Mutation-Based Fuzzer

Carolina Rojas Matos

School of Computing and Augmented Intelligence, Ira A. Fulton Schools of Engineering
Arizona State University

699 S Mill Ave, Tempe, AZ 85281, USA
crojasma@asu.edu

***Abstract*— Fuzz testing is a software testing technique that involves sending random or invalid input data to a program in order to uncover errors or vulnerabilities. The purpose of this project is to develop a Binary Fuzzer and utilize fuzz testing to test the ability of a program to handle unexpected or malformed input, which can help to identify bugs, crashes, or security vulnerabilities. To achieve this, large quantities of input data are generated using seeded random data generation, and send it to the target program. By analysing the results, potential issues can be identified in the program and make improvements to enhance its security and reliability.**

I. Introduction

Software testing is a crucial phase in the software development lifecycle that ensures the quality, reliability and functionality of a software system to identify any defects, bugs or vulnerabilities that may affect its performance or compromise its security.

One approach to software testing is fuzz testing, also known as fuzzing. Fuzz testing is a dynamic testing technique that aims to discover software vulnerabilities and unexpected behaviors by providing invalid, unexpected or random inputs to the target system [1].

Fuzz testing is particularly effective in identifying security vulnerabilities such as buffer overflows, format string vulnerabilities, injection flaws, and more.

A. Operation of a Mutative-Based Fuzzer

The main principle behind a Mutative-Based Fuzzer is to automatically generate a large number of test cases by applying random or semi-random mutations to the input data of a target program or system. These mutations can include changes to input values, files formats, network protocols, or command structures.

By feeding these mutated inputs into the target system, the Fuzzer aims to trigger unexpected behaviors that may have otherwise gone unnoticed.

The effectiveness of a Mutative-Based Fuzzer lies in its ability to explore a wide range of inputs and corner cases, allowing it to undercover potential security issues in the target system. It is commonly used by security researches, software developers and penetration testers as part of their testing and vulnerability assessment efforts [2].

B. Fuzz Testing in The Field

Fuzzing has made significant contributions to the field of software testing, particularly in the area of security testing. Here are some key contributions of fuzz testing:

- Vulnerability Discovery.
- Automation and Scalability by generating a vast number of test cases quickly.
- Finding Unknown Unknowns, the unexpected inputs given to the target system enables the discovery of novel vulnerabilities that might have been missed through manual testing or static analysis.
- Cost-Effectiveness.
- Improved Software Quality.

Overall, fuzz testing has revolutionized the field of software testing, providing an automates, scalable, and effective method for discovering vulnerabilities.

C. Implementing and Applying a Fuzzer for Sofware Testing

For this project, the main objective is to implement a fuzzer, but due to the lack of time, the implementation of a “Dumb Fuzzer” or Randomized Fuzzer was recommended. This is because, the implementation of a Smart Fuzzer would imply a deeper knowledge of the system that would be fuzzed and this would require more sophisticated implementation, configuration and resources. Smart Fuzzers are suitable for in-depth vulnerability analysis, undercovering complex bugs, and achieving higher code coverage [3].

On the other hand, Randomized Fuzzers, also known as black box or Blind Fuzzers, are a type of fuzz testing that generates inputs randomly with minimal knowledge of the target software. Despite its simplicity, Randomized Fuzzers can still offer certain advantages in the context of software testing and vulnerability discovery [4], such as:

- Rapid Testing: Randomized Fuzzers can generate a large number of test input quickly. By focusing on random mutations and broad input coverage, they can rapidly explore different code paths and identify crashes or unexpected behaviors.
- Wide Input Space Coverage: Randomized Fuzzers tend to provide good coverage od the input space, as their random input generation may uncover rare or edge-case scenarios that were not anticipated.
- General Vulnerability Discovery: While Randomized Fuzzers lack intelligence about the target software, they can still be effective in discovering vulnerabilities resulting from input validation or memory corruption issued.
- Complement to Other Testing Techniques: Randomized Fuzzers can be used in combination with manual code reviews and static analysis to

enhance overall test coverage and increase the likelihood of vulnerability discovery.

- Early Bug Detection: Even though Randomized Fuzzers may not provide detailed analysis or specific guidance, they can still help identify crashes by detecting issues in the early software development cycle.
- Simplicity: Randomized Fuzzers are relatively easy to implement compared to more advanced fuzzing techniques.

II. Methodology to develop a Random Fuzzer

A. *Understanding Pseudo-Random Number Generator*

Prior to introducing the steps taken to develop a Randomized Fuzzer, it is necessary to highlight the importance of understanding Pseudo-Random Number Generators (PRNGs) in the implementation of a Mutative-Based Fuzzer. An understanding of PRNGs is crucial due to their central role in generating the random inputs required by the fuzzer.

A Mutative-Based Fuzzer heavily relies on PRNGs to introduce variations and mutations into the input data. By gaining a comprehensive understanding of PRNGs, including their seed initialization, state management, and number generation algorithms, fuzzer developers can ensure that the generated mutations exhibit the desired properties of randomness, distribution, and independence.

In order to provide context, it is appropriate to provide a concise definition of a PRNG. A Pseudo-Random Number Generator is an algorithm that develops a series of numbers that closely approximates true randomness. Unlike truly random numbers that arise from unpredictable natural processes, PRNGs generate numbers deterministically based on an initial seed value. The following is a general overview of how a PRNG operates [5]:

1) *Seed Initialization:* The PRNG begins with an initial seed value, which can be a fixed value or obtained from an external source such as system time, user input, or hardware randomness. The seed determines the starting point of the PRNG's sequence.

2) *State and Sequence Generation:* The PRNG maintains an internal state, which is updated each time a new random number is generated. The state is used as the basis for producing subsequent numbers in the sequence.

3) *Number Generation:* When a random number is requested, the PRNG applies mathematical operations or algorithms to transform the current state into the next number in the sequence. These operations are designed to introduce randomness and ensure statistical properties, such as uniform distribution or independence.

4) *Periodicity and Repetition:* PRNGs have a limited period, which is the number of unique values that can be generated before the sequence starts repeating. Eventually, the PRNG will cycle back to a previously generated state and repeat the sequence from that point onwards. The period length depends on the specific PRNG algorithm and the size of the internal state.

5) *Seed Management:* To avoid predictable sequences and repetition, it's crucial to manage the seed value. Using a fixed seed or reusing the same seed can result in identical sequences being generated. It is recommended to seed the PRNG with a value that changes over time or is obtained from a source with sufficient entropy [6].

It is important to note that PRNGs are not inherently random, as their outputs are deterministic and predictable based on the seed value. However, high-quality PRNGs aim to generate sequences that demonstrate characteristics of randomness, including uniformity, independence, and unpredictability.

When utilizing PRNGs in security-critical applications, caution must be exercised to prevent the exploitation of predictable or biased sequences by potential attackers. In such scenarios, cryptographically secure random number generators (CSPRNGs) are employed. These generators meet stringent criteria for randomness and unpredictability, ensuring the security of cryptographic systems and protocols.

For this specific project, the programming language chosen was GoLang, and the implementation incorporated the math/rand package. This package provides Pseudo-Random Number Generators (PRNGs) that are suitable for developing a Randomized Fuzzer.

B. *Fuzzing Technique*

This section presents the development process of a Randomized Fuzzer, which aims to generate mutated inputs for the fuzzing target. The fuzzer operates based on a step-by-step procedure, as outlined below:

1) *Fuzzer Configuration*

- The fuzzer is designed to accept an initial seed and two arguments, namely `‘prng_seed’` and `‘num_of_iterations’`, as input.
- Both arguments are 32-bit integers where `‘prng_seed’` is used to feed or seed the Pseudo-Random Number Generator function, to ensure deterministic output, and `‘num_of_iterations’` determines the total number of loops cycles the fuzzer will execute.

2) *Input Generation*

- The fuzzer reads an initial seed file named *“seed”* which should be located in the current working directory.
- Optionally, if applicable, the fuzzer utilizes `‘prng_seed’` to seed any PRNG(s) employed during the fuzzing process.
- Note that, in the delivered project, as a way to enhance the functionality of the program, if the *“seed”* file is not found, then the program proceeds to use a default seed string taken from one of the seed files provided by the course staff.

3) *Iteration Process*

- The fuzzer will operate within a main loop that repeats `'num_of_iterations'` times. During each iteration, there is a 13% probability that a byte of the input data will be modified. This probability is determined by generating a random number within the range of 0 to 100 and checking if it is less than or equal to 13. If the generated number satisfies this condition, a byte is eligible for mutation. The mutation process happens inside a secondary loop that iterates for the number equal to the total length of the *"seed"* file.
- A third loop takes place after every 500 iterations, where the input string is extended by appending 10 random characters to the end.
- Note that, throughout the iterations, the *"seed"* file on disk remains unaltered.

4) *Output Generation*

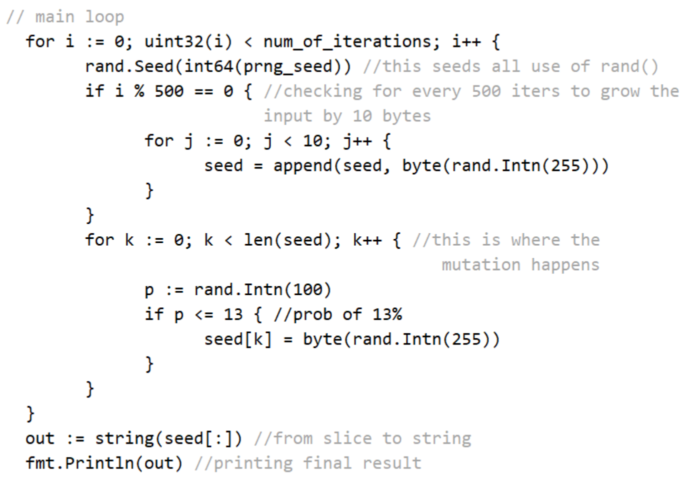
- Upon completing all iterations, the fuzzer writes the mutated input to stdout.

5) *Termination*

- The fuzzer terminates its execution upon completing the fuzzing process.

This methodolog00y allows the fuzzer to effectively generate mutated inputs for the fuzzing target, facilitating the discovery of potential vulnerabilities.

The following image shows the implementation with GoLang of the methodology explained before:



```
// main loop
  for i := 0; uint32(i) < num_of_iterations; i++ {
        rand.Seed(int64(prng_seed)) //this seeds all use of rand()
        if i % 500 == 0 { //checking for every 500 iters to grow the
                            input by 10 bytes
              for j := 0; j < 10; j++ {
                    seed = append(seed, byte(rand.Intn(255)))
              }
        }
        for k := 0; k < len(seed); k++ { //this is where the
                                            mutation happens
              p := rand.Intn(100)
              if p <= 13 { //prob of 13%
                    seed[k] = byte(rand.Intn(255))
              }
        }
  }
  out := string(seed[:]) //from slice to string
  fmt.Println(out) //printing final result
```

Fig. 1. Mutated string generation

III. Results

This section presents the results obtained from the Mutated String Generation program, based on the provided inputs. The table in this section include the following fields:

- `'prng_seed'`: this parameter seeds the PRNG function.
- `'num_of_iters'`: as a shorter version for `'num_of_iterations'` which specifies the number of iterations for the main loop.
- Seed: provided seed file.
- Output string: result.

TABLE I
Results

Mutated String Generation Results			
prng_seed	**num_of_iters**	**Seed**	**Output string**
25046	500	DDDDD DDDDDDD DDDD	�DDu�D�DD DbDu�DDD �b�����]踙
36587	150	abcd DDD DDDDDDD DDDDDD	ab�dp�jDD� DD�DQODD D:DQO�# Z:p �j(
658756	358	dddd aaaaaaaaaaa aaaaaaaaa	d�dd �aaa a�aaaaa a�aaaeaa \��=e��� �M

As depicted in Table I, upon comparing the provided seed string with the output string, noticeable character mutations and other alterations to the original string can be observed. The output string, generated with randomness in mind, was utilized as input for the test programs. Remarkably, in 9 out of 10 cases, the execution resulted in a "segmentation fault," indicating that the implemented Mutation-Based Fuzzer successfully fuzzed the targeted program with a success rate of 90%.

IV. Contributions

A. *Individual Contributions*

I undertook this project solely, encompassing tasks ranging from code development to the creation of this document.

B. *Knowledge Acquired*

Throughout the process of developing and delivering a functional Mutative-Based Fuzzer, I acquired various skills and general understanding of information assurance and Software Security. The following is a concise list along with brief descriptions of the skills and knowledge I obtained:

1) *Fuzzing Techniques:* I gained a comprehensive understanding of different fuzzing techniques, including mutation-based fuzzing. This knowledge facilitated the development of effective fuzzing strategies. [1]

2) *Software Security:* I developed a solid understanding of fundamental concepts and best practices related to software security, including secure coding principles and secure software development lifecycle.

3) *Risk Assessment:* I developed skills in assessing and mitigating risks associated with software vulnerabilities, enabling me to identify potential threats and implement appropriate countermeasures.

4) *Vulnerability Analysis:* I obtained knowledge of vulnerability analysis methodologies and techniques, enabling me to identify and exploit potential software vulnerabilities.

Through this project, I deepened my expertise in fuzzing, software security, risk assessment, and vulnerability analysis.

These skills and knowledge are invaluable in enhancing software quality and security.

V. Conclusion

In summary, the objective of this project was to develop a Binary Fuzzer and employ fuzz testing to evaluate a program's ability to handle unexpected or malformed input. The implementation encompassed the generation of a substantial amount of input data using seeded random data generation and its subsequent execution on the target program. Through the analysis of the obtained results, potential issues within the program were identified, thereby contributing to enhanced security and reliability.

The development process of the Randomized Fuzzer involved a comprehensive understanding of Pseudo-Random Number Generators (PRNGs) and their role in generating random inputs. The fuzzer operated through a systematic procedure that included fuzzer configuration, input generation, iteration process, output generation, and termination. The results achieved from the Mutated String Generation program demonstrated a high success rate of 90% in effectively fuzzing the target program.

Throughout the duration of this project, the researcher acquired a diverse range of skills and knowledge. This encompassed proficiency in various fuzzing techniques, a solid comprehension of software security principles and best practices, proficiency in risk assessment, and familiarity with vulnerability analysis methodologies.

Overall, this project underscores the significance of fuzz testing as a potent technique for uncovering software vulnerabilities and enhancing the security and reliability of software systems. The developed Randomized Fuzzer successfully identified potential issues, thereby emphasizing the importance of integrating fuzz testing into the software development lifecycle.

References

[1] Wikipedia. "American fuzzy lop (fuzzer)." Available: https://en.wikipedia.org/wiki/American_fuzzy_lop_%28fuzzer%29

[2] Fuzzingbook. "Introduction to Testing." Available: https://www.fuzzingbook.org/html/Intro_Testing.html

[3] Testfully.io. "Fuzz Testing." Available: https://testfully.io/blog/fuzz-testing/

[4] Google Scholar. "Fuzzing: The State of the Art." Available: https://wcventure.github.io/FuzzingPaper/Paper/CACM20_Fuzzing.pdf

[5] Lutz, M. "Mathematical Modelling and Computer Simulation." Available: https://www.math.arizona.edu/~tgk/mc/book_chap3.pdf

[6] Jones, D., et al. "Good Practice in Random Number Generation for Cryptographic Applications." Available: http://www0.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf