KRACE: Data Race Fuzzing for Kernel File Systems

Meng Xu Sanidhya Kashyap Hanqing Zhao Taesoo Kim
Georgia Institute of Technology

***Abstract*—Data races occur when two threads fail to use proper synchronization when accessing shared data. In kernel file systems, which are highly concurrent by design, data races are common mistakes and often wreak havoc on the users, causing inconsistent states or data losses. Prior fuzzing practices on file systems have been effective in uncovering hundreds of bugs, but they mostly focus on the sequential aspect of file system execution and do not comprehensively explore the concurrency dimension and hence, forgo the opportunity to catch data races.**

In this paper, we bring coverage-guided fuzzing to the concurrency dimension with three new constructs: 1) a new coverage tracking metric, alias coverage, specially designed to capture the exploration progress in the concurrency dimension; 2) an evolution algorithm for generating, mutating, and merging multi-threaded syscall sequences as inputs for concurrency fuzzing; and 3) a comprehensive lockset and happens-before modeling for kernel synchronization primitives for precise data race detection. These components are integrated into KRACE, an end-to-end fuzzing framework that has discovered 23 data races in `ext4`, `btrfs`, and the `VFS` layer so far, and 9 are confirmed to be harmful.

I. INTRODUCTION

In the current multi-core era, concurrency has been a major thrust for performance improvements, especially for system software. As is evident in kernel and file system evolutions [1–4], a whole zoo of programming paradigms is introduced to exploit multi-core computation, including but not limited to asynchronous work queues, read-copy-update (RCU), and optimistic locking such as sequence locks. However, alongside performance improvements, concurrency bugs also find their ways to the code base and have become particularly detrimental to the reliability and security of file systems due to their devastating effects such as deadlocks, kernel panics, data inconsistencies, and privilege escalations [5–12].

In the broad spectrum of concurrency bugs, data races are an important class in which two threads erroneously access a shared memory location without proper synchronization or ordering. Obstructed by the non-determinism in thread interleavings, data races are notoriously difficult to detect and diagnose, as they only show up in rare interleavings that require precise timing to trigger. Even worse, unlike memory errors that tend to crash the system immediately upon triggering, data races do not usually raise visible signals in the short term and are often identified retrospectively when analyzing assertion failures or warnings in production logs [13].

As the state of the practice, file system developers often rely on stress testing to find data races proactively [14, 15]. By saturating a file system with intensive workloads, the chance of triggering uncommon thread interleavings, and thus data races, can be increased. However, while useful, stress testing has significant shortcomings: handwritten test suites are far from sufficient to cover the enormous state space in file system execution, not to mention keeping up with the rapid increase in file system size and complexity.

More recently, coverage-guided fuzzing has proven to be a useful complement to handwritten test suites, with thousands of vulnerabilities found in userspace programs [16–20]. Without a doubt, kernel file systems can be fuzzed, and generic OS fuzzers [21–23] have demonstrated their viability with over 200 bugs found. In addition, file system-specific fuzzers, Janus [5] and Hydra [6], have extended the scope of file system fuzzing from memory errors into a broad set of semantic bugs, while the data race-specific fuzzer, Razzer [24], has shed lights on data race detection by combining fuzzing and static analysis. At the core of these fuzzers is the coverage measurement scheme, which summarizes unique program behaviors triggered by a given input in bitmaps. The fuzzer compares per-input coverage against the accumulated coverage bitmaps to measure the "novelty" of the input and determines whether it should serve as the seed for future fuzzing rounds.

However, almost all existing coverage-guided fuzzers focus on tracking the *sequential* aspect of program execution only and fail to treat *concurrency* as a first-class citizen. To illustrate, branch coverage (*i.e.*, control flow transition between basic blocks) has been the predominant coverage measurement metric. But such a metric captures little information about thread interleavings: different interleavings are likely to result in the same branch coverage (Figure 2), while only a small fraction may trigger a data race (Figure 3).

With the sequential view of program execution, existing kernel fuzzers have been very effective in mutating and synthesizing single-threaded syscall sequences based on seed traces [25, 26] to maximize branch coverage. But no heuristics have been proposed in synthesizing multi-threaded sequences to maximize thread interleaving coverage. Last but not least, given that data races often lead to silent failures, treating only kernel panics or assertions as bug signals is not sufficient: a data race checker that handles kernel complexity is needed.

To bring coverage-guided fuzzing to the concurrency dimension, in this paper, we present KRACE, an end-to-end fuzzing framework that fills the gap with new components in three fundamental aspects in kernel file system fuzzing:

Coverage tracking [§III] KRACE adopts two coverage tracking mechanisms. Branch coverage is tracked as usual to capture code exploration in the sequential dimension, analogous to the line coverage metric used in unit testing. In addition, to approximate exploration progress in the concurrency domain, KRACE proposes a novel coverage metric: alias instruction

pair coverage, short for alias coverage. Conceptually, if we could collect all pairs of memory access instructions `X`↔`Y` such that `X` in one thread *may-interleave* against `Y` in another thread, alias coverage tracks how many such interleaving points have been covered in execution. Consequently, if the growth of alias coverage stalls, it signals the fuzzer to stop probing for new interleavings in the current multi-threaded seed input.

Input generation [§IV] KRACE generates and mutates individual syscalls according to a specification [21, 27]. The novel part of KRACE lies in evolving multi-threaded seeds and merging them in an interleaved manner to preserve already-found coverage as well as to maximize the chances of inducing new interleavings. Another job of the input generator is to produce thread schedulings, (to explore the hidden input space). Although enforcing fine-grained control over thread scheduling is possible [7], the scheduling algorithm does not scale to whole-kernel concurrency, as the latter consists of not only user threads, but also background threads internally forked by file systems, work queues, the block layer, loop devices, RCUs, etc., and the total number of contexts often exceeds 60 at runtime. As a result, KRACE adopts a lightweight delay injection scheme and relies on the alias coverage metric as feedback to determine whether more delay schedules are needed.

Bug manifestation [§V] KRACE incorporates an in-house developed detector to reason about data races given an execution trace. In essence, KRACE hooks every memory access and for each pair of accesses to the same memory address, KRACE checks whether 1) they belong to two threads and at least one is a memory write; 2) these two accesses are strictly ordered (*i.e.*, happens-before relation); and 3) at least one shared lock exists that guards such accesses (*i.e.*, lockset analysis). The challenges for KRACE lie in modeling the diverse set of kernel synchronization mechanisms comprehensively, especially those uncommon primitives such as optimistic locking, RCU, and ad-hoc schemes implemented in each file system.

KRACE adopts the software rejuvenation strategy to avoid the aging OS problem, *i.e.*, every execution is a fresh run from a clean-slate kernel and empty file system image. Doing so trades performance for trackability and debuggability but is worthwhile for data race detection. As shown in §VII-B, the exploration gradually catches up and bypasses conventional speed-oriented fuzzers (*e.g.*, Syzkaller) upon saturation. KRACE also decouples data race checking from state exploration. Unlike prior works where the bug checker runs inline in each execution, in KRACE, the checker only kicks in when new coverage (either branch or alias) is reported. This prevents the expensive data race checking from slowing down the state exploration while still preserving the opportunity to test every new execution state found through fuzzing. The checking progress will eventually catch up when the coverage growth is toward saturation.

We evaluated KRACE by fuzzing two popular and heavily tested kernel file systems (`ext4` and `btrfs`) in recent kernel versions and we found 23 data races, nine of which are confirmed as potentially harmful races, and 11 are benign races (for performance or allowed by the POSIX specification).

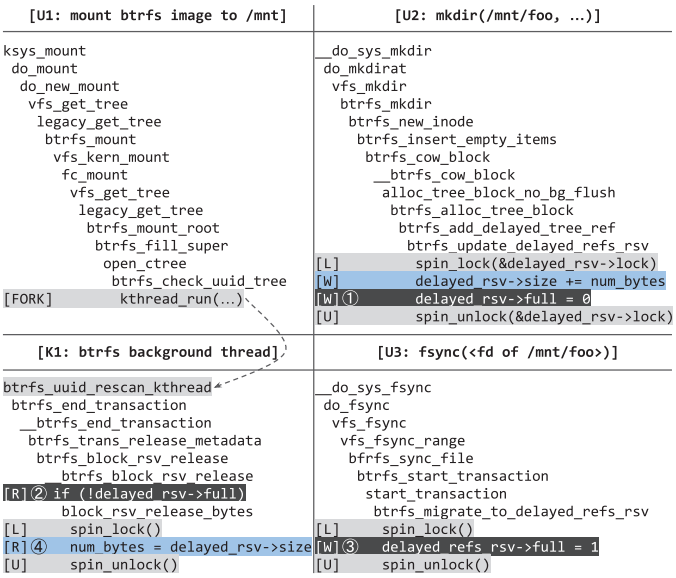

Fig. 1: A data race found by KRACE. This figure shows the complete call stack, thread ordering information, and locking information when the data race happens and the inconsistency it may cause (①-④).

Summary: This paper makes the following contributions:

- **Concept**: The alias coverage metric and interleaved multi-threaded syscall sequence merging are novel concepts that make coverage-guided fuzzing more effective in highly concurrent programs, possibly as a first step toward fuzzing for a wide range of concurrency bugs.
- **Implementation**: KRACE's data race checker encodes a comprehensive model of kernel synchronization mechanisms in the form of over 100 kernel patches (for code instrumentation), which are regularly updated as the kernel upgrades.
- **Impact**: KRACE has found 23 data races and will be continuously running to find new cases. We will open-source KRACE as well as the collection of syscall primitives for multi-threaded execution as quality seeds for future concurrent file system fuzzing research.

II. Background and Related Work

The past three decades have witnessed several efforts to find data races using various techniques. In this section, we show a data race example, discuss the types of approaches that prior works have taken, and introduce coverage-guided fuzzing as a generic bug finding technique.

Example. Intuitively, a data race is caused by two threads trying to perform unordered and unprotected memory operations to the same address. Figure 1 shows two data races found by KRACE that happen to make a complete scenario. The read of `full` is in race with both writes, as the read is not protected by the corresponding `delayed_rsv->lock` as is done on the writers' side. According to `btrfs` developers, this results in ineffective management of the reserve space internally used by `btrfs`, in particular, delays in releasing the reserved space or space releasing followed by reservation instead of migration from one reserve to another. Reflected in the call stack, if the execution takes the order of ①→②→③→④, then

`block_rsv_release_bytes` is inadvertently releasing bytes that will be used by the `fsync`. Such a case might eventually cause integer overflows in the reserve space but would probably require thousands of concurrent file operations to trigger.

Data race is a special type of race condition, and hunting data races in complex software involves two facets: *1) how to confirm an execution is racy* and *2) how to produce meaningful executions by exploring code and thread-scheduling*.

Dynamic data race detection algorithms. Most of the initial works [28] found race conditions by relying on the *happens-before* analysis [29]. However, one of the prime issues with this approach is that it leads to false negatives. To improve the detection accuracy, Eraser [30] proposed the *lockset analysis*, in which users annotate the common lock/unlock methods and find atomicity violations. Later, several works [31, 32] proposed optimizations to either mitigate the overhead or minimize false positives. To further improve the effectiveness of dynamic data race detection, several works [33, 34] combined the idea of happens-before relation with lockset analysis.

Unfortunately, most of these works target userspace programs using simple synchronization primitives (*e.g.*, those provided by `pthread` or Java runtime), which only represent a small subset of synchronization mechanisms available in the Linux kernel. KRACE follows the same trend in combining happens-before and lockset analysis, but unlike prior works, KRACE provides a comprehensive framework that includes not only simple locking methods, such as pessimistic locks (*e.g.*, mutex, readers-writer lock, spinlock, etc.), but also optimistic locking protocols, such as sequence locks, and other forms of synchronization mechanisms that imply more than just mutual exclusion, *e.g.*, RCU [35] and other publisher-subscriber models.

Both lockset and happens-before analysis require code annotations and suffer from incompleteness, *i.e.*, a missing lock model leads to false positives. Several works overcome this issue with timing-based detection, *i.e.*, a thread is delayed for a certain duration at some memory accesses while the system observes whether there are conflicting accesses to the same memory during the delay [13, 36, 37]. Moreover, most of these works resort to sampling [13, 34, 37–39], as an optimization over completeness, to further minimize the runtime overhead caused by tracking memory accesses or code paths.

However, complete timing-based detection relies on precise control of thread execution speed and results in an enormous search space (both in where to delay and how long to delay), which again is not scalable in the kernel scope. As a result, in terms of race detection, KRACE resorts to a trial-and-error approach and fixes false positives introduced by ad-hoc mechanisms along with the development. Fortunately, due to the high coding standard and strict code review practice, ad-hoc synchronization is not common in kernel file systems.

Code/thread-schedule exploration. The effectiveness of a data race checker depends not only on the detection algorithm but also on how well the checker can explore execution states and cover as many code paths and thread interleavings as possible. For code path exploration, prior detectors mostly rely on manually written test suites [7, 36, 37] that do not capture complicated cases. As shown in Figure 1, triggering the data race would require a user thread to `mkdir` on the same block the background `uuid_rescan` thread is working on, which (almost) in no way can be specified in manually written test cases. An alternative is to enumerate code paths statically [40–44], but this is not scalable. Recent OS fuzzers adopt specification-based syscall synthesization [5, 6, 21, 27]. However, these fuzzers mostly focus on generating sequential programs instead of multi-threaded programs and are not intended to explore interleavings in syscall execution. KRACE adopts a similar synthesization approach, but instead of focusing on single-threaded sequences, KRACE evolves multi-threaded programs.

In the case of thread-schedule exploration, prior approaches fall into three categories, in decreasing order of scalability but increasing order of completeness: 1) stressing the random scheduler with multiple trials [14]; 2) injecting delays at runtime [13, 36, 37]; and 3) enumerating every possible thread interleaving [7, 24]. KRACE uses delay injection, a trade-off among scalability, practicality, and completeness.

Data race detection in kernels. KRACE shares its design ideology with four prominent works [7, 24, 45, 46]. DataCollider [45] is the first work that tackles this problem by using randomized sampling of a small number of memory accesses in conjunction with code breakpoint and data breakpoint facilities for efficient sampling. DataCollider is simple enough to detect several bugs in the Windows kernel modules. A similar strategy is used by Syzkaller [21] with its Kernel Concurrent Sanitizer [46] (KCSan) module. KCSan is a dynamic data race detector that uses compiler instrumentation, *i.e.*, software watchpoints instead of hardware watchpoints, to detect bugs on non-atomic accesses that violate the Linux kernel memory model [47] using happens-before analysis.

SKI [7] focuses on comprehensive enumeration of thread schedules with the PCT algorithm [48] and hardware breakpoints. However, SKI permutes user threads only to find data races in the syscall handlers and thus forgoes the opportunities to find data races in kernel background threads. Furthermore, even with user threads only, the number of permutations can be huge to test thoroughly. In addition, the test suites used by SKI may be too small to explore an OS for bugs.

Razzer [24] combines static analysis with fuzzing for data race detection. In particular, Razzer first runs a points-to analysis across the whole kernel code base to identity potentially alias instruction pairs, *i.e.*, memory accesses that may point to the same memory location. After that, per each alias pair identified, Razzer tries to generate syscalls that reach the racy instructions at runtime. It does so with fuzzy syscall generation [21, 27], and sequential syscall traces are generated first. Once the alias relation is confirmed in the sequential execution, the trace is then parallelized into multi-threaded traces for actual data race detection.

Razzer presents an elegant pipeline for data race fuzzing, but it can be further improved: 1) running points-to analysis [49] on kernel file systems produces millions of may-alias pairs, which is almost impossible to enumerate one by one; 2) even

for one alias pair, how to generate syscalls that may reach the racy instructions is less clear. KRACE aims to improve both aspects with the novel notion of alias coverage. Instead of pre-calculating the search space with points-to analysis, KRACE relies on coverage-guided fuzzing to expand the search in the concurrency dimension gradually. Analogically, this is similar to not enumerating every path in the control-flow graph but instead using an edge-coverage bitmap to capture the search progress. Doing so also eliminates the concern on how to generate syscalls that lead execution to specific locations.

Fuzzing in general. Fuzzing has proven to be a practical approach to find bugs in today's software stack, both in the userspace [16, 20, 50–54] and in the kernel space [5, 6, 21, 22, 27, 55]. Unfortunately, existing works cannot be trivially adopted for data race fuzzing. One reason is that the main focus of fuzzing has been on finding memory corruptions or triggering assertions. Although Hydra [6] extends the scope beyond memory errors into semantic bugs in file systems, it does not provide any insight into finding data races.

Moreover, since modern coverage-guided fuzzing originates and prospers from testing single-threaded programs such as binutils, encoder/decoders, and the CGC and LAVA-M fuzzing benchmarks, recent fuzzing efforts have focused on optimizing fuzzers' performance on single-threaded executions too, such as approximating sequential execution with neural networks [51]. Not surprisingly, when the fuzzing practice is carried down to the OS level [21, 22, 27, 55–59], the same sequential view of program execution is inherited.

Although generating structured inputs has been a challenge for kernel fuzzing, many improvements have been proposed. For example, MoonShine [25] captures dependencies between syscalls and DIFUZE [26] generates interface-aware inputs. However, lacking a coverage metric and a seed evolution algorithm to handle state exploration in the concurrency dimension, existing OS fuzzers miss the opportunities to find the broad spectrum of concurrency bugs, including data races. The motivation behind KRACE is to fill this gap and to bring coverage-guided fuzzing to the concurrency dimension.

Static and symbolic analysis on kernels. Although KRACE is a dynamic analysis system, we are also aware of works that aim to find concurrency bugs with static analysis [40–44]. Most of these approaches rely on static lockset analysis and, hence, suffer from the high false-positive rate caused by missing the happens-before relation in the execution as well as the inherent limitations of the points-to analysis. For instance, RacerX [41] suffers from 50% false positives on the Linux kernel.

Beyond concurrency bugs, static analysis has proven effective in finding many security issues in kenrel drivers. For example, SymDrive [60] uses symbolic execution to emulate devices and verify the properties of kernel drivers; DrChecker [61] is capable of finding eight types of security issues by relaxing the completely sound analysis on unbounded loops with mostly sound versions. However, a major challenge in applying these works to data race detection in file systems is their lack of statefulness, *i.e.*, although extremely effective in finding bugs within one syscall execution, they miss bugs that occur because of the interaction between multiple syscalls, which happen to be the majority of cases in file system operations.

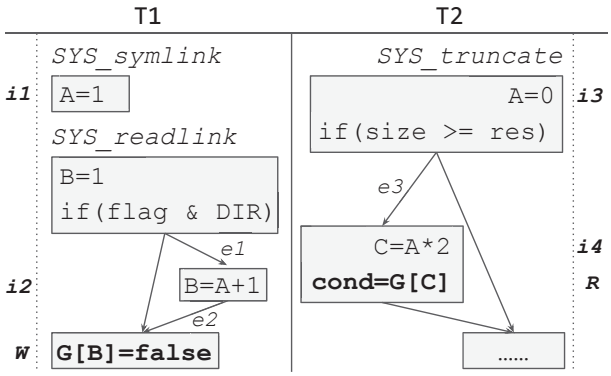

Fig. 2: A data race found by KRACE when `symlink`, `readlink`, and `truncate` on the same `inode` run in parallel (simplified for illustration). The race is on the indexed accesses to a global array `G` and occurs only when `B==C`. `A` is lock-protected. This is one example showing branch coverage is not sufficient in approximating execution states of highly concurrent programs. It is not difficult to cover all branches in this case with existing fuzzers, but to trigger the data race, merely covering branches `e1-e3` is not enough. The thread interleavings between four instructions `i1-i4` are equally important. The valid interleavings that may trigger the data race are shown in Figure 3.

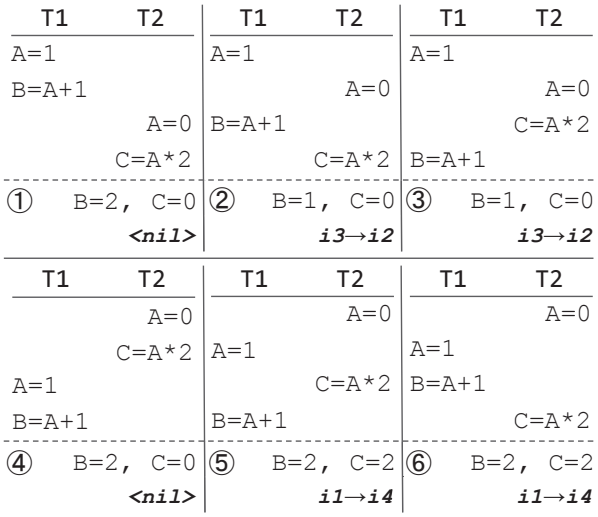

Fig. 3: Possible thread interleavings among the four instructions shown in Figure 2. Out of the 6 interleavings, only 3 interleavings (①/④, ②/③, ⑤/⑥) are effective depending on `A`'s value when `B` and `C` read it. Each effective interleaving results in different alias coverage. Only ⑤/⑥ may trigger the data race.

III. A Coverage Metric for Concurrent Programs

In this section, we show why branch coverage, the golden metric for fuzzing, might be insufficient to represent the exploration in the concurrency dimension, while at the same time, why alias coverage, our new proposal, fits this purpose.

A. Branch coverage for the sequential dimension

Branch coverage originates from the program control-flow graph (CFG), which is inherently a sequential view of program

execution. As shown in Figure 2, in CFGs, execution flows through basic blocks and user-controllable inputs, *e.g.*, `size` in `SYS_truncate`, determine the set of edges that join the basic blocks. For a branch coverage-guided fuzzer: given an input (*e.g.*, a list of syscalls), it tracks the set of edges that are hit at runtime and leverages this feedback to decide whether this input is "useful" and should be kept for more mutations. Intuitively, the fuzzer expects to probe more branches by mutating the seed, and not surprisingly, once the branch coverage growth stalls, the fuzzer will shift focus to other seeds.

In the case shown in Figure 2, exhausting all branches sequentially will only yield the status of `B==1`, `B==2`, and `C==0`. After that, these execution paths (represented by the seeds covering them) will be de-prioritized and considered non-interesting by the fuzzer. However, this is not the end of the story. To trigger the data race when `B==C==2`, the execution of four critical instructions (`i1-i4`) has to be interleaved in a special way, as shown in Figure 3. Unfortunately, all six interleavings yield the same branch coverage, and the fuzzer is likely to give up the seed upon hitting a few of them.

Further note that this is an extremely simplified example that involves only six possible interleavings among two threads. In actual executions, the concurrency dimension can be huge, as the instructions executed by each thread are usually in the thousands or even millions, while there will be tens of threads running at the same time. As a result, when fuzzing highly concurrent programs, we need to pay attention to not only code paths explored, but also meaningful thread interleavings explored that yield to the same branch coverage. In other words, if the fuzzer believes that there could be unexplored thread interleavings in a seed, the seed should not be de-prioritized.

B. Alias coverage for the concurrency dimension

Intuition. At first thought, recording the exploration of thread interleavings can be futile. A realistic kernel file system at its peak time may use over 60 internal threads, where each thread may execute over 100,000 instructions. The total possible number of thread interleavings is 60^{100000}, an enormous search space that no bitmap can ever approximate.

However, it is worth noting that not all interleaved executions are useful. In fact, only interleavings of memory-accessing instructions to the same memory address matters. As shown in Figure 2, interleaving instructions apart from `i1-i4` has no effect on the final results of `B`, `C`, as well as the manifestation of the data race. This is true in the actual code, where hundreds and thousands of instructions sit between `i1`, `i3` and `i2`, `i4`.

In other words, based on the crucial observation that data races, and even in the broader term, concurrency bugs, typically involve unexpected interactions among a few instructions executed by a small number of threads [7, 62, 63], if KRACE is able to track how many interactions among these few memory-accessing instructions have been explored, it is sufficient to represent thread interleaving coverage and to find data races. This is precisely what gets tracked by alias coverage.

A formal definition. First, suppose all memory-accessing instructions in a program are uniquely labeled: `i1`, `i2`, ..., `iN`. At runtime, each memory address `M` keeps track of its last *define operation*, *i.e.*, the last instruction that writes to it as well as the context (thread) that issues the write, represented by `A` ← `<ix, tx>`. Now, in the case in which a new access to `M` is observed, carried by instruction `iy` from context `ty`: if `iy` is a write instruction, update `A` ← `<iy, ty>` to reflect the fact that `A` is redefined. Otherwise:

- if `tx == ty`, *i.e.*, same context memory access, do nothing,
- or else, record *directed* pair `ix`→`iy` in the alias coverage.

Figure 3 is a working example of this alias coverage tracking rule. In cases ① and ④, there is no inter-context define-then-use of memory address `A`, and hence, the alias coverage map is empty. On the other hand, in cases ② and ③, the calculation of `B` in `T1` relies on `A` defined in `T2`, hence the pair `i3`→`i2`. The same rule applies to cases ⑤ and ⑥.

Feedback mechanism. Essentially, alias coverage provides a signal to the fuzzer on whether it should expect more useful thread interleavings out of the current test case, *i.e.*, a multi-threaded syscall sequence. If the alias coverage keeps growing, the fuzzer should come up with more delay schedules to inject at the memory-accessing instructions (detailed in §IV-B) in the hope of probing unseen interleavings. Otherwise, if the coverage growth stalls, it is a sign that the concurrency dimension of the current test case is toward saturation, and the most economical choice is to switch to other seeds for further exploration.

Coverage sensitivity fine-tuning. Finding one-suits-all coverage criteria has been a never-ending quest in software engineering [64]. Even the branch coverage has several variations, such as N-gram branch coverage, context-sensitive coverage [52], etc., which are well-documented and compared in a recent survey [65]. However, despite the fact that branch coverage is always subsumed by program whole-path coverage, branch coverage is still preferred over path coverage, as the latter is overly sensitive to input changes and thus requires a much larger bitmap to hold and compare. On the other hand, branch coverage strikes a balance among effectiveness, execution speed, and bitmap accounting overhead.

Similarly, alias coverage strives to find such a balance point in the concurrency dimension. In our experiments with kernel file system fuzzing, KRACE observed 63,590 unique pairs of alias instructions (directed access). Based on the data, for an empirical estimation, a bitmap of size 128KB should be sufficient to avoid heavy collisions, which is close to AFL's branch coverage bitmap size (64KB). In addition, if more sensitivity is needed for alias coverage, KRACE can be easily adopted from 1st-order alias pair (alias coverage) to 2nd-order alias pair, Nth-order alias pair, and up-to total interleaving coverage. We leave this for future exploration.

IV. INPUT GENERATION FOR CONCURRENCY FUZZING

In this section, we present how to synthesize and merge multi-threaded syscall sequences for file system fuzzing, as well as how to exploit a hidden input domain—thread delay schedule—to accelerate thread interleaving probing.

Fig. 4: Illustration of four basic syscall sequence evolution strategies supported in KRACE: mutation, addition, deletion, and shuffling. For KRACE, each seed contains multi-threaded syscall sequences and each thread trace is highlighted in different shades of grayscale.

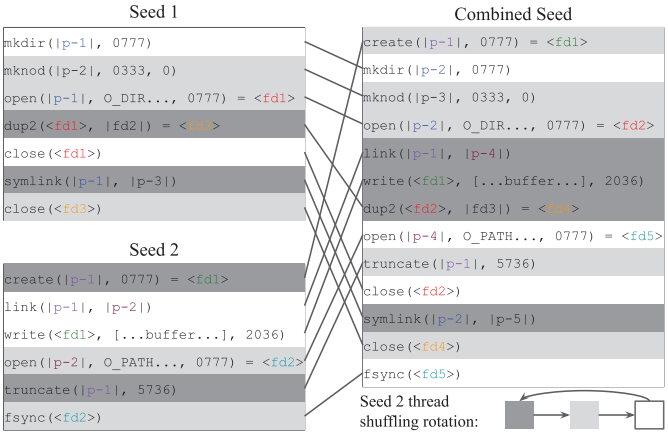

Fig. 5: Semantic-preserving combination of two seeds. For KRACE, each seed contains multi-threaded syscall sequences and each thread trace is highlighted in different shades of grayscale.

A. Multi-threaded syscall sequences

Specification-based synthesization. The goal of syscall generation and mutation is to generate diverse and complex file operations that are otherwise difficult for human developers to contemplate. Given that syscalls are highly structured data, it is almost fruitless to mutate their arguments blindly. As a result, we use a specification to guide the generation and mutation of syscall arguments. A feature worth highlighting in KRACE's specification is the encoding of inter-dependencies among syscalls, especially path components and file descriptors (fd), which are most relevant to file system fuzzing. To illustrate, as shown in Figure 5, the `open` syscall in seed 1 reuses the same path component in the `mkdir` syscall, while the `write` syscall in seed 2 relies on the return value of `creat`.

Seed format. The seed input for KRACE is a multi-threaded syscall sequence. Internally, it is represented by a single list of syscalls (*a.k.a*, the main list) and a configurable number of sub-lists (3 in KRACE) in which each sub-list contains a disjoint sequence of syscalls in the main list. Each sub-list represents what will be executed by each thread at runtime. To illustrate, as shown in Figure 5, seed 1 has three threads, where each thread will be executing `mkdir-close`, `mknod-open-close`, and `dup2-symlink`, respectively, marked in different grayscale.

Evolution strategies. KRACE uses four strategies to evolve a seed for both branch and alias coverage, as shown in Figure 4.

- Mutation: a randomly picked argument in one syscall will be modified according to specification. If a path component is mutated, it is cascaded to all its dependencies.
- Addition: a new syscall can be added to any part of the trace in any thread, but must be after its origins.
- Deletion: a random syscall is kicked out of the main list and the sub-list. In case a file descriptor is deleted, its dependencies are forced to re-select another valid file.
- Shuffling: syscalls in the main list are redistributed to sub-lists, but their orders in the main list are preserved.

Merging multi-threaded seeds. The power of fuzzing lies not only in evolving a single seed but also in joining two seeds to produce more interesting test cases. To enable seed merging in KRACE, a naive solution might be simply to concatenate two traces. However, this is not the most economical use of seeds, as it forgoes the opportunities to find new coverage by further interleaving these high-quality executions.

KRACE adopts a more advanced merging scheme: upon merging, the main lists of the two seeds are interweavingly joined, *i.e.*, the relative orders of syscalls are still preserved in the resulting main list as well as in the sub-lists. As a result, the syscall inter-dependencies are preserved too. As shown in Figure 5, all the dependencies on path and fds are properly preserved after merging (highlighted in corresponding colors).

Primitive collection. Successful syscalls are valuable assets out of the file system fuzzing practice, not only because they lead to significantly broader coverage than failed syscalls, but also because they can be difficult, and sometimes even fortunate, to generate due to the dependencies among them. This is true especially for long traces of closely related syscalls. As a result, upon discovering a new seed, KRACE first prunes it and retains only successful syscalls and further splits these syscalls into non-disjoint primitives where each primitive is self-contained, *i.e.*, for any syscall, all its path and fd dependencies (also syscalls) are captured in the same primitive.

Over the course of fuzzing, KRACE has accumulated a pool of around 10,000 primitives covering 68 file system related syscalls for which KRACE has a specification. In each primitive, file operations span across 3 threads, with each thread containing 1-10 syscalls, and most importantly, all syscalls succeed. We will open-source this collection in the hope that these primitives may serve as quality seeds for future concurrent file system fuzzing.

B. Thread scheduling control (weak form)

Thread scheduling is a hidden input domain for concurrency programs. Unfortunately, there is no way to control kernel scheduling by merely mutating syscall traces. Hooking the scheduling implementation (or using a hypervisor) and systematically permuting the schedules might be possible for small-scale programs [63] or for a few user threads in the kernel [7, 24]. But these algorithms are far from being

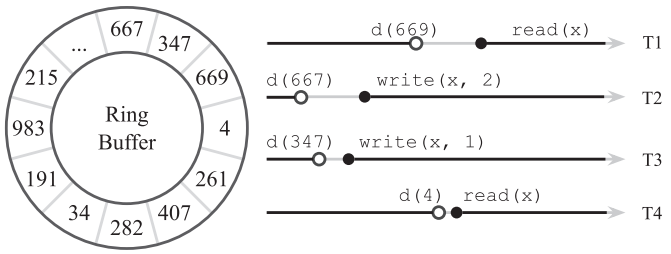

Fig. 6: The delay injection scheme in KRACE. In this example, white and black circles represent the memory access points before and after delay injection. Injecting delays uncovers new interleavings in this case, as the read and write order to the memory address `x` is reversed.

scalable enough to cover all kernel threads. For a taste of the scalability requirement, Figure 14 shows the level of concurrency introduced by the `btrfs` module alone, not to mention other background threads forked by the block layer, loop device, timers, and RCU.

Runtime delay injection. KRACE resorts to delay injection to achieve a weak (and indirect) control of kernel scheduling, based on the observation that only shared memory accesses matter in thread interleavings. KRACE's delay injection scheme is extremely simple, as shown in Figure 6. Before launching the kernel, KRACE generates a ring buffer of random numbers and maps it to the kernel address space. At every memory access point, the instrumented code fetches a random number from the ring buffer, say `T`, and delays for `T` memory accesses observed by KRACE system-wise (*i.e.*, in other threads).

A ring buffer is used to hold the random numbers, as KRACE cannot pre-determine how many injection points are needed for each execution, not to mention that such a number may be extremely large. Injecting delays at memory access points is at the finest granularity for delay injection. Although this works well in file system fuzzing, it might nevertheless be too fine-grained and introduces too much overhead. The injection points can be at the granularity of basic blocks or functions or even customized locations such as locking operations, etc.

V. A Data Race Checker for Kernel Complexity

Although the definition of data races is simple, finding them in a kernel execution trace can be difficult, primarily because of the variety of synchronization primitives available in the kernel code base as well as the ad-hoc mechanisms implemented by each individual file system. In this section, we enumerate the major categories of kernel synchronization primitives and describe how they can be modeled in KRACE.

A. Data race detection procedure

Overview. We say a pair of memory operations, `<ix, iy>`, is a data race candidate if, at runtime, we observed that

- they access the same memory location,
- they are issued from different contexts `tx` and `ty`,
- at least one of them is a write operation.

Such information is trivial to obtain dynamically by simply hooking every memory access. The difficulty lies in confirming whether a data race candidate is a true race. For this, we need two more analysis steps to check that:

- no locks are commonly held by both contexts, `tx` and `ty`, at the time when memory operations `ix` and `iy` are issued from them, respectively. [lockset (§V-B)]
- no ordering between `ix` and `iy` can be inferred based on the execution: *i.e.*, there is no reason `ix` *must happen-before* `iy` or the other way around, regardless of how `tx` and `ty` are scheduled. [happens-before (§V-C)]

Conceptually, lockset analysis produces no false negatives, *i.e.*, if there is a data race in the execution trace, it is guaranteed to be flagged by the lockset analysis. But lockset analysis is prone to false positives, as it ignores the ordering information. Happens-before analysis helps in filtering these false positives.

Kernel complexity. Although conceptually simple, lockset analysis requires a complete model of all locking mechanisms available in the kernel, and similarly, happens-before analysis requires all thread ordering primitives to be annotated. Otherwise, false positives will arise. However, after nearly 30 years of development, the Linux kernel has accumulated a rich set of synchronization mechanisms. KRACE takes a best-effort approach in modeling all major synchronization primitives as well as ad-hoc ones if we encounter them in our experiment. Due to space constraints, we present some representative ad-hoc schemes modeled by KRACE in appendix §C.

Besides the variety of synchronization events, the number of ordering points in the kernel execution is enormous. To get a taste of the complexity in real-world executions, Figure 18 shows a snippet of the ordering relation (*e.g.*, task queuing, waiting for conditions, etc.) across all user and kernel threads.

B. Lockset analysis

Most kernel locking primitives differentiate between reader and writer roles. The major difference is that a reader-lock can be acquired by multiple threads at the same time, as long as its corresponding writer-lock is not held; while a writer-lock can only be held by at most one thread. KRACE follows this distinction and tracks the acquisitions and releases of both reader- and writer-locks for each thread at runtime. Formally, such information is stored in the form of a lockset: denoted by $LS^R_{<t,i>}$ for the reader-side lockset for thread `t` at instruction `i` as well as $LS^W_{<t,i>}$ for the writer-side lockset. Both locksets are cached and attached to a memory cell whenever a memory access on that thread is observed, as shown in Figure 7.

The lockset analysis is simple as the following: for each data race candidate `<tx, ix>` and `<ty, iy>`, if any of the following conditions holds, this candidate cannot be a true data race.

$$LS^R_{<tx,ix>} \cap LS^W_{<ty,iy>} \neq \varnothing \quad (1)$$

$$LS^W_{<tx,ix>} \cap LS^R_{<ty,iy>} \neq \varnothing \quad (2)$$

$$LS^W_{<tx,ix>} \cap LS^W_{<ty,iy>} \neq \varnothing \quad (3)$$

On the other hand, if none of the conditions hold for a data race candidate, then the execution of `tx` and `ty` can be interleaved without restrictions around those memory accesses, as shown in the reading and writing of addresses `0x34` and `0x46` in Figure 7, hence, leading to data races.

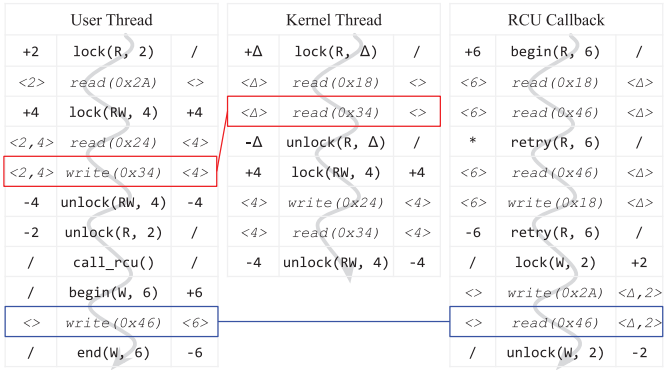

Fig. 7: Illustration of lockset analysis in KRACE. This example shows almost all locking mechanisms commonly used in the kernel, including
1) spin lock and mutexes—`[un]lock(RW, -)`,
2) reader/writer lock—`[un]lock(R/W, *)`,
3) RCU lock—specially denoted with symbol Δ, and
4) sequence lock—`begin/end/retry(R/W, *)`.
The left column shows the content in the reader lockset at the time of memory operation or changes to the lockset caused by other operations (/ denotes no change). The right column shows the writer counterpart. The two data races are highlighted in red and blue squares.

Pessimistic locking. Most of the kernel locking primitives are pessimistic locking, *i.e.*, whoever tries to acquire the lock will be blocked from further execution until the lock holder releases it. As a result, their APIs are always in pairs of `lock` and `unlock` to mark the start and end of a critical section. Examples of such locks include spin lock, reader/writer spin lock, mutex, reader/writer semaphore, and bit locks.

A slightly trickier primitive is the RCU lock, in which only reader-side critical sections are marked with `rcu_read_[un]lock` and the writer-side critical section is not marked by any lock/unlock APIs, instead, it is guaranteed by the RCU grace period waiting. More specifically, when `__rcu_reclaim` schedules an RCU callback into execution, it is guaranteed that there is no RCU reader-side critical section running. Hence, in KRACE, we hook the RCU callback dispatcher and mark RCU writer lock and unlock before and after the callback execution.

Optimistic locking. The Linux kernel is gradually shifting toward lock-free design and the most prominent evidence in recent years is the wide adoption of sequence locks [66]. A sequence lock is, in fact, more similar to a transaction than to a conventional lock. The reader is allowed to run optimistically into the critical section, hoping that the data it reads will not be modified during the transaction (hence the optimism), and aborts and retries if the data does get modified.

While boosting performance, a challenge brought by the sequence lock is that there is no clear end of the reader-side critical section. As shown in Figure 7, after a transaction `begins`, the `retry` can be called multiple times, perhaps one for mid-of-progress checking and the other one for before-commit checking; in theory, each `retry` could be an `unlock`-equivalent that marks the end of the critical section. If the lockset analysis is performed online (*i.e.*, during execution), the lockset states should fork to capture that the `retry` may or may not be an

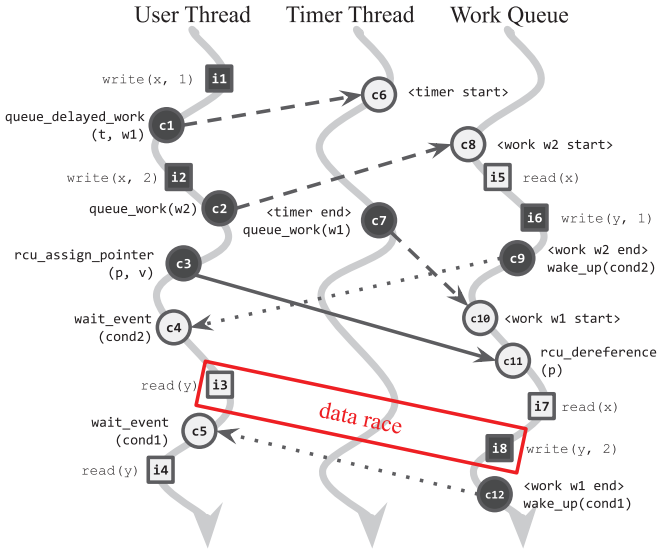

Fig. 8: Illustration of happens-before reasoning in KRACE. This example shows a very typical execution pattern in kernel file systems where the user thread schedules two asynchronous works on the work queue and checks for their results later in the execution. In particular, one of the asynchronous works is a delayed work that also goes through the timer thread. Fork-style, join-style, and publisher-subscriber relations are represented by dashed, dotted, and solid arrows, respectively. The only data race is highlighted in the red square.

`unlock`. For KRACE, since it uses offline lockset analysis, it may simply read the execution trace ahead to know whether there are more `retries` and behave correspondingly.

C. Happens-before analysis

Intuitively, happens-before analysis tries to find the causal relations between specific execution points in the threads. For example, a kernel thread only gets into running if another thread forks it; as a result, there is no way to schedule the spawned thread before the parent thread creates it. This implies that whatever happens before the thread creation points cannot be data racing against anything in the spawned thread. In the example shown in Figure 8, there is no way for `i2` to be racing against `i6`, as without queuing the work on the work queue (`c2`→`c8`), `i6` won't even be executed in the first place. Similarly, scheduling a thread that is waiting for a condition to be true will not make it run bypassing the barrier. Therefore, it is not possible for `i4` to race against `i8`, as only when the `wake_up` call is reached (`c12`→`c5`) can `i4` be executed.

This intuition shows how a happens-before relation can be formally checked: by hooking kernel synchronization APIs, *e.g.*, when a callback function is queued and when it is executed, we could find the synchronization points (nodes) between threads as well as the causality events (represented by edges), as shown in Figure 8. Since the nodes in one thread are already inherently connected according to program order, the whole execution becomes a directed acyclic graph. Consequently, determining whether two points, `<tx, ix>` and `<ty, iy>`, may race is translated into a graph reachability problem. If a path exists from `<tx, ix>` to `<ty, iy>`, it means that point `X` *happens-before* `Y` and thus cannot be racing. The same applies if we

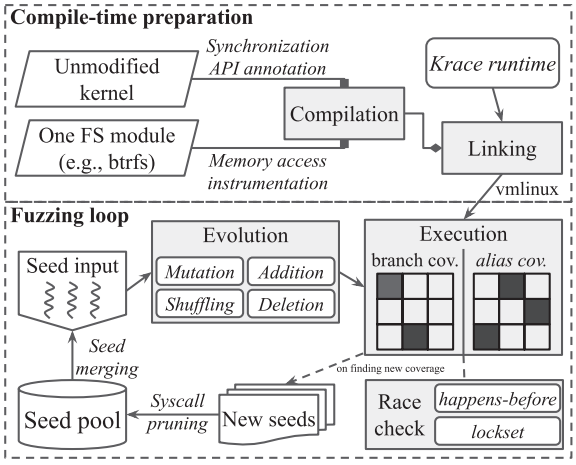

Fig. 9: An overview of KRACE's architecture and major components. Components in *italic* fonts are either new proposals from KRACE or existing techniques customized to meet KRACE's purpose.

can establish `Y` *happens-before* `X`. On the other hand, if no such path can be found, a happens-before relation cannot be established and the pair should be flagged, as in the case of `i3` and `i8`. All other accesses are reachable in the graph, and hence, they cannot be racing even without lock protections.

The happens-before relation commonly found in kernel file systems can be broadly categorized into three types:

Fork-style relations include RCU callbacks registered with `call_rcu`, work queues and `kthread`-simulated work queues, direct `kthread` forking, timers, software interrupts (`softirq`), as well as inter-processor interrupts (`IPI`). Hooking their kernel APIs is as easy as finding corresponding functions that register the callback and dispatch the callback.

Join-style relations include the completion API and a wide variety of `wait_*` primitives such as `wait_event`, `wait_bit`, and `wait_page`. Hooking their kernel APIs requires locating their corresponding `wake_up` calls besides the `wait` calls.

Publisher-subscriber model mainly refers to the RCU pointer assignment and dereference procedure [35]. For example, if one user thread retrieves a file descriptor (`fd`) from the `fdtable` which is RCU-guarded, the new `fd` must have been published first, hence the causality ordering. The object allocate-and-use pattern also falls into this realm: the publisher thread allocates memory spaces for an object, initializes its fields, and inserts the pointer to a global or heap-based data structure (usually a list or hashtable), while the subscriber thread later dereferences the pointer and uses the object. As a result, KRACE also tracks the memory allocation APIs and monitors when the allocated pointer is first stored into a public memory slot and when it is used again to establish the ordering automatically.

VI. PUTTING EVERYTHING TOGETHER

A. Architecture

Figure 9 shows the overall architecture of KRACE. The primary purpose of having the compile-time preparation is to embed a KRACE runtime into the kernel such that alias coverage (as well as branch coverage) can be collected dynamically. The runtime is also responsible for collecting information for data race checking, leveraging the kernel API hooking. On the other hand, the fuzzing loop is still conventional, covering seed selection, mutation, and execution, with the exception that in KRACE, a test case is considered "interesting" as long as new progress is found in either of the coverage bitmaps. In addition, all components are updated to handle the new seed format for concurrency fuzzing: multi-threaded syscall sequences.

Code instrumentation. Since the focus of KRACE is file systems, we only instrument memory access instructions in the target file system module and its related components such as the virtual file system layer (`VFS`) or the journaling module, *e.g.*, `jbd2` for `ext4`. On the other hand, API annotations are performed on the main kernel code base and have an effect even when the execution goes out of the functions in our target file system: the locks acquired and released, as well as the ordering primitives (*e.g.*, queuing a timer), will be faithfully recorded. In this way, KRACE does not suffer from false positives in cases like block layer calls into a callback in the file system layer but we do not know the prior locking contexts.

Fuzzing loop. Figure 15 shows the fuzzing evolution algorithm in KRACE. Fuzzing starts with producing a new program by merging two existing seeds. The seed selection criterion used in KRACE so far is simply frequency count, *i.e.*, less used seeds receive priority. We expect more advanced seed selection algorithms to be developed later. After merging, each program goes through several extension loops on which the program structure is altered with syscalls added and deleted. Each structurally changed program will further go through several modification loops in which the syscall arguments and distribution among the threads are mutated. Finally, each modified program runs repeatedly for several times, each with a different delay schedule, to probe for alias coverage.

Several implicit parameters can be used to fine-tune the process, *e.g.*, how many times to loop at each stage (see §B for details). In general, we give preference to alias coverage exploration over growing the multi-threaded syscall sequences, as we prefer to explore the concurrency domain as much as possible when the number of syscalls executed is small, making it easier for kernel developers to debug a reported data race.

Offline checking. Data race checking is conducted offline, *i.e.*, only when new coverage, either branch or alias, is found. The reason is that data race checking is slow (several minutes) and significantly hinders the fast fuzzing experience (which only requires a few seconds to finish one execution). As a result, we allow the fuzzers to quickly expand coverage and only dump execution logs without checking them. A few background threads check the execution logs for data races whenever they have free capacity. The checking progress has difficulty keeping up with seed generation in the beginning but will gradually catch up, especially when the coverage is toward saturation.

B. Benign vs harmful data races

An unexpected problem we encountered when reporting the data races found by KRACE is on differentiating benign and

harmful data races. Despite the common belief that being data-race free is one of the coding practices in the kernel, benign data races are not totally uncommon. One major category is statistics accounting, such as `__part_stat_add` in the block layer. These statistics are meant for information and hints only and do not provide any accuracy guarantees. Another example is the reading and writing of different bits in the same 2-, 4-, 8-byte variable, especially bit-flags such as `inode->i_flag` or flags in file system control structures like `fs_info`.

Based on our experience, checking whether a data race is benign or harmful is often time consuming, as it requires careful analysis of the code and documentation to infer developers' intentions. In the worst cases, it may require consulting the file system developers, who may not even agree among themselves. One possibility to confirm a harmful data race is to keep the system running until the data race causes any visible effects such as violating assertions or memory errors. However, this is not always feasible, as shown in the case in Figure 1. It might need thousands of file operations running in parallel to trigger an integer overflow. By then, debugging such an execution trace will be another problem.

To avoid reporting benign data races to developers, KRACE uses several simple heuristics to filter the reports. In particular, a data race is mostly benign if:

- the race involves variables that have `stat` in their names or occurs within functions for statistics accounting;
- the race involves reading and writing to different bits of the same variable;
- the race involves kernel functions that can tolerate being racy, *e.g.*, `list_empty_careful`.

Unfortunately, these heuristics typically offer limited help for the more complicated cases.

C. The aging OS problem

When fuzzing file systems, most generic OS fuzzers do not reload a fresh copy of the kernel instance or file system image [21–23] for a new fuzzing session. Instead, they directly issue the syscall sequence on the old kernel state. The intention is to remove the overhead of kernel booting, as a VM emulator might take seconds to load and boot the kernel, as is evident in our evaluations as well (§VII-B). However, this also means that any bugs found in this approach might come from the accumulated effects of hundreds or even thousands of prior runs, making them extremely difficult to debug and confirm by kernel developers, as is evident in the case when many bugs found by Syzkaller cannot be confirmed [67].

The aging OS problem is already difficult for fuzzing in the sequential domain, and bringing in the concurrency dimension further complicates the story. Moreover, for KRACE, the aging OS situation creates more problems, as the lengthy thread interleaving traces are not only difficult to debug but also renders analysis impossible. Slicing the execution traces does not seem feasible either, as cutting the trace at the wrong points means losing the locking and happens-before context, ultimately leading to false alarms. As a result, KRACE is forced to use a clean-slate execution for every fuzzing run, *i.e.*, a fresh kernel and a clean file system image.

The aging OS problem is also reported by Janus [5], which uses a library OS—LKL [68]—to enable quick reloading. But unfortunately, LKL does not support the symmetrical multi-processing (SMP) architecture, which is the prerequisite for multi-threading (*e.g.*, without SMP, all `spin_locks` becomes no-ops). As a result, LKL is mostly suitable for sequential fuzzing, not for concurrency fuzzing.

D. Discussion and limitations

Deterministic replay. Being able to replay an execution deterministically is extremely helpful for debugging and also opens the door for advanced data race triaging techniques such as controlled re-interleaving of thread executions. Unfortunately, we are sorry to report that even with a totally linearized trace of basic block enter/exit, memory accesses, lock acquisition/releases, and kernel synchronization API calls, KRACE is unable to deterministically replay an execution end-to-end. Part of the reason is the missing instrumentation in other kernel components, including the kernel core (including the task and IO scheduler), memory management, device drivers (except the block device), and most of the library routines. We expect that deterministic replay may be possible if we instrument all kernel components but at the expense of huge execution footprints (*e.g.*, GB-level logs) as well as significant performance drops. We are unaware of a system that permits deterministic replay of over 60 kernel threads, but we are eager to integrate if possible.

Debuggability. To partially compensate for not being able to replay a found data race deterministically, KRACE tries to generate a comprehensive report for each data race, including 1) the conflicting lines in source code, 2) the full call stack for each thread, and 3) the callback graph. Since each instruction is labeled with a compile-time random number, KRACE is able to pinpoint the conflicting lines in the source code when a data race occurs. Further coupled with the basic block branching information, KRACE is able to recover the full call trace, up to the syscall entry point or the thread creation point, for all involving threads during the race condition. The report may also involve the callback graph derived from the happens-before analysis, to further assist the developers with the origin of the threads. In fact, kernel developers have never asked for a deterministic replay of the trace and are able to judge whether the race is harmful or benign based on the information provided.

Missing bugs. Offlining the data race checker means that KRACE might miss data race bugs. As discussed in §III-B, alias coverage is just an approximation of state exploration progress in the concurrency dimension, and there might be new program states explored at runtime but that do not show up as new coverage, *i.e.*, meaningful interleavings missed by alias coverage. KRACE forgoes the opportunities to check data races in those cases and is a trade-off made in favor of expanding the coverage with efficiency.

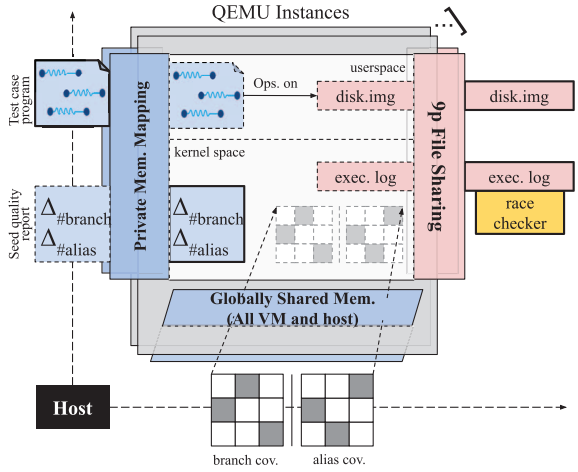

Fig. 10: Implementation of the QEMU VM-based fuzzing executor in KRACE. The VM instance and the host have three communication channels: 1) private memory mapping, which contains the test case program to be executed by the VM and the seed quality report generated by KRACE runtime; 2) globally shared memory mapping, which contains the coverage bitmaps globally available to the host and all VM instances; 3) file sharing under the `9p` protocol for sharing of large files, including the file system image and the execution log.

E. Implementation

KRACE's code base is divided into two parts: 1) compile-time preparation, including annotations to the kernel source code (in the form of kernel patches), an LLVM instrumentation pass, and the KRACE library compiled into the kernel that provides coverage tracking and logging at runtime; and 2) a VM-based fuzzing loop that evolves test cases, executes them in QEMU VMs, and checks for data races. The complexity of each component is described in Table III and an overview of the runtime executor is shown in Figure 10. Due to space constraints, more details can be found in §D.

VII. EVALUATION

In this section, we evaluate KRACE as a whole as well as per each component. In particular, we show the overall effectiveness of KRACE by listing previously unknown data races found (§VII-A); provide a comprehensive view of KRACE's performance characteristics, *e.g.*, speed, scalability, etc., as a file system fuzzer (§VII-B); justify major design decisions with controlled experiments (§VII-C); and compare KRACE against recent OS and data race fuzzers (§VII-D).

Experiment setup. We evaluate KRACE on a two-socket, 24-core machine running Fedora 29 with Intel Xeon E5-2687W (3.0GHz) and 256GB memory. All performance evaluations are done on Linux v5.4-rc5, although the main fuzzer runs intermittently across versions from v5.3. We build the kernel core with minimal components but enable as many features as possible for the `btrfs` and `ext4` file system modules. For all evaluations, the fuzzing starts with an empty file system image created from the `mkfs.*` utilities. We run 24 VM instances in parallel for fuzzing and each VM runs a three-thread seed.

ID	FS	Racing access	Status
1	btrfs	heap struct: `cur_trans->state`	pending
2	btrfs	heap struct: `cur_trans->aborted`	harmful
3	btrfs	heap struct: `delayed_rsv->full`	harmful
4	btrfs	heap struct: `sb->s_flags`	benign
5	btrfs	global variable: `buffers`	harmful
6	btrfs	heap struct: `inode->i_mode`	benign
7	btrfs	heap struct: `inode->i_atime`	harmful
8	btrfs	heap struct: `BTRFS_I(inode)->disk_i_size`	harmful
9	btrfs	heap struct: `root->last_log_commit`	harmful
10	btrfs	heap struct: `free_space_ctl->free_space`	benign
11	btrfs	heap struct: `cache->item.used`	harmful
12	ext4	heap struct: `inode->i_mtime`	benign
13	ext4	heap struct: `inode->i_state`	benign
14	ext4	heap struct: `ext4_dir_entry_2->inode`	benign
15	ext4	heap array: `ei->i_data[block]`	harmful
16	VFS	heap string: `name` in `link_path_walk`	pending
17	VFS	heap struct: `inode->i_state`	benign
18	VFS	heap struct: `inode->i_wb_list`	benign
19	VFS	heap struct: `inode->i_flag`	benign
20	VFS	heap struct: `inode->i_opflags`	benign
21	VFS	heap struct: `file->f_mode`	benign*
22	VFS	heap struct: `file->f_pos`	pending
23	VFS	heap struct: `file->f_ra.ra_pages`	harmful

TABLE I: List of data races found and reported by KRACE so far. Status of benign* means that it is a benign race according to the execution paths we submitted, but the kernel developers suspect that there might be other paths leading to potentially harmful cases.

A. Data races in popular file systems

Across intermittent fuzzing runs on two popular kernel file systems (`btrfs` and `ext4`) during two months, KRACE found and reported 23 new data races, of which nine have been confirmed to be harmful, 11 are benign, and the rest of them are still under investigation, as listed in Table I. Note that besides bugs in concrete file systems, KRACE also finds data races in the virtual file system (VFS) layer, which might affect all file systems in the kernel.

Consequence. Based on our preliminary investigation, only one bug (#5) is likely to cause immediate effects (null-pointer dereference) when triggered. Others are likely to cause performance degradation or specification violations, but we do not see a simple path toward memory errors. This also means that relying on bug signals such as KASan reports or kernel panics might not be sufficient to find data races.

B. Fuzzing characteristics

Coverage growth. The growth patterns for both branch and alias coverage are plotted in Figure 11 (for `btrfs`) and Figure 12 (for `ext4`). There are several interesting observations:

Alias coverage size. Although branch coverage for the two file systems grow into roughly the same level (25K vs 20K), compared with `ext4`, `btrfs` has a significantly larger alias coverage bitmap, (60K vs 9K). Given that the number of user threads is the same (3 threads), the difference is caused by the level of concurrency inherent in `btrfs` and `ext4` design. As shown in Figure 14, `btrfs` uses at least 22 background threads and each thread may additionally fork more helper threads, while the only background thread for `ext4` is the `jbd2` journaling thread. In other words, `btrfs` is inherently more concurrent than `ext4`, and dividing works among more

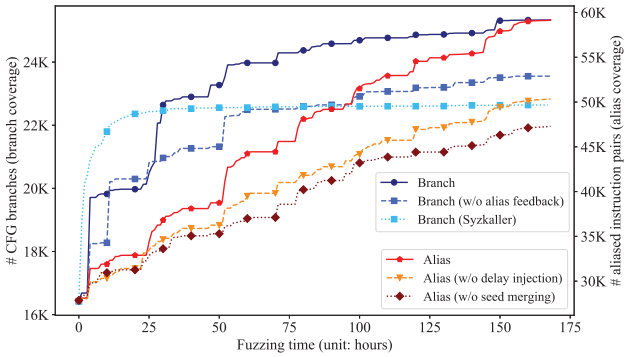

Fig. 11: Evaluation of the coverage growth of KRACE when fuzzing the `btrfs` file system for a week (168 hours) with various settings.

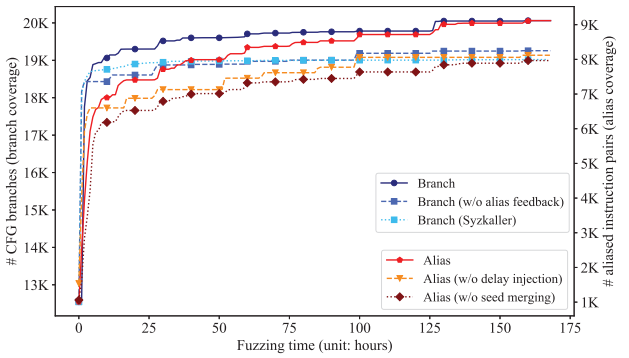

Fig. 12: Evaluation of the coverage growth of KRACE when fuzzing the `ext4` file system for a week (168 hours) with various settings.

threads naturally leads to more alias pairs. The similar logic also applies to why alias coverage saturates much faster in `ext4`, the less concurrent file system.

Growth synchronization. In general, the two coverage metrics grow in synchronization. It is expected that progresses in the branch coverage will yield new alias coverage too because new code paths mean new memory accessing instructions and hence, new alias pairs. However, it is the other direction that matters more: branch coverage saturates but alias coverage keeps growing, *e.g.*, starting from hour 75 in the `btrfs` case or hour 25 in the `ext4` case. In other words, KRACE keeps finding new execution states (thread interleavings) that would otherwise be missed if only branch coverage is tracked.

Instrumentation overhead. The code instrumentation from KRACE is heavy, and we expect it to cause significant overhead in execution. To show this, we present the aggregated statistics on the execution time for seeds bearing different numbers of syscalls. For comparison, we also run these seeds on a bare-metal kernel built without KRACE instrumentation. The results are plotted in Figure 13. In summary, in the zero-syscall case, *i.e.*, by merely loading (file system module) → mounting (image) → unmounting → unloading, KRACE already incurs 47.6% and 34.3% overhead, and the more syscalls KRACE executes, the more overhead it accumulates.

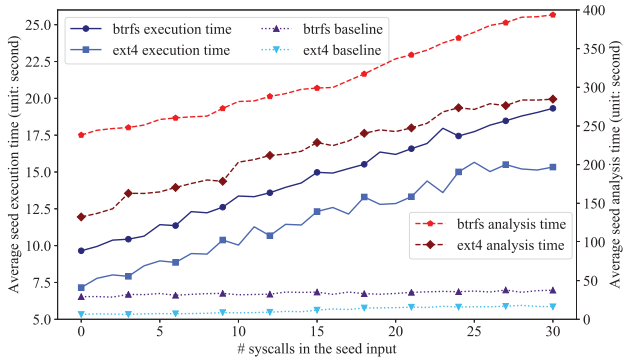

Fig. 13: Evaluation of seed execution and analysis time in KRACE with a varying number of syscalls in the seed

The overhead mainly comes from memory access instrumentation, as every memory access is now turned into a function call where atomic operations are performed and synchronized, not only with respect to all other threads on the VM, but also against all threads across all VMs, as the thread is updating the global bitmap on the host directly (implicitly handled by the QEMU `ivshmem` module). As a result, further optimizations are possible. For example, a VM instance may accumulate coverage locally and update the global bitmap in batches instead of on every memory access.

It is, however, debatable whether the overhead is detrimental to KRACE as a fuzzer since lower overhead simply means that the coverage growth will converge and saturates faster. In our opinion, we consider the overhead caused by tracking more coverage (including alias coverage) as a trade-off between execution speed and seed quality. A fuzzer with fast executions may waste resources in non-interesting test cases, while a fuzzer with slow executions but finer-grained tracking might eventually have higher chances to explore more states.

Data race checking cost. Another limiting factor for KRACE is the time needed to analyze the execution logs for data race detection, which also depends on the length of the execution trace. The trend is also plotted in Figure 13. In summary, the analysis time ranges from 4-7 minutes (0-30 syscalls per seed) for `btrfs` and 2-6 minutes for `ext4`. Such a time cost is obviously not feasible for online checking (even after optimization) but can be tolerated for offline checking, *i.e.*, KRACE schedules a data race check only when a seed is discovered. This strategy works especially when fuzzing saturates, as the bottleneck for making further progress then becomes finding new execution states instead of checking the trace. Based on our experience, running four checker processes alongside 24 fuzzing VM instances is more than sufficient to catch up to the progress within 96 hours in both cases.

C. Component evaluations

Coverage effectiveness. Although the two coverage metrics represent different aspects of program execution, we are also curious whether tracking explorations in the concurrency dimension may help in finding new code paths (represented by

branch coverage). To check this, we disabled the alias coverage feedback and let KRACE explore the states mimicking the feedback loop of existing OS and file system fuzzers. The results (Figure 11 and Figure 12) show that exploring the concurrency domain also helps to find new code coverage. Most notably, without alias coverage feedback, branch coverage grows much faster at the beginning, because it does not spend fuzzing effort on exploring the thread interleavings, but saturates at a lower number (7.2% and 4.0% less). Moreover, if just counting the new branches explored (besides the branches in the initial seed), the coverage reduces by 20.4% and 10.7%, respectively. The more concurrent the file system is, the more branch coverage will be explored by enabling alias coverage feedback. This is not surprising, as certain code paths exist to handle contention in the system, such as the paths executed when `try_lock` fails or when sequence lock retries. Exploring in the concurrency dimension helps to reveal these paths and boost the branch coverage.

Delay injection effectiveness. To test whether injecting delays helps in exploration in the concurrency dimension, we disabled delay injection in this fuzzing experiment, and the alias coverage growth is shown in Figure 11 and Figure 12. With delay injection disabled, KRACE found 28.7% and 12.3% less alias coverage in `btrfs` and `ext4`, respectively. This shows that delay injection is important in finding more alias coverage. Especially, when the branch coverage saturates, delay injection becomes the leading force in finding alias coverage, as shown by the enlarging gap between the growth. The more concurrent the file system is, the more important delay injection becomes.

Seed merging effectiveness. To test whether reusing the seed helps in exploration in the concurrency dimension, we disabled seed merging in this fuzzing experiment, *i.e.*, KRACE only adds, deletes, and mutates syscalls but never reuses the found seeds. The alias coverage growth is shown in Figure 11 and Figure 12. With seed merging disabled, KRACE found 37.7% and 14.2% less alias coverage in `btrfs` and `ext4`, respectively. This experiment shows that reusing the seed is important in quickly expanding the coverage. More importantly, preserving the semantics among the syscalls and interleaving the seeds help find more alias coverage.

Components in the data race checker. To show that it is important to have both happens-before and lockset analysis (and their sub-components) in the data race checker, we sampled a simple fuzzing run: load `btrfs` module, mount an empty image, execute two syscalls × three threads, unmount the image, and unload the `btrfs` module. The following shows the filtering effects of each component in the data race checker:

- data race candidates: 35,658
+ after lockset analysis on pessimistic locks: 13,347
+ after lockset analysis on optimistic locks: 8,903
+ after tracking fork-style happen-before relation: 6,275
+ after tracking join-style happen-before relation: 3,509
+ after handling publisher-subscriber model: 103
+ after handling ad-hoc schemes: 7 (all benign races)

D. Comparison with related fuzzers

Execution speed vs coverage. In terms of efficiency, KRACE is not comparable to other OS and file system fuzzers, as one execution takes at least seven seconds in KRACE, while the number can be as low as 10 milliseconds for libOS-based fuzzers [5, 6] or never-refreshing VM-based fuzzers like Syzkaller. However, the effectiveness of a fuzzer is not solely decided by fuzzing speed. A more important metric is the coverage size, especially when saturated. Intuitively, if the saturated coverage is low, being fast in execution only implies that the coverage will converge faster and mostly stall afterward.

On the metric of saturated coverage, KRACE outperforms Syzkaller for both `btrfs` and `ext4` by 12.3% and 5.5%, respectively, as shown in Figure 11 and Figure 12. Even without the alias coverage feedback, the branch coverage from KRACE still outperforms Syzkaller, showing the effectiveness of KRACE's seed evolution strategies, especially the merging strategy for multi-threaded seeds, which is currently not available in Syzkaller. In fact, KRACE is able to catch up to the branch coverage progress with Syzkaller within 30 hours and eight hours for `btrfs` and `ext4`, respectively.

Data race detection. Razzer [24] reports four data races in file systems and we find the patches for two of them, both in the VFS layer. To check that KRACE may detect these cases, we manually revert the patches in the kernel and confirm that both cases are found. We would like to do the same for SKI [7], but the data races found by SKI are too old (in 3.13 kernels) and locating and reverting the patches is not easy.

VIII. CONCLUSION AND FUTURE WORK

This paper presents KRACE, an end-to-end fuzzing framework that brings the concurrency aspects into coverage-guided file system fuzzing. KRACE achieves this with three new constructs: 1) the alias coverage metric for tracking exploration progress in the concurrency dimension, 2) the algorithm for evolving and merging multi-threaded syscall sequences, and 3) a comprehensive lockset and happens-before modeling for kernel synchronization primitives. KRACE has uncovered 23 new data races so far and will keep running for more reports.

Looking forward, we plan to extend KRACE in at least three directions: 1) data race detection in other kernel components; 2) semantic checking for more types of concurrency bugs; and 3) fuzzing distributed file systems that involve not only thread interleavings but also network event ordering, which requires completely new coverage metrics to capture.

IX. ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd, Yan Shoshitaishvili, for their insightful feedback. This research was supported, in part, by NSF under award CNS-1563848, CNS-1704701, CRI-1629851, SFS-1565523, and CNS-1749711; ONR under grant N00014-18-1-2662, N00014-15-1-2162, and N00014-17-1-2895; DARPA TC (No. DARPA FA8650-15-C-7556); ETRI IITP/KEIT[2014-3-00035]; and gifts from Facebook, Mozilla, Intel, VMware, and Google.

References

[1] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," *Trans. Storage*, vol. 10, no. 1, pp. 3:1–3:32, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2560012

[2] J. Huang, M. K. Qureshi, and K. Schwan, "An Evolutionary Study of Linux Memory Management for Fun and Profit," in *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Berkeley, CA, USA, Jun. 2016, pp. 465–478.

[3] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, "File Systems Unfit As Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: ACM, Oct. 2019, pp. 353–369.

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

[5] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing File Systems via Two-Dimensional Input Space Exploration," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[6] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, Oct. 2019.

[7] P. Fonseca, R. Rodrigues, and B. B. Brandenburg, "SKI: Exposing Kernel Concurrency Bugs Through Systematic Schedule Exploration," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[8] MITRE Corporation, "CVE-2009-1235," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1235, 2009.

[9] J. Corbet, "Unprivileged filesystem mounts, 2018 edition," https://lwn.net/Articles/755593, 2018.

[10] Kernel.org Bugzilla, "Btrfs bug entries," https://bugzilla.kernel.org/buglist.cgi?component=btrfs, 2018.

[11] MITRE Corporation, "F2FS CVE entries," http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=f2fs, 2018.

[12] Kernel.org Bugzilla, "ext4 bug entries," https://bugzilla.kernel.org/buglist.cgi?component=ext4, 2018.

[13] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient Scalable Thread-safety-violation Detection: Finding Thousands of Concurrency Bugs During Testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: ACM, Oct. 2019, pp. 162–180.

[14] Silicon Graphics Inc. (SGI), "(x)fstests is a filesystem testing suite," https://github.com/kdave/xfstests, 2018.

[15] SGI, OSDL and Bull, "Linux Test Project," https://github.com/linux-test-project/ltp, 2018.

[16] M. Zalewski, "American Fuzzy Lop (2.52b)," http://lcamtuf.coredump.cx/afl, 2019.

[17] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[18] Google Inc., "honggfuzz," http://honggfuzz.com/, 2019.

[19] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[20] Google, "OSS-Fuzz - Continuous Fuzzing for Open Source Software," https://github.com/google/oss-fuzz, 2018.

[21] Google Inc., "Syzkaller is an Unsupervised, Coverage-guided Kernel Fuzzer," https://github.com/google/syzkaller, 2019.

[22] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[23] NCC Group, "AFL/QEMU Fuzzing with Full-system Emulation," https://github.com/nccgroup/TriforceAFL, 2017.

[24] D. R. Jeong, K. Kim, B. A. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding Kernel Race Bugs through Fuzzing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[25] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[26] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface Aware Fuzzing for Kernel Drivers," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[27] D. Jones, "Linux system call fuzzer," https://github.com/kernelslacker/trinity, 2018.

[28] R. N. Netzer and B. P. Miller, "Detecting Data Races in Parallel Program Executions," in *In Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*. MIT Press, 1989, pp. 109–129.

[29] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997.

[31] M. D. Bond, K. E. Coons, and K. S. McKinley, "PACER: Proportional Detection of Data Races," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, Jun. 2010, pp. 255–268.

[32] Z. Anderson, D. Gay, R. Ennals, and E. Brewer, "SharC: Checking Data Sharing Strategies for Multithreaded C," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, Jun. 2008, pp. 149–158.

[33] E. Pozniansky and A. Schuster, "Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs," in *Proceedings of the 9th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. New York, NY, USA: ACM, Jun. 2003, pp. 179–190.

[34] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective Sampling for Lightweight Data-race Detection," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, Jun. 2009, pp. 134–143.

[35] P. McKenney, "The RCU API, 2019 edition," https://lwn.net/Articles/777036/, 2019.

[36] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places," in *Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, Mar. 2009, pp. 25–36.

[37] K. Sen, "Race Directed Random Testing of Concurrent Programs," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, Jun. 2008, pp. 11–21.

[38] Y. Cai, J. Zhang, L. Cao, and J. Liu, "A Deployable Sampling Strategy for Data Race Detection," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 810–821.

[39] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data Race Detection in Practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: ACM, 2009, pp. 62–71.

[40] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric, "Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers (T)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 166–177.

[41] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[42] S. Hong and M. Kim, "Effective Pattern-driven Concurrency Bug Detection for Operating Systems," *J. Syst. Softw.*, vol. 86, no. 2, pp. 377–388, Feb. 2013.

[43] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*. Stevenson, WA: ACM, Oct. 2007, pp. 103–116.

[44] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: Static Race Detection on Millions of Lines of Code," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT*

Symposium on The Foundations of Software Engineering, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 205–214.
[45] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective Data-race Detection for the Kernel," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Berkeley, CA, USA: USENIX Association, Oct. 2010, pp. 151–162.
[46] M. Elver, "Add Kernel Concurrency Sanitizer (KCSAN)," https://lwn.net/Articles/802402/, 2019.
[47] J. Alglave, W. Deacon, B. Feng, D. Howells, D. Lustig, L. Maranget, P. E. McKenney, A. Parri, N. Piggin, A. Stern, A. Yokosawa, and P. Zijlstra, "Who's afraid of a big bad optimizing compiler?" https://lwn.net/Articles/793253/, 2019.
[48] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs," in *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, Mar. 2010, pp. 167–178.
[49] Y. Sui and J. Xue, "SVF: Interprocedural Static Value-Flow Analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction (CC)*, Barcelona, Spain, Mar. 2016.
[50] LLVM Project, "libFuzzer - a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, 2018.
[51] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.
[52] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.
[53] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.
[54] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.
[55] NCC Group, "AFL/QEMU fuzzing with full-system emulation." https://github.com/nccgroup/TriforceAFL, 2017.
[56] MWR Labs, "Cross Platform Kernel Fuzzer Framework," https://github.com/mwrlabs/KernelFuzzer, 2016.
[57] H. Han and S. K. Cha, "IMF: Inferred Model-based Fuzzer," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.
[58] NCC Group, "A linux system call fuzzer using TriforceAFL," https://github.com/nccgroup/TriforceLinuxSyscallFuzzer, 2017.
[59] MWR Labs, "macOS Kernel Fuzzer," https://github.com/mwrlabs/OSXFuzz, 2017.
[60] M. J. Renzelmann, A. Kadav, and M. M. Swift, "SymDrive: Testing Drivers without Devices," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.
[61] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. Checker: A Soundy Analysis for Linux Kernel Drivers," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.
[62] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics," in *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, Mar. 2008.
[63] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs," in *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, PA, Mar. 2010.
[64] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. New York, NY, USA: Cambridge University Press, 2016.
[65] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 1–15. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/wang
[66] K. Owens and A. Arcangeli, "Seqlock implementation in linux," https://github.com/torvalds/linux/blob/master/include/linux/seqlock.h, 2019.
[67] Google, "syzbot," https://syzkaller.appspot.com, 2018.
[68] O. Purdila, L. A. Grijincu, and N. Tapus, "LKL: The Linux kernel library," in *Proceedings of the 9th Roedunet International Conference (RoEduNet)*. IEEE, 2010.
[69] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma, "Ad Hoc Synchronization Considered Harmful," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

APPENDIX

A. Level of concurrency in the btrfs file system

```
struct btrfs_fs_info {
    /* work queues */
    struct btrfs_workqueue *workers;
    struct btrfs_workqueue *delalloc_workers;
    struct btrfs_workqueue *flush_workers;
    struct btrfs_workqueue *endio_workers;
    struct btrfs_workqueue *endio_meta_workers;
    struct btrfs_workqueue *endio_raid56_workers;
    struct btrfs_workqueue *endio_repair_workers;
    struct btrfs_workqueue *rmw_workers;
    struct btrfs_workqueue *endio_meta_write_workers;
    struct btrfs_workqueue *endio_write_workers;
    struct btrfs_workqueue *endio_freespace_worker;
    struct btrfs_workqueue *submit_workers;
    struct btrfs_workqueue *caching_workers;
    struct btrfs_workqueue *readahead_workers;
    struct btrfs_workqueue *fixup_workers;
    struct btrfs_workqueue *delayed_workers;
    struct btrfs_workqueue *scrub_workers;
    struct btrfs_workqueue *scrub_wr_completion_workers;
    struct btrfs_workqueue *scrub_parity_workers;
    struct btrfs_workqueue *qgroup_rescan_workers;
    /* background threads */
    struct task_struct *transaction_kthread;
    struct task_struct *cleaner_kthread;
};
```

Fig. 14: 20 work queues and 2 background threads used by `btrfs`. This does not cover all asynchronous activities observable at runtime.

B. Seed evolution in KRACE

```
def fuzzing_loop(ext_limit, mod_limit, rep_limit):
    while True:
        program = merge_seeds(select_seed_pair())

        ext_stall = 0
        while ext_stall < ext_limit:
            ext_stall++
            [50%] program.add_syscall()
            [50%] program.del_syscall()

            mod_stall = 0
            while mod_stall < mod_limit:
                mod_stall++
                [80%] program.mutate()
                [20%] program.shuffle()

                rep_stall = 0
                while rep_stall < rep_limit:
                    rep_stall++
                    delay = randomize_delay()
                    cov, log = run(program, delay)

                    if not cov.empty():
                        rep_stall = mod_stall = ext_stall = 0
                        schedule_data_race_check(log)
                        prune_and_save_seed(program)
```

Fig. 15: The seed evolution process (a.k.a the fuzzing loop) in KRACE

Three parameters tunes the behaviors of the seed evolution loop: namely `ext_limit`, `mod_limit`, and `rep_limit` as shown in Figure 15. In KRACE, they take the values of 10, 10, and 5 respectively. That is,

- if any new coverage, either branch or alias, is observed in 5 consecutive runs, KRACE will continue to run the same multi-threaded seed for 5 more times but with a new delay schedule each time;
- if no new coverage is observed for 5 consecutive runs, KRACE starts to mutate the syscall arguments in the multi-threaded trace or shuffle the syscalls;
- if no new coverage is observed for 50 consecutive runs, KRACE starts to alter the input structure by adding or deleting the syscalls in the multi-threaded traces;
- if no new coverage is observed for 500 consecutive runs, KRACE starts to merge two seeds for a new seed.

C. Ad-hoc synchronization schemes in kernel file systems

Although ad-hoc synchronization schemes are considered harmful [69], they may still exist in kernel file systems for performance or functionality enhancements. Whenever we encounter an ad-hoc scheme (usually when analyzing false positives), we annotate it in the same way as major synchronization APIs so that subsequent runs will not report the false data races caused by it. In this section, we present two examples we encountered in `btrfs`.

Ad-hoc locking. An ad-hoc lock has two implications: 1) there will be data races in the lock implementation and these data races are all benign races; and 2) lock internals should be abstracted in a way that the lockset analysis can easily understand. A representative example is the `btrfs` tree lock, and the purpose of having the tree lock is to be convertible between blocking and non-blocking mode, as shown in Figure 16.

```
 /* acquire a spinning write lock, wait for both
  * blocking readers or writers */
void btrfs_tree_lock(struct extent_buffer *eb)
{
    u64 start_ns = 0;
    if (trace_btrfs_tree_lock_enabled())
        start_ns = ktime_get_ns();

    WARN_ON(eb->lock_owner == current->pid);
again:
    wait_event(eb->read_lock_wq,
           atomic_read(&eb->blocking_readers) == 0);
    wait_event(eb->write_lock_wq, eb->blocking_writers == 0);
    write_lock(&eb->lock);
    if (atomic_read(&eb->blocking_readers)
            || eb->blocking_writers) {
        write_unlock(&eb->lock);
        goto again;
    }
    btrfs_assert_spinning_writers_get(eb);
    btrfs_assert_tree_write_locks_get(eb);
    eb->lock_owner = current->pid;
}
/* drop a spinning or a blocking write lock. */
void btrfs_tree_unlock(struct extent_buffer *eb)
{
    int blockers = eb->blocking_writers;
    BUG_ON(blockers > 1);

    btrfs_assert_tree_locked(eb);
    eb->lock_owner = 0;
    btrfs_assert_tree_write_locks_put(eb);

    if (blockers) {
        btrfs_assert_no_spinning_writers(eb);
        eb->blocking_writers--;
        cond_wake_up(&eb->write_lock_wq);
    } else {
        btrfs_assert_spinning_writers_put(eb);
        write_unlock(&eb->lock);
    }
}
```

Fig. 16: A snippet of the `btrfs` tree lock (writer side only).

Tree lock API	Lockset mapping
tree_lock	writer-lock
tree_unlock	writer-unlock
tree_read_lock	reader-lock
tree_read_lock_atomic	reader-lock
tree_read_unlock	reader-unlock
tree_read_unlock_blocking	reader-unlock
tree_set_lock_blocking_read	no-op if read-locked
tree_set_lock_blocking_write	no-op if write-locked
try_tree_read_lock	reader-lock if succeed
try_tree_write_lock	writer-lock if succeed

TABLE II: Semantic mapping between the tree lock and conventional locks (in particular, the readers-writer lock).

In these functions, almost every memory access to the fields in the extent buffer, `eb`, could be racing against other accesses. *e.g.*, `eb->lock_owner` at line 12 against `eb->lock_owner = 0` at line 40. So the first annotation for KRACE is to assume all data races within these functions are safe and benign races.

To further encode the locking semantics for lockset analysis, we study the tree lock APIs and map their functionality into a simple reader-writer lock format as shown in Table II. In other words, calling the, *e.g.*, `tree_lock` will be treated equally as calling the writer-lock in the conventional locking mechanisms. Although `tree_lock` performs much more computation (*e.g.*, waiting for both blocking and non-blocking readers), from the lockset perspective, it is equivalent to a writer-lock.

Ad-hoc ordering. Ad-hoc ordering implies undocumented casual relations between thread executions and a good example is the customization of the conventional kernel work queue in `btrfs`, as shown in Figure 17.

```
1  static inline void __btrfs_queue_work(struct __btrfs_workqueue *wq,
2          struct btrfs_work *work)
3  {
4      unsigned long flags;
5      work->wq = wq;
6      if (work->ordered_func) {
7          spin_lock_irqsave(&wq->list_lock, flags);
8          list_add_tail(&work->ordered_list, &wq->ordered_list);
9          spin_unlock_irqrestore(&wq->list_lock, flags);
10     }
11     queue_work(wq->normal_wq, &work->normal_work);
12 }
13 static void normal_work_helper(struct btrfs_work *work) {
14     /* ... */
15     work->func(work);
16     if (need_order)
17         set_bit(WORK_DONE_BIT, &work->flags);
18     /* ... */
19 }
20 static void run_ordered_work(struct __btrfs_workqueue *wq) {
21     /* ... */
22     work = list_entry(list->next, struct btrfs_work, ordered_list);
23     if (test_bit(WORK_DONE_BIT, &work->flags))
24         work->ordered_func(work);
25     /* ... */
26 }
```

Fig. 17: A snippet of the `btrfs` work queue implementation.

In this example, the `set_bit` and `test_bit` (line 17 and 23), establish an additional causal relation beyond the normal `queue_work` semantic: the ordered function only gets into execution when the normal function finishes. Thus, although the observed happens-before relation is line 8 → line 24 and line 11 → line 15, the actual relation is line 8 → line 11 → line 15 → line 24.

D. KRACE *implementation details*

Component	LoC	Languange
Compile-time preparation		
Kernel annotations	5,653	C
LLVM instrumentation pass	1,977	C++
KRACE kernel runtime library	1,749	C
Fuzzing loop		
Seed evolution (including syscall spec.)	9,394	Python
QEMU-based fuzzing executor	5,878	Python
Initramfs and the `init` program	2,527	Python
Data race checker	6,883	Python
Debugging tools and utilities	1,096	Python

TABLE III: Implementation complexity of KRACE in terms of LoC measurement of the major components shown in Figure 9.

Runtime executor. The most challenging part of KRACE's implementation is to establish information-sharing channels between the host and VM-based fuzzing instances for seed injection, coverage tracking, and feedback collection. KRACE uses private memory mapping (PCI memory bar), public memory mapping (`ivshmem`), and the `9p` file sharing protocols for this purpose, as shown in Figure 10.

Kernel building. Building the Linux kernel with LLVM is straightforward since kernel v5.3 and LLVM 9.0. In addition, to get the smallest possible boot time, we opt for a minimal kernel build with only necessary components enabled, including the block layer, loopback device, and all other related drivers to support and accelerate execution in QEMU and KVM. File systems are built as modules, not built-in, and these modules will be loaded by our fuzzing agent (*i.e.*, the `init` program) such that we could track the modules in full, including the thread they fork on loading and their synchronization orders.

Initramfs. Again, to shorten the execution time, KRACE does not rely on full-blown OSes, not even tools like `busybox`, as they may interfere with the file system under testing. Instead, the `init` program in KRACE is the fuzzing agent that takes the multi-threaded seed and interprets it. In particular, the `init` 1) starts tracing, 2) loads file system modules, 3) mounts the file system image, 4) interprets the program, 5) unmounts the file system image, 6) unloads the modules, and 7) stops tracing.

Coverage tracking. Coverage tracking is handled by the instrumented code which are essentially stub calls, *e.g.*, `on_basic_block_enter`, `on_memory_read`, etc., into the KRACE runtime library. KRACE directly updates the coverage bitmaps maintained in the host memory regions that are globally visible to all VM instances (and their threads). Effectively, each update is a `test_and_set_bit` operation while the QEMU `ivshmem` protocol ensures atomicity.

Execution log. An execution log is simply an array of `[<event-type>, <thread-id>, <arg1>, <arg2>, ...]` filled by the KRACE runtime library and consumed by the data race checker for data race detection as well as reporting purposes such as call trace reconstruction.

E. A taste of the happens-before complexity in actual execution

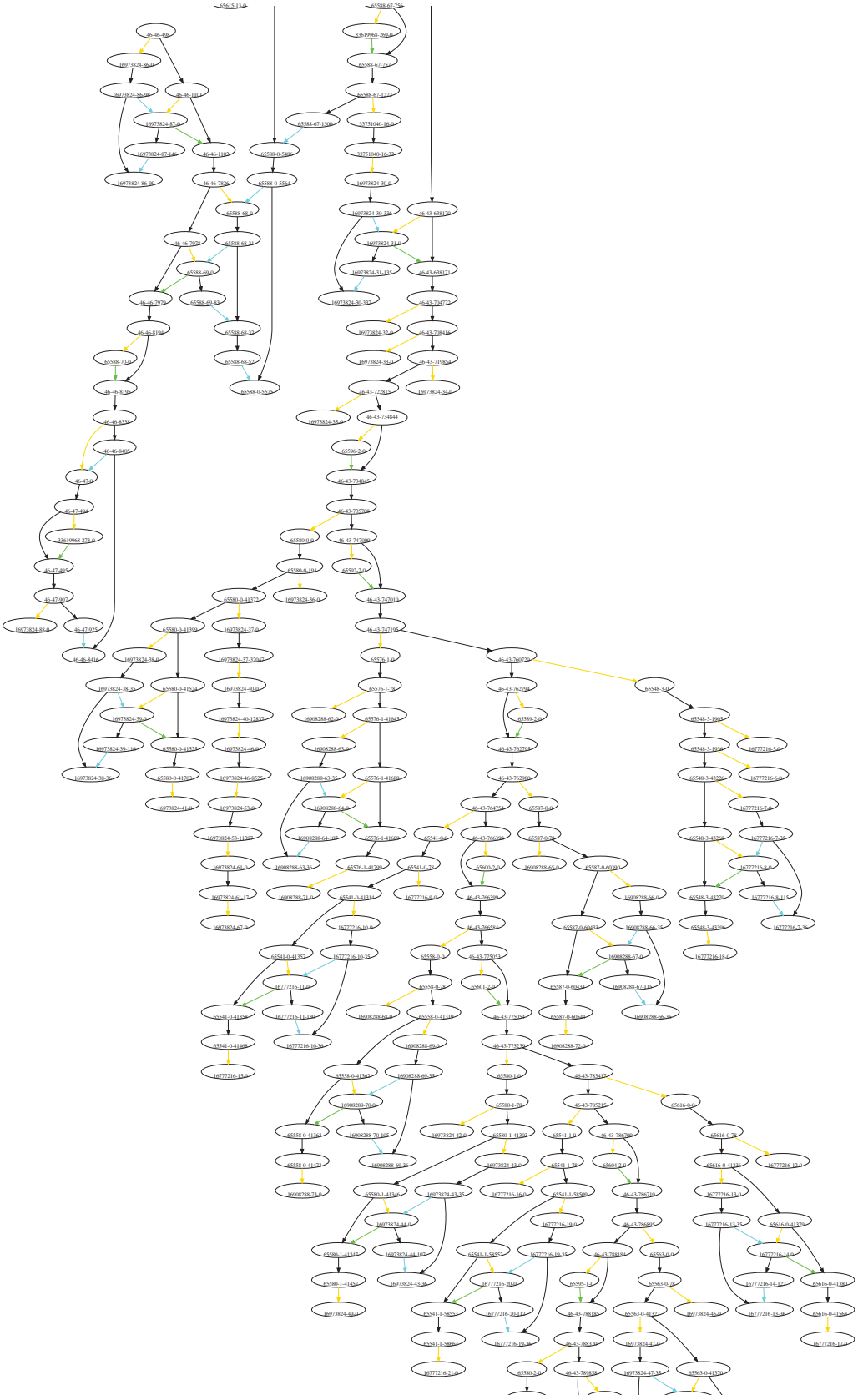

Fig. 18: A taste of the happens-before relation tracking in `btrfs` file system. This snippet is only around 10% of the actual happens-before graph tracked in this execution. Each node in the graph is a synchronization point represented by a three-tuple <`thread id, context id, instruction id`> and the directed edge between two nodes `A` and `B` means `A` happens-before `B`.