Software Fuzzing & Testing

Functional Specification

December 2024

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.

Although many fuzzers are focused on finding the bugs and crashes I mentioned above, the fuzzer that I will be making is mostly a coverage based fuzzer with the sole purpose of obtaining the largest amount of code coverage on a problem file as possible. This fuzzer will intelligently introduce new input based on previous results via a ‘generational algorithm.’

1.1 Purpose of Specification

This specification is designed to outline all of the functional and non-functional requirements that will be in the software. Since the concept of a fuzzer is a single CLI tool with a single task, the functional and non-functional requirements of this software will be relatively short and simple.

2 Problem Statement

Creating tests for software testing is a slow but necessary process. It is common for software engineers and developers to dislike the process of writing tests and achieve sufficient code coverage when they would rather be writing useful code. A fuzzer is a tool that they can use to automate the process of testing their code by generating random inputs to try and break the software in unexpected ways. The fuzzer serves as an alternative, but not absolute, solution to easy software testing.

Another issue is that achieving deep code coverage through manual testing is a labour intensive and time consuming process. It is common that developers may struggle to reach all execution paths in a file, thus leading to untested code that could introduce potential issues further down the line. This fuzzer will automate the process of increasing code coverage through efficient testing methods such as generational algorithms.

3 Project Scope

3.1 Goal

The main goal of a software fuzzer would be to, by itself, find vulnerabilities, edge cases, and surprising behaviours of the target software by generating a large number of random, malformed, or semi-valid inputs. The fuzzer's aim is to improve the robustness, security, and stability of the software by catching issues like crashes, memory corruption, buffer overflows, or any security flaw at an early stage in the development life cycle. This reduces the risk of these vulnerabilities being exploited in production. It should integrate easily into the testing pipeline and provide actionable insights for remediation by developers and security teams.

3.2 Target Audience

The use of a fuzzer is not limited to one or two target audiences. It can be narrowed down into a few categories. It will mostly suit those who wish to achieve higher code coverage on their software.

QA Engineers / Test Engineers: Those involved in software robustness testing to ensure the application does not crash or misbehave in cases of unexpected or invalid inputs. They increase coverage with fuzzers and automate stress tests.

Software Developers: The developers integrate fuzzing tools into their test frameworks to find those bugs or edge cases that other forms of testing may not be able to locate. This demographic is particularly interested in the early detection of vulnerabilities within the development phase.

Some roles may use a fuzzer as much, especially if it is a coverage based fuzzer, such as the following.

Security researchers: Professionals who focus on discovering vulnerabilities within software applications. They make use of fuzzers, which allow the automation of detecting security inefficiencies that may include buffer overflows and memory leaks in software applications.

Penetration Testers: Ethical hackers, normally assess the security posture of systems through simulating cyberattacks. They use fuzzers to automate the process of generating malformed inputs to study the responses of a system or application to such inputs.

Compliance Teams: These teams are in charge of ensuring the software matches specific criteria on security and robustness. To ensure that the software aligns with applicable regulations, standards, or certifications, they use fuzzers.

4 Requirements

4.1 Functionality

Platform Compatibility

Since it depends on the GNU C library, the fuzzer must be run on a Unix-like system, for instance, on Linux or macOS. Other operating systems, such as Windows, are not supported simply because they do not provide native support for the GNU C library or system calls originally from Unix.

Language and Library

This will be implemented in the C programming language. Besides that, the GNU C library as a standard library will be used to enable low-level system calls. The reason behind choosing C and the GNU C library is that they offer advantages regarding performance and thus so far form the basis for efficient system-level tool development under Unix variants. Similarly, the presence of the GNU C library within an implementation allows for proficient interaction with the Unix kernel, where memory management, file input/output, and threading operations take place-a necessary requirement for lots of operations of the fuzzer.

Unix Compatibility

The fuzzer is incompatible with non-Unix systems due to the inexistence of equivalents for the GNU C library and its system calls in the Windows world, for instance. In case a Windows-based environment has to be supported, considerable porting effort would be needed, including using alternative libraries such as mingw or cygwin, which is beyond the scope of this project.

Code Coverage Maximisation

It should be able to generate new inputs based on the previous results from other code coverage reports in an effort to increase code coverage for future tests. It should be able to generate a diverse range of inputs to ensure maximal code coverage is being achieved.

4.2 Usability

User Interface and Usability

The fuzzer should take a problem file as an input and it should run fuzzing test input on this file, producing code coverage reports on the file and displaying the output as a graph showing coverage percentage over numerous different tests.

Dependencies:
GNU C library: This is used for system calls, process control, memory management, and threading for Unix-based systems.
C Compiler (gcc): In order to compile the fuzzer. Unix-based environment.

4.3 Reliability

The fuzzer should be able to find defects, crashes, and vulnerabilities in target software in some cases, given enough time. Since the nature of a fuzzer is random and non-deterministic, it could detect an issue immediately one time but take an hour to detect the same bug the next time, but the main point is that it still detects it. The tool should reduce the number of false positives by verifying every potential bug against known patterns or re-running test cases to confirm reproduction.

The fuzzer shall handle some of the errors occurring in runtime of its execution, for example, crashes of the target program or memory overflows; it should never crash itself.
Any test results found in failures or anomalies should be kept in a reliable and durable format, such that if anything happens to any system or tool crashing, information is not lost.
It would mean that the fuzzer, in the case of any target program failure, has to keep on executing any subsequent case of input without disruption to the fuzzing process.

The fuzzer should be able to improve upon its code coverage score incrementally as it uses generational algorithms to create better inputs based on older code coverage results.
The fuzzer should perform reasonably well within the results of Test-Comp. This will act as a way to measure the effectiveness of the tool vs. other fuzzing tools, such as American Fuzzy Lop (AFL) [1], a “state of the art” fuzzer.

4.4 Performance

Fine-tuning is required for a fuzzer to be effective, meaning generating test cases and processing the results as fast as possible to extend input space coverage for a given, usually quite limited, period of time.

It should be able to use system resources correctly: optimise CPU and memory in such a way that it could run several on multi-core and grow by using multi-threading or distribution.
A performance benchmark needs to be set. For instance, the number of test cases it can generate in one second or the maximum time taken to execute on any given input.

The generation of inputs should focus on edge cases, try to maximise the possibilities of generating hard-to-detect bugs, without overwhelming the system with inputs it will not use.
It should also not be hungry for memory or CPU, lest it interfere with the execution of the subject software, or other uses from the system.

Since the fuzzer will also be generating test case files for TestCov, this is a major bottleneck that needs to be considered. It is a possibility that this work could be handled by a separate thread while the fuzzer continues to run on its own thread, but it could pose some potentials issues with timing.

4.5 Supportability

It should be maintainable, easy to understand and modular so any developer can contribute by adding the functionality of new input formats or a test case generation strategy.
It will be fully documented on setup and usage, supported arguments and configuration options, and how to read the output.

4.6 Approach

The approach that I aim to take with this fuzzer is to use a combination of random fuzzing, mutation based fuzzing and coverage based fuzzing. These all fall under the category of either black-box or grey-box fuzzing, which are two of the three major types of fuzzing, with the other one being white-box fuzzing.

The fuzzer should be able to perform static analysis on a target file in an attempt to find valid inputs and then, using those inputs, produce test cases for the file and produce code coverage reports. It should then be using those reports to create better, valid inputs to enhance code coverage further.

References

1. Wikipedia. (2024). *American Fuzzy Lop (software)*. [online] Available at: https://en.wikipedia.org/wiki/American_Fuzzy_Lop_(software) [Accessed 25 Oct. 2024].