Lab 3: Random Input Generation

Synopsis

Building a coverage-guided random input generator a.k.a. “fuzzer” for testing C programs.

Objective

In this lab, you will develop a *fuzzer* for testing C programs. Fuzzing is a popular software testing technique wherein the program under test is fed randomly generated inputs. Such inputs help uncover a wide range of security-critical and crashing bugs in programs. For this purpose, your fuzzer will begin with seed inputs, and generate new inputs by mutating previous inputs. It will use output from previous rounds of test as *feedback* to direct future test generation. You will use the code coverage metrics you saw in Lab 2 to help select interesting inputs for your fuzzer to mutate.

Pre-Requisites

- Watch the video lectures corresponding to the module on “Random Testing”. The lectures introduce various terminology used throughout this lab such as seed inputs, mutations, and feedback-directed testing.

Setup

The code for Lab3 is located under `cis547vm/lab3`. We will frequently refer to the top level directory for Lab 3 as `lab3` when describing file locations for the lab. Open the `lab3` directory in VSCode following the Instructions from Course VM document

This lab builds off the code coverage instrumentation from Lab 2. You are provided with a `Instrument.cpp` file in `lab3/src`; it contains two instrumentations, namely coverage and sanitize. You have already seen code coverage in the previous lab and the implementation here is identical to it. In lab 1, you have seen that when a program encounters a divide-by-zero error it causes a Floating Point Exception and leads to a core dump. The sanitizer instrumentation inserts a call to the `__sanitize__` function before every division instruction. This function gracefully exits the program with return code `1` if the denominator is zero, otherwise the program continues running normally.

Step 1.

The fuzzer and the instrumentation is built using CMake and you can run the following command to build both of them:

```
lab3$ mkdir build && cd build
lab3/build$ cmake ..
lab3/build$ make
```

After running `make`, you should notice `InstrumentPass.so` and `fuzzer` in `lab3/build`. The `fuzzer` is the tool that will feed randomized input (that you will generate) to a compiled C program that was instrumented to exit gracefully when it hits a Divide-by-Zero error and report code coverage during execution.

Step 2.

Next, we want to prepare a test program to fuzz with the `fuzzer`. This will be done by first instrumenting the program, similar to Lab 2. So to instrument and build the program `sanity1.c` you would run:

```
lab3/test$ clang -emit-llvm -S -fno-discard-value-names -c -o sanity1.ll sanity1.c -g
lab3/test$ opt -load ../build/InstrumentPass.so -Instrument -S sanity1.ll -o sanity1.instrumented.ll
lab3/test$ clang -o sanity1 -L${PWD}/../build -lruntime -lm sanity1.instrumented.ll
```

Alternatively you can use the provided `Makefile` to do the same with:

```
lab3/test$ make sanity1  # To instrument an build just sanity1.
lab3/test$ make all      # To instrument and build everything.
```

Step 3.

Now to run the the `fuzzer` you will need to create the output directory where fuzzer will store its results.

```
lab3/test$ mkdir fuzz_output_sanity1
```

You may recall from lab 1, that AFL could generate new inputs forever and never stop running. This is also the case for your fuzzer. So for this we will use `timeout` to stop the fuzzer after a specified time.

After this you can run your fuzzer on sanity for 1 second with:

```
lab3/test$ timeout 1s ../build/fuzzer ./sanity1 fuzz_input fuzz_output_sanity1
```

Note: the `./` before `sanity1` is required to let the fuzzer find the executable.

You can also use the Makefile to setup output directory and run the fuzzer for you:

```
lab3/test$ make fuzz-sanity1
```

This will run the `fuzzer` on `sanity1` for ten seconds and store the results to `lab3/test/fuzz_output_sanity1` Additionally, it will use the `lab3/config.txt` to set the `seed` which used to generate random numbers and `freq`, which determines how often we write a non-crashing input to output (larger is less frequent). Since we expect to see many more non-crashing input `freq` is used to control how often we log a non-crashing input.

Once you have run the `fuzzer` you should expect to see `failure` directory to get populated with several randomly generated inputs that crash `sanity1.c`. You may also see some of the randomly generated inputs that don’t cause a crash under the `success` directory.

```
fuzz_output_sanity1
├── success         # Some of the generated inputs that didn't cause a crash.
│   ├── input0
│   └──  ...
├── randomSeed.txt  # The seed that was used to generate random numbers.
└── failure         # All the generated inputs that cause a crash.
    ├── input0
    ├── input1
    │     ...
    └── inputN
```

Here `N` is the last case that caused a crash before the timeout.

Lab Instructions

A full-fledged fuzzer consists of three key features:

1. test case generation matching the grammar of the program input,
2. strategies to mutate test inputs to increase code coverage,
3. a feedback mechanism to help drive the types of mutations used.

Mutation-Fuzzing Primer

Consider the following code that reads some string input from the command line:

```
int main() {
  char input[65536];
  fgets(input, sizeof(input), stdin);
  int x = 13;
  int z = 21;

  if (strlen(input) % 13 == 0) {
    z = x / 0;
  }

  if (strlen(input) > 100 && input[25] == 'a') {
    z = x / 0;
  }

  return 0;
}
```

We have two very obvious cases that cause divide-by-zero errors in the program:

- If the length of the program input is divisible by 13, or
- if the length of the input is greater than 100 and the 25th character in the string is an `a`.

Now, let’s imagine that this program is a black box, and we can only search for errors by running the code with different inputs.

We would likely try a random string, say `"abcdef"`, which would give us a successful run. From there, we could take our first string as a starting point and add some new characters,`"ghi"`, giving us `"abcdefghi"`. Here we have mutated our original input string to generate a new test case. We might repeat this process, finally stumbling on “abcdefghijklm” which is divisible by 13 and causes the program to crash.

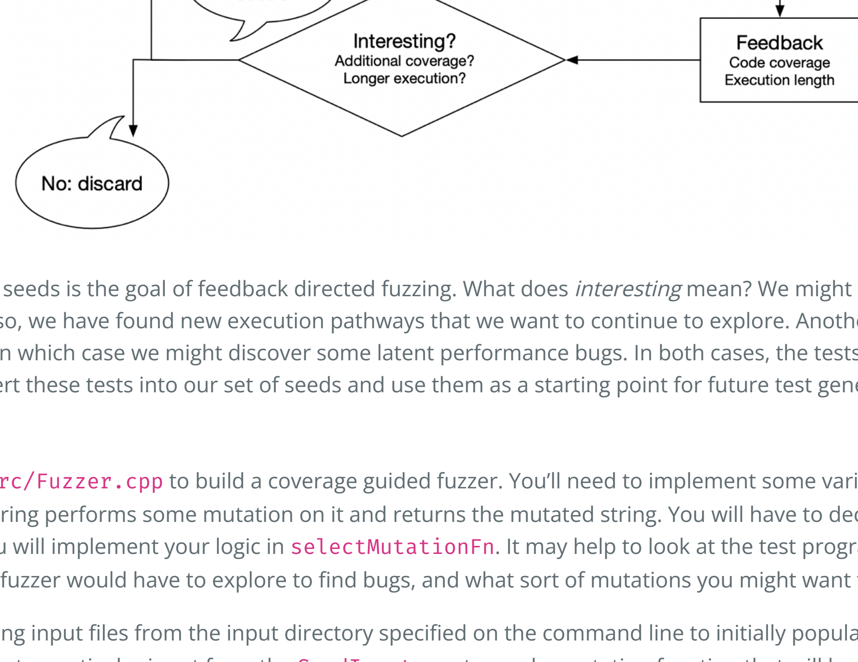
How about the second case? We could keep inserting characters onto the end of our string, which would eventually get us some large string that satisfies the first condition of the if statement (input length greater than 100), but we need to perform an additional type of mutation — randomly changing characters in the string — to eventually satisfy the second condition in the if statement.

Through the use of various mutations on an input string, we were able to exhaust all program execution paths, i.e., more varied mutations in the input increased our code coverage. In its simplest form, this is exactly what a fuzzer does. You may take a look at the Mutation-Based Fuzzing chapter in the Fuzzing Book.

Feedback-Directed Fuzzing

We’ve seen how randomized testing can find bugs and is a useful software analysis tool. The previous section describes a brute force generation of test cases; we simply perform random mutations hoping that we find a bug. This results in a lot of test cases being redundant, and therefore unnecessary.

We can gather additional information about a program’s execution and use it as *feedback* to our fuzzer. The following figure shows at a high level what this process looks like:



Generating new, interesting seeds is the goal of feedback directed fuzzing. What does *interesting* mean? We might consider whether a test increases code coverage. If so, we have found new execution pathways that we want to continue to explore. Another test might significantly increase program runtime, in which case we might discover some latent performance bugs. In both cases, the tests increased our knowledge of the program; hence, we insert these tests into our set of seeds and use them as a starting point for future test generation.

Building the Fuzzer

In this lab, you will modify `src/Fuzzer.cpp` to build a coverage guided fuzzer. You’ll need to implement some variety of mutation functions, a mutation function takes a string performs some mutation on it and returns the mutated string. You will have to decide which mutation strategies to choose and you will implement your logic in `selectMutationFn`. It may help to look at the test programs in `src/test/` to see what sort of programs your fuzzer would have to explore to find bugs, and what sort of mutations you might want to perform.

The fuzzer will start by reading input files from the input directory specified on the command line to initially populate the `SeedInputs` vector. After that, it will need to select a particular input from the `SeedInputs` vector and a mutation function that will be used to mutate it. For this, you will need to implement your logic for `selectInput`, and `selectMutationFn` respectively. Once the fuzzer has selected an input and a mutation function, it will mutate the input. The mutated input will be run on the target program, and feedback will be provided based on the coverage of that run. Using this coverage, you will then decide if this is an *interesting* seed and insert it into the `SeedInput` vector if you find it so. This allows the mutated input to be picked later on and be further mutated. This process continues until the fuzzer gets interrupted (via timeout, or on the terminal by Ctrl+C).

The following pseudo-code illustrates this logic:

```
readSeedInputs(SeedInputs)  // Initialize SeedInputs

while (true) {
  input ← selectInput()                  // Pick seed input
  mutation ← selectMutationFn()          // Pick mutation function
  mutatedInput ← mutation(input)         // Mutate input
  test(Target, MutatedInput)             // Run target with mutated input
  feedBack(Target, MutatedInput);        // Provide feedback from the run
}
```

Refer to the function `fuzz` in `src/Fuzzer.cpp` for the implementation of this logic.

Possible Mutations

The following is a list of potential suggestions for your mutations:

- Replace bytes with random values.
- Swap adjacent bytes.
- Cycle through all values for each byte.
- Remove a random byte.
- Insert a random byte.

Feel free to play around with additional mutations, and see if you can speed up the search for bugs on the binaries. You may use the C++ function `rand()` to generate a random integer.

You will notice that different programs will require different strategies, or that in some cases you may even have to switch between different mutation strategies in the middle of the fuzzing process. You are expected to include a mechanism that will try to choose the best strategy for the input program based on the coverage feedback.

Overview of the tasks

The lab consists of the following tasks in `Fuzzer.cpp`:

1. Implement your logic for `selectInput` function, which selects a mutant string from the `SeedInputs` vector.
2. Implement mutation functions you think will help your fuzzer generate a rich variety of strings. Take inspiration from the aforementioned list of mutations.
3. Implement your logic for `selectMutationFn` to decide which mutation function to pick.
4. In `feedback` decide whether the mutation was interesting based on success or failure of the program, and the code coverage. Again, you may follow our groundwork and fill in `feedback`.
5. Insert an interesting Mutant into the pool of `SeedInput` to drive further mutation.

One thing to keep in mind is that none of these tasks are compulsorily required, your fuzzer can use the default implementations we provide for some of these and still get full points, as long as it meets the grading requirements.

Code Coverage Metric

Recall that you have a way of checking how much of a particular program gets executed using the coverage information output by the instrumentation. A `.cov` file will get generated in the working directory for the program that is getting fuzzed. This file is read and is made available to you through `RawCoverageData` variable inside the `feedback` function. You can then use it to decide if a particular mutation is interesting.

Few tips

Read through the Notes, Hints, and Comments in `Fuzzer.cpp` file before you start, to get a better idea of how everything fits together.

Start small. Implement one mutation strategy at a time and try to crash the easier test cases first before moving to harder ones. Once successful, you can move on to implementing more strategies and more sophisticated ways of choosing between them based on the feedback you get.

Do not be afraid to keep track of any state between rounds of the fuzzing.

You may want to try each of your mutation strategies initially to see which one generates a test that increases code coverage, and then exploit that strategy.

Grading

We expect your fuzzer to be able to generate crashing inputs for all programs we have provided you in `lab3/test`.

Beyond that we will also be testing your fuzzer on ten hidden test programs. These programs serve as more challenging test cases for your fuzzer. To get full points on the hidden tests, your fuzzer should be able to find a crashing input in at least seven of them.

Submission

Once you are done with the lab, you can create a `submission.zip` file by using the following command:

```
lab3$ make submit
...
submission.zip created successfully.
```

Then upload the `submission.zip` file to Gradescope.

If you’d like us to use a specific seed value for your fuzzer during grading, update `lab3/config.txt` with the seed value you’d like us to use. The same seed value will be used for all test cases.