TuneFuzz: adaptively exploring target programs

Han Zheng, Flavio Toffalini and Mathias Payer
HexHive, EPFL

***Abstract*—The trade-off between *exploration* and *exploitation* stages poses a major challenge to Greybox fuzzing. TUNEFUZZ addresses this challenge through a novel input prioritization algorithm that maximizes the reached and triggered sanitizer labels. Our multi-distance metric and dynamic target ranking improve both *exploration* and *exploitation*. TUNEFUZZ found 56 new bugs (38 CVEs) in well-tested open source software.**

***Index Terms*—fuzzing, sanitizer**

I. INTRODUCTION

Greybox fuzzing has proven its efficacy in hunting real-world bugs. Coverage-guided greybox fuzzers like AFL [1] instrument the target programs and collect their edge coverage. Based on the simplifying assumption that vulnerabilities are evenly distributed among the whole program, AFL adopts a seed selection strategy that treats each edge equally to maximize the overall coverage, thus enhancing its bug-finding capability according to the baseline assumption.

However, existing works point out that not all code regions are equally vulnerable. Evidently, vulnerable areas deserve more (or even most) of the fuzzing energy. Instead of evenly fuzzing all code regions, an optimal greybox fuzzer should assess the threat capability of each area to then prioritize security-sensitive regions for bug finding. Several related works have focused on this challenge. TortoiseFuzz [2] proposes coverage accounting to estimate the memory corruption threat from basic blocks, loop and function levels respectively, further prioritizing seeds with estimated higher risk. While coverage accounting cannot model non-memory error, ParmeSan [3] and SAVIOR [4] propose to use sanitizer labels [5] as threat indicators. Compared to coverage accounting, sanitizer-guided fuzzers can handle arbitrary types of bugs if there are corresponding sanitizer, introducing significant portabilities.

A Sanitizer Directed Greybox Fuzzers (SDGF) should balance between two phases: (1) **exploration**, *i.e.,* fuzzers should direct the exploration towards sanitizer targets; and (2) **exploitation**, *i.e.,* a fuzzer should repetitively test the reached targets and try to satisfy the sanitizer conditions.

For existing SDGFs, we observe two core limitations that affect both **exploration** and **exploitation**. First, existing SDGFs calculate the distance from one seed to a target set. Regardless of the target set size, the distance will be normalized to one via harmonic average. For SDGFs that leverage sanitzer labels, the target set may contain up to hundred thousand targets, posing a great challenge for fuzzer to distinguish which target is closer to a specific seed. Second, current SDGFs assign a time-invariant priority to the target. By reachable analysis or profiling, they remove less promising targets before the fuzzing campaign, and treat the remaining targets equally at runtime. Unfortunately, we observe that the target priority changes during the fuzzing campaign. Therefore, a time-invariant approach might misclassify the targets, thus missing the real bugs or wasting energy in bug-free areas.

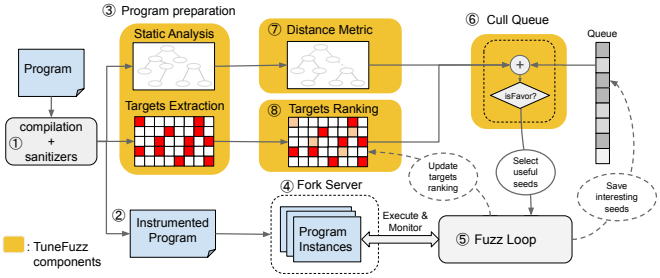

Fig. 1: Overview of TUNEFUZZ workflow.

We propose TUNEFUZZ [6], which addresses the above challenges through our three core contributions:

- A distance metric between seeds and targets that is independent of the size of the target set and robust against unresolved indirect jumps.
- A dynamic target ranking that automatically guides the fuzzer's energy towards promising locations, while discarding thoroughly explored ones.
- TUNEFUZZ, a novel queue culling strategy that maximizes the number of explored and exploited targets.

II. DESIGN AND IMPLEMENTATION

Figure 1 depicts the overall workflow. During compilation, TUNEFUZZ customizes the plug-in LLVM pass and collects the static callgraph, control flow graph, and sanitizer labels accordingly ③. The instrumented programs ② follow the existing greybox fuzzing approach in which a forkserver repeatedly executes during the life cycle via the pipelines ④. Meanwhile, TUNEFUZZ calculates the seed-to-target distance according to its execution trace ⑦, ranking the target ⑧ and select corresponding seeds ⑥. During execution ⑤, runtime feedback allows TUNEFUZZ to update the target priority map.

A. Distance Metric

To overcome the imprecision of current seed-target distance metrics [3], [7], we propose to estimate the "closest" seed for each target independently. However, tracing the distance at basic block level would require an enormous amount of information, which would slow down the fuzzing campaign. To reduce the tracing overhead, we measure the minimal distance between the seed and functions on the call graph.

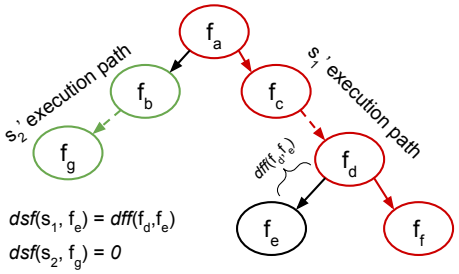

Fig. 2: Example of handling indirect calls.

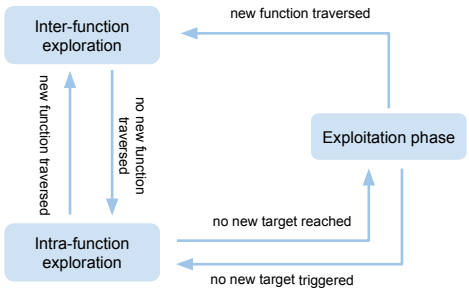

Fig. 3: TUNEFUZZ queue culling model

We first pre-calculate a *static distance map* that contains the minimum distance among functions. Then, we leverage the *static distance map* to estimate the closest seed to a function at runtime (*i.e.,* the seed that visits the function closest to the target). In the last part of the section, we discuss how TUNEFUZZ handles indirect calls.

Static Distance Map: The *static distance map* is a look-up table that contains the minimum distance for each function pair. The distance is estimated over the control-flow-graph (CFG) and the call-graph (CG), which we extract during compilation from LLVM-IR. This initial analysis, for now, is oblivious to indirect calls.

To calculate the *static distance map*, TUNEFUZZ assigns a *weight* for each function pair (f_i, f) such that f is a callee of f_i. The *weight* represents the minimum number of conditional edges that a seed might traverse from the entry point of f_i to the callee function f, and is computed with the function $dbb(m_a, m_b)$ (*i.e.,* distance from basic block m_a to m_b).

After computing the *weights*, TUNEFUZZ defines the distance between two functions as the sum of their *weight* along the shortest path between two functions based on the compilation-extracted CG.

Dynamic Seed to Function Distance: Having the *static distance map*, we define a function $dsf(s, f)$ that represents the distance between the functions traversed by the seed s and a function f, which iterate the f_i in the path of seed s, then take the minimum $weight(f_i, f)$ as the final $dsf(s, f)$.

Figure 2 illustrates how we also handles indirect calls.

B. Dynamic Target Priority

One of the key features of TUNEFUZZ is to focus its energy by selecting the most promising targets. Inspired by ASOP [8], we notice that frequently hit targets are "*already well explored*" and therefore less likely to be buggy. Less explored targets should therefore be prioritized to uncover new bugs. Thus, we record the hit frequency of each target during the fuzzing campaign, then assign less frequently visited target a higher priority and further prioritize corresponding seeds.

C. Queue Culling Algorithm

Our queue culling module relies on the Distance Measurement (§ II-A), Target Ranking (§ II-B), and default algorithm of the base fuzzer. These three components are used in *inter-function exploration*, *exploitation*, and *intra-function exploration*, respectively – all pictured in Figure 3. The purpose of the *inter-function* exploration is to reach interesting functions (*i.e.,* expanding the search scope). This phase leverages the Distance Measurement (§ II-A) to select seeds closer to functions containing untriggered targets. The *intra-function* exploration focuses on testing internal functions based on the original fuzzer algorithm, and tries to hit as many targets as possible. Finally, the *exploitation phase* maximizes the triggered targets (*i.e.,* reaping the benefits) by using the Dynamic Target Ranking (§ II-B).

CONCLUSION

Fuzzers find bugs by *exploring* new code regions and repeatedly trying to *exploit* them. TUNEFUZZ's novel input prioritization mechanism balances these two phases explicitly. For *exploration*, TUNEFUZZ precisely tracks ten thousands of target locations through its multi-distance metric. For *exploitation*, TUNEFUZZ adaptively discards exhausted targets for better energy distribution. So far, TUNEFUZZ discovered 56 new bugs (38 CVEs), two of which were acknowledged by Apple and Huawei (EulerOS-SA-2022-2226, HT213340), TUNEFUZZ will be released as open source and integrated with FuzzBench.

REFERENCES

[1] M. Zalewski, "american fuzzy lop," https://lcamtuf.coredump.cx/afl/.
[2] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization." in *NDSS*, 2020.
[3] S. Osterlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan: Sanitizer-guided greybox fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2289–2306.
[4] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "Savior: Towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1580–1596.
[5] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.
[6] H. Zheng, J. Zhang, Y. Huang, Z. Ren, H. Wang, C. Cao, Y. Zhang, F. Toffalini, and M. Payer, "Fishfuzz: Catch deeper bugs by throwing larger nets," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1343–1360.
[7] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.
[8] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 866–879.