SOEN 7481 Software Verification and Testing Assignment on Fuzzing, Winter 2025

Khandaker Rifah Tasnia
Concordia University
Montreal, QC, Canada
khandakerrifah.tasnia@mail.concordia.ca

Soumit Kanti Saha
Concordia University
Montreal, QC, Canada
saha.soumit884@gmail.com

Abstract

Fuzz testing is a widely used technique for uncovering edge cases and vulnerabilities in software by generating and injecting malformed or unexpected inputs. This report presents a Python-based fuzzer designed to systematically test various modules by mutating seed inputs and evaluating their impact on code coverage. The fuzzer incorporates a semi-priority queue, which prioritizes mutated inputs based on their effectiveness in increasing coverage, ensuring a targeted and efficient exploration of execution paths. We applied this fuzzer to multiple Python libraries, including requests (HTTP requests), bs4 (BeautifulSoup for HTML parsing), and email_validator demonstrating its capability to expose unexpected behaviors and potential issues. Our experiments reveal that different modules exhibit varying levels of sensitivity to fuzzed inputs, with some achieving high coverage while others demonstrate fluctuating results. This study highlights the importance of intelligent input selection and coverage-guided mutation in automated software testing.

Keywords

Software Testing, Fuzzing, Mutation testing

ACM Reference Format:
Khandaker Rifah Tasnia and Soumit Kanti Saha. 2018. SOEN 7481 Software Verification and Testing Assignment on Fuzzing, Winter 2025. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 2 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software robustness is a critical aspect of modern application development, ensuring that programs function correctly under a wide range of conditions, including unexpected or malformed inputs. Fuzz testing (or fuzzing) is an automated technique used to evaluate software reliability by generating and injecting random, mutated, or intentionally malformed inputs to identify potential crashes, exceptions, or unexpected behaviors. Traditionally, fuzzing has been widely applied in security testing, but its utility extends to general software validation by uncovering edge cases that traditional testing may overlook.

In this report, we present a Python-based fuzzer designed to systematically test various modules, including requests (HTTP requests), email_validator, and bs4 (BeautifulSoup for HTML parsing). The fuzzer employs a semi-priority queue, which dynamically ranks and selects mutated inputs based on their contribution to code coverage. This approach ensures that the fuzzer prioritizes inputs that explore new execution paths, improving efficiency over purely random mutation strategies.

Our experiments involve running the fuzzer on different Python modules to observe variations in coverage and error patterns. The results demonstrate that some libraries exhibit high coverage scores, while others react unpredictably to different mutated inputs, revealing areas where robustness improvements may be needed. This study highlights the importance of adaptive fuzzing strategies and their potential in improving software reliability beyond conventional testing methodologies.

2 Methodology

At first we install the modules to tested and the modules for pytest and coverage. Then we implement out Semi Priority Queue and the Fuzzer. Semi Priority Queue has 3 major operations, push, pop, and peek. Each element of the semi priority queue consists of 3 elements, coverage score, corresponding input and iteration number. In each iteration fuzzer selects a seed (input) that has higher coverage score and mutates that. Mutation can be different kinds.

- flip 2 characters (in case of string) or digits(in case of integer or float)
- replace a character or from random index with another character or digit from alphabet
- append a character or digit at the end of the selected seed
- deleting a random character or digit
- insert a character or digit from the alphabet in a random position
- reverse the input

After that we are ready to experiment with our fuzzer. We experimented 3 python pypi module's functions. These are:

- $validate\_email$ from email_validator module
- get from requests module
- $BeautifulSoap(input, "html.parser")$ from bs4 module

3 Results

When testing our fuzzer with email_validator module's validate_email, we have found that some input worsen the coverage score. This are normally reverse of the previous seed. So the coverage are not consistent in Fig 1

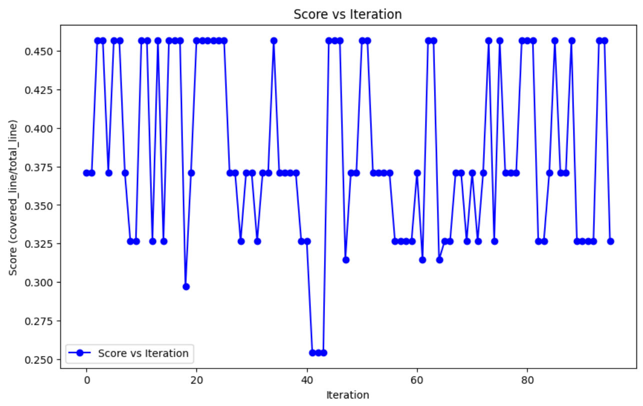

Figure 1: Score for validate_email

When testing get function of requests module, we saw that the coverage scores are polarized. It actually means that the execution path of get function of requests module is too simple. There is only 1 line more coverage for valid urls.

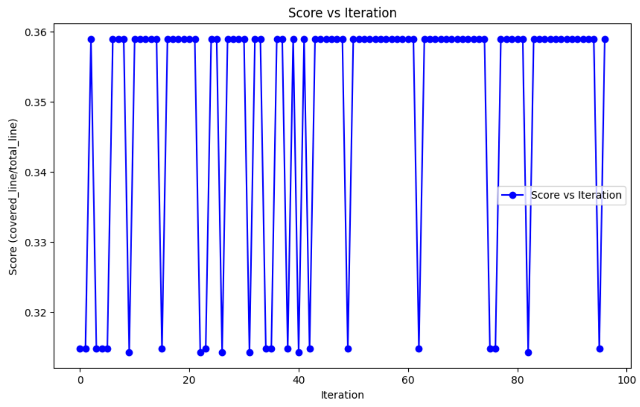

Figure 2: Score for request's get function

When testing BeautifulSoap constructor function of bs4 module, we found that changing characters in critical indices worsen the coverage score (i.e. <, >, !, letters inside <> etc.).

These tests indicates that the fuzzer we have implemented is too simple to compete with AFL [1], which is currently the state of the art. We need proper grammar or rules for mutation and some other advanced adaptive techniques to get a high and consistent coverage.

4 Conclusion

This study has demonstrated the strengths and limitations of our fuzz testing approach in evaluating software robustness. By implementing SmartFuzzer and SemiPriorityQueue, we were able to

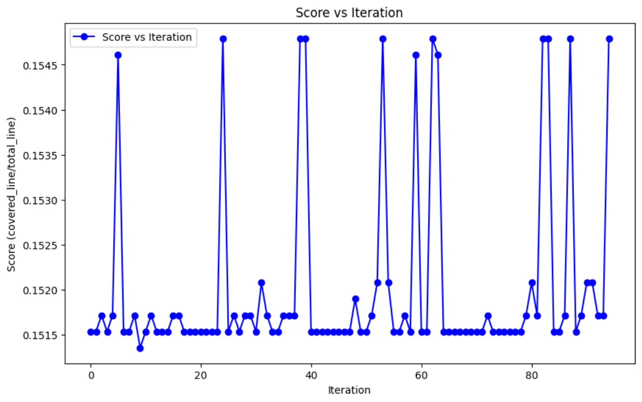

Figure 3: Score for bs4 module's BeautifulSoup(input, "html.parser") function

generate diverse mutated inputs and prioritize test cases that contributed to increased code coverage. The results revealed varying behaviors across different modules. In the case of email validation, we observed inconsistent coverage, with certain mutations—especially reversed inputs—resulting in a decline in effectiveness. The requests.get function exhibited a simple execution path, where valid URLs contributed only a marginal increase in coverage. In the case of BeautifulSoup, we found that the parser handled most malformed HTML gracefully, but modifications to critical indices such as <, >, !, and elements within < > caused significant reductions in coverage.

These findings highlight the importance of adaptive fuzzing techniques in software testing. While our fuzzer successfully uncovered unexpected behaviors, it is relatively simple compared to state-of-the-art tools like AFL. The lack of feedback-driven mutation and deeper input exploration limits its ability to uncover more complex vulnerabilities. Future improvements could focus on refining mutation strategies, incorporating feedback mechanisms, and extending the fuzzer to more sophisticated software systems.

5 Artifacts

Our fuzzer's code can be found here.

References

[1] Michal Zalewski. 2017. American fuzzy lop. (2017).