PennState

CSE 597: Security of Emerging Technologies

Module: Testing and Fuzzing

Prof. Syed Rafiul Hussain

Systems and Network Security (SyNSec) Research Group

Department of Computer Science and Engineering

The Pennsylvania State University

Our Goal

- Develop techniques to detect errors/vulnerabilities automatically before they can cause damage
  - How to find them?

Security Analysis Techniques

PennState

- Testing/Fuzzing (Dynamic Analysis)
- Symbolic Execution
- Concolic Execution
- Static Analysis
- Formal Verification

Automatic test case generation → Static analysis → Program verification

Fuzzing → Dynamic symbolic execution

Lower coverage
Lower false positives
Higher false negatives

Higher coverage
Higher false positives
Lower false negatives

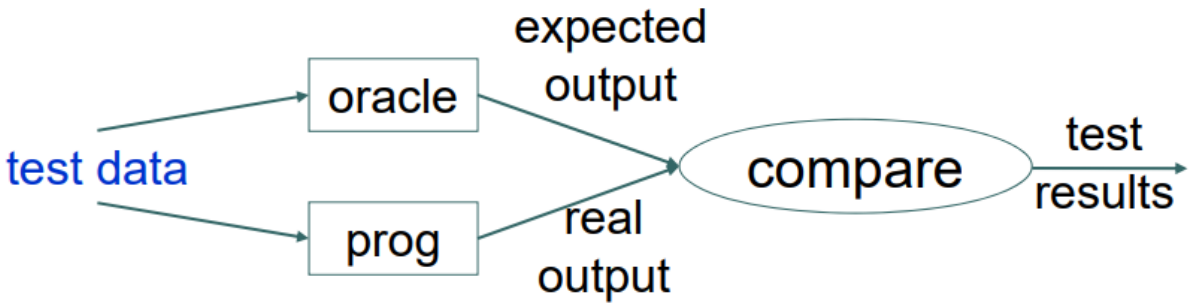
Dynamic Analysis

- Analyze the program when it is running with a specific input
- Many techniques
  - Testing, fuzzing, taint tracking, differential testing, execution integrity monitoring, …
  - We will discuss some of these in detail during lectures
  - And supplement the discussion with paper presentations
    - E.g., dynamic taint analysis, automated test generation, differential testing, …

Testing

PennState

- Testing: the process of running a program on a set of test cases and comparing the actual results with expected results (according to the specification).



- For the implementation of a factorial function, test cases could be {0, 1, 5, 10}. What is missing?
- Can it guarantee correctness?
  - Correctness: For all possible values of n, your factorial program will provide correct output.
  - Verification: High cost!

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
    - Suppose each test takes 1 second
    - Exhaustive testing would take ~31 years
- Question: How do you design the test set?
  - Black-box testing
  - White-box testing (or, glass-box)
  - Grey-box testing

Black-Box Testing

- Generating test cases based on specification alone
  - Without considering the implementation (internals)
- Advantage
  - Test cases are not biased toward an implementation
    - E.g., boundary conditions

Generating Black-Box Test Cases

- Example

```
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

```
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 3

```
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

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?
  - Now suppose 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 and x>z
    - x>y and x<=z
    - x<=y, and y>z
    - x<=y, and y<=z
  - A reasonable strategy then is to cover all four cases

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:

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

Test Coverage

- Test quality is determined by the coverage of the program by the test set so far
- Benefits
  - Can be used as a stopping rule: 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
    - The key idea behind graybox fuzzing and dynamic symbolic execution for automated test generation

Example Program

static void appendVector (Vector v1, Vector v2)

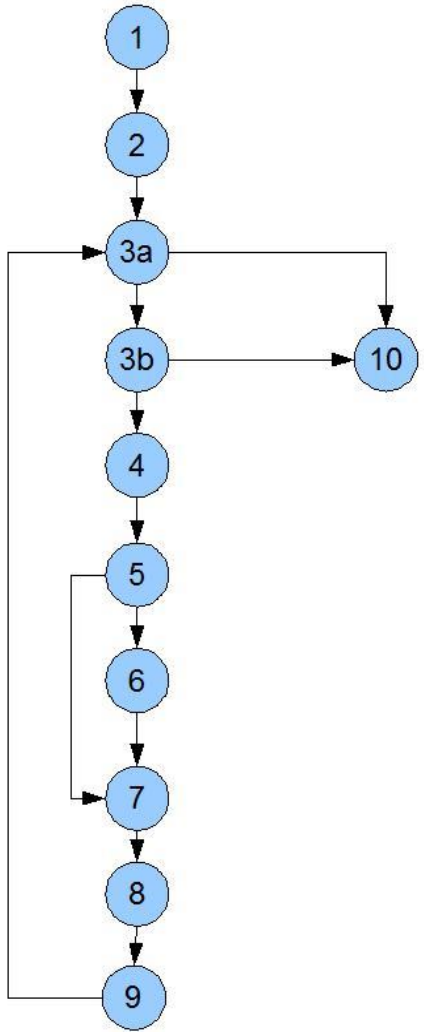
// Effects: If v1 or v2 is null throws NullPointerException else removes all elements of v2 and appends them in reverse order to the end of v1

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

A Running Example

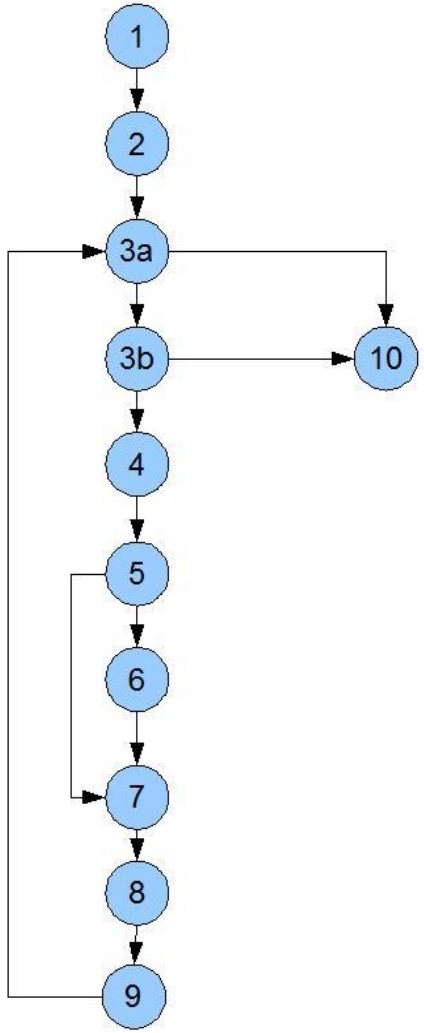
```
// Input: table is an array of numbers;
// Input: n is the size of table
// Input: element is the element to be found
// Output: found indicates whether the element
//         is in the table

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: }
```



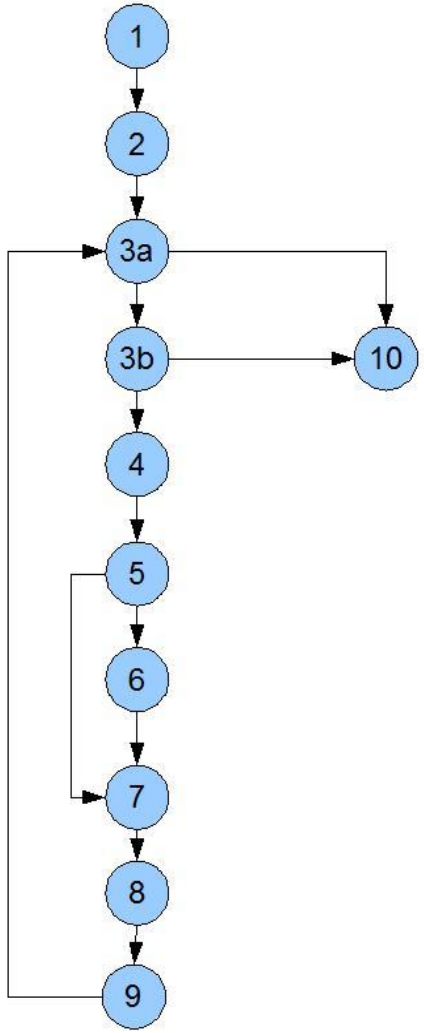
- Test data: table={3,4,5}; n=3; element=3
  - Does it cover all statements?
    - Yes
  - But does it cover all edges?
  - No, missing the edge from 3a to 10 and 5 to 7

Statement Coverage in Practice

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

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: }
```



- 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;
  if (y>z) return y; else return z; }
// Effects: Return the maximum value of x, y and z
```

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

Covering All Paths

- A program passes path-complete test data doesn't mean it's correct

```
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 # of Paths

- If there is a loop in the program, then there are possibly 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

```
void computeGPA(int grades[], int numCourses) {
 int total = 0; // Total should be reset for each student.
 for (int i = 0; student < 30; i++) {
  for (int j = 0; j < numCourses; j++) {
   total += grades[j];
  }
  float average = (float)total / numCourses;
  printf("Student %d: GPA = %.2f\n", i + 1, average); }
}
```

Exercise: Figuring Out a Test Suite that Covers zero, one, and two iterations of the loop

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

- Test data
  - 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

Combining 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

```
// Effects: If s is null throws NullPointerException,   else
  returns true iff s is a palindrome
boolean checkPalindrome (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;
}
```

Test Data for the Example

- Based on spec.
  - 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

Linux Test Coverage Tool: gcov

- Given a C/C++ program
  - Insert additional code through a compiler to track line coverage and branch coverage
  - Assumption: takes a complete program without input; i.e., the program already contains test cases
    - If not, add a test driver with test cases; e.g., via the google test framework
- Bundled within gcc
  - "gcc --coverage demo.c -o demo"

A Quick Demo

```
#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;
}
```

```
gcc --coverage demo.c -o demo
./demo
gcov demo.c
```

Generate demo.c.gcov

demo.c.gcov

```
        -:    0:Source:demo.c
        -:    0:Graph:demo.gcno
        -:    0:Data:demo.gcda
        -:    0:Runs: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);
        -:   10:    }
        1:   11:  return 0;
        -:   12:}
```

an extra tool lcov: can generate better visualization

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 based test generation for linked data structures**
  - Fuzzing can be viewed as a way of automated test generation
  - Symbolic execution can also be used for test generation

Pre- and Post-Conditions

- A pre-condition is a predicate
  - Assumed to hold before a function executes
- A post-condition is a predicate
  - Expected to hold after a function 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
```

Pre- and Post-Conditions

- Most useful if they are executable
  - Written in the programming language itself
  - A special case of assertions
- Need not be complete
  - Useful even if they do not cover every situation

Test Generation Using Pre- and postconditions

- A simple algorithm

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

Differential Testing

- Many specifications have multiple implementations
  - E.g., multiple crypto libraries: openssl, libgcrypt, LibreSSL, GNUTLS, …
  - E.g., multiple C/C++ compilers: gcc, clang, icc, …
  - E.g., multiple PDF readers: Adobe Acrobat, Foxit, Javelin, okular, …
- They are supposed to implement the same functionalities, but are they?

Differential Testing

- Idea: feed the same input to different implementations
  - If two implementations behave differently, we know one of them has a bug
- E.g., CSmith
  - Randomly generate syntactically valid C programs
  - Feed the same program to multiple C compilers
  - Check if the output binaries are behaviorally equivalent
  - Found hundreds of bugs in gcc and clang
- We will have a research paper presentation on this topic

Fuzz Testing

- Run program on many random, abnormal inputs and look for bad behavior in the responses
  - Bad behaviors such as crashes or hangs

Fuzz Testing (Bart Miller, U. Of Wisconsin)

- 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 grad-seminar 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

PennState

Fuzz Testing

- Idea proposed by Bart Miller at Wisconsin in 1988 after experiencing an unusual crash while accessing a Unix utility remotely

```
format.c (line 276):
... while (lastc != '\n') { //reading line
      rdc(); }

input.c (line 27):
   rdc() {
       do { //skipping space and tab
         readchar();
       } while (lastc == ' ' || lastc == '\t');
       return (lastc);
   }
```

Fuzz Testing

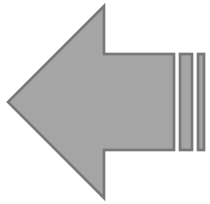
PennState

Fuzz Testing

- Idea proposed by Bart Miller at Wisconsin in 1988 after experiencing an unusual crash while accessing a Unix utility remotely

```
format.c (line 276):
... while (lastc != '\n') { //reading line
      rdc(); }

input.c (line 27):
   rdc() {
       do { //reading words
         readchar();
       } while (lastc == ' ' || lastc == '\t');
       return (lastc);
   }
```



When end of file, readchar() sets lastc to be 0; then the program hangs (infinite loop)

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 indices out of bounds, not checking null pointers, …

Fuzz Testing Overview

- Black-box fuzzing
  - Treating the system as a blackbox during fuzzing; not knowing details of the implementation
- Grey-box fuzzing
- White-box fuzzing
  - Design fuzzing based on internals of the system

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 path 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 ); // crash here
          }
     }
}
```

Mutation-Based Fuzzing

- User supplies a well-formed input
- Fuzzing: Generate random changes to that input
- No assumptions 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`
  - Try 1M random seed values (-s) from command line (-c) and keep running if crashed (-C 0) with timeout (-T 3)

Mutation-Based Fuzzing

- Easy to setup, and not dependent on program details
- But may be strongly biased by the initial input
- Still prone to some problems
  - May re-run the same path over again (same test)
  - May be very hard to generate inputs for certain paths (checksums, hashes, format checks, restrictive conditions)

Generation-Based Fuzzing

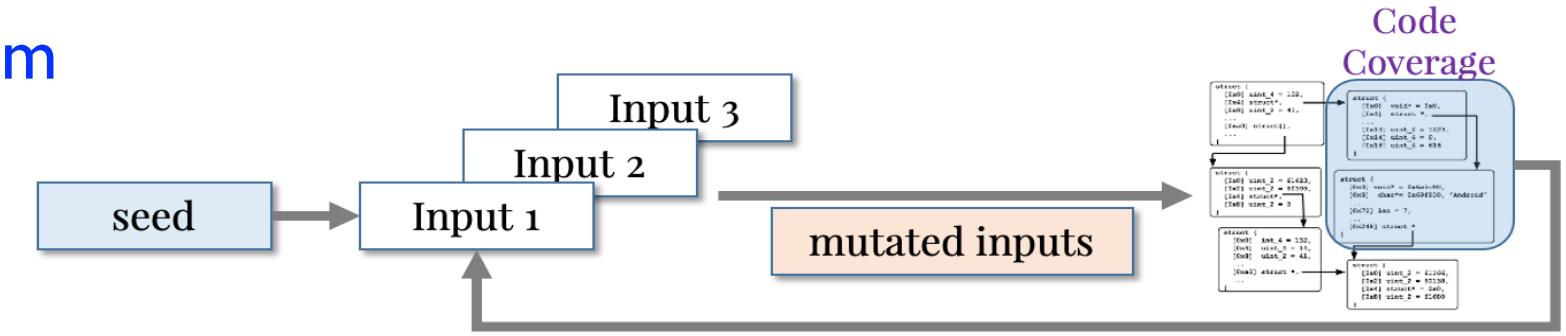
- Generate inputs "from scratch" rather than using an initial input and mutating
- However, require the user to specify a format or protocol spec to start
  - Alternatively, write a generator for generating well-formatted input
- Examples include
  - SPIKE, Peach Fuzz

Generation-Based Fuzzing

- Can be more accurate, but at a cost
- Pros: More complete search
  - Values more specific to the program operation
  - Can account for dependencies between inputs
- Cons: More work
  - Get the specification
  - Need to specify a format for each program or write the generator
    - program specific

Coverage-Based Fuzzing

- AKA grey-box fuzzing
- Rather than treating the program as a black box, instrument the program to track coverage
  - E.g., the edges covered



- Maintain a pool of high-quality tests
  1) Start with some initial ones specified by users
  2) Mutate tests in the pool to generate new tests
  3) Run new tests
  4) If a new test leads to new coverage (e.g., edges), save the new test to the pool; otherwise, discard the new test
  5) Any inputs that crash the program are recorded.

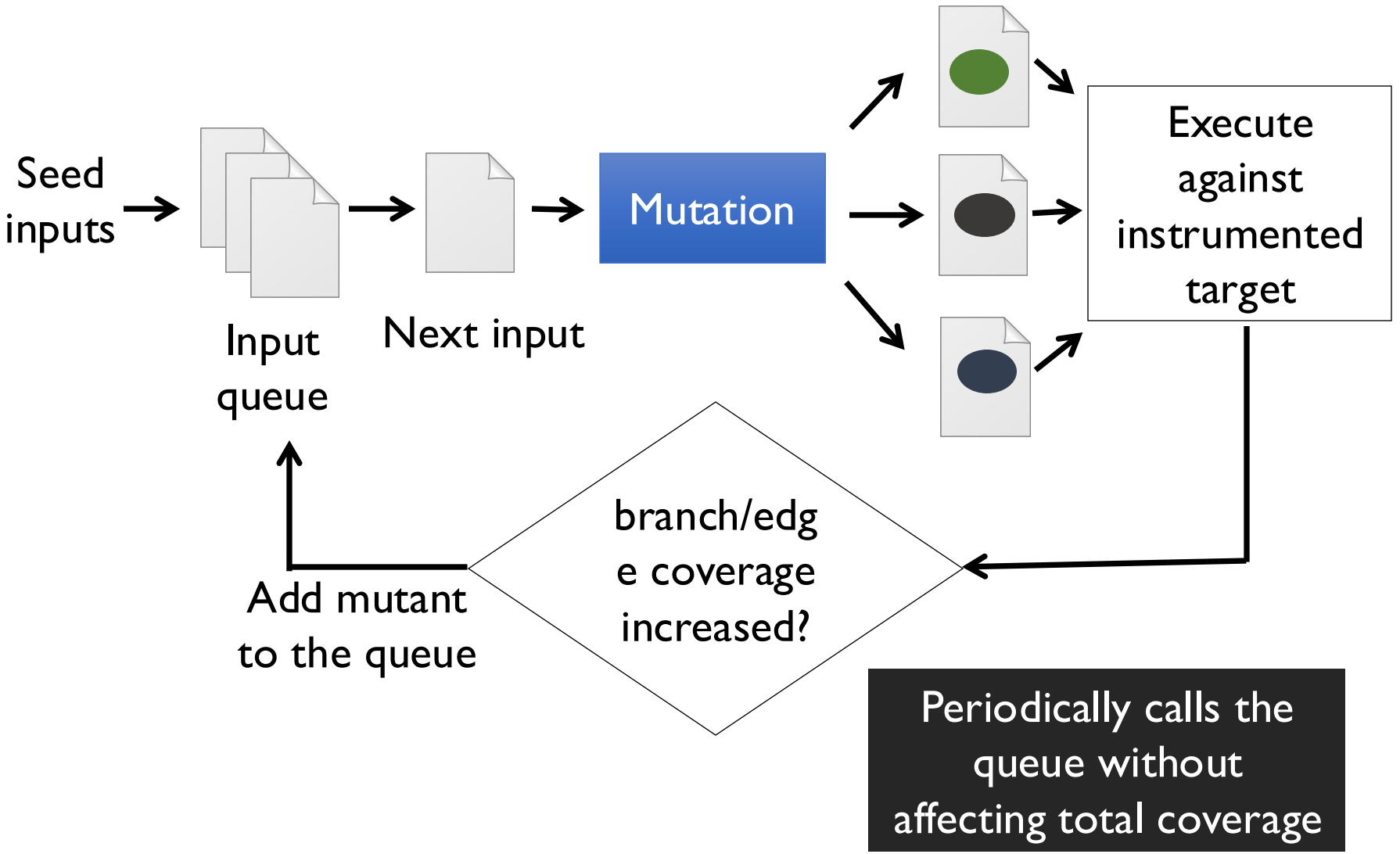
  6) Crashes are then sorted, reduced, and bugs are extracted. Bugs are then analyzed individually (is it a security vulnerability?).

AFL

- Example of coverage-based fuzzing
  - American Fuzzy Lop (AFL)
  - The original version is no longer maintained; afl++ is the newer version

American Fuzzy Lop (AFL)

PennState



AFL learning tutorials

- https://github.com/mykter/afl-training
- https://volatileminds.net/blog/

AFL Build

- Provides compiler wrappers for gcc to instrument target program to track test coverage
- Replace the gcc compiler in your build process with afl-gcc
- Then build your target program with afl-gcc
  - Generates a binary instrumented for AFL fuzzing

Toy Example of Using AFL

```
int main(int argc, char* argv[]) {
  …
  FILE *fp = fopen(argv[1],"r");  …
  size_t len;
  char *line=NULL;
  if (getline(&line,&len,fp) < 0) {
    printf("Fail to read the file; exiting...\n");
    exit(-1);
  }

  long pos = strtol(line,NULL,10); …

  if (pos > 100) {if (pos < 150) { abort();} }
  fclose(fp);  free(line);
  return 0;
}
```

* Omitted some error-checking code in "…"

Setting up the Fuzzing

- Compiling through AFL
  - Basically, replace gcc by 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 testcase, the output goes to the output dir
  - @@ tells AFL to take the file names under testcase and feed it to test

Setting up the environment

- After you install AFL but before you can use it effectively, you must set the following environment variables
  - E.g., On CentOS

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

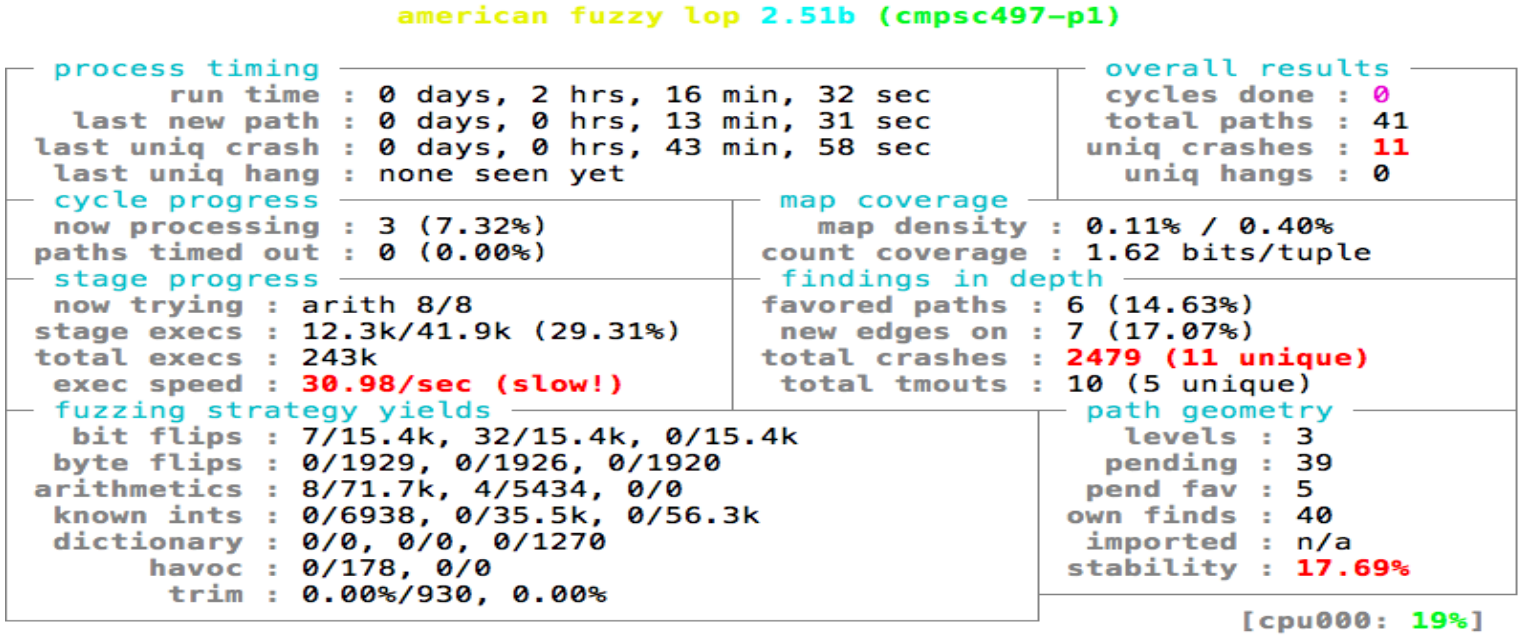
- The former speeds up response from crashes
- The latter suppresses AFL complaint about missing some short-lived processes

Initial Test Cases are Important for Fuzzing Speed

- For the toy example,
  - If the only test case is 55, it typically takes 3 to 15 mins to get a crashing input
  - If the test cases are 55 and 100, it typically takes only 1 min
    - Since crashing tests are in (100,150), the test is close to it syntactically; that's why the fuzzing speed is faster

AFL Display

- Tracks the execution of the fuzzer



- Key information are
  - "total paths" – number of different execution paths tried
  - "unique crashes" – number of unique crash locations

AFL Output

- Shows the results of the fuzzer
  - E.g., provides inputs that will cause the crash
- File "fuzzer_stats" provides summary of stats – UI
- File "plot_data" shows the progress of fuzzer
- Directory "queue" shows inputs that led to paths
- Directory "crashes" contains input that caused crash
- Directory "hangs" contains input that caused hang

AFL Operation

- How does AFL work?
  - http://lcamtuf.coredump.cx/afl/technical_details.txt

AFL Coverage Measurements

- Branch coverage + coarse-grained branch-taken hit counts
  - Execution trace broken into (branch_source, 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

AFL Mutation Strategies

- Highly deterministic at first – bit flips, add/sub integer values, and insert interesting integer values
- Then, non-deterministic choices – insertions, deletions, and combinations of test cases

Grey Box Fuzzing

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

White Box Fuzzing

- Combines test generation with fuzzing
  - Test generation based on static analysis and/or symbolic execution – more later
  - Rather than generating new inputs and hoping that they enable a new path to be executed, compute inputs that will execute a desired path
    - And use them as fuzzing inputs
- Goal: Given a sequential program with a set of input parameters, generate a set of inputs that maximizes code coverage

Take Away

- Fuzz testing aims to achieve good program coverage with little effort for the programmer
- Challenge is to generate the right inputs
- Black box (Mutational and generation), Grey box, and White box approaches are being investigated
  - AFL (Grey box) is now commonly used

To be organized

AFL Crashes

- May be caused by failed assertions – as they abort
  - Had several assertions caught as crashes, but format violated my checks

Penetration Testing (Pen Testing)

- Security-oriented testing
  - Typically performed on a whole IT system, not just a single program
- Good intentioned
  - Performed by white hackers
  - With the goal of reporting found vulnerabilities
  - Can be part of a security audit
- National Cyber Security Center definition:
- "A method for gaining assurance in the security of an IT system by attempting to breach some or all of that system's security, using the same tools and techniques as an adversary might."

Penetration Testing: Attack Phase

1. Reconnaissance
   - Gather information on the target system
   - E.g., gather publicly available information
2. Scanning
   - Use technical tools to further understand the system
   - Decide on the attack surface
   - E.g., use a port scanning tool to get open ports
3. Gaining Access
   - Use a payload to exploit the targeted system
   - E.g., use a tool such as Metasploit to exploit known vulnerabilities

Penetration Testing: Attack Phase

4. Maintaining Access

- Take steps to make threat persistent in the target system to gather as much data as possible
- E.g., install some monitoring software on the target
- Advanced Persistent Threat (APT)

5. Covering Tracks

- Clear traces of the attack

Penetration Testing: Analysis, Reporting and Clean up

- Consolidating the gathered information
- Perform analysis, draw conclusions, and make recommendations
  - What components in the system are vulnerable?
  - What mitigations are recommended?
    - New tools, new recommended processes, new personnel, etc.
- Deliver a report/presentation to the organization
- This can be followed by a clean-up phase
  - To restore the system to the original state

Program Verification

- A program takes some input and has some output
- Verification: an argument that a program works **on all possible inputs**
  - The argument can be either formal or informal and is usually based on the static code of the program
  - If so, we say a program is **correct**
  - E.g., given an implementation of a factorial function f, we argue in program verification for all n, f(n) = n!
- In general, the cost of program verification is high

Example Program For Verification

- How should we argue the following program computes the factorial of n?

```
int f (int n) {
  y := 1;
  z := 0;
  while (z != n) do {
    z := z + 1;
    y := y * z
  }
  return y;
}
```

Q: Actually, does the function work for all n?

Take Away

- Goal is to detect vulnerabilities in our programs before adversaries exploit them
- One approach is dynamic testing of the program
  - Fuzz testing aims to achieve good program coverage with little effort for the programmer
  - Challenge is to generate the right inputs
- Black box (Mutational and generation), Grey box, and White box approaches are being investigated
  - AFL (Grey box) is now commonly used
- What about the correctness of the program?

Thanks

PennState

Thanks to Omar Chowdhury, Gang Tan, Suman Jana and Baishakhi Ray for some slides.