System Under Test

I spent some time researching some codebases to use for this assignment but could not find something I liked on GitHub. I did not want to spend more time trying to find a codebase than actually testing the code. Instead, I decided to use the string functions I tested for assignment 2, but this time I added a few functions to increase the number of paths.
The code I used was from a repository called Star Ruler 2, and in this repository is a str_util.cpp file. This file contains methods to manipulate strings, e.g., replace, split, and join. I decided to test the methods in this file because they are pretty straight-forward and easy to fuzz. To test these methods, I created one big GTest called "StrManipulation". I tested each individual method first to make sure the output was the same as std::string output. I then created a OneOf, so that I could test a random number of methods calls on a string. I also created helper methods that compare the toUpper, toLower, and replace methods with equivalent c methods. I used the DeepState method DeepState_CStr_C to create a string from these characters: "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-_()/?*&%$#@![]{}". I wanted to see if any special characters would cause any unwanted behavior.

This package contains split/join methods to split a string into a vector, and then join that vector into a string. I would assume that you would be able to do a roundtrip and get the original string, but this does not seem the case. The test fails every time on any input. My guess is that I may be splitting on a char that is not in the string, but this should not cause any problems with the join. The method may not handle splitting on a character not in the string. I removed the code so that I could test the rest of the package. The package also contains code to replace a substring, but the results from assignment 2 show that there are a few random errors with this code. Since the code is unstable, I decided to not include it.

To make the codebase more robust, I added the is_binary(), getSubString(), and findSubString() functions. The is_binary() method has many if-else branches and is good for increasing paths in the code. I also added a few tests that check if a Boolean string is a substring and then convert those substrings into Boolean values. The values I look for are true, false, on, off, yes, and no.

Fuzzing/Testing Approaches

For fuzzing/testing this codebase, I created a testing file using GTest and Deepstate. I also ran all of the tools using Deepstate.

The first tool I used to test was AFL. For the input, I gave AFL a file that contained the following text: "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-_()/?*&%$#@![]{}". Since there are a few specific cases I am looking for with the Boolean checks and the is_binary() function has multiple if/else statements, I believe that the number of paths AFL finds is going to be high. I let AFL run for 3 hours. In that time, it went through 3 cycles and found 344 paths with no crashes occurring. It took about 2 hours and 10 minutes to find all paths, or at least it stopped finding paths at that time. I was surprised to find a few timeouts occurring since I thought I had taken out the code that used to cause crashes/timeouts. I meant to include a screenshot of the output, but I forgot to before running AFL and lost the output.

The second tool I used was LibFuzzer (LF). The only input I passed into LF was a corpus directory so that test failures would be stored. I let LF run for 3 hours, the same as AFL, and the last execution can be seen below:

```
EXTERNAL: #3701318      NEW
EXTERNAL:  cov: 813
EXTERNAL:  ft: 2993
EXTERNAL:  corp: 147
EXTERNAL: /13942b
EXTERNAL:  exec/s: 326
EXTERNAL:  rss: 33Mb
EXTERNAL:  L: 68/321
EXTERNAL: MS: 1
EXTERNAL: CopyPart-
EXTERNAL:
```

There were 813 code blocks covered by corpus 147. I checked the output directory, and there were 936 test cases saved. I reran each of these tests by passing them into the test executable, and all tests passed. Similar to AFL, LF did not find any bugs.

The last tool I used was angr to try symbolic execution. Before running angr, I did not believe that it would find any problems since AFL and LF seem to handle strings a lot better than symbolic execution. I let angr run for 3 hours and it did not find any crashes.

Bug Information

Since there were no bugs found in the code, I had to inject one. I decided to inject one in the getSubString() function since it will be a lot harder for the fuzzers to generate input that causes the test to call getSubString() and fail on toBool(). For the bug, I changed the while loop to check for sounrceIndex < endIndex instead of sourceIndex <= endIndex. This will make the toBool() call I make immediately after the getSubString() call to fail since getSubString() will cut off the last char in the found boolean string. For example, it will return “tru” instead of “true”. Since there are only 6 calls to this function, I suspect that only 6 crashes will be found by the fuzzers. I added a new file called “str_util_bug.cpp” that contians the code with the bug. The original file “str_util.cpp” remains untouched.

```
void getSubString(char *destStr, char *sourceStr,
                                         int startIndex, int endIndex)
{
    int sourceStrLen = length_of_buffer(sourceStr);
    int destIndex = 0;
    int sourceIndex = startIndex;
    char *tempSourceStr;

    if( (startIndex >= 0) && (startIndex <= endIndex)
                                       && (endIndex < sourceStrLen) )
    {
        tempSourceStr = (char *) malloc(sourceStrLen + 1);
        copyString(tempSourceStr, sourceStr);

        while(sourceIndex < endIndex)
        {
            destStr[destIndex] = tempSourceStr[sourceIndex];

            destIndex++;
            sourceIndex++;

            // assign null character to next destination element
            destStr[destIndex] = NULL_CHAR;
        }

        free(tempSourceStr);
    }
    else
    {
        destStr[0] = NULL_CHAR;
    }
}
```

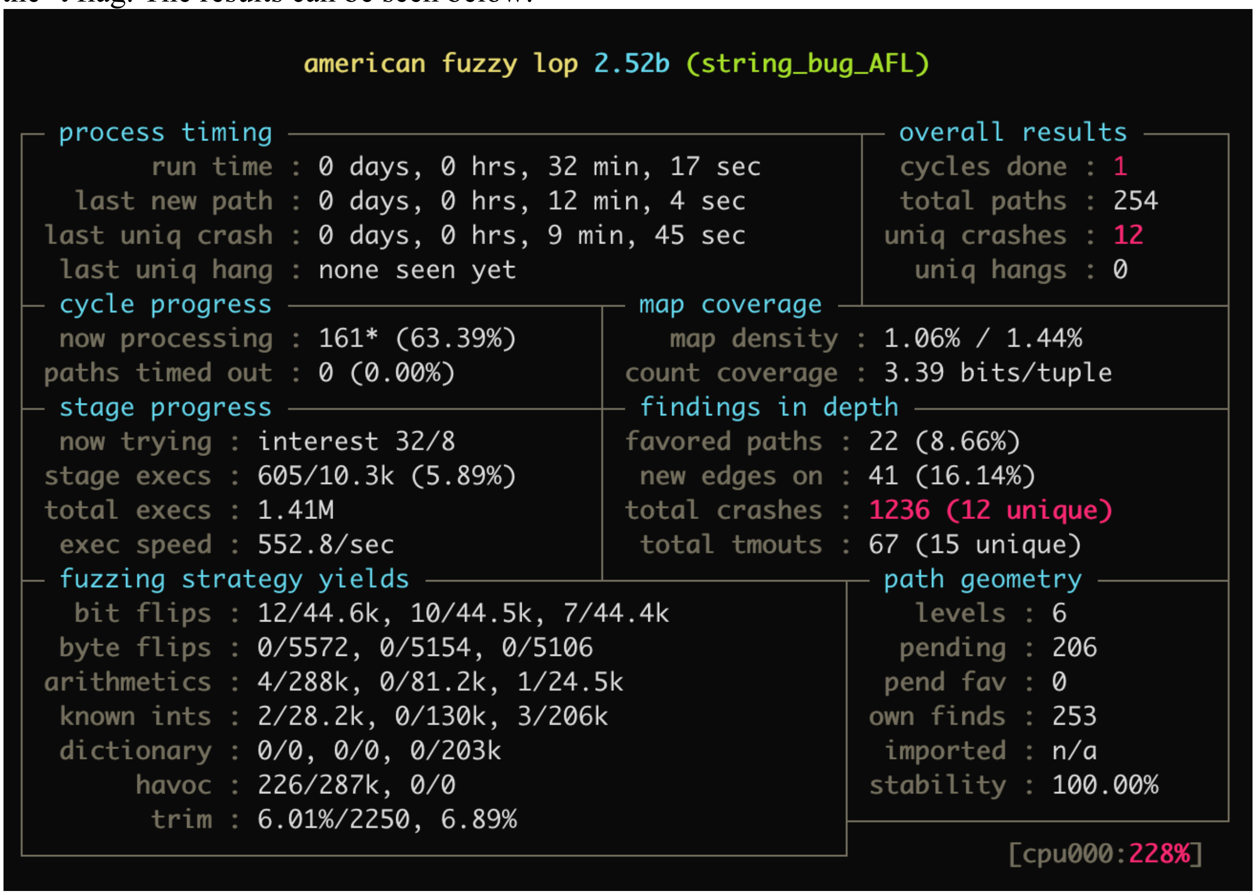
```
void getSubString(char *destStr, char *sourceStr,
                                         int startIndex, int endIndex)
{
    int sourceStrLen = length_of_buffer(sourceStr);
    int destIndex = 0;
    int sourceIndex = startIndex;
    char *tempSourceStr;

    if( (startIndex >= 0) && (startIndex <= endIndex)
                                       && (endIndex < sourceStrLen) )
    {
        tempSourceStr = (char *) malloc(sourceStrLen + 1);
        copyString(tempSourceStr, sourceStr);

        while(sourceIndex <= endIndex)
        {
            destStr[destIndex] = tempSourceStr[sourceIndex];

            destIndex++;
            sourceIndex++;

            // assign null character to next destination element
            destStr[destIndex] = NULL_CHAR;
        }

        free(tempSourceStr);
    }
    else
    {
        destStr[0] = NULL_CHAR;
    }
}
```

Comparison of Tools

To test the tools with the bug in the code, I set a 30 minute timeout to give the fuzzers some time to generate strings with boolean substrings. For the input, I used the same input used for AFL above: “abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-_()/?*&%$#@![]{}”. For coverage results, I am looking at the Tests.cpp and str_util_bug.cpp coverage results.

<u>AFL without -d</u>

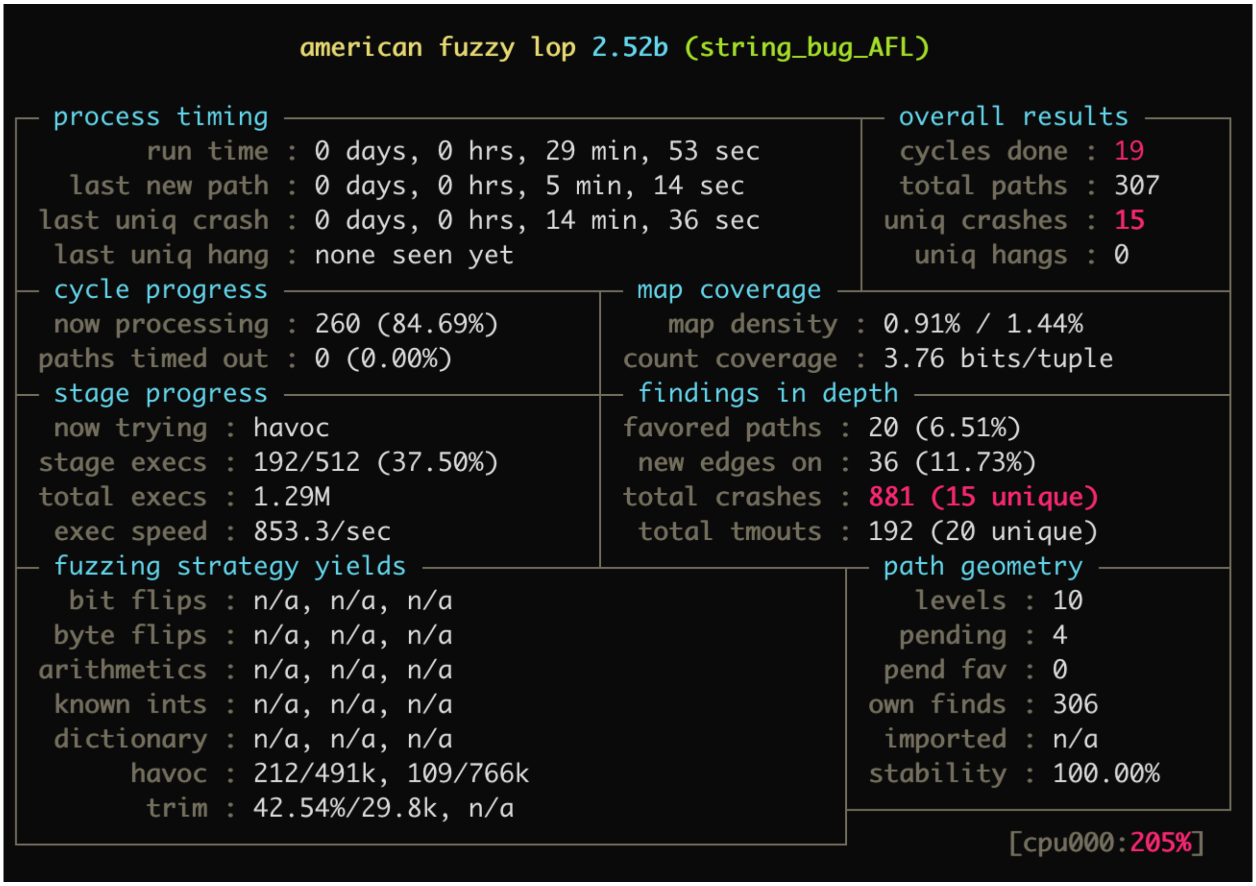
Since I am using Deepstate, I did not need to write a script to set a timeout; I only needed to use the -t flag. The results can be seen below:



AFL completed 1 cycle (not surprising) and found 254 paths. To my surprise, there were 1236 crashes and 12 unique crashes, instead of the 6 I originally predicted. After rerunning each test, I found that 6 of those tests failed on the “on” Boolean check, 5 failed on the “no” Boolean check, and 1 failed on “off” Boolean check. It looks like the shorter Boolean strings, and the ones that look similar, had more failure cases because it is easier to randomly generate those strings more than the longer ones. Only 3/6 tests failed when I expected all 6 of them to. For the coverage, I reran the test executable with the test cases that were in the queue. There was 67.66% line coverage for Tests.cpp and 42.66% for str_util_bug.cpp. Since all the test cases in the queue passed, it makes sense why the test coverage is low because we will never call getSubString() or toBool().

AFL with -d

In order to run deepstate-afl with the same behavior as afl-fuzz -d, I used the –dirty_mode flag. The results can be seen below:



Since the -d AFL flag skips the deterministic fuzzing steps, fuzzing is a lot faster. This is the reason why 19 cycles were completed, and AFL found 307 paths in the same amount of time as AFL without the -d flag. It did only find 15 crashes, 3 more than the previous run. After rerunning the tests, I found that 8 failed on the "on" Boolean check, 5 failed on the "no" Boolean check, 1 failed on the "off" Boolean check, and 1 failed on the "yes" Boolean check. This had similar results as the previous run, but this time we hit more test cases. We still haven't been able to make the "true" and "false" checks fail. For the test coverage, I reran the tests in the queue directory. All of these tests passed. Tests.cpp had 67.66% line coverage and str_util_bug.cpp had 42.66% line coverage. This is the same coverage as AFL without -d. This means that this AFL run only generated new test failures but did not discover more. More specifically, it did not find more inputs to reach different if-else branches in the is_binary method.

--fuzz

The next tool I used was deepstate –fuzz because I wanted to see if this bare bones fuzzer would be able to find the bug. The fuzzer ran 123,852 tests with 549 of them failing. I reran all of the tests and grepped for certain failure messages. I found that 284 of them failed on the "on" check, 257 failed on the "no" check, 4 failed on the "off" check, 4 failed on the "yes" check, and none failed on "true" or "false" checks. I was surprised that it found so many test failures, but not surprised that it could generate inputs to make the "true" or "false" check fail. For the coverage results, since only the failure test cases were saved, Tests.cpp had a coverage of 64.67%, str_util_bug.cpp 55.97%, and 59.13% total. Since we are only keeping track of the failures, it makes sense why the test coverage is lower.

<u>LibFuzzer</u>

The last tool I used was LibFuzzer without -use_value_profile=1. I gave it an input seed and corpus directory. In 30 minutes, it did 1499783 runs and saved 152 tests to the corpus directory. Of those 152 tests, only 13 failed. Using grep to filter the tests, I found that 3 failed the "no" check, 5 failed the "on" check, and 5 failed the "off" check. LF performed differently than the three previous runs; LF found more "off" failures. After running the fuzzer, I generated a coverage report for LF. LF covered 83.83% of Tests.cpp and 63.14% of str_util_bug.cpp with a total coverage of 70.65%.

<u>Conclusion</u>

After running all of the fuzzers, it makes sense why they found "no", "on", and "off" way easier than the other three because of how similar those strings are. If I would have let the fuzzers run more, i.e., set a longer timeout, I think the fuzzers could have found inputs that caused the other checks to fail. I do not believe that 30 minutes is sufficient for good fuzzing, especially when working with strings. If I reran the fuzzers, I think they would generally behave the same, but they may find different numbers of failures for "on", "off", and "no". On another note, LibFuzzer had the highest line coverage percentage. This could mean that it found different inputs to reach more paths in the is_binary() method. Additionally, the brute-force fuzzer had a higher line coverage than AFL. I found this surprising, but it could be that AFL was focused on trying to find more failing inputs for the Boolean checks.