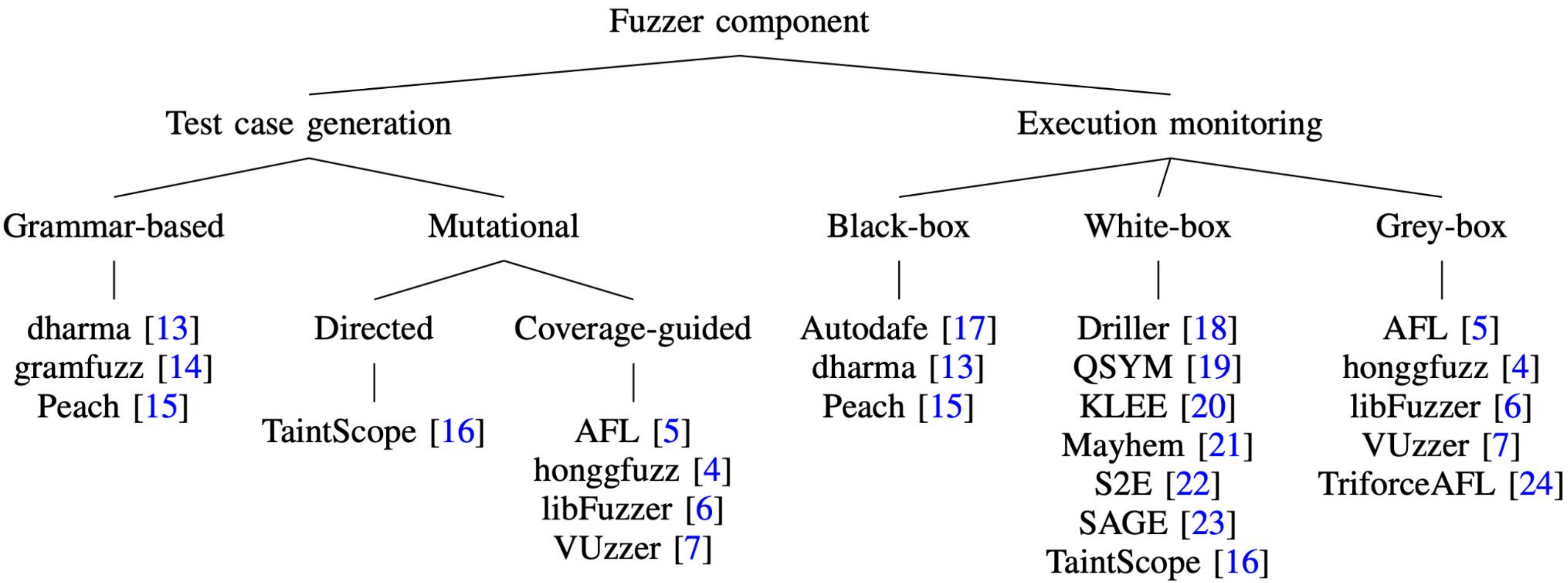
- Fuzzing = today's most popular bug-finding technique
  - Most real-world fuzzing is **coverage-guided**



Google: We've open-sourced ClusterFuzz tool that found 16,000 bugs in Chrome

New fuzzing tool finds 26 USB bugs in Linux, Windows, macOS, and FreeBSD

Taxonomy of Fuzzers

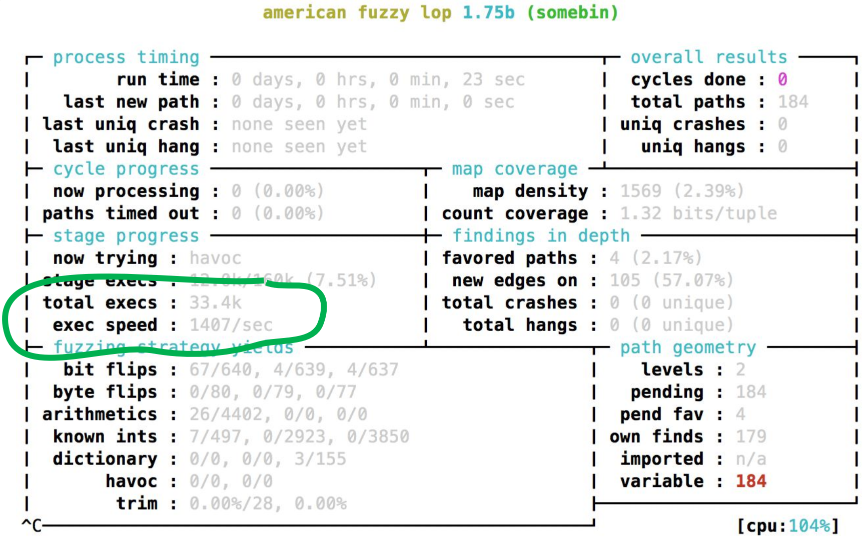


Tools of the trade: AFL

- Most historically significant fuzzer ever developed
- Authors: Michal Zalewski (2013)
  - Google (2019–2022)
  - The AFL++ team (2020–onwards)
- Versatile, easy to spin up & modify
  - Spawned probably ~100 PhD & MS theses
  - (mine included)
- **Mix of carefully chosen trade-offs**

What AFL aims to be...

- Primary goal: **high test case throughput**
- Sacrifice precision in most areas
  - Lightweight, simple mutators
  - Coarse, approximated code coverage
  - Little reasoning about seed selection
- Revolutionary & still insanely effective
  - Ideas ported over to honggFuzz, libFuzzer
  - and nearly all other fuzzers



```
                  american fuzzy lop 1.75b (somebin)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 23 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec         │  total paths : 184    │
│ last uniq crash : none seen yet                       │ uniq crashes : 0      │
│  last uniq hang : none seen yet                       │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0 (0.00%)         │    map density : 1569 (2.39%)          │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.32 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 4 (2.17%)              │
│ stage execs : 12.0k/160k (7.51%)    │  new edges on : 105 (57.07%)           │
│ total execs : 33.4k                 │ total crashes : 0 (0 unique)           │
│  exec speed : 1407/sec              │   total hangs : 0 (0 unique)           │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 67/640, 4/639, 4/637                  │    levels : 2          │
│  byte flips : 0/80, 0/79, 0/77                      │   pending : 184        │
│ arithmetics : 26/4402, 0/0, 0/0                     │  pend fav : 4          │
│  known ints : 7/497, 0/2923, 0/3850                 │ own finds : 179        │
│  dictionary : 0/0, 0/0, 3/155                       │  imported : n/a        │
│       havoc : 0/0, 0/0                              │  variable : 184        │
│        trim : 0.00%/28, 0.00%                       ├────────────────────────┘
^C────────────────────────────────────────────────────┘             [cpu:104%]
```

Tools of the trade: ~~AFL~~ AFL++

- **By far today's most popular fuzzer**

- Official successor to vanilla AFL
    - Started out as a community-led fork
    - Google has since archived vanilla AFL

- **A platform for trying-out new features**
    - Integrated lots of academic prototypes
    - Easily tailorable to your target's needs

https://github.com/AFLplusplus/AFLplusplus

Trade-offs are target-dependent…

Building a good fuzzer is all about finding the right balance of **performance & precision**.

Any fuzzing is better than not fuzzing!

If something has not been fuzzed before,
any fuzzing will probably find *lots* of bugs.

Questions?