Designing New Operating Primitives to Improve Fuzzing Performance

Wen Xu Sanidhya Kashyap Changwoo Min[†] Taesoo Kim

Georgia Institute of Technology Virginia Tech[†]

ABSTRACT

Fuzzing is a software testing technique that finds bugs by repeatedly injecting mutated inputs to a target program. Known to be a highly practical approach, fuzzing is gaining more popularity than ever before. Current research on fuzzing has focused on producing an input that is more likely to trigger a vulnerability.

In this paper, we tackle another way to improve the performance of fuzzing, which is to shorten the execution time of each iteration. We observe that AFL, a state-of-the-art fuzzer, slows down by 24× because of file system contention and the scalability of `fork()` system call when it runs on 120 cores in parallel. Other fuzzers are expected to suffer from the same scalability bottlenecks in that they follow a similar design pattern. To improve the fuzzing performance, we design and implement three new operating primitives specialized for fuzzing that solve these performance bottlenecks and achieve scalable performance on multi-core machines. Our experiment shows that the proposed primitives speed up AFL and LibFuzzer by 6.1 to 28.9× and 1.1 to 735.7×, respectively, on the overall number of executions per second when targeting Google's fuzzer test suite with 120 cores. In addition, the primitives improve AFL's throughput up to 7.7× with 30 cores, which is a more common setting in data centers. Our fuzzer-agnostic primitives can be easily applied to any fuzzer with fundamental performance improvement and directly benefit large-scale fuzzing and cloud-based fuzzing services.

CCS CONCEPTS

• **Security and privacy** → Vulnerability scanners; • **Software and its engineering** → Software testing and debugging;

KEYWORDS

fuzzing; scalability; operating system

1 INTRODUCTION

Attackers exploit vulnerabilities to gain complete or partial control of user devices or achieve user privacy. In order to protect users from malicious attacks, various financial organizations and communities, that implement and use popular software and OS kernels, have invested major efforts on finding and fixing security bugs in their products, and one such effort to finding bugs in applications and libraries is fuzzing. Fuzzing is a software-testing technique that works by injecting randomly mutated input to a target program. Compared with other bug-finding techniques, one of the primary advantages of fuzzing is that it ensures high throughput with less manual efforts and pre-knowledge of the target software. In addition, it is one of the most practical approach to finding bugs in various critical software. For example, the state-of-the-art coverage-driven fuzzer, *American Fuzzy Lop* (AFL), has discovered over thousand vulnerabilities in open source software. Not only that, even the security of Google Chrome heavily relies on its fuzzing infrastructure, called *ClusterFuzz* [3, 20].

Fuzzing is a random testing technique, which requires huge computing resources. For example, ClusterFuzz is a distributed infrastructure that consists of several hundred virtual machines processing 50 million test cases a day. The recently announced *OSS-Fuzz* [22], Google's effort to make open source software more secure, is also powered by ClusterFuzz that processes ten trillion test inputs a day on an average, and has found over one thousand bugs in five months [14]. Besides Google, Microsoft proposed *Project Springfield* [31] that provides a cloud-based fuzzing service to developers for security bug finding in their software. This movement clearly indicates that large-scale fuzzing is gaining popularity [1, 25].

By looking at the complex software stacks and operating systems, the performance of a fuzzer is critical. In other words, a better fuzzer likely hits more security bugs in the target program more quickly than other fuzzers. There are two broad research directions to tackle this problem: 1) producing an input that is more likely to trigger a vulnerability (*i.e.*, shorter time required to reach a bug); and 2) shortening the execution time of each iteration (*i.e.*, more coverage at a given time).

Prior research has mainly focused on the first approach. In particular, coverage-driven fuzzers [21, 29, 41] evaluate a test case by the runtime coverage of a target program, and then try to generate test cases that are likely to cover untouched branches. More advanced techniques include an evolutionary technique [12, 34], a statistical model such as Markov chain [7], or combine fuzzing with symbolic execution [13, 27, 36]. However, shortening the execution time of each iteration of fuzzing also brings huge benefits. This approach reduces the time to find a new bug given a fuzzing strategy. Nowadays it takes fuzzers days, weeks or even months to find an exploitable vulnerability in popular software due to its internal complexity and security mitigations. Thus a slight improvement on

CCS '17, October 30-November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134046

the performance makes a big difference, and the huge operational costs of fuzzing are thereby saved.

A critical performance issue is that current fuzzers are not at all scalable on commodity OSes with manycore architectures that are readily available today (*e.g.*, 2-4 sockets with 16-64 physical cores). As Figure 2 shows, the performance scalability of fuzzing is *surprisingly poor*. Most of the CPU cycles are wasted because of contention in underlying OS, which *degrades* the scalability of fuzzing with no more than 30 CPU cores. This is conceptually counter-intuitive for fuzzing tasks since each fuzzing instance runs independently without explicit contention.

We found that many state-of-the-art fuzzers [7, 12, 21, 29, 34, 38, 41] have a similar structure (*i.e.*, launching and monitoring the target application, creating and reading test cases, and optionally syncing test cases among fuzzer instances in an iteration of fuzzing). To complete these involved steps, they extensively rely on several OS primitives while fuzzing a target application, which are the root causes of the similar performance and scalability bottlenecks they suffer from.

First, in each fuzzing iteration of a state-of-the-art fuzzer such as AFL, it invokes fork() to clone the target application for a fresh run. However, spawning processes simultaneously on multiple cores by using the fork() system call is not scalable [8, 9, 11]. fork() is designed to duplicate any running process. In the context of fuzzing, a large portion of operations in fork() is repeated but has the same effects, as the target process never changes.

Second, a fuzzer instance intensively interacts with the file system by creating, writing, and reading small test cases for each run. These file operations cause heavy updates to the metadata of the file system, which is not scalable in terms of parallel fuzzing on multiple cores [32].

Last but not least, existing fuzzers including AFL and LibFuzzer share test cases among fuzzer instances in parallel fuzzing. A fuzzer instance periodically scans the output directories of other instances to discover new test cases. Moreover, it re-executes these external test cases to determine whether they are interesting or not. The number of directory enumeration operations and executions of the target application increases non-linearly with more fuzzers and longer fuzzing, which causes the third bottleneck.

In this paper, we propose three operating primitives for fuzzers to achieve scalable fuzzing performance on multi-core machines. These three primitives are specific to fuzzing and they aim at solving the three performance bottlenecks described above. In particular, we propose 1) snapshot(), a new system call which clones a new instance of the target application in an efficient and scalable manner; 2) dual file system service, which makes fuzzers operate test cases on a memory file system (*e.g.*, tmpfs) for performance and scalability and meanwhile ensures capacity and durability by a disk file system (*e.g.*, ext4); 3) shared in-memory test case log, which helps fuzzers share test case execution information in a scalable and collaborative way.

In this paper, we make the following contributions:

- We identify and analyze three prominent performance bottlenecks that stem in large-scale fuzzing and they are caused by the intensive use of existing operating primitives that are only better for the general purpose use.

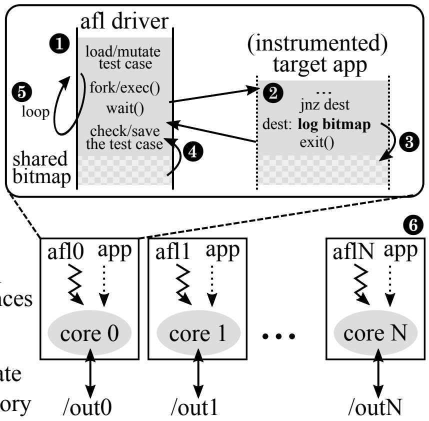

Figure 1: Overview of the AFL design. ❶ Read/sync a candidate test case from own/other's output directory into a buffer, and mutate the buffer for a new input and feed it to the target process; ❷ (target process) fork a child process to execute the program with visited paths recorded in the tracing bitmap; ❸ (target process) wait for the child to terminate and send back its exit status; ❹ save the generated input into the output directory if it covers new paths by observing the shared tracing bitmap; ❺ repeat this fuzzing process; and ❻ on multi-core systems, each AFL instance run independently in parallel.

- We design and implement three new fuzzing specific operating primitives that can improve the performance and scalability for the state-of-the-art fuzzers in a multi-core machine.
- We apply and evaluate our proposed operating primitives to AFL and LibFuzzer. By leveraging our proposed primitives, AFL has at most 7.7×, 25.9×, and 28.9× improvement on the number of executions per second on 30, 60, and 120 cores, respectively. Meanwhile, LibFuzzer can speed up by at most 170.5×, 134.9×, and 735.7× on 30, 60, and 120 cores respectively.

§2 describes the roles of operating primitives in fuzzing and how they become critical performance bottlenecks. The design of new operating primitives specialized for fuzzing is proposed in §3, and §4 describes how new primitives help the state-of-the-art fuzzers scale on multiple cores. §5 mentions the related implementation details, and the evaluation of the proposed primitives is illustrated in §6. The related works are listed in §7. §9 concludes the paper.

2 BACKGROUND AND MOTIVATION

In this section, we first describe how modern fuzzers work (§2.1) and then explain how and why operating primitives, on which fuzzers rely, become critical scalability bottlenecks (§2.3, §2.4).

2.1 Fuzzing Explained

Fuzzing is a widely used software-testing technique to find bugs in applications. It tests software by injecting randomly mutated input to a target program and monitors whether the target behaves abnormally (*e.g.*, crashed, raced, hanged, or failed on assertions). The input that triggers an abnormal behavior is reported as a potential bug. To detect bugs quickly, various techniques [7, 13, 20, 29, 34, 36, 41] have been proposed to wisely mutate inputs with smart policies and efficiently explore the program state. In particular, popular fuzzers such as `AFL` [41] and `LibFuzzer` [29] use the past code coverage to determine whether the current mutated input is interesting, and use it as a feedback for the next execution. To get coverage information, fuzzers require either instrumentation (if the source code is available) or a system emulation layer (for binaries) such as QEMU. In general, a fuzzer starts with a set of seed inputs (also known as a corpus), runs a target program, and mutates the input based on the feedback (e.g., coverage or crash) from the past execution. This whole process is known as a fuzzing loop, and a fuzzer can iterate for a certain amount of time or until it reaches a saturation point (alas, indefinitely in most cases). Precisely, a fuzzing loop consists of the following procedures:

(1) Load a test case (or input) with the highest priority from a disk to a memory buffer.
(2) Mutate the buffered input by randomly modifying certain bytes in the input or appending random bytes to the end of the input.
(3) Launch the target application with the newly generated input, and monitor its runtime execution.
(4) If the test case is interesting, save the input to the disk as a new test case for further mutation in successive runs (*e.g.*, explored a new branch).
(5) Repeat step (1) for another fuzzing iteration.

When multiple instances of fuzzers run in parallel, every instance performs an additional *syncing phase* for exchanging useful test cases among instances. After executing a certain number of fuzzing loops, a fuzzer periodically checks any new test cases generated by other fuzzers and re-executes some of them to decide whether they are interesting to the fuzzer itself, meaning that they cover any execution path or basic block that the fuzzer has yet to discover. These interesting ones are saved in the private corpus directory of the fuzzer.

2.2 Design of AFL

`AFL` is the state-of-the-art fuzzer, which discovers numerous security-crucial bugs of non-trivial, real-world software. We now focus on explaining the concrete design of `AFL` and its design considerations on performance and scalability for multi-core systems. We illustrate its overall design and workflow in Figure 1.

Mutating inputs (❶). `AFL` uses an evolutionary coverage-based mutation technique to generate test cases for discovering new execution paths of the target application [42]. In `AFL`, an execution path is represented as a sequence of taken branches (*i.e.*, a coverage bitmap) in the target instance for a given input. To track whether a branch is taken, `AFL` instruments every conditional branch and function entry of the target application at the time of compilation.

Launching the target application (❷). Traditional fuzzers call `fork()` followed by `execve()` to launch an instance of the target application. This process occurs in every fuzzing loop to deliver a new input to the target application. It is not only time consuming, but also a non-scalable operation. Previous research shows that the majority of fuzzing execution explores only the shallow part of the code and terminates quickly (*e.g.*, because of invalid input format), which results in frequent executions for the input test cases. Thus, the cost of `fork()` and `execve()` dominates the cost of fuzzing [7, 34, 36]. To mitigate this cost, `AFL` introduced a fork server, which is similar to a Zygote process in Android [39] that eliminates the heavyweight `execve()` system call. After instantiating a target application, the fork server waits for a starting signal sent over the pipe from the `AFL` instance. Upon receiving the request, it first clones the already-loaded program using `fork()` and the child process continues the execution of the original target code immediately from the entry point (*i.e.*, `main`) with a given input generated for the current fuzzing loop. The parent process waits for the termination of its child, and then informs the `AFL` process. The `AFL` process collects the branch coverage of the past execution, and maintains the test input if it is interesting.

Bookkeeping results (❸, ❹) The fork server also initializes a shared memory (also known as tracing bitmap) between the `AFL` instance and the target application. The instance records all the coverage information during the execution and writes it to the shared tracing bitmap, which summarizes the branch coverage of the past execution.

Fuzzing in parallel (❻). `AFL` also supports parallel fuzzing to completely utilize resources available on a multi-core machine and expedite the fuzzing process. In this case, each `AFL` instance independently executes without explicit contention among themselves (*i.e.*, embarrassingly parallel). From the perspective of the design of `AFL`, the fuzzing operation should linearly scale with increasing core count. Moreover, to avoid apparent contention on file system accesses, each `AFL` instance works in its private working directory for test cases. At the end of a fuzzing loop, the `AFL` instance scans the output directories of other instances to learn their test cases, called the *syncing phase*. For each collaborating neighbor, it keeps a test case identifier, which indicates the last test case it has checked. It figures out all the test cases that have an identifier larger than the reserved one, and re-executes them one by one. If a test case covers a new path that has not been discovered by the instance itself, the test case is copied to its own directory for further mutation.

2.3 Perils to Scalable Fuzzing

During a fuzzing loop, fuzzers utilize several OS primitives such as `fork()` and file operations, as described in §2.1 and §2.2. Unfortunately, when scaling fuzzers to run multiple instances in parallel, these primitives start to become performance bottlenecks, resulting in a worse end-to-end performance than that of a single fuzzing instance. We now give an in-depth detail of the potential system bottlenecks in each phase of the fuzzing process.

Cloning a target application. Every fuzzing execution requires a fresh instance of a target application to test a generated input. Most existing fuzzers use the `fork()` system call to quickly clone the target application, *i.e.*, for each fuzzing run, the parent process clones a new target instance that starts in an identical process state (*e.g.*, virtual memory space, files, sockets, and privileges).

Bottlenecks. From the performance aspect, there are two problems with the `fork()` system call. First, an OS repeatedly performs redundant operations in `fork()` that are neither used nor necessary for executing a target application during the fuzzing loop. Second, concurrent spawning of processes by using the fork() system call is not scalable because of various contentions (*e.g.*, spinlocks) and standard requirements (*e.g.*, PID should be assigned in an incremental order in POSIX). These operations are required for a general-purpose fork() in an OS, but significantly deter the scalability of fuzzing. For example, fork() needs to update the reverse mapping of a physical page for swapping under memory contention, which is a well-known scalability bottleneck in the current Linux kernel [8, 9, 11]. Moreover, fork() stresses the global memory allocator for allocating various metadata required for a new task, needs to set up security and auditing information, and has to add the task to the scheduler queue, which are all known to be non-scalable with increasing core count. Hence, none of the above operations are necessary and important for the purpose of fuzzing.

Creating a mutated test case. Each cloned process runs with a different, freshly mutated test case, to discover new paths and crashes. To support a variety of applications, existing fuzzers store a test case as a standard file and pass it to the target application, either as a command line argument or via standard input. At the end of a fuzzing loop, fuzzers store interesting test cases on the disk and fetch them later to generate mutated test cases for the next run. The number of test cases stored in disk increases monotonically until fuzzers terminate because any saved test case can be evolved again by mutation in a later fuzzing loop. Therefore, the most common practice is to "keep the test cases as small as possible" (*e.g.*, 1 KB is ideal for AFL [43]) because this minimizes file I/O as well as the search space for mutation, which keeps typical test cases to merely hundreds of bytes. At every run, fuzzers heavily interact with the file system to manage test cases.

Bottlenecks. Typical file system operations that fuzzers rely on are, namely, open/creat (for generating the mutated test case), write (for flushing interesting test cases), and read (for loading test cases) of small files in each fuzzing loop, importantly in parallel. Two benchmarks in FxMark [32] can be used to explain in detail the scalability of the fuzzer: MWCL (*i.e.*, creating a file in a private directory) and DWOL (*i.e.*, writing a small file in a private directory). More specifically, the process of creating and writing small files heavily modifies the file system metadata (*e.g.*, allocating inode for file creating and disk blocks for file writing), which is a critical section in most file system implementations, and not scalable [32].

Syncing test cases each other. A fuzzer running in parallel can expedite the search space exploration by sharing useful test cases with other fuzzers at the syncing phase. In this phase, each fuzzer checks the test cases of other fuzzer by iterating their test case directories. For example, the file name of a test case for an AFL instance starts with a prefix recording a sequence number as its identifier, which denotes that a test case starting with a greater sequence number was created later in fuzzing runs. While syncing, an AFL instance scans the directory of its neighbors and decides whether or not a test case is synced by that prefix. Then, it re-executes the obtained test case to get its tracing bitmap and decides whether the test case is useful.

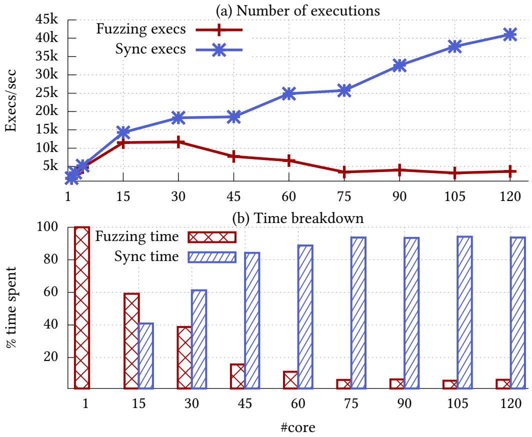

Figure 2: Multi-core scalability of AFL for libjpeg library. Figure 2(a) shows the numbers of fuzzing executions (*i.e.*, testing newly mutated test cases) and sync executions (*i.e.*, evaluating test cases of other fuzzers). Figure 2(b) shows the execution time breakdown for fuzzing and syncing executions. Even though fuzzing is an embarrassingly-parallel workload without dependencies among fuzzers, the scalability of AFL is *surprisingly poor*. The number of executions of AFL saturates at 15 cores, and starts degrading from 30 cores onward, and completely collapses at 120 cores. The reason for scalability collapse by around 24× in total is because of 1) the inherent design limitation of the syncing phase (2× slowdown) of AFL; 2) the fork() system call overhead (6× slowdown); and 3) the file system overhead for opening and closing small files (2× slowdown).

Bottlenecks. Scanning directories at the syncing phase is not scalable for the following reasons: First, the number of directory enumeration operations to discover new, unsynced test cases increases non-linearly with more fuzzers, which results in a longer syncing phase. For instance, each fuzzer will take $O(f \times t)$, where f is the number of fuzzers and t is the number of test cases in a test case directory. Second, directory enumeration severely interferes with creating a new test case because a directory write operation (*i.e.*, creating a file) and a read operation (*i.e.*, enumerating files) cannot be performed concurrently [32].

2.4 Scalability of AFL

To evaluate the impact of these bottlenecks, we ran AFL, a state-of-the-art fuzzer, for fuzzing the libjpeg library, JPEG decompressor, by varying the number of cores from 1 to 120 (see the environment details in §6). We used the input corpus (*i.e.*, seed inputs) provided by AFL [40]. Figure 2 presents the number of executions for fuzzing the libjpeg library, which shows that the scalability of AFL is surprisingly poor. The number of executions saturates at 15 cores and completely collapses at 120 cores. Considering that a typical server in a data center is a two- or four-socket machine with 32 or 64 cores, the current state-of-the-art fuzzers cannot fully exploit the typical servers.

Such poor scalability is counter-intuitive because fuzzing is an embarrassingly-parallel workload without any dependency among

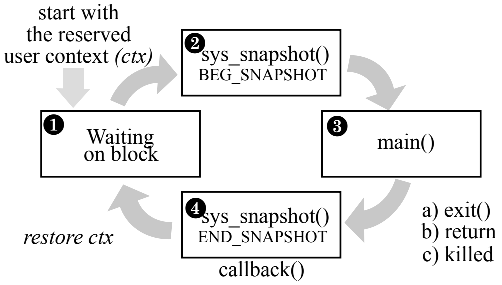

Figure 3: Simplified use of snapshotting via the `snapshot()` system call. A process (*e.g.*, fork server in `AFL`) starts by storing user context using `sigsetjmp()` system call and then waits for a "go" signal to start its execution ❶. Once it gets the signal, it prepares for execution (*e.g.*, setting up command line arguments) and creates a snapshot ❷, then starts executing the `main()` ❸. If the process terminates for any reason (*e.g.*, `exit()`, `SEGFAULT`), a callback function, `callback()`, which is registered at ❷, is called. It can be used for any purpose (*e.g.*, bookkeeping trace bits). By restoring the snapshot ❹, the kernel restores the memory and OS resources (*e.g.*, file and socket descriptors) to their original state ❷. Finally, program execution is reverted to the initial location ❶ using `siglongjmp()`.

fuzzers. The performance breakdown in Figure 2(b) shows the evidence that with increasing core count, the actual time spent on mutating and fuzzing new test cases decreases, whereas increasing time is spent on the syncing phase. One important point is that all synced test cases from other fuzzers have already been executed and re-executing them is meaningless considering the overall progress of the exploration of the target application. In addition, Figure 2(a) indicates that starting from 45 cores, the OS kernel becomes the main scalability bottleneck, in which most of the time, each fuzzing instance first suffers from the file system contention, and also the non-scalable `fork()` system call which contributes to a total 24× overhead due to the aforementioned scalability bottlenecks in both OS as well as the inherent design flaws of fuzzers.

Summary. Fuzzing looks embarrassingly parallel, but the design choices of both the fuzzer itself and the OS introduce performance and scalability bottlenecks in non-trivial ways, which require performance engineering in underneath layers to completely exploit the full potential of the hardware.

3 OPERATING PRIMITIVES

We now present the design of our three scalable operating primitives for fuzzing: a new `snapshot()` system call to replace the heavyweight `fork()` system call (§3.1), file system service specialized for optimizing small file operations in fuzzing (§3.2), and a shared in-memory test-case log for efficient, scalable syncing (§3.3).

3.1 Snapshot System Call

As we discussed in §2.3, fuzzers rely on `fork()` to take a snapshot of the target application and easily catch its crash. However, the general-purpose `fork()` is heavyweight for fuzzing: this includes a lot of unnecessary features such as creating the reverse mapping of the child's physical pages for swapping, which is a well-known performance bottleneck in Linux [8, 9, 11]. Nevertheless, by treating fuzzer as a first-class citizen in an OS, these known contentions can be either alleviated or completely avoided without compromising the correctness of the execution of a target process in fuzzing.

We propose a new system call, `snapshot()`, which is a lightweight, scalable `fork()` substitute for fuzzing. Figure 3 illustrates a simplified example of how to use the `snapshot()` system call. `snapshot()` creates a snapshot of a process (*e.g.*, its memory and OS resources such as file and socket descriptors). After that, the process can continue its execution. Upon request, `snapshot()` reverts the status of the process to the snapshotted state. Its prototype is as follows:

```
int snapshot(unsigned long cmd, unsigned long callback,
             struct iovec *shared_addr)
```

`cmd` is either `BEG_SNAPSHOT` for snapshotting or `END_SNAPSHOT` for reverting. At `BEG_SNAPSHOT`, a user can register a callback function, `callback`, and a vector of shared address ranges, `shared_addr` that OS should keep intact. For instance, in the case of fuzzing, `shared_addr` indicates the trace bitmap shared between a fuzzer instance and the target process (check §4.2 for its practical use in `AFL`). When a snapshotting process terminates, the OS will call the registered callback function. Currently, we do not support nested snapshotting. `snapshot()` is more lightweight and more scalable than `fork()`. It even has better performance than `pthread_create()` because it does not need to allocate and free thread stack, which is required for `pthread_create()`. The unnecessary operations in `fork()` and `pthread_create()` eventually incur costly TLB shootdowns, scheduler invocations, memory allocations, auditing, and security related modifications (see Figure 8).

In the rest of this section, we describe the details of `snapshot()` especially in the context of fuzzing. We divide a complete fuzzing run into three phases, and describe how `snapshot()` cooperates with the kernel and the target process at different phases.

3.1.1 Before Fuzzing: Process Snapshotting. For explanation, we assume that the fuzzer we deal with applies the basic design of `AFL`'s fork server. More specifically, we launch the target application in the beginning. The application is instrumented with a prologue, which keeps waiting for a signal sent from the fuzzer instance. Once a request from the fuzzer instance is received, the application invokes `fork()`. Then, the child process executes its actual main function while its parent process is waiting for the child to terminate.

In our design, before performing any fuzzing run, the target process first needs to save the user space execution context *ctx*, or specifically, all of the current register values and signal masks using `sigsetjmp()`. Then it goes into a loop and keeps waiting for a new fuzzing request from the fuzzer instance. On receiving a request, the process uses `snapshot()` to reserve its kernel context (`BEG_SNAPSHOT`) and registers a callback function: `callback()`. This user callback function acts as an epilogue of every fuzzing execution, which we describe later in more detail (§3.1.3). By invoking it with the `BEG_SNAPSHOT` command argument, the kernel operates on its data structures as follows:

- **Virtual Memory Area (VMA).** `snapshot()` iterates the virtual memory areas (*i.e.*, `vma`s) of the process and temporarily stores the start and end address of every `vma`.
- **Page.** `snapshot()` maintains a set of pages that belong to a writable `vma` because it is possible that the target application may modify these writable pages, which the kernel should

revert to the original state when the application terminates. To track these writable pages and maintain their original memory status, we use the copy-on-write (CoW) technique. We change the permission of writable pages to read-only by updating their corresponding page table entries (PTE) and flushing TLB to maintain the consistency. Thus, any write on these pages incurs a page fault, which the page-fault handler captures and handles. Our approach includes an optimization: We do not change the permission of mapped writable virtual address for which the kernel is yet to allocate a physical page because memory access on those pages will always incur a page fault.

- **brk.** A process's `brk` defines the end of its data segment, which indicates the valid range of its heap region in Linux. Thus it influences the results of heap allocations during the execution of the process. `snapshot()` saves the `brk` value of the current process.
- **File descriptor.** At the end of a snapshotted process, the kernel closes file descriptors that are opened after snapshot but revert the status of file descriptors that were already opened before snapshot. `snapshot()` saves the status of open file descriptors by checking the file descriptor table and bitmap (*i.e.*, `open_fds`).

Besides modifying these data structures, `snapshot()` saves the registered callback function. When the snapshotted process is about to terminate in the middle of a fuzzing run (*e.g.*, calling `exit()`), `snapshot()` can safely redirect the control flow to `callback()` function in the user space.

3.1.2 During Fuzzing: Demanding Page Copy. The target process continues to execute its real main function with the runtime arguments and environment variables after returning from `snapshot()`. When the target application is running, each memory write (at the page boundary) triggers a page fault because of our modification of the page permission to read-only, as described in §3.1.1, for any writable page. In our modified page-fault handler, we first check whether the fault address is in a snapshotted page originally. If that is the case, then a copy of the page data is modified and linked to its corresponding PTE with additional write permission. For the unallocated pages with no corresponding physical pages, we just allocate new pages and update the corresponding PTE with additional write permission. Lastly, we flush TLB entries to maintain the consistency.

3.1.3 After Fuzzing: Snapshot Recovering. Before terminating a snapshotted process, we call the registered `callback()`. In the case of normal termination, *i.e.*, returning from `main()`, it first informs the exit status (0 in this case) to its parent process, which is the controlling fuzzer instance. To deal with the situation where the target process calls `exit()` to terminate, we modify the entry function of `sys_exit_group` and check whether the process is snapshotted. If so, it calls the registered callback function in the user space. On the other hand, if the target process times out or crashes in the middle of the execution, it will receive the corresponding signal such as `SIGSEGV` or `SIGALARM` which terminates the victim process by default. To inform the abnormal status to the parent process and avoid re-creating the target process, our instrumented prologue registers a particular handler for every crucial signal at the very beginning. The handler calls `callback()` with the corresponding exit status (*e.g.*, 139 for the process, which has segmentation fault).

After calling the registered callback function, the process invokes `snapshot()` with `END_SNAPSHOT` to revert to the original snapshotted state. Then the reverted process restores the saved user-space context `ctx` using `siglongjmp()`, which directs it to start waiting for another fuzzing run. We describe the detailed procedure of `snapshot()` for the `END_SNAPSHOT` command, which involves four clean-up steps as follows:

- **Recovering copied pages**. `snapshot()` recovers the pages that have a modified copy of the original one; it also deallocates the allocated physical memory, reverts corresponding PTE, and flushes the corresponding TLB entries.
- **Adjusting memory layout**. `snapshot()` iterates the VMAs of the target process again and unmaps all of the newly mapped virtual memory areas.
- **Recovering brk**. The `brk` value of a process affects the heap allocations and it is restored to the saved `brk` value.
- **Closing opened file descriptors**. By comparing the current file descriptor bitmap with the one saved before the past fuzzing run, `snapshot()` determines the opened file descriptors and closes them.

Compared with `fork()`, `snapshot()` saves a great amount of time spent on copying numerous kernel data structures (*e.g.*, file descriptors, memory descriptor, signals, and namespaces). Moreover, it also avoids setting up a new copy of security and auditing structures and allocating a new stack area for the snapshotted process. As a result, `snapshot()` does not stress the kernel memory allocator and `cgroup` module. Moreover, `snapshot()` also removes the scheduling cost, which involves adding and removing a new or exiting process from the run queue of the scheduler, thereby eliminating the contention from the run queue as well as the re-scheduling interrupts. In summary, `snapshot()` plays the same role as `fork()`-ing for fuzzing but in a much more lightweight and scalable fashion.

3.2 Dual File System Service

As we discussed in §2.3, mutating test cases actually involves small file operations, including `creat`, `write` and `read`, that are not scalable in any existing file system: the root causes vary between file systems, but examples include journalling, lock contention in the implementation, or more severely, in the common VFS layer (*e.g.*, the block allocator) [32].

We introduce our second operating primitive, dual file system service, to provide efficient and scalable file operations for fuzzing. The key idea of our approach is to exploit the fact that neither a fuzzer instance nor the target instances require such a strong consistency model. Only the fuzzer instance requires consistency for serializing mutation states to the persistent storage for durability. Upon unexpected failures (*e.g.*, power or system crashes), the certain loss of test cases is expected, but a fuzzer can always reproduce them within an acceptable period of time. The new file system service provides a two-level tiering of file systems: a memory file system (*e.g.*, tmpfs) seamlessly to the fuzzer instance and target processes for performance, and a disk file system (*e.g.*, `ext4`) to the fuzzer instance for capacity and durability to store and maintain crashes, inputs and mutation states.

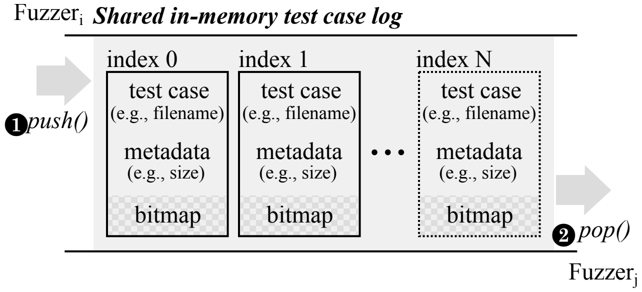

Figure 4: An illustration of shared in-memory test-case log to efficiently share test case information (*e.g.*, file name and trace bitmap) among fuzzers. By sharing test case information, fuzzers can judge if a test case created by other fuzzers is useful or not without re-executing the test case or re-enumerating the test case directory. Conceptually, a test case log is a fixed-size circular log supporting `push()` and `pop()` operations. Unlike conventional circular log, however, an element is removed from the log once all fuzzers perform `pop()` operations.

Our dual file system service creates a separate instance of a memory file system as the private working directory for every launched fuzzer instance. This approach completely avoids the contention of accessing test cases simultaneously under the same folder. In addition, it also gives the illusion to fuzzers that only the memory file system is available so that fuzzers create and read test cases in the memory file system. To give such an illusion while providing large capacity beyond memory and durability, our test case flusher, the core component of the file system service, periodically checks whether the memory usage of the memory file system exceeds the pre-defined threshold. If so, it moves test cases in the memory file system to the disk file system. These test case files are replaced with the symbolic links that point to their corresponding copies on the disk file system. We choose a victim file to be moved based on its age (*i.e.*, we always choose the oldest file). This age-based eviction policy works well in fuzzing because the test cases generated earlier are more likely to have lower coverage and are less likely to be re-read by the fuzzer instance for mutation. Both the threshold (h) and the proportion of the oldest test cases to be moved to the disk (α) can be configured when launching the file system service daemon.

Our approach defers the durability guarantee until test cases are moved to the disk file system so as to support eventual durability. However, in the context of fuzzing, such deferring is fine. Comparing that relying on a single disk file system, there are two different cases: 1) we can loose files on the memory file system upon system crash and 2) there is a very small window in which a fuzzer cannot see the moved file, when a file is moved but its symbolic link is not yet created. However, those cases are completely fine in fuzzing because they do not affect the correctness of fuzzing but makes fuzzers re-generate or re-test such test cases.

3.3 Shared In-memory Test-Case Log

In the case of parallel fuzzing, a fuzzer leverages the test cases generated by its neighbors in the syncing phase. However, as mentioned earlier, the current design of fuzzers requires expensive directory enumeration to find new test cases and re-execute them to get trace information. Thus, with many fuzzing instances, the syncing phase incurs a lot of unnecessary re-execution of test cases and imposes an overhead on the OS kernel.

We introduce a shared in-memory test case log for real collaborative fuzzing without any overhead. The test case log is a fixed-size, in-memory circular log (see Figure 4), which helps fuzzers efficiently manage and share generated test cases. Each test case log is created and controlled by a master fuzzer instance. Meanwhile any other instances can access the log as a slave. Each element of the log stores the information of a test case, including its filename, size, hash value and tracing bitmap. Like conventional circular queues, our test case log maintains `HEAD`. Only the master is allowed to push a new element into the log on `HEAD`. Note that, unlike conventional circular queues, any slave which attaches to the log can perform `pop()` to get the oldest element. Each slave maintains its local `TAIL` ($\texttt{TAIL}_i$). When it invokes `pop()`, an element at $\texttt{TAIL}_i$ is popped. Once all the slaves pop the oldest element, we move the global `TAIL` forward by one. For scalability, the test case log is designed in a lock-free manner. In addition, even if the process runs indefinitely, our fixed-size design guarantees a bound in memory usage.

For fuzzer developers to leverage the test case log in practice, we design several interfaces listed in Table 1.

Each fuzzer instance creates its own test case log at the very beginning (`create_log()`). It also needs to attach to the test case log of all other fuzzers for test case sharing (`attach_log()`). During fuzzing, a fuzzer pushes the executed test case information to its test case log if the test case is interesting (`push_testcase()`) at the end of each fuzzing run; at the syncing phase, a fuzzer pops a test case from its neighbor (`pop_testcase()`) to examine whether the test case is useful or not. Note that each fuzzer always gets the first test case from the log, which it is yet to examine while avoiding the costly directory enumeration. More importantly, since the tracing bitmap (*i.e.*, `trace_bits` in `AFL`, and `__sancov_trace_pc_guard_8bit_counters` in `LibFuzzer`) of every generated test case is directly achieved, re-execution is not necessary. The log is eventually destroyed by the fuzzer (`close_log()`) when fuzzing ends.

4 SCALING STATE-OF-THE-ART FUZZERS

We claim that all the application fuzzers that follow the five-step fuzzing loop listed in §2 can benefit from at least one of our operating primitives. Table 2 concludes the applicability of these primitives on 10 known open-source fuzzers developed in recent years.

In this section, we first explains on how our three operating primitives can generally improve the performance of an application fuzzer in §4.1. Among these selected fuzzers, American Fuzzy Lop (`AFL`) and `LibFuzzer` are two representative ones which are widely used and successful in finding numerous bugs. They also serve as a foundation for many later research works and fuzzing projects ([7, 21, 22, 36]). Thus, we implement two working prototypes based on the Linux x86_64 platform by applying our new operating primitives to `AFL` and `LibFuzzer`. §4.2 and §4.3 present the related technical details in practice.

Method Call	Semantics
`create_log(int id, size_t tc_size)`	Create a shared in-memory test case log for the calling instance identified by *id*; *tc_size* indicates the size of the metadata of a test case
`attach_log(int id)`	Attach to the test case log belonging to the fuzzer instance *id*
`push_testcase(int id, testcase_t *tc)`	Push a newly generated test case *tc* into the log of the instance *id*
`pop_testcase(int id, testcase_t *tc)`	Fetch a test case from the test case log of the instance *id* into *tc*
`flush_log(int id)`	Flush out all the stale test cases from the instance *id* by force
`close_log(int id)`	Destroy the test case log owned by the instance *id*

Table 1: Shared in-memory test case log interface overview. Application fuzzer can leverage these interfaces to share test cases among running instances and achieve scalable collaborative fuzzing. A fuzzer instance invokes `push_testcase()` to save the metadata of its generated test case (*e.g.*, filename, tracing bitmap) into its test case log, and invokes `pop_testcase()` to reference the information of the test cases evolved by other fuzzer instances. Hence, directory enumeration and test case re-execution are no longer required.

Fuzzers	**Snapshot**	**Dual FS**	**In-memory log**
AFL	✓	✓	✓
AFLFast	✓	✓	✓
Driller	✓	✓	✓
LibFuzzer	-	✓	✓
Honggfuzz	-	✓	✓
VUzzer	✓	✓	✓
Choronzon	✓	✓	✓
IFuzzer	✓	✓	✓
jsfunfuzz	✓	✓	-
zzuf	✓	-	-

Table 2: It shows how three proposed operating primitives can benefit 10 chosen mainstream fuzzers including AFL [41], AFLFast [7], Driller [36], LibFuzzer [29], Honggfuzz [21], VUzzer [34], Choronzon [12], IFuzzer [38], jsfunfuzz [35], and zzuf [2], according to their design and implementation.

4.1 Overview

4.1.1 Snapshot System Call. First, the `snapshot()` system call can be applied as a perfect substitute for `fork()` in the fork server of `AFL` (see §4.2.1 for more details).

Quite a number of existing fuzzers such as Choronzon [12], VUzzer [34], IFuzzer [38], jsfunfuzz [35] and zzuf [2] launch a target instance by invoking `fork()` and `execve()` (*i.e.*, `subprocess.Popen()` in Python). The combination of these two operations are not scalable and dominate the time cost of fuzzing an application (see §2.2). To apply the `snapshot()` system call to these fuzzers, we can instrument the target with the prologue which connects to the fuzzer and sets the restore point, and the epilogue which is the callback function even without the source of the target. The fuzzers can thereby leverage the `snapshot()` system call in the same way as `AFL`.

`LibFuzzer` creates a running thread for each target instance. Its in-process model avoids the contention brought by `fork()`. Therefore, the `snapshot()` system call becomes useless in such a case.

4.1.2 Dual File System Service. In most cases, a fuzzer instance always needs to create and read test cases under a particular directory on a file system. And it is not scalable for multiple instances to perform these operations on a disk file system. Thus all the application fuzzers can benefit from our two-tiered file system service. Considering the fact that the working directory of a fuzzer process is configurable for most fuzzers, applying the file system service to them is straightforward. `LibFuzzer` has multiple fuzzer instances in one process. Consequently, assigning a separate directory for each instance requires extra modification (see §4.3).

As a special case, zzuf does not generate input before launching the target. It intercepts file and network operations and directly mutates the program's input with a given seed for reproduction. Therefore, our two-tiered file system service does not bring much benefit to zzuf by default.

4.1.3 Shared In-memory Test Case Log. Feedback-driven fuzzers like `AFL`, `LibFuzzer` and Choronzon support parallel fuzzing by sharing test cases among all the running instances. A fuzzer instance periodically checks the test cases generated by its neighbors and archives the interesting ones for itself. The performance bottlenecks of this syncing phase mainly come from directory enumeration and test case re-execution (see §2.3), which can be well solved by applying the shared in-memory test case log.

Some evolutionary fuzzers such as VUzzer and IFuzzer do not natively support collaborative fuzzing. The fuzzer instance solely evolves without a syncing phase in its fuzzing loop. By applying our shared in-memory test case log, these fuzzers can be extended to support real multi-tasking on multiple cores without contention.

More general fuzzers like zzuf and jsfunfuzz are not driven by any metric or feedback to evolve their test input at runtime. A test case is considered interesting only when it triggers abnormal behaviors (*e.g.*, memory violations, timeout, assertion failures). For these fuzzers, online test case sharing is not effective because of the limited size of their generated corpus.

Derivatives. Driller [36] and AFLFast [7] are two `AFL`-based fuzzers proposed in recent years. Driller extends `AFL` with symbolic execution while AFLFast optimizes the test case scheduling algorithm applied in `AFL`. As these two fuzzers do not modify the controlling part of `AFL`, our primitives can also be applied to them in the same way as `AFL`. Similarly, Honggfuzz [21] is a derivative of `LibFuzzer`. It leverages hardware features to collect coverage information without modifying the working model of `LibFuzzer`. Thus, we can apply our operating primitives to it in the same way as `LibFuzzer`.

4.2 Scaling AFL

Each of our three operating primitives can benefit AFL in solving a corresponding performance bottleneck that AFL suffers from (see §2).

4.2.1 Snapshot Server. Applying the snapshot() system call to AFL is straightforward that does not require much engineering effort. We instrument a new snapshot server before the main() entry of a target application as a substitute of the old fork server.

The snapshot server first invokes sigsetjmp() to reserve the user space execution context, including the current register values and signal masks to a global area. Then, similar to the fork server, it waits for the starting signal from the controlling AFL instance.

Once the snapshot server receives a request, it invokes snapshot() with the command BEG_SNAPSHOT, the address of a callback function cleanup(), which is also instrumented into the target application. Moreover, the base address and size of the tracing bitmap are passed into the kernel through shared_addr. The memory updates to the bitmap, that occur during a fuzzing run, should not be reverted because AFL relies on the bitmap state to determine whether or not the previous test case is interesting.

The snapshot server directly calls the original main() with runtime arguments and environment variables. If the snapshotted process normally returns from main(), the instrumented callback function cleanup() will be invoked to inform the controlling AFL instance about the exit status and call snapshot() with END_SNAPSHOT to revert back to the snapshotted state.

In the instrumented assembly payload, we also register different signal handlers for various kill signals, which call cleanup() with their corresponding exit status. Note that AFL sends a SIGKILL to the spawned child process of the fork server if the current run is timed out. In our case, the AFL instance cannot kill the target instance. Thus we make AFL send a SIGUSR1 to the snapshot server instead when timeout occurs, which is also handled by redirecting the control flow to cleanup().

4.2.2 Removing File System Contention. AFL can be directly deployed on our two-level tiered file system without any modification. The first tier, which is a memory file system (*i.e.*, tmpfs), contains a private directory for each AFL instance. AFL only interacts with the memory file system to read and save test cases. The second tier is the disk-based file system, which is transparent to the running fuzzers. The dual file system service daemon periodically moves some set of oldest test cases (α) from the first tier of the AFL's private directory, including all the saved test cases that cause either crashes or hangs, to the second tier (disk-based file system) if the memory usage is higher than the pre-defined threshold value (h). Moreover, all generated test cases are eventually saved to the disk file system by the service daemon if a running AFL instance terminates.

4.2.3 Efficient Collaborative Syncing. Here we explain how to use our third operating primitive, shared in-memory test case log at the syncing stage of AFL which saves time that an instance wastes while iterating the directory and re-executing test cases obtained from its neighbors.

During initialization, each AFL instance is assigned a test case log that is shared among all the other fuzzers. And then it connects to the test case logs of all its neighbors. Note that we already know the number of running instances beforehand, which is a reasonable assumption for parallel fuzzing on multiple cores and is also a common practice in various concurrent applications such as databases, runtime systems etc. Each time a new interesting test case is added to the fuzzing log (*i.e.*, add_to_queue()), its file path, file size, and the complete trace bitmap (*i.e.*, trace_bits of 65,536 bytes by default) are combined as an element and saved into the shared logging sequence.

During the syncing stage (*i.e.*, sync_fuzzers()), a particular AFL instance pops out unsynced elements in the shared test case log from the other fuzzers. Then it directly references the saved trace bitmap of the unsynced elements to determine whether or not the corresponding test case is interesting. Later, the AFL instance sweeps the stale elements out of its test case log after they have been checked by all other fuzzers at the end of every syncing phase. Note that AFL tries to trim the saved test cases to simpler ones with the same path coverage, which means that the size of a test case can shrink during fuzzing. This results in the stale file information being saved in the test case log. However, an AFL instance always makes a copy of the interesting test case synced from other fuzzers in its own output directory. Thus, we rely on the return value of the final write() to determine the up-to-date size of a test case from another AFL instance.

4.3 Scaling LibFuzzer

LibFuzzer [29] is a specialized form of coverage-guided, in-process fuzzer for fuzzing single-threaded libraries and applications. It is fragile and restrictive compared with the AFL. To fuzz a library, the programmer should link the LibFuzzer with the library and provide a specific fuzzing entry point, also called target function, for feeding the inputs to the library. At the time of fuzzing, LibFuzzer tracks the reachability of the code, that is executed with either the seed corpus data (input test cases) or the mutated results based on the generated corpus data (mutated test cases). For code coverage, LibFuzzer relies on the SanitizerCoverage [37] instrumentation of the LLVM that catches various memory-related errors and keeps track of memory accesses.

At a high level, each LibFuzzer instance maintains a trace program bitmap, which it updates after obtaining the coverage information provided by the instrumentation engine after every run. Moreover, it also maintains a local, in-memory hash table that stores the test cases LibFuzzer thinks are interesting, The key of the hash table is SHA1 of the interesting test case it ran, which is also saved on the disk so that other LibFuzzer instances can use to make further forward progress. In particular, each LibFuzzer instance periodically scans the shared corpus directory to obtain any new test case that has a SHA1 value missing in the hash table. Currently, this is done at the granularity of a second (default) if an instances is unable to obtain new coverage after the mutation.

Launching the target application. LibFuzzer works by invoking an entry function that wraps the target code a developer wants to test. LibFuzzer also provides the option of running a multi-threaded version in which the master process creates a pre-defined number N of threads, where N is given by the user. Then LibFuzzer keeps invoking itself within a single threaded mode for N times.

Bookkeeping results. After invocation, each LibFuzzer instance first creates a local log file, which logs information about the coverage, test case output, and applied mutating strategies, as well as the current syncing states. After creating the log, a LibFuzzer instance L reads the corpus directory, if provided, and adds all the input test cases into its hash table and then starts mutating each of them. L fuzzes a library by first resetting the instrumentation bitmap and then calling the target function with the generated input buffer and its size. L updates its trace bitmap with the help of the instrumentation engine and then calculates the coverage as soon as the target function returns. If the coverage increases, L first inserts the new input with its SHA1 into the hash table, and later saves the input to the shared corpus directory on the disk.

Fuzzing in parallel. After several unsuccessful mutations, L periodically peeks through the shared corpus directory to obtain new test cases saved by other running instances. To obtain a new test case from the shared directory, L first traverses the directory and only reads those files whose time is greater than the time when invoking the directory read operation. After reading input test cases, L calculates the SHA1 value and only re-executes those test cases whose SHA1 value is missing in the hash table. Later, L saves the newly executed test case to its own corpus after deciding whether it was interesting and then updates the overall coverage L has achieved. L repeats this whole process for all the saved test cases in the shared directory and then it moves to the mutating phase to create new test cases that can further improve the coverage of the program.

Bottlenecks. LibFuzzer's in-process model brings restrictions on the execution of the target but overcomes the issue of forking (unlike AFL). Thus, the snapshot() system call makes no improvement on its performance. However, it still suffers from two design issues which can be solved by the other two primitives: 1) The current design of collaborative syncing is vague, as each instance re-executes the interesting test cases even if other instances have already gained their related features. It is not wise for different LibFuzzer instances to sync through the file system because they already have a copy of the generated corpus in their own memory, which can be easily shared among themselves. 2) Moreover, the syncing phase of LibFuzzer suffers from file system overhead because it does read and write to the same shared-corpus directory, which is not scalable, as shown by prior research [32]. The file system induces a lot of overhead in order to maintain the consistency of the data, which is not entirely required for applications like fuzzers. We now describe our proposed changes to improve the scalability of LibFuzzer with increasing core count based on the new operating primitives.

4.3.1 *Efficient Collaborative Syncing.*

LibFuzzer already provides a notion of collaborative fuzzing by allowing any LibFuzzer instance to get the new test cases from other instances because all instances write their new test cases to a specific shared corpus directory. To remove the overhead of the redundant file system based operations, we use the shared in-memory test case log (see §3.3) by exposing the test case, its coverage bitmap, and the SHA1 of the test case. Moreover, each LibFuzzer instance (L) also maintains a local table of how many test cases it has popped out from the log of other instances. Thus, after finishing a syncing round, L increases the number of new test cases it has read from a collaborating instance in its local table. In addition, now the LibFuzzer does not re-execute the copied corpus since it can directly update the coverage bitmap, which is also shared among all collaborators. Hence, by merely utilizing a fraction of memory, we not only remove the redundant executions, but also improve the the file system overhead in terms of reading corpus data from the shared directory.

4.3.2 *Removing File System Contention.*

The shared in-memory test case log partially resolves the file system overhead by removing the directory traversal and file reading during the syncing stage. However, the current design of LibFuzzer still suffers from the contention of writing the interesting corpus data to the shared directory. In addition, it also maintains the log of each running instance in the root directory where the application is being fuzzed. Unfortunately, both of these designs are non-scalable. First, Linux holds a directory lock when creating a new file. To solve the contention, our two-tiered file system service creates a local directory on memory file system for each LibFuzzer instance to write both the interesting corpus data and its log. The service daemon copies the data from the first tier, memory file system, to the second tier file system (storage medium) in order to provide an eventual durability (see §3.2).

5 IMPLEMENTATION

We implemented and applied our design decisions to both AFL (version 2.40b) and LibFuzzer, a part of LLVM v4.0.0, on Linux v4.8.10.

5.1 Operating Primitives

We implemented a new x86_64 system call snapshot() for generic fuzzers to clone an instance of the target program. It supports snapshotting the memory and other system states of a running process and rolling a snapshotted process back to its original state. snapshot() has system call number 329 and its implementation involves 750 lines of code (LoC) introduced into the Linux kernel. snapshot() also requires the cooperation of the page fault handling process in the kernel in order to track memory updates during a fuzzing run. We also modified the exit and signal handling routine of a user process in the kernel, in order to prevent a snapshotted process from being accidentally terminated at runtime, and meanwhile ensure that a process can normally handle these signals in an expected way if it is not in snapshot. These changes involve around 100 LoC scattering at different places in the kernel source.

We also developed a library containing six interfaces of the shared in-memory test case log in around 100 LoC. In particular, the test case log is implemented as a POSIX shared memory object supported by /dev/shm utility in Linux.

Furthermore, we wrote a 100 LoC simple dual file system service daemon which can be launched by a fuzzer for a private working directory on partitioned tmpfs and periodical test case flushing (see §3.2).

5.2 Applications

We applied all of our three operating primitives to AFL, and only the second and third ones to LibFuzzer.

In order to make AFL use snapshot() instead of fork() to clone a new target instance, we rewrote AFL's fork server instrumented into the target application during compilation without modifying much of its original working mechanism. However, the timeout and other kill signals are particularly handled since the snapshot server runs within the same process of the target instance in our design. Whatever errors occur during a fuzzing run, the snapshot server must be maintained. Our new snapshot server (for 64-bit target) has around 350 LoC.

Moreover, we implemented a new function to replace the old one for syncing test cases from other fuzzer instances. It utilizes the shared in-memory test case log by linking the library and invoking related interfaces (see §3.3). An individual AFL instance updates its own test case log with the path, size and coverage information of the generated test cases. It also keeps track of the test case logs belonging to other AFL instances. The new syncing phase introduces around 50 LoC into AFL's code.

In LibFuzzer, similar to AFL, we can have a per-thread in-memory shared queue among the instantiated instances of LibFuzzer, which is individually updated by the instance, and a list table to keep tracks of how many corpus data an instance has read from each neighbors. We added around 200 LoC into LibFuzzer to implement our ideas.

For both fuzzers, we launch our two-tiered file system service daemon to have a separate directory on tmpfs for each fuzzer instance. The daemon provides the eventual durability, which allows us to resume previous fuzzing in the future.

6 EVALUATION

We evaluate our proposed design decisions by answering the following questions:

- What is the overall impact of our design choices on two classes of fuzzers (*i.e.*, AFL and LibFuzzer)? (§6.1, §6.2)
- How does each design choice help improve the scalability of fuzzers? (§6.3)

Experimental setup. We evaluate the scalability of two fuzzers—AFL and LibFuzzer—on an eight-socket, 120-core machine with Intel Xeon E7-8870 v2 processors. Even though it is not conventionally used at this moment, we use it for the purpose of research to extrapolate the performance characteristics of future, large multi-core machines. We choose libraries from Google's fuzzer test suite [24] to evaluate both AFL and LibFuzzer.

Since the fuzzer test suite already provides the wrapper function for fuzzing libraries with LibFuzzer, we wrote our own wrapper to fuzz these libraries with AFL. In particular, we changed the fuzzing entry for LibFuzzer into the main() function, which can be compiled into an executable. LibFuzzer assumes that the input is stored in a buffer with a certain size. To test AFL by reusing the test suites for LibFuzzer, our wrapper opens the input test case whose path is indicated by the argument of the program, and then loads the test case into a memory buffer. Moreover, we observe that AFL also allows the target application to directly receive input data from stdin and performs different file operations from ones used for delivering input through file. To evaluate this interactive fuzzing mode, we also fuzzed djpeg built on top of IJG's libjpeg, which accepts standard input. Each fuzzing test lasts for 5 minutes for both AFL and LibFuzzer, and for every used core, we bind a single fuzzer instance to it.

In the experiment, the file system service checks the usage of the in-memory file system (*i.e.*, tmpfs) every 30 seconds in the experiment. If the usage exceeds 80%, then the service daemon flushes the oldest 25% test cases to the disk file system (*i.e.*, SSD) (see §4.2.2).

Performance metric. We aim at improving the overall performance of fuzzers, regardless of their fuzzing strategies (*e.g.*, mutation policies). To demonstrate such performance benefits, we use *executions per second* to evaluate a fuzzer, which is more direct and scientific. Using the notation of path coverage or bugs found to show performance improvement tends to be subjective as fuzzers saturate too quickly with a small corpus size in a laboratory environment [7].

6.1 AFL

Figure 5 presents the results of our optimized AFL version (*opt*—AFL$^{\text{OPT}}$) against the original version (*stock*—AFL). AFL$^{\text{OPT}}$ improves the fuzzing execution by 6.1 to 28.9× for nine tested libraries at 120 cores. Moreover, our techniques not only drastically decrease the average syncing time per core of AFL$^{\text{OPT}}$ by 41.7 to 99.8%, but also enable generating more new test cases by mutating the input corpus data. Furthermore, we simply provide an empty file as the seed input when fuzzing openssl-1.0.2d. Because of the serious performance bottleneck, timeout occurs when several AFL instances process the only input test case while AFL cannot proceed if there is no valid input test case for mutation. That explains why there is no experimental result for openssl-1.0.2d at 90, 105 and 120 cores.

Note that AFL$^{\text{OPT}}$ does not change the fuzzing algorithm of *stock*—AFL, but our operating primitives remove the contention either from the inherent design flaws of AFL or from the underlying OS. Moreover, other evolutionary techniques [7, 34] can benefit from our operating primitives to expedite the process of concurrently finding new bugs. Another interesting observation is that besides libpng, woff, boringssl, and c-ares, the other five applications show almost linear scalability because these applications do not frequently invoke dynamic memory allocations and thus are free from kernel memory allocation overhead such as cgroup accounting or even page cache maintenance by the zone allocator of the kernel, which is currently the hot spot in the kernel.

We also found that some applications (harfbuzz, libpng, woff, libxml and c-ares) suffer from the known open()/close() system call overhead while reading input test cases [32]. We further remove this overhead by employing a multi-partition file system in which each slave saves the data in the own partition as described in §3.2. This approach further improves the scalability of these libraries by 1.2 to 2.3×.

In summary, our AFL$^{\text{OPT}}$ version is the most current lightweight approach to fuzz any application without realizing on any specialized fuzzers that limit the scope of fuzzing, and finding new bugs efficiently.

6.2 LibFuzzer

Figure 6 presents the results of fuzzing some of the libraries with LibFuzzer. We choose a set of libraries because LibFuzzer inherently suffers from its threading design limitation, which exits if any error occurs or if the program encounters any signals. We observe that LibFuzzer improves applications' fuzzing execution

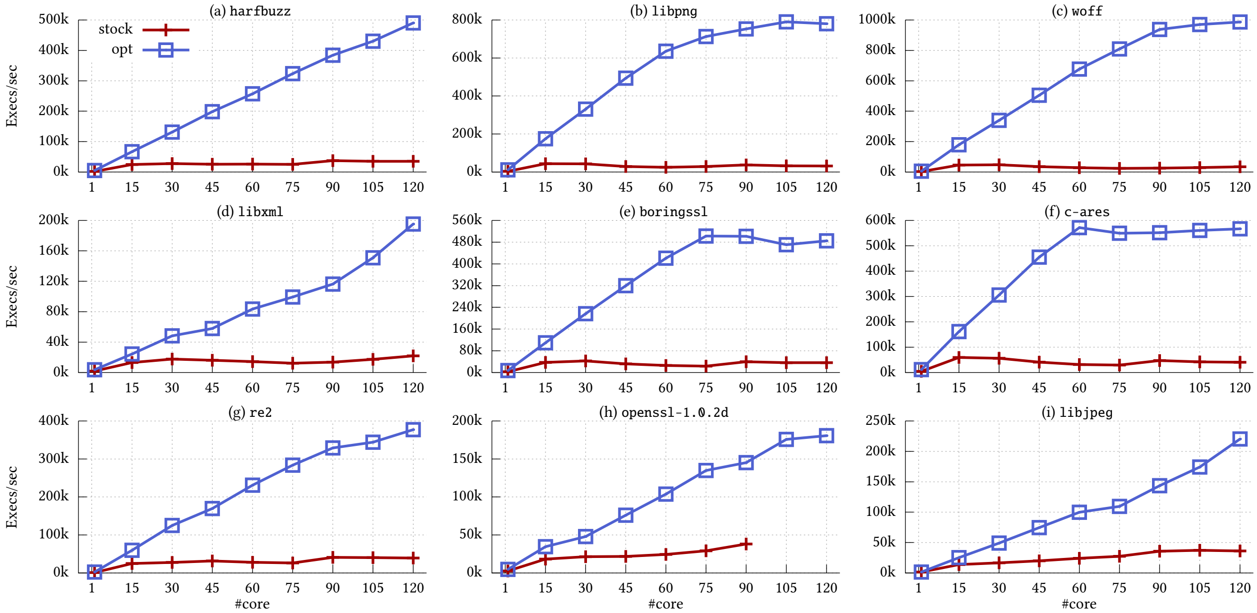

Figure 5: The execution numbers per second of the original and our optimized version of AFL by fuzzing Google's fuzzer test suite plus libjpeg on 1 to 120 cores. In the case of openssl-1.0.2d, from 90 cores, each fuzzing instance times out, which results in no further execution as there are no valid input test cases for mutations due to the severe performance bottlenecks in the stock version of AFL.

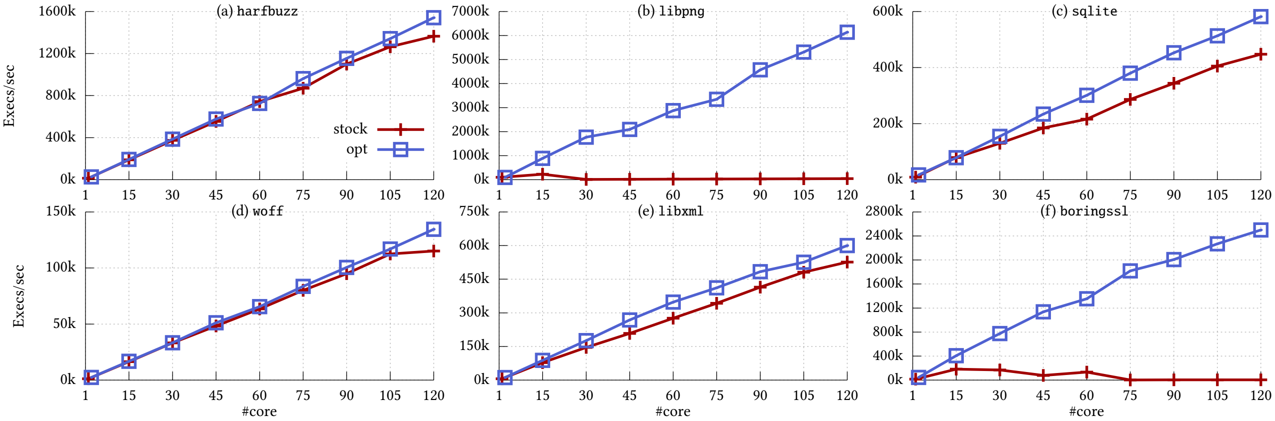

Figure 6: The execution numbers per second of the original and our optimized version of LibFuzzer by fuzzing Google's fuzzer test suite on 1 to 120 cores.

count from 1.1 – 735.7×. We measure at least two orders of magnitude in change with libpng and boringssl because both suffer from the inefficient syncing phase of the LibFuzzer, and we improve their scalability by 145.8× and 735.7× via the in-memory test case log (see §3.3). Another reason for such overhead is that neither of them can find a good test case, which results in periodic corpus directory enumeration to find new interesting test cases from collaborating LibFuzzer instances. While other libraries do not suffer from the poor test cases, their scalability is improved by 1.1 – 1.3× because they benefit from our shared in-memory test case log as well as the dual file system service.

6.3 Evaluating Design Choices

We now evaluate the effectiveness of each of our design decisions by evaluating some sets of benchmarks. We choose libpng for evaluation, as both Figure 5 and Figure 6 illustrate that it has the highest improvement over the version of fuzzers. We evaluate the scalability behavior of our optimized AFL on the tmpfs file system for the in-memory test case log (refer §3.3) and the snapshot() (§3.1) system call experiments. Later, we show the impact of the physical medium on AFL (§3.2).

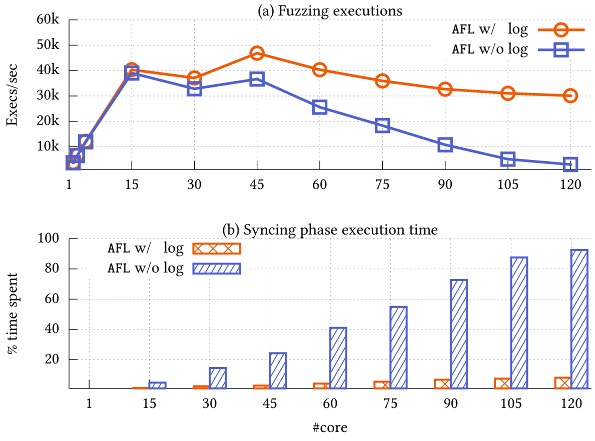

Figure 7: Evaluation of the shared in-memory test case log in terms of the number of executions and the time spent while fuzzing the `libpng` library. While Figure 7(a) shows the number of new executions, Figure 7(b) shows the percentage of time spent in the syncing phase.

6.3.1 Shared In-memory Test Case Log. We run *stock*—AFL with and without the shared in-memory test case log applied on the `libpng` library, and Figure 7 presents the experimental results. In terms of the new fuzzing executions that exclude the re-executions at the syncing stage, the in-memory shared logging speeds up AFL at roughly 13× at 120 cores. We can observe that the overall time spent on syncing by AFL linearly increases and around 90% of the total fuzzing time is used for syncing at the worst case. Note that the syncing stage does not contribute anything directly to the whole progress of the exploration of the target program. Without modifying the working mechanism AFL follows at the syncing stage, our new primitive successfully removes the performance bottleneck and the percentage of time spent at the syncing stage drops to at most 8.05%, which is totally acceptable.

6.3.2 Snapshotting. Even though the in-memory queue removes the overhead from the syncing phase and brings it down to 8.1%, we observe that the scalability of AFL is still saturated after 45 cores (Figure 7). The primary reason for such saturation is the `fork()` system call. Figure 8(a) shows the impact of replacing the `fork()` with the `snapshot()` system, which improves the scalability of the `libpng` fuzzing by 12.6× and now fuzzing is bottlenecked by the file operations (*i.e.*`open()`/`close()`). To further validate the necessity of our `snapshot()` system call, we create a micro benchmark to stress test the existing process creation APIs such as `fork()` and `pthread_create()`. The micro benchmark first spawns a per-core process that individually creates processes or threads using the aforementioned APIs, including the `snapshot()` system call, and then terminates immediately. Figure 8(b) presents the results for the process creation along with the number of fuzzing executions of the `libpng` library, which clearly shows that both `fork()` and `pthread_create()` do not scale beyond 15 cores and suffer from scalability collapse after 45 cores. On the other hand, `snapshot()` system call scales almost linearly with increasing core count and outperforms both of the process spawning APIs by 3004.5×. Moreover, `snapshot()` is considered a generic system call for fuzzers. The allowed number of snapshotted processes is sufficient to handle the executions that AFL can have concurrently in practice.

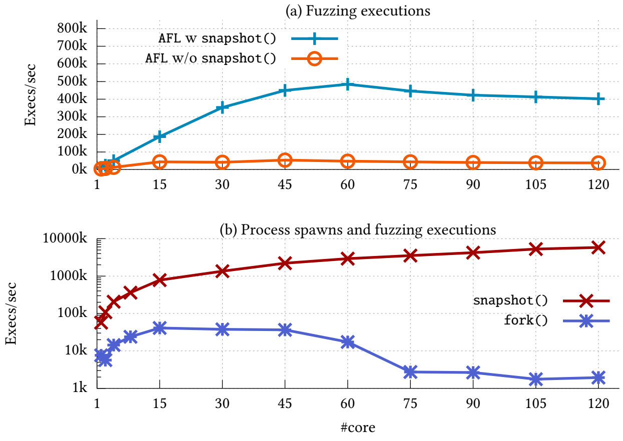

Figure 8: Evaluation of our `snapshot()` system call against `fork()` system call and `pthread_create()` function. Figure 8(a) shows the impact of `snapshot()` system call while fuzzing the `libpng` library. Figure 8(b) shows the scalability of all the primitives that are used to create a new process along with the optimized AFL case while fuzzing the `libpng` library.

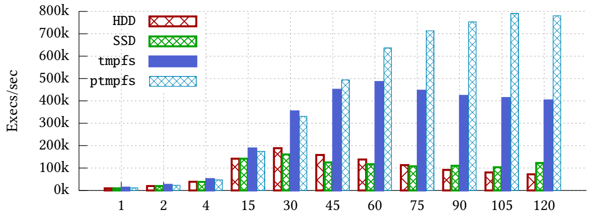

Figure 9: Impact of the file system on our optimized version of AFL for the `libpng` library.

6.3.3 File-system Overhead. Most existing fuzzers save interesting test cases as well as the crash information in a directory specified by a user. Figure 9 presents the impact of the physical medium while fuzzing the `libpng` library with our optimized AFL, which clearly illustrates that AFL does a lot of file system-specific operations and is affected by the physical medium as well as the file system overhead. For example, by changing the physical medium from HDD to SSD, the scalability of AFL improves by 1.7×, which further improves by §3.2 while switching from SSD to an in-memory file system (`tmpfs`). However, even with `tmpfs`, the most trimmed version of a file system, we observe that its performance is saturated at 60 cores, which happens because opening and closing files in a shared directory is not a scalable operation [32]. To mitigate this problem, we use a partitioned (`ptmpfs`) approach to partially mitigate the scalable bottleneck of the file system, which improves the performance by 1.9×, 6.3×, and 10.8× over simple `tmpfs`, SSD, and HDD, respectively. In summary, by using our partitioned approach, we improve the scalability of AFL by 24.3× over the stock version of AFL on 120 cores.

7 RELATED WORK

Our work is motivated by previous research on fuzzing techniques [7, 12, 13, 21, 23, 27, 29, 34, 36, 41], which aim to explore program paths wisely, and large-scale fuzzing [20, 22, 31], which aims to explore program paths faster using many, networked machines. Our goal in this work is improving the performance and scalability of fuzzing in a single, multi-core machine so our techniques can be orthogonally used with previous work. We also influenced by previous work on OS scalability [4, 5, 8–10, 16–18, 26, 32] and solutions to improve scalability [6, 15, 19, 28, 30, 33] but we analyzed OS scalability bottlenecks with interactions of fuzzers and proposed practical solutions for scalable, performant fuzzing.

Fuzzing. Existing fuzzing techniques strive to mutate input wisely for a better exploration of target programs and the earlier detection of bugs. For instance, feedback-driven fuzzers [12, 21, 23, 29, 34, 41] profile various runtime characteristics of past fuzzing runs and the profiled results directs the generation of the following inputs. More specifically, coverage-driven fuzzers [23, 29, 41] use past code coverage to determine whether or not a mutated input is interesting. Some fuzzers [12, 34] retrieve more advanced knowledge such as code- and data- flow feature, or various metrics to catalyze the evolution of input. Honggfuzz [21] uses hardware features (*e.g.*, Intel PT, Intel BTS) as a more general solution to execution tracing than software instrumentation. Fuzzing is likely to stuck at particular branches with complex conditions for uncertain time due to its randomness. Existing research addresses this issue by either wisely scheduling the sequence of the test cases in the waiting queue [7] or combining fuzzing with symbolic execution [13, 27, 36]), which was originally proposed to solve sophisticated condition checks.

Note that our research on solving the performance bottleneck of fuzzing is orthogonal to the previous works mentioned above. What we propose are fuzzer-agnostic primitives from the operating system side to speed up fuzzing, especially with a number of fuzzer instances running concurrently.

Large-scale fuzzing. In recent years, serious vulnerabilities in modern software and OS exploited by attackers for profit are on a rapid increase. As a response, large companies and organizations expend a huge amount of hardware resources on automated fuzzing to discover bugs in their own products. For example, the fuzzing infrastructure *ClusterFuzz* [20] by Google consists of several hundred virtual machines running around 6,000 Chrome instances simultaneously. It also powers project *OSS-Fuzz* [22] to process trillions of test cases targeting open source software. Furthermore, Microsoft provides a cloud-based fuzzing service called *Project Springfield* [31] for developers to find security bugs in the software. Our proposed operating primitives can help boost the fuzzers deployed on large clusters of cloud servers with abundant hardware resources and thus save significant cost.

Process snapshot. Recent research works [6, 15, 28] propose several OS primitives based on process snapshot to provide a temporary and isolated execution context for running particular code flexibly. *lwC*s (light-weight contexts) [28] provides independent units of protection, privilege, and execution state within a process. The `lwCreate` call creates an in-process child *lwC* with an identical copy of the calling *lwC*'s states. Different from the `snapshot()` system call, the new *lwC* gets a private copy of per-thread register values, virtual memory, file descriptors, and credentials for isolation purpose. Shreds [15] provide an in-process execution context for a flexibly defined segment with private memory access. After the code enters a sensitive section, it is granted a private memory pool isolated from the virtual memory space while all the other process states remain the same. Wedge [6] is another similar system used for splitting complex applications into fine-grained and least-privilege compartments. Note that the goal of these works is to create a lightweight execution context for sandboxing. As a result, the new execution context possesses private memory space, credentials, and other system states that are isolated from the calling context. By contrast, the execution context before and after the `snapshot()` system call is completely the same for fuzzing purposes.

OS scalability. Researchers have been optimizing existing OSes [4, 8–10, 16, 17, 26] or completely rewriting them based on new design principles [5, 18]. Our design decisions for fuzzing are inspired by these prior works and concurrent programming in general. For instance, while Wickizer et al. [4] improved the performance of the Linux `fork()` in general, we resolve the issue by designing a lightweight process spawning API that is specific to applications like fuzzing. In addition, prior works have used in-memory file systems to hide the file system overhead; instead we use it in the form of two-level caching to provide a required file system interface as well as the memory bandwidth.

OS specialization. Prior research works [19, 30, 33] have also focused on removing the underlying overhead of OS in both the bare metal and cloud environments. Even though, our work on specializing OS for fuzzing ventures into a similar direction, it is still generic from an OS perspective compared with library OS, which has focused on rewriting the application for performance.

8 DISCUSSION AND FUTURE WORK

Applicable fuzzers. We only analyze and improve performance and scalability of the general application fuzzers that *natively* execute the target with *concrete* input values in round. Our operating primitives may bring less benefit to many other fuzzing tools which rely on symbolic execution, taint analysis or instruction emulation to find security bugs. OS kernel fuzzers are also out-of-scope. These fuzzers may suffer from different performance bottlenecks and require corresponding solutions.

Cross-platform implementation. We implemented the working prototype of our design choices on Linux platform. However, there is a greater demand on finding bugs in MacOS and Windows applications because of their popularity. We will port our implementation to these two platforms in the near future.

Scalable fuzzing on VM clusters. The scalability of fuzzers depends not just on the design of fuzzers. As the cloud provider starts adopting fuzzing as one of its major services (*e.g.*, Project Springfield on Microsoft Azure [31]) or abstracting fuzzing instances inside a VM or container (*e.g.*, OSS-Fuzz by Google [22]), the scalability of underlying abstractions plays an important role in determining the fuzzing performance. Our goal in this work is to improve the performance and scalability of fuzzing in a multi-core machine. However, there may exist different bottlenecks when fuzzing with a large-scale VM cluster because of the semantic gap between a VM and the hypervisor. Efficient and scalable hypervisor primitives are required to bridge the semantic gap between a hypervisor, a guest OS, and all the way up to the fuzzing instance.

9 CONCLUSION

Fuzzing is now one of the most critical tools to find several security bugs in various organizations and communities. However, with increasing code bases and trivial bugs vanishing out of air, fuzzers nowadays spend days, weeks or even months to find critical bugs that not only requires large computing resources but also results in monetary expenditure. Till now, prior works have only focused on producing interesting input test cases to find new bugs quickly, but have forgo the design aspects from a system's perspective. In this work, we carefully study and profile the various components of two state-of-the-art fuzzers and their interaction with the OS and find three design flaws, which we address for two fuzzers: `AFL` and `LibFuzzer`. With our proposed operating primitives, `AFL` has at most 7.7×, 25.9×, and 28.9× improvement on the number of executions per second on 30, 60, and 120 cores, respectively. Meanwhile, `LibFuzzer` can speed up by at most 170.5×, 134.9×, and 735.7× on 30, 60, and 120 cores respectively.

10 ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful feedback. This research was supported, in part, by the NSF award DGE-1500084, CNS-1563848, CNS-1704701 and CRI-1629851, ONR under grant N000141512162, DARPA TC (No. DARPA FA8650-15-C-7556), and XD3 programs (No. DARPA HR0011-16-C-0059), and ETRI IITP/KEIT[B0101-17-0644], and gifts from Facebook, Mozilla and Intel.

REFERENCES

[1] Nightmare, 2014. https://github.com/joxeankoret/nightmare.

[2] zzuf, 2016. https://github.com/samhocevar/zzuf.

[3] Pwn2Own 2017: Chrome Remains the Winner in Browser Security, 2017. https://securityzap.com/pwn2own-2017-chrome-remains-winner-browser-security/.

[4] B. Wickizer, S., Kaashoek, M. F., Morris, R., and Zeldovich, N. OpLog: a library for scaling update-heavy data structures. *CSAIL Technical Report* (2013).

[5] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Dec. 2008).

[6] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (San Francisco, CA, Apr. 2008).

[7] Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016).

[8] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, M. F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Dec. 2008).

[9] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, Oct. 2010).

[10] Boyd-Wickizer, S., Kaashoek, M. F., Morris, R., and Zeldovich, N. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium* (Ottawa, Canada, July 2012).

[11] Boyd-Wickizer, S., Kaashoek, M. F., Morris, R., and Zeldovich, N. OpLog: a library for scaling update-heavy data structures.

[12] CENSUS. Choronzon - An evolutionary knowledge-based fuzzer. ZeroNights Conference.

[13] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. Unleashing MAYHEM on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)* (San Francisco, CA, May 2012).

[14] Chang, O., Arya, A., Serebryany, K., and Armour, J. OSS-Fuzz: Five months later, and rewarding projects, 2017. https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html.

[15] Chen, Y., Reymondjohnson, S., Sun, Z., and Lu, L. Shreds: Fine-grained execution units with private memory. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, May 2016).

[16] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (London, UK, Mar. 2012).

[17] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys)* (Prague, Czech Republic, Apr. 2013).

[18] Clements, A. T., Kaashoek, M. F., Zeldovich, N., Morris, R. T., and Kohler, E. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (Farmington, PA, Nov. 2013).

[19] Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)* (Copper Mountain, CO, Dec. 1995), pp. 251–266.

[20] Google. Fuzzing for Security, 2012. https://blog.chromium.org/2012/04/fuzzing-for-security.html.

[21] Google. Honggfuzz, 2016. https://google.github.io/honggfuzz/.

[22] Google. OSS-Fuzz - Continuous Fuzzing for Open Source Software, 2016. https://github.com/google/oss-fuzz.

[23] Google. syzkaller - linux syscall fuzzer, 2016. https://github.com/google/syzkaller.

[24] Google. fuzzer-test-suite: Set of tests for fuzzing engines, 2017. https://github.com/google/fuzzer-test-suite.

[25] iSEC. PeachFarmer, 2014. http://github.com/iSECPartners/PeachFarmer.

[26] Kashyap, S., Min, C., and Kim, T. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (Santa Clara, CA, July 2017).

[27] Kim, S. Y., Lee, S., Yun, I., Xu, W., Lee, B., Yun, Y., and Kim, T. CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (Santa Clara, CA, July 2017).

[28] Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., and Druschel, P. Light-weight contexts: an OS abstraction for safety and performance. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Savannah, GA, Nov. 2016).

[29] LLVM. libFuzzer - a library for coverage-guided fuzz testing, 2017. http://llvm.org/docs/LibFuzzer.html.

[30] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013), pp. 461–472.

[31] Microsoft. Microsoft previews Project Springfield, a cloud-based bug detector, 2016. https://blogs.microsoft.com/next/2016/09/26/microsoft-previews-project-springfield-cloud-based-bug-detector.

[32] Min, C., Kashyap, S., Maass, S., Kang, W., and Kim, T. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)* (Denver, CO, June 2016).

[33] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R., and Hunt, G. C. Rethinking the Library OS from the Top Down. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, CA, Mar. 2011), pp. 291–304.

[34] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., and Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb.–Mar. 2017).

[35] Ruderman, J. Releasing jsfunfuzz and domfuzz, 2015. http://www.squarefree.com/2015/07/28/releasing-jsfunfuzz-and-domfuzz/.

[36] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2016).

[37] The Clang Team. Clang 5 documentation - SanitizerCoverage, 2017. http://clang.llvm.org/docs/SanitizerCoverage.html.

[38] Veggalam, S., Rawat, S., Haller, I., and Bos, H. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *Proceedings of the 21th European Symposium on Research in Computer Security (ESORICS)* (Crete, Greece, Sept. 2016).

[39] Zalewski, M. Fuzzing random programs without execve(), 2014. https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html.

[40] Zalewski, M. AFL starting test cases), 2017. https://github.com/mirrorer/afl/tree/master/testcases.

[41] Zalewski, M. american fuzzy lop (2.41b), 2017. http://lcamtuf.coredump.cx/afl/.

[42] Zalewski, M. Technical "whitepaper" for afl-fuzz), 2017. https://github.com/mirrorer/afl/blob/master/docs/technical_details.txt.

[43] Zalewski, M. Tips for performance optimization, 2017. https://github.com/mirrorer/afl/blob/master/docs/perf_tips.txt.