Software Fuzzing & Testing

Design Document

February 2025

Supervisor: Chris Meudec
Student: Jack Foley (C00274246)
A tool used for testing software applications using diverse inputs.

1 Introduction

A fuzzer is a tool used in software testing that will automatically create and input a large amount of random, unusual or unexpected data (called fuzz) into a program. The goal of a fuzzer is to find as many assert violations, unexpected exceptions, vulnerabilities, crashes and/or other issues that regular unit testing, coverage testing, condition testing, etc. will not normally find. It can help discover vulnerabilities and other weaknesses in software.

Since this project is entirely used as a command line tool, there is no user interface to show. Therefore I will instead give brief overviews of the functions used inside the fuzzer.

1.2 Purpose of Document

The purpose of the design document is to outline the internal structure and functionality of the fuzzing tool. It will assist in the planning of the project by outlining and defining time estimates and estimating effort required for the project. The document will describe any interactions between any form of front end and back end as well as other situations such as file I/O, compilation, etc.

2 Technologies

2.1 C

The main programming language that is being used for this project is the C programming language. It is a high level, general purpose language that boasts extreme efficiency and speed if used correctly. I chose this language because of the efficiency and also because the benchmarks that the fuzzer will be used on are also written in C, so it seems sensible to use it knowing this.

2.2 TestCov (No longer in use)

TestCov is a testing framework made by SoSy-Lab, who also made the benchmarks (Test-Comp) for the fuzzing tasks. Since I am creating a coverage-based fuzzer, it makes sense to include a coverage framework from the creators of the benchmark itself.

3 Code Overview

3.1 fuzz.c

Fuzz.c is the main file for producing the fuzzing inputs for the target files. It has numerous different input generation techniques that can be used. The current functions are as follows:

- `generateRandomNumber()`
- `generateSequence()`

- `generateRandomString()`
- `generateMutatedString()`
- `flipBitInString()`
- `insertCharIntoString()`
- `removeCharFromString()`

It also has implementations for the specific VERIFIER functions that are used in the benchmarking test from TestComp. At the moment, the only one is __VERIFIER_nondet_int(), which returns a random, non-deterministic integer. The current implementation just delegates the task to `generateRandomNumber()`

3.2 io.c

Io.c is the main file for handling file operations for the fuzzer, such as creating new folders, files and writing to files. It only has a handful of functions as of now.

- `closeFile(FILE* file)`
- `openFile(const char* filename)`
- `writeStringToFile(FILE* file, const char* string)`
- `openFile(const char* fileName)`
- `createFolder(const char* basePath, const char* folderName)`
- `getFullPath(const char* filePath)`
- `getCurrentTime(void)`
- `getHash(const char* filePath)`

3.3 lex.c

This file is used to initialise the lexical analyser and to call it when it is needed. This is used to attempt to discover a valid input range for the fuzzer for it to use during runtime.

- `generateLexer()`
- `lexScanFile(const char* filename)`
- `extractInputRange(const char* filename)`

3.4 testcase.c

This file is used to create a test case that will be then saved as a file in the respective test suite. These are required for TestCov to produce code coverage reports.

- `createTestSuiteFolder(const char* filename)`
- `getCurrentTime()`
- `createMetadataFolder(const char* filePath, const char* filename)`
- `createTestSuiteAndMetadata()`

- `createTestInputFile(const int* inputs, size_t num_inputs, const char* test_suite)`

3.5 target.c

This file will be used to perform dynamic compilation and execution of the target file via system calls.

- `compileTargetFile(const char* filePath)`
- `executeTargetInt(int input)`
- `cleanupTarget()`

3.6 range.c

This file is just used to store a min and max range that can be used across different files.
It contains:

```
int minRange
int maxRange
```

3.7 generational.c

This file will be used to handle generational related code. It manages population sizes, the amount of generations, mutation rates and tournament sizes.

- `calculatePlaceholderFitness(int input_value, int min_range, int max_range)`
- `selectParent(Individual population[], int population_size)`
- `mutateInteger(int original_value, int min_range, int max_range)`
- `getNextPopulationIndex(int population_size)`
- `generateNewPopulation(Individual population[], int population_size, Individual next_generation[], int min_range, int max_range)`
- `initializePopulations(void)`
- `cleanupPopulations(void)`

3.8 corpus.c

This file will be used to handle any work to do with the corpus, a large text document used to store the inputs for fuzzing.

- `createCorpus(char* path);`
- `selectRandomValueFromCorpus(char* path);`

- `addValueToCorpus(char* path, int value);`
- `removeValueFromCorpus(char* path, int value);`
- `validateCorpusContent(char* path);`

3.9 logger.c

This file is a basic utility file for logging purposes. It can be used anywhere in the project.

- `app_log(const char* level, const char* message);`
- `app_log_with_value(const char* level, const char* message, const char* format, ...);`