MYSTIQUE: Enhancing Mutators in Fuzzing

Jikai Wang
Huazhong University of Science and Technology
Wuhan, China
wangjikai@hust.edu.cn

Yuekang Li
University of New South Wales
Sydney, Australia

Kailong Wang
Huazhong University of Science and Technology
Wuhan, China

***Abstract*—To overcome constant bytes checking during grey-box fuzzing, a lightweight and effective method, known as CmpLog, was proposed and widely implemented in existing fuzzers. However, when combined with other mutators, good mutations that are made by CmpLog can be broken. We proposed and implemented a new method to protect the mutations made by CmpLog during the mutation process of Fuzzing. During mutation, we mark the bytes that CmpLog mutates to prevent other mutators from modifying the range with a probability. Based on our local fuzzbench experiments, our optimizations have resulted in a fuzzer coverage improvement on 3 out of 6 benchmarks.**

I. MOTIVATION

Passing byte sequence checks is important during fuzzing. There are constant byte sequence checks in many programs, e.g., checking the magic header for file type. If the byte sequence is placed correctly, then the checking is passed and a new chunk of the program can be covered.

The CmpLog instrumentation, proposed by Cornelius et al. [1], can be used to effectively overcome magic bytes checking. It is based on the observation that Input-to-State correspondence exists between the program's input and its state. When performing bytes comparison, one side of the check often directly corresponds, without modification, to a portion of the input. The basic idea of CmpLog is to instrument all multi-byte checks in the program, including comparison instructions on assembly level and strcmp/memcmp function calls, and then the fuzzer uses a mutation that alters the byte sequence on one side of the comparison to match the other side. Compared to other methods, CmpLog is effective and fast, which coincides with the philosophy of fuzzing. CmpLog has been adopted by existing fuzzers such as LibAFL and AFL++.

Although CmpLog can help to make the byte sequences correct in the generated test input, the random mutators adopted by the fuzzer may corrupt the byte sequences, hindering the path exploration. To address this issue, our idea is to add a mask to the input ranges that have already been mutated by CmpLog, to prevent it from being modified again by other mutators. When combined with other mutators, the modifications made by CmpLog can be overwritten by other mutators. Since the modifications made by CmpLog are likely to overcome certain checks, we hope to let other mutators focus on other areas by preserving CmpLog's modifications.

The implementation detail and the code is available on Github [3].

II. APPROACH

Fig. 1 illustrates the workflow of MYSTIQUE and highlights the optimizations made by MYSTIQUE. To maximize the performance of the fuzzer, we implemented two optimizations. Firstly, we maintain masks for the CmpLog mutator to protect certain input bytes. Secondly, we implemented the optimization from HasteFuzz to increase code coverage.

A. Preserving Good Bytes in Input

As shown in Fig. 1, we maintain and associate masks (a list of ranges) for each input in the queue to reduce the impact of unnecessary mutations. When a seed is initially imported into the queue, it is not associated with any mask. During the mutation phase, if the CmpLog mutator is used, the mutated range is inserted into the mask list. If other mutations are used, the range of the mask is probabilistically rolled back afterward (with a probability of 5/16 in our case). If the mutated input is considered interesting by subsequent stages, it is saved in the queue along with the mask.

There are two design considerations behind the choice of restoring mutations based on masks. Firstly, it allows for large-scale input transformations. That is, when the input is truncated, if the masked range exceeds the input range, the corresponding part will not be restored. Secondly, always preserving the content within the mask range does not maximize coverage. For example, completely disallowing the modification of certain magic bytes to incorrect bytes will fail to cover some failed code checks.

By employing this approach, we adjusted the probability distribution of the mutator to enhance the effectiveness of mutations.

B. Implementing the Idea of HasteFuzz

In the previous competition, SBFT'23, HasteFuzz [2] won first place in the coverage category, so we also adopted their idea. As HasteFuzz does not achieve great result on bug based benchmarks, we only focus on the optimization for coverage based benchmarks.

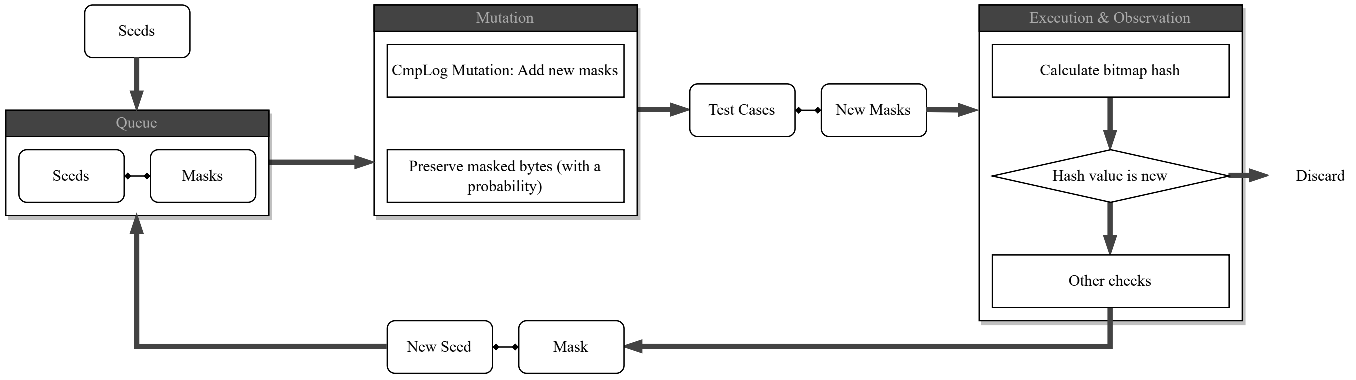

Fig. 1: Workflow of Mystique

TABLE I: The code coverage sample statistics on two benchmarks influenced by the optimization (23 hours with 2 retries). libafl_n_16 represent the probability of mask recovery is n/16.

Benchmarks	libafl_8_16	libafl_5_16	libafl_3_16	libafl_1_16
freetype2_ftfuzzer	11499	11061	11268.5	10536.5
mbedtls_fuzz_dtlsclient	2937.5	3161	3094.5	3581.5

The AFL's bitmap reflects the edge coverage of the program under test. At the same time, it maintains another global coverage map. If a program covers new edges (more precisely, updates the global coverage map), it is considered interesting and saved in the queue for mutation. This means that after one run, even if the pattern of its edge coverage is new, the input may not necessarily be saved to the queue. However, we can infer that if the edge coverage pattern is old, it will not be considered interesting.

The optimization added a filter step at the beginning of the feedback process to filter out some uninteresting inputs. After each run of the instrumented program, the edge bitmap is filled with edge coverage info in this run. We use the pattern of this bitmap to filter out uninteresting input. We hash this bitmap and check if this hash has been seen previously. If the hash is seen previously, we will directly treat the input as uninteresting.

To efficiently record and lookup known hashes, we allocated a 256MB memory as a bitmap for hash values. We truncate the hash to 30 bits and use each bit to represent whether or not we have seen the hash value.

III. Results

A. Testing the Probability of Masking

We conducted a local experiment to test the relationship between the probability of recovering masks and coverage score. We observed that as the probability of mask recovery increased, the coverage on freetype2_ftfuzzer improved, while the coverage on mbedtls_fuzz_dtlsclient decreased. This indicates that the effectiveness of the masking method is closely related to the code being tested.

B. Comparison With the Original Fuzzer

Table 2 presents the coverage statistics of our fuzzer and the original LibAFL on 6 benchmarks in fuzzbench. It is shown that our optimization outperforms the original LibAFL on three benchmarks.

TABLE II: The code coverage sample statistics on 6 benchmarks (23 hours with 2 retries). libafl_opt contains our optimizations, while libafl does not.

	libafl	libafl_opt
bloaty_fuzz_target	6305.4	6194
draco_draco_pc_decoder_fuzzer	1447.4	1348
lcms_cms_transform_fuzzer	2098.6	2041.2
libpcap_fuzz_both	2748	2816.2
mbedtls_fuzz_dtlsclient	2982.8	3112.2
stb_stbi_read_fuzzer	2143	2182.2

IV. Discussion

This method assumes that the mutations made by CmpLog can indeed effectively bypass the corresponding byte checks. However, CmpLog is not perfect. If there is a method that can determine that the transformation of a certain input region does bypass certain byte sequence checks, we can protect those regions with a higher probability.

Currently, LibAFL's fuzzbench fuzzer is an in-process fuzzer that is linked into the binary with CmpLog instrumentation enabled. It uses a global variable to determine whether the instrumentation is enabled. We also attempted to compile a binary without CmpLog and use a persistent fork server to launch it. However, experiments show that the coverage scores on benchmarks decreased. The overhead introduced by inter-process synchronization may be greater than the overhead of the additional variable check introduced by CmpLog.

Sander et al. [4] mentioned that the implementation of CmpLog in AFL++ has performance issues. We have observed that LibAFL implemented a simplified version of CmpLog, particularly excluding the colorization phase mentioned in Redqueen [1]. This indicates that there is still room for improvement in the implementation details of CmpLog.

References

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[2] Zhengjie Du and Yuekang Li. Hastefuzz: Full-speed fuzzing. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 73–75. IEEE, 2023.

[3] Wang Jikai. Mystique: A Fuzzer in the SBFT'24 Fuzzing Competition. https://github.com/am009/fuzzbench/tree/SBFT24, 2024.

[4] Sander Wiebing, Thomas Rooijakkers, and Sebastiaan Tesink. Improving afl++ cmplog: Tackling the bottlenecks, 2022.