Directed Fuzzing for Use-After-Free Vulnerabilities Detection

(*Doctoral Work Presentation*)

Manh-Dung Nguyen [†]
Université Paris-Saclay, CEA LIST, France
manh-dung.nguyen@cea.fr

1 Introduction

Context. *Coverage-based Greybox Fuzzing* (CGF) [1] shows its ability to find various types of bugs in real-world applications. While the main goal of CGF is to cover as many program states as possible in a limited time, *Directed Greybox Fuzzing* (DGF) [2,3] aims to perform stress testing on potentially vulnerable target locations, with applications to different security contexts: (1) bug reproduction, (2) patch testing or (3) static analysis report verification. We focus mainly on *bug reproduction*, which is the most common practical application of DGF. It consists in generating Proof-of-Concept (PoC) inputs of disclosed vulnerabilities given bug report information. It is especially needed since only 54.9% of usual bug reports can be reproduced [4]. Even with a PoC provided in the bug report, developers may still need to consider all corner cases of the bug in order to avoid regression bugs or incomplete fixes. Bug stack traces, sequences of function calls at the time a bug is triggered, are widely used for guiding directed fuzzers [2, 3].

Problems. Fuzzers have made tremendous progress on specific problems (e.g., magic bytes comparison, deep execution, lack of directedness and complex file formats), but existing greybox fuzzers still have a hard time finding *complex vulnerabilities* such as Use-After-Free (UAF), non-interference or flaky bugs, which require bug-triggering paths satisfying very specific properties. *We focus on UAF bugs – one of the most critical exploitable vulnerabilities.* They appear when a heap element is used after having been freed (Listing 1) and have serious consequences.

```
1 char *buf = (char *) malloc(BUF_SIZE);
2 free(buf); // pointer buf becomes dangling
3 ...
4 strncpy(buf, argv[1], BUF_SIZE-1); // Use-After-Free
```

Listing 1: Code snippet illustrating a UAF bug.

[†]PhD student co-supervised at CEA LIST & Univ. Grenoble Alpes (27/11/2017 – 20/11/2020). Thanks to my supervisors for their inputs: Roland Groz, Richard Bonichon, Sébastien Bardin and Matthieu Lemerre. This research was supported by a grant from the H2020 C4IIOT project.

Challenges. Fuzzers targeting the detection of UAF bugs confront themselves with the following challenges.

- **C1. Complexity** – Exercising UAF bugs require to generate inputs triggering a *sequence* of 3 events – *alloc*, *free* and *use* – *on the same memory location*, spanning multiple functions of the program under test (PUT). This combination of both *temporal* and *spatial* constraints is difficult to meet in practice;
- **C2. Silence** – UAF bugs often have *no observable effect*, such as segmentation faults. Thus, fuzzers simply observing crashing behaviors do not detect that a test case triggered such bugs. Sadly, popular profiling tools such as ASan or VALGRIND cannot be used in a fuzzing context due to their high overhead.

Related Work. AFLGO [2] was the first directed fuzzer. It features a simulated annealing-based power schedule that gradually favors seeds whose traces are closer to the target locations. HAWKEYE [3] improves it in terms of seed prioritization, power scheduling and mutation strategies. Although DGF effectively solves the reachability problem of the target locations, existing DGF is limited in detecting UAF vulnerabilities in binaries even given the UAF bug target locations.

Research Directions. We propose 3 research directions as follows:

- **D1.** *Directed fuzzer towards* UAF *targets extracted from a bug trace*, which is a sequence of function calls at the time a bug is triggered (*discussed in details in this paper*). It could be obtained by running the PUT with a PoC input under profiling tools such as VALGRIND [5].
- **D2.** *Typestate directed fuzzer detects common bugs* (e.g., UAF, buffer overflows) that are considered as the violation of typesate properties (*work in progress*).
- **D3.** *Hybrid directed fuzzer towards* UAF *targets extracted from static reports* of UAF detector like GUEB [6] (*as future work*). GUEB, which is the *only binary-level static analyzer for UAF*, performs a dedicated value-set analysis with different heap modeling approaches.

Contributions. Our work **D1** [7, 8] makes the followings contributions:

- We design the *first* directed greybox fuzzing technique dedicated to *UAF bugs* working directly *on executables*.
- We implement the fuzzer UAFUZZ on top of AFL [1] and BINSEC [9].
- We construct a *new UAF fuzzing benchmark* [10] of 30 real UAF bugs.
- For *bug reproduction* setting, we evaluate UAFUZZ against state-of-the-art coverage-guided and directed greybox fuzzers using our UAF fuzzing benchmark. For *patch testing* setting, UAFUZZ successfully discovers 11 new UAF (4 buggy patches) in critical projects like Perl, GNU Patch and GPAC.

2 The UAFUZZ approach

Overall we propose 3 dynamic input metrics specialized for UAF vulnerabilities detection, used in conjunction with a dedicated power schedule assigning more energy (i.e., number of mutants) to favored seeds that are wisely selected by our new heuristic during fuzzing (as shown in Fig. 1).

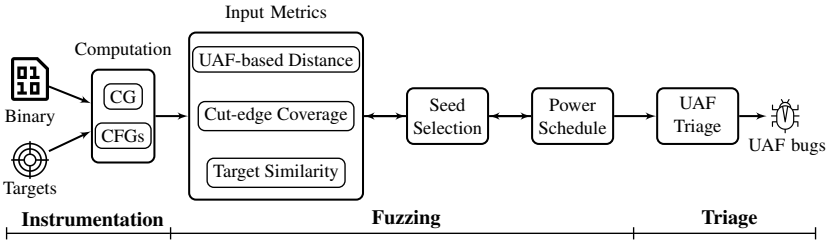

Figure 1: The UAFUZZ workflow.

Target Similarity metric & Seed Selection heuristic. A *target similarity metric* measures the similarity between the execution of a seed and the target UAF bug trace. Our seed selection algorithm is based on two insights. First, we *should prioritize seeds that are most similar to the target bug trace*. Second, *target similarity should take ordering (a.k.a., sequenceness) into account*, as traces covering sequentially a number of locations in the target bug trace are closer to the target than traces covering the same locations in an arbitrary order.

UAF-based Distance metric. Previous seed distances [2, 3] do not account for any order among the target locations, while it is essential for UAF. We address this issue by modifying the distance between functions in the call graph to favor paths that *sequentially* go through the 3 UAF events *alloc, free and use* of the bug trace.

Cut-edge Coverage metric. We propose the *lightweight* cut-edge coverage metric by measuring the "*reachability progress*" at the *edge level* but on the *critical decision nodes only*. Our heuristic is that an input exercising more cut edges whose edge destinations are more likely to reach the next target in the bug trace, is more likely to cover more locations of the target UAF bug trace.

Power Schedule. The power scheduler determines the energy for each selected seed based on its dynamic metric scores at runtime. We therefore spend more time fuzzing seeds that (1) *are closer* (using the UAF-based seed distance); (2) are *more similar to the UAF target bug trace* (using the target similarity); (3) *make better decisions* at critical code junctions (using the cut-edge coverage).

UAF Triage. False positive inputs are finally filtered by running the PUT with *potential inputs* which exercise in sequence all target locations of the target UAF bug trace under a profiling tool (here VALGRIND [5]) in the bug triaging phase.

3 Current Results

Implementation. We develop a lightweight static analysis as a plugin of the binary analysis platform BINSEC [9] and the dynamic fuzzing part based on AFL-QEMU 2.52b [1]. We also re-implement the best state-of-the-art DGF, named AFLGOB[1]

[1]The comparison between AFLGOB and source-based AFLGO is discussed in [7].

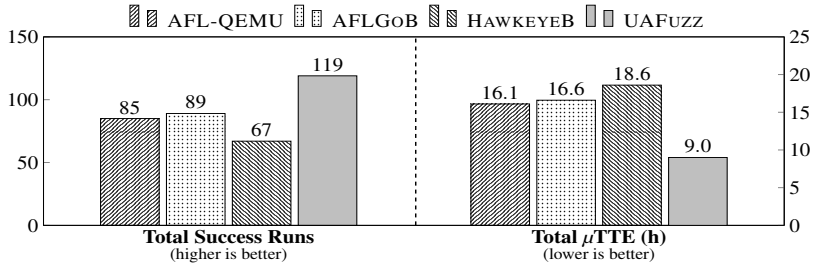

Figure 2: Summary of fuzzing performance (RQ1)

and HAWKEYEB, in our own framework for binary-level fuzzing because HAWKEYE [3] is not available and AFLGO [2] works on source code only.

RQ1 – Bug-reproduction Ability. Overall UAFUZZ *significantly outperforms* existing fuzzers in terms of UAF bugs reproduction. We use Time-to-Exposure (TTE) and the number of success runs in which a fuzzer could trigger the bug as two important metrics to compare the efficiency of each fuzzer. As shown in Fig. 2, UAFUZZ has the *largest* total number of success runs and achieves 34% (up to $+300\%$) more than the second best fuzzer AFLGOB. Furthermore, UAFUZZ can find bugs around $2\times$ faster than other fuzzers. Interestingly, UAFUZZ is able to find the bugs faster than AFLGO with source code in 4 cases, which implies the efficiency of our techniques.

RQ2 – Overhead. UAFUZZ is *an order of magnitude faster* ($14.7\times$ in total) than the source-based directed fuzzer AFLGO in the instrumentation phase. The total numbers of executions done of UAFUZZ are *slightly smaller* (-4% in total) than AFL-QEMU, which implies that UAFUZZ's runtime overhead is negligible.

RQ3 – UAF Triage. UAFUZZ reduces a *large portion* (i.e., more than 90% of triaging inputs) in the post-processing phase. Unsurprisingly, UAFUZZ spends the *smallest* amount of time (i.e., an average of $7.4s$ – a speedup of up to $17.5\times$ over AFLGOB) in the bug triaging step thanks to our target similarity metric.

RQ4 – Individual Contribution. Our experiments show that UAF-based distance, power schedule and seed selection heuristic *individually contribute* to improve the performance of corresponding variants built on top of AFLGOB. Combining these components can *further improve* effectiveness and efficency of our technique.

4 Conclusion & Future Work

UAFUZZ is the *first directed* greybox fuzzing approach tailored to detecting UAF bugs *in binary* given only the bug trace. By specializing standard (directed) greybox fuzzing components to UAF, UAFUZZ outperforms existing directed fuzzers, both in terms of time to bug exposure and number of success runs. Our technique also enjoys only a small overhead (instrumentation- and run- time), and speeds up the bug triage step by significantly reducing the number of seeds to be sent to an external UAF checker. Finally, we currently follow the directions **D2** and **D3**.

References

[1] "Afl," http://lcamtuf.coredump.cx/afl/, 2020.

[2] 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 (CCS17)*, 2017.

[3] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2095–2108.

[4] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 919–936. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/mu

[5] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan*, vol. 42-6. ACM, 2007, pp. 89–100.

[6] J. Feist, L. Mounier, and M.-L. Potet, "Statically detecting use after free on binary code," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 211–217, 2014.

[7] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for use-after-free vulnerabilities," *The 23nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '20)*, 2020.

[8] ——, "About Directed Fuzzing and Use-After-Free: How to Find Complex & Silent Bugs?" Black Hat USA, 2020.

[9] "Binsec," https://binsec.github.io/, 2020.

[10] "Uaf fuzzing benchmark," https://github.com/uafuzz/UAF-FuzzBench, 2020.