Lec 25: Fuzzing

CSED415: Computer Security
Spring 2025

Seulbae Kim

POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

- Lab 05 is due by the end of Friday, May 23
  - Attend office hours for help!
    - TA: Mondays and Thursdays 7-8 PM
    - Prof: Thursdays 1-2 PM

Administrivia

- Project presentations – Next week
  - Each team: 15-minute presentation + 5-minute Q&A (20 minutes total)
    - Three teams will present on Tue, May 27
    - The other three teams will present on Thu, May 29
  - Presentation must include a demonstration (live or recorded)
  - All teams MUST submit their slides, code and/or binary, and report by **May 26**
    - Check PLMS assignment for details

Administrivia

- Final exam:
  - Time: Thursday, June 5, 2:00-3:15 PM (75 minutes)
  - Location: Classroom (Science Building II, Room #106)
  - Format: Closed book, closed notes, no electronic devices allowed
    - Allowed: One-page (US letter- or A4-sized) double-sided handwritten cheat sheet
  - Structure: 6 main questions (each may have sub-questions)
  - Scope: Lectures 15-26, Labs 03-05

Program Analysis for Bug Finding

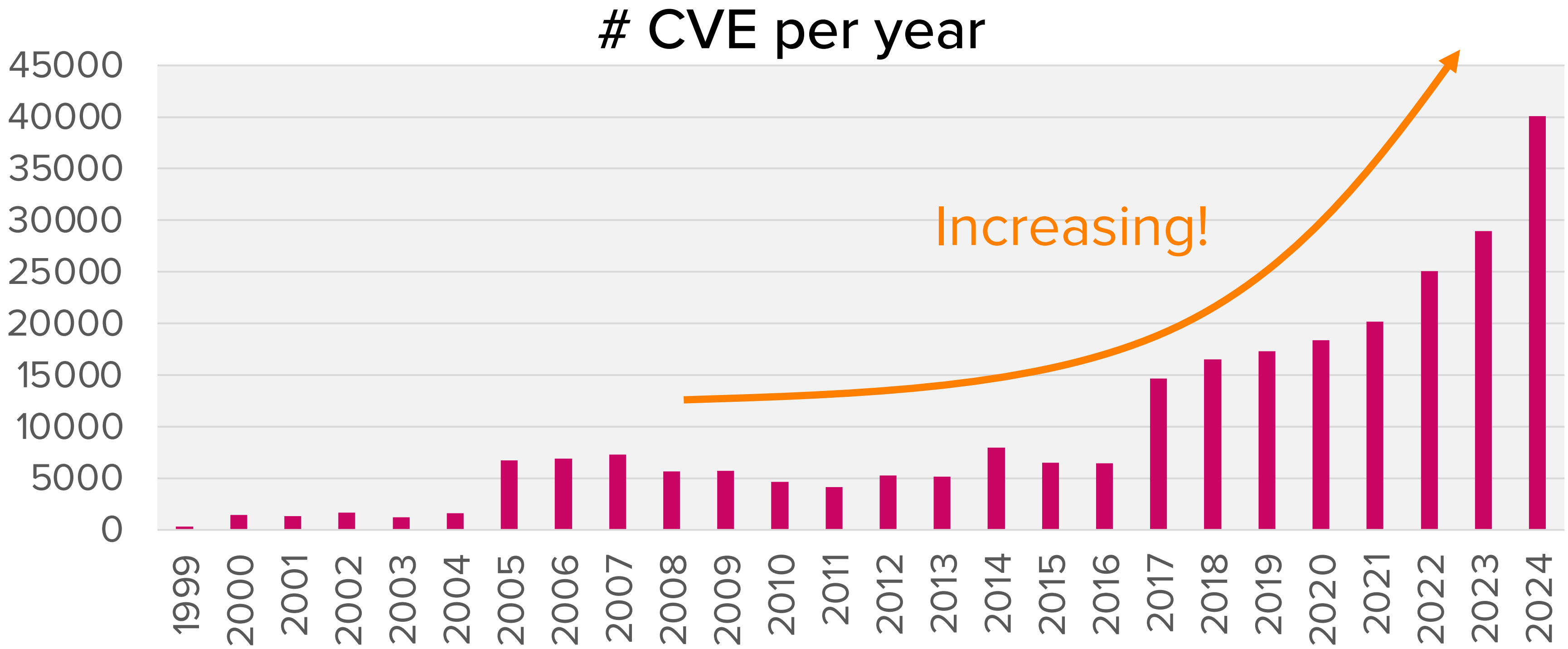
Motivation

- There are many bugs in the wild
  - Some bugs are security vulnerabilities that are exploitable by attackers

If we eliminate bugs, we can prevent attacks

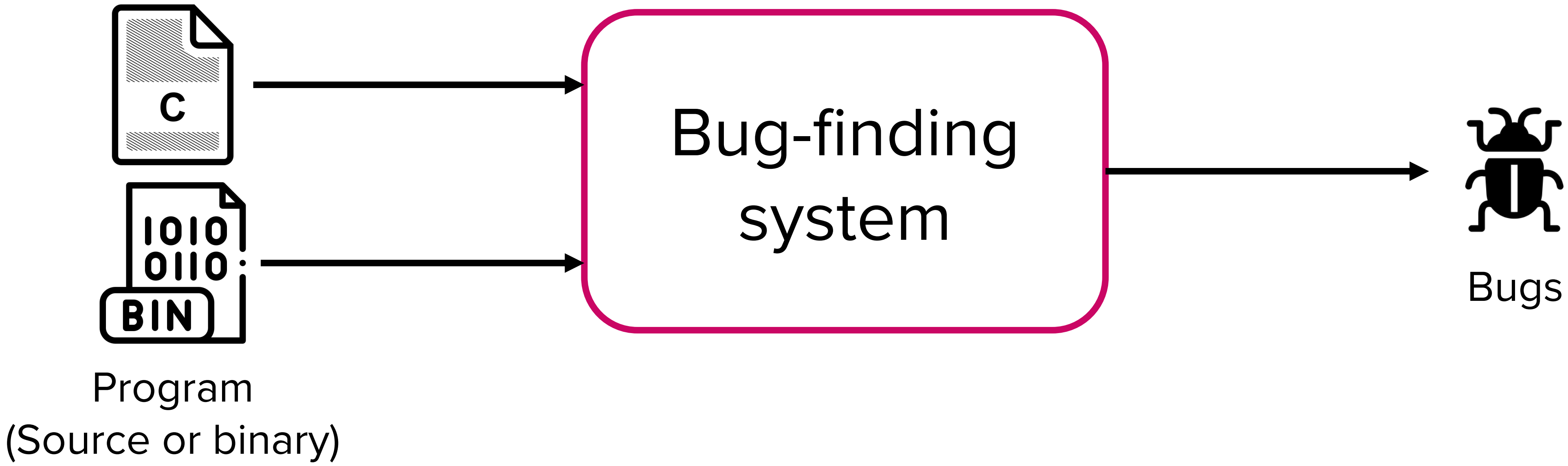
Motivation

- CVE (Common Vulnerability Enumeration)
  - CVE is an indentifier assigned to publicly disclosed vulnerabilities
  - # vulnerabilities keeps increasing - The attack surface is growing!



Key question

- Can we build a system that automatically finds bugs?



Informal proof

- Define a function `is_buggy`
  - Input: A program
  - Output: `1` if the program has at least one bug, `0` if not

```
def is_buggy(prog):
    # test prog and return 1 or 0
```

Informal proof

- Write a program `buggy_prog`

```
# buggy_prog.py

if is_buggy("buggy_prog.py"):
    return # do nothing

else:
    corrupt_memory()
    launch_root_shell()
    return
```

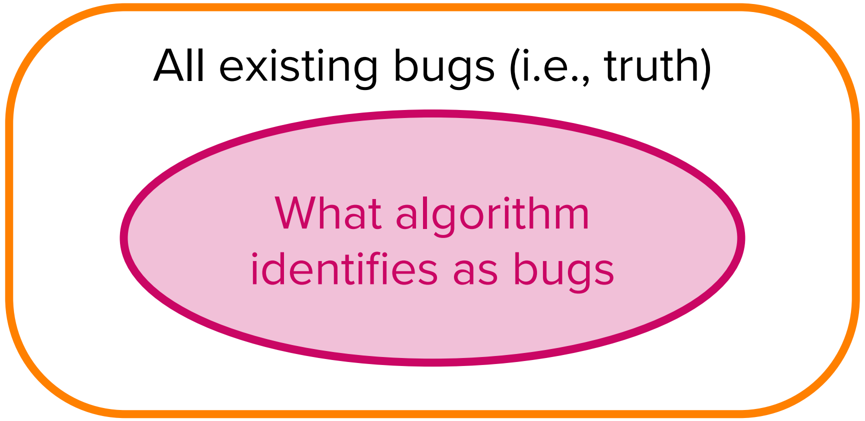
Self-contradictory! (Similar to the case of anti-virus)

Back to the question..

- Can we build a system that automatically finds bugs?
  - A a perfect bug-finding system cannot exist
- Therefore, we use best-effort approaches for partial bug identification
  - Bounded model checking
  - Static analysis
  - Dynamic analysis
  - etc.

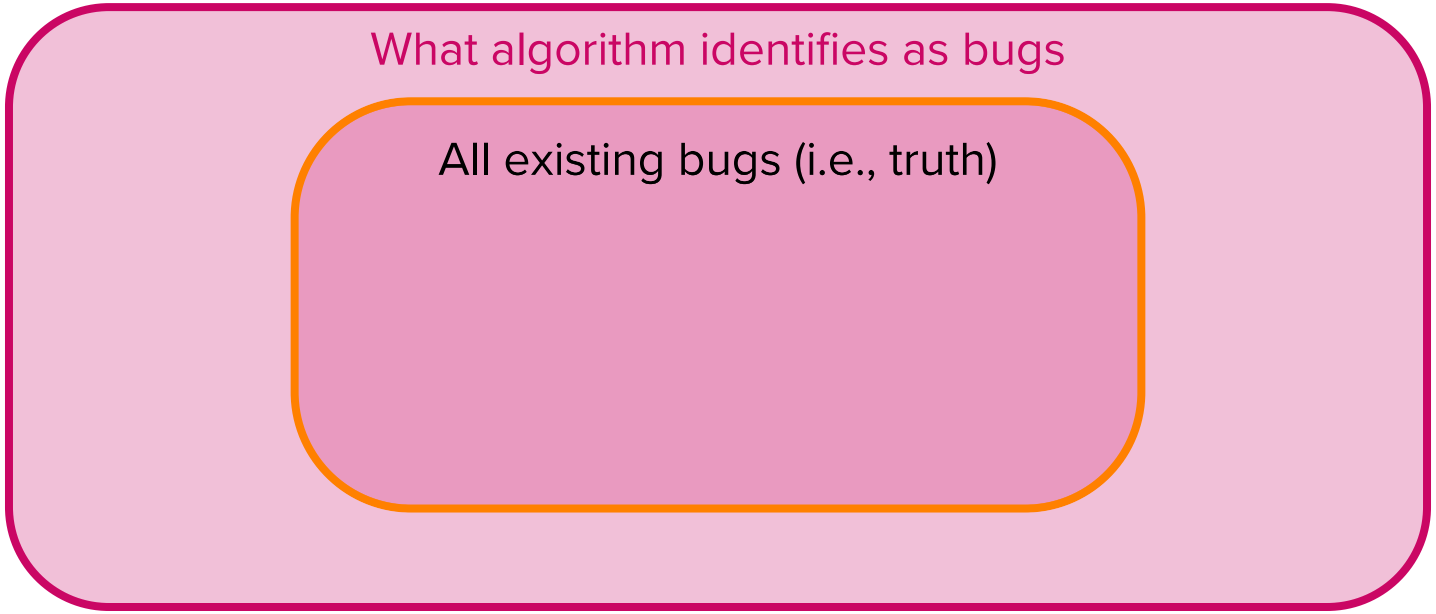
Definition of "partial"

- Soundness vs Completeness
  - An algorithm is sound if every result it produces is in fact true
    - Every reported bug is real if algorithm is sound
    - Soundness guarantees that there is no false positive
      - A sound algorithm never misclassifies a non-bug as bug



Definition of “partial”

- Soundness vs Completeness
  - An algorithm is complete if it can derive all truths
    - Every real bug is reported if algorithm is complete
    - Completeness guarantees that there is no false negative
      - A complete algorithm never misclassifies a bug as non-bug



Perfect analysis

- Soundness vs Completeness
  - Perfect algorithm is sound and complete
    - Very challenging to achieve in practice

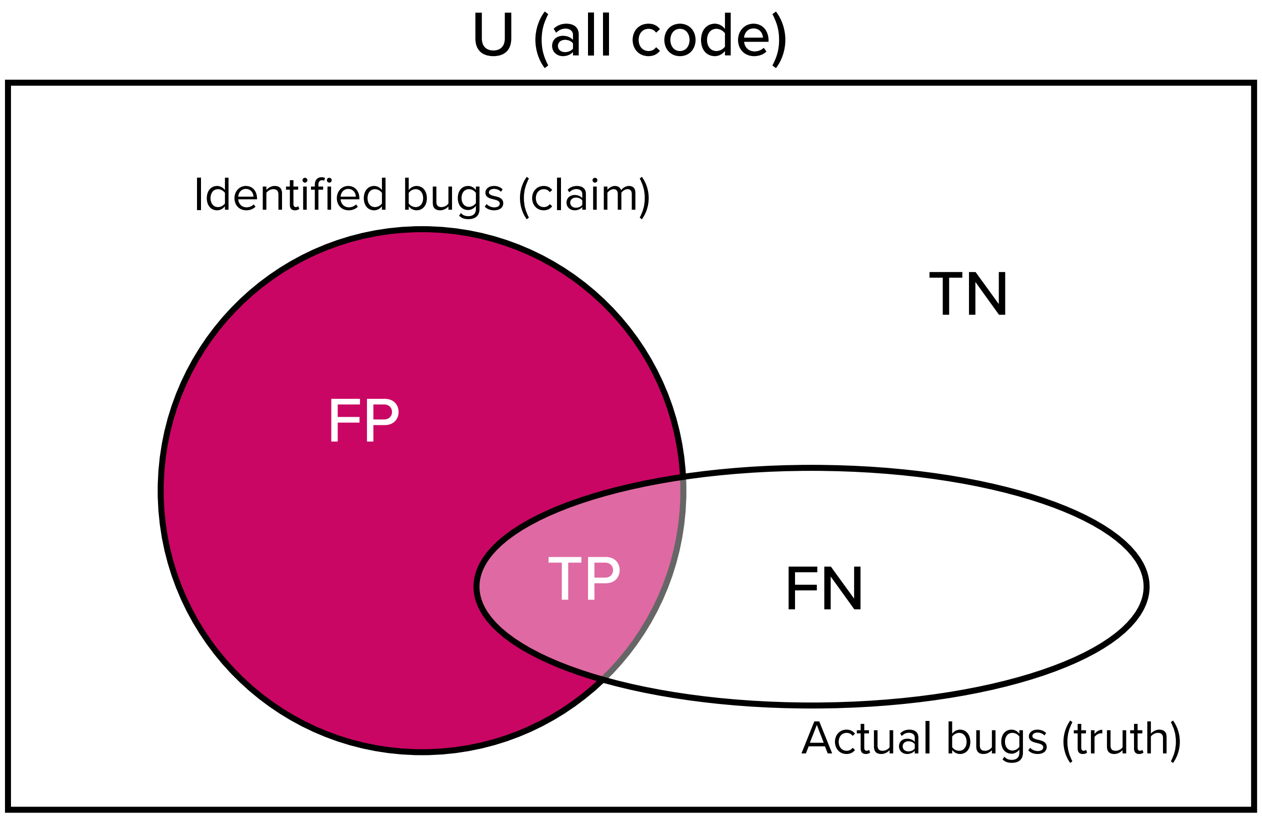
All existing bugs (i.e., truth)

=

What algorithm identifies as bugs

Metrics to evaluate a bug finding algorithm

- Precision, recall, and accuracy



- Precision: Quality of identification
  = TP / (TP + FP)
- Recall: Quantity of identification
  = TP / (FN + TP)
- Accuracy
  = (TP + TN) / U

Static vs Dynamic analysis

- Static analysis:
  - Examine program (binary or code) without running it
  - Examples:
    - Decompilation
    - Pointer analysis
    - Symbolic execution (Next topic)
- Dynamic analysis:
  - Monitor program's runtime behavior during execution
  - Examples:
    - Fuzzing (Today's topic)
    - Concolic execution

Fuzzing

Fuzzing (or fuzz-testing)

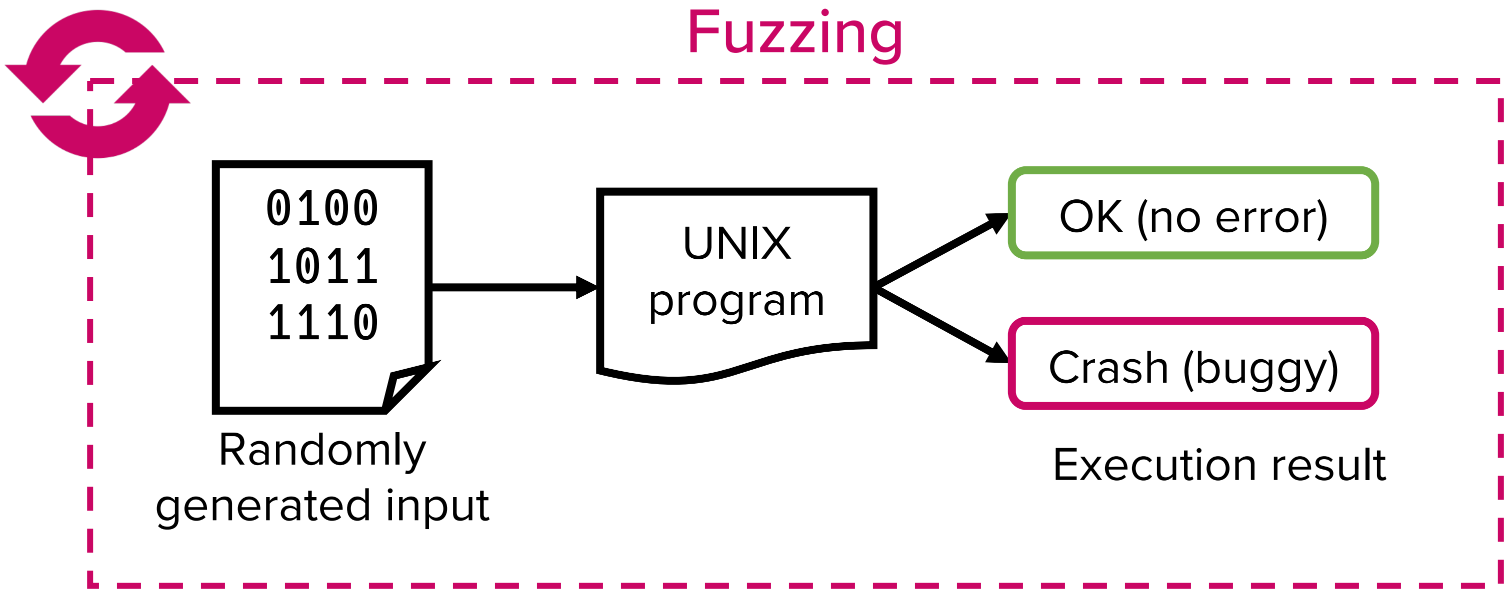
- Definition
  - Automated testing technique that feeds invalid/unexpected/random inputs to a program under test (PUT)
  - During the process, the program is monitored for anomalous behaviors
    - Crash, hang, memory leak, etc.
  - Goal is to uncover as many bugs (and vulnerabilities) as possible

Origin of fuzzing

- Experience of Barton Miller in 1990
  - He was logged on to his workstation through a modem (dial-up line)
  - Due to a storm there were a lot of line noise (i.e., line was fuzzy)
  - The noise kept generating spurious characters on the line
  - Programs on the workstation kept crashing due to the junk characters
  - He coined the term “fuzzing” from the experience

Early days of fuzzing

- Paper: Barton Miller, et al.,
  "*An Empirical Study of the Reliability of Unix Utilities*",
  Communications of the ACM, 1990



Early days of fuzzing

- Effectiveness
  - Tested 90 Unix utility programs
    - awk, cat, cc, diff, emacs, grep, …
  - The fuzzer crashed 36 utilities!
    - Due to various bugs including unbounded pointer/array accesses, overflows, race conditions, …
    - Randomly generated inputs were strikingly effective in triggering the bugs within poorly-written Unix programs of 1980s

Experiment

- Let's put Miller's fuzzer to the test with a simple program
  - Target program reads 4 bytes from stdin
  - If the four bytes are `0xde 0xad 0xbe 0xef`, it crashes by raising segmentation fault signal

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void bug(void) {
  printf("bug!\n");
  raise(SIGSEGV);
}

int main(void) {
  setvbuf(stdout, NULL, _IONBF, 0);
  setvbuf(stdin, NULL, _IONBF, 0);

  char in[16];
  FILE *fp = fopen("/dev/stdin", "rb");
  fread(&in, 4, 1, fp);
  if (in[0] == '\xde')
    if (in[1] == '\xad')
      if (in[2] == '\xbe')
        if (in[3] == '\xef')
          bug();
  fclose(fp);
  return 0;
}
```

target.c

Experiment

- Let's put Miller's fuzzer to the test with a simple program
  - Fuzzer: Brute-force 4-byte random inputs until the target crashes
  - Let's check the result at the end of today's lecture

```python
import os
import subprocess as sp

if __name__ == "__main__":
    trials = 0
    while True:
        _input = os.urandom(4)

        p = sp.Popen(["./target"], stdout=sp.PIPE, stdin=sp.PIPE, stderr=sp.PIPE)
        out, err = p.communicate(input=_input) # send _input to stdin and read stdout
        if b"bug!" in out:
            print(f"found in {trials} trials")
            print(f"Test input: {_input}")
            exit(0)

        print(trials)
        trials += 1
```

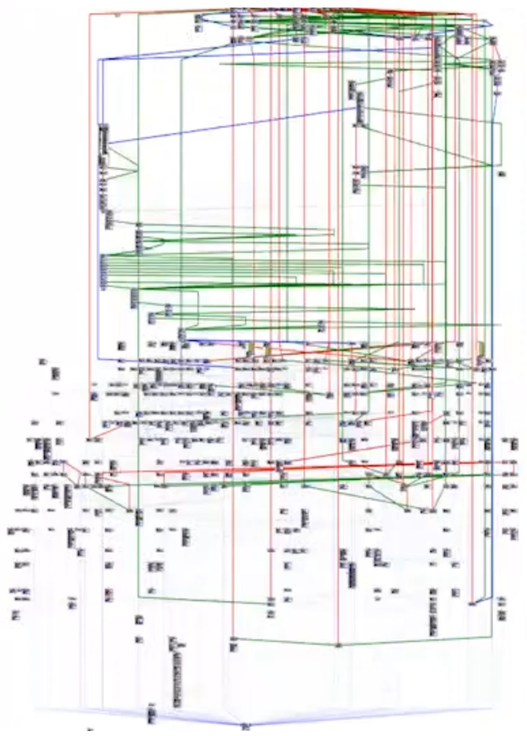
fuzz.py

Interpretation of Miller's success

- Fuzzing is simple, yet effective. Why?
  - Recall the software bugs we covered in this course
    - Many attacks originate from unsanitized user inputs
      - e.g., buffer overflow, control flow hijacking, authentication bypass, DoS, SQL injection, …
    - Fuzzing is a way to "simulate" hostile input with minimal effort

Is fuzzing still effective against modern software?

- Modern software have become very large and complex
  - Chromium browser codebase has 28 million lines of code (LoC)
  - Linux kernel comprises over 27 MLoC
  - FFmpeg has 1.4 MLoC
- Manual review of every code path is impractical
  - Imagine manually analyzing a program with the control flow graph (CFG) displayed on the right
  - Time consuming, error-prone, and hardly scalable



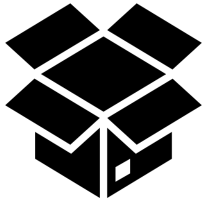
Is fuzzing applicable to large and complex programs?

Evolution of fuzzing

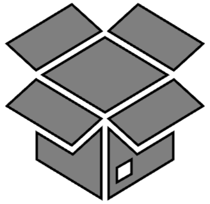
- Types of fuzzing
  - Blackbox, greybox, and whitebox fuzzing
  - Mutation-based vs generation-based fuzzing

Greybox Fuzzing

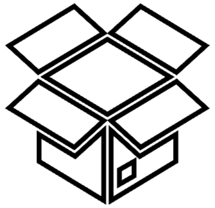
Overview of Black, grey, and whitebox fuzzing



- Generates random inputs
- Fuzzer has no knowledge of program's code and internal states
- The approach of Miller et al.
- Pros:
  - Extremely fast
  - Easy to use
  - Scalable
- Cons:
  - Poor effectiveness
  - Poor code coverage



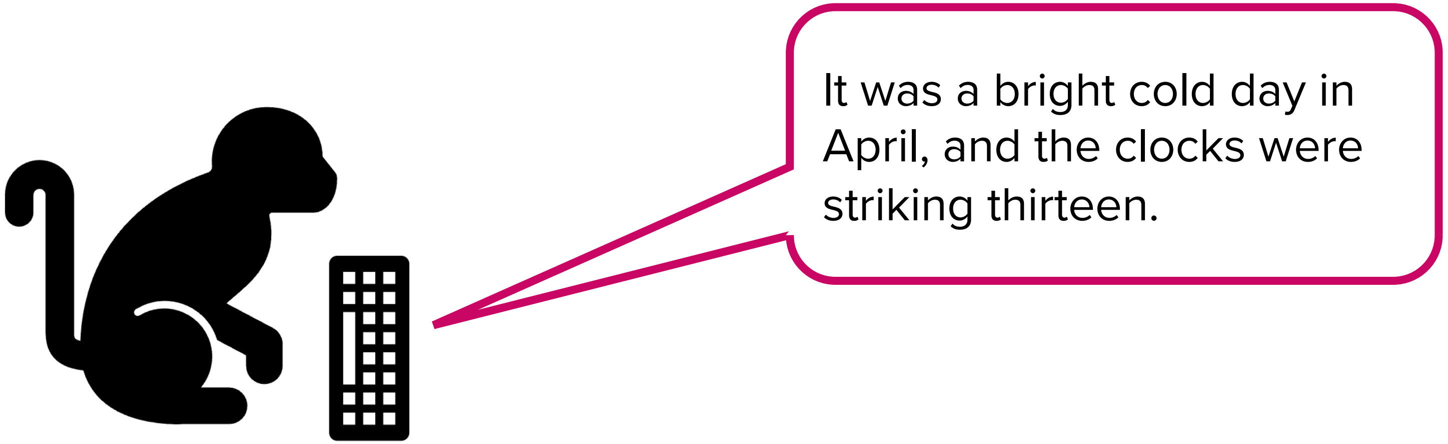
- Relies on "lightweight" instrumentation of the program under test
- Fuzzer has some knowledge of the program internals during fuzzing
  - Generates semi-random inputs based on the knowledge
- Pros: Best of both worlds
  - Scalable
  - Relatively fast
  - Decent code coverage



- Fuzzer has full knowledge of the program internals and code
- Solves path constraints to generate concrete inputs for all program branches
- Pros:
  - High code coverage
- Cons:
  - Complex
  - Slow
  - Not scalable

Breakdown of fuzzing efficiency

- A typing monkey problem
  - Given infinite amount of time, can a monkey, hitting keys at random on a keyboard, type a full sentence?

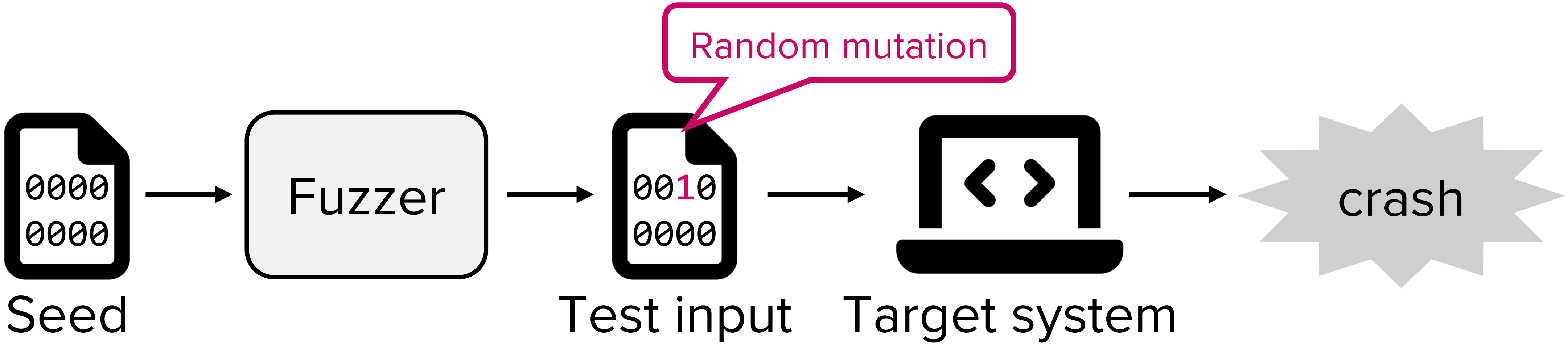


The possibility is non-zero; the monkey will "almost surely" type any given sentence

However, it will take astronomical amount of time

Breakdown of fuzzing efficiency

- Blackbox fuzzing



Target

```
x = input()

if x[0] == 'H':
  if x[1] == 'A':
    if x[2] == 'R':
      if x[3] == 'D':
        crash()
```

Seed `x = "LIFE"`

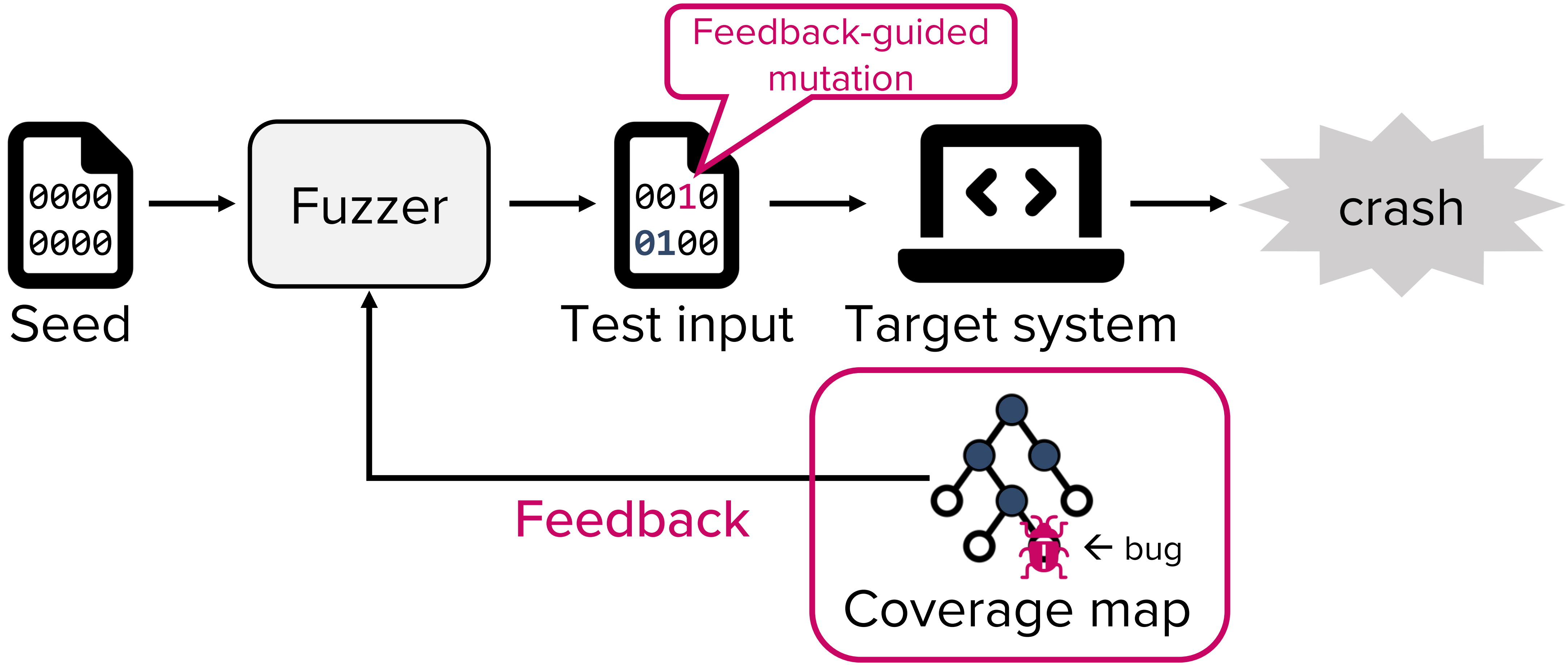
Test input

`x = "LIFO"`	`x = "5IFE"`	`x = "L0VE"`
`x = "HEFE"`	`x = "DOVE"`	`x = "LIFF"`

→ $P(\text{crash}) = \frac{1}{2^{32}}$

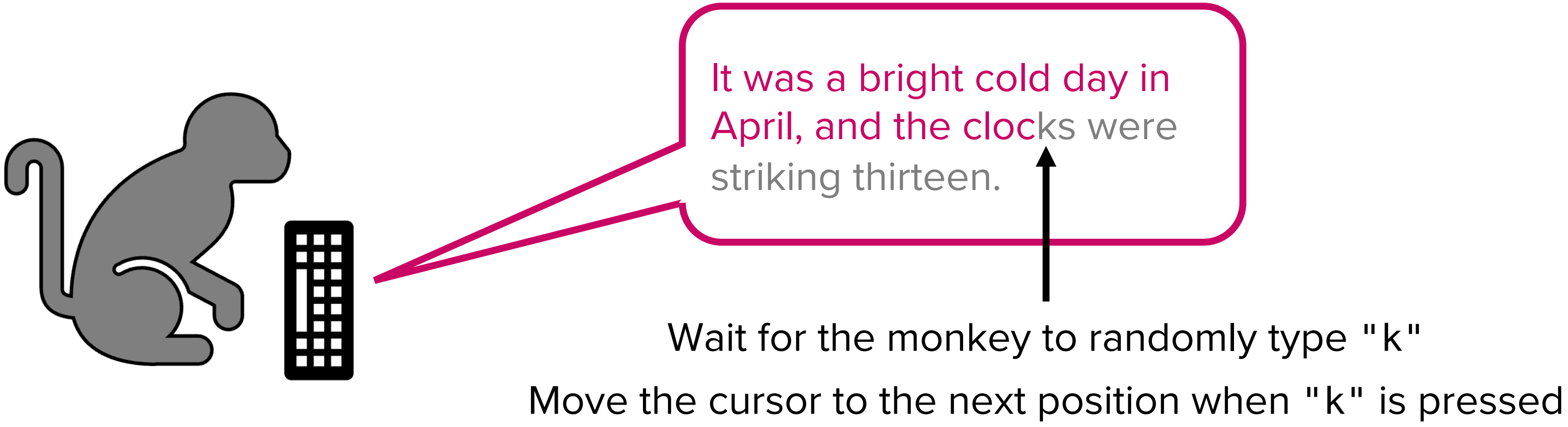
Recent breakthrough

- Greybox fuzzing with code coverage feedback



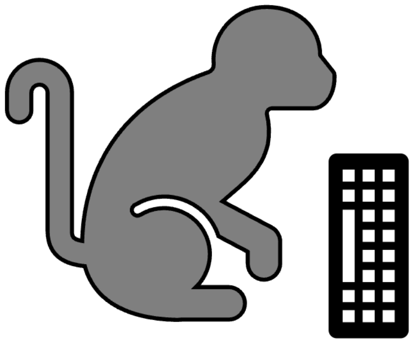
Breakdown of fuzzing efficiency

- A typing monkey problem (Greybox edition)
  - Keep the typed letters that are correct
  - Restart typing from the next position



Breakdown of fuzzing efficiency

- A typing monkey problem (Greybox edition)
  - Keep the typed letters that are correct
  - Restart typing from the next position



It was a bright cold day in April, and the clocks were striking thirteen.

Repeat for the rest of the sentence

The possibility is dramatically increased

Coverage feedback leads to better exploration

POSTECH

Target

```
x = input()

if x[0] == 'H':
  if x[1] == 'A':
    if x[2] == 'R':
      if x[3] == 'D':
        crash()
```

Seed `x = "LIFE"`

Test input `x = "LIFO"` `x = "5IFE"` `x = "L0VE"`

`x = "HEFE"` New branch. Interesting!

New seed `x = "HEFE"`

Test input `x = "LEFE"` `x = "HAVE"` New branch. Interesting!

New seed `x = "HAVE"`

⋮

$$\rightarrow P(\text{crash}) = \frac{1}{2^8} \times \frac{1}{4} = \frac{1}{2^{10}} > \frac{1}{2^{32}}$$

Get correct byte

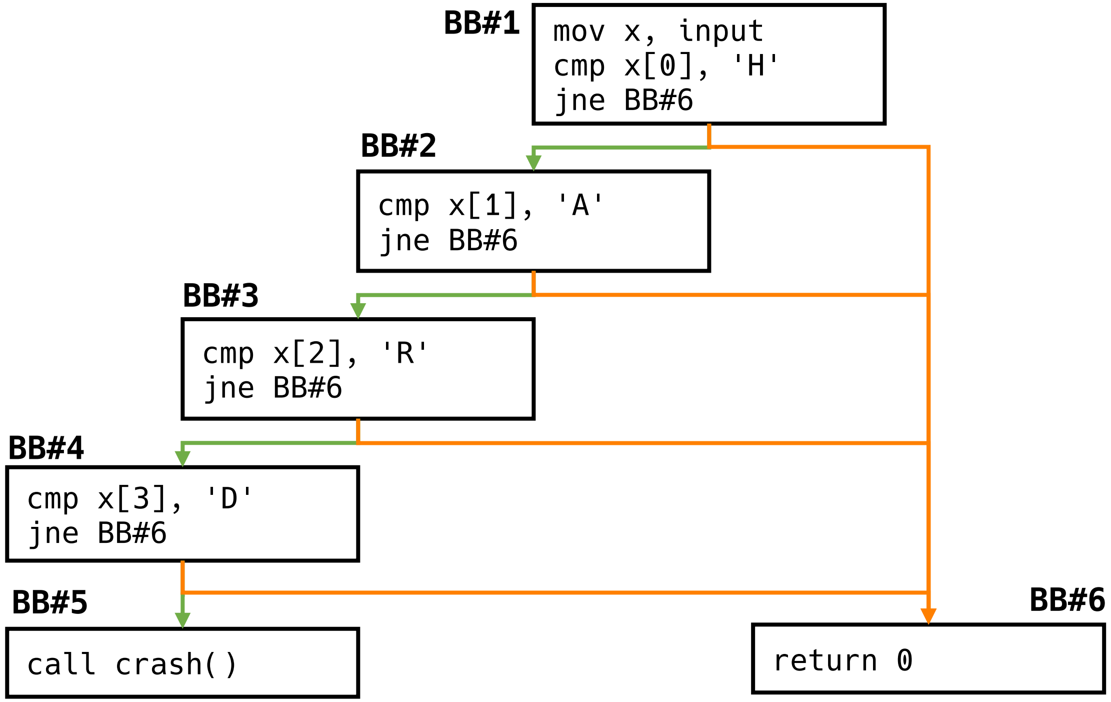
Select right position

How to track code coverage?

- Instrumentation: Modifying a program to enable analysis
  - For code coverage tracking, we want to record which branches of a program has been executed
  - We can instrument basic blocks
    - Basic block (BB): A sequence of code representing one branch of a software

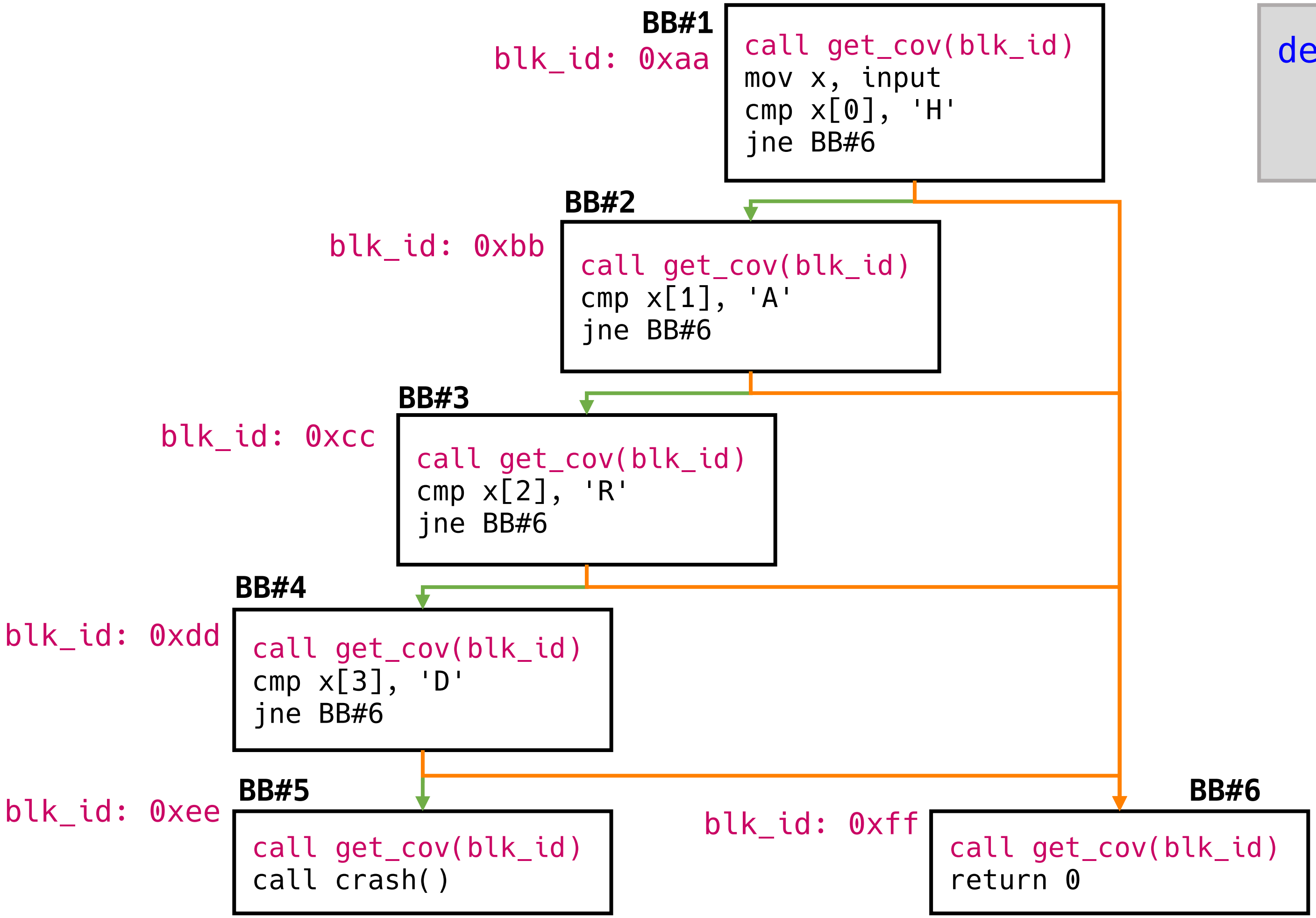
How to track code coverage?

- Control flow graph (CFG) of the "HARD" example
  - Consists of six basic blocks



How to track code coverage?

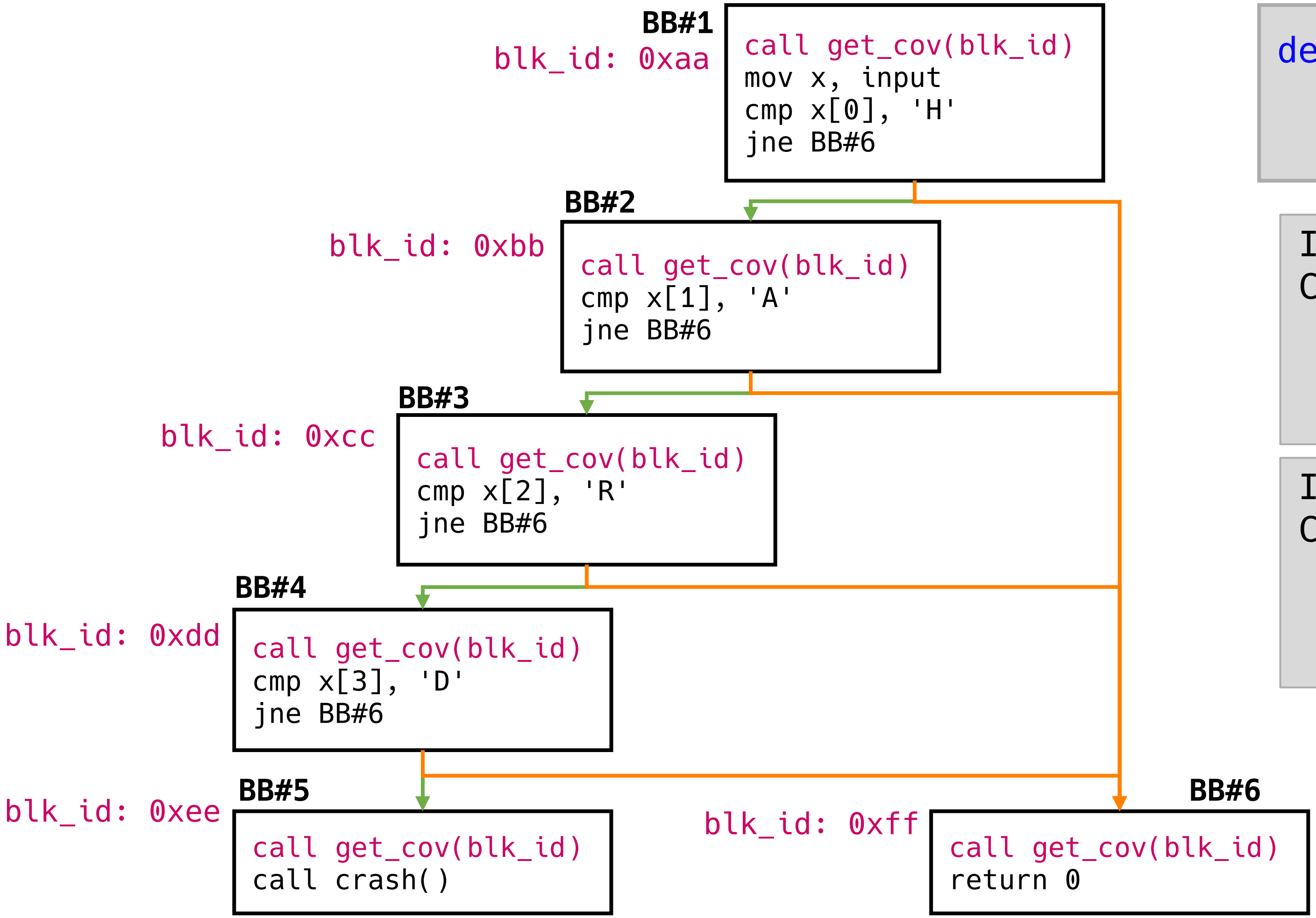
- Instrumentation for code coverage tracking




```
def get_cov(blk_id):
  global prev_blk_id
  record(prev_blk_id, blk_id)
```

How to track code coverage?

- Instrumentation for code coverage tracking




```
def get_cov(blk_id):
  global prev_blk_id
  record(prev_blk_id, blk_id)
```

```
Input: HASH
Coverage map:
  (0xaa,0xbb)
  (0xbb,0xcc)
  (0xcc,0xff)
```

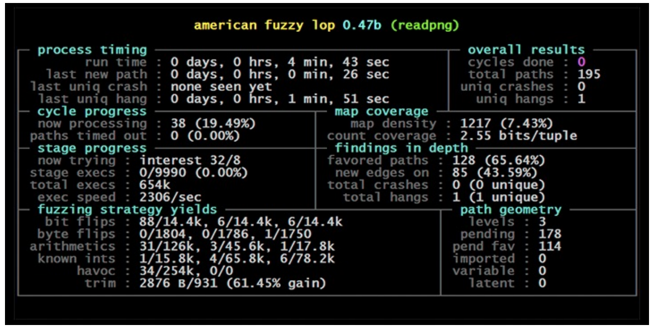
```
Input: HANK
Coverage map:
  (0xaa,0xbb)
  (0xbb,0xcc)
  (0xcc,0xff)
```

```
Input: HAND
Coverage map:
  (0xaa,0xbb)
  (0xbb,0xcc)
  (0xcc,0xff)
```

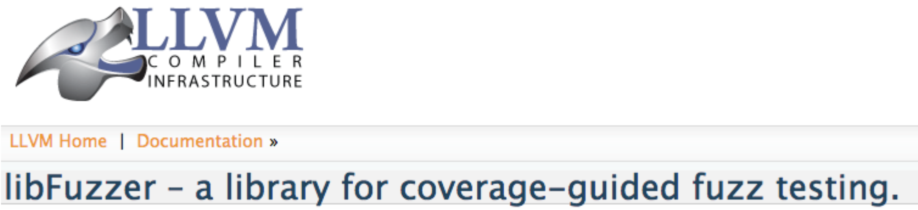
```
Input: HARM
Coverage map:
  (0xaa,0xbb)
  (0xbb,0xcc)
  (0xcc,0xdd)
  (0xdd,0xff)
```

New coverage found!

Feedback-driven greybox fuzzing is effective



AFL



libFuzzer

OSS-Fuzz

Discovered millions of <u>crashes</u> in complex software systems

Test Input Generation

Mutation- vs Generation-based fuzzing

POSTECH

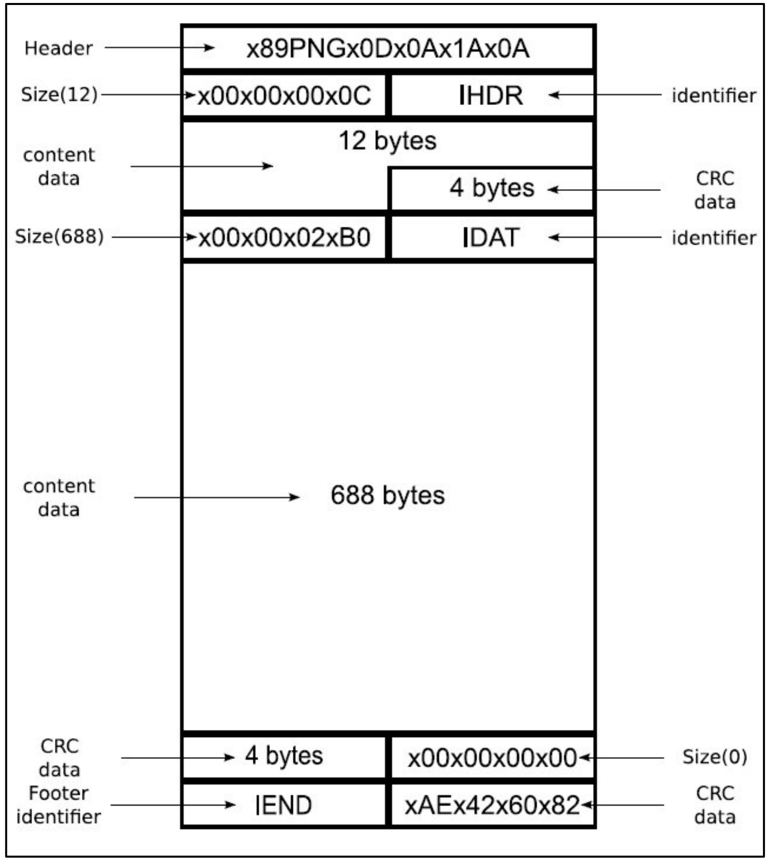
- Motivation: Randomly generated inputs are likely rejected by the program under test
  - e.g., When fuzzing a video player application, it is very unlikely that a fuzzer generates a properly formatted `mp4` file at random
- Two methods for better input generation
  - Mutation: Mutate a given seed to generate test inputs
    - Seed: A legitimate `mp4` file
  - Generation: Generate test inputs from an input model
    - Model: Specification of `mp4` file format

Mutation

- Frequently used mutation operators
  - Bit-flipping: Flip a randomly selected bit
    - e.g., `0xdead` (`0b1101 1110 1010 1101`) → `0xdeaf` (`0b1101 1110 1010 1111`)
  - Arithmetic operation: Select a byte and add/subtract a value
  - Randomization: Select a byte and randomize the value
  - Insertion and deletion: Add or remove bytes
  - Splicing: Crossover two test inputs
    - e.g., First half of input #1 + second half of input #2

Generation

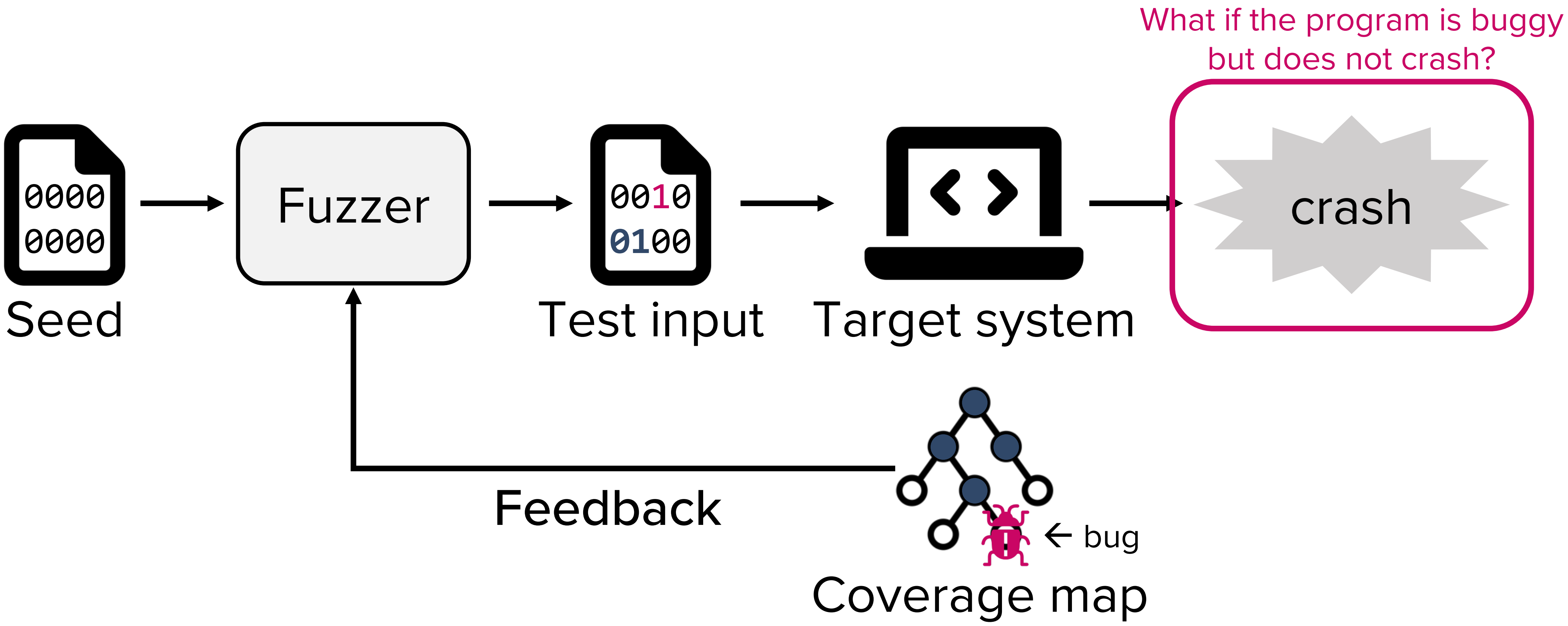
- Generate inputs that the program under test would accept
- A model describes the correct format
  - e.g., a grammar specifying the input format
    - PNG input has header and size fields
    - The header field must have the “magic number” of PNG in order for the input to be accepted by a PNG parser



PNG format

Bug Oracles

Mutation-based greybox fuzzing overview



A need for bug oracles

POSTECH

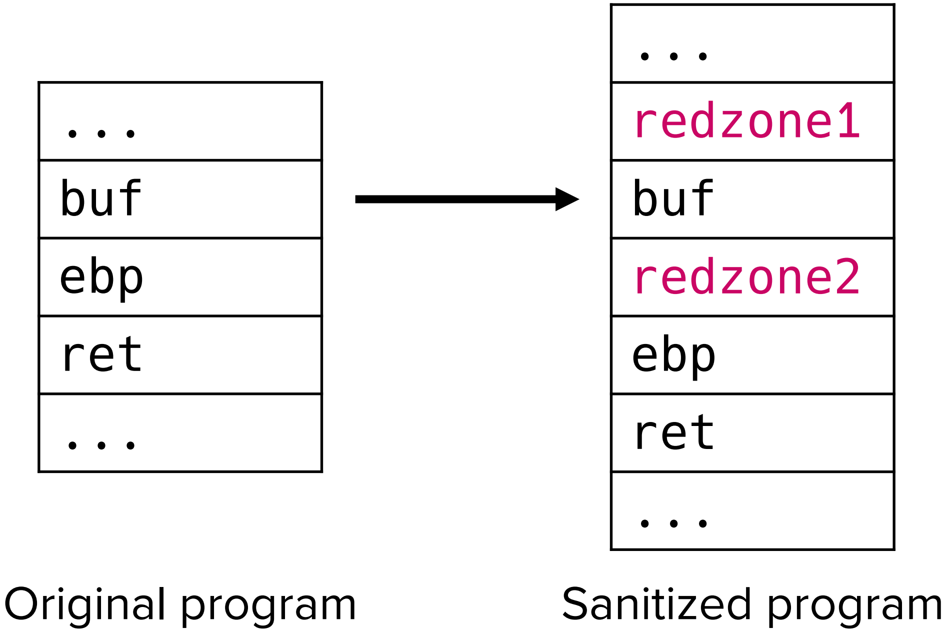
- What types of anomalous behavior do we want to find?
  - Crashes, but not all vulnerabilities lead to crashes (e.g., Lab 01)
  - Memory corruption: e.g., Use-After-Free (UAF) vulnerabilities
  - Hang: Program does not finish within a timeout period
  - Memory leaks, race conditions, specification violation, …
- A bug oracle detects any interesting behavior occurred during the execution of a program with the test input

Bug oracles in practice

- AddressSanitizer (ASan)
  - Detects buffer overflows and use-after-free
- ThreadSanitizer (TSan)
  - Detects data races
- MemorySanitizer (MSan)
  - Detects uses of uninitialized memory

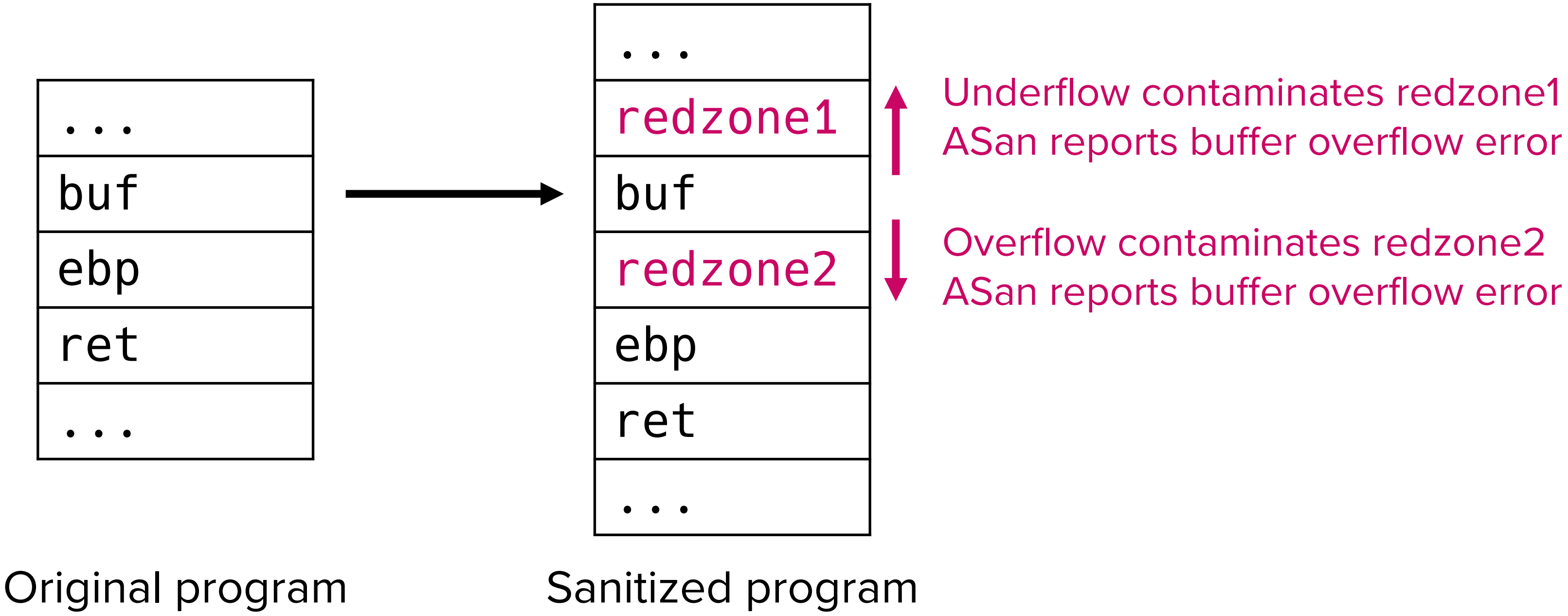
Address sanitizer

- Implemented as compiler module (available in clang and gcc)
  - Instruments all load and store instructions
  - Inserts redzones around each stack and global variable



Address sanitizer

- Runtime module checks whether redzones are touched when `buf` is read or something is written to `buf`



Address sanitizer in action

- Without ASan

```c
// oob.c
#include <stdio.h>
int numbers[] = { 1, 2, 3 };
int main() { /* classic out of bounds read error. */
  printf("The 4th number in my array is: %i\n", numbers[4]);
}
```

```
$ gcc oob.c -o oob
```

```
$ ./oob
The 4th number in my array is: 0
```

The bug is missed

Address sanitizer in action

- With ASan

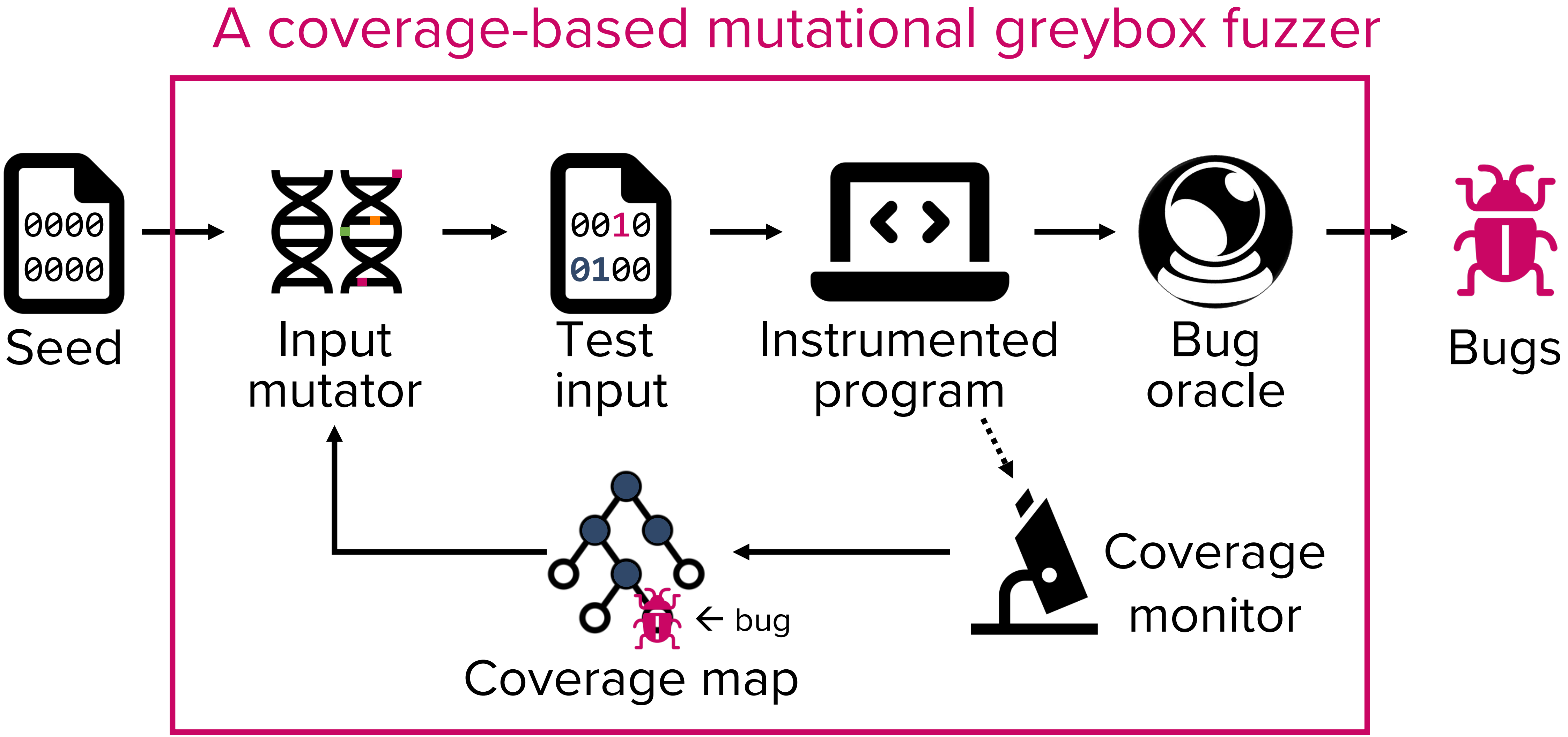
```
// oob.c
#include <stdio.h>
int numbers[] = { 1, 2, 3 };
int main() { /* classic out of bounds read error. */
  printf("The 4th number in my array is: %i\n", numbers[4]);
}
```

```
$ gcc oob.c -fsanitize=address -o oob_asan
```

```
$ ./oob_asan
=================================================================
==365994==ERROR: AddressSanitizer: global-buffer-overflow on address 0x55aceaed5030 at pc 0x55aceaed2223 bp
0x7ffe8cfc2c20 sp 0x7ffe8cfc2c10
READ of size 4 at 0x55aceaed5030 thread T0
    #0 0x55aceaed2222 in main (/home/seulbae/test/asan/oob_asan+0x1222)
    #1 0x7fa6faf1ed8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #2 0x7fa6faf1ee3f in __libc_start_main_impl ../csu/libc-start.c:392
    #3 0x55aceaed2124 in _start (/home/seulbae/test/asan/oob_asan+0x1124)

0x55aceaed5030 is located 4 bytes to the right of global variable 'numbers' defined in ‘oob.c:8:5' (0x55aceaed5020)
of size 12
SUMMARY: AddressSanitizer: global-buffer-overflow (/home/seulbae/test/asan/oob_asan+0x1222) in main
Shadow bytes around the buggy address:
  0x0ab61d5d29f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0ab61d5d2a00: 00 00 00 00 00 04[f9]f9 f9 f9 f9 f9 00 00 00 00
  0x0ab61d5d2a10: f9 f9 f9 f9 f9 f9 f9 f9 f9 f9 f9 f9 f9 f9 f9 f9
```

Final picture



Let's check the fuzzing results (from page 23)

POSTECH

- How many trials were required to find the bug through blackbox fuzzing?
  - Random mutation, no coverage feedback
  - Crash: Random 4 bytes being identical to `"\xde\xad\xbe\xef"`
    - Theoretically requires $2^{32} \approx 4.2$ billion trials
    - Experimentally: (see terminal)

vs AFL

- AFL: The most widely used coverage-guided mutation-based fuzzer
  - Instrumentation for code coverage using AFL’s custom complier

```
$ afl-cc target.c -O0 -o target_afl
```

  - Prepare a seed input

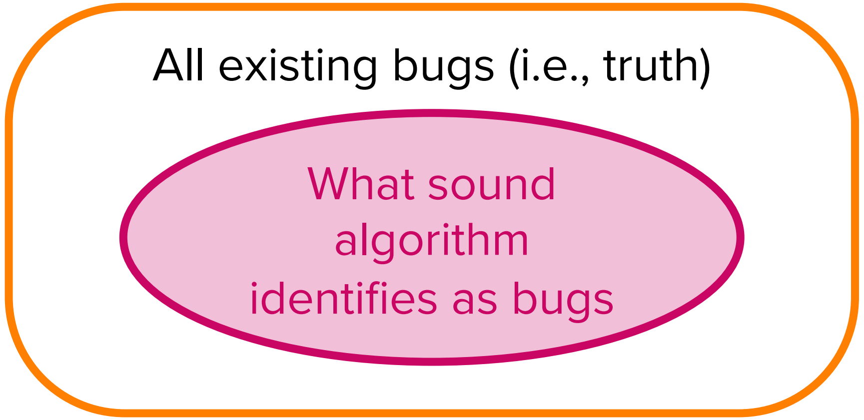
```
$ rm -rf in out
$ mkdir in out
$ echo -ne "\xff\xff\xff\xff" > in/seed
```

  - Run fuzzer

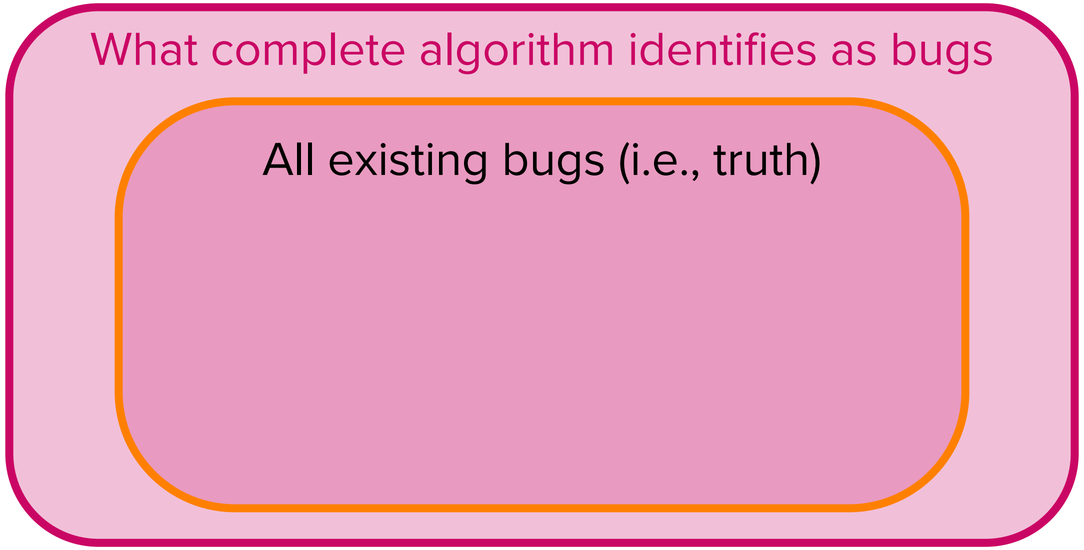
```
$ afl-fuzz -i in -o out -- ./target_afl
```

Questions

- Is fuzzing sound? (no false positives?)
- Is fuzzing complete? (no missed bugs?)



→ Fuzzer can have FP if its oracles are unsound



→ Fuzzer can miss bugs as it partially explores target program

Conclusion: Fuzzing is neither sound nor complete, but it is practical and scalable

Questions?