Profile-Driven System Optimizations for Accelerated Greybox Fuzzing

Yunhang Zhang[†] Chengbin Pang[‡,1] Stefan Nagy[†] Xun Chen[*] Jun Xu[†]
[†]University of Utah [‡]Nanjing University [*]Samsung Research America

ABSTRACT

Greybox fuzzing is a highly popular option for security testing, incentivizing tremendous efforts to improve its performance. Prior research has brought many *algorithmic advancements*, leading to substantial performance growth. However, less attention has been paid to the *system-level designs* of greybox fuzzing tools, despite the high impacts of such designs on fuzzing throughput.

In this paper, we explore system-level optimizations for greybox fuzzing. Throughout an empirical study, we unveil two system-level optimization opportunities. First, the common fuzzing mode with a fork server visibly slows down the target execution, which can be optimized by coupling persistent mode with efficient state recovery. Second, greybox fuzzing tools rely on the native Operating System (OS) to support interactions issued by the target program, involving complex but fuzzing-irrelevant operations. Simplification of OS interactions represents another optimization opportunity.

We develop two techniques, informed by a short profiling phase of the fuzzing tool, to achieve the optimizations above. The first technique enables reliable and efficient persistent mode by learning critical execution states from the profiling and patching the target program to reset them. The second technique introduces user-space abstractions to simulate OS functionality, reducing expensive OS interactions. Evaluated with 20 programs and the MAGMA benchmark, we demonstrate that our optimizations can accelerate AFL and AFL++ for higher code coverage and faster bug finding.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Greybox Fuzzing, System Optimizations, Profile-Driven

ACM Reference Format: Yunhang Zhang, Chengbin Pang, Stefan Nagy, Xun Chen, Jun Xu. 2023. Profile-Driven System Optimizations for Accelerated Greybox Fuzzing. In ACM Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark. ACM, New York, NY, USA, 15 page. https://doi.org/xxxxx.

1 INTRODUCTION

Greybox fuzzing [64, 66, 69, 88] is a useful method for security testing. It works by continuously mutating existing test cases to produce new ones for exercising the target software. In recent years, both research and adoption of greybox fuzzing have grown tremendously, thanks to the availability of two generic, easily extendable greybox fuzzing tools: AFL [86] and AFL++ [54].

Motivation: A major development goal of greybox fuzzing is to improve its performance — typically measured by code covered or bugs found in a given time window. In principle, this goal is dependent on both *algorithm factors* and *system factors*. Algorithm factors affect the scheduling of test cases (i.e., which test cases to mutate first and how long a test case should be mutated), mutation of test cases (i.e., how to mutate a test case), and retaining of test cases (i.e., what criteria to follow for keeping test cases). The majority of prior research focuses on improving these factors, leading to algorithmic advancements and a performance leap of greybox fuzzing [40, 47, 48, 67, 68, 71, 72, 80, 87**?**].

In contrast, system factors determine the time needed to complete a fuzzing iteration (or precisely, the process to generate, execute, and process a test case). Enhancing the system factors can bring a higher iteration frequency and, thus, a better fuzzing performance. However, less research has been conducted in this direction, motivating us to explore and optimize the system factors.

Study: To understand the optimization opportunities behind the system factors, we perform a quantitative study on AFL with a set of benchmark programs listed in Table 1. The study unveils two optimization opportunities which we explain in the following.

Optimization I: By default, both AFL and AFL++ run in the *fork server* mode, where they spawn a child process of the target program to execute each test case. The fork server mode can slow down the target execution in two ways. First, forking a child process takes time due to operations like page table duplication. Our study in §2.2 shows that the fork alone can consume 4% of the total time of a fuzzing iteration. Second, the child process involves redundant operations that are meaningless to fuzzing, such as page faults due to copy-on-write and cleanups when terminating the process.

Targeting **Optimization I**, AFL and AFL++ incorporate the *persistent mode* [26, 27], where they insert a loop into the target program that continuously runs different test cases without an exit. In this mode, no forking is needed, and the aforementioned redundant operations no longer occur. However, our study shows that the persistent mode often fails to run a fuzzing target because the effects left over by previous test cases interrupt the execution of follow-up fuzzing. To re-enable the persistent mode in those cases, a possibility is to use the *snapshot mode* proposed in recent research [75, 84]. Before running any test case, the snapshot mode saves a copy of the execution states. Once finishing a test case, it resets the affected states before switching to the next test case. However, as we will unveil in §2.3, the snapshot mode—due to operations like coarse-grained memory tracking and recovery—still visibly offsets the efficiency of the persistent mode.

Optimization II: During fuzzing, the target program can also issue interactions with the Operating System (OS) for goals like fetching the test case. Both AFL and AFL++ relay those interactions

[1]Pang contributed to this work while visiting Stevens Institute of Technology.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN XXXX…$XXX
https://doi.org/XXXX

to the native OS. However, modern OSes involve complex operations that are expensive but unnecessary for fuzzing. According to our study, the time spent on interactions with the OS can account for 16.72% of the total program execution time when running AFL (see Table 13 in Appendix). This illustrates another under-explored opportunity for optimization: *we can accelerate the target program execution by simplifying the OS interactions.*

Our Approach: In this paper, we aim to achieve **Optimization I** and **Optimization II**. Our key insight is that running the greybox fuzzing tool for a short period of time—a *profiling phase*—can help us gain the information needed to fulfill the desired optimizations. Following this insight, we design two techniques targeting **Optimization I** and **Optimization II**, respectively.

Profile-driven State Recovery: In the persistent mode, the states must be recovered to avoid crashes are global data (broadly defined, including global variables, arguments of `main` function, and environment variables, etc.). By dynamically analyzing the target program with test cases from the profiling phase, we understand what global variables are manipulated and then patch the target program to reset those global variables before running a test case. This enables us to reset the critical states and stabilize the persistent mode with high efficiency. Considering that we may miss global variables newly covered in post-optimization fuzzing, our profiling-and-patching process can be configured to run periodically.

Profile-driven OS Abstraction: Given specific fuzzing settings, the OS interactions from the target program are often predictable. We can build an understanding of those OS interactions by executing and tracing the test cases from the profiling phase. After replacing the observed OS interactions with in-process, behavior-preserving but minimized operations, we can significantly eliminate the cost of OS interactions. In particular, we create a virtual file system (VFS), in the form of a user-space library, to handle files identified in the profiling phase. During fuzzing, the VFS is linked to the target program and handles operations to the files it covers. For instance, OBJDUMP reads data from both the test case and `/usr/share/locale/locale.alias` on the disk. Our VFS can include a copy of the two files and intercept accesses to them with user-space operations. To cover a broader spectrum of system calls, we further extend the VFS to support socket operations.

Evaluation: We evaluate our two optimizations with 20 popular fuzzing benchmarks on both AFL and AFL++. In 7 cases where the persistent mode fails to run, our profile-driven state recovery successfully re-enables it. Compared to the fork server mode, the re-enabled persistent mode brings a 83%/206% increase in execution speed with AFL/AFL++. Coupled with the persistent mode (native or enabled by us), our profile-driven OS abstraction accelerates the execution speed by 35%/120% given AFL/AFL++. Considering that the snapshot mode represents the best option when the persistent mode is inapplicable, we compare the snapshot mode and the combination of our two optimizations. Our combined optimizations run 82.1%/59.9% faster than the snapshot mode on AFL/AFL++ and cover 5.3%/17.4% more code. We also compare our combined optimizations with the default fork server mode. Given AFL/AFL++, our optimizations increase the execution speed by 142%/381%, covering 8.8%/9.0% more code. We further run an evaluation on the MAGMA benchmark [59]. It shows that our optimizations enable AFL/AFL++ to discover not only more bugs but also in a faster manner.

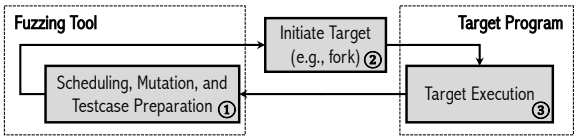


Figure 1: A typical workflow of greybox fuzzing.

Contribution: In summary, our contributions are as follows.

- We perform a study to understand how the system designs of greybox fuzzing tools affect their efficiency. The study further unveils two optimization opportunities behind the designs.
- We propose the idea of profile-driven optimizations to enhance the system designs of greybox fuzzing tools. Following this idea, we create two techniques to realize the optimization opportunities identified in our study. The two techniques represent the first of their kind.
- We implement the two optimization techniques on top of AFL and AFL++. Evaluating the two techniques with 20 common benchmark programs and MAGMA, we show that our optimizations can significantly increase the fuzzing speed of both AFL and AFL++, benefiting their code coverage and bug finding. Our code has been anonymized and released at https://anonymous.4open.science/r/Profile-guided-Fuzzing-4F4B.

2 BACKGROUND AND MOTIVATION

2.1 Greybox Fuzzing

Greybox fuzzing [54, 86] is an iterative process, following a typical workflow presented in Figure 1. In each iteration, the fuzzing tools perform scheduling (when needed) and mutations of existing test cases to derive a new one, based on feedback from previous fuzzing iterations (**Step** ①). Given a ready test case, the fuzzing tool initiates the target program, such as forking a child process, for running the test case (**Step** ②). The target program follows up to finish the test case and "return" the feedback (e.g., code coverage and crashes) to the fuzzing tool (**Step** ③).

The performance of greybox fuzzing—usually measured by code covered or bugs found in a given time window—depends on both *algorithmic factors* and *system factors*.

Algorithmic factors mainly consist of scheduling of test cases (i.e., which test cases should be mutated first and how long a test case should be mutated), mutations of test cases (i.e., how a test case should be mutated), and criteria to appraise test cases (e.g., code coverage, code coverage plus calling context, and path coverage). Past research has invested tremendous efforts in improving these factors, aiming to prioritize the scheduling of high-quality test cases, introduce smarter mutations, and retain test cases of high values. The efforts have led to a performance leap of greybox fuzzing [40, 46–49, 56, 67, 68, 71, 72, 80, 87**?**].

System factors affect the time needed to accomplish one fuzzing iteration, assuming that the scheduling, mutation, and feedback schemes are determined. In principle, the system factors are rooted in how the fuzzing tool is designed to accomplish steps ①-③ (e.g., how the fuzzing tool initiates the target program).

In this paper, we focus on optimizing the system factors of greybox fuzzing, motivated by that less research has been conducted in

Table 1: Benchmark programs used in our study.

Project	Version	Driver	Option	Seed
BINUTILS	2.38	OBJDUMP	-d @@	[12]
BINUTILS	2.38	READELF	-a @@	[12]
UNRTF	0.21.10	UNRTF	–latex @@	[37]
WOFF2	1.02	WOFF2_DECOMPRESS	@@	[38]
QUICKJS	2021-03-27	QUICKJS	@@	[32]
JPEG	9e	DJPEG	@@	[35]
LIBTIFF	4.4.0	TIFF2PS	@@	[35]
LIBXML2	2.9.2	XMLLINT	@@	[53]
TIDYHTML	5.9.20	TIDYHTML	-qicu @@	[53]
OPTIPNG	0.7.7	OPTIPNG	@@ -out /dev/null	[29]
LIBPCAP	5.0.0	TCPDUMP	-vvvvXX -ee -nn -r @@	[30]
MUPDF	1.20.2	MUTOOL	draw @@	[52]

this direction. Specifically, we aim to inspect and rectify the fuzzing tool's designs that slow down a fuzzing iteration, thus bringing a higher iteration frequency and a better fuzzing performance.

2.2 Motivating Study

To understand the potential opportunities for system optimization, we conduct a study to dissect the time consumed by greybox fuzzing tools in completing steps ①-③. For better generality, we consider AFL [86] (version 2.57b) since its system-level designs have been followed by many other tools [41, 42, 45, 49, 56, 63, 67, 85]

Experimental Setup: To support the study, we gather a set of 12 projects included in the OSS-Fuzz program [79] (summarized in Table 1). These projects have also been used for evaluations in top-venue papers (CCS/USENIX/NDSS/S&P/ACSAC/FSE/ASE/ICSE/ISSTA/PLDI). Instead of running manually-crafted fuzzing drivers shipped with OSS-Fuzz, we pick a popular application from each project as the target, as those applications cover complete execution and represent more general scenarios. Dictionaries from AFL are applied wherever available. As the experiments focus on performance, we keep the deterministic mutations to reduce randomness. Further, we use the test cases shipped with AFL as seeds and configure AFL to run the default **fork server mode**. Finally, we configure AFL to run with **both `ext4` file system mounted on an SSD and `tmpfs` memory file system mounted on a 64MB RAM disk** [61] to understand the impact of IO. All experiments are conducted on CloudLab [73] with machines of the same configurations: 16-core AMD 7302P@3.00GHz, 128GB ECC memory, Ubuntu 16.04 TLS.

We customize the code of AFL to gather information about the time they spend at different steps illustrated in Figure 1:

Step ①: In the `run_target` function, AFL initiates the target program and waits for it to finish a given test case. The period from finishing one test case to initiating the target program for the next test case is considered Step ①, where the time cost is measured.

Step ②: AFL invokes `fork` in `__afl_start_forkserver` to start a new process for executing the target program. We count the time between calling and return of `fork` as the time of Step ②.

Step ③: After forking in `__afl_start_forkserver`, AFL waits for the target program to finish the test case. We consider that period as ③ and count the time.

Results: We dry-run AFL with the 12 projects on test cases from 24 hours of fuzzing. The best and worst results are discarded to reduce outliers. Figure 2 visualizes the distribution of average time spent by the fuzzing tools at different steps (check the **FS_SSD** and **FS_TMPFS** bars). Of all the time, 84% is spent on **Step** ③ (executing the target program). In contrast, **Step** ① and **Step** ② account for 12% and 4% of the time. One thing worth discussing is that using `tmpfs` indeed accelerates fuzzing compared to using SSD. On average, `tmpfs` reduces 2% of the time needed for a fuzzing iteration. **In all follow-up experiments, `tmpfs` is used by default**.

2.3 Optimization Opportunities

Initiating Target Program: AFL adopts the fork server mode, where it spawns a child process to run the target program on each test case. The forking process takes time due to operations like page table duplication. As shown in Figure 2, the forking operation alone consumes about 4% of the total time. More critically, running a new process to execute each test case incurs other costs (e.g., process cleanup at exit and page faults due to copy-on-write). As we will unveil later in this subsection, the target execution can speed up by 62% after eliminating those costs. This inspires the need for **Optimization I**: *replacing the fork server mode.*

Snapshot mode: Targeting **Optimization I**, Xu et al. proposed to replace the fork server mode with the ***snapshot mode*** [75, 84]. The idea is to designate the same process to run the target program on all test cases. Technically, they customize the Linux kernel to snapshot the process' memory before running a test case. During fuzzing, all writable pages are configured read-only and get copied upon modification (i.e., copy-on-write). Once finishing a test case, the kernel restores the modified memory (and file status). This way, forking is avoided while every test case still runs in a clean state.

To understand the effectiveness of the snapshot mode, we extended our study to cover the snapshot mode developed for AFL. The public implementation [28] only works with AFL-2.4b and GCC. We ported the implementation to work with AFL-2.57b and LLVM such that the settings are aligned with the other studies. More details of our implementation are presented in Figure 10.

Results: On all the programs except for UNRTF where the snapshot mode malfunctioned, the snapshot mode indeed reduces the time of a fuzzing iteration (see Figure 2). On average, the reduction rate is 28%. This efficiency improvement mainly comes from two sources. First, the snapshot mode avoids fork and eliminates the associated time cost, as clearly illustrated in Figure 2. Second, as a side benefit of removing fork, the snapshot mode reduces the number of page faults. We utilize `perf` to count the page faults triggered by the experiments in our study. Table 11 in the Appendix summarizes the average number of page faults triggered by different fuzzing modes on each test case. Compared to fork server mode, the snapshot mode helps reduce page faults on every program, resulting in an 55% reduction rate on average.

Persistent mode: Going beyond snapshot mode, both AFL and AFL++ introduce the ***persistent mode*** [8, 27] where the target program is executed continuously on different test cases. Figure 10 in the Appendix illustrates how this is done with OBJDUMP following the official guidance of AFL [27]. In this mode, neither snapshot nor restore of memory is performed. In addition, no extra operations (e.g., the copy-on-write used by the snapshot mode to track modified memory) are needed at fuzzing time. Thus, the persistent mode runs even faster than the snapshot mode. Another crucial advantage of persistent mode is that it requires no OS kernel modification, presenting better user-friendliness and higher robustness.

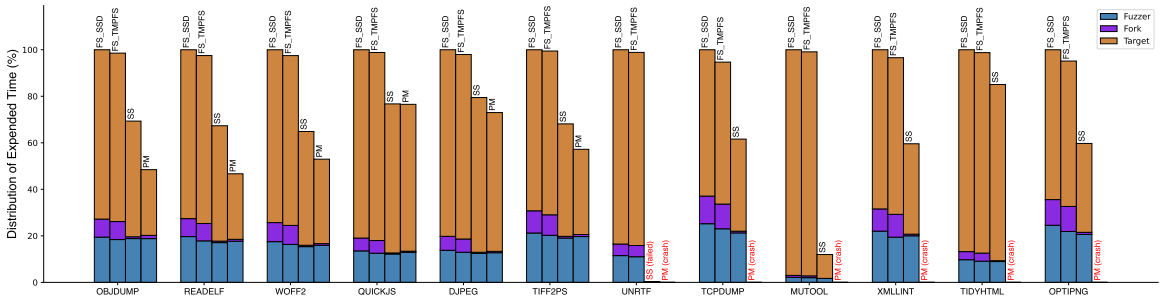


Figure 2: Distribution of time spent by AFL at different steps in a fuzzing iteration. `FS_SSD` refers to the fork server mode running with an SSD; `FS_TMPFS` stands for the fork server mode running with a tmpfs; `SS` means the snapshot mode; and `PM` is the persistent mode. UNRTF, TCPDUMP, MUTOOL, XMLLINT, TIDYHTML, and OPTIPNG fail to run under `PM`, and UNRTF incurs failures of `SS`. In the legend, `Fuzzer`, `Fork`, and `Target` represent steps ①, ②, and ③, respectively. All numbers are calculated using the total time of `FS_SSD` as the baseline. For instance, `Fuzzer` under `SS` is calculated as $\frac{\text{[time spent on step ① by SS]}}{\text{[total time spent by FS_SSD]}}$.

Table 2: Average number of pages recovered by the snapshot mode for each test case. UNRTF should be ignored as the tool was not running correctly on it.

OBJDUMP	READELF	WOFF2	QUICKJS	DJPEG	TIFF2PS	UNRTF	TCPDUMP	MUTOOL	XMLLINT	TIDYHTML	OPTIPNG
36	25	90	37	30	31	1,897	402	996	48	52	80

We further tested the persistent mode on the 12 benchmark programs. To avoid resource exhaustion, we re-fork a new process after looping the program `1,000` times. As shown in Figure 2 (the **PM** bars), the persistent mode — in cases where it can work (OBJDUMP, READELF, WOFF2, QUICKJS, DJPEG, TIFF2PS) — indeed runs faster than the snapshot mode. On average, persistent mode requires 19.6% less time to complete a fuzzing iteration. The reason is that snapshot mode needs to snapshot, track, and reset the memory while persistent mode does not. As summarized in Table 2, the snapshot needs to reset at least dozens of memory pages on each test case when applied to our benchmark programs. Given larger programs like MUTOOL and TCPDUMP, the number can increase to several hundred.

Despite its benefits, persistent mode carries an intrinsic restriction as stated in [26]: "*persistent mode requires that the target's state can be completely reset so that multiple calls can be performed without resource leaks, and that earlier runs will have no impact on future runs*". This restriction often limits the persistent mode's applicability. In our study, it fails to run 6 programs (UNRTF, TCPDUMP, MUTOOL, XMLLINT, TIDYHTML, and OPTIPNG). The reason is the effects left over by previous test cases are not cleaned, crashing the execution of follow-up test cases. For a better understanding, we present a detailed analysis of MUTOOL in Figure 3.

```
1 int optind = 0; // a global index
2 int main(int argc, char **argv){ ...
3   if (!scan || *scan == '\0') {
4     if (optind == 0) optind++;
5         scan = argv[optind]+1;
6         optind++; //the increase here accumulates infinitely
7   }
8 }
```

Figure 3: An example where persistent mode fails to run (MUTOOL). In the persistent mode, `optind` keeps increasing given different test cases. If `optind` grows too big, it incurs out-of-bound access at line 5 and crashes the execution.

To mitigate the above issue, the common strategy is to understand the target program and identify the critical states that must be recovered, followed by manually patching the target to perform recovery. In the example shown in Figure 3, we will need to insert code resetting `optind` to 0 before each fuzzing iteration. While this strategy can work, it incurs a heavy burden on the user and faces the risk of human errors.

> **Opportunity I:** Enhancing persistent mode with an automated, efficient method to reset the affected execution states will escalate **Optimization I**.

Table 3: Percentage of time spent in the kernel space during target program execution in the persistent mode.

OBJDUMP	READELF	WOFF2	QUICKJS	DJPEG	TIFF2PS	UNRTF	TCPDUMP	MUTOOL	XMLLINT	TIDYHTML	OPTIPNG
13.9	26.9	15.1	4.1	15.4	24.9	✗	✗	✗	✗	✗	✗

Target Program Execution: Target program execution consumes the most time of a fuzzing iteration. Replacing the fork server mode with persistent mode brings optimization. Another system-level but underexplored aspect is the interactions between the target program and the OS. Consider the 12 programs covered in our study. As summarized in Table 8 in Appendix, all the programs interact with the OS via system calls. They invoke tens or hundreds of system calls when processing a single test case (TIDYHTML even uses 2,000+ system calls). Among the system calls, file operations comprise the majority portion (85%+ in most cases). We further extend our study to measure the time costs of the system calls. We focus on the persistent mode as it represents the most optimized option. Table 3 shows the percentage of time spent in the kernel space during target program execution (**Step** ③), which should well approximate the time costs of system calls considering that persistent mode avoids fork, process cleanup, and most page faults (see Table 11). For all the programs except for QUICKJS, system calls account for nearly or over 15% of the total execution time. The percentage increases to 24%+ on READELF and TIFF2PS. This illustrates the opportunity of **Optimization II**: *by simplifying the OS interactions, we can further optimize the target program execution.*

Little effort has been taken specifically toward **Optimization II**. However, a common practice we adopt today can help indirectly. When fuzzing a project, people often opt to create drivers that directly invoke the target functions with a buffered input. Thus, the test case can be passed via memory, following the scheme of LIBFUZZER [22]. This way, interactions with the file system to load the test case are avoided. Consider LIBPCAP as an example. Instead of using TCPDUMP as the fuzzing driver, we can also use manually-created drivers where no file operations are needed (e.g., [24] can be used to fuzz the `pcap_setfilter` function via buffer input).

Despite the applicability, manually creating fuzzing drivers to avoid OS interactions is not a desired solution for **Optimization II**. ① Many functions have integrated OS interactions, which are

impossible to bypass during fuzzing. In the aforementioned LIBPCAP, many functions only take file input. Even manually created drivers for those functions must include interactions with the file system for test cases (e.g., [23]). In whole-application fuzzing, OS interactions are often more prevalent and less feasible to avoid. ② Creating fuzzing drivers to reduce OS interactions requires insensitive domain knowledge and heavy manual efforts, which is not always feasible and affordable.

> **Opportunity II:** Reducing OS interactions – **Optimization II**– is another promising direction to speed up target program execution. A solution without requiring manually-created fuzzing drivers remains missing.

3 PROFILE-DRIVEN SYSTEM OPTIMIZATIONS

3.1 Approach Overview

In this paper, we explore a new approach to realizing **Optimization I** and **Optimization II**. Our key insight is that by running the fuzzing tool for a short period (a ***profiling phase***), we can gain the information needed to achieve both optimizations.

Optimization I: Enabling reliable persistent mode faces the barrier of state recovery. In principle, the aforementioned snapshot mode can support state recovery. However, as unveiled in §2.3, snapshot mode can still offset the efficiency of persistent mode due to operations like fuzzing-time memory tracking and recovery.

To reliably run persistent mode, three types of critical states need to be recovered. **State-①:** Global data can persist across fuzzing iterations. Their values set by one test case can affect the following test cases, leading to incorrect computation or even crashes. The example described in Figure 3 shows how global data prevents MUTOOL from running the persistent mode. **State-②:** Memory allocated in one iteration should be recycled before entering the next fuzzing iteration. Otherwise, memory leakage can happen. **State-③:** System-level status is also carried between test cases. In particular, files opened by a test case may not be closed, leaving over non-recycled kernel data structures and file descriptors.

Non-recycled memory (**State-②**) is considered less critical as it typically causes resource consumption instead of interrupted execution. In practice, people tend to ignore non-recycled memory even in manual fuzzing drivers. We follow AFL [27] and AFL++ [8] to handle it. Once the persistent mode completes a threshold number of test cases, we re-fork the target program such that memory can be recycled. Files (**State-③**) can be efficiently recovered together with our **Optimization II**. Related details will be covered shortly.

To handle global data (**State-①**), an intuitive idea is to identify all global objects compiled into the fuzzing target and reset all of them before each fuzzing iteration. However, this idea is not optimal. As shown in Table 12 (column "`All Global Objects`") in Appendix, a program can often use hundreds of global objects that occupy KBs or even MBs of memory. Snapshotting and resetting all of them for every test case can take significant time. An alternative method is to track global objects modified during a fuzzing iteration and reset them before the next iteration. Doing so requires recovering fewer objects but faces the challenge of tracing modifications of global objects. We can adopt the snapshot mode [84] to mark the global data segment as non-writable and trace modification at the page level. This, however, would essentially degrade to the snapshot mode, not to mention the high engineering complexities.

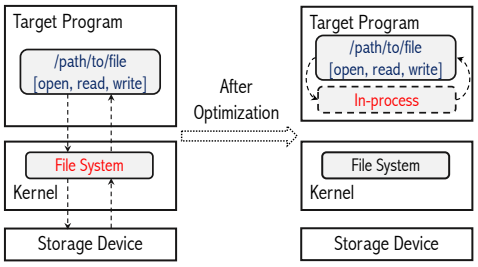


Figure 4: Optimizing file system interactions.

In this paper, we explore an approach for more aggressive optimization. Running dynamic analysis on the test cases from the profiling phase, we learn what global variables are manipulated. During fuzzing, we patch the target program to reset those global variables before running a test case. This way, we require neither recovering all global variables nor tracing the modifications at runtime, offering better optimizations. An evident limitation of our approach is that we will miss global variables newly covered in post-optimization fuzzing. We address this issue by repeating the profiling-and-patching process once observing reduced **stability** during fuzzing. Stability is a metric used by AFL and AFL++ to measure the percentage of bitmap bytes that behave consistently. Low stability indicates different executions of the same test case present disparate behaviors and signifies abnormal fuzzing [7]. As we will show in §5.2, our profiling and patching are highly efficient and infrequently happen, imposing insignificant impacts on fuzzing.

Optimization II: Given specific fuzzing settings, the target program often presents predictable OS interactions. Running test cases from the profiling phase can give us a good understanding of the OS interactions. This brings the key insight of our optimization: *we can replace the observed OS interactions with in-process, behavior-preserving but simpler operations.* Figure 4 reflects the idea on file system interactions. We create a virtual file system (VFS), as a library, to handle files identified during the profiling phase. In fuzzing, the VFS is linked to the target program and handles all operations on the included files. For instance, OBJDUMP, when fuzzed in our study, reads data from `/usr/share/locale/locale.alias` on the disk. Our VFS can include a copy of the file and intercept all accesses to the file with user-space operations.

```
uint vfs_read(int fd, void *buf, uint cnt){
  vfs *file = VFS_GET_FILE(fd);//managed in memory
  if (file == NULL) native_read();
  cnt = CHK_COUNT(cnt, file);
  if (cnt > 0){
    memcpy(buf, file->data+file->off, cnt);//managed in memory
    file->offset += cnt;
    return cnt;
  }
}
```

Figure 5: Optimized `read` operations.

To better illustrate the effectiveness of our optimization, we showcase the `read` operation of our VFS in Figure 5. Roughly, its time cost approximates a `memcpy`. More importantly, this example illustrates four dimensions where our VFS brings optimization: (i) it manages the data and metadata of files in memory, avoiding interactions with the IO devices; (ii) it runs everything in the user space,

avoiding interrupts and context switches to the kernel; (iii) it only keeps functionality mandated to maintain the needed behaviors, removing all fuzzing-irrelevant operations of modern file systems (e.g., journaling, caching, and crash persistence); (iv) an independent VFS can run to support each fuzzing instance, eliminating contention in the file system and improving the scalability.

Besides offering optimizations, our VFS also facilitates **Optimization I**. When the target program finishes a test case, we recover the file system status by resetting the VFS (e.g., closing opened files). The resetting is often equivalent to editing several memory bytes

3.2 Profile-driven State Recovery

Given a target program **P** and a fuzzing tool **F** (e.g., AFL), we perform the profiling phase by running **F** on **P** under the desired configurations and fork server mode for time **T**. Then we proceed to enable persistent mode with affected global data reset.

Collecting Target Objects: This step replays the test cases from the profiling phase and gathers global objects that have been manipulated. As discussed above, we can remove the write permission of the global data segment. A segmentation fault happens when a test case writes to a global object, allowing us to catch the object. However, the method can be very slow, particularly when there are many test cases and many affected global objects. Further considering that we may need to rerun the profiling process periodically, this permission-based method is less suitable.

In this paper, we propose an over-approximating but highly efficient approach. Our idea is to instrument the target program to add a check on every storing or address-taking operation of global objects. Once any test case hits a check during profiling, the check will log the corresponding global object by its name and type, and we will deem the object modified.

```
1 int glob1;
2 int glob2;
3 int glob_buf[10];
4 int *glob_ptrs[] = {&glob1, &glob2};
5 void func(int arg1, arg2){
6   glob1 = 1; //check(glob1,int)
7   int *ptr1 = &glob2; //check(glob2,int)
8   int *ptr2 = &glob_buf[arg1]; //check(glob_buf,int[10])
9   *ptr1 = 0;
10  *(glob_ptrs[arg2]) = 0; //check(glob1,int); check(glob2,int)
11 }
```

Figure 6: Demonstration of checks to log global object accesses.

To better illustrate how our approach works, we show an example in Figure 6. The checks on storing operations (e.g., `line 6`) catch direct writing, while the checks on address-taking operations (e.g., `line 7`) capture indirect writing via pointer dereferences (e.g., `line 9`). Besides, another form of indirect writing can exist. As shown in `line 4`, a global array/struct can be initialized with pointers to other global objects. By retrieving and dereferencing pointers from that array/struct, the nested global objects can be manipulated (e.g., `line 10`). To identify these cases, we enumerate the initialization of each global array and struct, and if the array/struct is accessed, we collect the nested global objects referred to by the fields (e.g., line 4).

We also gather two special types of "global objects": (i) arguments to the `main` function and (ii) environment variables, considering that their states also propagate between test cases. For `main`'s arguments, we consider them always modified. For environment variables, we trace and collect those fetched by standard functions (e.g., `setenv` and `unsetenv` on Linux). As we will show in Table 9, our approach is highly efficient, typically only taking several seconds to profile all the test cases.

Patching Target Program: After gathering the global objects, we de-duplicate the results and instrument the target program to reset them during fuzzing. The instrumented code allocates a shadow copy for each object and saves/restores the object to/from its shadow copy before/after running a test case. For conservatives, given a global object with a composite type (e.g., array or struct), we save and restore the data occupying the entire type (e.g., the entire array or struct) even if only some elements or fields are identified.

Handling Dependent Libraries: Programs often depend on external libraries. We support external libraries. The easiest way is to link the libraries to the target program statically, and our approach can be applied seamlessly. Alternatively, we can instrument the libraries to support profiling and patching. The resulting libraries can be dynamically linked and covered by our approach.

3.3 Profile-driven OS Abstraction

Modern OSes like Linux support hundreds of system calls [25]. Designing optimization to cover all of them is impractical. According to our study in §2.3, interactions with the file system are most frequent. Other system calls, occurring less frequently, are primarily associated with process management (e.g., `rt_sigaction`), and memory management (e.g., `mmap`). These system calls are less feasible to be simulated (e.g., signals are hardware-dependent). Considering these factors, we focus on file system interactions.

File System: To optimize file system interactions, we build a VFS, as illustrated in § 3.1, to support the target program during fuzzing. We start with understanding how the test cases from the profiling phase interact with the file system. Specifically, we observe *what files are accessed* and *how they are accessed* by the test cases. Depending on the properties of an observed file, we handle it differently:

- *Existing files* are files existing in the native file system and accessed for reading/writing during profiling. For such a file, we allocate a virtual counterpart in the VFS that fully resides in memory and keeps a copy of the file's data and metadata. To ensure correctness, the virtual file shares the same path as the native file.
- *Non-existing files* do not exist in the native file system but are opened during profiling. We create an empty node for such a file in the VFS, indicating the file at the path is nonexistent. At fuzzing time, our VFS makes a quick return upon access to this file.
- *New files* are files newly created during the profiling process. We omit these files as file creation is supported by VFS.
- *The test case* is a special file in the VFS, which we place at the path specified by the fuzzer. We allocate shared memory to save the file so the fuzzer can access it directly for updates.
- *Standard input, output, and error* are also handled by the VFS. In principle, standard output and standard error do not affect fuzzing. Thus, the VFS simply ignores them and fakes the return results (e.g., how many bytes are written). Further, the VFS maintains a buffer to simulate the standard input. The buffer is supported by shared memory, which can be accessed by other processes like the fuzzing tool. On request for data from standard input, contents from the buffer will be returned without involving the IO.

To enable efficient access to the VFS, all files are organized in a binary-search-tree based on their paths. More advanced structures, such as red-black-tree, are not used as the number of files is typically small (1 or 2). For each virtual file, we use a memory buffer of a fixed size (256K for the data plus 432 bytes for the metadata), unless our profiling indicates that a larger space is needed. It is possible that some files are not observed during profiling fuzzing but get accessed during fuzzing. The VFS accommodates those files by redirecting their operations to the native file system. The creation of new files will be handled by inserting new items into the VFS with the desired metadata. However, to avoid excessive memory use, we allow at most 96 files (16MB of memory in total). New files beyond the quota will be sent to the native file system.

When opening or creating a file in the VFS, we create an unused file descriptor starting at 4096. This is to reduce the chance of conflicts with file descriptors assigned by the native file system. Since the native file system assigns file descriptors starting from a small number (typically 3), it will unlikely reach a big one like 4096 during fuzzing. In rare cases where the native file system creates a file descriptor beyond 4095, we abort the execution and start a new process where the VFS will use larger file descriptors (e.g., 9012).

To connect our VFS to the target program, we create interfaces to operate on the VFS. The interfaces resemble the GLIBC wrappers of POSIX system calls on files [31] (`read`, `open`, `write`, `stat`, `lseek`, `close`, etc.). We do not implement higher-level interfaces like `fopen`, considering that the low-level interfaces are more unified and easier to be captured comprehensively.

Socket: We also extend our VFS to support socket operations. When a socket is created, we add a virtual file to work as the socket. The virtual file, also maintained in memory, consists of a stream buffer and management metadata (e.g., status of the socket). When two sockets are connected via interfaces like `connect` and `socketpair`, the VFS records the two related virtual files as paired. This way, the VFS can correctly send the data from one socket to the other.

Our VFS only intends to support non-blocking operations (precisely, operations that immediately return, no matter whether the data is ready or not). Blocking operations are not covered because their user-space simulation (e.g., using a continuous loop) does not bring optimizations. To avoid chaos due to the involvement of blocking operations, we only mount the VFS for sockets if our profiling shows that all socket-based operations are non-blocking. In addition, the appearance of blocking operations during fuzzing will abort the execution and restart the target program without VFS for sockets. While the socket extension is an integrated part of our research, we do not claim novelty credit for it. Similar ideas have been used in several existing projects [1, 2], although they have intentions other than optimizations.

4 IMPLEMENTATION

We have implemented a prototype to enable our optimizations on AFL (version 2.57b) and AFL++ (version 4.01c).

Profiling: To gather global objects, we create an LLVM pass with 900 lines of C++ code to instrument the checks into the target program. The pass supports LLVM 10, 11, and 12. To handle applications and libraries that cannot be compiled with Clang (in particular GLIBC), we developed a GCC (version 8.4.0) plugin with 500 lines of C++ code to add the checks. Both our LLVM pass and GCC plugin run after the built-in optimizations to ensure the compiler does not throw away our operations. To profile the OS interactions, we trace system calls with `strace` and then analyze the results as needed.

Table 4: Extra benchmark programs used in our evaluation.

Project	Version	Driver	Option	Seed
LIBTIFF	4.4.0	TIFF2PDF	@@	[17]
LIBXSLT	1.1.37	XSLTPROC	@@ /out/sample.xml	[18]
OPENSSL	3.1.0-beta	X509-TEST	@@	[18]
BASH	5.2.0	BASH	-n @@	[9]
EXIF	0.6.22.1	EXIF	@@	[13]
MJS	1.26	MJS	-f @@	[21]
FLVMETA	1.22	FLVMETA	@@	[14]
JQ	1.6-159	JQ	. @@	[20]

Persistent Mode: We develop another LLVM pass for the saving and restoring of global objects. The pass injects a constructor function into the `main` module for copying the target global objects. For efficiency, all the objects are sequentially saved in a buffer added by us. Our pass also adds code into the `main` function to reset the global objects after each test case. A challenge is many global objects are declared as `static`, which is invisible to the `main` module/function. To handle a static object, we leverage the above LLVM pass and GCC plugin to add a global pointer to the object. Thus, we can indirectly reference the object via the pointer anywhere.

The target program may call `exit,_exit,_Exit` to terminate the program. Thus, the execution will escape the persistent mode loop (recall Figure 10). We intercept those functions using linker option `"-Wl,-wrap,_exit -Wl,-wrap,exit -Wl,-wrap,_Exit"` and switch the execution to the persistent mode loop via `longjmp`. The target program may register callbacks at the exit using interfaces like `atexit` and `on_exit`. We intercept `atexit` and `on_exit` to gather the callbacks and invoke them on `exit,_exit,_Exit`.

VFS: We implement the VFS with around 1,400 lines of C code. The implementation includes interfaces corresponding to GLIBC wrappers of POSIX system calls for files and sockets [31]. To enable efficient test case sharing, we modify AFL and AFL++ to write the prepared test cases to VFS directly, using the interfaces we provide.

Mounting VFS: To mount our VFS to the target program in fuzzing, we rely on run-time hooking. We compile the VFS as a dynamic library and load it to the target process. Before the target process enters the persistent mode loop, we hijack the functions offered by standard libraries (GLIBC and LIBPTHREAD) for file and socket operations. When a hijacked function is called, we transfer the execution to the VFS. This way, all accesses to the native file system and sockets are redirected to our VFS. The hijacking is done by rewriting the first several bytes of a target function with a call to our VFS interface, using the FUNCHOOK library [15].

5 EVALUATION

Our evaluation centers around the following questions:

- **Q1:** *Can profile-driven state recovery achieve **Optimization I**?*
- **Q2:** *Can profile-driven OS abstraction achieve **Optimization II**?*
- **Q3:** *Can our profile-driven optimizations benefit fuzzing tools?*

5.1 Experimental Setup

To support our evaluation, we gather 20 benchmark programs. 12 of them, listed in Table 1, are borrowed from our motivating study. The remaining, listed in Table 4, are further identified from

Table 5: Statistic results of our evaluation. ✗ indicates the fuzzing tool failed to run in that mode. `Fuzzing Speed` is measured by the number of test cases processed in 24 hours and normalized using `FS` as the baseline. For example, the result for `PM` is calculated by dividing the fuzzing speed of `PM` by that of `FS`. Stability lower than 90% is highlighted in purple. Programs in a box are covered in our study. We did not identify global variables for MJS.

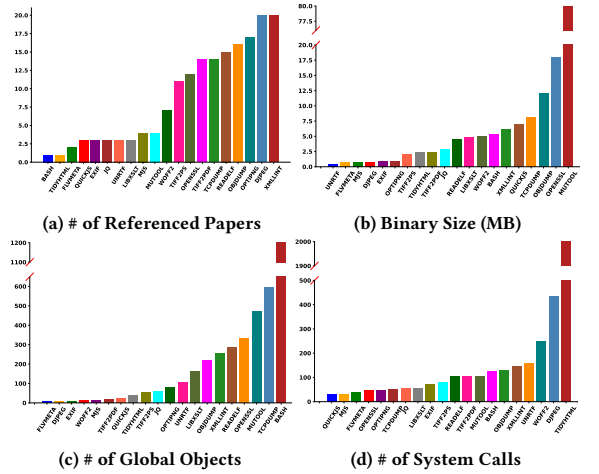
Project	AFL											AFL++										
	Stability (%)						Fuzzing Speed (baseline:FS)					Stability (%)						Fuzzing Speed (baseline:FS)				
	FS	SS	PM	PM_VOS	PM_REC	PM_REC_VOS	SS	PM	PM_VOS	PM_REC	PM_REC_VOS	FS	SS	PM	PM_VOS	PM_REC	PM_REC_VOS	SS	PM	PM_VOS	PM_REC	PM_REC_VOS
OBJDUMP	100	98.7	98.7	98.7	99.9	99.9	1.86x	2.15x	2.51x	2.02x	2.55x	100	100	98.8	98.9	99.7	99.8	6.14x	4.79x	5.78x	4.63x	5.66x
READELF	100	100	95.8	95.6	99.8	99.7	2.92x	4.94x	5.42x	4.01x	4.56x	100	100	95.8	95.6	99.8	99.7	2.80x	5.24x	7.07x	5.60x	7.02x
WOFF2	100	100	100	100	100	100	1.72x	1.99x	2.19x	2.06x	2.19x	100	100	100	100	100	100	1.24x	18.6x	21.5x	16.2x	17.5x
QUICKJS	100	100	99.4	99.1	99.4	99.4	1.1x	1.32x	1.45x	1.25x	2.02x	100	100	97.2	97.2	97.6	97.5	1.73x	1.96x	2.02x	1.86x	1.93x
DJPEG	100	100	100	100	100	100	1.79x	2.73x	3.37x	2.09x	2.85x	100	100	99.9	99.9	99.9	99.9	8.45x	2.59x	6.09x	2.46x	6.09x
TIFF2PS	100	99.3	99.4	100	99.3	100	0.89x	1.47x	1.81x	1.25x	1.63x	100	100	99.0	99.1	99.9	99.9	3.42x	4.98x	7.06x	5.34x	6.39x
TIFF2PDF	100	88.2	79.5	78.3	98.9	99.3	1.99x	2.12x	2.19x	2.16x	2.48x	100	94.0	76.0	73.0	99.9	99.9	2.87x	3.96x	6.16x	4.22x	6.34x
EXIF	100	100	100	100	100	100	0.17x	2.59x	3.02x	2.91x	3.12x	100	100	99.9	99.9	99.9	99.9	3.57x	2.93x	3.33x	2.75x	3.29x
MJS	100	100	99.7	99.7	99.7	99.7	1.08x	1.23x	1.52x	1.43x	1.74x	100	100	99.5	99.5	—	—	2.97x	4.49x	5.29x	—	—
FLVMETA	100	100	100	100	100	100	2.31x	2.32x	2.85x	2.69x	2.97x	100	100	99.6	99.6	99.6	99.6	2.52x	3.99x	5.29x	3.33x	6.58x
OPENSSL	100	100	76.9	77.5	99.8	99.8	3.1x	4.93x	5.80x	3.73x	3.97x	100	100	78.2	78.3	99.8	99.8	1.82x	8.74x	9.63x	1.82x	2.03x
BASH	100	100	20.2	20.8	95.7	95.7	0.38x	4.28x	4.38x	1.45x	1.53x	100	100	21.5	23.3	95.7	95.8	0.29x	4.95x	6.24x	3.47x	3.99x
JQ	100	100	100	100	100	100	1.02x	1.21x	1.26x	1.25x	1.36x	100	100	99.1	99.1	99.1	99.1	1.17x	1.35x	1.39x	1.31x	1.37x
UNRTF	100	✗	✗	✗	99.1	99.1	✗	✗	✗	1.24x	1.29x	100	93.1	✗	✗	97.0	97.0	0.37x	✗	✗	2.11x	2.18x
TCPDUMP	100	99.9	✗	✗	96.1	95.1	1.63x	✗	✗	1.72x	2.23x	100	100	✗	✗	98.6	98.8	4.80x	✗	✗	5.30x	6.25x
MUTOOL	99.9	99.9	✗	✗	99.9	99.8	1.71x	✗	✗	2.33x	2.66x	99.9	99.9	✗	✗	99.9	99.9	1.26x	✗	✗	2.98x	3.07x
XMLLINT	100	100	✗	✗	99.9	99.9	1.43x	✗	✗	1.70x	2.02x	100	100	✗	✗	99.8	99.8	2.27x	✗	✗	3.13x	3.96x
TIDYHTML	100	100	✗	✗	99.9	99.9	1.06x	✗	✗	1.24x	1.51x	100	100	✗	✗	99.9	99.9	1.54x	✗	✗	1.76x	2.33x
OPTIPNG	100	100	✗	✗	100	100	1.66x	✗	✗	1.55x	2.73x	100	100	✗	✗	99.9	99.9	3.20x	✗	✗	1.45x	2.02x
LIBXSLT	100	100	✗	✗	99.9	99.9	2.65x	✗	✗	3.06x	3.08x	100	100	✗	✗	99.8	99.8	2.92x	✗	✗	4.66x	5.90x

OSS-Fuzz [79]. We measure the diversity of the programs from four dimensions: ① *number of fuzzing papers using the programs*, ② *binary size*, ③ *# of global objects identified by profiling after 24-hour fuzzing*, and ④ *# of average system calls incurred by a test case*. The distribution on the four dimensions is displayed in Figure 7.

To ensure that our state recovery covers the libraries, we statically compile the dependent libraries into the target program. For GLIBC, we only handle the global objects that the target program explicitly imports, considering that GLIBC is more self-contained and many of its states only affect resource consumption (e.g., heap allocation). We include both AFL (version 2.57b) and AFL++ (version 4.01c), following the experimental setup in §2.2. All the experiments are done with the `tmpfs` file system. We run AFL and AFL++ on the 20 projects for 24 hours and repeat each test 10 times. The best and worst results are discarded to mitigate outliers. For comparison, we configure both AFL and AFL++ to run in the following modes:

- `FS` stands for the fork server mode. This is the default mode of both fuzzers, where no extra configurations are needed.
- `SS` represents the snapshot mode we discussed in §2.2. The snapshot mode shipped with AFL++ [3] has unrepaired issues, crashing most benchmark programs. We use a more stable version available at https://github.com/galli-leo/AFL-Snapshot-LKM.
- `PM` refers to the persistent mode described in [26, 27]. We instrument each benchmark to add a loop iterating the `main` function (see Figure 10). We force the loop to exit after every 1,000 iterations by default. On OPENSSL, we reduce that to 100 as resource exhaustion frequently happens when doing 1,000 iterations.
- `PM_REC` means we run the persistent mode with state recovery.
- `PM_VOS` means we run the persistent mode with OS abstraction.
- `PM_REC_VOS`, combing `PM_REC` and `PM_VOS`, runs the persistent mode with both profile-driven state recovery and OS abstraction.

Our state recovery and OS abstraction require a profile. We run the `FS` mode for 5 minutes and analyze the outcomes to build the profile. While a longer profiling phase may lead to better accuracy, 5 minutes already produce satisfying results. To ensure fairness, we stop the experiments $[300 + X]$ seconds earlier for `PM_REC`, `PM_VOS`, and `PM_REC_VOS` (X is the number of seconds needed for profiling and patching the binary). During fuzzing, we redo the profiling-and-patching if the fuzzer's stability drops under 95%.



(a) # of Referenced Papers (b) Binary Size (MB)

(c) # of Global Objects (d) # of System Calls

Figure 7: Distributions of benchmark programs in different dimensions

5.2 Profile-driven State Recovery

Our state recovery aims to promote `PM` mode as a replacement of the `FS` mode, thus achieving **Optimization I**. As pointed out in §2.2, the `PM` mode can fail to run a fuzzing target due to exceptions. In this evaluation, we measure whether our state recovery can re-enable the `PM` mode in those cases and, if so, assess the benefits.

Enabling PM: As shown in Table 5, the native `PM` mode fails to run 7 out of 20 fuzzing targets (UNRTF, TCPDUMP, MUTOOL, LIBXML2, TIDYHTML, OPTIPNG, and LIBXSLT). All the failures are caused by non-recovered global objects that corrupt follow-up fuzzing iterations. The reason for MUPDF has been illustrated in Figure 3. UNRTF,

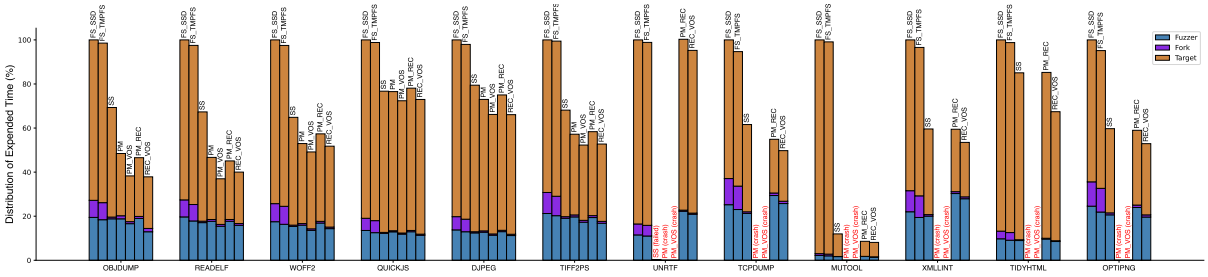


Figure 8: Distribution of time spent by AFL at different steps in a fuzzing iteration. In the legend, `Fuzzer`, `Fork`, and `Target` represent steps ①, ②, and ③ discussed in §2.1. All numbers are calculated using the total time of `FS_SSD` as the baseline. For instance, `Fuzzer` under `SS` is calculated as $\frac{\text{[time spent on step ① by SS]}}{\text{[total time spent by FS\_SSD]}}$.

TCPDUMP, TIDYHTML, and OPTIPNG share similar patterns. The failures of LIBXML2 and LIBXSLT have similar root causes, which we explain in Figure 11 in Appendix.

In all the cases above, our profiling identifies the responsible global objects and guides our state recovery to reset their values in each fuzzing iteration. As shown in Table 5 (the PM_REC column), the state recovery re-enables the PM mode in all the cases with stability close to 100% (96.1%- 100% for AFL; 97.0%- 99.9% for AFL++).

Moreover, our state recovery can help stabilize the PM mode. On TIFF2PDF, BASH, and OPENSSL, the PM mode has a lower than 80% stability (only 20% on BASH) due to unrecovered execution states. Our state recovery increases their stability to 95%+.

Performance Gain: The direct benefit of enabling PM mode is faster fuzzing speed. As shown in Table 5, on the 7 programs where the native PM mode fails, reenabling PM with state recovery (i.e., PM_REC) can increase the fuzzing speed of AFL/AFL++ by 83%/206% compared to their default fork server mode. On the other 13 programs, PM_REC can similarly accelerate the fork server mode.

To understand why PM_REC brings performance gains and what leads to a higher gain, we expand our study in §2.2 to include the PM_REC mode. Figure 8 shows that PM_REC fully eliminates fork and significantly reduces the time of target program execution. But why can PM_REC accelerate target program execution? The main reason is PM_REC decreases page faults. As shown in Table 11 in Appendix, PM_REC reduces 96% page faults incurred by the FS mode. The number of reduced page faults positively affects the time reduction for target program execution. PM_VOS reduces the page faults more significantly on READELF (90.8→0.2 per test case) and MUTOOL (343.7→0.2 per test case). Correspondingly, PM_VOS optimizes the two programs the most.

Another option to "enable" the PM mode on the 7 programs is the SS mode. In our evaluation, the SS mode of both AFL and AFL++ can run all 7 programs (except for AFL on UNRTF). As shown in Table 5, the SS mode also accelerates the PM mode, thanks to its elimination of fork (Figure 8) and reduction of page faults (Table 11). However, SS is less efficient than the PM_REC mode. On average, SS only increases the fuzzing speed of FS mode by 60.4% on AFL and 177% on AFL++. On the same set of programs, PM_REC increases the fuzzing speed by 110% and 283% for AFL and AFL++. The performance advantage of PM_REC over SS is attributed to two reasons. ① The states that PM_REC needs to reset are pre-determined based on the profiling. In contrast, SS needs to trace those states at run-time, introducing extra costs. As shown in Table 2, SS often needs to trace dozens or hundreds of pages. ② PM_REC performs object-level memory recovery, which incurs lower costs than PM_REC's page-level recovery. As we can see in Table 12 in Appendix, PM_REC only needs to reset several dozens/hundreds of bytes in most cases.

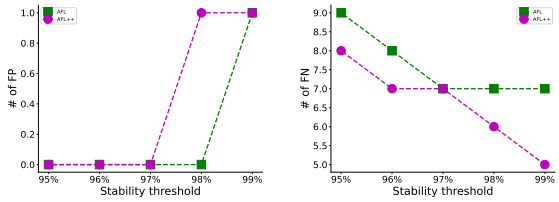
While PM_REC can run the target program faster than SS, its fuzzing speed can be lower. The reason is that, for every 1,000 fuzzing iterations, PM_REC downgrades to the FS mode once while SS does not have to. OPTIPNG with AFL is such a case. The execution of OPTIPNG under PM_REC needs a shorter time than SS (see Figure 8). However, the fuzzing speed of PM_REC is lower than SS (check Table 5). One may note that, when applied to AFL++, the fuzzing speed of PM_REC falls behind SS more often. This does not necessarily reflect the ineffectiveness of PM_REC. Instead, the SS mode of AFL++ still contains implementation flaws, often failing to run correctly after several hours. With AFL++, SS runs faster than PM_REC on OBJDUMP, DJPEG, EXIF, and OPTIPNG. However, when fuzzing OBJDUMP and EXIF, SS presents zero growth of code coverage after 4 hours, indicating that it runs into an abnormal state.

Profiling Statistics: In the PM_REC and PM_REC_VOS fuzzing modes, most programs do not need re-profiling because their stability stays 95%+. The only exception is OPTIPNG, which requires re-profiling once when running with AFL++. To understand why re-profiling is not frequent, we compare the number of global objects collected with profiling after 5-minute fuzzing and after 24-hour fuzzing. As shown in Table 12, the set of global objects in most cases does not change much in 24 hours. For 9 programs, the set remains identical.

We further measure the time needed to complete the profiling-and-patching process and summarize the results in Table 9. It shows that our profiling-and-patching process is efficient. The total time cost ranges from 4 seconds to 61 seconds, depending on the fuzzing targets. This indicates that our approach shall not interrupt the fuzzing progress much even if frequent re-profiling is needed.

Stability Threshold: We consider fuzzing stability as the metric to decide when to run re-profiling (with 95% as the default). Evidently, the threshold can affect the fuzzing progress, so we further measure its impact. At the end of each 24-hour test, we use different thresholds (95%-99%) to decide on re-profiling. We then count the false positives (FPs) and false negatives (FNs) defined as follows:
► False Positive: re-profiling is done because the stability drops below the threshold, but no new global objects are identified.
► False Negative: re-profiling is not done as the stability is above the threshold, but new global objects can be identified by re-profiling.

Figure 9 shows the change of FPs/FNs with the stability threshold. In summary, a lower threshold leads to fewer FPs (avoiding unnecessary re-profiling) but more FNs (missing the identification of new



(a) # of FPs (b) # of FNs

Figure 9: Number of FPs & FNs given different stability thresholds. The numbers mean program numbers (i.e., each program is counted at most once).

global objects). A higher threshold presents an opposite trend. This is expected as lower stability is often caused by that more global objects have emerged and hurt the fuzzing status. Our threshold choice, 95%, introduces zero FP but presents more FNs than higher thresholds. We stick to 95% for three reasons. First, stability staying over 95% is considered sufficient for greybox fuzzing [7]. Second, the FNs introduced by 95% do not lead to severe consequences like crashes. Third, a threshold over 95% (e.g., 96%) reduces FNs, but it incurs more re-profiling. For example, 95% triggers re-profiling for one program, while 96% triggers that for 3. Although the re-profiling identifies new global objects, it does not necessarily benefit fuzzing. On the 3 programs identified with a 96% threshold, the re-profiling only increases the stability for 1 program, which is also covered by a 95% threshold. To sum up, 95% represents a decent trade-off between effectiveness and efficiency.

5.3 Profile-driven OS Abstraction

Our OS abstraction is transparent to the fuzzing tool. It can be applied to various tools under different modes. In our evaluation, we run it with both AFL and AFL++ in the PM and PM_REC modes (resulting in the PM_VOS and PM_REC_VOS modes). *For simplicity, we use* VOS *to refer to our OS abstraction.*

Correctness Validation: A concern about VOS is that it may deviate from the real system and affect the fuzzing target. Our first evaluation measures the correctness of our VOS in supporting fuzzing tools. We replay each test case from our 24-hour experiments with and without VOS under the same mode. If the test case produces standard outputs or errors, we check whether they remain the same with and without VOS. Further, we dump the basic blocks executed by the test case and inspect whether VOS causes any divergence. We pass this validation test in all cases unless random timeout or resource exhaustion happens.

Performance Gain: The PM mode can run 13 benchmark programs, where VOS can be applied seamlessly. As shown in Table 5, VOS maintains the stability of PM. More importantly, it accelerates the PM mode on all 13 programs. On average, VOS increases PM's fuzzing speed by 14.1%/29.6% with AFL/AFL++ (check PM *v.s.* PM_VOS in Table 5). To unveil the reason for the performance gain, we extend our study in §2.2 to include the PM_VOS and PM_REC_VOS modes. The results in Table 8 in Appendix show that PM_VOS removes 87% of the system calls invoked by the PM mode. Consequently, PM_VOS reduces 42% of the time spent by PM in the kernel space during target program execution, as shown in Table 13. Further, the optimization of VOS is positively affected by how many system calls are removed.

Table 6: Code coverage of different fuzzing modes. The numbers are normalized using FS as the baseline. PRV refers to PM_REC_VOS (to reduce space).

Project	AFL					AFL++				
	Code Coverage (baseline:FS)					Code Coverage (baseline:FS)				
	SS	PM	PM_VOS	PM_REC	PRV	SS	PM	PM_VOS	PM_REC	PRV
OBJDUMP	1.01x	1.03x	1.03x	1.03x	1.07x	0.72x	1.06x	1.06x	1.06x	1.06x
READELF	1.10x	1.07x	1.10x	1.14x	1.15x	1.04x	1.05x	1.09x	1.12x	1.12x
WOFF2	0.88x	1.02x	1.02x	1.02x	1.02x	0.99x	1.07x	1.09x	1.10x	1.10x
QUICKJS	1.08x	1.08x	1.11x	1.09x	1.09x	1.08x	1.08x	1.09x	1.07x	1.09x
DJPEG	1.02x	1.03x	1.04x	1.04x	1.05x	0.89x	1.04x	1.07x	1.02x	1.09x
TIFF2PS	1.02x	1.02x	1.03x	1.02x	1.03x	1.05x	1.07x	1.09x	1.07x	1.08x
TIFF2PDF	1.39x	1.38x	1.40x	1.40x	1.40x	1.04x	1.07x	1.08x	1.07x	1.09x
EXIF	0.90x	1x	1.01x	1.01x	1.02x	0.86x	1.01x	1.01x	1.01x	1.01x
MJS	1x	1.01x	1.01x	1.01x	1.01x	1.07x	1.08x	1.09x	–	–
FLVMETA	1x	1x	1x	1.01x	1.01x	1x	1x	1x	1x	1x
OPENSSL	0.87x	0.89x	0.89x	1x	1.11x	1.10x	1.11x	1.15x	1.11x	1.11x
BASH	0.91x	0.75x	0.75x	1.13x	1.23x	0.83x	0.62x	0.66x	1x	1.03x
JQ	1x	1x	1x	1.01x	1.02x	1x	1x	1x	1.02x	1.02x
UNRTF	✗	✗	✗	1.01x	1.01x	1x	✗	✗	1.01x	1.02x
TCPDUMP	1x	✗	✗	1.03x	1.05x	0.47x	✗	✗	1.04x	1.04x
MUTOOL	1.02x	✗	✗	1.02x	1.06x	1.05x	✗	✗	1.11x	1.49x
XMLLINT	1.21x	✗	✗	1.25x	1.25x	1.01x	✗	✗	1.06x	1.06x
TIDYHTML	1x	✗	✗	1.01x	1.03x	1.04x	✗	✗	1.04x	1.05x
OPTIPNG	0.99x	✗	✗	1.05x	1.05x	1.10x	✗	✗	1.02x	1.05x
LIBXSLT	1.05x	✗	✗	1.08x	1.08x	1.04x	✗	✗	1.04x	1.12x

Consider DJPEG as an example. As presented in Table 8, PM_VOS removes system calls more intensively for DJPEG (431.0→16.6 per test case), leading to a more effective reduction of kernel execution time (see Table 13) and a faster fuzzing speed (see Table 5).

The PM_REC mode can support all our benchmark programs. We evaluate VOS under this mode and add the results in Table 5. As expected, the VOS maintains the stability of the fuzzing tools. Efficiency-wise, the VOS brings a 20.2%/29.6% increase in fuzzing speed to the PM_REC mode when applied to AFL/AFL++. As unveiled in Table 8, the increase in execution speed is similarly attributed to the reduction of system calls.

5.4 Benefits to Code Coverage

This evaluation measures how our optimizations can benefit greybox fuzzing tools. In the first part, we assess the contribution of our optimizations to code coverage. We count the number of control flow edges—without hit counts—covered by different modes in 24 hours. Table 6 presents the code coverage at the end of 24 hours. Due to the space limit, we focus on discussing the scenarios where both optimizations are deployed (i.e., PM_REC_VOS), considering FS, SS, and PM as the baselines.

Zooming into Table 6, we can see that PM_REC_VOS outperforms FS, SS, and PM in code coverage. When applied to AFL, PM_REC_VOS consistently achieves the highest code coverage. PM_REC_VOS averagely covers 8.8%, 5.3%, and 8.4% more code than FS, SS, and PM, respectively. The reason for the high code coverage is more than just fuzzing speed. As shown in Table 5, PM_VOS runs faster than PM_REC_VOS. However, in many cases (e.g., OBJDUMP, READELF, TIFF2PS, OPENSSL, BASH), PM_REC_VOS presents higher code coverage. This is because PM_REC_VOS achieves higher stability via its state recovery (see Table 5). In turn, stability alone cannot turn into code coverage. FS and SS provide higher stability than PM_REC_VOS but often cover less code. In short, the combined fuzzing speed and stability of PM_REC_VOS enable it to produce higher code coverage.

The situation with AFL++ is similar. PM_REC_VOS covers 9.0%, 17.4%, and 7.7% more code than FS, SS, and PM on average. Similarly,

Table 7: Results of bug-finding evaluation with MAGMA. The numbers stand for the average time-to-trigger of the corresponding bug. Under the same setting, the fuzzing mode producing the shortest time-to-trigger is highlighted with filled color. A lighter color is used when more modes produce the shortest time-to-trigger. "-" means the bug is not triggered by any instance in 24 hours, "✗" shows that the corresponding mode crashes, and "NA" means we did not run that mode (PHP and SQLITE3 do not provide standalone apps for fuzzing; on LIBPNG, our state recovery did not identify new global variables for the fuzzing driver).

Project	Bug ID	Standalone App										Driver			
		AFL					AFL++					AFL		AFL++	
		FS	PM	PM_VOS	PM_REC	PM_REC_VOS	FS	PM	PM_VOS	PM_REC	PM_REC_VOS	PM	PM_REC	PM	PM_REC
LIBXML2	XML017 [AAH041]	5m	✗	✗	32s	28s	11m	✗	✗	32s	30s	15s	15s	1m	48s
	XML009 [AAH032]	21h	✗	✗	12h	9h	14h	✗	✗	44m	40m	7h	6h	7h	4h
	XML003 [AAH026]	-	✗	✗	-	-	-	✗	✗	-	-	-	-	21h	21h
	XML001 [AAH024]	-	✗	✗	-	-	-	✗	✗	-	-	-	-	8h	4h
LIBPNG	PNG007 [AAH008]	-	-	-	-	-	-	-	-	-	-	20h	NA	1h	NA
	PNG006 [AAH007]	11m	2m	1m	2m	1m	3m	1m	51s	2m	1m	-	NA	-	NA
	PNG003 [AAH003]	-	-	-	-	-	-	-	-	-	-	15s	NA	15s	NA
	PNG001 [AAH001]	-	-	-	-	-	-	-	-	-	-	-	NA	19h	NA
LIBTIFF	TIF014 [AAH022]	-	17h	12h	20h	7h	15h	6h	1h	5h	3h	16h	14h	6h	3h
	TIF012 [AAH020]	23h	13h	5h	13h	5h	4h	3h	1h	2h	1h	2h	3h	1h	30m
	TIF009 [AAH017]	22h	15h	11h	13h	10h	10h	6h	4h	2h	1h	-	-	-	-
	TIF008 [AAH016]	-	-	-	-	-	-	-	-	-	-	-	-	20h	17h
	TIF007 [AAH015]	5h	6h	3h	4h	3h	24m	14m	1m	15m	3m	54m	26m	10m	2m
	TIF006 [AAH014]	21h	21h	12h	21h	20h	23h	19h	18h	18h	14h	-	-	-	-
	TIF002 [AAH010]	-	-	-	-	-	-	-	-	-	-	-	-	-	18h
POPPLER	PDF021 [JCH212]	-	-	-	-	-	-	21h	21h	19h	21h	-	-	-	-
	PDF019 [JCH210]	-	-	-	-	-	-	20h	20h	20h	19h	-	-	-	-
	PDF016 [JCH207]	3m	1m	1m	2m	1m	4m	3m	58s	1m	58s	1h	42m	10m	10m
	PDF011 [JCH201]	-	-	-	-	15h	23h	23h	21h	23h	20h	-	-	20h	20h
	PDF010 [AAH052]	4h	3h	2h	3h	3h	-	19h	13h	18h	11h	6h	6h	-	21h
	PDF003 [AAH045]	-	-	-	-	-	-	20h	20h	22h	17h	-	-	-	-
OPENSSL	SSL020 [MAE115]	14h	-	-	14h	11h	18h	23h	18h	14h	14h	12h	8h	23h	20h
	SSL009 [MAE104]	-	-	-	-	19h	-	-	-	21h	20h	1m	2m	20s	20s
	SSL002 [AAH056]	8m	7m	7m	5m	5m	8m	10m	9m	6m	6m	1m	1m	30s	30s
	SSL001 [AAH055]	-	-	-	-	-	21h	23h	21h	21h	21h	-	-	-	19h
PHP	PHP011 [MAE016]	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	12m	11m	13m	12m
	PHP009 [MAE014]	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	14m	14m	3h	50m
	PHP004 [MAE008]	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	7m	9m	3h	1h
SQLITE3	SQL018 [JCH232]	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	20h	14h	13h	10h
	SQL014 [JCH228]	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	18h	18h	16h	16h
	SQL002 [JCH215]	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	11h	7h	53m	45m

the high code coverage is due to PM_REC_VOS's combined fuzzing speed and stability: (i) PM_VOS has a higher fuzzing speed but lower code coverage on READELF, JQ, and BASH; (ii) FS and SS have higher stability but can barely compete with PM_REC_VOS in code coverage.

5.5 Benefits to Bug Finding

Benchmark Programs: During the evaluation, we observe crashes in BASH, MJS, OPTIPNG, and TIDYHTML. We replay the crashes using a clean binary with libraries statically linked and AddressSanitizer [77] enabled. All the crashes trigger a segmentation fault before ASan starts or incur an ASan abortion, showing no false positives. We triage the crashes based on ASan output and manually analyze the results. We identify 9 unique bugs, which all have been reported to the developers. Table 14 summarizes the number of unique crashes and bugs triggered by different fuzzing modes (aggregated from all 10 runs). The PM_REC_VOS mode leads other modes on both crash and bug numbers. More importantly, all bugs discovered by other modes are also triggered by PM_REC_VOS.

MAGMA: We further run a bug-finding evaluation on MAGMA [59]. We consider all seven projects from the official publication, for which reference results are available [34]. The fuzzing targets of MAGMA come in standalone applications and fuzzing drivers with the LIBFUZZER entry point. Our evaluation covers both forms. We find that PNGTEST, an application of LIBPNG, is not included in MAGMA but can lead to many MAGMA bugs. Thus, we add PNGTEST to our evaluation, producing 22 targets in total. We compile each target with AddressSanitizer (ASan) [77] and UndefinedBehaviorSanitizer (UBSan) [36] enabled, following the setting of OSS-Fuzz [33]. We disable the LeakSanitizer by setting `ASAN_OPTIONS=detect_leaks=0` because, otherwise, the persistent mode of AFL and AFL++ cannot run. We focus on UBSan's `signed-/unsigned-integer-overflow` checks because its checks on memory, such as `array-bounds`, are already covered by ASan. The other UBSan checks, such as `shift`, incur many false positives [48]. We run all fuzzing targets with the six modes described in §5.1. The snapshot mode fails to run as its restore operations on ASan's shadow memory trigger timeouts even if we use an excessive limit. Thus, we exclude the snapshot mode. Further, the fuzzing drivers run in the persistent mode, so we exclude them from the fork server mode. Finally, the fuzzing drivers do not incur system calls. Hence, we omit the VOS modes when testing them. We run each test 10 times for 24 hours and reuse the scripts [4] from MAGMA to count the bugs and measure the average time-to-trigger. The script defaults one week (168 hours) as the time-to-trigger to an instance not triggering a bug. Since our evaluations only run 24 hours, we change the default value to 24 hours.

Table 7 shows our evaluation results. Given standalone apps, our optimizations together (i.e., PM_REC_VOS) benefit AFL and AFL++ in two ways. First, they help trigger more bugs. PM_REC_VOS triggers 14/18 bugs when applied to AFL/AFL++, while FS only triggers

11/14 bugs and PM only triggers 9/15 bugs. Further, all the bugs triggered by FS and PM are covered by PM_REC_VOS. Second, our optimizations enable AFL and AFL++ to find bugs quicker. Compared to FS/PM, PM_REC_VOS triggers every bug faster. The average speedup is 3.71x/1.66x with AFL and 6.43x/2.11x with AFL++. Fundamentally, this is because our optimizations enable both FS-level stability and PM-level execution speed, as shown in Table 10.

Given fuzzing drivers, our state recovery alone (i.e., PM_REC) also benefits AFL and AFL++. When applied to AFL++, PM_REC finds 3 more bugs than PM. Furthermore, PM_REC consistently finds bugs faster than PM, leading to a 1.71x speedup. When applied to AFL, PM_REC does not discover more bugs but similarly reduces the time-to-trigger. Except for 3 cases (TIF012, SSL009, PHP004), PM_REC finds the bugs quicker than PM. The average speedup is 1.15x. The better effectiveness and efficiency are attributed to the improvement brought by PM_REC to the fuzzing stability by resetting affected global objects, as per Table 10. We believe the differences are due to the extra global variables REC resets. Resetting global variables benefits fuzzing in two ways. ① It improves the fuzzing stability, as shown in Table 10. ② It reduces invalid/incorrect fuzzing executions. Consider PHP009 as an example. The fuzzing driver for this bug uses a global variable `last_resource_number`. It should be reset to 0 on each new test case. Otherwise, the code at https://github.com/php/php-src/blob/bc39abe8c3c492e29bc5d60ca58442040bbf063b/Zend/zend_extensions.c#L260 will be affected, leading to incorrect and (thus) invalid executions. However, the driver never resets `last_resource_number` (see https://github.com/php/php-src/blob/bc39abe8c3c492e29bc5d60ca58442040bbf063b/sapi/fuzzer/fuzzer-exif.c). The built-in resetting logic lies in the initialization function, which is only executed once in the persistent mode. As a result of not handling `last_resource_number` and other similar cases, executions in the PM mode often become invalid and waste fuzzing cycles, eventually slowing down the discovery of the bug.

Compared to the reference results [34], our evaluation presents more visible differences on XML009 (with AFL++ on apps and drivers), TIF014 (with AFL on apps), PDF010 (with AFL++ on apps and drivers), and SSL001 (with AFL on apps and drivers). In those cases, AFL and AFL++ in our experiments run slower. The primary reason is we enabled sanitizers, which slowed the execution. Besides, some other reasons may also have contributed. ① With sanitizers, the targets run slower. More seeds may time out during the dry-run phase and get removed, and the fuzzer may pick different timeout thresholds. ② We used AFL++ 4.01c as required by our implementation, while MAGMA used 3.14a; ③ We keep AFL++'s deterministic mutations to reduce randomness, but it is disabled in MAGMA. ④ We used different hardware.

During the evaluation, ASan also detected a non-MAGMA bug in libtiff (fixed by [55]). To trigger the bug in 24 hours, one of our optimizations is mandated. We further checked and confirmed all other crashes are caused by MAGAM bugs, showing that our optimizations did not bring false positives when coupled with ASan.

6 DISCUSSION

Compatibility with Sanitizers: Sanitizers, such as ASan [77] and MemorySanitizer (MSan) [81], are critical to successful bug finding. An interesting question is whether our state recovery can co-exist with popular sanitizers. In principle, we follow the same mechanism as libFuzzer [78]: continuously running a piece of code on different test cases without restoring sanitizer metadata. Thus, we share the same level of sanitizer compatibility as libFuzzer.

Both libFuzzer and our state recovery can co-exist with ASan. We do not interfere with memory allocation and deallocation. Thus, the shadow memory—metadata of ASan—stays consistent with the memory layout. libFuzzer and our approach also do not affect MSan. We only affect global objects, which are created with default initialization and thus, excluded by MSan [81]. Nevertheless, MSan has two intrinsic issues when applied to fuzzing. First, it can incur false positives from un-instrumented dependencies [33]. Second, MSan is incompatible with ASan due to conflicts on the heap [10]. Compilers like Clang disallow ASan and MSan at the same time.

libFuzzer and our approach can impact LSan. We continuously execute the fuzzing target without cleanup operations (e.g., constructors registered by the program). Thus, allocated memory may not get recycled and trigger false LSan alarms. To avoid the false alarms, we must add the cleanup operations into the fuzzing loop.

Side Effects of VOS: Our VOS replaces partial functionality of the native OS. This may cause two side effects [58]. First, the VOS and the native OS can conflict when they have interleaving operations. For instance, VOS `open`s a file but misses `stat` operations on it. Those operations will be sent to the native OS, which will eventually fail as the native OS has no context. Our approach involves two mechanisms for mitigation: (i) we hook the low-level POSIX interfaces, which are more unified and easier to be identified comprehensively, and (ii) our profiling gives an estimation of the operations on the file, helping us inspect whether VOS can capture all expected operations.

Second, the VOS can have different behaviors from the OS. This will likely happen as VOS is heavily simplified, which is, however, OK if VOS preserves the behaviors of the target program. To this end, we follow the POSIX standard to implement the interfaces. As shown in §5.3, our implementation can preserve desired behaviors. In more complex scenarios such as multi-threaded execution, VOS can affect the target program differently from the OS as we may enforce different locks or incur disparate latencies. However, this is expected as such disparity also exists in real systems.

7 RELATED WORK

Algorithm Optimization for Fuzzing: Algorithm optimization is a major focus of improving greybox fuzzing. §2.1 has discussed algorithm optimizations via scheduling and mutations of test cases [40, 47, 48, 67, 68, 71, 72, 80, 87**?**]. Another perspective is to explore more effective *feedback*. AFL [86] considers code branches as feedback, which is refined by Steelix [65], CollAFL [56], and PTrix [49] to incorporate extra control-flow information. More aggressively, TaintScope [82], Vuzzer [72], GREYONE [57], REDQUEEN [40], and Angora [46] exploit feedback informed by data flow.

System Optimization for Fuzzing: Going beyond algorithm optimizations, past research has also extended efforts to improve the system design of grey-box fuzzing tools. Xu et al. [84] and NYX [75] replace heavy child-process spawning with lightweight snapshot recovery when executing different test cases. Xu et al. [84] further develop new OS primitives to mitigate the contention in the file system. These new primitives speed up the execution of the target programs and reduce the time needed for each round of fuzzing.

Other research in this line investigates more efficient approaches to gather feedback from the target programs. PTrix [49], Honggfuzz [16], and kAFL [76] take advantage of Intel PT [19] to efficiently collect control-flow-based feedback from the target program. UnTracer [70], instead of tracing every round of execution, instruments the target programs such that only new code is traced. RetroWrite [51] proposes static binary rewriting to trace code coverage in binary-only targets without heavy dynamic instrumentation.

Profile-driven Optimization: Our optimizations are based on dynamic analysis of some test cases. A closely related area is profile-guided optimization (PGO) [5] in the programming language community. Past PGO research has explored using profiles to improve pointer analysis for loop optimization [39, 83], detect data locality [60] or customize heap allocators [74] for reducing cache misses, and predict control flow frequency for improving function inline [44, 83]. Compared to PGO, we are more ambitious: we aim to eliminate the related operations or replace them with a minimized version.

OS Abstraction: Some symbolic executors also abstract OS interactions. For instance, KLEE [43] and CLOUD9 [50, 62] include components to simulate POSIX interfaces [6, 11]. However, those components are less suited for fuzzing optimizations. First, they intend to symbolize interactions with the OS, which can run even slower than the native version. Second, they are not informed by profiling results like ours. Thus, they only handle OS interactions specified by users, routing all the remaining to the native kernel.

8 CONCLUSION

This paper presents an empirical study that unveils how system-level designs of mainstream greybox fuzzing tools impact the execution speed. Following the study, this paper introduces two profile-driven optimizations to improve the system designs toward higher efficiency. The evaluations show the effectiveness of the proposed approach, illustrating the promise of profile-driven optimizations.

9 ACKNOWLEDGMENTS

We thank the anonymous reviewers and Marcel Busch, our shepherd, for their valuable comments. This work was supported by NSF award CNS-2213727. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government or NSF.

REFERENCES

[1] https://github.com/AFLplusplus/AFLplusplus/tree/stable/utils/socket_fuzzing. (Accessed on 01/19/2023).
[2] github.com/zardus/preeny. (Accessed on 01/19/2023).
[3] github.com/AFLplusplus/AFL-Snapshot-LKM. (Accessed on 01/19/2023).
[4] github.com/HexHive/magma/blob/v1.2/tools/report_df/main.py. (Accessed on 01/19/2023).
[5] en.wikipedia.org/wiki/Profile-guided_optimization. (Accessed on 01/19/2023).
[6] github.com/klee/klee/tree/master/runtime/POSIX. (Accessed on 01/19/2023).
[7] Afl. afl-1.readthedocs.io/en/latest/user_guide.html. (Accessed on 01/19/2023).
[8] Afl++ persistent mode. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md. (Accessed on 01/19/2023).
[9] Bash. tinyurl.com/bashtestcase. (Accessed on 01/19/2023).
[10] C and c++ error checking. tinyurl.com/asanmsan. (Accessed on 01/19/2023).
[11] Cloud9 posix. github.com/dslab-epfl/cloud9/tree/master/runtime/POSIX. (Accessed on 01/19/2023).
[12] Elf. https://tinyurl.com/elftestcase. (Accessed on 01/19/2023).
[13] Exif. https://github.com/ianare/exif-samples. (Accessed on 01/19/2023).
[14] Flvmeta. https://tinyurl.com/flxmetatest. (Accessed on 01/19/2023).
[15] Funchook. https://github.com/kubo/funchook. (Accessed on 01/19/2023).
[16] Honggfuzz. http://honggfuzz.com. (Accessed on 01/19/2023).
[17] Html. https://tinyurl.com/htmltestcase. (Accessed on 01/19/2023).
[18] Http. github.com/curl/curl-fuzzer/tree/master/corpora/curl_fuzzer_http. (Accessed on 01/19/2023).
[19] Intel pt. https://tinyurl.com/intelptdoc. (Accessed on 01/19/2023).
[20] Jq. https://tinyurl.com/jqtestcase. (Accessed on 01/19/2023).
[21] Js. github.com/cesanta/mjs/raw/master/tests/test_1.js. (Accessed on 01/19/2023).
[22] libfuzzer. https://llvm.org/docs/LibFuzzer.html. (Accessed on 01/19/2023).
[23] libpcap fuzzing driver on pcap_next_ex. https://github.com/the-tcpdump-group/libpcap/blob/master/testprogs/fuzz/fuzz_pcap.c. (Accessed on 01/19/2023).
[24] libpcap fuzzing driver on pcap_setfilter. https://github.com/the-tcpdump-group/libpcap/blob/master/testprogs/fuzz/fuzz_filter.c#L22. (Accessed on 01/19/2023).
[25] Linux system call table. https://thevivekpandey.github.io/posts/2017-09-25-linux-system-calls.html. (Accessed on 01/19/2023).
[26] llvm_mode persistent mode. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md. (Accessed on 01/19/2023).
[27] New in afl: persistent mode. https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html. (Accessed on 01/19/2023).
[28] New os primitives specialized for fuzzing. https://github.com/sslab-gatech/perf-fuzz. (Accessed on 01/19/2023).
[29] Optipng. https://tinyurl.com/pngtestcase. (Accessed on 01/19/2023).
[30] Pcap. https://tinyurl.com/pcaptest. (Accessed on 01/19/2023).
[31] Posix apis. https://tinyurl.com/posixapi. (Accessed on 01/19/2023).
[32] Quickjs. https://github.com/bellard/quickjs/raw/master/examples/hello.js. (Accessed on 01/19/2023).
[33] Setting up a new project. https://google.github.io/oss-fuzz/getting-started/new-project-guide/#sanitizers. (Accessed on 01/19/2023).
[34] Survival report. https://hexhive.epfl.ch/magma/reports/sample_2/. (Accessed on 01/19/2023).
[35] Tiff. lcamtuf.coredump.cx/afl/demo/afl_testcases.tgz. (Accessed on 01/19/2023).
[36] Ubsan. https://tinyurl.com/undefsan. (Accessed on 01/19/2023).
[37] Unrtf. https://tinyurl.com/unrtftest. (Accessed on 01/19/2023).
[38] Woff2. https://tinyurl.com/woff2test. (Accessed on 01/19/2023).
[39] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. Runtime pointer disambiguation. In *Proceedings of the 2015 ACM SIGPLAN OOPSLA*, 2015.
[40] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Network and Distributed System Security Symposium*, NDSS, 2018.
[41] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017.
[42] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing As Markov Chain. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2016.
[43] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.
[44] Dehao Chen, Tipp Moseley, and David Xinliang Li. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2016.
[45] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2018.
[46] Peng Chen and Hao Chen. Angora: efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, Oakland, 2018.
[47] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: Fuzzing Deeply Nested Branches. In *ACM SIGSAC CCS*, 2019.
[48] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Taowei, and Long Lu. SAVIOR: Towards Bug-Driven Hybrid Testing. In *IEEE Symposium on Security and Privacy*, Oakland, 2020. arXiv: 1906.07327.
[49] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary. In *ACM ASIA Conference on Computer and Communications Security*, ASIACCS, 2019. arXiv: 1905.10499.
[50] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review*, 43(4):5–10, 2010.
[51] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy*, Oakland, 2020.
[52] Dor1s. Testcase of pdf. https://github.com/google/AFL/blob/master/testcases/others/pdf/small.pdf. (Accessed on 01/19/2023).
[53] Dor1s. Testcase of xml. https://github.com/google/AFL/tree/master/testcases/others/xml. (Accessed on 01/19/2023).
[54] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX WOOT*, 2020.

[55] fixheaptiffcp. Libtiff fix. https://gitlab.com/libtiff/libtiff/-/commit/88d79a45a31c74cba98c697892fed5f7db8b963a. (Accessed on 01/19/2023).

[56] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy*, Oakland, 2018.

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

[58] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS*, 2003.

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

[60] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. Dmon: Efficient detection and correction of data locality problems using selective profiling. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 163–181, 2021.

[61] Petros Koutoupis. The linux ram disk. *LiNUX+ magzine*, pages 36–39, 2009.

[62] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 2012.

[63] Caroline Lemieux and Koushik Sen. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *ACM/IEEE International Conference on Automated Software Engineering*, ASE, 2018.

[64] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 2018.

[65] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *ACM Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, 2017.

[66] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[67] Chenyang Lv, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimize Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, USENIX, 2019.

[68] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. Ems: History-driven mutation for coverage-based fuzzing. In *Annual Network and Distributed System Security Symposium*, 2022.

[69] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[70] Stefan Nagy and Matthew Hicks. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. In *IEEE S&P*, 2019.

[71] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy*, Oakland, 2018.

[72] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium*, NDSS, 2017.

[73] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[74] Joe Savage and Timothy M Jones. Halo: Post-link heap-layout optimisation. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 94–106, 2020.

[75] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium*, 2021.

[76] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, USENIX, 2017.

[77] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 {USENIX} Annual Technical Conference ({USENIX} {ATC} 12)*, pages 309–318, 2012.

[78] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE, 2016.

[79] Kostya Serebryany. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *USENIX Security Symposium*, USENIX, 2017.

[80] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *IEEE Symposium on Security and Privacy*, Oakland, 2019.

[81] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55. IEEE, 2015.

[82] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *IEEE Symposium on Security and Privacy*, Oakland, 2010.

[83] Bob Wilson, Diego Novillo, and Chandler Carruth. Pgo and llvm status and current work. https://llvm.org/devmtg/2013-11/slides/Carruth-PGO.pdf, 2013.

[84] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing New Operating Primitives to Improve Fuzzing Performance. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017.

[85] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. ProFuzzer: On-the-fly Input Type Probing for Better Zero-day Vulnerability Discovery. In *IEEE S&P*, 2019.

[86] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl, 2014.

[87] Kunpeng Zhang, Xi Xiao, Xiaogang Zhu, Ruoxi Sun, Minhui Xue, and Sheng Wen. Path transitions tell more: Optimizing fuzzing schedules via runtime program states. In *44th IEEE/ACM International Conference on Software Engineering*, 2022.

[88] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 2022.

Table 8: Average number of system calls invoked by the target program to handle one test case. `File: {open, read, write, close, access, stat, seek, fcntl, unlink, readlink, rename, dir, cmd, dup, ioctl}`; `Proc: {mmap, munmap, brk, protect, shm}`; `Mem: {pipe, exec, pid, sid, wait, clone, rlimit, sysinfo, rt_sigaction, prctl, exit}`.

Program	Syscall Group	FS_FSTMP	SS	PM	PM_VOS	PM_REC	REC_VOS
OBJDUMP	File	116.7 (91.3%)	118.3 (90.6%)	51.8 (88.3%)	9.4 (57.6%)	79.8 (91.8%)	9.9 (57.6%)
	Proc	4.0 (3.1%)	2.0 (1.5%)	1.0 (1.7%)	1.0 (6.2%)	1.9 (2.2%)	2.0 (11.6%)
	Mem	4.1 (3.2%)	3.1 (2.4%)	0.1 (0.1%)	0.1 (0.1%)	0.1 (0.1%)	0.1 (0.2%)
	Other	3.1 (2.4%)	7.2 (5.5%)	5.9 (9.9%)	5.9 (36.1%)	5.2 (5.9%)	5.2 (30.2%)
READELF	File	97.6 (91.4%)	106.5 (90.5%)	69.1 (91.8%)	9.1 (59.6%)	77.0 (92.6%)	10.3 (62.6%)
	Proc	3.0 (2.8%)	1.0 (0.8%)	1.0 (1.3%)	1.0 (6.5%)	1.0 (1.2%)	1.0 (6.1%)
	Mem	3.0 (2.8%)	3.0 (2.6%)	0.1 (0.1%)	0.1 (0.3%)	0.1 (0.1%)	0.1 (0.3%)
	Other	3.1 (2.9%)	7.1 (6.1%)	5.1 (6.8%)	5.1 (31.0%)	5.1 (6.2%)	5.1 (31.0%)
UNRTF	File	135.5 (86.4%)	132.6 (85.1%)	–	–	118.6 (91.5%)	17.4 (60.6%)
	Proc	3.0 (1.9%)	1.0 (0.6%)	–	–	1.0 (0.8%)	1.0 (3.5%)
	Mem	15.2 (9.7%)	15.2 (9.7%)	–	–	4.8 (3.7%)	5.0 (17.5%)
	Other	3.1 (1.9%)	7.1 (4.6%)	–	–	5.2 (4.0%)	5.2 (18.3%)
WOFF2	File	18.5 (74.5%)	16.5 (66.4%)	16.5 (72.8%)	9.5 (60.4%)	16.5 (72.8%)	9.5 (6.0%)
	Proc	3.0 (12.1%)	1.0 (4.0%)	1.0 (4.4%)	1.0 (6.4%)	1.0 (4.4%)	1.0 (6.4%)
	Mem	0.2 (0.8%)	0.2 (0.8%)	0.1 (0.1%)	0.1 (0.3%)	0.1 (0.1%)	0.1 (0.3%)
	Other	3.1 (12.6%)	7.1 (28.7%)	5.1 (22.7%)	5.1 (32.8%)	5.1 (22.6%)	5.1 (32.8%)
QUICKJS	File	22.7 (70.8%)	20.5 (66.5%)	20.5 (76.2%)	9.1 (59.2%)	20.5 (76.2%)	9.1 (59.2%)
	Proc	3.0 (9.4%)	1.0 (3.2%)	1.0 (3.7%)	1.0 (6.5%)	1.0 (3.7%)	1.0 (6.5%)
	Mem	2.2 (6.9%)	2.2 (7.2%)	0.2 (0.9%)	0.1 (0.9%)	0.3 (1.0%)	0.1 (0.9%)
	Other	4.1 (12.9%)	7.1 (2.3%)	5.1 (19.1%)	5.1 (33.4%)	5.1 (19.1%)	5.1 (33.3%)
DJPEG	File	426.8 (97.8%)	426.8 (98.0%)	423.7 (98.3%)	9.3 (55.8%)	427.3 (98.3%)	9.3 (55.8%)
	Proc	3.0 (0.7%)	2.0 (0.5%)	1.0 (0.2%)	1.0 (6.1%)	1.0 (0.2%)	1.0 (6.1%)
	Mem	3.4 (0.8%)	3.4 (0.8%)	0.9 (0.2%)	0.9 (5.4%)	0.9 (0.2%)	0.9 (5.4%)
	Other	3.1 (0.7%)	3.1 (0.7%)	5.4 (1.3%)	5.4 (32.7%)	5.4 (1.3%)	5.4 (32.7%)
TIFF2PS	File	68.7 (85.9%)	65.9 (83.0%)	59.6 (87.5%)	9.2 (55.3%)	59.6 (87.5%)	9.9 (57.1%)
	Proc	3.9 (4.9%)	1.9 (2.5%)	1.0 (1.5%)	1.0 (6.0%)	1.0 (1.5%)	1.0 (5.8%)
	Mem	4.1 (5.2%)	4.1 (5.2%)	2.1 (3.1%)	1.1 (6.2%)	2.1 (3.0%)	1.1 (6.0%)
	Other	3.1 (3.9%)	7.4 (9.3%)	5.4 (7.9%)	5.4 (32.5%)	5.4 (8.0%)	5.4 (31.1%)
TCPDUMP	File	33.9 (63.9%)	31.7 (57.8%)	–	–	24.2 (55.8%)	11.4 (37.3%)
	Proc	9.0 (17.0%)	8.0 (14.6%)	–	–	8.0 (18.4%)	8.0 (26.1%)
	Mem	2.0 (3.8%)	2.0 (3.7%)	–	–	0.1 (0.1%)	0.1 (0.2%)
	Other	8.1 (15.3%)	13.1 (23.9%)	–	–	11.2 (25.7%)	11.2 (36.4%)
MUTOOL	File	91.5 (85.6%)	88.6 (84.4%)	–	–	82.7 (92.6%)	14.1 (69.4%)
	Proc	3.0 (2.8%)	1.0 (0.9%)	–	–	1.0 (1.1%)	1.0 (4.9%)
	Mem	8.3 (7.7%)	8.3 (7.9%)	–	–	0.4 (0.5%)	0.1 (0.4%)
	Other	4.1 (3.9%)	7.1 (6.8%)	–	–	5.1 (5.8%)	5.1 (25.3%)
XMLLINT	File	135.2 (93.0%)	133.0 (91.6%)	–	–	105.2 (92.7%)	17.3 (67.8%)
	Proc	3.0 (2.1%)	1.0 (1.0%)	–	–	1.0 (1.0%)	1.0 (3.9%)
	Mem	4.0 (2.7%)	4.1 (2.8%)	–	–	2.0 (1.8%)	2.0 (7.9%)
	Other	3.1 (2.2%)	7.1 (4.9%)	–	–	5.2 (4.6%)	5.2 (20.4%)
TIDYHTML	File	2187.7 (99.5%)	2185.9 (99.5%)	–	–	2184.9 (99.6%)	10.2 (58.4%)
	Proc	3.0 (0.1%)	1.0 (0.1%)	–	–	1.0 (0.1%)	1.0 (5.7%)
	Mem	4.0 (0.2%)	4.0 (0.2%)	–	–	2.0 (0.1%)	1.1 (6.5%)
	Other	3.1 (0.1%)	7.1 (0.3%)	–	–	5.1 (0.2%)	5.1 (29.3%)
OPTIPNG	File	41.2 (83.3%)	39.2 (82.6%)	–	–	39.2 (86.3%)	9.3 (58.8%)
	Proc	3.0 (6.1%)	1.0 (2.1%)	–	–	1.0 (2.2%)	1.0 (6.3%)
	Mem	2.1 (4.3%)	0.1 (0.2%)	–	–	0.1 (0.2%)	0.4 (2.5%)
	Other	3.1 (6.3%)	7.1 (15.0%)	–	–	5.1 (11.3%)	5.1 (32.4%)

Table 9: Time needed for profiling-and-patching. Analysis means the dynamic analysis in the profiling phase. Rebuild refers to patching the target program.

Program	# of Testcases		Analysis		Rebuild		Total	
	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++
OBJDUMP	449	671	6s	5s	10s	10s	16s	15s
READELF	716	1,705	6s	5s	10s	9s	16s	15s
UNRTF	357	371	6s	5s	12s	2s	18s	7s
WOFF2	456	529	6s	8s	11s	6s	17s	14s
QUICKJS	879	1,255	6s	6s	5s	5s	11s	11s
DJPEG	536	688	3s	3s	1s	1s	4s	4s
TIFF2PS	593	737	3s	1s	1s	5s	4s	6s
TCPDUMP	365	545	12s	11s	41s	50s	53s	61s
MUTOOL	569	1,117	14s	13s	22s	16s	26s	29s
XMLLINT	789	1,582	17s	12s	22s	24s	39s	36s
TIDYHTML	1,721	1,828	10s	12s	5s	5s	15s	17s
OPTIPNG	328	525	3s	4s	3s	3s	6s	7s
TIFF2PDF	588	862	10s	9s	1s	10s	11s	19s
LIBXSLT	1,382	1,300	27s	32s	22s	25s	49s	57s
OPENSSL	208	228	23s	25s	16s	18s	39s	43s
BASH	614	1,157	20s	21s	25s	28s	45s	49s
EXIF	406	499	12s	10s	12s	15s	24s	25s
MJS	806	896	5s	8s	10s	12s	15s	20s
FLVMETA	120	173	11s	15s	21s	22s	33s	37s
JQ	283	295	15s	17s	18s	22s	33s	39s

Table 10: Statistic results of MAGMA evaluation. ✗ indicates the fuzzing tool failed to run that mode. "# of GV" means the number of global variables identified by our state recovery (the numbers are the same for AFL and AFL++). Fuzzing Speed is measured by the number of test cases processed in 24 hours and normalized using FS as the baseline. For example, PM's result is calculated by dividing the execution speed of PM by that of FS. Stability lower than 90% is highlighted in purple.

Mode	Project	Programs	# of GV	AFL									AFL++								
				Stability (%)					Fuzzing Speed (baseline:FS)				Stability (%)					Fuzzing Speed (baseline:FS)			
				FS	PM	PM_VOS	PM_REC	PM_REC_VOS	PM	PM_VOS	PM_REC	PM_REC_VOS	FS	PM	PM_VOS	PM_REC	PM_REC_VOS	PM	PM_VOS	PM_REC	PM_REC_VOS
Driver	LIBXML2	XML_READ_MEMORY	36	-	97.1	-	99.8	-	1x	-	0.84x	-	-	97.9	-	99.0	-	1x	-	0.98x	-
	LIBPNG	PNG_READ_FUZZER	0	-	99.1	-	-	-	1x	-	-	-	-	99.1	-	-	-	1x	-	-	-
	LIBTIFF	TIFF_READ_RGBA	4	-	99.7	-	99.8	-	1x	-	0.98x	-	-	99.7	-	99.7	-	1x	-	0.98x	-
	POPPLER	PDF_FUZZER	29	-	99.7	-	99.7	-	1x	-	1.07x	-	-	99.6	-	99.9	-	1x	-	0.90x	-
	OPENSSL	ASN1_DRIVER	132	-	88.4	-	96.5	-	1x	-	0.86x	-	-	97.5	-	96.2	-	1x	-	0.71x	-
		ASN1PARSE_DRIVER	56	-	77.4	-	99.8	-	1x	-	1.31x	-	-	74.7	-	99.1	-	1x	-	1.33x	-
		BIGNUM_DRIVER	50	-	98.9	-	99.4	-	1x	-	1.16x	-	-	98.3	-	99.0	-	1x	-	1.38x	-
		SERVER_DRIVER	142	-	90.9	-	97.2	-	1x	-	0.89x	-	-	91.1	-	96.1	-	1x	-	0.74x	-
		CLIENT_DRIVER	139	-	86.2	-	95.0	-	1x	-	0.51x	-	-	86.8	-	95.0	-	1x	-	0.62x	-
		X509_DRIVER	111	-	83.3	-	95.6	-	1x	-	0.62x	-	-	82.7	-	95.4	-	1x	-	0.49x	-
	PHP	PHP_EXIF	363	-	94.7	-	99.4	-	1x	-	0.51x	-	-	95.3	-	99.4	-	1x	-	0.75x	-
	SQLITE3	SQLITE3_FUZZ	21	-	97.1	-	97.4	-	1x	-	1.03x	-	-	98.2	-	98.4	-	1x	-	0.73x	-
Standalone App	LIBXML2	XMLLINT	51	99.9	✗	✗	99.9	99.9	✗	✗	2.63x	2.69x	99.9	✗	✗	99.9	99.9	✗	✗	4.74x	6.69x
	LIBPNG	PNGTEST	12	99.8	98.7	98.7	99.8	99.8	1.33x	2.15x	1.30x	1.70x	99.8	99.8	99.8	99.8	99.8	4.06x	4.77x	3.66x	4.31x
	LIBTIFF	TIFFCP	13	99.9	99.5	99.5	99.8	99.8	3.95x	5.95x	3.44x	5.34x	99.9	98.0	98.0	99.1	99.1	3.61x	4.61x	3.51x	4.11x
	POPPLER	PDFTOPPM	28	99.9	97.3	97.3	99.8	99.8	1.52x	1.78x	1.55x	1.57x	99.9	95.5	95.5	99.8	99.8	1.16x	1.19x	1.55x	1.65x
		PDFIMAGES	17	99.9	98.7	98.7	99.9	99.9	1.67x	1.92x	1.86x	2.01x	99.9	97.9	97.9	99.8	99.8	1.30x	1.71x	1.41x	1.67x
	OPENSSL	ASN1	132	99.9	30.6	30.1	96.3	96.3	1.74x	1.82x	1.54x	1.59x	99.9	33.5	33.4	99.1	99.1	1.63x	1.68x	1.49x	1.57x
		ASN1PARSE	56	99.9	55.6	55.8	99.3	99.3	1.94x	3.03x	1.74x	1.82x	99.9	55.9	55.8	99.1	99.1	1.99x	2.08x	2.38x	2.55x
		BIGNUM	50	99.9	77.1	76.8	99.2	99.2	1.87x	2.17x	1.65x	2.09x	99.9	74.4	74.4	99.0	99.0	1.74x	3.27x	1.86x	2.00x
		SERVER	142	99.9	✗	✗	96.1	96.2	✗	✗	1.42x	1.47x	99.9	✗	✗	96.2	96.2	✗	✗	1.21x	1.22x
		CLIENT	139	99.9	✗	✗	95.0	95.0	✗	✗	1.77x	1.81x	99.9	✗	✗	95.1	95.1	✗	✗	1.72x	1.73x
		X509	111	99.9	38.0	38.1	99.2	99.2	2.87x	3.41x	2.78x	3.08x	99.9	37.2	37.2	95.4	95.4	4.15x	4.62x	2.23x	2.48x

Table 11: Average number of page faults incurred by the target program to handle one test case (using AFL as the fuzzing tool).

Program	FS_FSTMP	SS	PM	PM_VOS	PM_REC	REC_VOS
OBJDUMP	131.1	38.3	0.1	0.1	0.7	1.2
READELF	90.8	16.2	0.2	1.4	0.2	1.4
WOFF2	185.1	112.4	9.9	10.1	9.9	10.1
QUICKJS	116.9	32.3	6.9	3.1	6.8	3.2
DJPEG	112.4	106.5	5.3	6.4	5.3	6.4
TIFF2PS	88.3	23.5	5.7	4.7	5.6	4.7
UNRTF	97.3	✗	✗	✗	12.7	13.7
TCPDUMP	90.5	31.9	✗	✗	0.7	1.9
MUTOOL	343.7	223.1	✗	✗	4.4	14.8
XMLLINT	98.6	20.9	✗	✗	2.2	2.2
TIDYHTML	89.5	23.4	✗	✗	1.2	1.4
OPTIPNG	72.2	12.4	✗	✗	0.6	1.7

```
1 int main (int argc, char **argv){
2     int c; ...
3     /* loop we insert to enable persistent mode*/
4     while (__AFL_LOOP(10000)) {
5         take_snapshot();
6         /* beginning of main function */ ...
7         xmalloc_set_program_name (program_name);
8         ... /* end of main function */
9         restore_snapshot();
10    }
11    return exit_status;
12 }
```

Figure 10: Enabling persistent mode for OBJDUMP. In the snapshot mode or variable recovery mode, code is inserted to snapshot/restore the memory or data objects at the beginning (line 5)/end (line 9) of a loop iteration.

```
1 static type1 defaultEntityLoader = NULL; // a global pointer
2 int main(int argc, char **argv){
3     xmllintExternalEntityLoader();
4     defaultEntityLoader = xmllintExternalEntityLoader; //set up
          defaultEntityLoader'' in first iteration
5 }
6 type2 xmllintExternalEntityLoader() {
7     if (defaultEntityLoader != NULL) { // False in first iteration; True
          in follow-up iterations
8       defaultEntityLoader(...); //After first iteration,
            xmllintExternalEntityLoader is called recursively
9     }
10 }
```

Figure 11: Why persistent mode fails to run LIBXML2. defaultEntityLoader is initialized as NULL (line 1). In the first iteration, it will be assigned to function xmllintExternalEntityLoader (line 4), thus becoming non-NULL. In the second iteration, when xmllintExternalEntityLoader is called (line 3), the condition at line 9 will be true. Thus, defaultEntityLoader (pointing to the function xmllintExternalEntityLoader itself) will be called at line 8. This will lead to endless calls to xmllintExternalEntityLoader

Table 12: Number of objects identified by profiling: 5-minute profiling *v.s.* 24-hour profiling *v.s.* all global objects. Each cell is formatted as [X(Y, Z)] (X: # of objects; Y: # of memory pages of the objects; Z: total size of the objects.

Program	5 min		24 hour		All
	AFL	AFL++	AFL	AFL++	–
OBJDUMP	51(6, 6.1K)	51(6, 6.1K)	56(5, 9.4K)	55(7, 9.4K)	219(10, 25.0K)
READELF	64(5, 10.4K)	64(5, 10.4K)	75(6, 10.7K)	86(8, 11.5K)	284(16, 25.4K)
UNRTF	39(4, 756)	39(4, 756)	42(4, 202.7K)	42(4, 202.7K)	107(7, 209.8K)
WOFF2	3(1, 496)	3(1, 496)	3(1, 496)	3(1, 496)	11(1, 1.6K)
QUICKJS	3(3, 56)	3(3, 56)	3(3, 56)	3(3, 56)	22(3,392)
DJPEG	3(1, 20)	3(1, 20)	3(1, 20)	3(1, 20)	9(1, 256)
TIFF2PS	21(3, 104)	21(3, 104)	21(3, 104)	21(3, 104)	53(4, 704)
TCPDUMP	34(16, 836.7K)	35(15, 830.0K)	78(28, 1.1M)	78(28, 1.1M)	593(46, 1.4M)
MUTOOL	19(6, 4.6K)	19(6, 4.6K)	37(13, 5.6K)	37(13, 5.6K)	472(61, 209.5K)
XMLLINT	27(6, 1.1K)	27(6, 1.1K)	27(6, 1.1K)	27(6, 1.1K)	253(10, 56.2K)
TIDYHTML	9(5, 6.4K)	9(5, 6.4K)	9(5, 6.4K)	9(5, 6.4K)	41(12, 81.6K)
OPTIPNG	21(2, 592)	16(2, 504)	24(2, 560)	20(2, 532)	80(4, 65.4K)
TIFF2PDF	1(1, 4)	1(1, 4)	1(1, 4)	1(1, 4)	16(1, 692)
LIBXSLT	35(3, 1.6K)	35(3, 1.6K)	35(3, 1.6K)	35(3, 1.6K)	161(3, 6.2K)
OPENSSL	110(13, 8.8K)	110(13, 8.8K)	121(13, 8.8K)	121(13, 8.8K)	330(24, 24.3K)
BASH	88(10, 3.2K)	88(10, 3.2K)	203(14, 7.9K)	203(14, 7.9K)	1238(19, 78.7K)
EXIF	3(1, 64)	3(1, 64)	3(1, 64)	3(1, 64)	10(1, 304)
MJS	6(1, 448)	0	6(1, 448)	0	14(1, 692)
FLVMETA	3(1, 860)	3(1, 860)	3(1, 860)	3(1, 860)	7(1, 1.1K)
JQ	6(1, 40)	6(1, 40)	10(1, 56)	10(1, 56)	58(2, 17.8K)

Table 13: Percentage of time spent in kernel during target execution (on AFL).

Program	PM	PM_VOS	PM_REC	PM_REC_VOS
OBJDUMP	18.6%	13.9%	26.1%	14.2%
READELF	26.9%	12.9%	24.1%	10.9%
WOFF2	15.1%	11.9%	12.9%	11.9%
QUICKJS	4.1%	3.4%	4.8%	4.1%
DJPEG	15.4%	5.6%	16.0%	4.6%
TIFF2PS	24.9%	12.9%	24.0%	10.9%
UNRTF	✗	✗	10.8%	8.8%
TCPDUMP	✗	✗	29.1%	23.7%
MUTOOL	✗	✗	7.9%	4.9%
XMLLINT	✗	✗	33.5%	26.5%
TIDYHTML	✗	✗	22.6%	3.5%
OPTIPNG	✗	✗	15.6%	9.5%

Table 14: Unique crashes and bugs triggered by different fuzzing modes.

Program	Type	FS		SS		PM		PM_VOS		PM_REC		PM_REC_VOS	
		AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++
BASH	crash	9	421	0	273	0	0	0	0	0	837	7	1,391
	bug	1	4	0	3	0	0	0	0	0	5	2	5
MJS	crash	134	22	100	102	204	130	107	116	107	–	177	–
	bug	1	2	1	2	5	2	2	2	2	–	5	–
OPTIPNG	crash	0	0	0	4	✗	✗	✗	✗	0	1	0	3
	bug	0	0	0	1	✗	✗	✗	✗	0	1	0	1
TIDYHTML	crash	41	0	44	4	✗	✗	✗	✗	82	1	52	3
	bug	1	0	1	1	✗	✗	✗	✗	1	1	1	1