Poster: Automated Evaluation of Fuzzers

Sebastian Surminski, Michael Rodler, Lucas Davi
University of Duisburg-Essen, Germany
{sebastian.surminski, michael.rodler, lucas.davi}@uni-due.de

***Abstract*—Fuzzing is a well-known technique for automatically testing the robustness of software and its susceptibility to security-critical errors. Recently, many new and improved fuzzers have been presented. One critical aspect of any new fuzzer is its overall performance. However, given that there exist no standardized fuzzing evaluation methodology, we observe significant discrepancy in evaluation results making it highly challenging to compare fuzzing techniques.**

To tackle this deficiency, we developed a new framework, called FETA, which automatically evaluates fuzzers based on a fixed and comprehensive test set enabling objective and general comparison of performance results. We apply FETA to various recently released academic and non-academic fuzzers, eventually resulting in a large-scale evaluation of the current state-of-the-art fuzzing approaches.

I. Introduction

Defects in software cause huge financial losses, threats to security, and privacy. Manual testing often does not manage to achieve the required coverage, and verification does not scale for typical sizes of software projects. As a result, fuzzing has emerged as a scalable solution to testing software robustness. Fuzzing challenges software with randomly generated and mutated input. The software is monitored for crashes or unexpected behavior, identifying possible bugs.

The main difficulty is to generate the appropriate input to trigger errors. Starting from valid initial seeds, fuzzers typically mutate the input and re-run the program under investigation. Due to the large number of execution repetitions, edge-cases and all kinds of malformed input are tested, revealing potential memory errors and software failures. Detecting poorly handled edge-cases is especially interesting in the context of software written in C and C++. Due to the largely manual memory management, memory errors are prevalent and often lead to critical software vulnerabilities [1].

Fuzzing has become a hot research topic with many new developments appearing over the last few years [2]. In particular, the AFL fuzzer [3] project has spiked interest in fuzzing as it has demonstrated the real-world effectiveness of fuzzing. AFL utilizes a greybox fuzzing technique combining coverage tracking with a genetic mutation algorithm to optimize code coverage. AFL requires only light-weight code instrumentation for coverage tracking giving it throughput similar to black-box fuzzing approaches, while being able to explore code paths more systematically. Several improvements to the search and mutation strategies of AFL have been proposed [4]–[6]. However, generating good input sets is highly challenging, especially for programs that require highly structured input formats. Often, the input must satisfy certain restrictions, which the fuzzer's mutation engine is not aware of. Thus, different methods have been proposed to make the fuzzer aware of the input structure. This allows the fuzzer to generate better inputs and explore deep code paths. Further, fuzzing has been combined with program-transformations [7], symbolic execution [8], taint tracking [9] and other combinations of program analysis techniques [10].

The many new developments in the domain of fuzzing lead to a need for systematic evaluation. As of its nature, fuzzing is a stochastic process. Hence, an evaluation has to be performed carefully to produce sound results. There are many factors influencing the performance of a fuzzer, e.g., seed, run time, number of repetitions, but also the selection of targets and bug identification [11]. Typically, when a new fuzzer is developed, it is compared to existing fuzzers. As there are no predefined guidelines for the evaluation of fuzzers, there is a huge variety in the performed evaluations. As a consequence, it is almost impossible to compare fuzzing techniques.

In this poster, we present the first framework, called FETA, which automatically evaluates fuzzers in a large set of predefined experiments. With FETA, new fuzzers can be evaluated based on a given test set and directly compared to previous developments. The evaluation of the results is automated directly generating graphs and figures.

II. Challenges

When evaluating fuzzers, there are several aspects that have to be considered. *Test sets:* The targets for the fuzzers should reflect realistic scenarios and cover a large variety of programs and bugs, so that all fuzzers under investigation can show their strengths and weaknesses. *Comparability:* All experiments must be run under equal circumstances. That is, no fuzzer has any advantages over the others. As fuzzing is a probabilistic process, experiments have to be repeated many times to produce sound results. *Scalability:* The number of targets, repetitions, and fuzzers lead to a vast amount of time-consuming experiments. Hence, the workload has to be distributed among different machines. *Automation:* The resulting system must be completely automated, as manual control is infeasible due to run-time and complexity.

III. FETA Experiment Framework

In FETA, an experiment consists of a specific fuzzer, a target, and an input seed that is run for a certain time. As fuzzing is a probabilistic process, we repeat this process numerous times. In order to allow fair comparison, all experiments run for the same time and with the same available system resources, e.g., processor cores and memory.

We distribute experiments evenly among different machines. Currently, we are using 14 servers with two dual-core

Intel Xeon processors each. As specific demands for the number of available processor cores can be set, all machines have the same available resources. Our framework automatically generates experiments, distributes Docker containers, monitors the execution and collects the results. The results of the various fuzzing runs are subsequently automatically aggregated and evaluated. This design of our framework easily allows us to integrate new targets and new fuzzers, which can be scheduled as further experiments. The outcome is then automatically collected from the docker instances, and integrated into the evaluation.

A. Selected Datasets

Currently there are two major datasets in use for evaluating bug-finding tools, which are already integrated into FETA: The LAVA [12] data sets (LAVA-{1,M}, Rode0day[1], and the DARPA Cyber Grand Challenge (CGC)[2]. These datasets are specially designed for evaluation and come with a ground truth, e.g., the actual number of errors. However, these datasets also have several shortcomings. Both LAVA and Rode0day contain standard programs with synthetically injected bugs, which are structurally similar. As such, the LAVA datasets currently do not contain a large variety of different bugs. The CGC dataset consists of a set of rather small handcrafted programs with known vulnerabilities and minimal OS interactions. The CGC dataset offers a broader variety of bug types. However, it is geared towards automatic binary analysis systems and automated exploit generation and as such does not reflect realistic conditions.

Real-world targets like popular software do not have a ground truth. However, it is possible to use old versions with known bugs. Previous work in this area use a similar set of vulnerable versions of popular programs, e.g., [4]. However, there is no standardized data-set consisting of real-world targets and bugs. We collected popular targets from existing fuzzing papers, projects[3], and other vulnerable software versions (e.g., 7zip, libpng) to build a comprehensive data-set. By combining existing datasets and aggregating ad-hoc datasets from related work, we provide a first step to a standardized dataset for evaluating new fuzzers.

B. Fuzzer Selection

Obviously, we cannot include fuzzers where the source code is not available. As such, we are currently running the fuzzers AFL, AFLFast, FairFuzz, and Angora, but are also working on the evaluation of further fuzzers like honggfuzz, VUzzer, QSYM, Driller. Unfortunately, we had to face several integration problems as several fuzzers either come with no clear instructions or rely on specific dependencies, such as old kernels or compilers.

IV. Related Work

Typically, all newly developed fuzzers come with a performance evaluation. However, there is a large variety in the realization of these evaluations. There are many parameters that have a result on the outcome of the fuzzing process: Apart of target and input seed, one needs to consider the duration of runs and number of repetitions of the process. As fuzzing involves a large random component, the results vary massively. This was shown recently by Klees et al. [11], who performed an evaluation of the experiments of 32 fuzzing papers. In general, the results of different evaluations are not directly comparable. Moreover, wrong evaluations might lead to wrong or misleading conclusions. Hence, a standardized evaluation methodology is urgently needed.

V. Summary and Future Work

We developed a framework to automatically run a standardized performance evaluation of fuzzers. The flexible architecture allows to integrate different fuzzers in a comparable manner. Pre-defined experiments are run on each fuzzer. At the current stage, our servers are running and generate results. We are working on matching bugs that were discovered, and exploring whether fuzzers trigger different bugs. In our future work, we aim at integrating and evaluating further fuzzers into the FETA framework. Any new fuzzer will automatically run in FETA with the pre-defined set of targets and seeds. FETA also enables automatically comparison of the new fuzzers to the already tested fuzzers.

Acknowledgment

This work has been partially funded by the DFG as part of project S2 within the CRC 1119 CROSSING.

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory", in *2013 IEEE Symposium on Security and Privacy*, 2013. DOI: 10.1109/SP.2013.13.

[2] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering", *arXiv preprint arXiv:1812.00140*, 2018. arXiv: 1812.00140.

[3] M. Zalewski, *Technical "whitepaper" for afl-fuzz*, 2016. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt.

[4] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain", *IEEE Transactions on Software Engineering*, 2018. DOI: 10.1109/TSE.2017.2785841.

[5] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage", in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ACM, 2018. DOI: 10.1145/3238147.3238176.

[6] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing", Nov. 2018. arXiv: 1811.09447 [cs.CR]. [Online]. Available: http://arxiv.org/abs/1811.09447.

[7] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by program transformation", in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018. DOI: 10.1109/SP.2018.00056.

[8] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution", *School of Computer Science Carnegie Mellon University*, 2012.

[9] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search", *CoRR*, vol. abs/1803.01307, 2018. arXiv: 1803.01307. [Online]. Available: http://arxiv.org/abs/1803.01307.

[10] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing", in *Proceedings 2017 Network and Distributed System Security Symposium*, 2017. DOI: 10.14722/ndss.2017.23404.

[11] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing", in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018. DOI: 10.1145/3243734.3243804.

[12] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-Scale automated vulnerability addition", in *IEEE Symposium on Security and Privacy*, 2016. DOI: 10.1109/SP.2016.15.

[1] https://rode0day.mit.edu/

[2] https://www.darpa.mil/program/cyber-grand-challenge

[3] e.g., https://github.com/google/fuzzer-test-suite/

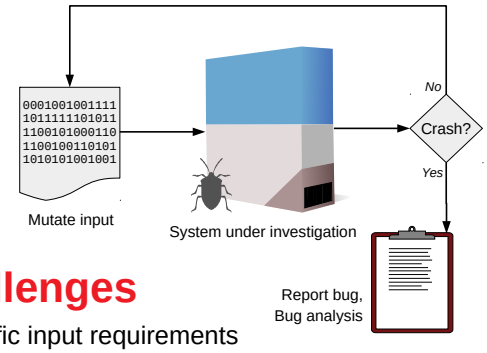
Automated Evaluation of Fuzzers

Sebastian Surminski, Michael Rodler, Lucas Davi
Secure Software Systems
University of Duisburg-Essen, Germany

PALUNO
The Ruhr Institute for Software Technology

Fuzzing

- Automated software testing
- Generate randomly mutated input
- Detect unexpected behavior



Challenges

- Specific input requirements
- Code coverage
- Edge cases
- Deep paths

Evaluation of Fuzzers

- Much research on optimizing fuzzers
- New development come with performance evaluation

But:

- No standardized methodology
- Weaknesses in evaluation

→ **No overall comparison!**

Solution

Standardized performance evaluation of different fuzzers

- Flexible adaptation to new fuzzers and new targets
- Usage of standard target test sets
- Scaling to large number of machines
- Similar computing power for each fuzzer
- Automatic evaluation

Status: Running!

The FETA Fuzzing Framework

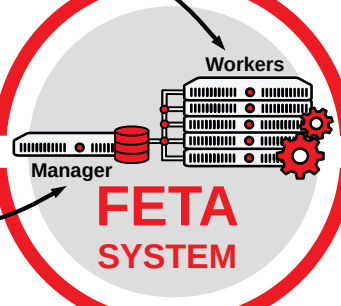
Fuzzer Workers

- Docker Containers
- Automatically distributed via Docker Swarm
- Similar resources given

Evaluation

Fuzzer evaluation using popular fuzzing test sets

- DARPA Cyber Grand Challenge
- Rode0day
- LAVA



Fuzzer Manager

- Automatic distribution
- Equal distribution of resources
- Distribution of fuzzing targets and seeds to workers
- Scheduling of experiments
- Collection of results
- Evaluation and graphs

Ongoing Work

- Extend the evaluation set by existing applications
- Integrate more fuzzers (Angora, hongfuzz, Vuzzer, QSYM, Driller)
- Detailed vulnerability and crash analysis

UNIVERSITÄT DUISBURG ESSEN
Open-Minded

Institute for Computer Science and BusinessInformation Systems (ICB)
Working Group for Computer Science
Secure Software Systems
University of Duisburg-Essen
Germany
https://www.syssec.wiki.uni-due.de/en/

CROSSING

This work has been partially funded by the DFG as part of project S2 within the CRC 1119 CROSSING.