CSCI 4907/6545 Software Security

Fall 2025

Instructor: Jie Zhou

Department of Computer Science

George Washington University

Slides materials are partially credited to Gang Tan of PSU.

Outline

- Review: Control-flow Integrity
- Testing to find software bugs
- Fuzz testing (Fuzzing)

What are essential to a programming language?

Essential Components of a Programming Language

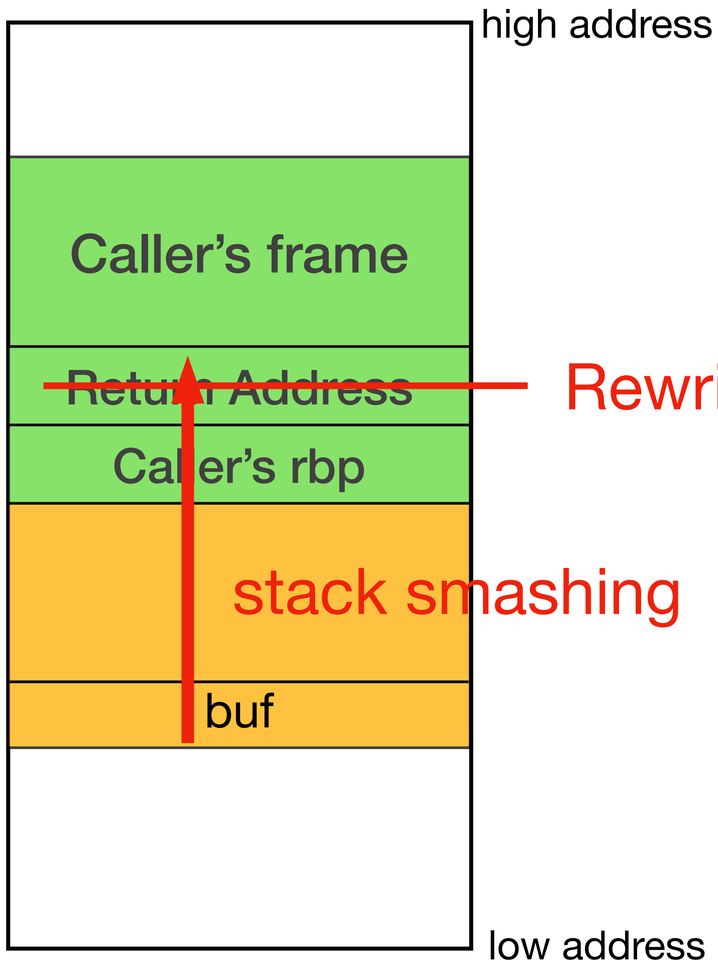
- Data types
  - int, char, boolean, etc.
- Operatiors
  - arithmetic, move, comparison, etc.
- **Control flow**
  - absolute/conditional transfer, loop

Attacks often start with misusing data types, and then misdirecting the control flow to launch malicious computations.

Control Flow

- Data types
  - int, char, boolean, etc.
- Operatiors
  - arithmetic, move, comparison, etc.
- Control flow
  - absolute/conditional transfer, loop
    - Function returns, calls, indirect jumps, goto statements (dangerous and rare today)

Return Address Corruption



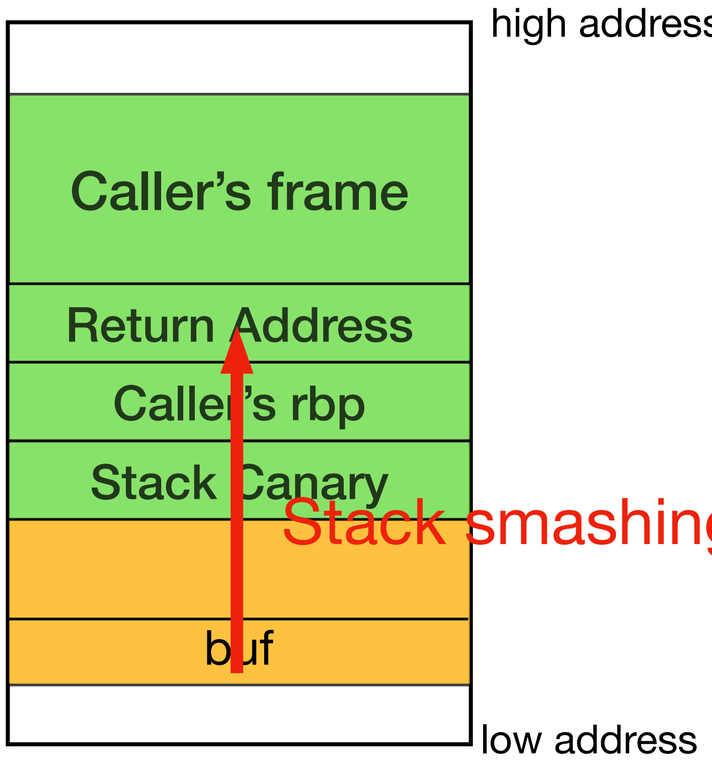
Rewritten with attacker-selected address

?

How to detect/prevent stack smashing?

Stack Canaries

- A random value put on the stack to detect stack buffer overflows
  - Located close to the return address
  - Function checking if the canary changed before using the return address



Stack smashing will corrupt the canary.

Use Compilers to Add Stack Canaries

- clang
  - `-fstack-protector`
    - Add stack canaries for functions with a `char` array or calls to `alloca()`
  - `-fstack-protector-strong`
    - Add stack canaries for all functions with arrays, alloca, or taking addr of local vars
  - `-fstack-all`
    - Add stack canaries for all functions
- gcc has `-fstack-protector` on by default

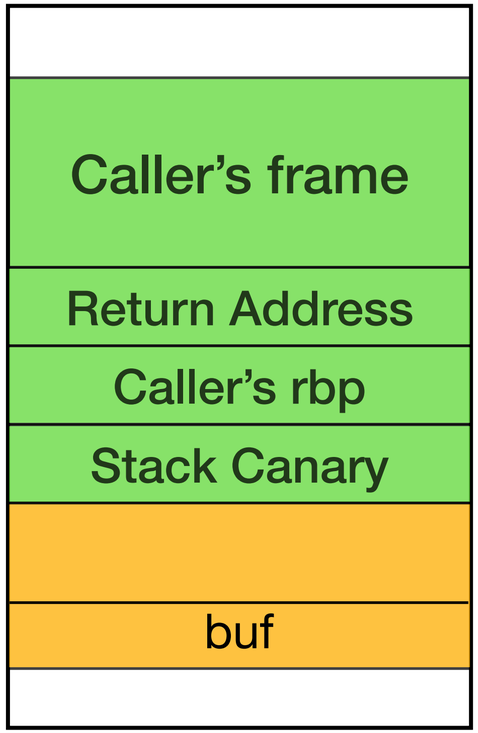
Function Frame with A Stack Canary

```
0x00001160 <+0>:   push   %rbp
0x00001161 <+1>:   mov    %rsp,%rbp
0x00001164 <+4>:   sub    $0x20,%rsp
0x00001168 <+8>:   mov    %fs:0x28,%rax
0x00001171 <+17>:  mov    %rax,-0x8(%rbp)
0x00001175 <+21>:  mov    %edi,-0x18(%rbp)
0x00001178 <+24>:  mov    %esi,-0x1c(%rbp)
0x0000117b <+27>:  xor    %eax,%eax
0x0000117d <+29>:  lea    -0x14(%rbp),%rdi
0x00001181 <+33>:  call   0x1050 <gets@plt>
0x00001186 <+38>:  mov    %fs:0x28,%rax
0x0000118f <+47>:  mov    -0x8(%rbp),%rcx
0x00001193 <+51>:  cmp    %rcx,%rax
0x00001196 <+54>:  jne    0x11a2 <foo+66>
0x0000119c <+60>:  add    $0x20,%rsp
0x000011a0 <+64>:  pop    %rbp
0x000011a1 <+65>:  ret
0x000011a2 <+66>:  call   0x1030 <__stack_chk_fail@plt>
```

- Load canary from `%fs:0x28` onto the stack
  - `fs`: segment register
  - Canary was initialized during program initialization

- Load original canary to a register
- Load canary saved by this function to a register
- Compare the two register
  - If equal, keep normal execution
  - If unequal, jump to `__stack_chk_fail()`

Weaknesses of Stack Canaries

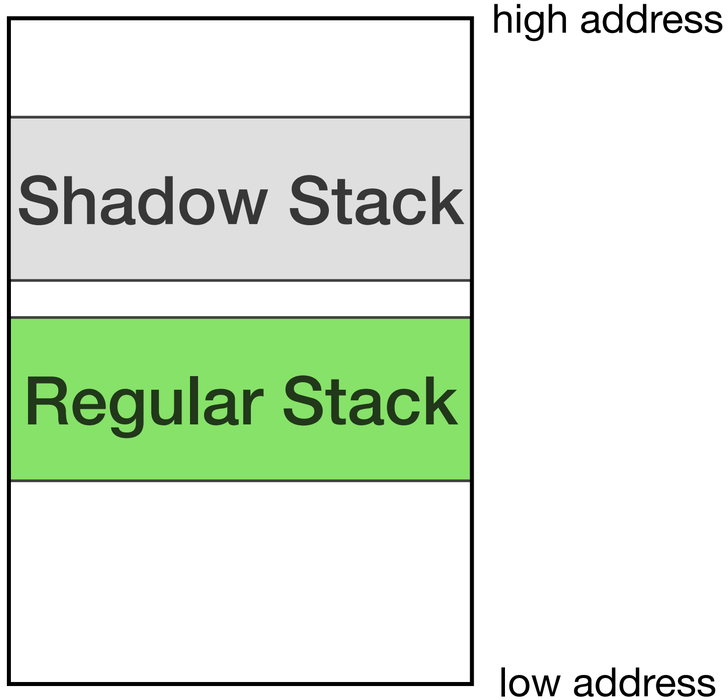


- Disclosure attacks
  - Buffer overread may leak the value of stack canaries.
    - Infamous buffer overread example: Heartbleed attack
    - Leak from the segment register
    - Leak from the stack
- Most effective in detecting *consecutive* stack overflows
  - Cannot detect arbitrary out-of-bound memory corruption
- Only protecting return addresses
  - Other security important data may still be corrupted
    - e.g., function pointers defined as local variables

Shadow Stack for Return Address Integrity

A separate stack dedicated to storing a copy of each return address

- A program can use the return address on the shadow stack
  - Checking the validity of the original return address
  - Directly using the copy on the shadow stack to return

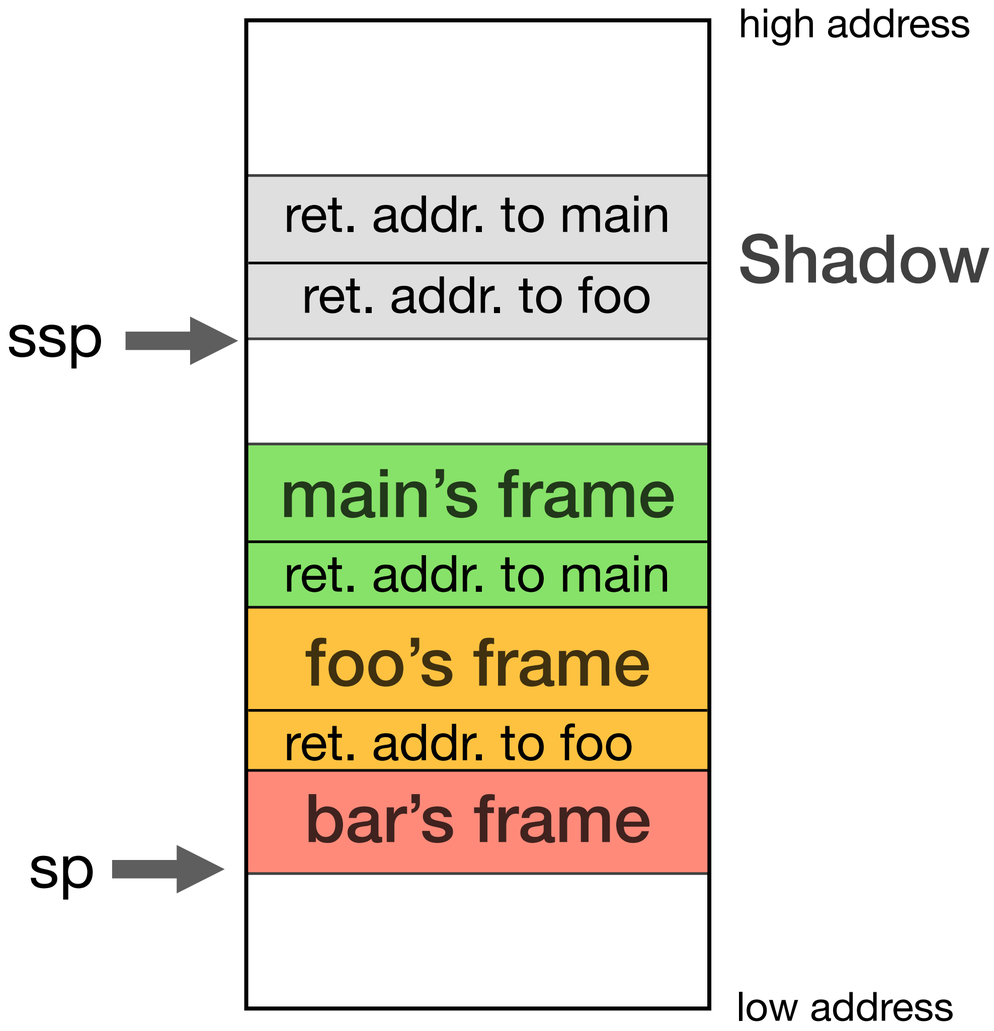


Compact Shadow Stack

- Where to put the shadow stack?
  - Usually also in the stack region, but could be in other memory regions.
- How to organize the shadow stack?
  - i.e., where exactly to store copies of return addresses a the shadow stack?
    - All return addresses are put together
- How to index the shadow stack?
  - Maintain a special shadow stack pointer (ssp) to the top of the shadow stack

Compact Shadow Stack

e.g., `main()` calls `foo()` and `foo()` calls `bar()`



Shadow Stack

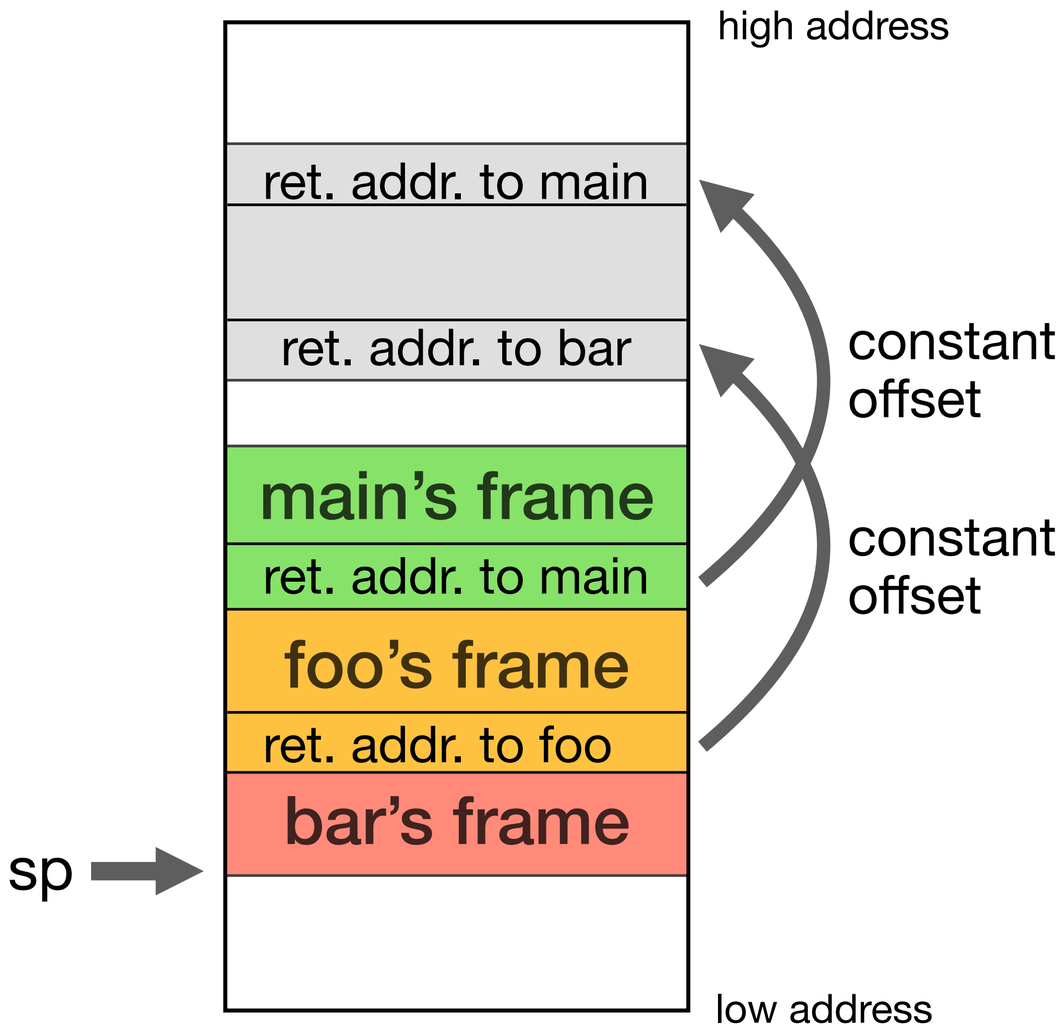
- Function prologue
  - Save ret addr to shadow stack via ssp
  - Update ssp
- Function epilogue
  - Load ret addr from shadow stack via ssp
  - Use the saved ret addr
  - Update ssp

Parallel Shadow Stack

- Where to put the shadow stack?
  - Usually also in the stack region, but could be in other memory regions.
- How to organize the shadow stack?
  - i.e., where exactly to store copies of return addresses a the shadow stack?
    - Return addresses on the shadow stack are scattered to match the layout of the regular stack.
- How to index the shadow stack?
  - Use a constant offset from the regular stack pointer

Parallel Shadow Stack

e.g., `main()` calls `foo()` and `foo()` calls `bar()`



- Shadow stack is of the same size as the regular stack.
- Constant offset between each ret addr on the regular and shadow stack
- Use sp + offset to index the ret addr on the shadow stack
  - Prologue: Save ret addr to sp + offset
  - Epilogue: Load ret addr from sp + offset

Strengths and Weaknesses of Parallel Shadow Stack

- Strengths
  - Fast
- Weaknesses
  - Memory consumption overhead is high.
  - Hard-coded offset is a security hazard (easily accessible to adversaries).
  - Compatibility issues for multi-threaded programs
    - Constrained address space layout

Weaknesses of Shadow Stack for Return Addresses

- Increased complexity due to additional abstraction
  - Increasing the complexity of the protected software
  - Performance and memory overhead
  - New security risks
    - e.g. race conditions due to x86's use of stack for return addresses
- Limited scope of protection
- Integrity of shadow stack itself
  - Shadow stack protects return addresses, who protects shadow stack?

Overflowing Heap Critical User Data

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];             /* vulnerable input buffer */
    void (*process)(char *);  /* pointer to function */
} chunk_t;

void showlen(char *buf) {
    int len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[]) {
    chunk_t *next = malloc(sizeof(chunk_t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

Overflow the buffer on the heap to set the function pointer to an arbitrary address.

Control-flow Graph (CFG)

A program representation using graph notations.

- Node: A sequence of instructions without control flow transfers
- Edge: Control flow transfer between nodes
- Usually represent one function, but can also represent a whole program
- Critical to analyzing programs

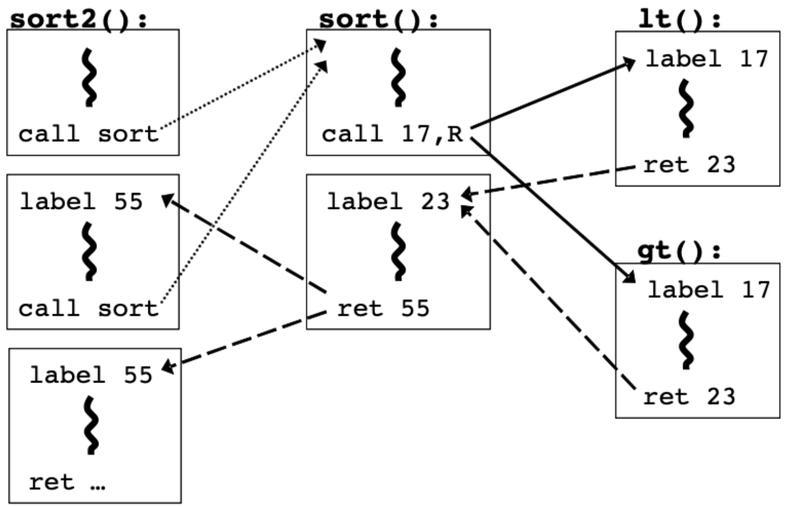
How to Enforce Control-flow Integrity

- Compute a CFG
- For indirect control flow transfers, compute their target destinations
  - Mostly via compiler or binary rewriting, but possible at run-time
- Before an indirect transfer, check the validity of the destination
- Two CFI policies:
  - Label-based and type-based

Label-based CFI

- Assign and insert a label (ID) before each indirect transfer destination
- Before executing an indirect transfer, check the destination's label
  - Similar to using stack canaries / shadow stacks

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



Direct forward transfer

Indirect forward transfer

Backward transfer

Issues/Weaknesses of Label-based CFI

- Collision of labels with existing data/code
  - Generally harder on x64 than RISC arch such as ARM
    - e.g., in ARM, “`mov r0, r0`” (`0x4600`) can serve as a unique label
- Very challenging to compute a precise CFG
  - A practical implementation:
    - Use the same label for all function entrance
    - Use the same label for all instructions following a call

Type-based CFI

- Use function signature (types of return value and args) as the target identifier
- Each signature is assigned a unique type ID
  - e.g., “`void foo()`” and “`int bar()`” will have different type IDs.
- Before an indirect call, check destination’s type ID against a predetermined ID

Weaknesses of Type-based CFI

- Needs source code or compiler IR; cannot do binary rewriting
- Only works for indirection function calls but not returns
- Allows indirect calls to a group of functions
  - What if we have "`int myFunc(const char *str)`", which has the same type ID as "`int system(const char *command)`"?

Fundamental Weaknesses of CFI

- Performance and code size overhead
- Cannot be 100% accurate
- Backward CFI (i.e., protecting return addresses) is critical.
- An extra lay of complexity
  - May need support from OS
- Limited protection scope
  - Does not prevent data-only attacks
  - Needs W^X/DEP

Goal of Testing

Detecting bugs automatically before they can cause damage.

Bugs vs. Vulnerabilities

Wikipedia: *“A software bug is a bug in computer software.”*

Wikipedia: *“In engineering, a bug is a design **defect** in an engineered system that causes an undesired result.”*

Wikipedia: *“Vulnerabilities are **flaws** in a computer system that weaken the overall security of the system.”*

Vulnerabilities -> Exploitable Bugs

https://en.wikipedia.org/wiki/Vulnerability_(computer_security)

“50% of my company employees are testers, and the rest spends 50% of their time testing!”

Bill Gates 1995

Dynamic Analysis

Analyzing the program when it is running with a specific input.

- Many techniques
  - Testing, fuzzing, taint tracking, differential testing, execution integrity monitoring, …

Program Testing

The process of running a program on a set of test cases and comparing the actual results with expected results.

- E.g., for the implementation of a factorial function, test cases could be {0, 1, 5, 10}.
- Testing cannot guarantee program correctness.
  - What's the simplest program that can fool the test cases above?

Impossibility of Testing

A software engineering walks into a bar and orders

- 1 beer
- 2 beers
- 9,999 beers
- -1 beer
- “abc” beers
- A lizard in a glass

Testing completed.

A real customer walks into the bar and ask where the toilets are.

The bar goes up in flames.

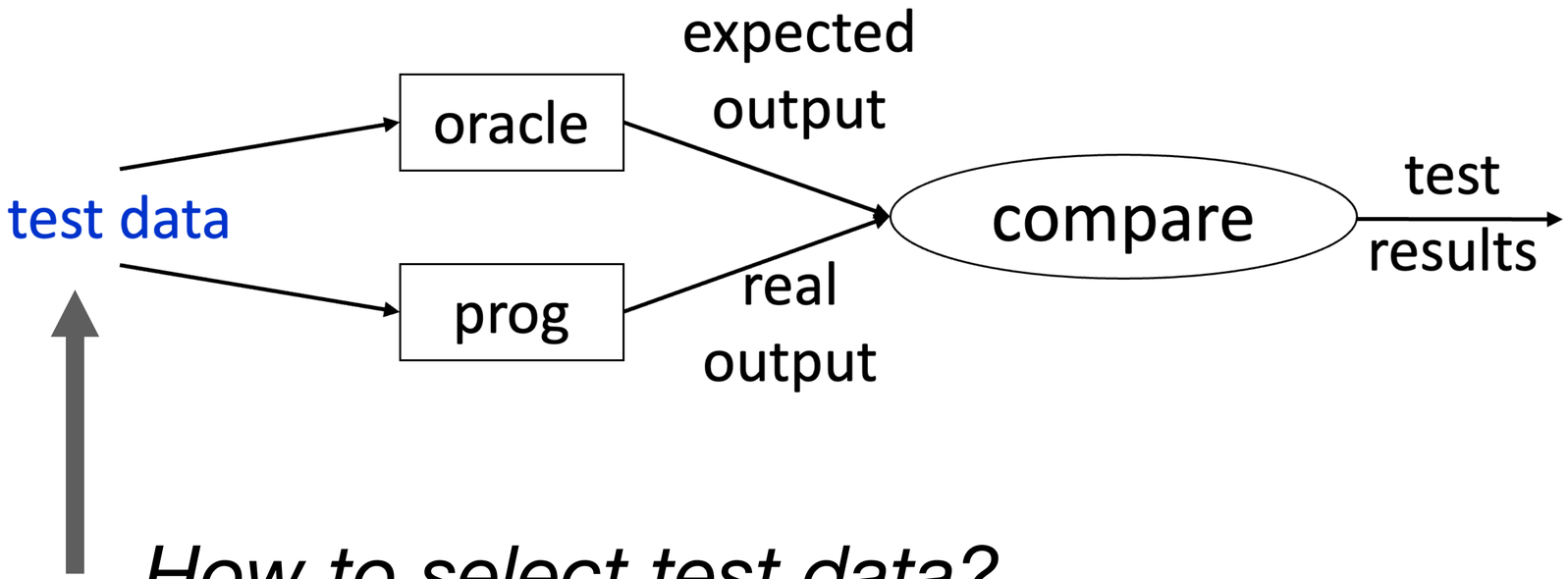
Program Testing

The process of running a program on a set of test cases and comparing the actual results with expected results.

- E.g., for the implementation of a factorial function, test cases could be {0, 1, 5, 10}.
- Testing cannot guarantee program correctness.
  - What's the simplest program that can fool the test cases above?
- However, testing can catch many bugs.

Testing Process

Test oracle: A mechanism/tool that determines the correctness of the tested program under a test case (input).



How to select test data?

Selecting Test Data

- Testing is w.r.t. a finite test set.
  - Exhaustive testing is usually not possible
  - E.g, a function takes 3 integer inputs, each ranging over 1 to 1000
    - Assume each test takes 1 second
    - Exhaustive testing would take 10^9 = 1 billion seconds (~31.7 years!)
- How should we design the test set?
  - Black-box testing
  - White-box (or, glass-box) testing

Black-box Testing

Generating test cases based on specification alone, without considering the implementations (internals).

- Only focusing on the inputs and outputs
- Advantages
  - No need for code knowledge
  - Test cases are not biased toward an implementation.

Generating Black-box Testing Cases

```
static float sqrt (float x, float epsilon)
// Requires: x >= 0 && .00001 < epsilon < .001
// Effects: Returns sq such that x - epsilon <= sq*sq <= x + epsilon
```

- The precondition can be satisfied
  - Either “x=0 and .00001 < epsilon < .001”,
  - Or “x>0 and .00001 < epsilon < .001”
- Any test data should cover these two cases.
- Also test the case when x is negative and epsilon is outside the expected range.

More Examples of Black-box Testing

```
static boolean isPrime (int x)
// Effects: If x is a prime returns true else false
```

- Test cases: cover both true and false cases
- Also test numbers 0, 1, 2, and negative numbers

```
static int search (int[ ] a, int x)
// Effects: If a is null throws NullPointerException else if x is in a,
// returns i such that a[i]=x, else throws NotFoundException
```

- Test cases:
  - a = null
  - A case where `a[i] = x` for some `i`
  - A case where x is not in the array a

Boundary Conditions

- Common programming mistakes: not handling boundary cases
  - Input is zero
  - Input is negative
  - Input is null
  - …
- Test data should cover these boundary cases.

Example Program

```
static void appendVector (Vector v1, Vector v2)
// Effects: If v1 or v2 is null throws NullPointerException, else removes all
// elements of v2 and appends them in reverse order to the end of v1
```

- Test cases?
  - v1 = null;
  - v2 = null;
  - v1 is empty
  - v2 is empty
  - …
  - v1 and v2 refer to the same vector

White-box Testing

Looking into the internals of the program to figure out a set of test cases

```
static int maxOfThree (int x, int y, int z)
// Effects: Return the maximum value of x, y and z
```

- Black-box test cases?
- Assume you are given its implementation

```
static int maxOfThree (int x, int y, int z) {
    if (x>y)
      if (x>z) return x; else return z;
    else if (y>z) return y; else return z; }
```

- Looks like the implementation is divided into four cases
  - x > y & x > z
  - x > y & x <= z
  - x <= y & y > z
  - x <= y & y <= z

Test Coverage

- Idea: code that has not been covered by tests are likely to contain bugs.
  - Divide a program into a set of elements
    - The definition of elements leads to different kinds of test coverage.
  - Define the coverage of a test suite to be:

$$\frac{\text{\# of elements executed by the test suite}}{\text{\# of elements in total}}$$

Why is Test Coverage Important?

- Test quality is determined by the coverage of the program by the test set so far.
- Benefits:
  - Can be used as a **stopping rule**: e.g., stop testing if 95% of elements have been covered.
  - Can be used as a **metric**: a test set that has a test coverage of 80% is better than one that covers 70%
  - Can be used in a **test case generator**: look for a test which exercises new elements not covered by the tests so far

Different Coverage Criteria

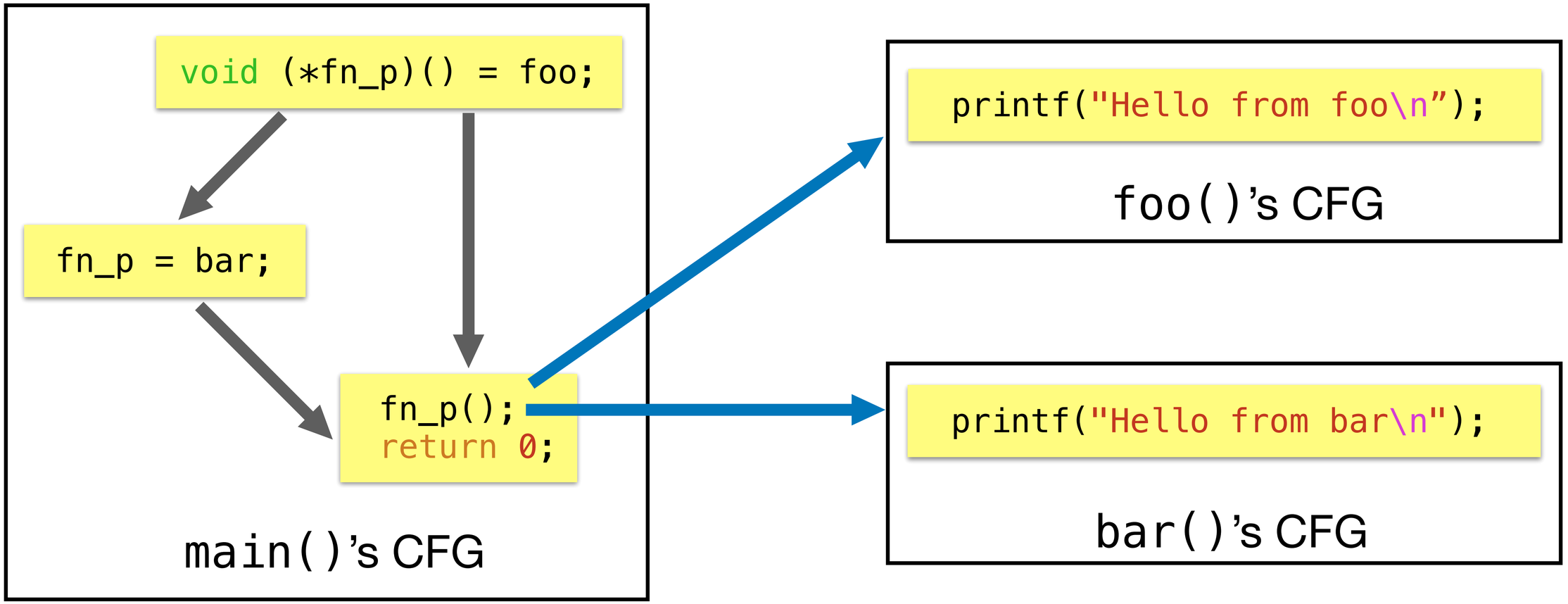
- Usually based on Control Flow Graph (CFG)

Control-flow Graph (CFG)

A program representation using graph notations.

- Node: A sequence of instructions without control flow transfers
- Edge: Control flow transfer between nodes
- Usually represent one function, but can also represent a whole program
- Critical to analyzing programs

Example of Control-flow Graph (CFG)

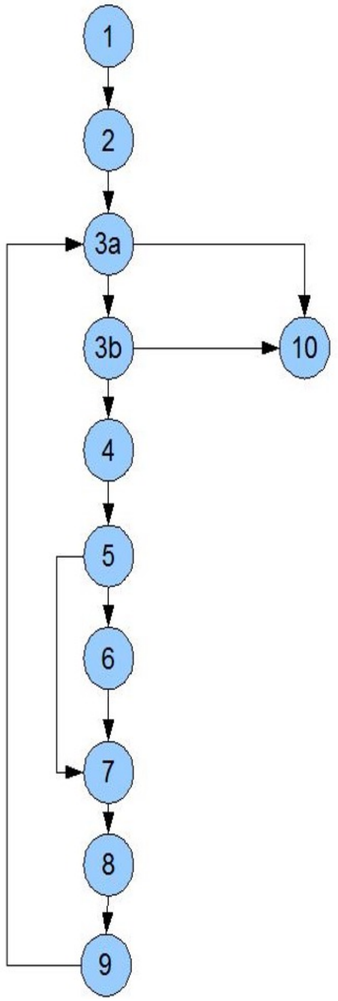


Different Coverage Criteria

- Usually based on Control Flow Graph (CFG)
  - Statement coverage
  - Edge coverage
  - Path coverage
  - …

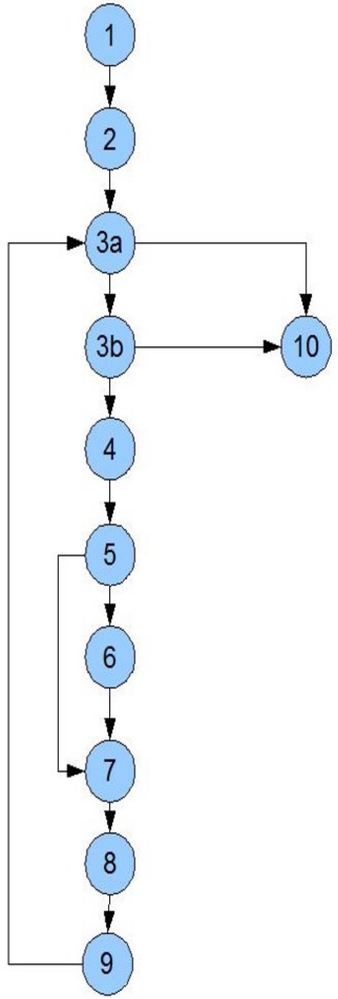
A Running Example

```
// Input: table is an array of numbers;
// Input: n is the size of table
// Input: element is the element to be found
// Output: found indicates whether the element
//         is in the table

1: found = false;
2: counter = 0;
3: while ((counter < n) && (!found))
4: {
5:   if (table[counter] == element)
6:     found = true;
7:
8:   counter++;
9: }
10:
```



Statement Coverage

```
1: found = false;
2: counter = 0;
3: while ((counter < n) && (!found))
4: {
5:   if (table[counter] == element)
6:     found = true;
7:
8:   counter++;
9: }
10:
```



- Test data: table = {3, 4, 5}; n= 3; elements = 3
  - Does it cover all statements?
    - Yes
  - But does it cover all edges?
    - No, missing 3a -> 10 and 5 -> 7

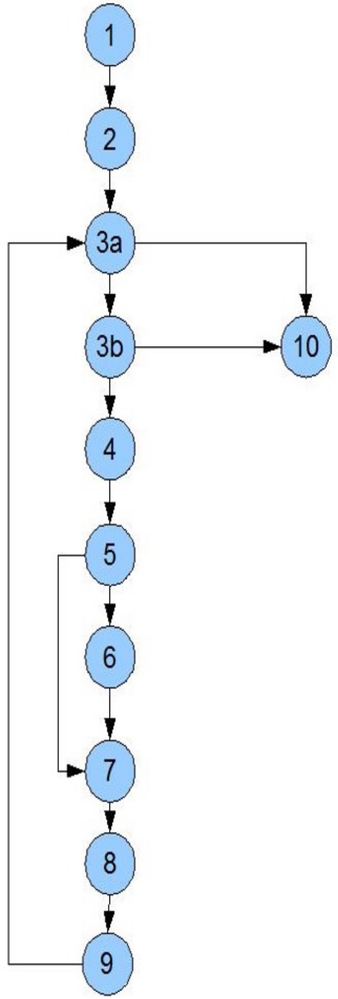
Statement Coverage in Practice

- 100% is very hard.
  - Usually about 85% coverage
- Microsoft reports 80%–90% statement coverage.
- Safety-critical applications usually require 100% statement coverage.
  - Boeing requires 100% statement coverage.

Does 100% statement coverage mean the program is correct?

Edge Coverage

```
1: found = false;
2: counter = 0;
3: while ((counter < n) && (!found))
4: {
5:   if (table[counter] == element)
6:     found = true;
7:
8:   counter++;
9: }
10:
```



- Test data to cover all edges
  - table={3,4,5}; n = 3; element=3
  - table={3,4,5}; n = 3; element=4
  - table={3,4,5}; n = 3; element=6

Path Coverage

- Path-complete test data: Covering every possible control flow path
- For example:

```
static int maxOfThree (int x, int y, int z) {
    if (x>y)
      if (x>z) return x; else return z;
    else if (y>z) return y; else return z; }
```

  - Test data is complete as long as the following four case are covered:
    - x > y & y > z
    - x > y & x <= z
    - x <= y & y > z
    - x <= y & y <= z

Does 100% path coverage
mean the program is correct?

Covering All Paths

- A program passes path-complete test data doesn't mean it's correct.
- For example:

```
static int maxOfThree (int x, int y, int z) {
    return x;
}
```

  - "x=5, y=4, z=3" would pass the test and be path complete.
- Same goes for the case of all-statement coverage, or all-edge coverage.

Possibly Infinite Number of Paths

- Loop may cause infinite # of paths
  - In general, impossible to cover all of them.
- One heuristic
  - Include test data that cover zero, one, and two iterations of a loop
  - Why two iterations?
    - A common programming mistake is failing to reinitialize data in the second iteration.
  - This offers no guarantee, but can catch many errors.

Path Coverage In The Presence of Loops

```
1: found = false;
2: counter = 0;
3: while ((counter < n) && (!found))
4: {
5:   if (table[counter] == element)
6:     found = true;
7:
8:   counter++;
9: }
10:
```

Figuring out a test suite that covers zero, one, and two iterations of the loop.

- Zero iterations: table={ }; n=0; element=3
- One iteration: table={3,4,5}; n=3; element=3
- Two iterations: table={3,4,5}; n=2; element=4

Combine Them All

- A good set of test data combines various testing strategies.
  - Black-box testing
    - Generating test cases by specifications
    - Boundary conditions
  - White-box testing
    - Test coverage (e.g., being edge complete)

Example: Palindrome

```
// Effects: If s is null, throws NullPointerException,
// else returns true if s is a palindrome.

boolean palindrome(String s) throws NullPointerException {
  int low = 0;
  int high = s.length() - 1;
  while (high > low) {
     if (s.charAt(low) != s.charAt(high))
       return false;
     low++;
     high--;
  }
  return true;
}
```

What test cases, esp. boundaries cases, should be used?

Test Data for the Example

- Based on specifications
  - s = null
  - s = “deed”
  - s = “abc”
  - s = “” (boundary condition)
  - s = “a” (boundary condition)
- Based on the program
  - Not executing the loop
  - Returning false in the first iteration
  - Returning true after the first iteration
  - Returning false in the second iteration
  - Returning true after the second iteration

```
int low = 0;
int high = s.length() - 1;
while (high > low) {
   if (s.charAt(low) != s.charAt(high))
     return false;
   low++;
   high--;
}
return true;
```

Test Coverage Tool: `llvm-cov/gcov`

- llvm-cov/gcov: Emit code coverage information
  - Insert additional code through a compiler to track line coverage and branch coverage
- Bundled with clang/gcc
  - e.g., `clang/gcc –coverage demo.c -o demo`
    - Generated binary contains instrumentation code for code coverage.
    - Many other interesting compiler options. See “clang/gcc --help” & “man gcov”

Example of Using gcov

```c
#include <stdio.h>

int main (void) {
  int i;
  for (i = 1; i < 10; i++) {
      if (i % 3 == 0)
        printf ("%d is divisible by 3\n", i);
      if (i % 11 == 0)
        printf ("%d is divisible by 11\n", i);
    }
  return 0;
}
```

- `clang --coverage demo.c -o demo`
- `./demo` // Will generate a demo-demo.gcda
- `gcov demo-demo-gcda` // Will generate a coverage report demo.c.gcov

gcov's Code Coverage Report

```
    -:    0:Source:demo.c
    -:    0:Graph:demo-demo.gcno
    -:    0:Data:demo-demo.gcda
    -:    0:Runs:1
    -:    0:Programs:1
    -:    1:#include <stdio.h>
    -:    2:
    1:    3:int main (void) {
    -:    4:  int i;
   10:    5:  for (i = 1; i < 10; i++) {
    9:    6:      if (i % 3 == 0)
    3:    7:        printf ("%d is divisible by 3\n", i);
    9:    8:      if (i % 11 == 0)
#####:    9:        printf ("%d is divisible by 11\n", i);
    9:   10:    }
    1:   11:  return 0;
    -:   12:}
    -:   13:
```

Automated Test Generation

- Designing tests with good coverage is hard; not as clean as the examples.
  - Manually designing a good test set is a major task.
  - 100% coverage almost never achieved in practice
- Q: Can we automate it?
- We can, for certain situations.
  - Pre-condition and post-condition based test generation
  - Fuzzing can be viewed as a way of automated test generation

Pre- and Post-Conditions

- A **pre-condition** is a predicate.
  - Assumed to hold before a piece of code executes
- A **post-condition** is a predicate.
  - Expected to hold after a piece of code executes, whenever the pre-condition holds.
- Example

```
static float sqrt (float x, float epsilon)
// Pre: x >= 0 && .00001 < epsilon < .001
// Post: Returns sq such that x - epsilon <= sq*sq <= x + epsilon
```

Test Generation Using Pre- and Post-conditions

- A simple algorithm

```
while (true) {
   test = randomlyGenerate();
   if (precondition(test)) {
       ret = runTest(test);
       if (!postcondition(ret,test)) error();
   }
}
```

Fuzzing

UW–MADISON
COMPUTER SCIENCES

Barton P. Miller

Vilas Distinguished Achievement Professor
Amar & Balinder Sohi Professor in Computer Sciences

Research Interests

Binary code analysis and instrumentation, distributed and parallel program performance and tools, software security, scalable systems, operating systems, software testing.

Brief Biography

Barton Miller is a Vilas Distinguished Achievement Professor and the Amar and Belinder Sohi Professor of Computer Sciences at the University of Wisconsin, Madison. He received his B.A. degree from the University of California, San Diego in 1977, and M.S. and Ph.D. degrees in Computer Science from the University of California, Berkeley in 1980 and 1984.

Professor Miller is a Fellow of the ACM.

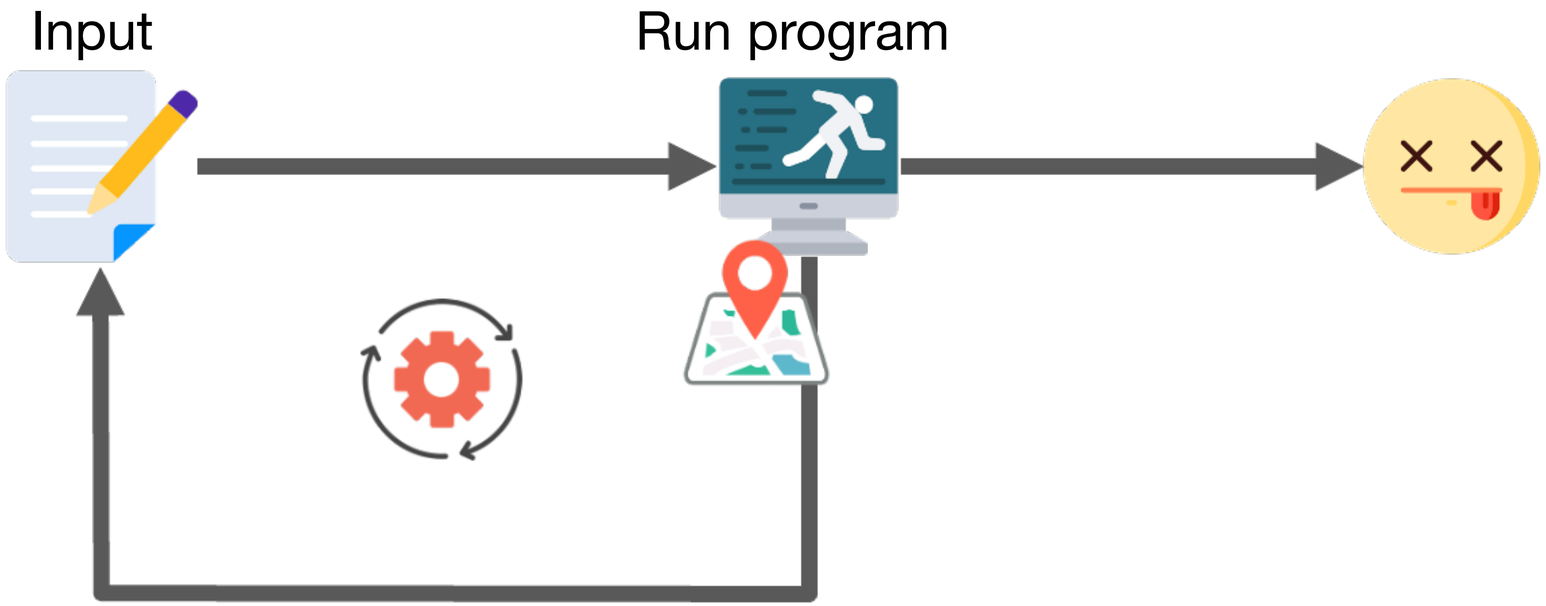
Origin of Fuzz Testing

- A night in 1988 with thunderstorm and heavy rain
- Connected to his office Unix system via a dial-up connection
- The heavy rain introduced noise on the line
- Crashed many UNIX utilities he had been using everyday
- He realized that there was something deeper
- Asked three groups in his advanced OS course to implement this idea of fuzz testing
  - Two groups failed to achieve any crash results!
  - The third group succeeded! Crashed 25-33% of the utility programs on the seven Unix variants that they tested

Fuzz Testing

Run programs on many random, abnormal inputs and look for bad behaviors in the responses.

- Bad behaviors such as crashes or hangs



Fuzz Testing

- Approach
  - Generate random inputs
  - Run lots of programs using random inputs
  - Identify crashes of these programs
  - Correlate random inputs with crashes
- Errors found:
  - Not checking returns
  - Array index out of bounds
  - not checking null pointers
  - …

Why does Fuzzing Matter?

- Fuzzing finds *reachable* bugs effectively.
- Two orthogonal use cases:
  - Proactive defense, part of testing
  - Preparing offense, part of exploit development

Example

format.c (line 276):

```
...
while (lastc != '\n') {
  rdc();
}
...
```

input.c (line 27):

```
rdc()
{ do { readchar(); }
  while (lastc == ' ' || lastc == '\t');
  return (lastc);
}
```

- Reading from a file and set each char to `lastc`
- If `lastc` is a white space or tab, keep reading.
- At the end of file, set `lastc` to null

Fuzz Testing Overview

- Black-box fuzzing
  - Treating the system as a black box during fuzzing, i.e., not knowing details of the implementation
- White-box fuzzing
  - Designing input generation with full knowledge of the target software
- Grey-box fuzzing
  - Having partial knowledge of the internals of the target

Black-box Fuzzing

- Like Miller, feed the program random inputs and see if it crashes.
- Pros: Easy to configure
- Cons: may not search efficiently
  - May re-run the same paths over again (low coverage)
  - May be very hard to generate inputs for certain paths (checksums, hashes, format checks, restrictive conditions)

Black-box Fuzzing

- Example that would be hard for black-box fuzzing to find the error

```
function(char *name, char *passwd, char *buf) {
    if (authenticate_user(name, passwd)) {
        if (check_format(buf)) {
            update(buf); // bugs in here
        }
    }
}
```

Mutation-based Fuzzing

- User supplies a well-formed input.
- Fuzzing: Generate random changes to that input, i.e., *mutating* the input
- Seed inputs: A set of initial inputs
- Mutations: bit flipping, truncation, duplications, byte changes, etc.
- No assumption about input
  - Only assumes that variants of well-formed input may be problematic for the program
- Example: zzuf
  - https://github.com/samhocevar/zzuf

Mutation-based Fuzzing

- Example of using zzuf
  - `zzuf -s 0:1000000 -c -C 0 -q -T 3 objdump -x win9x.exe`
    - Fuzz the program objdump using the sample input win9x.exe
    - `-s`: seed from 1 to 1,000,000, used to change bits of input
    - `-c`: fuzz files whose name is specified in the target application's command line
    - `-C 0`: Keep running when crashes detected
    - `-q`: quiet mode
    - `-T 3`: Each run limited to 3 seconds

Mutation-based Fuzzing

- Easy to set up, and not dependent on program details
- But may be strongly biased by the initial input
- Still prone to some problems, e.g., re-running the same paths over again

Generation-based Fuzzing

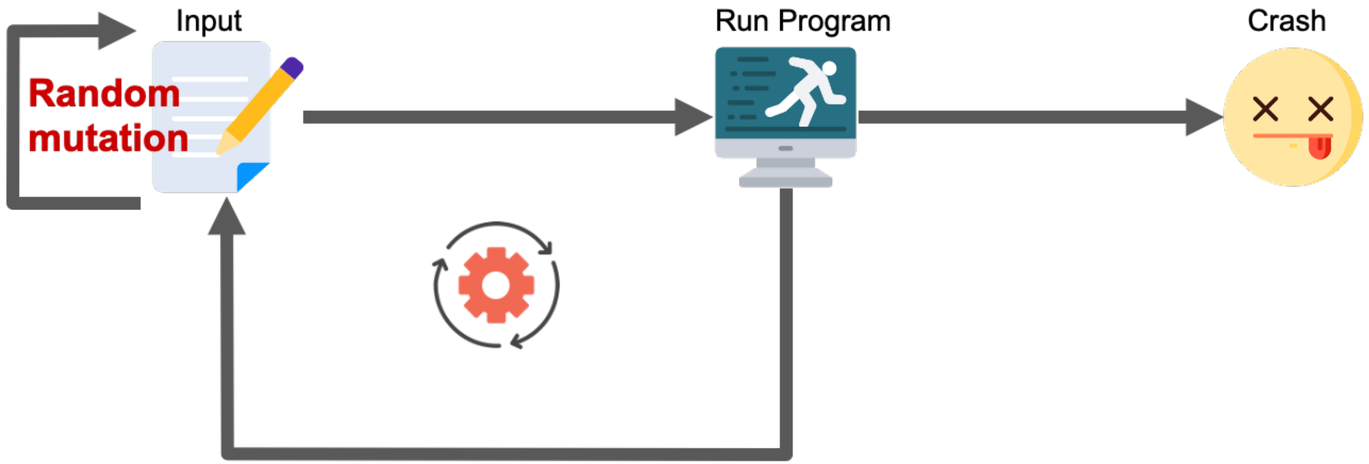
- Generate inputs from scratch according to predefined rules/specifications
- Generated inputs are well-formed, adhering to the specs
- Can write a generator to generate well-formatted inputs
- Suitable for inputs with a specific format requirement
  - e.g., JSON/XML files, network traffic of certain protocols

Generation-based Fuzzing

- Can be more accurate, but at a cost
- Pros: More complete search
  - Avoiding wasting time rejecting ill-formatted inputs
  - Values are more specific to the program operations
  - Can account for dependencies within inputs
- Cons
  - May miss bugs triggered by ill-formatted inputs
  - Writing good specifications is not easy.
  - Need to specify a format for each program, i.e., program specific

Coverage-based/guided Fuzzing

- Rather than treating the program as a black box, instrument the program to track coverage
  - E.g., the coverage of statements/edges/paths



Why is Test Coverage Important?

- Test quality is determined by the coverage of the program by the test set so far.
- Benefits:
  - Can be used as a **stopping rule**: e.g., stop testing if 95% of elements have been covered.
  - Can be used as a **metric**: a test set that has a test coverage of 80% is better than one that covers 70%
  - Can be used in a **test case generator**: look for a test which exercises new elements not covered by the tests so far

Coverage-based Fuzzing

- Rather than treating the program as a black box, instrument the program to track coverage
  - E.g., the coverage of statements/edges/paths
- Uses feedback from the program's execution to guide new input generation
- Also called grey-box fuzzing
- Maintain a pool of high-quality tests
  1. Start with some initial ones (seeds) specified by users
  2. Run tests and record the code coverage
  3. Mutate tests in the pool to generate new tests
  4. Run new tests
  5. If a new test leads to new coverage (e.g., edges), save the new test to the pool; otherwise, discard the new test

gcov's Code Coverage Report

```
    -:    0:Source:demo.c
    -:    0:Graph:demo-demo.gcno
    -:    0:Data:demo-demo.gcda
    -:    0:Runs:1
    -:    0:Programs:1
    -:    1:#include <stdio.h>
    -:    2:
    1:    3:int main (void) {
    -:    4:  int i;
   10:    5:  for (i = 1; i < 10; i++) {
    9:    6:      if (i % 3 == 0)
    3:    7:        printf ("%d is divisible by 3\n", i);
    9:    8:      if (i % 11 == 0)
#####:    9:        printf ("%d is divisible by 11\n", i);
    9:   10:    }
    1:   11:  return 0;
    -:   12:}
    -:   13:
```

Example of Coverage-based Fuzzer: AFL

- American Fuzzy Lop

AFL

- Mutation-based, coverage-guided, grey-box fuzzer
- The original version is no longer maintained; `afl++` is the newer version.

AFL Learning Tutorials and Documents

- https://github.com/mykter/afl-training
- https://volatileminds.net/blog/
- https://afl-1.readthedocs.io/en/latest/user_guide.html
- https://lcamtuf.coredump.cx/afl/

AFL Build

- Provides compiler wrappers for gcc/clang to instrument target programs to track fuzzing (testing) coverage
- Replace your C compiler in your build process with afl-gcc/clang, then build your target program with afl-gcc/clang
  - Generates a binary instrumented for AFL fuzzing

Test Coverage Tool: `llvm-cov/gcov`

- llvm-cov/gcov: Emit code coverage information
  - Insert additional code through a compiler to track line coverage and branch coverage
- Bundled with clang/gcc
  - e.g., `clang/gcc –coverage demo.c -o demo`
    - Generated binary contains instrumentation code for code coverage.
    - Many other interesting compiler options. See “clang/gcc --help” & “man gcov”

Setting Up the Fuzzing

- Compiling through AFL
  - Basically, replace gcc/clang with afl-gcc
  - `path-to-afl/afl-gcc test.c -o test`
- Fuzzing through AFL
  - `path-to-afl/afl-fuzz -i testcase -o output ./test @@`
  - Assuming test cases are under the `testcase` directory.
  - Output goes to the output directory.
  - @@ tells AFL to take the file names under `testcase` and feed them to test.

Setting Up the Environment

- After you install AFL but before you can use it effectively, you need to set the following environment variables

```
export AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1
export AFL_SKIP_CPUFREQ=1
```

- The former speeds up response from crashes.
- The latter suppresses AFL complaint about missing some short-lived processes.

AFL Mutation Strategies

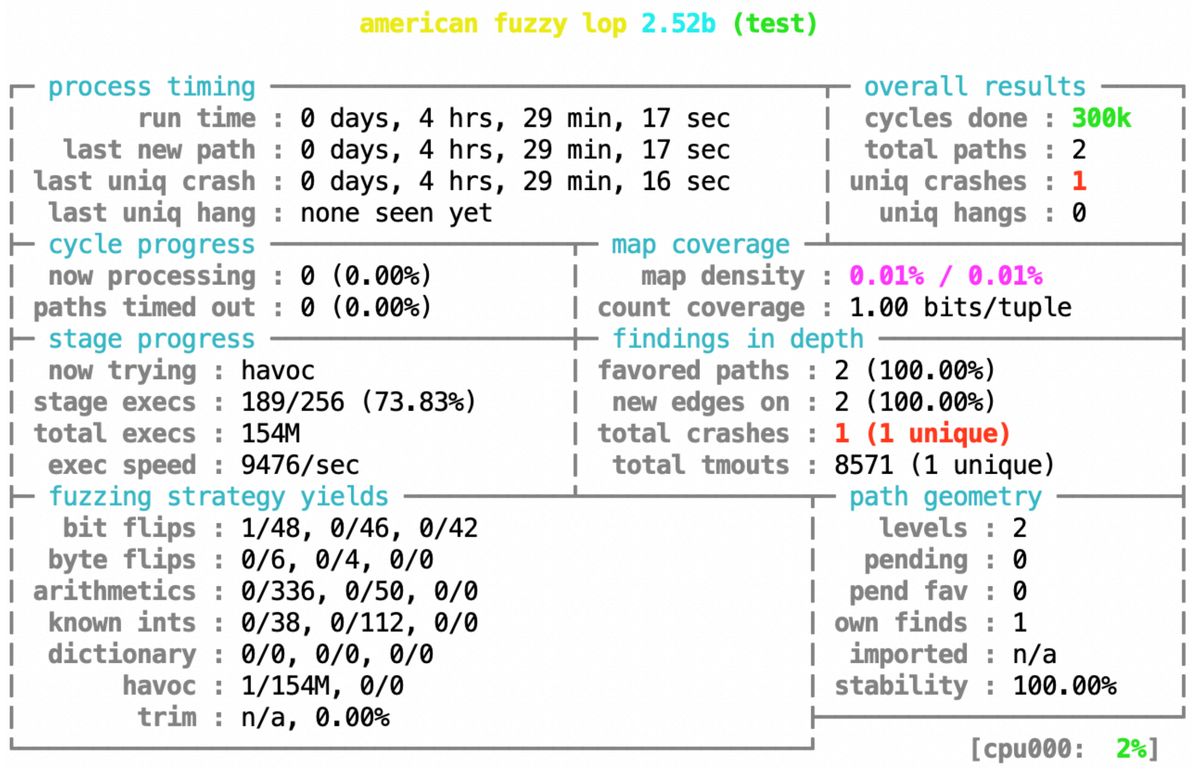
- Highly deterministic at first
  - bit flips
  - adding/subtracting integer values
  - Overwriting parts of the input with “interesting values” (e.g., INT_MAX)
  - Replacing parts of the input with predefined or auto-detected values
- Then, non-deterministic choices
  - insertion/deletion bytes
  - Overwriting with random values
  - Others

Example of Using AFL

```
int main(int argc, char* argv[]) {
  ... // Some error checking code

  FILE *fp = fopen(argv[1],"r");

  ... // Some error checking code

  size_t len;
  // Asking getline to malloc by setting *line be null
  char *line=NULL;
  if (getline(&line, &len, fp) < 0) {
    printf("Fail to read the file; exiting...\n");
    exit(-1);
  }

  // Convert the input  into a long integer
  long pos = strtol(line, NULL, 10);
  … // Some error checking code

  if (pos > 100) {
    if (pos < 150) {
      abort(); // Indicates abnormal termination
    }
  }

  fclose(fp);
  free(line);

  return 0;
}
```

AFL Display

- Track the execution of the fuzzer
- Input is a file containing number 55 for the previous toy program.



Google “afl display explained” for more detailed explanations.

AFL Output

- Files generated in the output directory
- File “fuzzer_stats” provides summary of stats
- File “plot_data” shows the progress of fuzzer.
- Directory “queue” shows inputs that led to paths.
- Directory “crashes” contains the input that caused crashes.
- Directory “hangs” contains input that caused hang.

Example of fuzzer_stats

- Input is a file containing number 55 for the previous top program.

```
$ cat output-55-only/fuzzer_stats
start_time        : 1728744413
last_update       : 1728760824
fuzzer_pid        : 2818949
cycles_done       : 305707
execs_done        : 156524727
execs_per_sec     : 9526.00
paths_total       : 2
paths_favored     : 2
paths_found       : 1
paths_imported    : 0
max_depth         : 2
cur_path          : 1
pending_favs      : 0
pending_total     : 0
variable_paths    : 0
stability         : 100.00%
bitmap_cvg        : 0.01%
unique_crashes    : 1
unique_hangs      : 0
last_path         : 1728744413
last_crash        : 1728744414
last_hang         : 0
execs_since_crash : 156517779
exec_timeout      : 20
afl_banner        : test
afl_version       : 2.52b
target_mode       : default
command_line      : ./afl-2.52b/afl-fuzz -i testcase -o output ./test @@
```

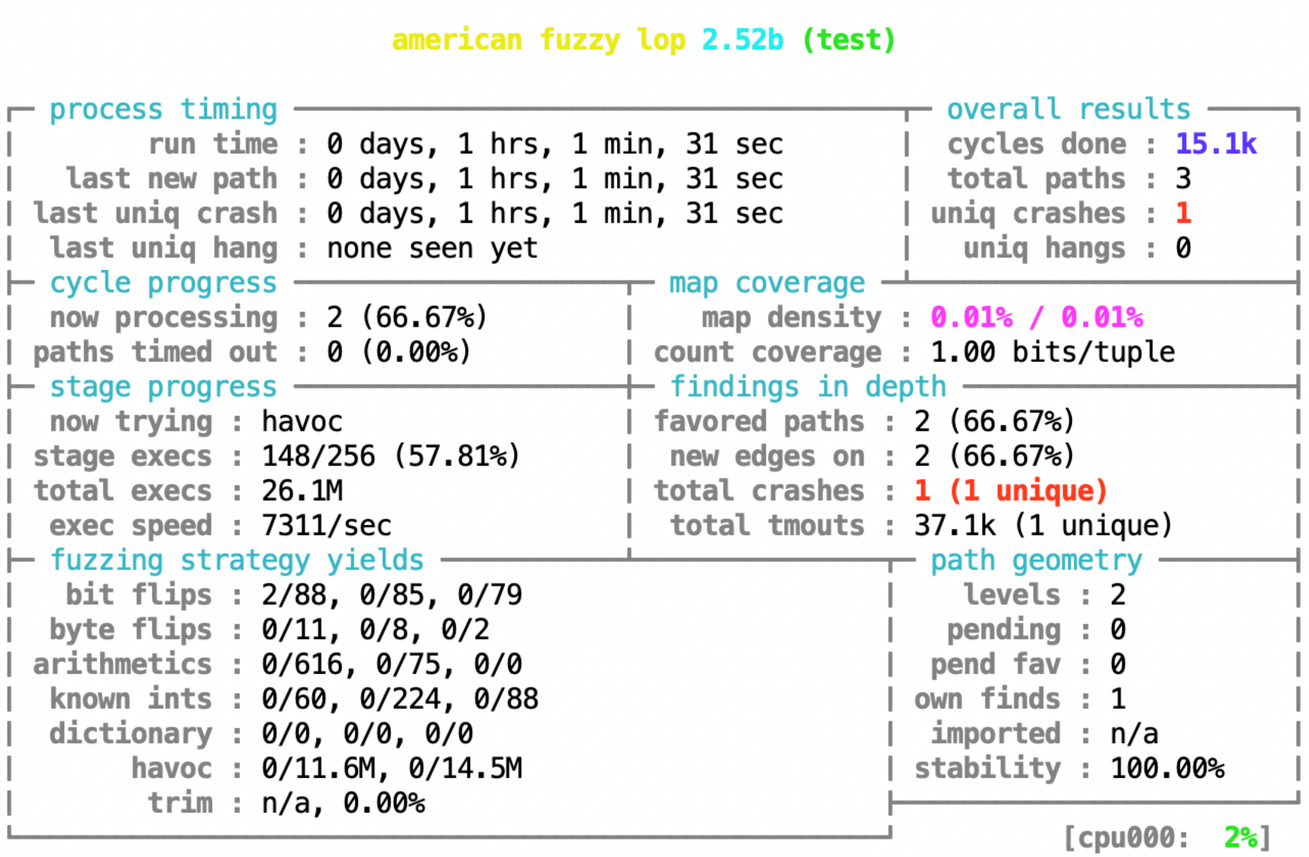
Initial Test Caes Are Important for Fuzzing Speed

```
int main(int argc, char* argv[]) {
  ... // Some error checking code

  FILE *fp = fopen(argv[1],"r");

  ... // Some error checking code

  size_t len;
  // Asking getline to malloc by setting *line be null
  char *line=NULL;
  if (getline(&line, &len, fp) < 0) {
    printf("Fail to read the file; exiting...\n");
    exit(-1);
  }

  // Convert the input  into a long integer
  long pos = strtol(line, NULL, 10);
  … // Some error checking code

  if (pos > 100) {
    if (pos < 150) {
      abort();
    }
  }

  fclose(fp);
  free(line);

  return 0;
}
```

- For the toy example,
  - If the only test case is 55, it takes longer to find a crash than if the test cases are 55 and 100
    - Since crashing tests are in [101, 149], the test is close to 100 syntactically.

AFL Display

- Track the execution of the fuzzer
- Inputs are files containing number 55 and 100 for the previous top program.



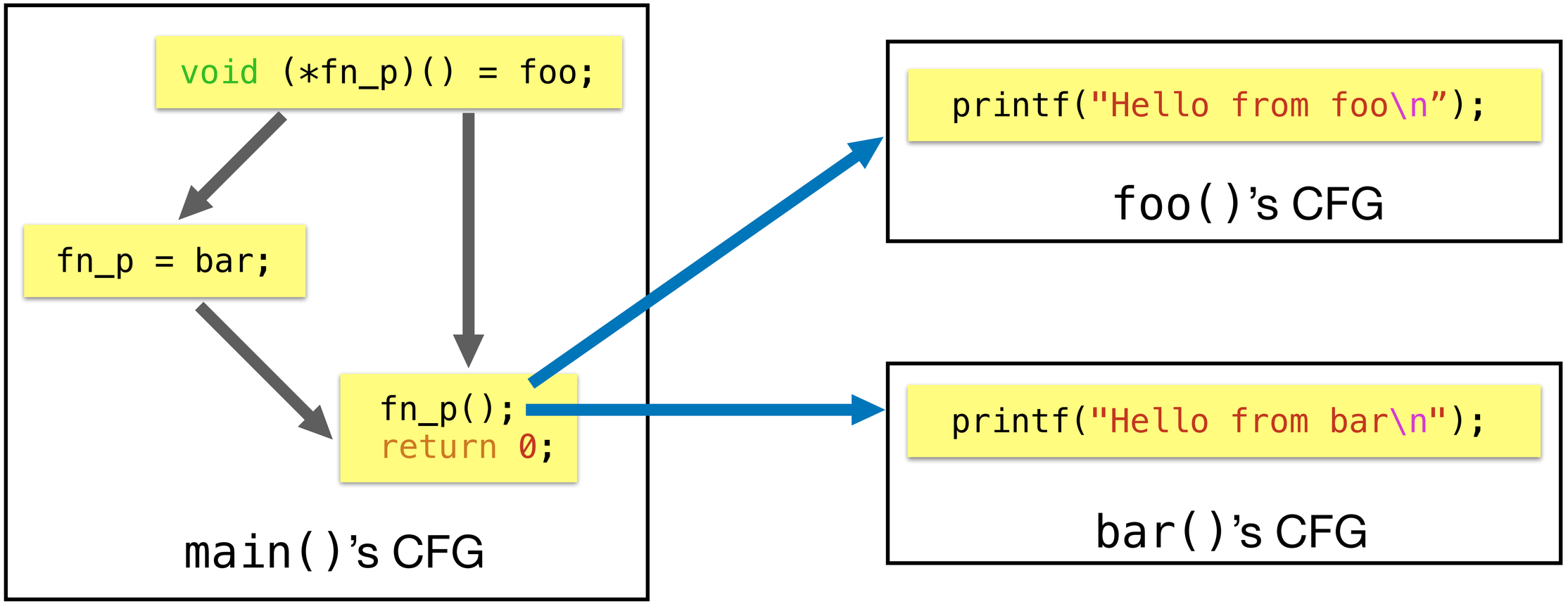
More AFL Documents

- How does AFL work?
  - http://lcamtuf.coredump.cx/afl/technical_details.txt
- AFL user guide
  - https://afl-1.readthedocs.io/en/latest/user_guide.html

AFL Coverage Measurements

- Branch coverage + coarse-grained branch-taken hit counts
  - Execution trace broken into (branch_src, branch_dest) pairs
    - "A->B->C->D" to (A, B), (B, C), (C, D)
  - A global map remembers whether a branch has been encountered and their hit counts
  - Coarse-grained branch hit counts: 8 hit-count buckets
    - 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+
- An input is considered interesting only if
  - It covers a new branch, or
  - It covers a new hit count bucket of a branch

Example of Control-flow Graph (CFG)



AFL Coverage Measurements

- Branch coverage + coarse-grained branch-taken hit counts
  - Execution trace broken into (branch_src, branch_dest) pairs
    - "A->B->C->D" to (A, B), (B, C), (C, D)
  - A global map remembers whether a branch has been encountered and their hit counts
  - Coarse-grained branch hit counts: 8 hit-count buckets
    - 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+
- An input is considered interesting only if
  - It covers a new branch, or
  - It covers a new hit count bucket of a branch

Grey-box Fuzzing

- Finds flaws, but still does not understand the program
- Pros: Much more effective than black-box fuzzing
  - Essentially no configurations
  - Lots of crashes have been identified
- Cons: Still a bit of a stab in the dark
  - Searches for inputs independently from the program
  - May not be able to execute some paths
- Need to improve the effectiveness further

Takeaway

- Fuzz testing aims to achieve good program coverage with little effort for the programmer.
- Challenge is to generate good inputs.
- AFL (grey-box) is now commonly used.

A4: Fuzzing and Fixing Programs

- Use AFL to fuzz a simple program to find inputs that trigger crashes/hangs
- Use those inputs to locate errors in the program and provide fixes
- Read the manual/documents!