MendelFuzz: The Return of the Deterministic Stage

HAN ZHENG, EPFL, Switzerland
FLAVIO TOFFALINI, EPFL, Switzerland and Ruhr-Universität Bochum, Germany
MARCEL BÖHME, Max Planck Institute for Security and Privacy, Germany
MATHIAS PAYER, EPFL, Switzerland

Can a fuzzer cover more code with minimal corruption of the initial seed? Before a seed is fuzzed, the early greybox fuzzers first systematically enumerated slightly corrupted inputs by applying every mutation operator to every part of the seed, once per generated input. The hope of this so-called *"deterministic" stage* was that simple changes to the seed would be less likely to break the complex file format; the resulting inputs would find bugs in the program logic well beyond the program's parser. However, when experiments showed that disabling the deterministic stage achieves more coverage, *i.e.,* applying multiple mutation operators at the same time to a single input, most fuzzers disabled the deterministic stage by default.

Instead of ignoring the deterministic stage, we analyze its potential and substantially improve deterministic exploration. Our deterministic stage is now the default in AFL++, reverting the earlier decision of dropping deterministic exploration. We start by investigating the overhead and the contribution of the deterministic stage to the discovery of coverage-increasing inputs. While the sheer number of generated inputs explains the overhead, we find that only a few critical seeds (20%), and only a few critical bytes in a seed (0.5%) are responsible for the vast majority of the coverage-increasing inputs (83% and 84%, respectively). Hence, we develop an efficient technique, called MendelFuzz, to identify these critical seeds / bytes so as to prune a large number of unnecessary inputs. MendelFuzz retains the benefits of the classic deterministic stage by only enumerating a tiny part of the total deterministic state space.

We evaluate MendelFuzz implementation on two benchmarking frameworks, FuzzBench and Magma. Our evaluation shows that MendelFuzz outperforms state-of-the-art fuzzers with and without the (old) deterministic stage enabled, both in terms of coverage and bug finding. MendelFuzz also discovered 8 new CVEs on exhaustively fuzzed security-critical applications. Finally, MendelFuzz has been independently evaluated and integrated into AFL++ as default option.

CCS Concepts: • **Security and privacy → Software security engineering**.

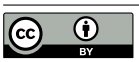
Additional Key Words and Phrases: Greybox Fuzzing, Deterministic Stage, Automatic Testing

ACM Reference Format:
Han Zheng, Flavio Toffalini, Marcel Böhme, and Mathias Payer. 2025. MendelFuzz: The Return of the Deterministic Stage. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE003 (July 2025), 21 pages. https://doi.org/10.1145/3715713

1 Introduction

Mutation and coverage-feedback are the two key ingredients of the Coverage-guided Greybox Fuzzer (CGF) [Böhme et al. 2016; Fioraldi et al. 2020b; Lemieux and Sen 2018; Lyu et al. 2019; Yue et al. 2020; Zalewski 2013], one of the most successful automatic testing techniques which executes

Authors' Contact Information: Han Zheng, han.zheng@epfl.ch, EPFL, Lausanne, Switzerland; Flavio Toffalini, flavio.toffalini@rub.de, EPFL, Lausanne, Switzerland and Ruhr-Universität Bochum, Bochum, Germany; Marcel Böhme, marcel.boehme@acm.org, Max Planck Institute for Security and Privacy, Bochum, Germany; Mathias Payer, mathias.payer@nebelwelt.net, EPFL, Lausanne, Switzerland.



ACM 2994-970X/2025/7-ARTFSE003
https://doi.org/10.1145/3715713

Table 1. For these (4 of 23) programs in the FuzzBench fuzzer benchmark, the edge coverage is higher for fuzzers where the DETERMINISTIC STAGE is enabled (det) than for fuzzers where it is not (havoc).

	AFL		AFL++		Entropic		Honggfuzz	
benchmark	det	havoc	det	havoc	det	havoc	det	havoc
systemd_fuzz-link-parser	1244	640	1256	640	1251	639	1283	639
woff2-2016-05-06	2306	1859	2316	1872	2372	1837	2318	1893
re2-2014-12-09	4152	3527	4121	3517	4191	3555	4032	3505
jsoncpp_jsoncpp_fuzzer	665	638	665	638	669	641	626	640

a target program with thousands of automatically created inputs per second. Given a *seed input* (like a JPEG image file), the CGF generates a new input by applying a random *mutation operator* (e.g., bit flip) to a random part of the seed input and stacking such random mutations. The generated input is executed on the target program (like the libJPEG image parser library). If there is a crash, the generated input is reported. If *code coverage is increased*, the generated input is added to the seed corpus. A common approach to boost the coverage achieved over time is to prioritize seeds [Böhme et al. 2016; Lemieux and Sen 2018; She et al. 2022; Zheng et al. 2023] or mutation operators [Lyu et al. 2019; Wu et al. 2022; Yue et al. 2020] that are more likely to increase coverage. However, if the initial seeds are corrupted too much, the fuzzer may fail to recover a valid structure in further mutations. On the one hand, the mutation-based approach allows developers to effectively test programs that take highly structured inputs without explicitly encoding any knowledge of the required input structure. For instance, to test the libJPEG image parser library, they would only need to provide valid JPEG-files as initial seeds. On the other hand, if fuzzing the initial seeds S yields highly corrupted, coverage-increasing inputs as the next generation of seeds S', it may be more difficult to maintain the structural validity of the inputs generated from seeds $s \in S'$ and subsequent seed generations, reducing the effectiveness of the campaign. To cope with this issue, popular CGFs, like AFL and AFL++ [Fioraldi et al. 2020b; Zalewski 2013], implement a DETERMINISTIC STAGE which systematically enumerates *every* mutation operator applied to *every* part of a seed, once per generated input, to generate new inputs with *minimal corruption* (§2). If enabled, the DETERMINISTIC STAGE runs before the HAVOC STAGE, which stacks random mutations on the seed to generate new inputs. Table 1 shows four programs in the FuzzBench fuzzer benchmarking platform where enabling the DETERMINISTIC STAGE (det) made a big difference [FuzzBench 2020a,b]. However, after observing that disabling the DETERMINISTIC STAGE often improves performance [Fioraldi et al. 2020b; Wu et al. 2022], the DETERMINISTIC STAGE was disabled by default (HAVOC-only) [Metzman et al. 2021]. Indeed, for the other nineteen programs in FuzzBench, the HAVOC-only mode performed better in terms of edge coverage. So, what is the bottleneck?

In this paper, we first analyze the overhead and the contributions of the DETERMINISTIC STAGE using the Magma benchmark (§3), which later inspires the development of MENDELFUZZ, a substantially improved DETERMINISTIC STAGE. On average, within a fuzzing campaign that enables the DETERMINISTIC STAGE, the fuzzer finds 68% more new edges during the DETERMINISTIC STAGE than during the HAVOC STAGE while spending 25.80X more time. This explains the substantial overhead. However, we also find that only a few critical seeds (20%), and only a few critical bytes in a seed (0.5%) are responsible for the vast majority of the coverage-increasing inputs (83% and 84%, respectively). This indicates that most of this overhead may actually be redundant.

Based on our finding that only some bytes of a few input seeds yield interesting coverage-increasing inputs during DETERMINISTIC STAGE, we develop MENDELFUZZ to boost the fuzzing

efficiency (§4). MendelFuzz selectively breeds seeds and allows the fuzzer to automatically identify which bytes and seeds profit from the deterministic stage mutators, thus enabling a more optimized allocation of resources. MendelFuzz introduces the new notion of critical byte (following the spirit of favored seeds [Zalewski 2013]) that approximates the bytes of an input that would profit from the deterministic stage. Additionally, MendelFuzz leverages a fast look-up table, called inference map, to efficiently identify the critical bytes. Finally, MendelFuzz relies on a deterministic fuzzed map to dynamically predict which seeds contribute to new coverage when using the deterministic stage mutators.

We implement MendelFuzz based on AFL++ and evaluate it in the Magma [Hazimeh et al. 2020] and FuzzBench [Metzman et al. 2021] benchmarks (§5). In our comparison to both baselines, MendelFuzz outperforms AFL++ with and without the (old) deterministic stage enabled, both in terms of coverage (10.13% and 0.44%) and bug finding (69.10% and 8.46%). In our comparison to four state-of-the-art fuzzers, AFLFast [Böhme et al. 2016], FairFuzz [Lemieux and Sen 2018], Mopt-AFL [Lyu et al. 2019] and Weizz [Fioraldi et al. 2020a], in terms of coverage, MendelFuzz discovers at least 10.11% more edges than fuzzers with deterministic stage *enabled*, and ranks first among fuzzers where deterministic stage is disabled, both in FuzzBench and Magma. In terms of bug finding, MendelFuzz triggers 62.33% and 8.46% more unique bugs than fuzzers where deterministic stage is enabled and disabled, respectively. In a separate evaluation of its practical utility on extensively fuzzed security-critical applications, MendelFuzz finds 21 new bugs (*e.g.*, liblouis as Apple fonts, Wireshark as basic network framework), among which 8 new CVEs are assigned. To further evaluate our suggested improvements, we analyze the overhead and contributions of our improved deterministic stage in MendelFuzz. Compared to the original deterministic stage, MendelFuzz takes notably less time (reduced from 96.27% to 2.13%), while still contributing to a large percentage of the coverage finding (from 62.63% to 30.35%). Our MendelFuzz implementation based on AFL++ is integrated and again enabled by default in the main line AFL++.

Contribution. In summary, our contributions are as follows:

- We thoroughly evaluate the performance impact of the deterministic stage and study the root cause of its performance trade-offs.
- We elaborate on the lessons learned from our study and design MendelFuzz, a new meta-strategy that boosts the efficiency of existing deterministic stage.
- We release our implementation and all the material to replicate our results at https://github.com/HexHive/MendelFuzz-Artifact.

2 Background

2.1 Deterministic Stage

deterministic stage was initially introduced in AFL [Zalewski 2013] and subsequently adopted by the AFL/AFL++ family fuzzers [Böhme et al. 2016; Fioraldi et al. 2020b; Lemieux and Sen 2018]. This stage constitutes a fundamental component of the fuzzing strategy. deterministic stage is usually executed first and linearly traverses the input bytes by applying various mutations (*e.g.*, bitflip, arithmetic addition/subtraction, interesting value replacement, and token replacement). After exhaustive testing, the fuzzer transits to the havoc stage where *multiple concurrent* mutations are applied to random input bytes. Figure 1 illustrates the overall workflow of the deterministic stage.

AFL's design philosophy [AFL 2019] is composed of two meta-stages. First, the fuzzer runs the deterministic stage to generate more test cases with single corrpution, thereby aiming at triggering the bugs while preserving the input structure. Subsequently, the fuzzer switches to

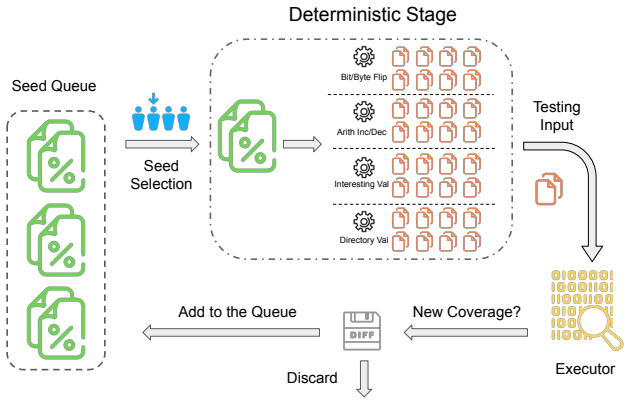


Fig. 1. Deterministic Stage Workflow.

the HAVOC STAGE that extensively corrput the seed, steering the target toward a different path to maximize the code coverage.

2.2 Effective Map Mechanism

To improve the DETERMINISTIC STAGE performance, AFL/AFL++ family fuzzers introduce the concept of EFFECTIVE MAP. This mechanism involves grouping input bytes into 8-byte chunks, with the fuzzer monitoring whether modifications to individual chunks change the program execution path. If mutating any byte within the chunk alters the program behavior, the whole data chunk is marked as effective in the EFFECTIVE MAP. Otherwise, all data within this chunk is disregarded in the later deterministic fuzzing.

EFFECTIVE BYTE alleviates the workload of DETERMINISTIC STAGE [AFL 2019]. Specifically, EFFECTIVE BYTE accelerates the DETERMINISTIC STAGE by reducing 47.67% of the deterministic executions. However, our investigation reveals that the existing DETERMINISTIC STAGE, despite being boosted by EFFECTIVE BYTE, still suffers from high overhead. In our study (Figure 2), 96.27% of the time is allocated to process the EFFECTIVE BYTE, most of which does not contribute to the fuzzing. Yet, EFFECTIVE BYTE, despite introducing a huge performance boost, is still insufficient for real-world fuzzing practice.

3 Effectiveness of the Deterministic Stage

In Table 1, we observe that some program exhibits better performance when the DETERMINISTIC STAGE is activated. To investigate this phenomenon, we evaluate the DETERMINISTIC STAGE coverage contribution and time consumption (§3.1). Our study reveals the potency of the DETERMINISTIC STAGE despite its sluggish efficiency. Moreover, our analysis on DETERMINISTIC STAGE's finding, for both intra-seed (§3.2) and inter-seed (§3.3), indicate that 84% and 83% of the path discovery originate from 0.5% bytes and 20% seeds, respectively. These observations establish the theoretical feasibility of the DETERMINISTIC STAGE optimization. Our study employs the following setup:
Benchmark: We choose Magma [Hazimeh et al. 2020] as our primary evaluation platform. Magma is a ground-truth fuzzing benchmark that focuses on bug-finding capability. We run our study on the latest stable Magma, which includes 18 programs with diverse functionalities and input formats.

Experimental Setup: All experiments are performed on a Xeon Gold 5218 CPU (22M Cache, 2.30 GHz) equipped with 64GB of memory. Regarding the initial seed corpus and CPU resources, we strictly follow Magma's default setup (except we update the AFL++ version to the latest), *i.e.,* using the same set of seed corpus and binding each container to one CPU core. We also disable the cmplog feature in AFL++ and MendelFuzz for a fair comparison between MendelFuzz and non-AFL++ based fuzzers (*e.g.,* AFLFast, FairFuzz, MOpt-AFL).
Other Configuration: We investigate the deterministic stage based on AFL++, a production-ready fuzzer that ranks first in FuzzBench [Google 2023]. Specifically, we run AFL++ with both deterministic stage and havoc stage enabled, log the time consumption of both stages, and analyze the generated seeds' names. Magma is using an outdated AFL++ version from 2021, which affects its performance [Heuse 2023]. So we update the version to 4.10c for both AFL++ and MendelFuzz.

3.1 Exploring of Deterministic Stage

During our experiment, we collect edge discovery data and time consumption for the deterministic stage and havoc stage respectively. Specifically, we customize AFL++ [Fioraldi et al. 2020b] to log these events and conduct a 24h Magma [Hazimeh et al. 2020] campaign repeated for 10 runs. Throughout the campaign, we track edge discovery and time for each stage, recording when AFL++ finishes testing a seed and collecting new edge findings and time consumption. We aggregate coverage and time data, excluding initial seed corpus coverage for fairness. Furthermore, we utilize the AFL++ instrumentation to collect collision-free edge coverage [AFLplusplus 2025] and remove the statistically non-representative data, i.e., programs that cannot complete exploration of one seed in 24h.

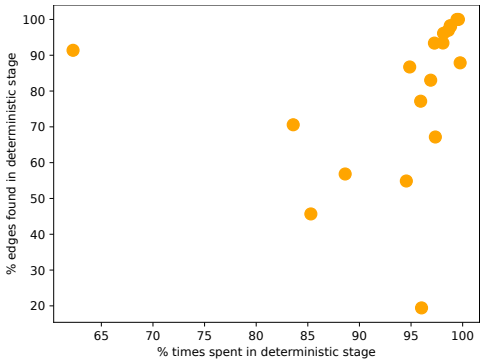


Fig. 2. Edge Discovery and Time Consumption of the deterministic stage. The numbers are collected from a 24h Magma fuzzing campaign with AFL++ running the deterministic stage. Each point stands for the result of one Magma program.

Figure 2 shows that in the majority of the targets, deterministic stage finds more than 80% of the edges. Overall, deterministic stage contributes 62.63% of the new code coverage throughout the whole fuzzing campaign. Despite deterministic stage's notable coverage contribution, the speed constrains its broader deployment. AFL++ generally allocates 96.27% of the fuzzing time to the deterministic stage, which is 25.80X times of the havoc stage. This substantial time consumption impedes the adoption of the deterministic stage in real-world fuzzing practice. We also explore different AFL++ setups (§5.2): AFL++ det-only, det+havoc, and havoc-only, The normalized coverage results are 86.3, 89.4, and 97.9, while the unique bug results are 25.3, 28.8, and 44.9 respectively, which further support our findings. In the following section, we illustrate a more in-depth investigation into the root cause of the overhead from both intra-seed (§3.2) and inter-seed (§3.3) dimensions.

> **Observation 1:** Preliminary results suggest that the deterministic stage consumes a significant amount of time during the fuzzing campaign. However, it may potentially explore new coverage.

3.2 Effective Deterministic Bytes

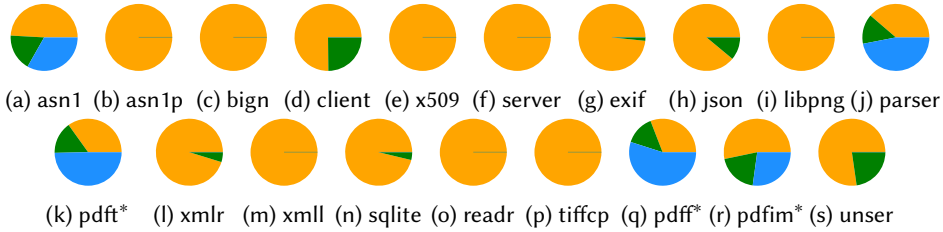


Fig. 3. Paths distribution grouped by input bytes. The numbers are recorded from a 24h fuzzing campaign with AFL++ running DETERMINISTIC STAGE against Magma benchmark. The orange (■), green (■), and blue (■) segments indicate the percentage of path found by $< 0.5\%$, $0.5\% - 1\%$ (excluding first 0.5%), and $> 1\%$ (excluding first 1%) input bytes, respectively. Programs that complete fewer than 10 seeds are excluded and programs that complete fewer than 20 seeds are marked (*).

The study in §3.1 indicates that the DETERMINISTIC STAGE has potential, discovering 1.67 times edges of the HAVOC STAGE (dividing the edge finding of DETERMINISTIC STAGE (59.39%) by the HAVOC STAGE (40.61%)), albeit at a significant cost of time. Given the substantial value offered by the DETERMINISTIC STAGE despite its time requirement, we look for options to retain the benefits without the associated cost. In this section, we aim to explore paths for reducing the time consumption of the DETERMINISTIC STAGE.

From our experiment, we parse the seed name in the queue to collect the number of paths found by each byte, sum up the top 0.5%, 0.5% to 1%, and bottom 99% bytes' finding, consequently plotting their distribution. Figure 3 illustrates our findings. We observe that the majority of the DETERMINISTIC STAGE findings originate from a small subset of the bytes. In practice, mutating the top 0.5% of the bytes can yield nearly the same findings in 12 out of 19 programs. Besides four outliers, the top 1% bytes can cover all current findings. For pdftoppm, pdffuzzer, and pdfimages (marked '*'), less than 20 seeds were tested during the first 24 hours, thus suggesting their results are incomplete and should be excluded. The main takeaway of these statistics is that mutating only 0.5% of the bytes discovers more than 84% of the DETERMINISTIC STAGE findings, while opting for around 1% of the bytes ends up covering 97.5% DETERMINISTIC STAGE finding in terms of new paths. Therefore, we can narrow the intra-seed search space of DETERMINISTIC STAGE while retaining its effectiveness.

> **Observation 2:** After analyzing how the seed bytes contribute to the coverage, we observe that 0.5% of the seeds' bytes contribute to 84% of the paths found in the DETERMINISTIC STAGE.

3.3 Effective Deterministic Seeds

As discussed in previous works [Böhme et al. 2016; Lemieux and Sen 2018; She et al. 2022], fuzzers iterate through an exploded set of seeds. According to the original DETERMINISTIC STAGE (§2.1), every seed, at the first time being tested, is fuzzed deterministically. An overgrowth seed queue introduces overhead, so we conduct a statistic per seed's finding in the DETERMINISTIC STAGE.

We first deduplicate our log to ensure each seed is only counted once. Next, we collect the new paths found by each seed to create the distribution pies, *i.e.,* we count the seeds that originate from the same seed. Figure 4 illustrates our discovery. The orange segments mean the seeds ranking at the top 5% in terms of path finding, green segments represent the findings for seeds ranking

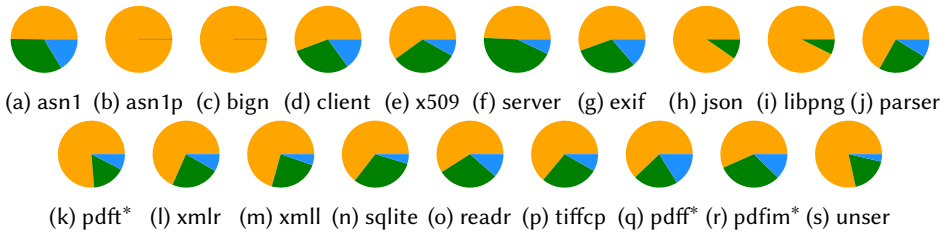


Fig. 4. Paths distribution grouped by seeds. The numbers are recorded from a 24h fuzzing campaign with AFL++ running DETERMINISTIC STAGE against Magma benchmark. The orange (■), green (■), and blue (■) segments indicate the percentage of path found by $< 5\%$, $5\% - 20\%$ (excluding first 5%), and $> 20\%$ (excluding first 20%) seeds, respectively. Programs that complete fewer than 10 seeds are excluded and programs that complete fewer than 20 seeds are marked (*).

between 5% to 20%, and the blue slices are the remaining seeds' findings. Note that three programs (marked '*' in the figure) fuzzed less than 20 seeds during the campaign. Therefore, we discarded those targets due to a lack of diversity.

In most of the programs, we observe that 20% seeds in the DETERMINISTIC STAGE find 83% of the paths found. In some crypto programs, like asn1parse and bignum, only 1% of the seeds contribute to the coverage. Figure 4 implies that the majority of seeds fail to contribute to new coverage in the DETERMINISTIC STAGE, and 20% of the seeds is sufficient to attain similar coverage.

> **Observation 3:** When measuring the coverage contribution of single seeds, we observe that 20% of the fuzzed seeds contribute to 83% of paths found in the DETERMINISTIC STAGE.

4 MendelFuzz Design

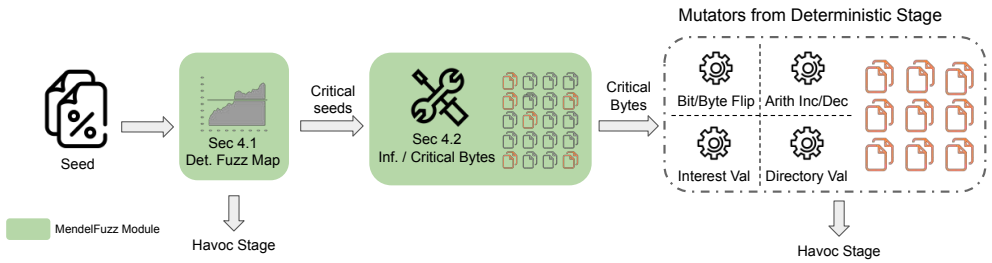


Fig. 5. MendelFuzz Workflow.

The main two takeaways of our study (§3) reveal that (i) less than 1% of the seed bytes, and (ii) 20% of seeds in the queue contributes to around 84% and 83% of the DETERMINISTIC STAGE path findings. Based on these observations, we design MendelFuzz, an optimized DETERMINISTIC STAGE that selectively mutates relevant input bytes and seeds in the queue, thus minimized required mutations. Figure 5 summarizes the MendelFuzz workflow. For every seed to mutate, MendelFuzz uses the DETERMINISTIC FUZZED MAP to select *critical seeds*, *i.e.,* seeds that are more likely to benefit from further mutations (§4.1). The seeds considered not *critical* are directly passed to the havoc stage. For the seeds considered *critical*, MendelFuzz performs quick testing to identify the CRITICAL

Algorithm 1: DETERMINISTIC FUZZED MAP handling.

```
1  MainFuzzWorkflow(Queue)
2      GlobalThreshold ← 0
3      DetFuzzMap ← {}
4      for Seed ∈ Queue do
5          if WorthDetFuzz(Seed, DetFuzzMap) then
6              DetFuzzMap ← DetFuzzMap ∪ Seed.cov_map
7              DeterministicFuzz(Seed, Queue)
8          end
9          HavocFuzz(Seed, Queue)
10     end
11 WorthDetFuzz(Seed, DetFuzzMap)
12     UndetBits ← 0
13     for edge ∈ Seed.cov_map do
14         if edge ∉ DetFuzzMap then
15             UndetBits ← UndetBits + 1
16         end
17     end
18     GlobalThreshold ← DynThreshold(Queue, UndetBits)
19     return UndetBits ≥ GlobalThreshold
20 DynThreshold(Queue, UndetBits)
21     for Seed ∈ Queue do
22         if GlobalThreshold == 0 then
23             GlobalThreshold ← INIT_RATIO × UndetBits
24         end
25         if UndetBits ≥ GlobalThreshold then
26             lastDetTime ← getTime()
27         end
28         else if getTime() − lastDetTime ≥ SEED_TIMEOUT then
29             GlobalThreshold ← REDUCE_RATIO × GlobalThreshold
30         end
31     end
32     return GlobalThreshold
```

BYTES (§4.2), to which apply the original mutators of the DETERMINISTIC STAGE. Finally, the seed is forwarded to the standard HAVOC STAGE.

4.1 Deterministic Fuzzed Map

The DETERMINISTIC FUZZED MAP helps the fuzzer predict whose seeds might contribute to new coverage when the DETERMINISTIC STAGE mutators are applied. The underlying intuition is that the DETERMINISTIC STAGE mutators find new coverage when they are applied to seeds different from each other, where the difference is represented by the seed's execution path. If two seeds exhibit a "similar" execution path, only one seed is processed by MENDELFUZZ, while the other is forwarded to the havoc stage.

Algorithm 1 illustrates our implementation. The deterministic fuzzed map (DetFuzzMap) is a coverage map that contains the cumulative execution traces of all seeds that have been mutated by the deterministic stage mutators (line 10). Notably, the map records the execution paths of the original *seeds*, rather than the mutated *inputs*, as deterministic fuzzed map is specifically designed to estimate the extent of code exploration achieved during the deterministic stage. At the beginning of the fuzzing campaign, the DetFuzzMap is empty. Then, the structure is updated with the execution trace of every seed selected for the deterministic stage mutators (line 6). The WorthDetFuzz routine decides if a seed is sufficiently different from the previous ones by comparing its execution trace and the DetFuzzMap (line 5). The comparison is done by computing the UndetBits, which represents the number of edges belonging to a seed path but do not appear in the DetFuzzMap (line 12 – 19). Seeds with a high UndetBits are considered sufficiently different from previous seeds mutated through the deterministic stage mutators.

The WorthDetFuzz routine determines whether a seed is sufficiently distinct based on a GlobalThreshold. In our prototype, we employ a dynamic threshold via the DynThreshold routine, which periodically updates the GlobalThreshold based on two principles: (i) during the early stages of the campaign, MendelFuzz prioritizes seeds with significant differences, and (ii) as the campaign progresses, MendelFuzz gradually accepts smaller differences. Based on empirical experimentation during MendelFuzz development, we set the INIT_RATIO and REDUCE_RATIO to 0.05 and 0.75, respectively, with a SEED_TIMEOUT of 15 minutes.

The INIT_RATIO and REDUCE_RATIO govern seed selection for deterministic mutation. A low INIT_RATIO or high REDUCE_RATIO may result in an excessive number of seeds being processed, reducing efficiency. The SEED_TIMEOUT ensures that no individual seed consumes excessive resources during the deterministic stage. If the deterministic stage for a seed exceeds the SEED_TIMEOUT, the current mutation state is recorded, allowing the process to resume efficiently if the seed is revisited for further deterministic mutation.

Note that alternative design choices, such as using a fixed threshold or simply disabling the deterministic stage after a fixed timeout, do not yield the same improvement. As MendelFuzz may continuously find critical seeds for the deterministic stage during the whole campaign. We explore this phenomenon in §5.1.

4.2 Critical Bytes

Our observation in Section §3.3 indicates that more than 99% of the input bytes do not contribute to new coverage. Therefore, we first define the bytes that contribute to mutation as critical bytes, and propose an algorithm to approximate them. Since the critical byte identification happens before the mutations, we desire to mark the bytes as fast as possible to reduce overhead. The underlying idea is to replace each input byte with a random value and execute the mutated input. If a new path is found, the byte is marked as critical. A native critical byte calculation samples every byte of the input, leading to O(N) complexity. To speed up the process, we introduce the concept of inference map, which mutates the input bytes in a binary search fashion, thus reducing the time complexity from O(N) to O(logN). One can see the inference map as a faster alternative to the effective map (§2.2), which uses the path checksum as the indicator for redundant bytes. Based on inference map, we identify the critical bytes as per Algorithm 2 and store them in a map to drive the deterministic stage mutators accordingly.

Overhead of Critical Byte Map Calculation. The critical byte calculation is an additional step independent from the original deterministic stage, potentially introducing additional overhead. In the worst case, when all input bytes are deemed critical, we expect len(seed) executions with no bytes pruned. However, this scenario also implies the discovery of len(seed) new paths. The deterministic stage requires 431 ∗ len(seed) executions if no bytes are pruned (*i.e.,* 27 ∗ len

Algorithm 2: Marking CRITICAL BYTE.

```
1  DeterministicFuzz(Queue, Input)
2      InfMap ← calcInfMap(Input)
3      CriticalBytes ← calcCriticalBytes(Queue, Input, InfMap)
4      for Mutator ∈ DeterministicMutators do
5          for Pos ∈ Input and Pos ∈ CriticalBytes do
6              MutateAndExecute(Input, Pos, Mutator)
7          end
8      end
9  calcCriticalBytes(Queue, Input, InfMap)
10     CriticalBytes ← {}
11     for Pos ∈ Input and Pos ∈ InfMap do
12         PrevCount ← Queue.count
13         MutateAndExecute(Input, Pos, XOR)
14         if Queue.count ≠ PrevCount then
15             CriticalBytes ← CriticalBytes ∪ {Pos}
16         end
17     end
18     return CriticalBytes
```

for bit/byte flip, 350 ∗ LEN for arith inc/dec, and 54 ∗ LEN for int replacement). In the worst case, the CRITICAL BYTE calculation introduces 0.23% additional overhead (1/431), which we consider well worth the potential benefits. The INFERENCE MAP calculation cuts down the CRITICAL BYTE calculation cost, as the search scope of CRITICAL BYTE is limited to the EFFECTIVE BYTE. In §5.5, we measure the overhead introduced by MENDELFUZZ.

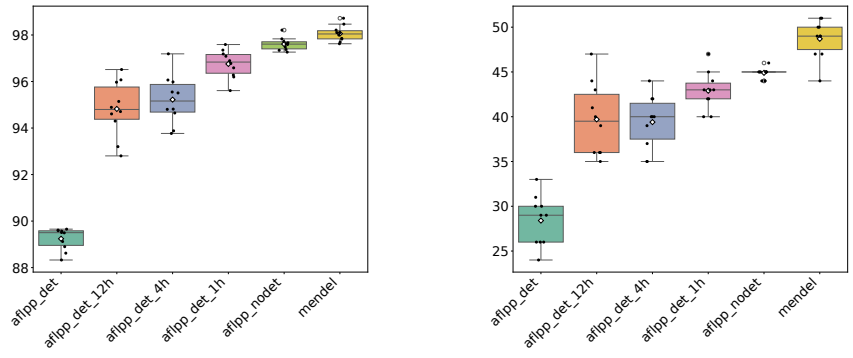
Under-tainted Critical Bytes. If there are additional stages before the CRITICAL BYTE calculation, these preprocessing steps may generate mutated inputs that traverse new program paths. This means that mutating CRITICAL BYTE may not be able to discover new paths, potentially resulting in under-taint risks. For example, in AFL++, RedQueen [Aschermann et al. 2019] stage always executes before the DETERMINISTIC STAGE. If the RedQueen operators find new paths ahead, mutating CRITICAL BYTE may not discover anything new, therefore under-tainting the CRITICAL BYTE. To address this challenge, relocating the CRITICAL BYTE calculation before the RedQueen stage helps.

5 Evaluation

We evaluate MENDELFUZZ to answer the following research questions: (RQ1) Does MENDELFUZZ outperform the baseline? (§5.1) (RQ2) Does MENDELFUZZ outperform fuzzers that enable DETERMINISTIC STAGE? (§5.2) (RQ3) Does MENDELFUZZ outperform fuzzers that only use HAVOC STAGE? (§5.3) (RQ4) Can MENDELFUZZ find unknown bugs? (§5.4) (RQ5) How do MENDELFUZZ's components contribute to its performance? (§5.5) (RQ6) Does MENDELFUZZ improve the efficiency of the DETERMINISTIC STAGE? (§5.6) (RQ7) Can other fuzzers benefit from MENDELFUZZ? (§5.7)

We extend the setup based on §3, and list the changes below:

Benchmarks: We primarily use Magma [Hazimeh et al. 2020] for bug-finding capability evaluation, as UNIFUZZ [Li et al. 2021] does not provide bug deduplicate metric, and FuzzBench [Metzman et al. 2021] is tailored for coverage assessment. Concerning coverage evaluation, we utilize both Magma and FuzzBench. Magma includes 18 programs, while FuzzBench provides 23 applications,



(a) normalized coverage (b) bug discovery

Fig. 6. Normalized coverage and bugs discovered for AFL++ with different DETERMINISTIC STAGE timeout vs MendelFuzz. For instance, aflpp_det_1h means running both stages for 1h and then run havoc only.

both of which cover a wide range of input types and well-constructed initial seed corpus to minimize bias. Given that FuzzBench necessitates massive computational resources, we request the campaign from Google and integrate the results with existing data. We host the generated report at https://github.com/HexHive/MendelFuzz-Artifact. To ensure reproducibility, we repeat 10 runs for Magma and 20 runs for FuzzBench evaluations.

Compared Fuzzers: In Magma, we evaluate fuzzers that enable both the DETERMINISTIC STAGE and the HAVOC STAGE through general purpose fuzzers (AFLFast [Böhme et al. 2016], AFL++ [Fioraldi et al. 2020b]) and byte-level fuzzers (FairFuzz [Lemieux and Sen 2018], Weizz [Fioraldi et al. 2020a]). We also include fuzzers that exclusively run the HAVOC STAGE (MOpt-AFL). For FuzzBench, we compare MendelFuzz against the integrated SoTA fuzzers(AFL++ [Fioraldi et al. 2020b], Honggfuzz [honggfuzz 2019], LibFuzzer [libfuzzer 2023], AFLSmart [Pham et al. 2019], Eclipser [Choi et al. 2019], and Centipede [centipede 2023]). All the FuzzBench fuzzers are configured as recommended, *i.e.,* only running HAVOC STAGE [Metzman et al. 2021]. As MendelFuzz extends AFL++, MendelFuzz inherits AFL++'s havoc implementation.

Bug Metric: We collect "Unique Bug" data according to the definition of Magma. Moreover, Magma already contains well-defined bug oracles that are independent of sanitizer, which we disable for fair performance.

Coverage Metric: For coverage, we replay all seed queues on AFL++-instrumented binaries using "afl-showmap" to obtain collision-free edge coverage. We further normalize the coverage in accordance with FuzzBench's practice [Metzman et al. 2021], *i.e.,* score = $\text{edge}_f / \text{edge}_{\max} * 100$, where edge_f represents a fuzzer's absolute edge coverage in a single run, and $\text{edge}_{\max}$ denotes the highest absolute edge coverage among all the fuzzers across all runs.

0day on OSS-Fuzz: OSS-Fuzz [google 2023] is a Google service that continuously tests the security-critical open-source libraries. Following best practices [Böhme and Falk 2020; Chen et al. 2020; Gan et al. 2018], we pick five programs from OSS-Fuzz that cover diverse input formats (*i.e.,* font, document, image, network packages) and use the latest commit for MendelFuzz testing.

5.1 RQ1) Does MendelFuzz outperform the baseline?

MendelFuzz is implemented based on AFL++. To demonstrate the effectiveness of MendelFuzz, we evaluate MendelFuzz against AFL++ with and without DETERMINISTIC STAGE enabled. Furthermore,

we include alternative AFL++ implementations to prove that the performance improvement of MendelFuzz attributes exclusively to our design.

A naive implementation of MendelFuzz may simply disable the deterministic stage after a fixed timeout. However, we observe that generated inputs can benefit the deterministic stage at any time of the campaign. To understand this phenomenon, we compare MendelFuzz against AFL++ with different deterministic stage timeout settings. In particular, we enable both deterministic stage and havoc stage for the fixed timeout, then opt for havoc only.

We run AFL++ with four different timeout for deterministic stage, *i.e.,* 24h, 12h, 4h, 1h (aflpp_det_$timeout) and 0h (aflpp_nodet – havoc-only) to compare against MendelFuzz. Figure 6 illustrates the finding. Decreasing the deterministic stage timeout notably boosts the AFL++ edge-finding and bug-finding capabilities. However, even deterministic stage with 1h timeout (aflpp_det_1h) performs worse than havoc-only (aflpp_nodet). Conversely, MendelFuzz outperforms any AFL++ timeout setting and havoc-only. Consequently, Figure 6 demonstrates that MendelFuzz enhances the deterministic stage, leading to a better performance than the vanilla havoc stage only or alternative implementations (*i.e.,* aflpp_det_$timeout).

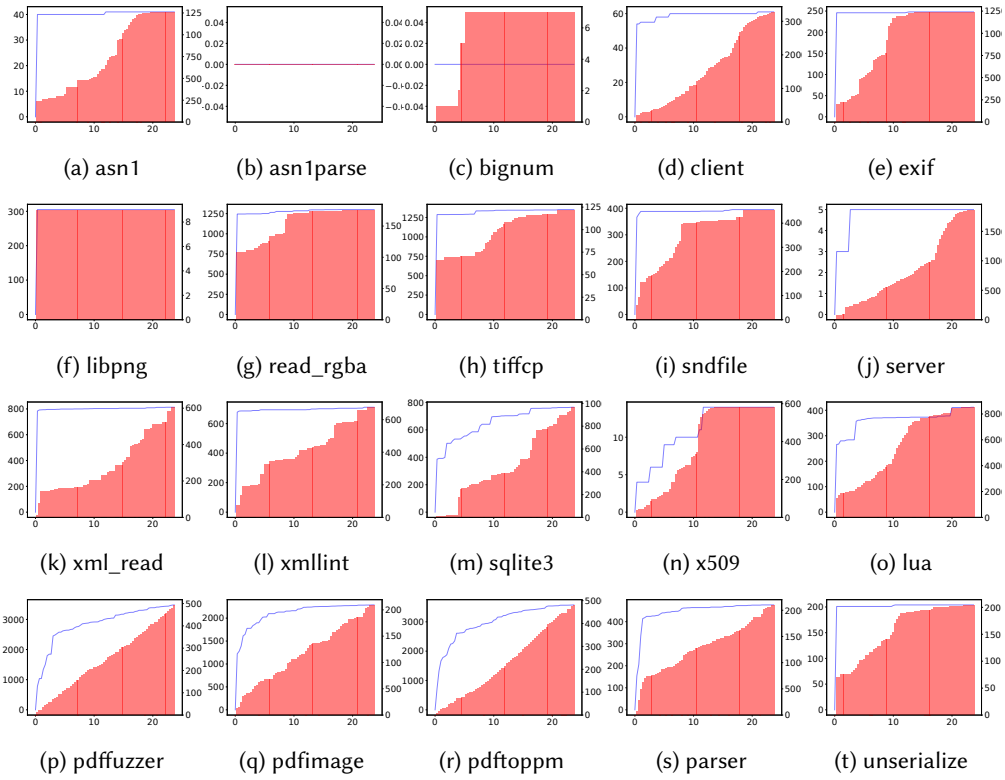
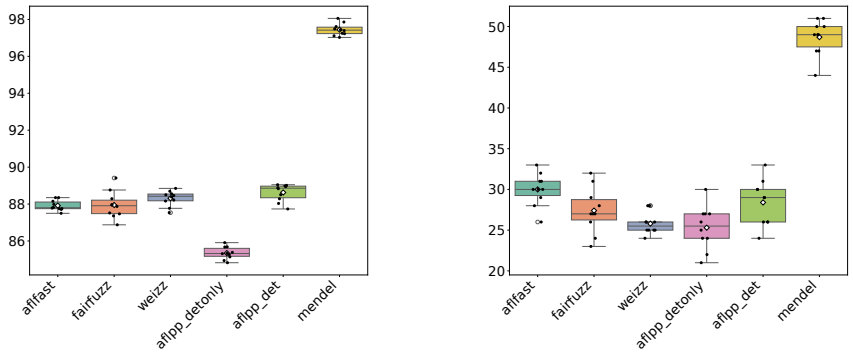


Fig. 7. Time spent and new edges found by the deterministic stage in 24h a campaign. The blue line shows the edges found by the deterministic stage, while the red box is the total time spent in the deterministic stage during the fuzzing campaign.

Figure 7 shows the accumulated time MendelFuzz spends on deterministic stage during the fuzzing campaign (red box) and total new edges discovered by deterministic stage (blue



(a) normalized coverage (b) bug discovery

Fig. 8. Normalized coverage and unique bugs found by fuzzers that enable DETERMINISTIC STAGE, the experiment is conducted in Magma benchmark and repeated for 10 runs. aflpp_detonly is AFL++ only enable DETERMINISTIC STAGE and aflpp_det is AFL++ enable both two stages.

line). Simple programs like libpng, bignum, and server only need DETERMINISTIC STAGE in the beginning. After the first hour, DETERMINISTIC STAGE does not yield any new edge findings. However, regarding more complex programs, the optimized DETERMINISTIC STAGE continuously contributes new coverage (*i.e.,* pdffuzzer and sqlite3). These results demonstrate our motivation: some programs benefit from deterministic mutations more than others. Instead of predicting ahead of the campaign, MendelFuzz dynamically switches between DETERMINISTIC STAGE and HAVOC STAGE during fuzzing. Figure 7 reveals the fundamental differences between MendelFuzz and AFL++ timeout settings.

> **Takeaway:** MendelFuzz proposes a dynamic DETERMINISTIC STAGE tuning algorithm that differs and outperforms any DETERMINISTIC STAGE timeout setting.

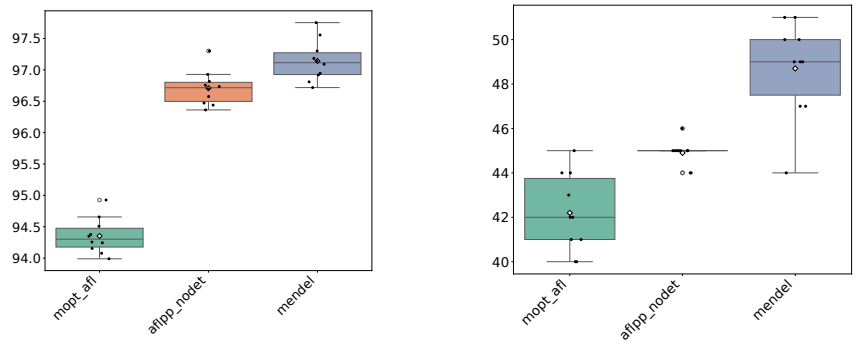
5.2 RQ2) Does MendelFuzz outperform fuzzers that enable DETERMINISTIC STAGE?

We evaluate MendelFuzz against fuzzers that use DETERMINISTIC STAGE, *i.e.,* where both stages are enabled by default (deterministic fuzzers in short). Specifically, we run all fuzzers on Magma, replay the generated queue on AFL++ instrumented binaries for edge coverage and analyze the Magma's bug canaries to obtain the number of unique bugs. The results are illustrated in Figure 8.

Overall, MendelFuzz significantly outperforms the deterministic fuzzers. As illustrated on Figure 8b, MendelFuzz finds 48.7 unique bugs in Magma, while the best deterministic fuzzer AFLFast only achieves 30.0 bugs,[1] MendelFuzz outperforms 62.33% in terms of bug-finding capability. Moreover, among 18 Magma targets, MendelFuzz performs the best in 13 targets. Compared to aflpp_det and aflpp_detonly (*i.e.,* AFL++ with two stages enabled and AFL++ that only enable DETERMINISTIC STAGE), MendelFuzz is only defeated in two programs (server and exif, aflpp_det finds 0.2 more unique bugs on both two programs while aflpp_detonly finds 0.5 more on both targets).

In Figure 8a, MendelFuzz shows a notable boost in terms of coverage. We first collect the seed queue from each campaign and replay them on AFL++ instrumented binaries to dump raw coverage. Then we normalize the data following the FuzzBench [Metzman et al. 2021] approach, which assigns each program the same weight regardless of their absolute edge coverage, as we

[1]Weizz times out for sndfile, lua, and exif



(a) normalized coverage

(b) bug discovery

Fig. 9. Normalized coverage and unique bugs found by fuzzers only running HAVOC STAGE, the experiment is conducted in Magma benchmark and repeated for 10 runs.

described in §5. Figure 8 illustrates that MendelFuzz introduces around 10% boost over other deterministic fuzzers. Additionally, MendelFuzz achieves the highest coverage in 12 out of 18 programs. While all deterministic fuzzers have similar coverage performance (around 88%), only MendelFuzz shows a notable enhancement.

> **Takeaway:** MendelFuzz improves 10% edge findings and around 62.33% unique bugs by optimizing the DETERMINISTIC STAGE.

5.3 RQ3) Does MendelFuzz outperform fuzzers that only use HAVOC STAGE?

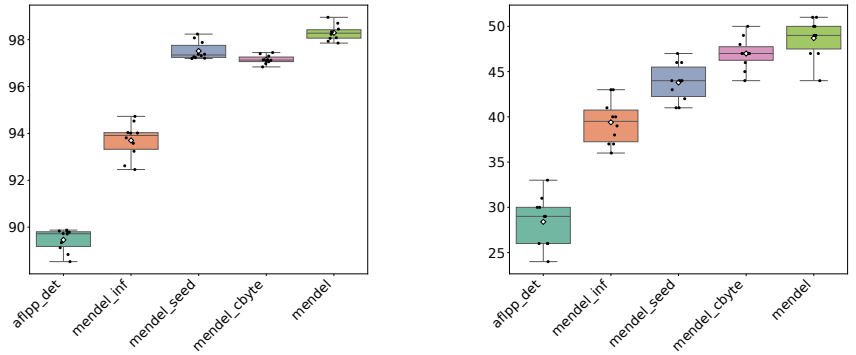
We compare MendelFuzz against the fuzzers that only use HAVOC STAGE (havoc-only fuzzers in short) in Magma, *i.e.,* Mopt-AFL and AFL++ with DETERMINISTIC STAGE disabled (the default AFL++ setup before the MendelFuzz integration), which we also called mopt_afl and aflpp_nodet. Additionally, we evaluate MendelFuzz against havoc-only fuzzers in FuzzBench [Metzman et al. 2021]. We request a 20 runs campaign from Google and include their baseline fuzzers from the public data (Honggfuzz [honggfuzz 2019], LibFuzzer [libfuzzer 2023], AFLSmart [Pham et al. 2019], Eclipser [Choi et al. 2019], and Ceptipede [centipede 2023]). All fuzzers in the FuzzBench are configured as havoc-only fuzzers.

Figure 9b illustrates the unique bugs found by each fuzzer. While deterministic fuzzers can find 30 unique bugs at most, havoc-only fuzzers perform notably better: both mopt_afl and aflpp_nodet find more than 40 bugs. Nevertheless, MendelFuzz finds 15.13% and 8.46% more bugs over mopt_afl and aflpp_nodet, respectively. Moreover, aflpp_nodet, the best havoc-only fuzzer, finds 46 bugs in the best trail, while MendelFuzz finds more than 47 bugs in 9 out of 10 trails. To sum up, MendelFuzz reliably discovers more bugs compared to the best havoc-only fuzzer.

Figure 9a shows the covreage results. Similarly to FuzzBench, we normalize the coverage on Magma (see §5). Overall, MendelFuzz achieves 97.14% while mopt_afl and aflpp_nodet achieve 94.35% and 96.71%, respectively. Even with less effect, MendelFuzz contributes to reach more coverage in Magma as well. For the FuzzBench coverage campaign, we host the full report at https://github.com/HexHive/MendelFuzz-Artifact. Among all 11 state-of-the-art fuzzers, MendelFuzz ranks first both in normalized coverage score and ranking. To sum up, MendelFuzz improves the performance compared to aflpp_nodet and ranks 1st among all fuzzers.

Table 2. Unknown bugs found by MendelFuzz in five real-world applications. Some issues contain multiple bugs while only one CVE is assigned, we count each bug in the report.

project	Bug ID
wireshark	CVE-2024-0209, CVE-2024-0210, issue 19501 #BUG0, issue 19577
libredwg	CVE-2023-36271, CVE-2023-36272, CVE-2023-36273, CVE-2023-36274
libjpeg	CVE-2023-37836. CVE-2023-37837
liblouis	issue 1357 - issue 1361
libjxl	issue 2661 #BUG0 - #BUG5
total	21 bugs, 8 CVEs



(a) normalized coverage (b) bug discovery

Fig. 10. Ablation study of MendelFuzz for 10 runs against Magma benchmark.

> **Takeaway:** MendelFuzz, by combining the optimized deterministic stage and vanilla havoc stage, outperforms havoc-only fuzzers both in coverage and bug discovery.

5.4 RQ4) Can MendelFuzz find unknown bugs?

We challenge the MendelFuzz capability in finding unknown vulnerabilities. In this experiment, we deploy MendelFuzz over widely used applications (*e.g.,* liblouis as Apple Device font library and WireShark as network protocol tools) and successfully find 21 new bugs, among which 8 CVEs get assigned (Table 2). Since all the libraries originate from Google's OSS-Fuzz [google 2023], we consider them to be exhaustively tested by state-of-the-art fuzzers and finding 0-day vulnerabilities demonstrate that MendelFuzz outperform other SoTAs in terms of bug-finding. Therefore, we do not have baseline/SoTA fuzzers in this evaluation. Compared to the OSS-Fuzz that drop the whole deterministic stage, MendelFuzz introduces an improved deterministic stage with finer-grained analysis, thus improving the probability of finding bugs on well-tested targets.

> **Takeaway:** MendelFuzz finds 21 unknown vulnerabilities in security-critical applications.

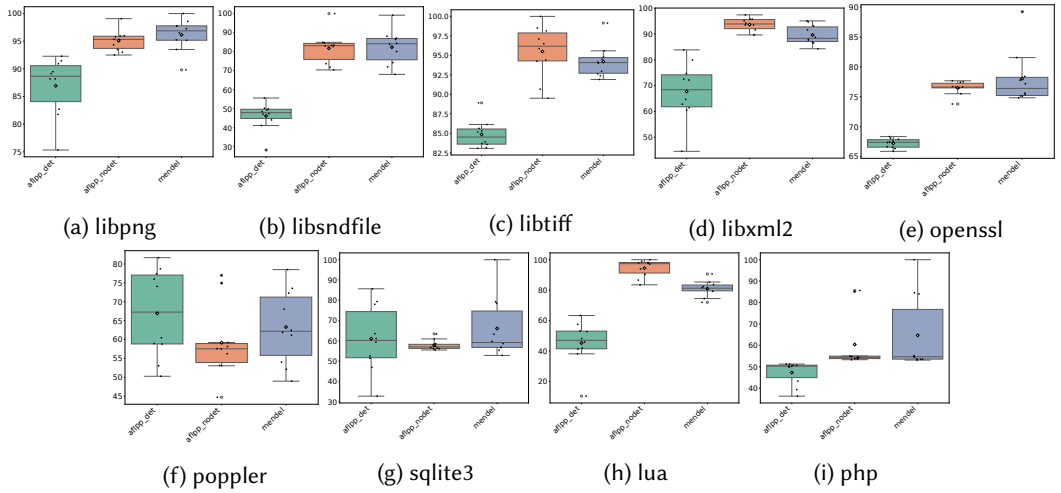


Fig. 11. Normalized throughput of MendelFuzz and AFL++-based fuzzer among 10 trails on Magma.

5.5 RQ5) How do MendelFuzz's components contribute to its performance?

We study the contribution of each MendelFuzz component. In the study, we use AFL++ with only the deterministic stage enabled (aflpp_det) as baseline, mendel_inf (inference map), mendel_seed (deterministic fuzzed map) and mendel_cbyte (critical byte) are the aflpp_det with one MendelFuzz module enabled, and mendel is MendelFuzz with all three modules. All the fuzzers are evaluated on Magma for 10 runs. As for the previous experiments, we normalize the coverage as described in §5.

Figure 10 demonstrates that all three components help improving coverage and bug-finding capabilities. Compared to the baseline aflpp_det, the inference map improves more than 4% of the coverage discovery, the deterministic fuzzed map and critical byte help discovering about 8% more edges. Moreover, aflpp_det with inference map exposes 39.4 unique bugs in the evaluation, deterministic fuzzed map and critical byte assists aflpp_det finds 43.8 and 47 unique bugs, while the aflpp_det standalone only discover 28.4 bugs. In short, all three components, including inference map, deterministic fuzzed map, and critical byte, notably contribute to fuzzing performance.

However, we notice that the combination MendelFuzz, despite ranking 1st in both coverage and bug discovery, fails to yield a substantial boost when compared to critical byte or deterministic fuzzed map. The possible explanation is any component greatly narrows the search space, therefore limiting the room for optimization. To tailor MendelFuzz towards more complex targets and initial corpus, we keep each module enabled by default.

We also investigate if MendelFuzz introduces runtime overhead. As MendelFuzz does not modify the instrumentation, the same input should yield the same execution speed. The primary differences originate from the type of inputs generated by different mutation algorithms of two stages. Figure 11 illustrates the normalized throughput of AFL++ with deterministic stage (aflpp_det), only havoc stage (aflpp_nodet), and MendelFuzz. In more than half of the programs (a - e), MendelFuzz have nearly the same throughput as the apflpp_nodet, in which MendelFuzz invests little time in deterministic stage according to Figure 7. In contrast, for those benchmarks that MendelFuzz spends more time on deterministic stage (*i.e.,* poppler, sqlite3), MendelFuzz's

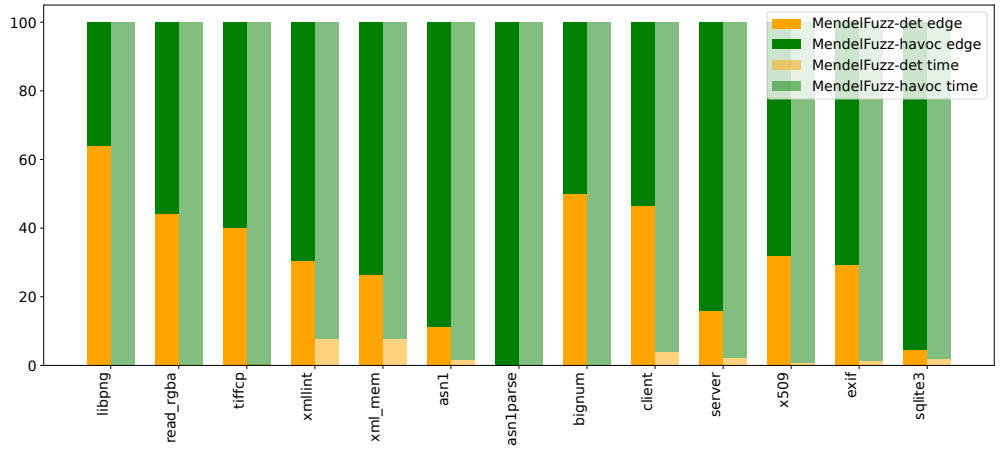


Fig. 12. Time spent and edges found by the The efficiency of MendelFuzz and have in terms of edge finding and time consumption. The experiemtn was conducted in a 24h Magma campaign.

throughput is closer to aflpp_det. Overall, MendelFuzz's throughput is between aflpp_det and aflpp_nodet, depending on amount of time MendelFuzz investing for the deterministic stage.

> **Takeaway:** All MendelFuzz componets (critical bytes, det fuzzed map and inf map) significantly boost deterministic stage (65.5%, 54.2% and 38.7%) with an acceptable overhead.

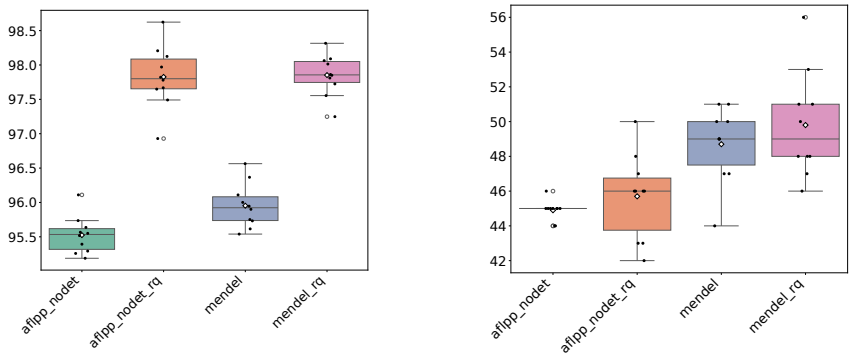
5.6 RQ6) Does MendelFuzz improve the efficiency of the deterministic stage?

To understand whether MendelFuzz improves the efficiency of the deterministic stage, we conduct a dedicated study to measure the time consumption and coverage findings of the optimized deterministic stage and the unmodified havoc stage in MendelFuzz, respectively.

Figure 12 presents the ratio of the edge finding and time consumption between the two stages. The light and dark green bars represent the time spent and edges found by havoc stage, while the light and dark orange bars represent the time spent and the edges found by MendelFuzz optimized deterministic stage. Among 18 targets, deterministic stage in MendelFuzz only contributes more than 50% edges in libpng and bignum. In most targets, deterministic stage in MendelFuzz only introduces around 30% to 50% of the new coverage, while contributes up to 60% coverage discovery in the campaign. Whatsmore, we notice that the optimized deterministic stage in MendelFuzz is significantly faster than the original one. As depicted at Figure 2, vanilla deterministic stage cost more than 90% of the fuzzing time. In MendelFuzz, the optimized deterministic stage only takes up to 10% times while retaining similar coverage findings.

There are some exceptions in Figure 12, *e.g.,* in asn1parse, MendelFuzz only finds one new edge in the whole campaign and fails to discover any new edge in the deterministic stage. However, for asn1parse, MendelFuzz's deterministic stage spends less than 1% of the testing time, thus incurring a negligible overhead. While MendelFuzz cannot guarantee that all programs benefit from deterministic stage, our strategy minimizes the deterministic stage cost.

> **Takeaway:** In the whole campaign, the optimized deterministic stage in MendelFuzz takes less than 10% of the time and finds up to 60% of the new coverage.



(a) normalized coverage

(b) bug discovery

Fig. 13. MendelFuzz and AFL++ with/without RedQueen mutators. The "_rq" stands for the MendelFuzz and AFL++ with RedQueen mutator enabled.

5.7 RQ7) Can other fuzzers benefit from MendelFuzz?

As a better implementation for deterministic stage, MendelFuzz can cooperate with other techniques and enhance the overall performance. To illustrate its potential, we evaluate MendelFuzz in conjunction with RedQueen [Aschermann et al. 2019], the default SoTA mutators in AFL++.

RedQueen [Aschermann et al. 2019] is a checksum-solving mutator in greybox fuzzing. Compared to concolic executions [Chen et al. 2022; Poeplau and Francillon 2020; Yun et al. 2018] or taint analysis [Chen and Chen 2018; Rawat et al. 2017], RedQueen is more lightweight and can scale to large binaries. While MendelFuzz has a similar implementation as RedQueen (*i.e.,* prioritize the critical location and mutate accordingly), they are orthogonal techniques. To demonstrate this fact, we conduct a 10 run study against Magma benchmark.

Figure 13 illustrate how MendelFuzz and aflpp_nodet works with RedQueen mutators. Overall, the RedQueen mutator significantly improves the coverage performance of both MendelFuzz and aflpp_nodet. AFL++ with RedQueen enabled, i.e., aflpp_nodet_rq, successfully overcomes MendelFuzz standalone (mendel). However, if MendelFuzz enable the RedQueen mutator (mendel_rq), it slightly achieves higher coverage compared to aflpp_nodet_rq.

In terms of bugs, the RedQueen mutator slightly increase aflpp_nodet's bug discovery capability (1.8%). But even comparing aflpp_nodet_rq against MendelFuzz standalone, the number of unique bugs discovered is still 6.6% lower. This experiment proves that RedQueen benefits from MendelFuzz in terms of bug detection. Additionally, we notice that RedQueen improves about 2.3% unique bug discovery for MendelFuzz.

> **Takeaway:** MendelFuzz is an orthogonal techniques for RedQueen and can be used in conjunction with RedQueen for better performance.

6 Threats to Validity

While MendelFuzz significantly boost over existing approaches, some factors may affect the results.
Internal Threats. MendelFuzz predicts *critical* bytes and seeds for the deterministic stage mutators, narrowing the search space and potentially missing vulnerabilities. However, the deterministic stage ensures that seeds very similar to each other (e.g., differing only by one byte) are processed, thus improving the chances of finding critical bytes. Our evaluation demonstrates that

MendelFuzz has a higher likelihood of discovering such vulnerabilities compared to havoc-only fuzzers.

External Threats. The design of MendelFuzz is based on observations from Magma, which is susceptible to the risk of overfitting. To address this risk, we validate the design using both Magma and Fuzzbench, which together encompass 41 programs representing diverse targets. Both benchmarks are largely adopted by the fuzzing research community that continuously update them with representative programs and vulnerabilities. The adoption of a systematic approach, where we employ third-party benchmarks and decouple study and validation, reduces the risk of overfitting. Additionally, we conduct 10 trials for each campaign, adhering to the recommendation in [Schloegel et al. 2024], except for FuzzBench, which provides 20 runs per request. These repetitions help mitigate the threat of randomness.

7 Related Work

7.1 Havoc-Only Fuzzers

As observed in previous works [Metzman et al. 2021; Wu et al. 2022], disabling the deterministic stage enhances the fuzzer's performance. Researchers have spent lots of effort in improving the havoc stage. Mopt-AFL [Lyu et al. 2019] notices that different mutators perform differently across various programs. Mopt-AFL proposes a Particle Swarm Optimization algorithm (PSO) that schedules the havoc stage mutators. Similarly, EMS [Lyu et al. 2022] uses the fuzzing history to schedular the best mutator. Other havoc stage optimizations focus on better models. EcoFuzz [Yue et al. 2020] and HavocMAB [Wu et al. 2022] conceptualize the whole fuzzing campaign as a Multi-Armed Bandit model and find a better trade-off between in software testing. Entropic [Böhme et al. 2020], on the other hand, balances the energy allocation by maximizing the entropy. Compared to the previous works, MendelFuzz focuses on the deterministic stage, demonstrating this strategy can be used to reach new paths. MendelFuzz can enhance havoc-only fuzzers.

7.2 Input Structure Inference

Despite deterministic stage's pitfall performance, recent works [Fioraldi et al. 2020a; Gan et al. 2020; Lemieux and Sen 2018; Zhang et al. 2023] have noticed the deterministic stage's potential for input structure inference. In particular, the deterministic stage's feedback may indicate the role of specific bytes, *e.g.,* in the header or as data chunk, thus facilitating the construction of the input structure without manual intervention. FairFuzz [Lemieux and Sen 2018] proposes a RareBranch-based masking that prevents fuzzer from disrupting the input structure. GREYONE [Gan et al. 2020] infers constraints by iterating over all the bytes deterministically, monitors the state of the cmp register, and populates correct values to the data chunk. Weizz [Fioraldi et al. 2020a] introduces a new deterministic stage, named Surgical, to automatically infer the input byte dependencies. ShapFuzz [Zhang et al. 2023] learns the input format from the deterministic stage and then enhances the havoc stage with this prior knowledge. The goal of MendelFuzz is not to infer the input structure, but simply direct the deterministic stage mutators toward those bytes that are more likely to reach new coverage. Most imprtantly, MendelFuzz introduces a neglitable overhead.

8 Conclusion

Preserving correct input semantic is crucial to effectively fuzzing programs. Even though deterministic stage has been designed to automaticaly mutate inputs while preserving their semantic, its implementations is too slow, leading developers to move toward havoc stage, which provide more destructive mutations but better performances.

In our work, we propose MendelFuzz, an optimized deterministic stage that identifies the critical bytes and critical seeds in the deterministic stage, which reduces unnecessary mutations to retain the benefit of the (old) deterministic stage without its high cost. Our evaluation shows that MendelFuzz improves both the reached coverage and the bugs found in Magma and FuzzBench.

9 Data Availability

To enable full replication of all presented results, we integrate MendelFuzz into main line AFL++ and release all supporting materials at https://github.com/HexHive/MendelFuzz-Artifact.

Acknowledgments

We thank Daniele Cono D'Elia, Qiang Liu, Qinying Wang, the rest of the HexHive, and the anonymous reviewers for their detailed feedback. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2 186974, a gift from Intel Corporation, and funding from BMK, BMAW, the State of Upper Austria in the frame of the COMET Module DEPS (grant no. 888338) managed by FFG and ERC grant (Project AT SCALE, 101179366). This work is funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

AFL. 2019. technical_details.txt. https://github.com/google/AFL/blob/master/docs/perf_tips.txt.

AFLplusplus. 2025. AFLplusplus: edge coverage and collision-free coverage. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.llvm.md.

Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 713–724.

Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 678–689.

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.

centipede. 2023. centipede. https://github.com/google/centipede.

Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. 2022. {SYMSAN}: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. 2531–2548.

Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596.

Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 736–747.

Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020a. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 1–13.

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020b. AFL++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*. 10–10.

FuzzBench. 2020a. FuzzBench: 2020-03-11 report. https://www.fuzzbench.com/reports/paper/AFL-Deterministic-Experiment/index.html.

FuzzBench. 2020b. FuzzBench: 2021-04-23-paper report. https://www.fuzzbench.com/reports/paper/Main-Experiment/index.html.

Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. {GREYONE}: Data flow sensitive fuzzing. In *29th USENIX security symposium (USENIX Security 20)*. 2577–2594.

Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.

Google. 2023. FuzzBench Evaluation Report. https://fuzzbench.com/reports/experimental/2023-04-27-main/index.html.

google. 2023. oss-fuzz progress in 2023.txt. https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html.

Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

Marc Heuse. 2023. magma is using outdated afl++. https://github.com/HexHive/magma/pull/142.

honggfuzz. 2019. honggfuzz. https://github.com/google/honggfuzz.

Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.

Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. {UNIFUZZ}: A Holistic and Pragmatic {Metrics-Driven} Platform for Evaluating Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. 2777–2794.

libfuzzer. 2023. libfuzzer. https://llvm.org/docs/LibFuzzer.html.

Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers.. In *USENIX Security Symposium*. 1949–1966.

Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. 2022. Ems: History-driven mutation for coverage-based fuzzing. In *29th Annual Network and Distributed System Security Symposium. https://dx. doi. org/10.14722/ndss*, Vol. 10.

Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1393–1403.

Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1980–1997.

Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with {SymCC}: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. 181–198.

Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing.. In *NDSS*, Vol. 17. 1–14.

Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. Sok: Prudent evaluation practices for fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1974–1993.

Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective seed scheduling for fuzzing with graph centrality analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2194–2211.

Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering*. 1634–1645.

Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 2307–2324.

Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

Michal Zalewski. 2013. american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

Kunpeng Zhang, Xiaogang Zhu, Xiao Xi, Minhui Xue, Chao Zhang, and Sheng Wen. 2023. SHAPFUZZ: Efficient Fuzzing via Shapley-Guided Byte Selection. *arXiv preprint arXiv:2308.09239* (2023).

Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. {FISHFUZZ}: Catch Deeper Bugs by Throwing Larger Nets. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1343–1360.

Received 2024-08-27; accepted 2025-01-14