CMPT 473 - Assignment 4 Write-up

Group members:
1. Sherman Chao Wen Chow - 301232684
2. Junho Sohn - 301301147
3. Fleppine Choi - 301265433

Detecting bugs in an existing test suite using Valgrind

Project to analyze:

1. Identification of the open-source project: Google Test
    - Link to repo: https://github.com/google/googletest
2. Identification of the supporting organization: Google
3. Size of the code base:
    - Total lines: 117,925
    - Code lines: 75,615
    - Total comment lines: 26,655
    - Total blank lines: 15,655
4. Build time to compile and link an executable from source code: 3 minutes
5. Execution time for the test suite:
    - Without Valgrind: 5.79 seconds
    - With Valgrind: 75.05 seconds

Full instructions for reproducing our results based on files that we modified and how:

1. Clone the project repository
    - git clone https://github.com/google/googletest.git
2. Change directory into the project
    - cd googletest/googletest
3. Modify CMakeLists.txt located in “googletest/googletest”
    - Open CMakeLists.txt and add “include(CTest)” on line 181
4. Make a build directory
    - mkdir build
5. Change directory into the build directory
    - cd build
6. Build the project’s tests by setting the gtest_build_tests option to ON
    - cmake -Dgtest_build_test=ON ..
7. Build the project
    - make
8. Run tests without Valgrind
    - make test
        - All 41 tests will pass
9. Run tests with Valgrind
    - ctest -T memcheck
        - Valgrind produces text output files called MemoryChecker.*.log that contains all the errors associated with the tests
        - These files are located in googletest/googletest/build/Testing/Temporary

Errors indicated by the analysis:
Valgrind results are outputted into files called MemoryChecker.*.log in the directory googletest/googletest/build/Testing/Temporary

98% tests passed, 1 tests failed out of 41

- “googletest-death-test-test” fails
- Valgrind memory check reports 49358 defects for this test

Memory checking results:
IPW - 21
Potential Memory Leak - 116724
Uninitialized Memory Conditional - 119
Uninitialized Memory Read - 1163

Valgrind reported thousands of “still reachable” and “possibly lost” memory leaks.

Error 1: 32 bytes in 1 blocks are still reachable in loss record 268 of 492
This is a false positive. This is because Valgrind is detecting leaks in the “testing” library. This is due to the fact that the “testing” library uses its own memory pool allocator where memory for destructed objects are not immediately freed and given back to the OS, but are kept for later use. Therefore, this will cause Valgrind to detect the memory as “still reachable”.

Error 2: 136 bytes in 1 blocks are possibly lost in loss record 82 of 98
This is a false positive. This is similar to Error 1 where Valgrind is reporting a “possibly lost” memory leak in the “testing” library. In this case, interior-pointers are used. Thus, Valgrind has detected that a chain of one or more pointers to the block has been found, but at least one of the pointers is an interior-pointer.

Error 3: Use of uninitialised value of size 8
This is a false positive. This is because Python uses its own small-object allocation scheme on top of malloc, PyMalloc, which causes these false positives when using Valgrind.

Fuzzing: AFL & libFuzzer

Project to analyze:

6. Identification of the open-source project: Botan
    - Link to repo: https://github.com/randombit/botan
7. Identification of the supporting organization: Randombit
8. Size of the code base:
    - Total lines: 209,104
    - Code lines: 141,680
    - Total comment lines: 31,078
    - Total blank lines: 36,346
9. Build time to compile and link an executable from source code: 10 minutes
10. Execution time for the test suite: 47 seconds

Steps to compile the project:

1. Clone the project repository
    a. git clone https://github.com/randombit/botan
2. Change directory into the project
    a. cd botan
3. Run configure.py
    a. ./configure
4. Build the project
    a. make
5. Compile the project
    a. make install (might need sudo permissions if [Errno 13] pops up)
        i. sudo make install
6. Branch off into either AFL or libFuzzer compile instructions

American fuzzy lop (AFL)

Steps to run AFL (American Fuzzy Lop):

1. Go to AFL directory & build AFL
    a. make
2. Go to botan directory and run configure script with AFL flag, then make fuzzer binaries
    a. ./configure.py --with-sanitizers --build-fuzzer=afl --unsafe-fuzzer-mode --cc-bin=/path/to/AFL/afl-g++
    b. make fuzzers
3. Create new output directory to store AFL generated files
    a. mkdir AFLoutputFiles
4. Run AFL with /src/tests/data as input and /AFLoutputFiles as output, with binary selected from the test suite (‘asn1’ in our case) and “-m none” flag to move restriction on memory usage
    a. ./afl-fuzz -i /botan/src/tests/data -o /botan/AFLoutputFiles/ -m none /botan/build/fuzzer/asn1

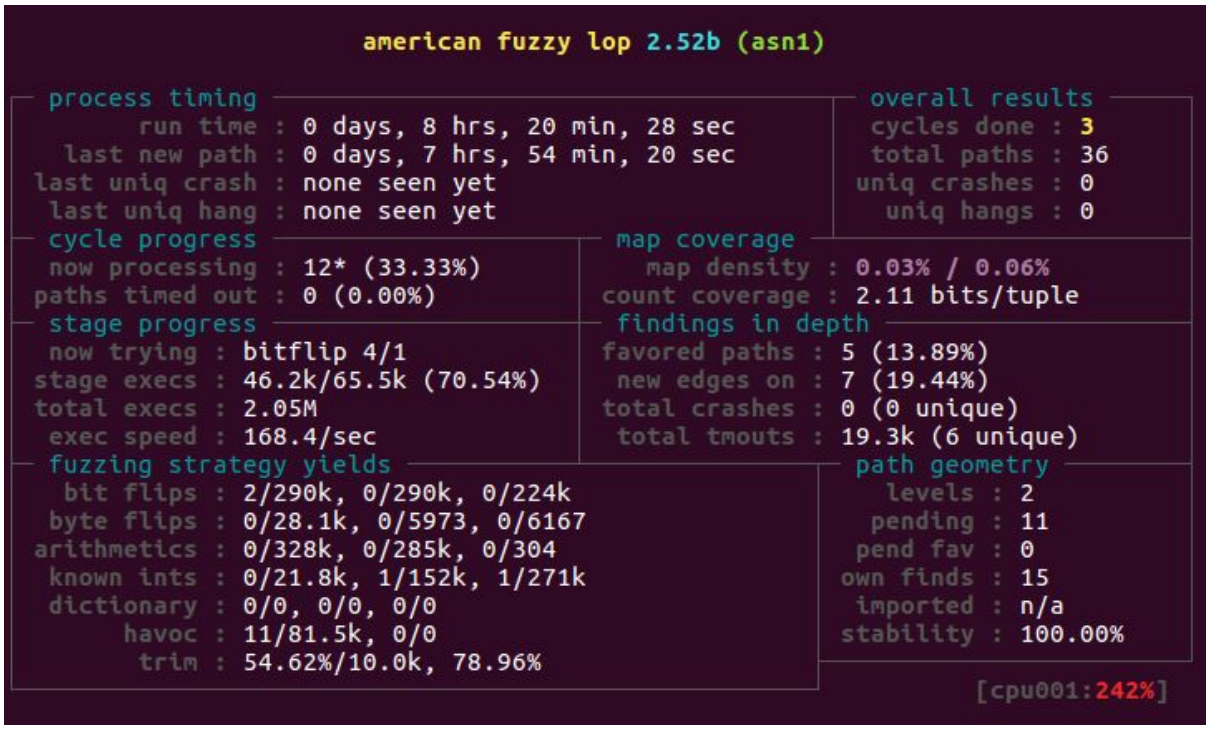

Figure 1: AFL on Botan

Errors indicated by the analysis:

After running AFL for 8 hours, no unique crashes and unique hangs were found. At the end of the run, AFL completed over 2 million executions with 3 cycles and 36 total paths of which 5 were favored. One odd behaviour was from the total number of tmouts which was 19.3k with 6 being unique, but overall no crashes occurred.

No crashes/hangs were found in Botan. Since this was a project that was created in 2006, we are thinking that this project is stable and has few (if any) remaining vulnerabilities. However, we cannot make any assumptions because we only ran this for 8 hours as required. Had we run AFL for weeks as mentioned in class, errors could have popped up. To further support our argument of few vulnerabilities, Openhub says its security confidence and vulnerability exposure index is very strong as shown in Figure 2.

Figure 2: Project Vulnerability Report

The vulnerabilities decrease per version release and even says that there is zero vulnerabilities for the current version shown in Figure 3. This implies security improvements over the last decade for every released version, hence the zero errors produced by our 8 hours of AFL testing.

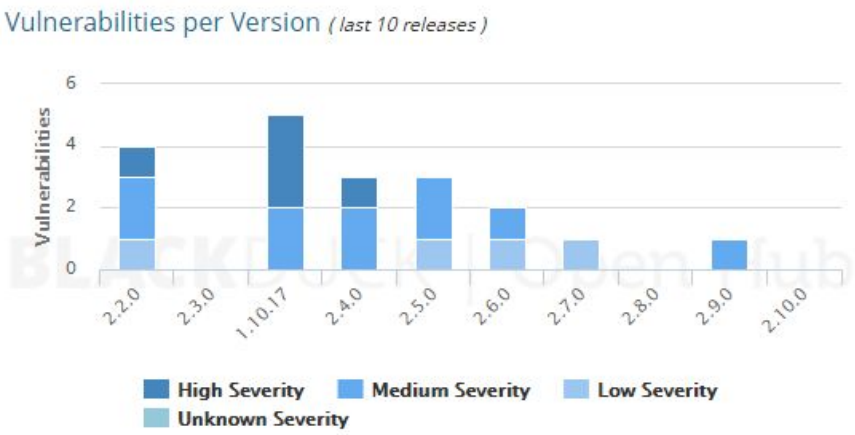

Figure 3: Vulnerabilities per Version

libFuzzer

Note the limitation of in process fuzzing: after finding the first crash, it stops. Why?
As libFuzzer is an in-process fuzzer, it takes in the code and compiles the code as if it were in a LibFuzzer main's build. When it finds a bug and the fuzzer crashes, this means that the libFuzzer's build crashed. This is similar to a normal build compiling - if a project has even one error, it would obviously crash. LibFuzzer takes that same principle and applies that with another build (in this case, "Botan"). AFL is an out-of-process fuzzer, meaning it does the opposite of libFuzzer.

Process we used to get libFuzzer working with the project:

1. Find a single parsing function for the software
    - We chose to test the parsing function, divide(), which is located in botan/src/lib/math/bigint/divide.cpp
2. Write a small test driver, fuzz_me.cc (provided in submitted files), calling the single function from the software and link against the necessary libraries and object files
3. Run the following command to compile the source using the Clang compiler with the necessary flags in order to build a fuzzer binary for the target:
    - clang++ -g -fsanitize=address,fuzzer fuzz_me.cc
4. Then execute ./a.out to run the binary
5. You will then see the libFuzzer output in the terminal

Errors indicated by the analysis:
No errors were found over 8 hours of libFuzzer.

Similar to AFL, we were unable to produce a crash/hang from running libFuzzer over 8 hours. The reasoning is equal to the reasoning of AFL's zero unique crashes/hangs shown in Figure 2 and 3. We are assuming that errors will be produced if libFuzzer was run longer than 8 hours (maybe weeks of run-time).

Contrast experiences with AFL and libFuzzer:
AFL was the simplest to implement - instructions were clear in the documentation and was easy to understand. Libfuzzer took the most time to integrate into the single parsing function that we chose (divide.cpp). This was due to the number of dependencies required to use any function in Botan, and the lack of understanding of the source code. Also, the instructions/examples given in the tutorial were unclear and lacked examples that show how to turn a function into a function that would accept the uint8_t and size_t parameters. The examples that were given are incredibly barebone and used functions that were not in their local repository, but rather sourced online (making it difficult to find).

Strengths and weaknesses of AFL and libFuzzer
Strengths of AFL is that it was easier to set up. We don't need to worry about the test harness and selecting a single parsing function to test. The weakness of AFL is there is less precision and customization in fuzzing like libFuzzer where we can go in and integrate fuzz testing for specific parsing functions.

The weakness of libFuzzer is that we had to integrate it with a single parsing function and link all dependencies in such complex open source projects. The strengths of libFuzzer is that it is very precise as it only tests one function, and we can observe the outputs more clearly.

Reflection

Challenges faced and effort required:

Valgrind
For the first part of the assignment using Valgrind, one of the challenges we faced was to integrate Valgrind into the project's existing test suite and automated testing infrastructure. Since Googletest's make command did not produce an executable for us to run "valgrind ./executable", we were stuck trying to figure out how to add Valgrind to the entire program via Makefile. We eventually found how to build Google Test's own tests in the CMakeLists.txt documentation by specifying "-Dgtest_build_tests=ON". We also spent some effort researching online on how to add valgrind tests to a CMake project in: https://stackoverflow.com/questions/40325957/how-do-i-add-valgrind-tests-to-my-cmake-test-target?fbclid=IwAR3ntbYFdqPHnDTlkTWFPk2Nki1Jl719sVclQYvYJcV5nDkmehuI9ydMHE0**.** After we found this post, it was trivial in retrieving/analyzing the output.

AFL
For AFL, the largest challenge was figuring out how to avoid identifying only one path as stated in the assignment. We had to read the documentation carefully to figure out how to clarify the inputs and outputs, along with which fuzzer/binary to run. When we tried running the compiled "botan" or "botan-test", the outputs were not desirable; this was fixed by running the fuzzers compiled in the /build/fuzzer/directory. Another challenge we faced was figuring out how to use AFL without relying on the QEMU mode, which allows non-instrumented binaries to be fuzzed in a 'black box' manner. When we tried to use the -Q flag, the fuzzing process was significantly slow and we felt that this was not the right way of using AFL. To fix this issue, we figured out how to recompile the target program using the given afl-gcc and afl-g++ files via configure.py script. By doing this, it allowed our fuzzer to run without QEMU and we were able to run AFL successfully.

LibFuzzer
The most challenging part of LibFuzzer was understanding its documentation and requirements - we spent many hours trying to figure out how to modify built-in functions into something that would follow the parameters

of LLVMFuzzerTestOneInput and link all the necessary libraries. We originally thought we could change the “const uint8_t *Data” parameter, but after reading some more documentation online, it ended up being incorrect. Another issue we had with LibFuzzer was its lack of documentation. It was incredibly difficult in finding easy to understand examples of using LLVMFuzzerTestOneInput. The share of effort spent on LibFuzzer was probably the most, when compared to Valgrind and AFL integration. We eventually overcame this challenge by finding a suitable parsing function that can use a sequence of bytes as inputs generated by libFuzzer, and linking all the necessary libraries and dependencies for the function.

Potential or received benefits of the tools used:

Valgrind

We received both potential and received benefits from using Valgrind on Googletest. When we ran the test suite without Valgrind, none of the tests failed. When we ran the tests with Valgrind, we were presented with one failed test (googletest-death-test-test). After reading the output, it was determined that the errors produced by Valgrind were simply false positives. Although these were false positives, we can attribute this to being potentially helpful in knowing that Valgrind can provide users with a “double-check your code” per-se to guarantee it isn’t actually an error. However, one could argue that this is a potential waste of time, because users would be spending time on false positives, as opposed to true positives. If these false positives were actually errors, they would be received benefits because then the users would be able to debug and find out where memory is leaking in the project.

AFL

The benefits of AFL was the simplicity in its usage. Unlike libFuzzer where we had to specify a certain parsing function, the AFL took in the full test suite and fuzzed everything at once. This meant that asides from the initial troubleshooting of setting up the fuzzer to run, the implementation was quite simple. Also, the GUI of the AFL was very intuitive, displaying important information such as the running time, number of crashes and hangs, detailed findings in depth, and the various path geometry. Having all of this information displayed while running the fuzzer meant that we could see the progress in realtime, which allowed us to see right away whether the fuzzing was working correctly. Another benefit of AFL was that it does not stop after finding a crash, unlike libFuzzer. Even though we did not detect any crashes in our program, this can be a very helpful feature in the future for fuzzing more error-prone programs.

LibFuzzer

The benefits of libFuzzer exist in using it for testing libraries with small inputs and where the library code is not expected to crash on invalid inputs. The potential of libFuzzer is that it can maximize code coverage by feeding fuzzed inputs via the target function. The fuzzer will track areas of the code reached and generate mutations on the corpus of input data. By using libFuzzer with our open source project, we were able to specify a single parsing function from the project as a fuzz target. From this, libFuzzer generated detailed outputs showing which areas of the code were reached. We also utilized the benefit of having libFuzzer integrated into the Clang compiler, and enabling it for our project with the addition of a compile flag.

Strengths and weaknesses of the different types of dynamic analysis tools used:

AFL

Strengths:

- Easy to implement
- Doesn't stop running tests even after finding hangs & crashes (could also be a weakness)
- Easy to understand GUI

Weaknesses:

- Less specific (could also be a strength)

libFuzzer

Strengths:

- More specific than AFL because it is focused on one function

Weaknesses:

- Hard to implement/understand the examples given in the documentation
- Stops after finding a crash (could also be a strength)
- If there are no crashes after a significant period of time, the user can't know if there is a bug that can occur later. You have to keep running this until something is found.

Are these reflected in your results? Why or why not?

Yes. The strengths of AFL were reflected in our results. AFL was the easiest to implement and understand, producing the results we needed. For weaknesses, AFL covers the entirety of the program, so we are assuming that it would be less specific than libFuzzer's specific testing on one function. libFuzzer's weaknesses were most prevalent in our results - it was the most time consuming implementation. For both analysis tools, they shared the lack of errors/crashes. Since Botan did not produce any errors with either AFL or libFuzzer, they can either be strengths or weaknesses in our results. They would be strengths if it means there are no errors, or weaknesses if it means the amount of time needed to analyze.

How might these compare to the strengths and weaknesses of static analysis tools?

Static analysis tools perform in a non-runtime environment, while the dynamic analysis tools we used are executed while the program is running. Dynamic analysis tools analyze system memory and overall performance, which can expose more complicated and subtle bugs than static analysis tools. However, dynamic analysis tools will only detect bugs in specific parts of the code that are being analyzed. In contrast, static analysis tools can detect potential bugs that would not have been reported by dynamic analysis tools.

Form and justify an opinion as to which was more useful for the project:

In our opinion fuzz testing was more useful than Valgrind for this project. This is because Valgrind produced thousands of false positives. This resulted in large log files that were difficult to interpret. Even though no errors were produced from AFL and libFuzzer, it was easier to understand why. Having to interpret thousands of false positives is useless compared to having no errors pop up. However, if we were to thoroughly test a program that we wrote, it would be better to use both Valgrind and one of the fuzz testing infrastructures to cover all the bases.

Submitted files:

- Valgrind:
  - valgrind-output.txt
- AFL:
  - /crashes folder
  - /hangs folder
  - fuzzer_stats
- libFuzzer:
  - fuzz_me.cc
  - output.txt