Introduction to Software Analysis

Understanding basic concepts of software analysis and metrics used to estimate the effectiveness of analysis.

Objective

The objective of this lab is to use standard analysis tools for static and dynamic analysis on C programs to discover divide-by-zero errors and interpreting their results to better understand various trade-offs between the techniques.

Pre-Requisites

The lectures introduce various terminology used throughout this lab such as: static and dynamic analysis, soundness, completeness, precision, and more.

Setup

Step 1.

The skeleton code for Lab 1 is located under `/lab1`. We will refer to this top-level directory for Lab 1 simply as `lab1` when describing file locations for the lab.

Step 2.

Throughout the labs, we will use `CMake`, a modern tool for managing the build process. If you're unfamiliar with `CMake` we recommend reading the CMake tutorial (especially pay attention to Step 1 and Step 2 in the tutorial). Running `cmake` produces a `Makefile` that you might be more familiar with. If you are not familiar with `Make`, read either the Makefile tutorial or Learn Make in Y minutes first, and then peruse file `lab1/Makefile`. Ensure that you are comfortable with using `Makefile` in this lab.

Step 3.

Inspect the Makefile to see the commands used to run AFL and Clang Static Analyzer (CSA).

```
# Compile the program with AFL
AFL_DONT_OPTIMIZE=1 afl-gcc c_programs/test1.c -o test1
# Run AFL for 30s on test1
timeout 30s afl-fuzz -i afl_input -o afl_output test1
# Run CSA on test1.c
clang -v --analyze c_programs/test1.c
```

Lab Instructions

In this lab, you will run two analysis tools on a suite of C programs, study the tools' results, and report your findings.

Step 1.

Run the provided analysis tools, AFL and CSA, on all C programs located under the `lab1/c_programs` directory. To do so, simply run the following command, which first runs AFL with a timeout of 30 seconds per program, and then runs CSA.

```
/lab1$ make check_versions && make all
```

On running the above command, you will see the output that looks something like this:

```
AFL_DONT_OPTIMIZE=1 afl-gcc c_programs/test1.c -o ...
timeout 30s afl-fuzz -i afl_input -o ...
Makefile:8: recipe for target 'results/afl_logs/test1/out.txt' failed
make: [results/afl_logs/test1/out.txt] Error 124 (ignored)
clang -v --analyze c_programs/test1.c ...
...
```

Ignore the error reported by Make above; it is normal because AFL keeps running until it is forcibly terminated by the timeout command. Feel free to experiment by changing the timeout which is set to 30 seconds. We do not expect everyone to report the same solutions since AFL is non-deterministic anyway.

On running the make command, the following files and directories should be generated under the `lab1/results` directory:

```
    ├── afl_logs/
    │   ├── test0/
    │   │   ├── out.txt
    │   │   ├── afl_output/
    │   │   └── test0
    │   ├── ... // similar for test1
    │   ...
    │
    └── csa_logs/
        ├── test0_out.txt
        ├── ... // similar for test1
        ...
```

Step 2.

Determine the ground truth (`right` vs. `wrong`) of the C programs with respect to division-by-zero errors. In particular, for each division instruction in each program, determine by inspecting the program whether it can result in a division-by-zero error on some program inputs. Write your answers in file `lab1/answers.txt` in the “ground truth” column of the table for each test.

Step 3.

Study the output of AFL and CSA and determine if they accept or reject each program. Fill in your answers in file `lab1/answers.txt` in the corresponding columns of the table for each test program.

The crashing inputs discovered by AFL are stored in separate files under directories `lab1/results/afl_logs/<test-name>/afl_output/crashes/`. The files have idiosyncratic names of the form `id:000000,sig:08,src:000000,op:arith8,pos:2,val:-8`.[^1]. It is the contents of these files that AFL used as the input to the test program when it encountered a crash.

Upon examining CSA's outputs, if a `core.DivideZero` warning emerges, it indicates that CSA has detected a division by zero error for that particular test case.

For example:

```
c_programs/test9.c:10:17: warning: Division by zero [core.DivideZero]
   10 |    int avg = sum / len;
      |              ~~~~^~~~~
```

However, it's worth noting that not every test case will trigger this warning.

Step 4.

Using your entries from Steps 2 and 3, calculate the Precision, Recall, and F1 Score of each column. Enter them in the corresponding rows in `lab1/answers.txt`.

Step 5.

Answer the questions in `answers.txt` with the help of the table you filled in.

Submission

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

```
lab1$ make submit
...
submission.zip created successfully.
```

Then send the `submission.zip` file to TA's email.

[^1]: The file name encodes various things such as the ID of the crashing input, the crashing signal, the non-crashing seed input from which this crashing input was produced -- which in our case is always the file lab1/afl_input/seed.txt, and the operations by which the non-crashing seed input was transformed into this crashing input.