HasteFuzz: Full-Speed Fuzzing

Zhengjie Du
Nanjing University
Nanjing, China

Yuekang Li
Nanyang Technological University
Singapore

***Abstract*—Fuzzing throughput is an important factor affecting the overall fuzzing results. HasteFuzz presents an input filtering mechanism to improve fuzzing throughput. During fuzzing, HasteFuzz filters out inputs whose execution traces are encountered before and only conducts further analysis (bug and new coverage checking) on inputs with new execution traces. In this way, HasteFuzz enables the fuzzer to test and analyze more diverse inputs and improve fuzzing throughput.**

I. Motivation Behind the Idea

The primary goal of our design is to be compatible with other fuzzing techniques as much as possible. Our secondary goal is to make sure that the proposed approach can constantly improve the fuzzing performance regardless of what kind of target programs we are trying to test. After some empirical study, we found that ***improving the overall fuzzing throughput*** is an excellent angle to start with. The reasons are as follows: ❶ Most other advanced fuzzing techniques (such as improved power scheduling [4] or mutation operators [7]) can benefit from the execution speed gain. So we can stand on the shoulders of the giants by integrating our technique with existing powerful fuzzers. ❷ Improving the fuzzing throughput is beneficial for all types of programs under test (PUT). In comparison, most heuristic-based improvements can work well on certain types of PUT but not all of them.

To facilitate the idea of improving fuzzing throughput, we scrutinized the key steps of greybox fuzzing, which are shown in Figure 1. In general, greybox fuzzers work in three steps. ❶ Test input generation. The greybox fuzzer needs to generate the test inputs either by mutating existing seeds or generating from scratch with the help of grammar rules/templates. ❷ Target program execution. After a test input is generated, it is fed to the PUT for execution. The greybox fuzzer will wait for the PUT to finish execution before making the next move. ❸ Execution feedback collection/analysis. After the target program finishes execution, the greybox fuzzer will collect and analyze the execution feedback, including signals of triggering bugs and the information collected through the instrumentation (such as basic-block transition coverage). The analyzed information can be used for identifying whether a bug has been detected or adjusting the fuzzing strategy dynamically to improve future results. Among the three steps, the speed of steps ❶ and ❸ are limited by the logic of the fuzzer while the speed of step ❷ is limited by the PUT. Based on our experience, the PUT execution normally takes more time. However, we have less control over the execution of the PUT. Therefore, we choose to focus on improving the execution

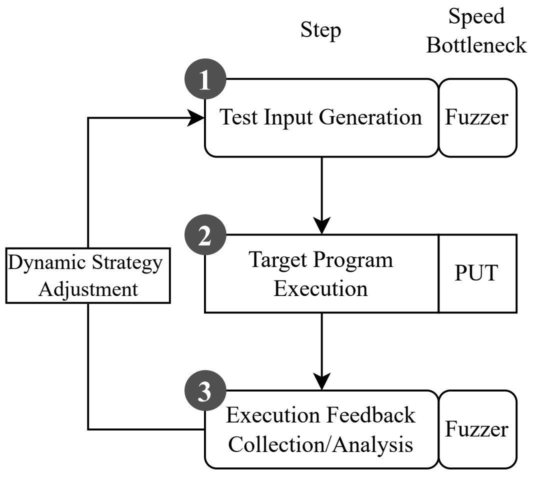


Fig. 1. Greybox Fuzzing Overview

speed of the fuzzer. Moreover, previously, when we were developing FOT [5], we tried to generate performance profiles for greybox fuzzers and we found that step ❸ consumes much more time than step ❶. Hence we focus on improving the fuzzing throughput from the aspect of step ❸.

II. Methods of HasteFuzz

Our basic idea is to use a filter to select valuable inputs which need further analysis. The rationale is to save the time spent for step ❸ in Figure 1 by stopping some inputs from going through the in-depth and time-consuming analysis. Figure 2 shows the workflow of HasteFuzz, we use the hash value of bitmap that can differentiate the execution traces to filter inputs. During fuzzing, given an input, HasteFuzz first executes the input on coverage-instrumented binary to get its coverage bitmap. Then the *Filter* calculates the hash value of the bitmap. If the hash value is never encountered before, the input is kept and goes for later analysis. The analysis includes executing the input on the sanitizer-instrumented binary for bug checking (if such a binary is provided) and comparing the bitmap for new coverage checking and global bitmap updating. If the hash value has been encountered, the input is discarded to save time and the next round of fuzzing begins afterwads.

The hash value of bitmap (the execution trace) is a reasonable metric and the filtering mechanism can improve fuzzing throughput because of the following reasons: ❶ If two inputs have the bitmap hash value, they usually have the same results

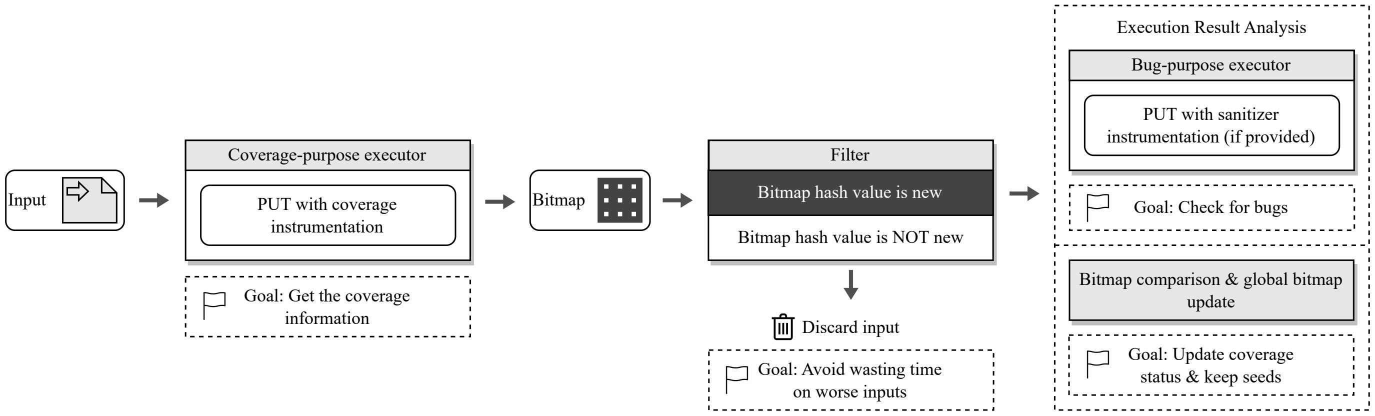

Input
Coverage-purpose executor
PUT with coverage instrumentation
Goal: Get the coverage information
Bitmap
Filter
Bitmap hash value is new
Bitmap hash value is NOT new
Discard input
Goal: Avoid wasting time on worse inputs
Execution Result Analysis
Bug-purpose executor
PUT with sanitizer instrumentation (if provided)
Goal: Check for bugs
Bitmap comparison & global bitmap update
Goal: Update coverage status & keep seeds

Fig. 2. Workflow of HasteFuzz

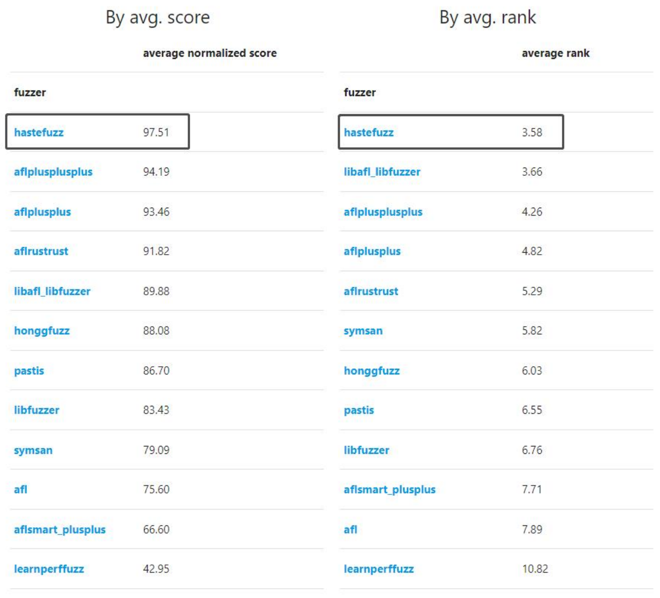
for triggering bugs. This may not hold absolutely in theory, because two identical execution traces may have different variable values, leading to different bug-triggering results. However, based on our experience and some preliminary experiments on dozens of bugs, we found that for real-world programs, the execution traces of different bugs are often unique and do not have collisions. ❷ For capturing bugs efficiently, target programs are normally compiled with sanitizers, such as address sanitizer (ASAN) [8]. However, using sanitizers hinders the execution speed of the PUT a lot. ❸ Through preliminary experiments, we also found that many generated inputs lead to the same execution trace during fuzzing. Because the number of such inputs is large, skipping the analysis of them can bring a considerable overall throughput boost. Previously, all the inputs are executed on the sanitizer-instrumented binary which is slow, even if the execution traces of these inputs are the same. With our filtering mechanism, the inputs are executed on the coverage-instrumented binary which is fast, and only the inputs with unique execution traces (the execution traces are never encountered before during fuzzing). This is why the filtering mechanism can speed up fuzzing and thus allows the fuzzer to try more inputs with diverse execution traces, finally improving the fuzzing throughput.

III. Implementation & Discussion about Competition Results

As for implementation, we implemented our ideas on AFLPlusPlus [6] because according to the previous performance reports (as of February 2023) on Fuzzbench, AFLPlusPlus performs very well in general. The source code of HasteFuzz is available on Github [3].

According to the final evaluation results of SBFT 2023 [1], [2], HasteFuzz secures the first place on coverage-based benchmarks while its performance on bug-based benchmarks is mediocre. Figure 3 shows the coverage competition results and figure 4 shows the bug detection results.

For coverage-based benchmarks [2], the programs are compiled without sanitizer instrumentation. The good results benefit from 1) the excellent performance of AFLPlusPlus itself



By avg. score

fuzzer	average normalized score
hastefuzz	97.51
aflplusplusplus	94.19
aflplusplus	93.46
aflrustrust	91.82
libafl_libfuzzer	89.88
honggfuzz	88.08
pastis	86.70
libfuzzer	83.43
symsan	79.09
afl	75.60
aflsmart_plusplus	66.60
learnperffuzz	42.95

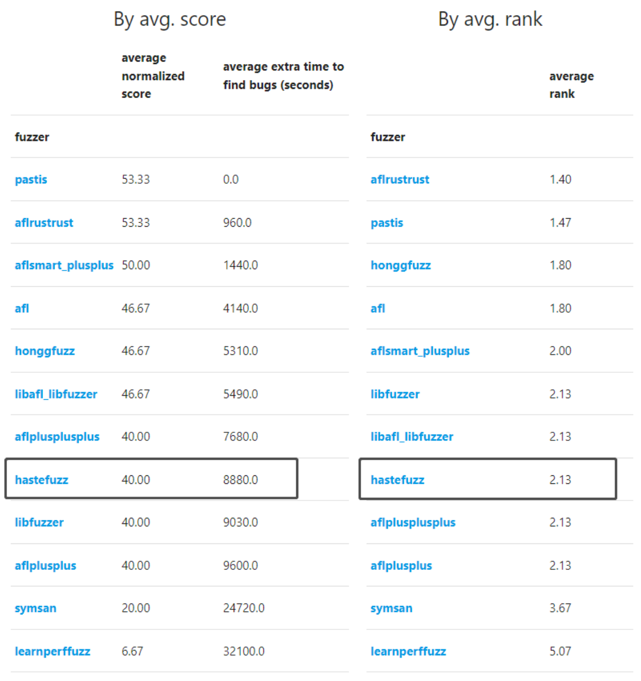
By avg. rank

fuzzer	average rank
hastefuzz	3.58
libafl_libfuzzer	3.66
aflplusplusplus	4.26
aflplusplus	4.82
aflrustrust	5.29
symsan	5.82
honggfuzz	6.03
pastis	6.55
libfuzzer	6.76
aflsmart_plusplus	7.71
afl	7.89
learnperffuzz	10.82

Fig. 3. Coverage results in SBFT'23 competition

and 2) the filtering mechanism of HasteFuzz. The performance of AFLPlusPlus is already very good and how not to hinder its performance while making improvements is challenging. The improvement brought by the filtering mechanism is orthogonal to most of the advanced features of AFLPlusPlus. Therefore, theoretically, the performance of HasteFuzz should be better and the results on the coverage-based benchmark verified this. Moreover, since the bitmap hash ID is already calculated before bitmap comparison in *fast* mode, which is the default mode of AFLPlusPlus, it can be reused by HasteFuzz without incurring extra time costs for calculation. This further helps with the fuzzing throughput from the implementation aspect.

For bug-based benchmarks [1], the programs are compiled with sanitizer instrumentation. The results of HasteFuzz are mediocre. From the benchmark aspect, each of the benchmark programs has only one bug inside. The fuzzers performing well on covering more code may perform badly on these programs



By avg. score

fuzzer	average normalized score	average extra time to find bugs (seconds)
pastis	53.33	0.0
aflrustrust	53.33	960.0
aflsmart_plusplus	50.00	1440.0
afl	46.67	4140.0
honggfuzz	46.67	5310.0
libafl_libfuzzer	46.67	5490.0
aflplusplusplus	40.00	7680.0
hastefuzz	40.00	8880.0
libfuzzer	40.00	9030.0
aflplusplus	40.00	9600.0
symsan	20.00	24720.0
learnperffuzz	6.67	32100.0

By avg. rank

fuzzer	average rank
aflrustrust	1.40
pastis	1.47
honggfuzz	1.80
afl	1.80
aflsmart_plusplus	2.00
libfuzzer	2.13
libafl_libfuzzer	2.13
hastefuzz	2.13
aflplusplusplus	2.13
aflplusplus	2.13
symsan	3.67
learnperffuzz	5.07

Fig. 4. Bug detection results in SBFT'23 competition

because they can get distracted by the code sections irrelevant to triggering the bug. From the fuzzer aspect, HasteFuzz still needs to be improved because the filtering mechanism is not always beneficial for all fuzzing stages. Detailed discussions about the shortcomings of HasteFuzz and potential future solutions can be found in Section IV.

In summary, the experimental results showed that HasteFuzz can indeed improve the overall fuzzing throughput while its performance can be further improved.

IV. Threats to Validity & Future Work

The first threat is about whether it is truly safe to discard certain inputs just according to the hash values of their corresponding bitmaps. First, the hash values can have collisions, which is rare but inevitable. Second, as discussed in Section II, having the same execution trace does not guarantee to reach the same program status. This threat is from the nature of the filtering mechanism and it cannot be avoided. However, the results and our previous experience show that the risk of discarding potentially useful inputs is acceptable since the overall result is improved.

The second threat is that at the early stage of fuzzing, for bug detection purposes, the current mechanism of HasteFuzz may bring penalties to the throughput. This is because if an input holds a bitmap that has never been encountered before, it will be executed twice. The first round of execution is with the binary instrumented only with the coverage collection logic. The second round of execution is with the binary instrumented by the sanitizers. In the early stage of fuzzing, a lot of inputs can have unique bitmaps. Therefore, many inputs are executed twice during the start, which can turn out to be a waste of time. Our proposed solution to this problem is to design an adaptive algorithm to decide whether to use the *Filter* or directly execute the input with the sanitizer-instrumented binary according to the stage of fuzzing.

V. Conclusion

In this paper, we present HasteFuzz, a technique to improve the overall fuzzing throughput by filtering out less interesting inputs earlier. Overall, HasteFuzz performed well in the Search-Based Fuzz Testing (SBFT) 2023 competition by ranking 1st on the coverage-based benchmark. However, HasteFuzz can still be improved by adaptively applying the filter during different stages of fuzzing and we leave this as future work.

References

[1] Fuzzbench: Sbft'23 final evaluation report (15 bug-based benchmarks). https://storage.googleapis.com/www.fuzzbench.com/reports/experimental/SBFT23/Final-Bug/index.html.

[2] Fuzzbench: Sbft'23 final evaluation report (38 coverage-based benchmarks). https://storage.googleapis.com/www.fuzzbench.com/reports/experimental/SBFT23/Final-Coverage/index.html.

[3] Hastefuzz source code. https://github.com/AAArdu/hastefuzz.

[4] Marcel Bohme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45:489–506, 2016.

[5] Hongxu Chen, Yuekang Li, Bihuan Chen, Yinxing Xue, and Yang Liu. Fot: A versatile, configurable, extensible fuzzing framework. FSE '18 tool demo, 2018.

[6] 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)*. USENIX Association, August 2020.

[7] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem A. Beyah. Mopt: Optimized mutation scheduling for fuzzers. In *USENIX Security Symposium*, 2019.

[8] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. 2012.