Testing and Fuzzing

Gang Tan

Penn State University

Spring 2019

CMPSC 447, Software Security

* Some slides adapted from those by Trent Jaeger

Our Goal

- Develop techniques to detect vulnerabilities automatically before they are exploited
  - How to find them?
- Many techniques
  - Software testing
  - Fuzzing
  - Program analysis

Program Testing

- Testing: the process of running a program on a set of test cases and comparing the actual results with expected results
  - 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

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;
}
```

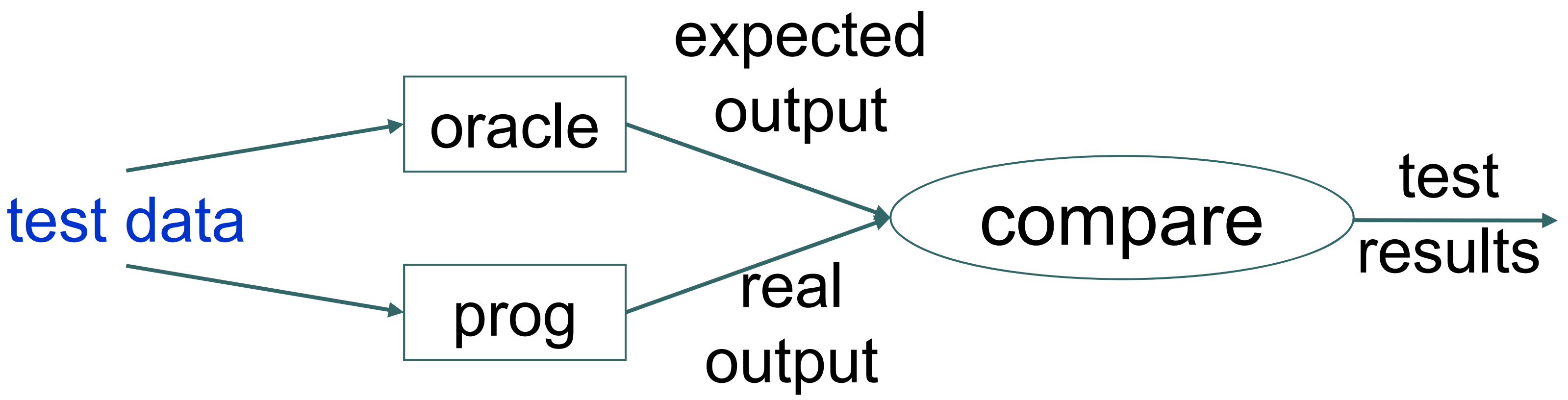
Q: Actually, does the function work for all n?

Software Testing

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

Bill Gates 1995

Testing Process



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)

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; 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 include 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 the empty vector
  - v2 is the empty vector
  - …
  - Another one is v1=v2
    - Aliases

White-Box Testing

- Looking into the internals of the program to figure out a set of sufficient 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 elements
  - 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

- Goodness 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 100% of elements have been tested
  - 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 as test case generator: look for a test which exercises some statements not covered by the tests so far
    - The idea behind AFL

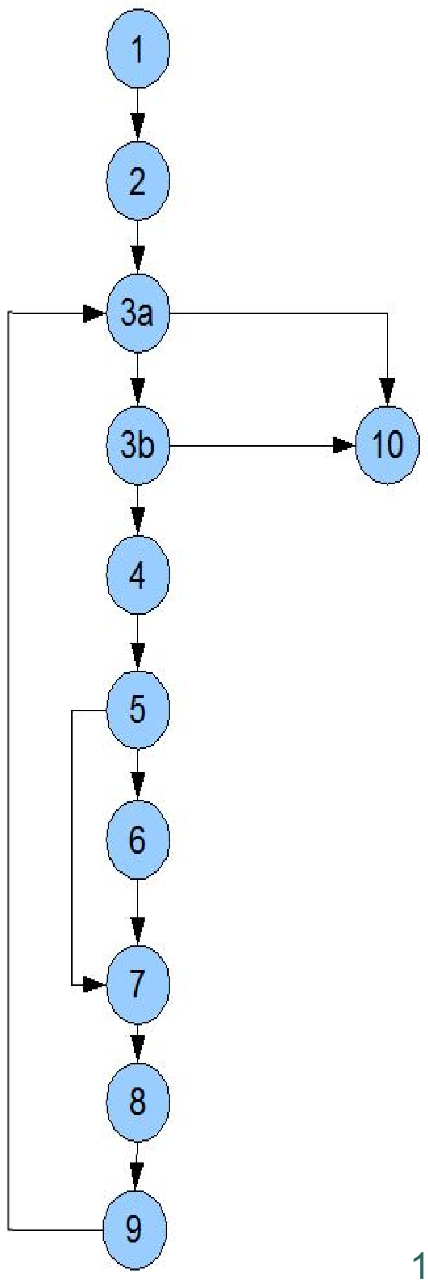
Different Coverage Criteria

- Usually based on control flow graphs (CFG)
  - Can have automated tool support
- Statement coverage
- Edge coverage
  - Edges in CFGs
- 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:
```



Covering Statements

```
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 application usually requires 100% statement coverage
  - Boeing requires 100% statement coverage

Edge Coverage

```
found = false;
counter = 0;
while ((counter < n) && (!found))
{
  if (table[counter] == element)
    found = true;

  counter++;
}
```

- 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;
}
```

  - Any non-empty test data is path-complete
- Same goes for the case of all-statement coverage, or all-edge coverage
  - In general, code coverage can't complain about missing cases

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

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

```
found = false;
counter = 0;
while ((counter < n) && (!found))
{
  if (table[counter] == element)
    found = true;

  counter++;
}
```

- Test data
  - Zero iteration: 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 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;
}
```

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

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

Testing Summary

- Design test cases
  - Black-box testing
    - Designing tests based on the specification of a program
  - White-box testing
    - Designing tests based on the implementation of a program
    - Statement, edge, and path coverage
- Penetration testing

Designing a Good Test Suite is Hard

- Not as clean as the examples
  - Figuring out a good test set is a major task
  - 100% coverage almost never achieved in practice
- One idea
  - Randomly generate test data (fuzzing)

Fuzzing

Fuzz Testing

- Run program on many random, abnormal inputs and look for bad behavior in the responses
  - Bad behaviors such as crashes or hangs
- What are the benefits of fuzz testing over regular testing?

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

- 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, …

Example Found

format.c (line 276):

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

input.c (line 27):

```
rdc()
{ do { 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 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, restrictive conditions)
  - May cause the program to terminate for logical reasons – fail format checks and stop

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
- Example: zzuf
  - http://sam.zoy.org/zzuf/
  - Reading: The Fuzzing Project Tutorial

Mutation-Based Fuzzing

- The Fuzzing Project Tutorial
  - zzuf -s 0:1000000 -c -C 0 -q -T 3 objdump -x win9x.exe
  - Fuzzes the program `objdump` using the sample input `win9x.exe`
  - Try 1M 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, restrictive conditions)

Generation-Based Fuzzing

- Generational fuzzer 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
  - Equivalently, write a generator for generating well-formated input
- Examples include
  - SPIKE, Peach Fuzz
- However format-aware fuzzing is cumbersome, because you'll need a fuzzer specification for every input format you are fuzzing

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
  - Write the generator – ad hoc
  - Need to do for each program

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
  - Start with some initial ones specified by users
  - Mutate tests in the pool to generate new tests
  - Run new tests
  - If a new test leads to new coverage (e.g., edges), save the new test to the pool; otherwise, discard the new test

AFL

- Example of coverage-based fuzzing
  - American Fuzzy Lop (AFL)
  - “State of the practice” at this time

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
- For example, in the Makefile for homework 3
  - `CC=path-to/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
```

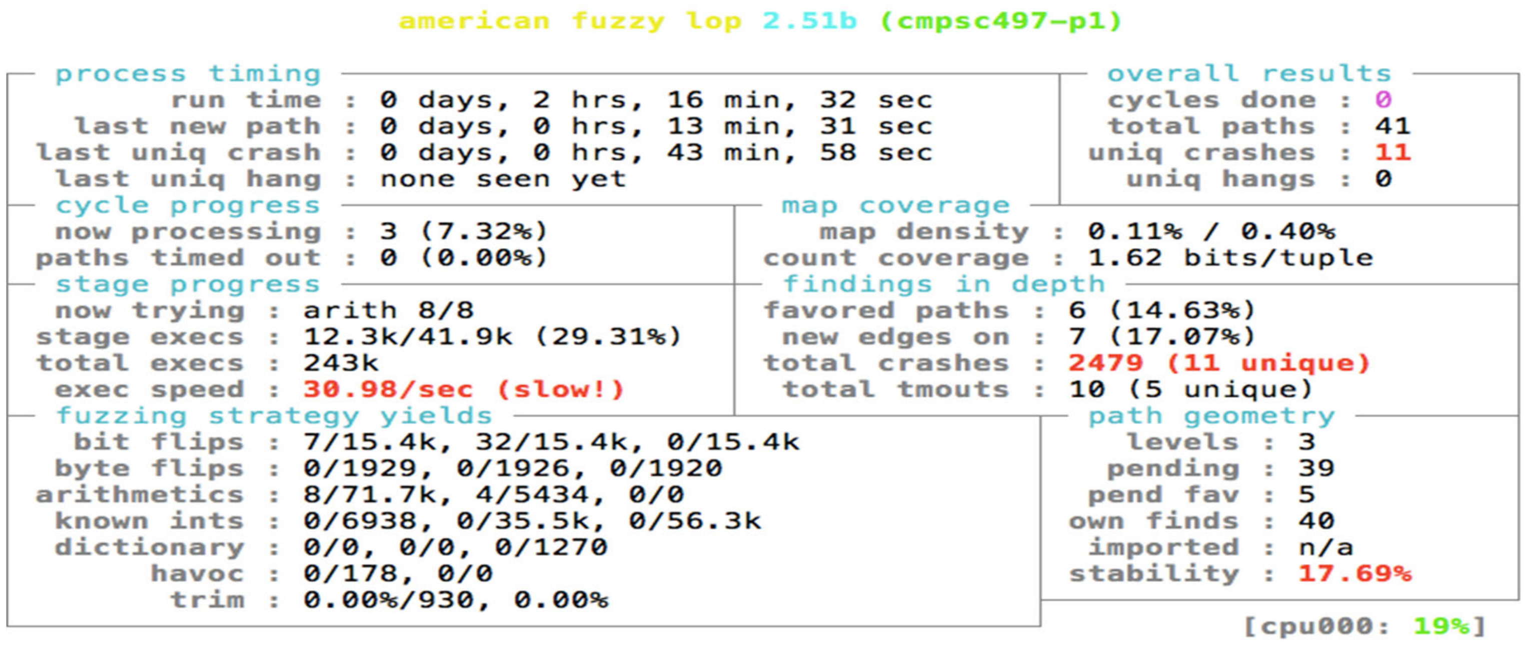
- The former speeds up response from crashes
- The latter suppresses AFL complaint about missing some short-lived processes

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 close is close to it syntactically; that’s why the fuzzing speed is faster

AFL Display

- Tracks the execution of the fuzzer



```
                 american fuzzy lop 2.51b (cmpsc497-p1)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 2 hrs, 16 min, 32 sec      │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 13 min, 31 sec      │  total paths : 41     │
│ last uniq crash : 0 days, 0 hrs, 43 min, 58 sec      │ uniq crashes : 11     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0      │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 3 (7.32%)         │    map density : 0.11% / 0.40%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 1.62 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 8/8             │ favored paths : 6 (14.63%)             │
│ stage execs : 12.3k/41.9k (29.31%)  │  new edges on : 7 (17.07%)             │
│ total execs : 243k                  │ total crashes : 2479 (11 unique)       │
│  exec speed : 30.98/sec (slow!)     │  total tmouts : 10 (5 unique)          │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 7/15.4k, 32/15.4k, 0/15.4k            │    levels : 3          │
│  byte flips : 0/1929, 0/1926, 0/1920                │   pending : 39         │
│ arithmetics : 8/71.7k, 4/5434, 0/0                  │  pend fav : 5          │
│  known ints : 0/6938, 0/35.5k, 0/56.3k              │ own finds : 40         │
│  dictionary : 0/0, 0/0, 0/1270                      │  imported : n/a        │
│       havoc : 0/178, 0/0                            │ stability : 17.69%     │
│        trim : 0.00%/930, 0.00%                      ├────────────────────────┘
└─────────────────────────────────────────────────────┘          [cpu000: 19%]
```

- 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 Crashes

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

AFL Operation

- How does AFL work?
  - http://lcamtuf.coredump.cx/afl/technical_details.txt
- Mutation strategies
  - Highly deterministic at first – bit flips, add/sub integer values, and choose 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 testing
  - 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

- 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