Fuzz Testing

Aaron Yang

Questions

Question 1 (10 pts): What is Fuzz Testing (basic idea) and its challenges? Which method does AFL use to solve the problems? (Hint: see its documentation)

- Fuzz testing is a software testing technique that involves sending a large number of random and invalid inputs or data to a program to detect vulnerabilities, bugs, and crashes. The basic idea behind fuzz testing is to identify unexpected and unhandled conditions, which can be exploited by attackers to compromise the security of the software. The main challenges of fuzz testing include generating inputs that are diverse, meaningful, and effective at triggering faults, as well as handling the large number of inputs and the crashes they may cause. AFL instruments the target program to measure code coverage, applying mutation operators to the initial input corpus, and using a feedback-driven genetic algorithm to guide the generation of new inputs.

Question 2 (10 pts): Briefly explain the overall algorithm of AFL approach. (Hint: see its documentation)

- The AFL approach involves instrumenting the target program to measure code coverage, using an initial set of inputs to generate
  executing the program, selecting the most promising inputs based on their fitness, and repeating the mutation and selection proc
  fuzzing process and uses a feedback-driven genetic algorithm to guide input generation towards achieving higher code coverag

Questions

Question 3 (10 pts): How does AFL measure the code coverage during the fuzzing process? List formula it uses for calculation. (Hint: see its technical details file)

- AFL measures the code coverage during the fuzzing process by instrumenting the target program with a code coverage feedback mechanism. AFL counts the number of basic blocks covered by each input and calculates the edge coverage and path coverage using the following formulas:
- Edge coverage: number of edges covered / total number of edges
- Path coverage: number of unique paths covered / total number of unique paths
- AFL also uses a feedback-driven genetic algorithm to guide the generation of new inputs towards achieving higher code coverage.

Question 4 (10 pts): What is fork server? Why does AFL use it during the fuzzing?

- A fork server is a mechanism used by AFL to improve the efficiency and stability of the fuzzing process. A fork server creates a child process of the target program that can be reused for multiple fuzzing iterations, instead of starting a new instance of the program for each input. This approach avoids the overhead of repeatedly loading and initializing the program, and it also isolates the target program from crashes and other errors caused by the inputs. The fork server can also detect crashes and other anomalies in the child process and report them to the AFL main process for analysis.

Extra Credit Question

One whitebox fuzzing approach that could be used to solve the problem is symbolic execution. On symbolic execution, the program is executed symbolically with input variables represented as symbols. The program path constraints are collected during the symbolic execution, which are then solved to generate inputs that can satisfy the constraints and explore new paths.

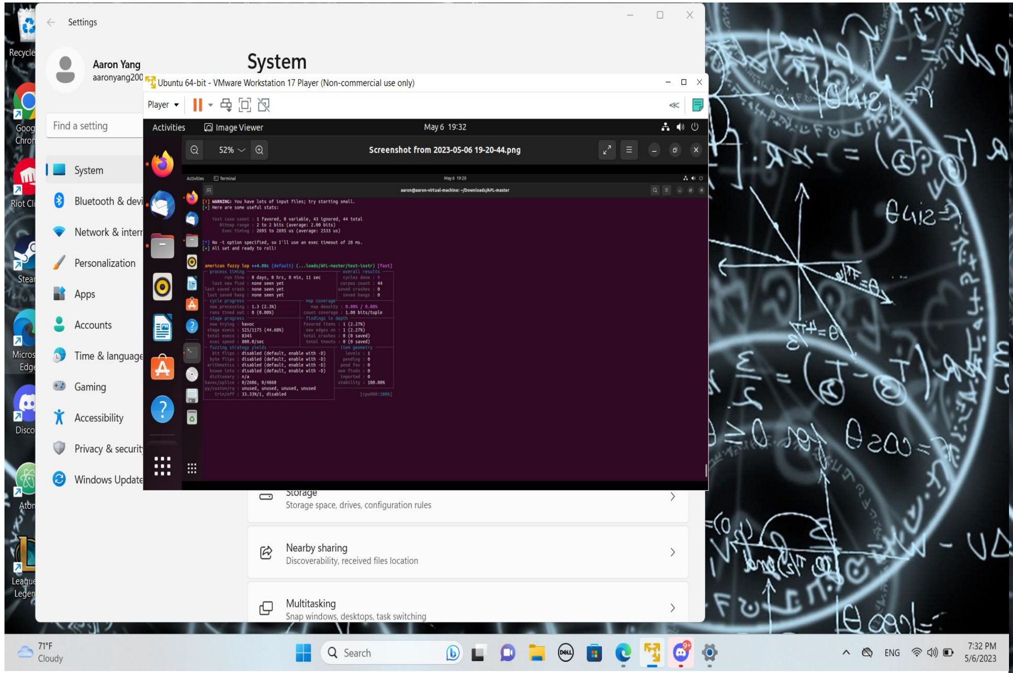
To apply symbolic execution to the given code example, the following steps could be taken:

1. Represent the input variable x symbolically.
2. Execute the program symbolically with x as input and collect path constraints during execution.
3. Use a constraint solver to generate inputs that satisfy the path constraints.
4. Execute the program with the generated input to explore new paths and repeat the above steps.

By using symbolic execution, the whitebox approach can generate a particular input that triggers the error.

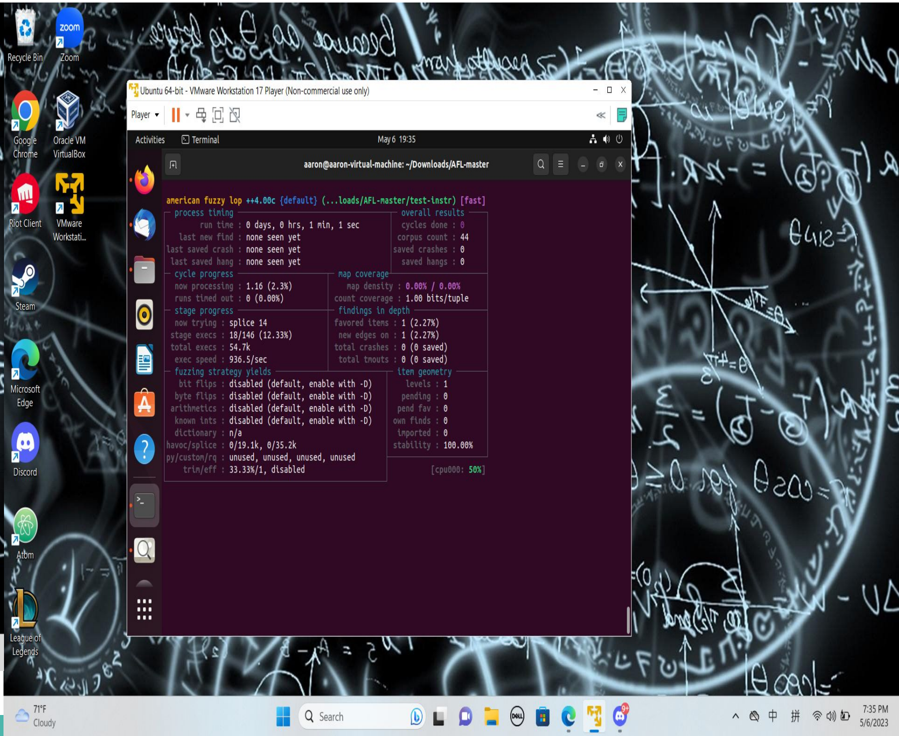
Screenshots

It is zero or not - 7:32



```
[!] WARNING: You have lots of input files; try starting small.
[+] Here are some useful stats:

    Test case count : 1 favored, 0 variable, 43 ignored, 44 total
       Bitmap range : 2 to 2 bits (average: 2.00 bits)
        Exec timing : 2695 to 2695 us (average: 2533 us)

[*] No -t option specified, so I'll use an exec timeout of 20 ms.
[+] All set and ready to roll!

 american fuzzy lop ++4.00c {default} (...loads/AFL-master/test-instr) [fast]
┌─ process timing ────────────────────────────────────┬─ overall results ──┐
│        run time : 0 days, 0 hrs, 0 min, 11 sec      │  cycles done : 0   │
│   last new find : none seen yet                     │ corpus count : 44  │
│last saved crash : none seen yet                     │saved crashes : 0   │
│ last saved hang : none seen yet                     │  saved hangs : 0   │
├─ cycle progress ─────────────────┬─ map coverage ───┴────────────────────┤
│  now processing : 1.3 (2.3%)     │    map density : 0.00% / 0.00%        │
│  runs timed out : 0 (0.00%)      │ count coverage : 1.00 bits/tuple      │
├─ stage progress ─────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc              │ favored items : 1 (2.27%)             │
│ stage execs : 525/1175 (44.68%)  │  new edges on : 1 (2.27%)             │
│ total execs : 8345               │ total crashes : 0 (0 saved)           │
│  exec speed : 800.0/sec          │  total tmouts : 0 (0 saved)           │
├─ fuzzing strategy yields ────────┴─────────────┬─ item geometry ─────────┤
│   bit flips : disabled (default, enable with -D)│    levels : 1          │
│  byte flips : disabled (default, enable with -D)│   pending : 0          │
│ arithmetics : disabled (default, enable with -D)│  pend fav : 0          │
│  known ints : disabled (default, enable with -D)│ own finds : 0          │
│  dictionary : n/a                               │  imported : 0          │
│havoc/splice : 0/2606, 0/4860                    │ stability : 100.00%    │
│py/custom/rq : unused, unused, unused, unused    ├────────────────────────┘
│    trim/eff : 33.33%/1, disabled                │          [cpu000:100%]
└─────────────────────────────────────────────────┘
```

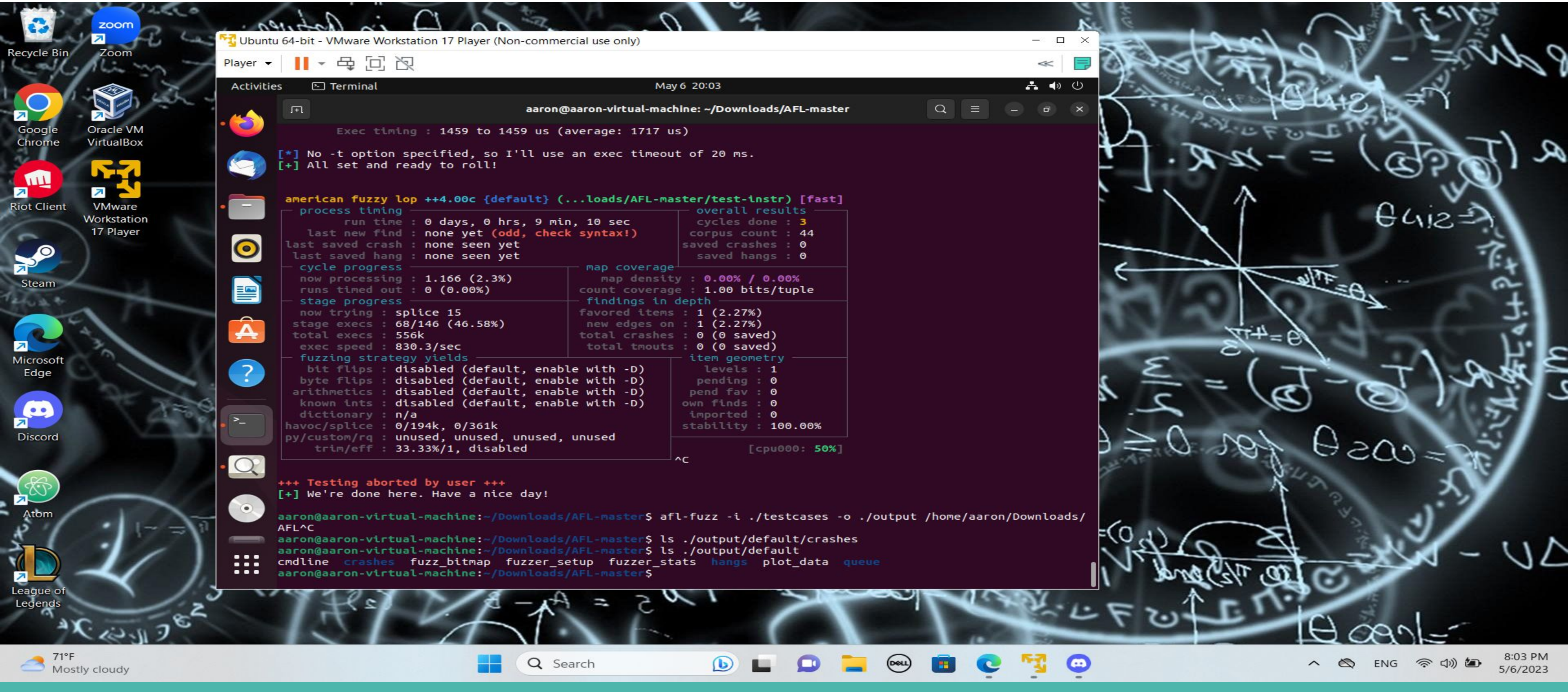
The screen - 7:35



```
 american fuzzy lop ++4.00c {default} (...loads/AFL-master/test-instr) [fast]
┌─ process timing ────────────────────────────────────┬─ overall results ──┐
│        run time : 0 days, 0 hrs, 1 min, 1 sec       │  cycles done : 0   │
│   last new find : none seen yet                     │ corpus count : 44  │
│last saved crash : none seen yet                     │saved crashes : 0   │
│ last saved hang : none seen yet                     │  saved hangs : 0   │
├─ cycle progress ─────────────────┬─ map coverage ───┴────────────────────┤
│  now processing : 1.16 (2.3%)    │    map density : 0.00% / 0.00%        │
│  runs timed out : 0 (0.00%)      │ count coverage : 1.00 bits/tuple      │
├─ stage progress ─────────────────┼─ findings in depth ───────────────────┤
│  now trying : splice 14          │ favored items : 1 (2.27%)             │
│ stage execs : 18/146 (12.33%)    │  new edges on : 1 (2.27%)             │
│ total execs : 54.7k              │ total crashes : 0 (0 saved)           │
│  exec speed : 936.5/sec          │  total tmouts : 0 (0 saved)           │
├─ fuzzing strategy yields ────────┴─────────────┬─ item geometry ─────────┤
│   bit flips : disabled (default, enable with -D)│    levels : 1          │
│  byte flips : disabled (default, enable with -D)│   pending : 0          │
│ arithmetics : disabled (default, enable with -D)│  pend fav : 0          │
│  known ints : disabled (default, enable with -D)│ own finds : 0          │
│  dictionary : n/a                               │  imported : 0          │
│havoc/splice : 0/19.1k, 0/35.2k                  │ stability : 100.00%    │
│py/custom/rq : unused, unused, unused, unused    ├────────────────────────┘
│    trim/eff : 33.33%/1, disabled                │          [cpu000: 50%]
└─────────────────────────────────────────────────┘
```

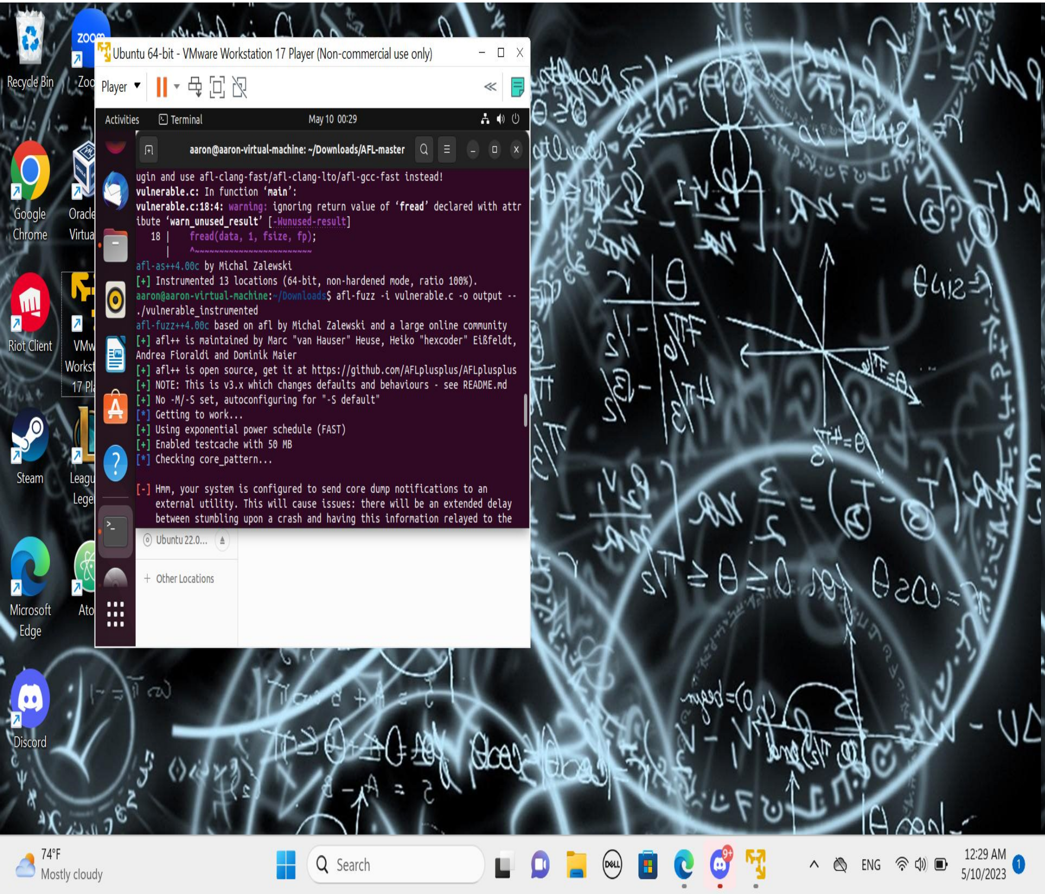
Screenshots

Indication of the crash file

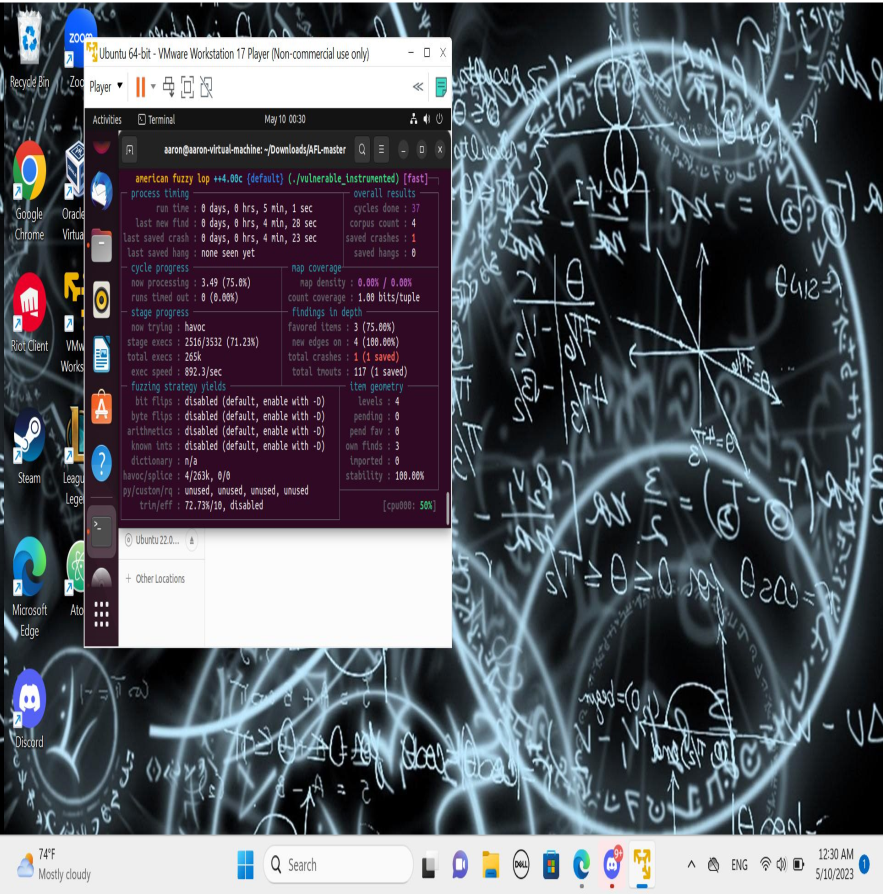


```
         Exec timing : 1459 to 1459 us (average: 1717 us)

[*] No -t option specified, so I'll use an exec timeout of 20 ms.
[+] All set and ready to roll!


 american fuzzy lop ++4.00c {default} (...loads/AFL-master/test-instr) [fast]
┌─ process timing ────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 9 min, 10 sec      │  cycles done : 3     │
│   last new find : none yet (odd, check syntax!)     │ corpus count : 44    │
│last saved crash : none seen yet                     │saved crashes : 0     │
│ last saved hang : none seen yet                     │  saved hangs : 0     │
├─ cycle progress ───────────────────┬─ map coverage─┴──────────────────────┤
│  now processing : 1.166 (2.3%)     │    map density : 0.00% / 0.00%       │
│  runs timed out : 0 (0.00%)        │ count coverage : 1.00 bits/tuple     │
├─ stage progress ───────────────────┼─ findings in depth ──────────────────┤
│  now trying : splice 15            │ favored items : 1 (2.27%)            │
│ stage execs : 68/146 (46.58%)      │  new edges on : 1 (2.27%)            │
│ total execs : 556k                 │ total crashes : 0 (0 saved)          │
│  exec speed : 830.3/sec            │  total tmouts : 0 (0 saved)          │
├─ fuzzing strategy yields ──────────┴─────────────┬─ item geometry ───────┤
│   bit flips : disabled (default, enable with -D) │    levels : 1         │
│  byte flips : disabled (default, enable with -D) │   pending : 0         │
│ arithmetics : disabled (default, enable with -D) │  pend fav : 0         │
│  known ints : disabled (default, enable with -D) │ own finds : 0         │
│  dictionary : n/a                                │  imported : 0         │
│havoc/splice : 0/194k, 0/361k                     │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused     ├───────────────────────┘
│    trim/eff : 33.33%/1, disabled                 │          [cpu000: 50%]
└──────────────────────────────────────────────────┘^C


+++ Testing aborted by user +++
[+] We're done here. Have a nice day!

aaron@aaron-virtual-machine:~/Downloads/AFL-master$ afl-fuzz -i ./testcases -o ./output /home/aaron/Downloads/
AFL^C
aaron@aaron-virtual-machine:~/Downloads/AFL-master$ ls ./output/default/crashes
aaron@aaron-virtual-machine:~/Downloads/AFL-master$ ls ./output/default
cmdline  crashes  fuzz_bitmap  fuzzer_setup  fuzzer_stats  hangs  plot_data  queue
aaron@aaron-virtual-machine:~/Downloads/AFL-master$
```

More Screenshots of Last Step

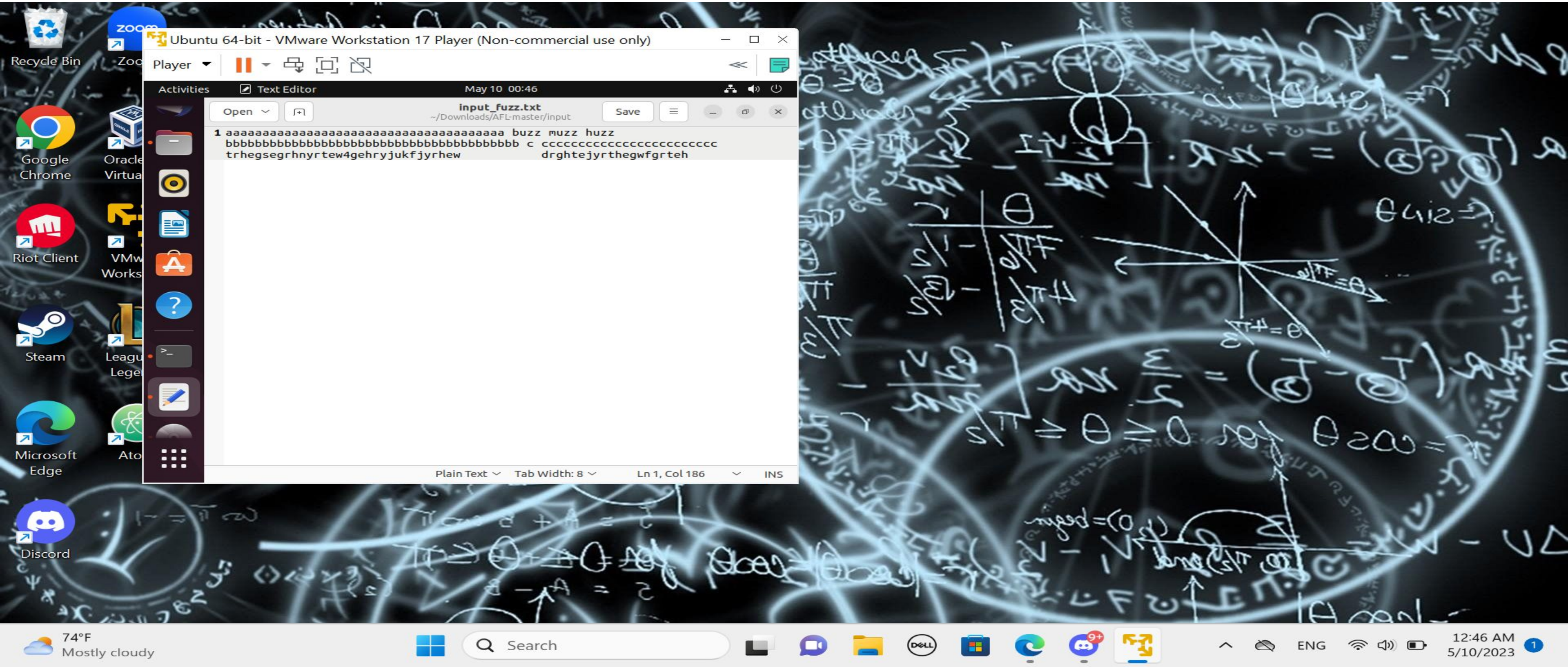

Ubuntu 64-bit - VMware Workstation 17 Player (Non-commercial use only)
Player
Activities
Terminal
May 10 00:29
aaron@aaron-virtual-machine: ~/Downloads/AFL-master
Ubuntu 22.0...
Other Locations
74°F
Mostly cloudy
Search
ENG
12:29 AM
5/10/2023

```
ugin and use afl-clang-fast/afl-clang-lto/afl-gcc-fast instead!
vulnerable.c: In function ‘main’:
vulnerable.c:18:4: warning: ignoring return value of ‘fread’ declared with attr
ibute ‘warn_unused_result’ [-Wunused-result]
   18 |    fread(data, 1, fsize, fp);
      |    ^~~~~~~~~~~~~~~~~~~~~~~~~
afl-as++4.00c by Michal Zalewski
[+] Instrumented 13 locations (64-bit, non-hardened mode, ratio 100%).
aaron@aaron-virtual-machine:~/Downloads$ afl-fuzz -i vulnerable.c -o output --
./vulnerable_instrumented
afl-fuzz++4.00c based on afl by Michal Zalewski and a large online community
[+] afl++ is maintained by Marc "van Hauser" Heuse, Heiko "hexcoder" Eißfeldt,
Andrea Fioraldi and Dominik Maier
[+] afl++ is open source, get it at https://github.com/AFLplusplus/AFLplusplus
[+] NOTE: This is v3.x which changes defaults and behaviours - see README.md
[+] No -M/-S set, autoconfiguring for "-S default"
[*] Getting to work...
[+] Using exponential power schedule (FAST)
[+] Enabled testcache with 50 MB
[*] Checking core_pattern...

[-] Hmm, your system is configured to send core dump notifications to an
    external utility. This will cause issues: there will be an extended delay
    between stumbling upon a crash and having this information relayed to the
```


Ubuntu 64-bit - VMware Workstation 17 Player (Non-commercial use only)
Player
Activities
Terminal
May 10 00:30
aaron@aaron-virtual-machine: ~/Downloads/AFL-master
Ubuntu 22.0...
Other Locations
74°F
Mostly cloudy
Search
ENG
12:30 AM
5/10/2023

```
   american fuzzy lop ++4.00c {default} (./vulnerable_instrumented) [fast]
- process timing ---------------------------------------- overall results -
        run time : 0 days, 0 hrs, 5 min, 1 sec          cycles done : 37
   last new find : 0 days, 0 hrs, 4 min, 28 sec        corpus count : 4
last saved crash : 0 days, 0 hrs, 4 min, 23 sec       saved crashes : 1
 last saved hang : none seen yet                        saved hangs : 0
- cycle progress ------------------ map coverage-----------------------------
  now processing : 3.49 (75.0%)          map density : 0.00% / 0.00%
  runs timed out : 0 (0.00%)          count coverage : 1.00 bits/tuple
- stage progress ------------------ findings in depth ----------------------
  now trying : havoc                  favored items : 3 (75.00%)
 stage execs : 2516/3532 (71.23%)      new edges on : 4 (100.00%)
 total execs : 265k                   total crashes : 1 (1 saved)
  exec speed : 892.3/sec               total tmouts : 117 (1 saved)
- fuzzing strategy yields ------------------------ item geometry ----------
   bit flips : disabled (default, enable with -D)      levels : 4
  byte flips : disabled (default, enable with -D)     pending : 0
 arithmetics : disabled (default, enable with -D)    pend fav : 0
  known ints : disabled (default, enable with -D)   own finds : 3
  dictionary : n/a                                   imported : 0
havoc/splice : 4/263k, 0/0                          stability : 100.00%
py/custom/rq : unused, unused, unused, unused
    trim/eff : 72.73%/10, disabled                     [cpu000: 50%]
```

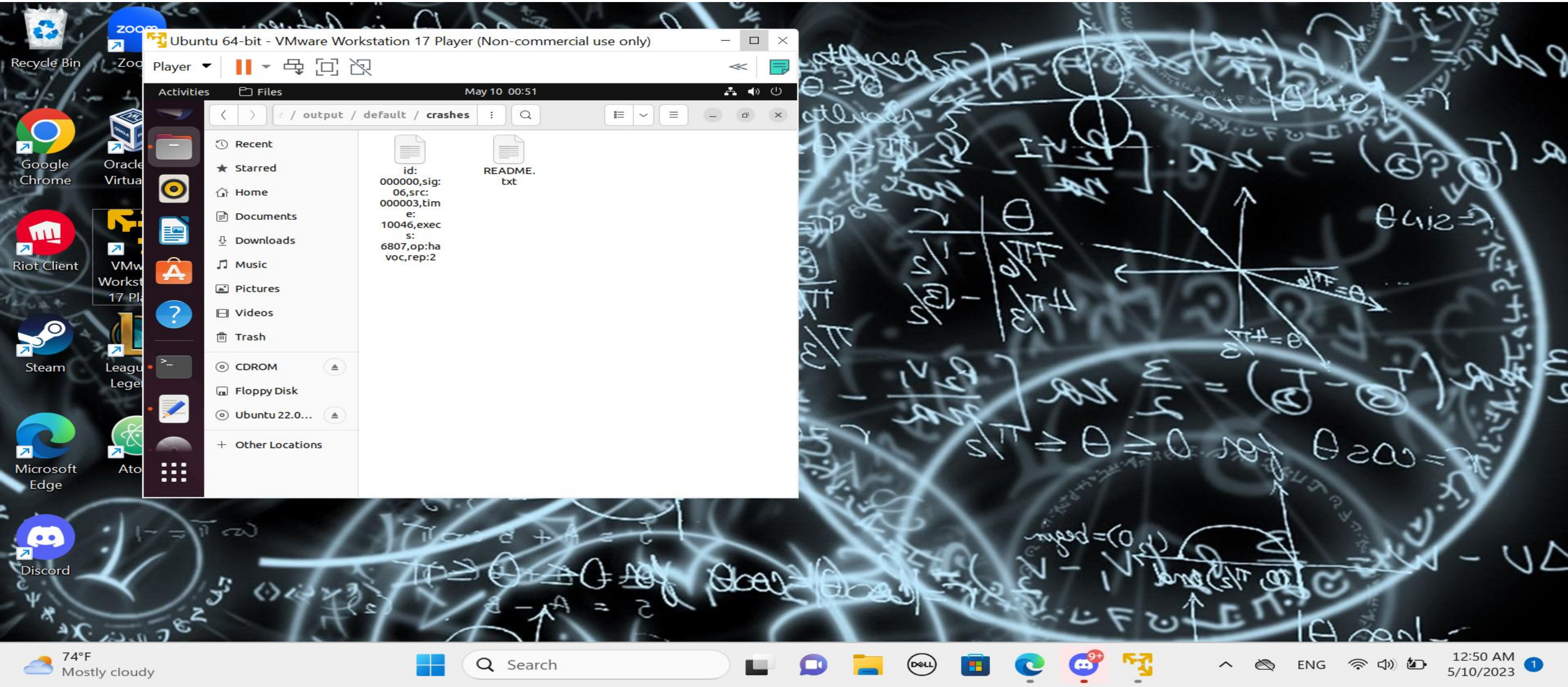
Input File



input_fuzz.txt
~/Downloads/AFL-master/input

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa buzz muzz huzz
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb c cccccccccccccccccccccccc
trhegsegrhnyrtew4gehryjukfjyrhew           drghtejyrthegwfgrteh
```

Crash File



```
id:000000,sig:06,src:000003,time:10046,execs:6807,op:havoc,rep:2
README.txt
```

Summary/Lab Report

First step indicates that we have to compile the test cases using this command gcc -o ./test-instr ./test-instr.c then ./test-instr to see the inputs and outputs. If you get put a value that is not 0, it will return this is not zero but if zero, then say it is zero.

Second step indicates that we have to set up the fuzzing process. Using gcc to compile afl-gcc -o ./test-instr ./test-instr.c, you get to instrument and the instrumentation of the process. I created an output folder named output to store my crashes in there. After this I ran into a problem until I found my path executable to directly compile and fuzz it. I used this to fuzz it afl-fuzz -i ./testcases -o ./output /home/aaron/Downloads/AFL-master/test-instr @@ with my path in there. We tested the cases and let it run for a few cycles.

Third step indicates that we have to find the crash file so it seems that it is stored in output so we did ls ./output. We examine the crashing test case and its content. It was stored in the default area so we access the crashes by ls ,/output/default/crashes in order to find the crash file. We run with the test-instr program. ./test-instr < ./output/default/crashes/id:000000,op:havoc,rep:32. It would cause a segmentation fault.

Last Step

For the last step, we want to compile the program of vulnerable.c using afl-gcc -o vulnerable_instrumented vulnerable.c.

Prepare the initial test input:

- Create a directory to store the initial test case files (e.g., "input").
- Place your initial test input file (e.g., "input_fuzz.txt") in the input directory.

2. Set up the AFL fuzzing command:
   - Open a terminal and navigate to the directory containing the compiled program and the input directory.
   - Run the AFL fuzzing engine with the following command: afl-fuzz -i input -o output -- ./vulnerable_instrumented @@
     i. Replace "input" with the actual path to the input directory.
     ii. Replace "output" with the desired path for AFL to store its findings.
     iii. "vulnerable_instrumented" is the name of the compiled program.
   - Monitor the AFL execution status:
     i. AFL will mutate the initial test input files and feed them to the program.
     ii. It will monitor code coverage, detect crashes, and explore different program paths.
     iii. The AFL execution status will be displayed in the terminal, providing information about the progress, number of iterations, unique crashes found, and other relevant details.
   - Analyze the output:
     i. If AFL discovers crashes or abnormal behaviors during the fuzzing process, it will save the input that triggered the issue in the output directory.
     ii. Analyze the crash inputs in the output directory to understand the vulnerabilities in the program.