NAME : RISHI KUMAR SONI ***UTA ID : 1001774020***

Question No 1:

How do you execute the vulnerable program with AFL? What is the fuzzing result? List necessary commands you use. Also show a screenshot of AFL result with timestamp.

ANSWER NO 1:

- ***Execute Program***

The Correct way to execute or recompile the target program depends on the build process but the common way is :

```
$ CC=/path/to/afl/afl-gcc ./configure
$ make clean all
```

Executing the File :

```
$ ./afl-fuzz -i testcase_dir -o findings_dir /path/to/program [...params...]
```

Reading the input from the directory.

OR

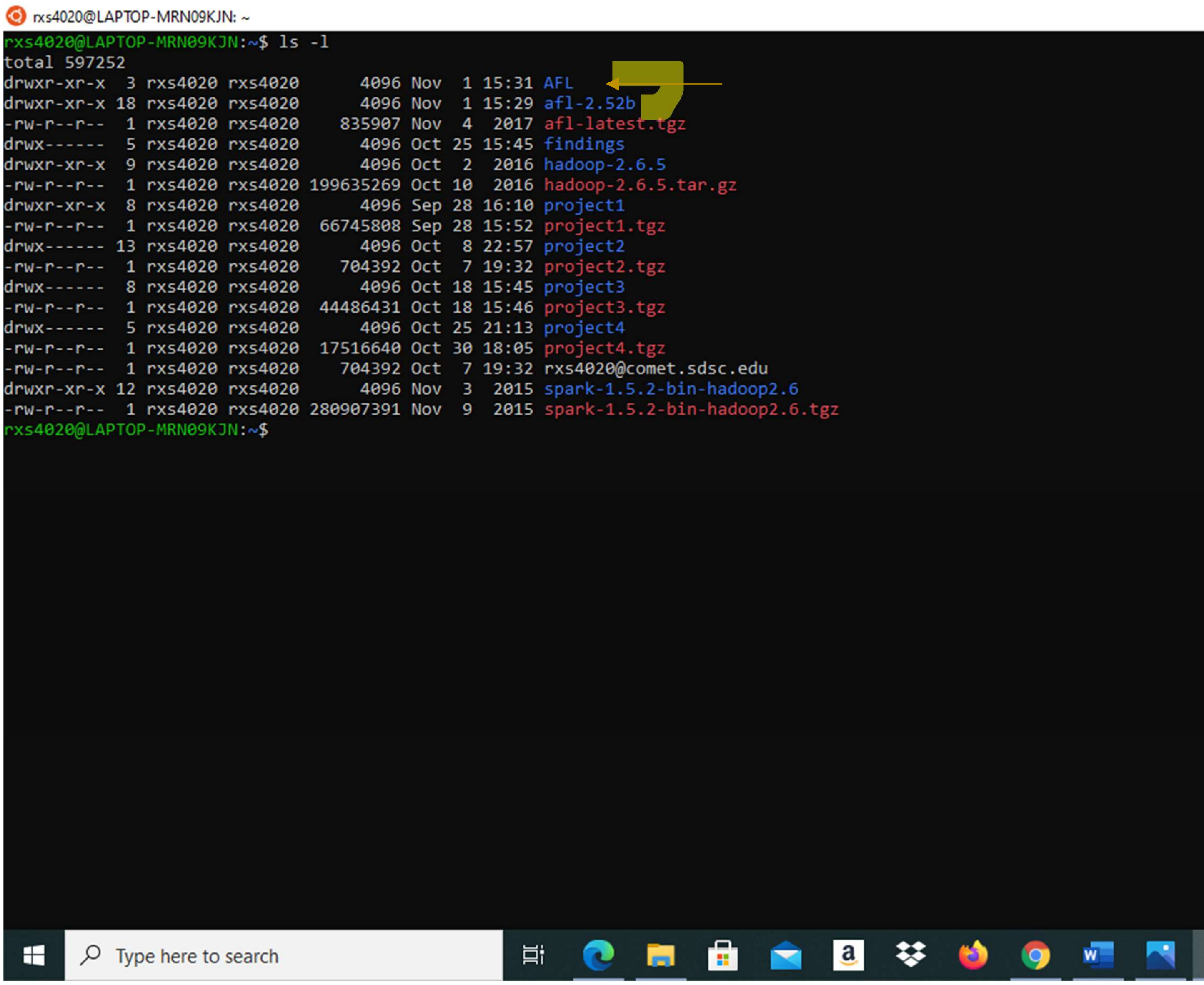
```
$ ./afl-fuzz -i testcase_dir -o findings_dir /path/to/program @@
```

'@@' to mark the location in the target command.

Output :

The Fuzzing process will execute until ctrl+c is pressed.
At minimum you want to allow the fuzzer to execute 1 cycle.

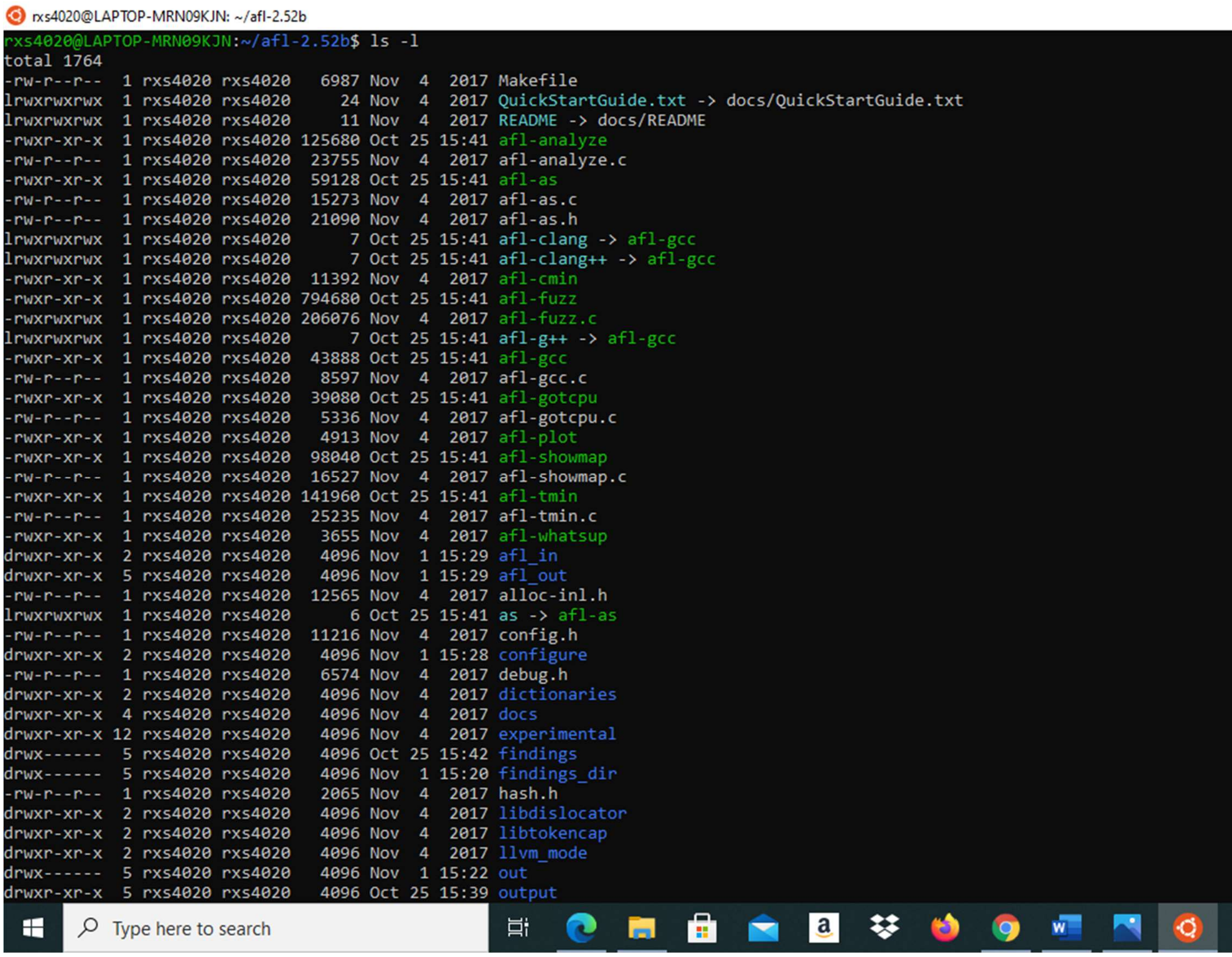
- ***SCREENSHOTS/ FUZZING RESULTS*** :



rxs4020@LAPTOP-MRN09KJN: ~

```
rxs4020@LAPTOP-MRN09KJN:~$ ls -l
total 597252
drwxr-xr-x  3 rxs4020 rxs4020      4096 Nov  1 15:31 AFL
drwxr-xr-x 18 rxs4020 rxs4020      4096 Nov  1 15:29 afl-2.52b
-rw-r--r--  1 rxs4020 rxs4020    835907 Nov  4  2017 afl-latest.tgz
drwx------  5 rxs4020 rxs4020      4096 Oct 25 15:45 findings
drwxr-xr-x  9 rxs4020 rxs4020      4096 Oct  2  2016 hadoop-2.6.5
-rw-r--r--  1 rxs4020 rxs4020 199635269 Oct 10  2016 hadoop-2.6.5.tar.gz
drwxr-xr-x  8 rxs4020 rxs4020      4096 Sep 28 16:10 project1
-rw-r--r--  1 rxs4020 rxs4020  66745808 Sep 28 15:52 project1.tgz
drwx------ 13 rxs4020 rxs4020      4096 Oct  8 22:57 project2
-rw-r--r--  1 rxs4020 rxs4020    704392 Oct  7 19:32 project2.tgz
drwx------  8 rxs4020 rxs4020      4096 Oct 18 15:45 project3
-rw-r--r--  1 rxs4020 rxs4020  44486431 Oct 18 15:46 project3.tgz
drwx------  5 rxs4020 rxs4020      4096 Oct 25 21:13 project4
-rw-r--r--  1 rxs4020 rxs4020  17516640 Oct 30 18:05 project4.tgz
-rw-r--r--  1 rxs4020 rxs4020    704392 Oct  7 19:32 rxs4020@comet.sdsc.edu
drwxr-xr-x 12 rxs4020 rxs4020      4096 Nov  3  2015 spark-1.5.2-bin-hadoop2.6
-rw-r--r--  1 rxs4020 rxs4020 280907391 Nov  9  2015 spark-1.5.2-bin-hadoop2.6.tgz
rxs4020@LAPTOP-MRN09KJN:~$
```

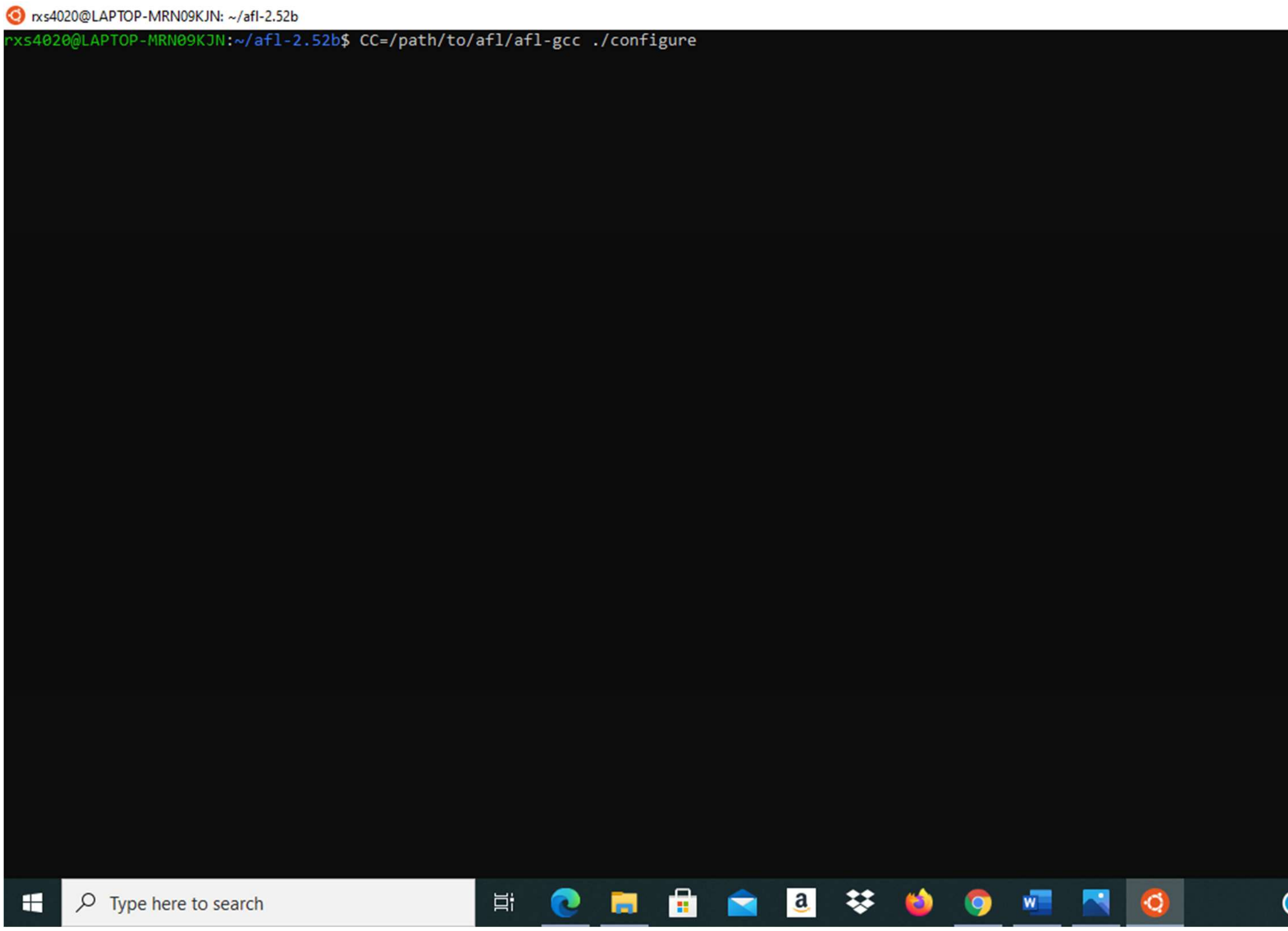
2. AFL FILES



rxs4020@LAPTOP-MRN09KJN: ~/afl-2.52b

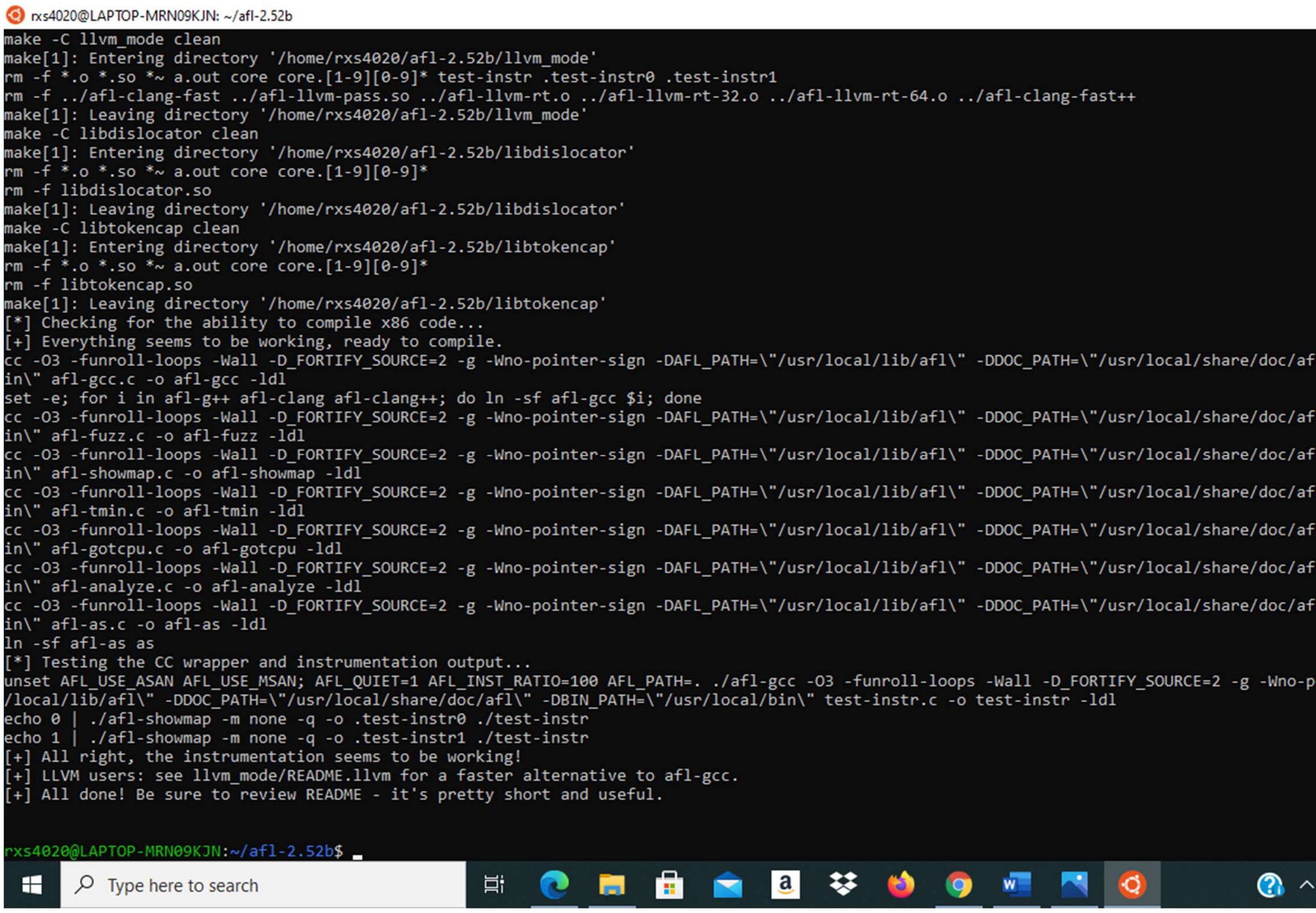
```
rxs4020@LAPTOP-MRN09KJN:~/afl-2.52b$ ls -l
total 1764
-rw-r--r--  1 rxs4020 rxs4020   6987 Nov  4  2017 Makefile
lrwxrwxrwx  1 rxs4020 rxs4020     24 Nov  4  2017 QuickStartGuide.txt -> docs/QuickStartGuide.txt
lrwxrwxrwx  1 rxs4020 rxs4020     11 Nov  4  2017 README -> docs/README
-rwxr-xr-x  1 rxs4020 rxs4020 125680 Oct 25 15:41 afl-analyze
-rw-r--r--  1 rxs4020 rxs4020  23755 Nov  4  2017 afl-analyze.c
-rwxr-xr-x  1 rxs4020 rxs4020  59128 Oct 25 15:41 afl-as
-rw-r--r--  1 rxs4020 rxs4020  15273 Nov  4  2017 afl-as.c
-rw-r--r--  1 rxs4020 rxs4020  21090 Nov  4  2017 afl-as.h
lrwxrwxrwx  1 rxs4020 rxs4020      7 Oct 25 15:41 afl-clang -> afl-gcc
lrwxrwxrwx  1 rxs4020 rxs4020      7 Oct 25 15:41 afl-clang++ -> afl-gcc
-rwxr-xr-x  1 rxs4020 rxs4020  11392 Nov  4  2017 afl-cmin
-rwxr-xr-x  1 rxs4020 rxs4020 794680 Oct 25 15:41 afl-fuzz
-rwxrwxrwx  1 rxs4020 rxs4020 206076 Nov  4  2017 afl-fuzz.c
lrwxrwxrwx  1 rxs4020 rxs4020      7 Oct 25 15:41 afl-g++ -> afl-gcc
-rwxr-xr-x  1 rxs4020 rxs4020  43888 Oct 25 15:41 afl-gcc
-rw-r--r--  1 rxs4020 rxs4020   8597 Nov  4  2017 afl-gcc.c
-rwxr-xr-x  1 rxs4020 rxs4020  39080 Oct 25 15:41 afl-gotcpu
-rw-r--r--  1 rxs4020 rxs4020   5336 Nov  4  2017 afl-gotcpu.c
-rwxr-xr-x  1 rxs4020 rxs4020   4913 Nov  4  2017 afl-plot
-rwxr-xr-x  1 rxs4020 rxs4020  98040 Oct 25 15:41 afl-showmap
-rw-r--r--  1 rxs4020 rxs4020  16527 Nov  4  2017 afl-showmap.c
-rwxr-xr-x  1 rxs4020 rxs4020 141960 Oct 25 15:41 afl-tmin
-rw-r--r--  1 rxs4020 rxs4020  25235 Nov  4  2017 afl-tmin.c
-rwxr-xr-x  1 rxs4020 rxs4020   3655 Nov  4  2017 afl-whatsup
drwxr-xr-x  2 rxs4020 rxs4020   4096 Nov  1 15:29 afl_in
drwxr-xr-x  5 rxs4020 rxs4020   4096 Nov  1 15:29 afl_out
-rw-r--r--  1 rxs4020 rxs4020  12565 Nov  4  2017 alloc-inl.h
lrwxrwxrwx  1 rxs4020 rxs4020      6 Oct 25 15:41 as -> afl-as
-rw-r--r--  1 rxs4020 rxs4020  11216 Nov  4  2017 config.h
drwxr-xr-x  2 rxs4020 rxs4020   4096 Nov  1 15:28 configure
-rw-r--r--  1 rxs4020 rxs4020   6574 Nov  4  2017 debug.h
drwxr-xr-x  2 rxs4020 rxs4020   4096 Nov  4  2017 dictionaries
drwxr-xr-x  4 rxs4020 rxs4020   4096 Nov  4  2017 docs
drwxr-xr-x 12 rxs4020 rxs4020   4096 Nov  4  2017 experimental
drwx------  5 rxs4020 rxs4020   4096 Oct 25 15:42 findings
drwx------  5 rxs4020 rxs4020   4096 Nov  1 15:20 findings_dir
-rw-r--r--  1 rxs4020 rxs4020   2065 Nov  4  2017 hash.h
drwxr-xr-x  2 rxs4020 rxs4020   4096 Nov  4  2017 libdislocator
drwxr-xr-x  2 rxs4020 rxs4020   4096 Nov  4  2017 libtokencap
drwxr-xr-x  2 rxs4020 rxs4020   4096 Nov  4  2017 llvm_mode
drwx------  5 rxs4020 rxs4020   4096 Nov  1 15:22 out
drwxr-xr-x  5 rxs4020 rxs4020   4096 Oct 25 15:39 output
```

3.



```
rxs4020@LAPTOP-MRN09KJN: ~/afl-2.52b
rxs4020@LAPTOP-MRN09KJN:~/afl-2.52b$ CC=/path/to/afl/afl-gcc ./configure
```

4.



rxs4020@LAPTOP-MRN09KJN: ~/afl-2.52b

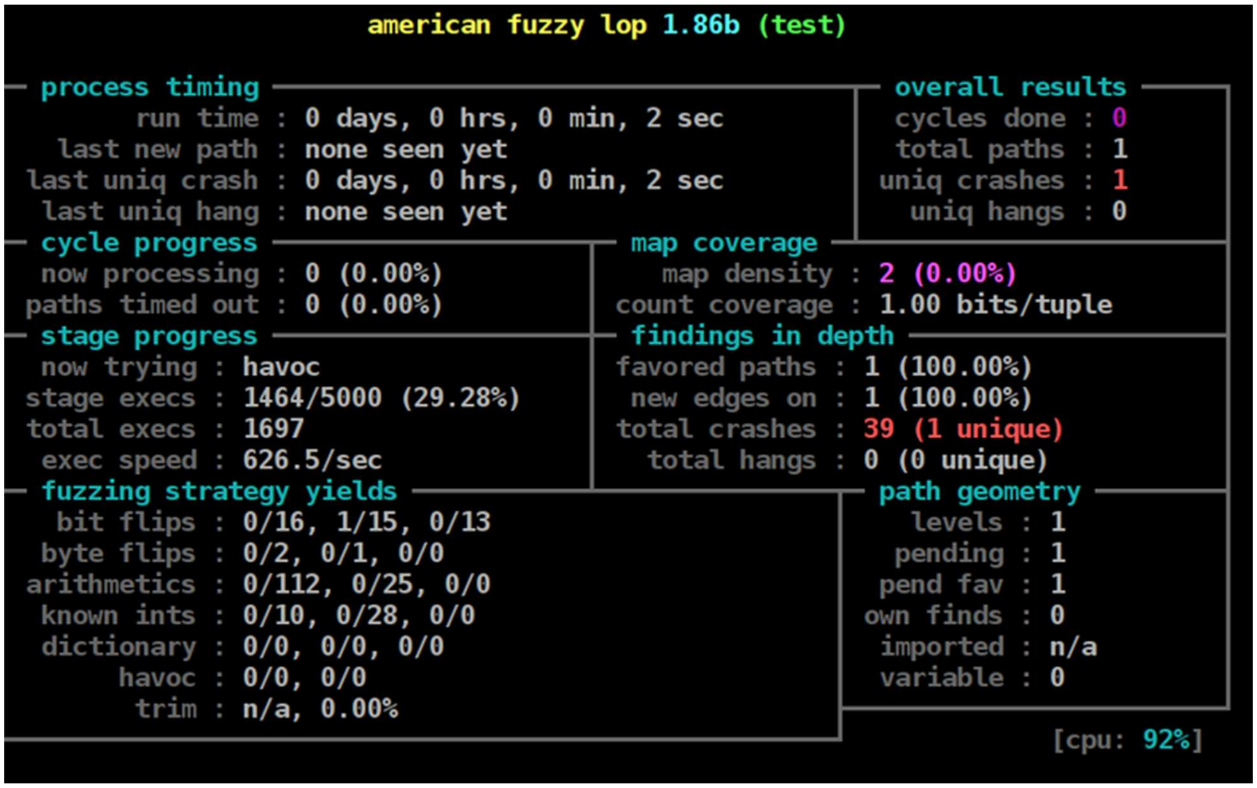
```
make -C llvm_mode clean
make[1]: Entering directory '/home/rxs4020/afl-2.52b/llvm_mode'
rm -f *.o *.so *~ a.out core core.[1-9][0-9]* test-instr .test-instr0 .test-instr1
rm -f ../afl-clang-fast ../afl-llvm-pass.so ../afl-llvm-rt.o ../afl-llvm-rt-32.o ../afl-llvm-rt-64.o ../afl-clang-fast++
make[1]: Leaving directory '/home/rxs4020/afl-2.52b/llvm_mode'
make -C libdislocator clean
make[1]: Entering directory '/home/rxs4020/afl-2.52b/libdislocator'
rm -f *.o *.so *~ a.out core core.[1-9][0-9]*
rm -f libdislocator.so
make[1]: Leaving directory '/home/rxs4020/afl-2.52b/libdislocator'
make -C libtokencap clean
make[1]: Entering directory '/home/rxs4020/afl-2.52b/libtokencap'
rm -f *.o *.so *~ a.out core core.[1-9][0-9]*
rm -f libtokencap.so
make[1]: Leaving directory '/home/rxs4020/afl-2.52b/libtokencap'
[*] Checking for the ability to compile x86 code...
[+] Everything seems to be working, ready to compile.
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/af
in\" afl-gcc.c -o afl-gcc -ldl
set -e; for i in afl-g++ afl-clang afl-clang++; do ln -sf afl-gcc $i; done
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/af
in\" afl-fuzz.c -o afl-fuzz -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/af
in\" afl-showmap.c -o afl-showmap -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/af
in\" afl-tmin.c -o afl-tmin -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/af
in\" afl-gotcpu.c -o afl-gotcpu -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/af
in\" afl-analyze.c -o afl-analyze -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/af
in\" afl-as.c -o afl-as -ldl
ln -sf afl-as as
[*] Testing the CC wrapper and instrumentation output...
unset AFL_USE_ASAN AFL_USE_MSAN; AFL_QUIET=1 AFL_INST_RATIO=100 AFL_PATH=. ./afl-gcc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-p
/local/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" test-instr.c -o test-instr -ldl
echo 0 | ./afl-showmap -m none -q -o .test-instr0 ./test-instr
echo 1 | ./afl-showmap -m none -q -o .test-instr1 ./test-instr
[+] All right, the instrumentation seems to be working!
[+] LLVM users: see llvm_mode/README.llvm for a faster alternative to afl-gcc.
[+] All done! Be sure to review README - it's pretty short and useful.


rxs4020@LAPTOP-MRN09KJN:~/afl-2.52b$ _
```

5.

FUZZING RESULT :

NOTE : I changed the program name to test.c



```
                         american fuzzy lop 1.86b (test)

- process timing ----------------------------------------- overall results --
        run time : 0 days, 0 hrs, 0 min, 2 sec           |  cycles done : 0
   last new path : none seen yet                         |  total paths : 1
 last uniq crash : 0 days, 0 hrs, 0 min, 2 sec           | uniq crashes : 1
  last uniq hang : none seen yet                         |   uniq hangs : 0
- cycle progress ---------------------- map coverage ------------------------
  now processing : 0 (0.00%)          |    map density : 2 (0.00%)
 paths timed out : 0 (0.00%)          | count coverage : 1.00 bits/tuple
- stage progress ---------------------- findings in depth -------------------
  now trying : havoc                  | favored paths : 1 (100.00%)
 stage execs : 1464/5000 (29.28%)     |  new edges on : 1 (100.00%)
 total execs : 1697                   | total crashes : 39 (1 unique)
  exec speed : 626.5/sec              |   total hangs : 0 (0 unique)
- fuzzing strategy yields ------------------------- path geometry -----------
   bit flips : 0/16, 1/15, 0/13                   |    levels : 1
  byte flips : 0/2, 0/1, 0/0                      |   pending : 1
 arithmetics : 0/112, 0/25, 0/0                   |  pend fav : 1
  known ints : 0/10, 0/28, 0/0                    | own finds : 0
  dictionary : 0/0, 0/0, 0/0                      |  imported : n/a
       havoc : 0/0, 0/0                           |  variable : 0
        trim : n/a, 0.00%                         |-----------------------
--------------------------------------------------              [cpu: 92%]
```

I STOPPED IT AT 92 %.

OUTPUT AS A TEXT :

american fluffy best 1.86b — prepare timing run time : O days, O last modern path : none seen last uniq crash . O days, O last uniq hang none seen — cycle progress now preparing . o (0.00%) paths planned out : o (0.00%) — organize progress hrs, yet hrs, yet O O min, min, 2 sec 2 sec (test) overall results cycles done : O total ways : 1 uniq crashes : 1 uniq hangs map coverage map thickness : 2 (O. 00%) 1.00 bits/tuple count scope : findings in depth now attempting : havoc 1464/5000 (29.28%) stage execs : total execs : 1697 exec speed 626.5/sec — fuzzing methodology yields favored ways : new edges on : total crashes : total hangs : 1 (100.00%) 1 (100.00%) 39 (1 unique) O (O unique) path geometry bit flips : byte flips . arithmetics : known ints : dictionary : havoc : trim : 0/16, 1/15, 0/13 0/2, 0/1, 0/0 0/112, 0/25, 0/0 0/10, 0/0, 0/0, n/a, 0/28, 0/0 0/0, 0/0 levels . pending : pend fav : own finds : impo rted : variable : 1 1 1 n/a [cpu:

Command Used :

For compiling :

```
$ env CC=afl-clang make

afl-clang -std=c99 -g -Wall -pedantic -O3 -c -o test-instr.o test-instr.c
```

We can use either GCC OR CLANG.

2 . Make a directory for input and copy the input there.

```
$ mkdir -p afl/in

$ cp testcases afl/in
```

```
3. Make Output Directory
```

```
$ mkdir -p afl/out
```

```
4. Start afl-fuzz
```

```
afl-fuzz -i afl/in -o afl/out guff -x -ly
```

Question No 2:

What is Fuzz Testing (basic idea) and its challenges? Which method does AFL use to solve the problems?

Answers 2:

Definition:

Fuzz Testing or Fuzzing is a software testing technique of putting invalid or random data called FUZZ into Software System to discover coding errors and security loopholes.

Purpose:

The purpose of Fuzz testing is inserting data using automated or semi-automated techniques and testing the system for various exceptions like system crashing or failure of build-in code etc.

Challenges:

There are some of the challenges in the Fuzzing testing and they are:

- Fuzzing is also relatively shallow; blind random mutations and make it very unlikely to reach certain code paths I the tested codes. Because of that it leaves some vulnerabilities outside the reach of this technique.

- Fuzz testing aims to bombard software with unexpected input. This practice is known as negative testing. The purpose of negative testing is to ensure the application remains stable in complex situation. The greatest challenge with negative testing is that there is an infinite combination of invalid inputs to test. It is impossible to test all the possible ways.

<u>Which method does AFL use ?</u>

American Fuzzy use ***a brute-force fuzzer,*** coupled with an exceedingly simple but rock-solid instrumentation-guided genetic algorithm. It uses a modified form of edge coverage to effortlessly pick up subtle, local-scale changes to program control flow.

QUESTION 3:

Briefly explain the overall algorithm of AFL approach?

ASNWER 3:

AFL usually use the brute-Force method to solve problems.In solving the problems, some of the basic steps involved are:

Simplifying a bit, the overall algorithm can be summed up as:

1.Load user-supplied initial test cases into the queue,

2.Take next input file from the queue,

3.Attempt to trim the test case to the smallest size that does not alter the measured behavior of the program,

4.Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies,

5.If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue.

Go to 2.

The fuzzer is thoroughly tested to deliver out-of-the-box performance far superior to blind fuzzing or coverage-only tools.

QUESTION 4:

How does AFL measure the code coverage? List formula if available. (Hint: see its technical details file)

Answer 4:

1. By ***CODE INJECTION.***

AFL(American Fuzzy Logic) , the instrumentation injected into compiled program captures branch (edge) coverage, along with coarse branch – taken hit counts.

OR

AFL – Coverage uses test case files produced by the AFL-FUZZER to produce gcov code coverage results of the targeted binary. Code coverage is interpreted from one case to the next by afl-cov in order to determine which new functions and lines are hit by AFL with each new test case.

The code injected at branch points is essentially equivalent to:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

LIST OF FORMUALS/COMMAND:

```
$ cd /PATH TO YOUR PROJECT/
$ afl-cov -d /path/to/afl-fuzz-output/ --live --coverage-cmd \
"cat AFL_FILE | LD_LIBRARY_PATH=./lib/.libs ./bin/.libs/somebin -a -b -c" \
--code-dir .
```

/PATH/TO/AFL-FUZZ-OUTPUT IS THE OUTPUT FILE.

CALCULATION OF CODE COVERAGE/FORMULAS :

1.Code Coverage Percentage = (Number of lines of code executed by a testing algorithm/Total number of lines of code in a system component) * 100.

2. $ID(A \rightarrow B)$ def $= (ID(A) \gg 1) \oplus ID(B)$.

Where ID = ID which is used for the transition between two phases.

A and B are two phase or states.

CODE COVERAGE CRITERIA:

To measure the code coverage, there are various criteria are as Follows.

1. Statement Coverage
2. Condition Coverage
3. Decision Coverage
4. Path Coverage

5. Branch Coverage
6. Function Coverage
7. Edge Coverage

QUESTION 5:

What is fork server? Why does AFL use it during the fuzzing?

ANSWER 5:

Definition :

The fork server is an integral aspect of the injected instrumentation and simply stops at the first instrumented function to await commands from afl-fuzz.

Why it's use during Fuzzing ??

In AFL, With fast targets, the fork server can offer considerable performance gains, usually between 1.5x and 2x. It is also possible to:

:Use the fork server in manual ("deferred") mode, skipping over larger, user-selected chunks of initialization code. It requires very modest code changes to the targeted program, and with some targets, can produce 10x+ performance gains.

:Enable "persistent" mode, where a single process is used to try out multiple inputs, greatly limiting the overhead of repetitive fork ()

calls. This generally requires some code changes to the targeted program,
but can improve the performance of fast targets by a factor of 5 or more
- approximating the benefits of in-process fuzzing jobs while still
maintaining very robust isolation between the fuzzer process and the
targeted binary.