System & Software Security - Assignment 2 - Replication Study on FixReverter

Gelencsér, Dániel
`s3228479@vuw.leidenuniv.nl`

Khan, Raashid
`s2705745@vuw.leidenuniv.nl`

Sitorus, Brian Arnesto
`s3277224@vuw.leidenuniv.nl`

December 19, 2022

1 Introduction

The aim of this report briefly explains our replication study of the paper - FixReverter [13] along with its experiments and the analysis of our findings from these experiments. The study comprises three components - FixReverter which is a bug injection tool, FuzzBench which is a benchmarking service that evaluates fuzzers and RevBugBench which is a fuzzing benchmark based on FuzzBench. Our main goal in this study was to replicate the experiments conducted in the paper and provide insights on bug fix patterns used by FixReverter and benchmarking of fuzzers on the bug-injected programs.

2 Background

Our replication study contains components that are widely used in the area of Fuzz testing, which is also called ”fuzzing” [12]. Fuzz testing is a Black Box testing method and its main goal is to find implementation flaws by injecting data that is automatically wrong or only partly right. Fuzzing is the process of automatically detecting bugs. The goal of fuzzing is to stress the application so that it behaves unexpectedly, loses resources, or crashes. The process involves providing incorrect, unexpected, or random data to software as inputs. Fuzzers will continue to employ this method and monitor the environment until a system flaw is discovered. Similarly, a fuzzing attack is a process of using fuzzing to discover zero-day vulnerabilities. This technique is commonly employed by thread actors and is known as a fuzzing attack. Fuzzing techniques, on the other hand, are utilized by security professionals in order to evaluate the safety and reliability of applications[2].

There are numerous reasons why fuzzing is crucial for ensuring the security of the system as follows:

1. The results of fuzzing tests provide a comprehensive picture of the software and system being tested. Using fuzzing will make it easier to determine the robustness and security risk profile of the tested system and software.

2. In order to exploit software vulnerabilities, hackers rely heavily on fuzz testing. By incorporating it into a security program, one can reduce the risk of zero-day exploits, which are attacks that take advantage of undiscovered flaws in a computer system.

3. Fuzzing is relatively inexpensive in terms of both cost and time. After being activated and made operational, a fuzzer can search for errors indefinitely without human intervention.

4. Fuzzing enables the detection of bugs that would not have been found using conventional testing methods or manual audits.

5. Fuzzing plays an important role in improving system security by allowing for large-scale vulnerability testing in a short period of time.

2.1 Related Work

Fuzz testing, also known as fuzzing, has been shown to be an effective technique for detecting security flaws. For example, AFL[3], one of the fuzzers that has advanced the most and is used by most people, has a sizable trophy collection. As a result of this success, research into the flaws of fuzzing has accelerated, with dozens of new breakthroughs published in the last few years. Google's FuzzBench employs code coverage as a standardized metric. If fuzzer A (the enhancement) can generate tests that execute more unique lines or branches in a target program than baseline B, then A may find more bugs. [10]. Several studies have been done to find out what the relationship is between code coverage and finding bugs. Recent research indicates that there is a correlation between the amount of coverage a fuzzer receives and the number of bugs discovered [4, 7, 5]. Despite this, there is little consensus regarding the superior fuzzer when coverage is used as a comparison metric [1].

Another often-used metric is the number of unique crash-causing inputs. Researchers use "deduplication heuristics" like AFL's "coverage profiles" and fuzzy stack hashes because the same bug can be triggered by two different inputs [11]. Klees et al. [6] discovered that both heuristics continue to generate false positives and false negatives (many "deduplicated" inputs still trigger the same bug) ("deduplicating" an input can remove evidence of a distinct bug). When ground truth was substituted for "unique" crash heuristics in one program, the result indicated that fuzzer A was superior to baseline B.

There are many benchmarks for fuzzing, but no fuzzer meets all of these requirements for a perfect fuzzer, and many still have problems. These fuzzers are all susceptible to overfitting. Fuzzer developers might use heuristics and strategies that try to beat a benchmark instead of finding more bugs [6]. An automated benchmark for fault injection can aid in avoiding overfitting and achieving other goals, such as bugs can be automatically injected into real programs so that they signal when triggered, and the tool can generate new fault-injected programs as necessary to prevent fuzzers from becoming overfit to a fixed set of programs [13].

3 Methodology

3.1 FuzzBench

FuzzBench[9] is a free service that evaluates fuzzers on a wide variety of real-world benchmarks, at Google scale. The goal of FuzzBench is to make it painless to rigorously evaluate fuzzing research and make fuzzing research easier for the community to adopt.

The process by which FuzzBench operates, as depicted in Figure 1, works in this procedure.

1. The developer of a fuzzer, or anyone else, can integrate it with FuzzBench.

2. After completion, the integration was added to the FuzzBench repository.

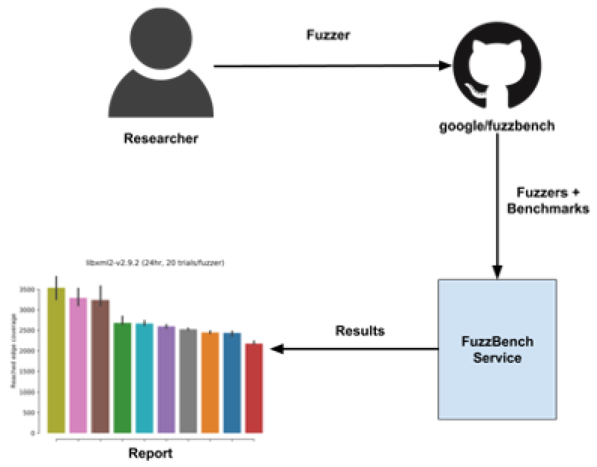

Figure 1: The architecture of Fuzzbench[8]

3. FuzzBench runs an experiment on the benchmarks using the most recent fuzzer version.
4. FuzzBench generates a report that compares the performance of the subject fuzzer to that of other fuzzers, both individually and collectively.

3.2 RevBugBench with FuzzBench

RevBugBench was developed by Zhang et al.[13] by using FixREVerter on eight different programs in FuzzBench and 2 Binutils programs. A total of 8,000 bugs have been injected. To make RevBugBench's use easier when evaluating fuzzers on a large scale, they integrated it into Google's FuzzBench service.

3.3 FixReverter

FixReverter is a bug injection tool that uses Revbugbench as a benchmark. FixReverter enhances the realism of its injected errors by using patterns that correspond to previous Common Vulnerabilities and Exposures (CVE) fixes. FixReverter's task is to find a pattern that corresponds to an observed fix and then reverse it in order to undo the "fix" and (re)introduce a bug [13].

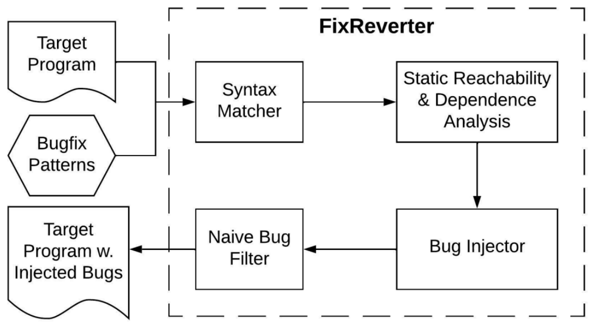

Figure 2: The architecture of FIXREVERTER [13]

FixReverter is illustrated in Figure 2. As inputs, the system requires both a program and descriptions of bugfix patterns to function properly. A bugfix pattern consists of two components: a syntactic pattern and a semantic condition. Using grammar, the syntactic pattern identifies code located at a potential injection site. The semantic condition indicates whether reverting the matched code could cause an application crash [13].

4 Experiments

There were three main goals of experiments in this replication study as outlined below-

1. Using FixReverter to inject bugs on programs.
2. Using RevBugBench/FuzzBench to benchmark fuzzers on the bug-injected programs.
3. Bug triage to find unique causes of crashes caused by the fuzzers.

4.1 Using FixReverter to inject bugs on programs

The first step in our experiment was to inject bugs into real-world programs by using FixReverter. The programs were chosen in the original FixReverter[13] paper and provided in their GitHub repository. This step requires building FixReverter locally on our systems. In order to build and run it efficiently, the paper suggested the following requirements-

- Minimum 200GB RAM
- 2x Intel(R) Xeon(R) Silver 4116 CPUs 192GB RAM Ubuntu 16.04 or,
  2x Intel(R) Xeon(R) Gold 5218 CPUs 384GB RAM Ubuntu 18.04

Hence, due to the unavailability of the aforementioned resources, we decided to use programs that were already injected with bugs from the FixReverter GitHub repository.

The goal of FixReverter is to inject bugs into target programmes in a way that can make it easier to identify unique crashes after benchmarking, and also aim to prevent the benchmark overfitting of certain bugs or target programmes. This is achieved by defining specific bugfix patterns to identify injection sites. The paper discusses three bugfix common patterns. However, they admit that to have meaningful protection against overfitting many more bugfix patterns should be added.

Their implementation includes a syntax matcher which defines bugfix patterns by using grammar files, where the description of a certain syntax is presented symbolically. The program then uses this file to create a state machine, which will be fed with tokens from the Clang abstract syntax tree.

After the matches are collected a static reachability and dependence analysis is conducted, which makes sure that all the injection sites could be a valid cause for a crash of the program. In this stage around 71% of the matched sites are dropped, however, they specify that the analysis may identify some of the unreachable sites incorrectly. We have also noted this part as a possible improvement, but improving this did not fit within the scope of this assignment.

After filtering out potentially uninteresting injection sites which have caused a crash the fuzzer with the default fuzzing feed is used and finally, bugs are injected into each of the target programs.

4.2 Using RevBugBench/FuzzBench to benchmark fuzzers on the bug-injected programs

The next step is to run the fuzzers on the bug-injected programs using RevBugBench/-FuzzBench. As mentioned in the section 3.2, RevBugBench is integrated into FuzzBench and adds more scalability and reproducibility. For this, it builds a dispatcher image with Docker with parameters that set which fuzzers to run and on which benchmark programs. It also requires a configuration file that defines the no of trials, the total amount of time to run and file locations for experiment data as well the report data. This image is then deployed on a docker container.

Following both RevBugBench and FuzzBench documentation we managed to set up and start the experiments, however, the benchmarks could not finish within a reasonable time. The reason for this is that FuzzBench is inherently developed to be run on clusters within the Google Cloud Platform (eg. Kubernetes), running this kind of benchmark locally would require a large number of computing cores with comparable performance to FuzzBench's recommended 96 core Google Compute Engine.

One of the difficulties in replicating RevBugBench was that it is also developed on a previous version of FuzzBench, which has used now deprecated libraries. Our main contribution to the experiment stage of the process is integrating RevBugBench with the current version of FuzzBench, so it can be run without any errors. We verified our fix by running the program over 13 hours, while this wasn't enough to generate reports due to lack of computing power, it was sufficient to confirm that the benchmark is working correctly.

4.3 Bug triage to find unique causes of crashes caused by the fuzzers

This step is to find unique causes for crashes caused by the fuzzers in the previous step. Since the tool did not yield any reports we could not perform this step effectively.

After the benchmarks are completed, the program tries to classify crashes by matching them to the bugs that were initially injected. There are two main categories identified by the paper:

- "individual cause" - This happens when a single bug causes the target program to crash.
- "combinational cause" - When a set of bugs all must be present to cause the crash.

The documentation of FixReverter specifies that running this part with a computer with less than 24 cores is not recommended, because of long execution times. However, it is not specified in detail how long the expected execution time of the bug triage should be. Insufficient computing power was another reason we could not replicate this part.

5 Findings

As mentioned in the experiment section, we were only able to perform the experiments for the second step where we ran the build image of RevBugBench/FuzzBench on a docker container. This was run with different fuzzers for over 13 hours but the tool did not yield any report. Our analysis determined that the tool requires machines with much higher configurations than our local systems. Hence, we made an attempt on the DAS5 system to install dependencies and run the docker image, but the dependencies required by RevBugBench such as docker could not be installed due to limited access/unsupported infrastructure.

Although we were unsuccessful in gaining any results from the experiments due to many issues in replicating the environment the original experiments were on, the steps in this study are quite simple and easy to follow. However, our main contribution to the study was upgrading RevBugBench with the newer version of dependencies and making it executable with the latest commit on FixReverter and FuzzBench. Our opinion is that the study can be replicated with its simple steps but requires machines with higher configurations as well as a lot of time to effectively yield results that can be analysed as these fuzzing tools require longer times in order to cause crashes with their random inputs.

6 Conclusion

Our primary focus replicating the work done by Zhang et al. [13], in their paper, they describe an innovative framework to run fuzzing benchmarks as well as their own benchmark: RevBugBench. The goal FixReverter is to inject realistic bugs that can be uniquely identified and matched to distinct crashes after the benchmarking has been done. Currently, there is little standardization in evaluating fuzzers which makes it challenging to effectively evaluate them. During our research, we also found that most fuzzer benchmarks still use code coverage instead of unique crashes. However, different research papers all seem to agree that unique crashes are a better indication of fuzzing quality. As a part of our contributions, we made changes to make the RevBugBench work with the newest FuzzBench version. However, our evaluation of FixReverter was somewhat inconclusive because of difficulties running the tool locally as it is mentioned in section 5

References

[1] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *44th IEEE/ACM International Conference on Software Engineering, ser. ICSE*, volume 22, 2022.

[2] Supun Jeevaka Dissanayake. *Fuzz Driver Generation.* PhD thesis, 2022.

[3] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[4] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 72–82, 2014.

[5] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*, pages 435–445, 2014.

[6] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[7] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pages 560–564. IEEE, 2015.

[8] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1393–1403, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: An open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1393–1403, New York, NY, USA, 2021. Association for Computing Machinery.

[10] László Szekeres Jonathan Metzman, Abhishek Arya, and Laszlo Szekeres. Fuzzbench: Fuzzer benchmarking as a service. *Google Security Blog*, 2020.

[11] David Molnar, Xue Cong Li, and David A Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, volume 9, pages 67–82, 2009.

[12] Eugene Yang. Fuzz testing & software composition analysis in software engineering. In *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–3. IEEE, 2018.

[13] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. {FIXREVERTER}: A realistic bug injection methodology for benchmarking fuzz testing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3699–3715, 2022.