

Lab 1: Fuzzing

Fall Semester 2021

Due: 13 September, 8:00 a.m. Eastern Time

Corresponding Lesson: Lesson 3 (Random Testing)

Objective

In Part 1 you will implement a simple tool to automatically check for divide-by-zero errors in C programs at runtime. You will create an LLVM pass that will instrument C code with additional instructions that will perform runtime checks, thus creating a *sanitizer*, a form of lightweight dynamic analysis. In the spirit of automated testing, your tool will provide a code coverage mechanism that will show the actual instructions that execute when a program runs.

In Part 2 you will implement a fuzzer that will use mutation strategies to create inputs to automatically test simple programs. As we discussed in the lesson, we hope to get lucky and cause the input program to crash on some generated data. You will see how this specialized form of *mutation analysis* can perform well enough to encourage developers to use this technique to help test their software.

In Part 3 you will extend your fuzzer from Part 2 to make more interesting choices about the kinds of input it generates to test a program. The fuzzer will use output from previous rounds of test as *feedback* to *direct* future test generation. You will use the code coverage metrics implemented in Part 1 to help select more interesting seed cases for your fuzzer to mutate.

Please note that this lab is demonstrating a random testing technique. For us to give you a real taste of random testing, we have opted not to design a deterministic lab where you will see the exact same results every time you run your solution since real random testing is not deterministic. We have some tolerances built into the grader so you can have high confidence that if you are consistently crashing the provided programs, you will also consistently crash the hidden programs. As this lab is the only lab covering random testing, you can expect future labs to be easier to know that your solution is correct as most of our labs are deterministic techniques where you will get the exact same results on every execution of your code.

Note on Past Issues

In past semesters, it has caused a high number of students to be submitted to the Office of Student Integrity for Academic Integrity violations. In particular, it's possible to find solutions to similar analyses on the internet. Looking at these solutions in any form is likely to influence your

thinking and cause your solutions to be similar to them, which is an Academic Integrity violation in this class. If you are unclear of our guidelines for what is collaboration and what is cheating, we suggest reviewing that section of the syllabus. If you have any questions about what is allowed and what is not allowed, please privately on Ed Discussions for clarification.

Students who submit solutions found to be similar to online resources or other students should expect a 0 grade on the lab, a disciplinary record of an Academic Integrity issue through the Office of Student Integrity, and will not be eligible to receive a final grade of A in the course. Students who have had past Academic Integrity issues may find that OSI assigns them higher penalties.

Resources

- Enumerating basic blocks and instructions in a function:
 - <http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#basic-inspection-and-traversal-routines>
- Instrumenting LLVM IR
 - <http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#creating-and-inserting-new-instructions>
- Important classes
 - <http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#the-function-class>
 - http://cs6340.cc.gatech.edu/LLVM8Doxygen/classllvm_1_1CallInst.html
 - http://cs6340.cc.gatech.edu/LLVM8Doxygen/classllvm_1_1DebugLoc.html
- Fuzzing
 - <https://www.fuzzingbook.org/html/Fuzzer.html>
- Code Coverage
 - <https://www.fuzzingbook.org/html/Coverage.html#Comparing-Coverage>

Setup

Download fuzzing.zip from Canvas and unzip in the home directory on the VM (note: there is an extraneous fuzzing directory on the VM already - you can delete or ignore the contents). This will create a fuzzing directory with part1, part2, and part3 subdirectories.

Part 1 - Simple Dynamic Analysis

Setup

The skeleton code is located under `/fuzzing/part1/`. We will refer to the top level directory for Part 1 as `part1` when describing file locations.

Run the following commands to setup this part:

```
$ cd part1
$ mkdir build
$ cd build
$ cmake ..
$ make
```

You should see several files created in the current directory. This builds an LLVM pass from code that we provide, `part1/src/Instrument.cpp`, named `InstrumentPass.so`.

Note each time you update `Instrument.cpp` you will need to rerun the make command in the build directory before testing.

Next, let's run our dummy `Instrument` pass over some C code that contains a divide-by-zero error:

```
$ cd part1/test
$ clang -emit-llvm -S -fno-discard-value-names -c -o simple0.ll simple0.c -g
$ opt -load ../build/InstrumentPass.so -Instrument -S simple0.ll -o
simple0.instrumented.ll
$ clang -o simple0 ../lib/runtime.c simple0.instrumented.ll
$ ./simple0
```

If you've done everything correctly up to this point, you should see `Floating point exception (core dumped)`. For the lab, you will complete the `Instrument` pass to catch this error at runtime.

Format of Input Programs

All C programs are valid input programs.

Lab Instructions

In this lab, you will implement a dynamic analysis tool that catches divide-by-zero errors at runtime. A key component of dynamic analysis is that we inspect a *running* program for information about its state and behavior. We will use an LLVM pass to insert runtime checking and monitoring code into an existing program. In this lab, our instrumentation will perform

divide-by-zero error checking, and record coverage information for a running program. In the following part of the lab, we will introduce an automated testing framework using our dynamic analysis.

Instrumentation and Code Coverage Primer. Consider the following code snippet where we have two potential divide-by-zero errors, one at Line 1, the other at Line 2.

```
int main() {
    int x1 = input();
    int y = 13 / x1;    // Line 1
    int x2 = input();
    int z = 21 / x2;    // Line 2
    return 0;
}
```

If we wanted to program a bit more defensively, we would manually insert checks before these divisions, and print out an error if the divisor is 0:

```
int main() {
    int x1 = input();
    if (x1 == 0) { printf("Detected divide-by-zero error!"); exit(1); }
    int y = 13 / x1;
    int x2 = input();
    if (x2 == 0) { printf("Detected divide-by-zero error!"); exit(1); }
    int z = 21 / x2;
    return 0;
}
```

Of course, there is nothing stopping us from encapsulating this repeated check into some function, call it `__dbz_sanitizer__`, for reuse.

```
void __dbz_sanitizer__(int divisor) {
    if (divisor == 0) {
        printf("Detected divide-by-zero error!");
        exit(1);
    }
}

int main() {
    int x1 = input();
    __dbz_sanitizer__(x1);
    int y = 13 / x1;
    int x2 = input();
    __dbz_sanitizer__(x2);
    int z = 21 / x2;
    return 0;
}
```

```
}
```

We have transformed our unsafe version of the code in the first example to a safe one by instrumenting all division instructions with some code that performs a divisor check. In this lab, you will automate this process at the LLVM IR level using an LLVM compiler pass.

Debug Location Primer. When you compile C code with the `-g` option, LLVM will include debug information for LLVM IR instructions. Using the aforementioned instrumentation techniques, your LLVM pass can gather this debug information for an `Instruction`, and forward it to `__dbz_sanitizer__` to report the location a divide-by-zero error occurs. We will discuss the specifics of this interface in the following sections.

Instrumentation Pass. We have provided a framework from which to build your LLVM instrumentation pass. You will need to edit the `part1/src/Instrument.cpp` file to implement your divide-by-zero sanitizer, as well as the code coverage analysis `part1/lib/runtime.c` contains functions that you will use in your lab:

- `void __dbz_sanitizer__(int divisor, int line, int col)`
 - Output an error for `line:col` if `divisor` is 0
- `void __coverage__(int line, int col)`
 - Append coverage information for `line:col` in a file for the current executing process

As you will create a runtime sanitizer, your dynamic analysis pass should instrument the code with these functions. In particular, you will modify the `runOnFunction` method in `Instrument.cpp` to perform this instrumentation for all LLVM instructions encountered inside a function.

Note that our `runOnFunction` method returns true in Lab 1. In Lab 0, we returned false in similar places. As we are instrumenting the input code with additional functionality, we return true to indicate that the pass modifies, or *transforms* the source code it traverses over.

In short, part 1 consists of the following tasks:

1. Implement the `instrumentSanitizer` function to insert a `__dbz_sanitizer__` check for a supplied `Instruction`
2. Modify `runOnFunction` to instrument all *signed and unsigned integer division* instructions with the sanitizer for a block of code

3. Implement the `instrumentCoverage` function to insert `__coverage__` checks for all debug locations
4. Modify `runOnFunction` to instrument all instructions with the coverage check

Inserting Instructions into LLVM code. By now you are familiar with the `BasicBlock` and `Instruction` classes and working with LLVM instructions in general. For this lab you will need to use the LLVM API to insert additional instructions into the code when traversing a `BasicBlock`. [There are many ways to traverse programs in LLVM](#). One common pattern when working with LLVM is to create a new instruction and insert it directly after some previous instruction.

For example, in the following code snippet:

```
Instruction* Pi = ...;
auto *NewInst = new Instruction(..., Pi);
```

A new instruction (`NewInst`) will get created and implicitly inserted before `Pi`; you need not do anything further with `NewInst`. Other subclasses of `Instruction` have similar methods for this - consider looking at `llvm::CallInst::Create`.

Loading C functions into LLVM. We have provided the auxiliary functions `__dbz__sanitizer` and `__coverage__` for you, but you have to insert them into the code as LLVM `Instructions`.

You can load a function into the `Module` with `Module::getOrInsertFunction`. `getOrInsertFunction` requires a string reference to a function to load, and a `FunctionType` that matches the function type of the actual function to be load (you will have to construct these items). It's up to you to take the loaded `Function` and invoke it with an LLVM `instruction`.

Debug Locations. As we alluded to in the primer, LLVM will store code location information of the *original C program* for LLVM instructions when compiled with `-g`. This is done through the [DebugLoc](#) class:

```
Instruction* I1 = ...;
DebugLoc &Debug = I1->getDebugLoc();
printf("Line No: %d\n", Debug.getLine());
```

You will need to gather and forward this information to the sanitizer functions. As a final hint, not every *single* LLVM instruction corresponds to a specific line in its source C code. You will

have to check which instructions have debug information. Use this to help build the code coverage metric instrumentation.

Example Input and Output

Your sanitizer should run on any C code that compiles to LLVM IR. For each test program, you will need to run the following command substituting your program name for `simple0`. (Note: there is also a Makefile provided for you that will run the commands for all the programs at once by executing the “make” command)

```
$ cd part1/test
$ clang -emit-llvm -S -fno-discard-value-names -c -o simple0.ll simple0.c -g
```

As we demonstrated in the Setup section, we will create an instrumented executable using your LLVM compiler pass. To test a different program, replace `simple0` with your program name.

```
$ opt -load ../build/InstrumentPass.so -Instrument -S simple0.ll -o
simple0.instrumented.ll
$ clang -o simple0 ../lib/runtime.c simple0.instrumented.ll
$ ./simple0
```

If there is a divide by zero error in the code, your code should output the following (recall the print statement is already set up in `/lib/runtime.c` - the line and column number will come from your code).

```
Divide-by-zero detected at line 27 and col 13
```

Code coverage information should be printed out in a file named `EXE.cov` where `EXE` is the name of the executable that is run (in the above case, look for `simple0.cov`.) Our auxiliary functions will handle the creation of the file; your instrumented code should populate it with line:col information:

```
25,7
25,7
26,7
26,11
26,7
27,7
27,11
27,15
```

Note that a correct solution will produce the exact same output as given.

Part 2 - Mutational Fuzzing

Setup

The skeleton code for Part 2 is located under `/fuzzing/part2`. We will refer to the top level directory for Part 2 as `part2` when describing file locations.

The following commands setup the lab:

```
$ cd part2
$ mkdir build
$ cd build
$ cmake ..
$ make
```

This time we will use the `fuzzer` tool to feed randomized input (that you will create) into compiled C programs that will run with a reference implementation of the sanitizer from Part 1:

```
$ cd part2/test
$ mkdir fuzz_output
$ ../build/fuzzer ./sanity fuzz_input fuzz_output MutationA
```

In the above command, `sanity` will receive input from `fuzzer` starting from an initial seed file at `part2/test/fuzz_input/seed.txt` located in `fuzz_input`. Cases that cause a program crash will get stored in `part2/test/fuzz_output/failure`. The starter code will guarantee that the fuzzer runs for a maximum of 10,000 iterations and exits as soon as a crash is found.

Format of Input Programs

All C programs that read from `stdin` are valid input programs.

Lab Instructions

In this lab, we will build a small fuzzer that feeds a C program inputs to try and cause it to crash. A full-fledged fuzzer consists of three key features: i) test case generation matching the grammar of the program input, ii) mutation strategies on test input to increase code coverage, iii) a feedback mechanism to help drive the types of mutation used. We will focus on the first two

features in this part of the lab, and restrict the testable C programs to those that read strings from the command line. In the next part, we will focus on the third feature.

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: (1) if the length of the program input is divisible by 13, and (2) 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 input.

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 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 various mutations on an input string, we ended up exhausting all program execution paths, i.e., more varied mutation in the input increased our code coverage. In its simplest form, this is exactly what a fuzzer does.

Building the Fuzzer. In this lab, you will implement three mutation functions, `mutateA`, `mutateB`, and `mutateC` in `part2/src/Part2Mutate.cpp` to perform some form of mutation on its input. You will decide how to implement each mutate function. There are many ways to go about this, but for starters, consider creating strings of different lengths. Additionally, you can look at some of the programs that we will run the fuzzer on, like `easy.c`, and see what input could cause a crash. If you can randomly generate such input, your `mutate` will likely cause other similar programs to crash.

Note each time you update `Part2Mutate.cpp` you will need to rerun the make command in the build directory to build your changes before testing.

After you have filled in your first mutate function, you should move on to create additional mutations. The specific mutation that will be run on the seed inputs is determined from the command line input. For example, `../build/fuzzer ./easy fuzz_input fuzz_output MutationA` will run your `mutateA` function on the input. Likewise, you can run `mutateB` and `mutateC` on the input by passing `MutationB` and `MutationC` instead of `MutationA`. The fuzzer will start by mutating the seed values based on the mutation you've selected on the command line. The mutated value will be run on the input program, and then inserted as an additional seed in the `SeedInput` vector which causes the mutated input to again be mutated. This process continues until the fuzzer discovers `maxCrashes` crashes or completes `maxTestsPerSeed` mutations. These values are defined in `part2/include/Utils.h`. The following code snippet illustrates this main loop:

```
std::string CampaignStr(argv[4]);
Campaign FuzzCampaign;
if (!toCampaign(CampaignStr, FuzzCampaign)) {
    return 1;
}

initialize(OutDir);

if (readSeedInputs(SeedInputDir)) {
    fprintf(stderr, "Cannot read seed input directory\n");
    return 1;
}

while (true) {
    for (int i = 0; i < SeedInputs.size(); i++) {
```

```

std::string currentSeed = SeedInputs[i];
std::string currentMutant = currentSeed;
for (int j = 0; j < maxTestsPerSeed; j++) {
    if (Count % Freq == 0) {
        std::cerr << "Count is " << Count << std::endl;
    }
    std::string newMutant = mutate(currentMutant, FuzzCampaign);
    test(Target, newMutant, CampaignStr, OutDir);
    if (failureCount >= maxCrashes) {
        break;
    }
    currentMutant = newMutant;
}
}

```

Fuzzing the Binaries. We have provided three binary programs `hidden1`, `hidden2`, and `hidden3`. These programs serve as more challenging test cases for your fuzzer. We will not provide the raw source; instead, your mutations should be versatile enough to find test cases that crash these programs. Try each of your three fuzzing strategies; it's likely that some mutation strategies won't find the bugs, and some will find them faster than others. All provided binaries have been instrumented with the coverage and divide-by-zero error detection familiar from Part 1.

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

Please note that these mutation ideas are basic suggestions. It is unlikely that using the most trivial version of any of these mutations will get you the results you need in the time required for this lab. Typically, when mutations are not effective enough, we see that students have taken a very conservative approach to a mutation, such as rotating a single character by a single value in an iteration. Try rotating more characters at once, or rotating your characters farther (advance by more than one value) per iteration. Feel free to play around with additional mutations, and see if you can speed up the search for bugs on the binaries.

Note that while feeding continually random information into the program does fall under the general category of fuzzing, in this lab, we want to work smarter than just using random inputs. We expect you to use *mutation strategies* to generate your input strings. Each input string should be based on a previous input string with changes. If your mutation does not take the previous

input into account, it is not a valid mutation strategy for this lab. Namely, you cannot ignore the input and replace it with random values.

In short, this part consists of the following tasks:

1. Decide on a mutation to implement from the aforementioned list of mutations.
2. Implement your mutation in one of the templated mutate functions (`mutateA`, `mutateB`, or `mutateC`).
3. Repeat 1-2 for the remaining mutation functions.
4. Test the strength of your fuzzer by trying to crash the three binaries, `hidden1`, `hidden2`, and `hidden3`.

Example Input and Output

Your fuzzer should run on any valid C code that accepts standard input:

```
$ cd part2/test
$ ../build/fuzzer ./sanity fuzz_input fuzz_output MutationA
```

Expect `fuzz_output` to get populated with several files that contain the random input that caused the crash, e.g,

```
fuzz_output/failure/input1-MutationA
fuzz_output/failure/input2-MutationA
...
fuzz_output/failure/inputN-MutationA
```

with N being the last case that caused a crash before the timeout. As you can see, the name of the mutation that was used to cause the crash is included in the filename.

As a sanity check, you can feed these inputs directly into the program at test:

```
$ cd part2/test
$ cat fuzz_output/failure/input1-MutationA | ./sanity
```

Expect to see `Divide-by-zero detected at line 12 and col 13`. `sanity` happens to fail on all input, and is a good sanity check, but for a slightly more challenging test case, try `easy`.

Part 3 - Feedback-Directed Fuzzing

Setup

The skeleton code for Part 3 is located under `/fuzzing/part3`. We will refer to the top level directory for Part 3 as `part3` when describing file locations.

This lab builds off the work you did in Part 2.

Copy your `Part2Mutate.cpp` from `fuzzing/part2/src` to `part3/src` as `Part3Mutate.cpp`.

```
$ cp ~/fuzzing/part2/src/Part2Mutate.cpp ~/fuzzing/part3/src/Part3Mutate.cpp
```

The following commands setup the lab:

```
$ cd part3
$ mkdir build
$ cd build
$ cmake ..
$ make
```

We will again use our `fuzzer` tool to feed randomized input (that you will create) into compiled C programs that will run with your sanitizer:

```
$ cd part3/test
$ mkdir fuzz_output
$ ../build/fuzzer ./sanity fuzz_input fuzz_output
```

Except this time, you will use information about the program run, specifically, coverage information, to drive the fuzzing.

Format of Input Programs

All C programs that read from `stdin` are valid input programs for this lab.

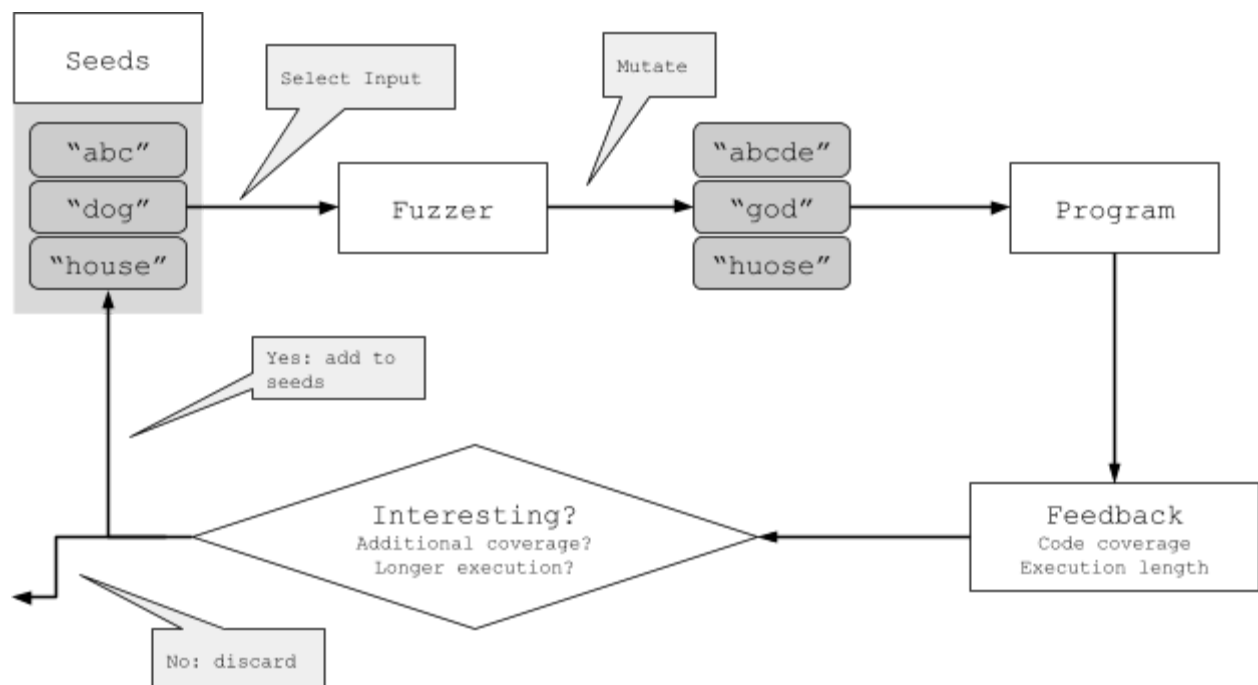
Lab Instructions

In the previous section, we built a small fuzzer to automate the testing of C code. Recall that a full-fledged fuzzer consists of three key features: i) test case generation matching the grammar of the program input, ii) mutation strategies on test input to increase code coverage, iii) a feedback

mechanism to help drive the types of mutation used. We will now extend our fuzzer with the third feature to build feedback-directed fuzzing.

Feedback-Directed Fuzzing. Thus far, we have seen how randomized testing can find bugs and is a useful software analysis tool; however, we currently brute force generate test cases which results in a lot of redundant testing.

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.

Extending the Fuzzer. Thus far, you defined three generic mutation functions, `mutateA`, `mutateB`, and `mutateC`, to fuzz a given program. We iteratively applied a selected mutation to a set of `SeedInputs`. As you saw, this strategy worked well for some programs, and performed poorly on others. In this lab, you will guide the direction of the fuzzing using feedback metrics from a program's execution.

We have adjusted the scaffolding in Part 2 to give you an idea of how to structure your implementation. Previously, we only cared about performing the fuzzing for a selected `Campaign`; now we wish to use each `Campaign` to guide our search. As such, it may help to track separate seeds for each `Campaign`. We now maintain a mapping from a `Campaign` to a list of seeds for that campaign:

```
std::map<Campaign, std::vector<std::string>> SeedInputs;
```

In addition, we have rewritten the main run loop of the fuzzer to better fit the feedback-directed pattern shown in the figure at the start of the section. Note the expression used to terminate the while loop. The fuzzer will exit after executing `maxTests` tests or finding `maxCrashes` crashes, whichever comes first.

```
while (Count < maxTests && failureCount < maxCrashes) {  
    // NOTE: You should feel free to manipulate this run loop  
    std::pair<std::string, Campaign> SC = selectSeedAndCampaign();  
    auto Mutant = mutate(SC.first, SC.second);  
    auto Success = test(Target, Mutant, SC.second, OutDir);  
    updateSeedInputs(Target, Mutant, SC.second, Success);  
}
```

Here we select a seed and a fuzzing campaign to mutate the seed, perform the mutation, run the program on the resulting mutant, and then decide how we wish to update the set of seed inputs. The specifics are left to you; moreover, you should consider the structure as a guide and modify the scaffolding as needed. Be creative: we will test your fuzzer for performance as well as correctness.

In short, the lab consists of the following tasks:

1. Select a seed and fuzzing campaign to generate the next test case. You may follow our groundwork and fill in `selectSeedAndCampaign`.
2. Decide whether the mutation was *interesting* based on success or failure of the program, code coverage, and any additional metrics you feel will increase the performance of your fuzzer. Again, you may follow our groundwork and fill in `updateSeedInputs`.
3. Insert the `Mutant` into the pool of `SeedInput` to drive further mutation.

Code Coverage Metric. Recall that you have a way of checking how much of a particular program gets executed with the coverage information output by your sanitizer. A `.cov` file will get generated from your sanitizer in the working directory for the program that is getting fuzzed. For example, running:

```
$ ../build/fuzzer ./path fuzz_input fuzz_output
```

Will create and update `path.cov` in the working directory that your `updateSeedInputs` function should read and use between each test.

A Few More Hints. Do not be afraid to keep state between rounds of the fuzzing. You may want to try each of your fuzzing campaigns initially to see which generate a test that increases code coverage, and then exploit that campaign. Unlike your implementation in Part 2, we expect this fuzzer to be versatile; it should generate crashing tests on all inputs.

Example Input and Output

Your fuzzer should run on any valid C code that accepts standard input:

```
$ cd part3/test
$ ../build/fuzzer ./path fuzz_input fuzz_output
```

Expect `fuzz_output` to get populated with several files that contain the random input that caused the crash, e.g,

```
fuzz_output/failure/input1-MutationA
fuzz_output/failure/input2-MutationA
...
fuzz_output/failure/inputN-MutationA
```

with N being the last case that caused a crash before the timeout. As you can see, the name of the mutation that was used to cause the crash is included in the filename. We will also output test cases that resulted in a successful program run under `fuzz_output/success/`; however, since it's likely you will get many successful program runs before a failure, we write every 1000th successful run. You can adjust this frequency by supplying an additional optional argument to the fuzzer:

```
$ ../build/fuzzer ./path fuzz_input fuzz_output 100
```

Grading

For each part, your solution will be tested against the provided test cases as well as some additional hidden test cases that are similar to the provided ones.

Items to Submit

For this lab, we will be using Gradescope to submit and grade your assignment. For full credit, the files must be properly named and there must not be any subfolders or additional files or folders. Submit the following files through Gradescope.

- Part 1 - Submit file `Instrument.cpp` (50% of grade)
- Part 2 - Submit file `Part2Mutate.cpp` (25% of grade)
- Part 3 - Submit files `Part3Mutate.cpp` and `Part3Fuzzer.cpp` (25% of grade total)
 - Note: If you did Part 3 without changing `Part2Mutate.cpp` from Part 2, you must still submit `Part3Mutate.cpp` - just submit a duplicate of your Part2 version.

Do not submit anything else. Make sure all of your code modifications are in the files listed above as your other files will not be submitted. In particular, past students have modified header files to import additional headers, leading to a compile error in their submission.

Through Gradescope, you will have immediate feedback from the autograder as to whether the submission was structured correctly as well as verification that your code successfully runs the tests on the grading machine. For Lab1, Gradescope will provide a score that includes all tests released with the project starter, and additional hidden tests.

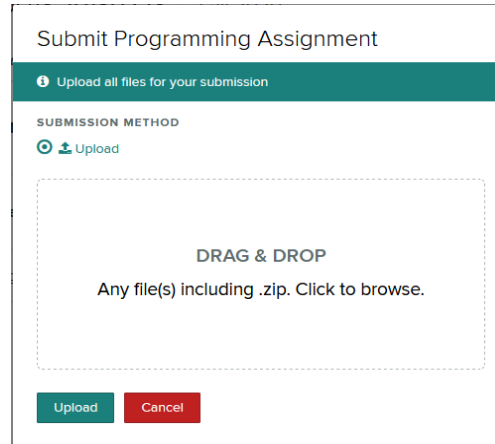
How Do We Use It?

Since the technique was [first published in 1989](#), thousands of software vulnerabilities have been found through fuzzing. Of note, the 2014 [Shellshock](#) security flaw was [discovered](#) using the [American Fuzzy Lop \(AFL\)](#) fuzzer. Both Google and Microsoft [continuously fuzz their web browsers](#) (see page 148).

If you want to use a well-established fuzzer on your software, there are lots of great options today such as [Microsoft OneFuzz](#), CERT [Basic Fuzzing Framework \(BFF\)](#), and [AFL](#) (originally from Google).

Tips for submitting your assignment with Gradescope

- When you go to the Canvas Assignment, you will see a blank Gradescope submission window embedded in the Canvas Page. Follow the on-screen prompt to submit your files.



Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

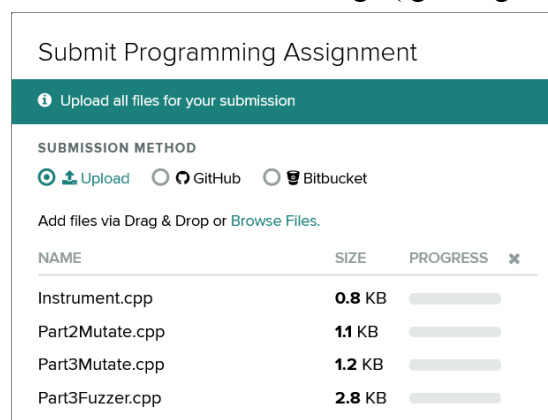
Upload

DRAG & DROP

Any file(s) including .zip. Click to browse.

Upload Cancel

- If there are multiple files in this lab, you can zip the files together and upload the single zip file to the submittal page, or you may select each file individually on the page. If you select the files individually, they must all be uploaded in the same submission.
- A correct submission should look like this image (ignoring size and order):



Submit Programming Assignment

Upload all files for your submission

SUBMISSION METHOD

Upload GitHub Bitbucket

Add files via Drag & Drop or [Browse Files](#).

NAME	SIZE	PROGRESS	✕
Instrument.cpp	0.8 KB	<div></div>	
Part2Mutate.cpp	1.1 KB	<div></div>	
Part3Mutate.cpp	1.2 KB	<div></div>	
Part3Fuzzer.cpp	2.8 KB	<div></div>	

- You may resubmit your assignment as many times as you want up to the assignment due date. The **Active** submission will be the final grade. By default, every time you upload a new submission, it will become the **Active** submission, but you can change the **Active** submission in your **Submission History** window if needed.
- If you resubmit your assignment, you must include all files as part of the new submissions.
- We provide comments as output of your submission. Use these to make sure your submission is as complete as possible. For example, if you see a compile error, you will not receive any points for that part of the assignment.