Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis

Sishuai Gong
Purdue University
USA

Deniz Altınbüken
Google Research
USA

Pedro Fonseca
Purdue University
USA

Petros Maniatis
Google Research
USA

Abstract

Kernel concurrency bugs are challenging to find because they depend on very specific thread interleavings and test inputs. While separately exploring kernel thread interleavings or test inputs has been closely examined, jointly exploring interleavings and test inputs has received little attention, in part due to the resulting vast search space. Using precious, limited testing resources to explore this search space and execute just the right concurrent tests in the proper order is critical.

This paper proposes Snowboard a testing framework that generates and executes concurrent tests by intelligently exploring thread interleavings and test inputs jointly. The design of Snowboard is based on a concept called *potential memory communication (PMC)*, a guess about pairs of tests that, when executed concurrently, are likely to perform memory accesses to shared addresses, which in turn may trigger concurrency bugs. To identify PMCs, Snowboard runs tests sequentially from a fixed initial kernel state, collecting their memory accesses. It then pairs up tests that write and read the same region into candidate concurrent tests. It executes those tests using the associated PMC as a scheduling hint to focus interleaving search only on those schedules that directly affect the relevant memory accesses. By clustering candidate tests on various features of their PMCs, Snowboard avoids testing similar behaviors, which would be inefficient. Finally, by executing tests from small clusters first, it prioritizes uncommon suspicious behaviors that may have received less scrutiny.

Snowboard discovered 14 new concurrency bugs in Linux kernels 5.3.10 and 5.12-rc3, of which 12 have been confirmed by developers. Six of these bugs cause kernel panics and filesystem errors, and at least two have existed in the kernel for many years, showing that this approach can uncover hard-to-find, critical bugs. Furthermore, we show that covering as many distinct pairs of uncommon read/write instructions as possible is the test-prioritization strategy with the highest bug yield for a given test-time budget.

CCS Concepts: • **Security and privacy** → **Operating systems security**; • **Software and its engineering** → **Concurrency control**; **Software testing and debugging**.

ACM Reference Format:
Sishuai Gong, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. 2021. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–28, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3477132.3483549

1 Introduction

Kernel developers employ fine-grained concurrency to achieve high performance in the multi-core era [10, 34, 73]. This includes implementing parallel algorithms [39, 45, 59, 98], reducing locking granularity [55, 88], and adopting optimistic concurrency-control schemes (e.g., RCU [46, 48, 68]). However, these optimizations are notoriously error-prone and easily lead to hard-to-find concurrency bugs [14, 17, 81].

In practice, kernel concurrency bugs have serious impact on users [33, 42, 89] by causing kernel panics [36, 37], data loss [59] and enabling privilege escalation attacks [11, 16, 74]. Furthermore, a recent study [54] shows attackers can reliably trigger concurrency bugs from user-space—bugs that rarely happen by accident can happen almost deterministically by adversarial attacks. Thus, finding concurrency bugs is crucial for building a reliable and safe kernel.

Automatically finding kernel concurrency bugs is particularly challenging for several reasons. First, kernels are huge—the Linux kernel currently approaches 30 million source-code

SOSP '21, October 26–28, 2021, Virtual Event, Germany
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483549

lines [53]—and have complex interfaces with more than 400 system calls [49]. Second, concurrency bugs typically require the execution of at least two threads with very specific inputs. Third, concurrency bugs are only triggered on specific thread interleavings, requiring automated techniques to explore the vast interleaving space. Hence, the input space is at least quadratic in the number of sequential tests because at least two threads must be tested together, and exponential in the number of instructions that can be interleaved in each test. Exhaustive search is intractable. Consequently, the problem warrants a systematic concurrency-testing approach that navigates the search space intelligently.

These challenges limit fuzzing [31, 44] and stress testing [3] effectiveness at finding inputs that expose hard-to-find kernel concurrency bugs. In particular, common kernel fuzzers, such as Syzkaller [31], are mostly designed to generate test inputs for sequential execution. These fuzzers generally use straightforward approaches, such as providing the same input to several threads or splitting inputs across threads, to generate concurrent tests, without control over thread interleavings. A naïve algorithm extension could randomly pair distinct sequential tests into a concurrent test, but given the search-space size, such non-targeted approaches would have a low chance of finding hard-to-hit bugs.

Different approaches have been proposed to test kernels for concurrency bugs dynamically but have limitations. For example, tools relying on static data-race analysis are imprecise and miss concurrency bugs that are not caused by data races (e.g., atomicity violations) [43, 62, 92]. Other tools only focus on exploring the interleaving space, requiring manual or ad-hoc input generation [27] (see §7). In practice, none of the existing tools consider all classes of kernel concurrency bugs while also exploring the input and interleaving search spaces jointly and interdependently.

This paper proposes Snowboard, a testing framework that systematically generates and prioritizes concurrent tests and associated interleavings through heuristics, to find kernel concurrency bugs efficiently. The design of Snowboard relies on the core insight that individual kernel API operations (i.e., system-call invocations) tend only to execute a relatively small amount of kernel code. Hence, we expect that the potential interactions between different threads can be predicted offline by analyzing the memory accesses of each thread when executed independently and sequentially. In turn, this lets Snowboard determine which concurrent tests should be executed, and under what interleaving, to explore suspicious non-deterministic behavior.

Snowboard has to achieve two goals to accomplish its objective: 1) *the generation of tests* likely to trigger concurrency bugs, and 2) *the prioritization of generated tests* to exercise uncommon inter-thread communications that are unlikely to be observed in other testing or production environments.

Snowboard must reduce the search-space size across combinations of inputs, behaviors, and interleavings to generate concurrency tests. It starts with a corpus of sequential (single-threaded) tests (provided by a traditional fuzzing tool), which it executes sequentially and independently, collecting memory accesses induced by each. It then groups pairs of tests—a *writer* and a *reader*—that access the same memory location, thereby constituting a *potential memory communication (PMC)*, that is, a data-flow channel from the writer to the reader that *may* be triggered. This channel is *not* purely aspirational, as would happen with an imprecise static analysis; instead, as long as the two tests run concurrently with the same memory layout as during profiling, and with an interleaving that schedules the writer's write instruction before the reader's read instruction, this data-flow *will* occur, subject to any synchronization that (hopefully) occurs, or (sadly) does not. A PMC, if triggered in otherwise unsynchronized or incorrectly synchronized code, and under an unfortunate interleaving that, say, interposes the writer's write to invalidate the reader's state invariants before the reader reads and uses the shared value, can lead to a concurrency bug. Such concurrent tests (i.e., the sequential tests for the writer and reader and a scheduling hint that triggers the communication) constitute the concurrency test corpus generated by Snowboard.

Snowboard must choose tests that will cover as many of the execution behaviors of the system as possible to prioritize generated concurrency tests. Traditional fuzzing approaches systematize exploration using structural-coverage metrics, such as control-flow edge coverage [31, 50, 70] and def-use coverage [60, 80, 94], which were also extended to the concurrent case, e.g., with instruction-pair coverage [92]. We generalize this coverage intuition to not only test selection but also test execution as guided by PMCs.

We applied Snowboard to mature versions of Linux, including stable versions and release candidates. In total, Snowboard discovered 14 concurrency bugs, of which nine were found in a stable version of the kernel. Of the bugs found, four are non-data-race concurrency bugs that have serious impact. For instance, two of the non-data-race concurrency bugs cause kernel panics, and one causes filesystem errors. Some of the bugs found were insidious enough to require more than lock-guarding a variable access. In addition, our results revealed that some of these bugs had been present for several months and, in some cases, more than three years. After we reported them, twelve have been confirmed by developers and six have been fixed already [19, 29, 30, 41, 86, 90].

This paper makes the following contributions:

- **Predicting interactions between threads**. This paper proposes an approach, scalable to kernels, to predict possible interactions between sequential tests when executed in parallel.

- **Systematizing concurrency testing**. Snowboard uses a set of PMC-based clustering strategies that select PMCs with similar properties, thus having a similar effect in causing concurrency bugs, and a search strategy that prioritizes PMCs less likely to be covered by typical, production testing.
- **Finding kernel concurrency bugs**. Snowboard found 14 kernel concurrency bugs so far and continues to find more. Its artifact is publicly available[1].

2 Background and Motivation

2.1 Kernel Concurrency Testing

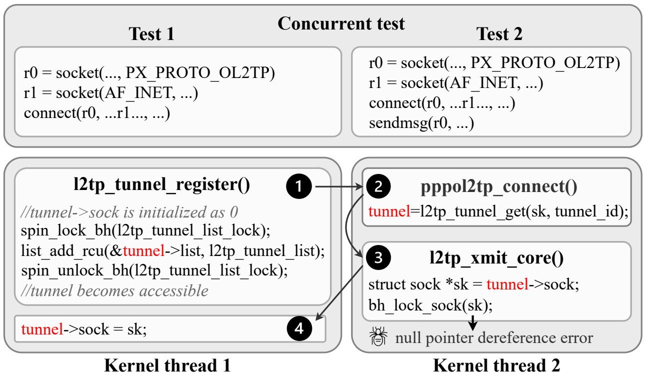
Although most kernel testing focuses exclusively on sequential tests, some work has also explored concurrency testing. However, none has studied concurrent test input generation that targets all classes of kernel concurrency bugs.

Razzer [43] generates concurrent tests to find kernel data races, which are responsible for the *subset of concurrency bugs* that happen in the absence of appropriate synchronization (§2.2). To identify possible data race instruction pairs, Razzer employs static data race analysis, which is prone to false positives, and then attempts to pair sequential tests that execute both instructions of the suspected data races. Compared to Razzer, Snowboard performs a dynamic analysis on kernel sequential tests to identify all memory-based inter-thread communication, regardless of whether they represent data races or not.

Similar to Razzer, Krace [92] is a fuzzing framework that finds data race-inducing test cases, but Krace focuses on file systems. It automatically generates concurrent tests using mutation techniques and specific coverage metrics to guide test generation based on file system specifications. However, Krace concurrent tests generated do not include scheduling hints, i.e., target interleavings that should be tested. Hence, Krace needs to explore a (very large) interleaving space for every shared memory access triggered by the test. In contrast, Snowboard generates concurrent tests with target interleavings and tests both data race and non-data races concurrency bugs.

Furthermore, unlike prior work, Snowboard studies prioritizing concurrent kernel test inputs, which is vital considering the huge PMC (and data race) space. Snowboard heuristically prioritizes concurrent tests and associated interleavings, significantly reducing the search space (§5.3).

Another line of work focuses on kernel interleaving exploration. DataCollider [21] detects data races by sampling kernel memory accesses and randomly delaying them using hardware watchpoints. SKI [27] focuses on achieving systematic kernel schedule exploration by generalizing the PCT algorithm [8] and requires an external source of concurrent tests that specify kernel input (e.g., fstress [3]). Unlike Snowboard, these tools do not jointly consider the kernel input and interleaving spaces, so their testing effectiveness largely depends on the quality of provided concurrent inputs.



```
r0 = socket(..., PX_PROTO_OL2TP)
r1 = socket(AF_INET, ...)
connect(r0, ...r1..., ...)
```

```
r0 = socket(..., PX_PROTO_OL2TP)
r1 = socket(AF_INET, ...)
connect(r0, ...r1..., ...)
sendmsg(r0, ...)
```

```
l2tp_tunnel_register()
//tunnel->sock is initialized as 0
spin_lock_bh(l2tp_tunnel_list_lock);
list_add_rcu(&tunnel->list, l2tp_tunnel_list);
spin_unlock_bh(l2tp_tunnel_list_lock);
//tunnel becomes accessible

tunnel->sock = sk;
```

```
pppol2tp_connect()
tunnel=l2tp_tunnel_get(sk, tunnel_id);
```

```
l2tp_xmit_core()
struct sock *sk = tunnel->sock;
bh_lock_sock(sk);
```

Figure 1. A non-data-race concurrency bug in the network stack found by Snowboard. Bug causes a kernel null pointer deference error. #12 in Table 2.

2.2 Potential Memory Communication (PMC)

During concurrent execution of two kernel threads[2], thread A may affect the behavior of another thread B if thread A updates a shared memory location that is later read by thread B. For a PMC between threads to occur, 1) thread A has to make a write memory access, 2) thread B has to make a read memory access, 3) the memory regions of the two accesses must overlap, and 4) the write access by thread A has to update the memory area with a different value from what the read access by thread B would have fetched if thread B ran sequentially. Note that inter-thread PMCs occur *regardless of synchronization*. Hence, the definition of PMC is unrelated to data races, which occur when a pair of data memory accesses are not synchronized.

When the write of a PMC interferes with the reader's read, the reader's subsequent execution may change drastically, potentially unmasking a concurrency bug. However, pairs of write accesses that update the same memory may also lead to bugs if the result is eventually read. Since such situations still require a read after a write to occur, PMCs are general enough to capture all classes of memory-level non-determinism induced by instruction schedules.

A Case Study. Figure 1 illustrates a PMC and how it may lead to a concurrency bug (in fact, it is a bug found by our system). At the top of the figure, two user-space processes execute two tests concurrently, involving different system calls. Note that we are not considering user-space shared accesses here; the two user processes are isolated in distinct user address spaces but operate on top of the same kernel. The kernel then services the two processes via two kernel threads—the writer on the left and the reader on the right—which execute in the shared kernel address space.

[1] Snowboard artifact: https://github.com/rssys/snowboard

[2] Concurrent execution of three or more threads is discussed in §6.

The reader attempts to fetch a previously registered tunnel (in `pppol2tp_connect()`), which it then uses to transmit, in `l2tp_xmit_core()`. If, however, the reader's retrieval of the tunnel occurs right after the writer has registered a new tunnel (in `l2tp_tunnel_register()`) and before the writer has initialized the socket field of the tunnel (❶→❷→❸), the reader will retrieve a tunnel with an uninitialized `sock` field (❹), which will cause a null pointer dereference when transmitting.

The PMC occurs between inserting the freshly allocated `tunnel` into the `l2tp_tunnel_list` structure on the writer's side, and the read from the tunnel list of the partially uninitialized `tunnel` structure on the reader's side. Note that an RCU lock protects the tunnel list; however, this lock fails to guarantee what seems to be the reader's invariant: that the tunnel list always contains fully initialized tunnel structures.

This concurrency bug is hard to find through random exploration. The two particular tests chosen may come from a large corpus of sequential tests that combine socket communications and PPP tunnels; not all such tests will happen to register a new tunnel. Among those tests that do, due to the sequential corpus generation, some may happen to cause the same tunnel ID to be retrieved, while others may not.

This concurrency bug is also hard to find even with the assistance of static analysis. The analysis would have to determine that the `tunnel` variable in the writer may alias (i.e., refer to the same address as) the `tunnel` variable in the reader, which is challenging and imprecise when pointers and lookup structures (i.e., the tunnel list) are involved [5, 67]. In addition, the analysis may deem that the initialization of the `sock` field (in `l2tp_tunnel_register()`) could race with the read of `sock` (in `l2tp_xmit_core()`), which is another PMC involved in this bug. However, simply generating a concurrent input that executes both instructions of the data race (e.g., Razzer) would likely fail to trigger it because the two memory accesses will only visit the same `tunnel` when the writer creates a `tunnel` and then the reader retrieves the newly created `tunnel`. Thus, the key to exposing this bug is still the PMC (❶→❷) between `tunnel` registration and retrieval.

It is also instructive to note that the actual address of the `tunnel` structure—or the tunnel ID it corresponds to—matters little when generating concurrent tests, as long as the reader and writer "agree" on the structure to jointly access. If multiple tests exercise this shared access (e.g., because other system calls preceded the creation of the tunnel), but at different `tunnel` locations or with even different read/write instruction addresses, they are likely to trigger the same null pointer dereference. An intelligent strategy to choose concurrent tests could deprioritize tests that exercise the same or very similar behavior; however, given that a concurrent test might not in fact exercise a latent PMC, this deprioritization may need to be balanced with more concurrent tests that do exercise different, "similar" PMCs.

3 Goals and Approach

3.1 Problem Definition

Snowboard aims to generate concurrency tests likely to uncover concurrency bugs in software, such as the Linux kernel (see §6 for a discussion of generality). It assumes the following capabilities:

- An external tool produces a corpus of *sequential tests*: these are self-sufficient snippets of code that set up and perform several system operations, such as system calls. This includes code to set up some inputs into initial buffers and execution of logic.
- An *execution framework* runs chosen tests—either sequential tests from the corpus above or concurrent tests consisting of two sequential tests and an interleaving schedule.
- A *bug detector* monitors executions and identifies system failures (e.g., kernel panics, data races, deadlocks).

Given the above, the goals of Snowboard are as follows:

- *Construct concurrent tests*, each including a pair of sequential tests.
- *Prioritize concurrent tests* to increase the efficient use of the execution framework.
- *Execute concurrent tests* to exercise and trigger potentially dangerous concurrent behavior.

The design of Snowboard draws inspiration from the following principles:

- *Potential memory communications (PMCs)*—dynamic information flow from memory writes by one thread into subsequent memory reads by another thread—are predictive of shared memory accesses.
- *Similar PMCs cover similar behaviors* and this similarity can help prune the search space.
- *Uncommon memory channels*—PMCs that rarely occur within a corpus of tests are likely to exhibit concurrency bugs that are not encountered often or tested extensively.
- *A PMC can be viewed as a scheduling hint* and interleaving exploration should focus on instructions involved in potential shared-memory communication.

Our definition of success is 1) finding concurrency bugs that existing tools cannot find or have not found in a long time, and 2) finding those bugs faster.

3.2 Snowboard Design Overview

We now present how Snowboard acts on its goals above.

3.2.1 Identifying Kernel PMCs.

Snowboard uses a hybrid dynamic analysis on sequential tests. It executes them and profiles their memory accesses. It then identifies PMCs between pairs of sequential tests based on their memory-access profiles. Thus, two sequential tests that are likely to have shared memory accesses can be identified via PMCs, and explored with interleavings influenced by their PMCs.

Note that this design choice relies on the ability of Snowboard to *reproduce* those same memory accesses when two, formerly sequential, tests run concurrently. Snowboard employs checkpoint-based replay to encourage this reproducibility.

3.2.2 Cluster PMCs by Sensitive Behavior Covered. PMCs that share certain common characteristics—e.g., access the same memory range, use the same instruction addresses or read/write the same value—may lead to similar buggy behavior. Snowboard defines a *clustering strategy*, which selects some PMC features to cluster them by. One exemplar PMC from each cluster is then tested, assuming that the remaining PMCs would exhibit similar behavior and uncover no new bugs. The choice of clustering strategy is critical.

3.2.3 Prioritize Uncommon PMCs. Finding PMCs that are uncommon (i.e., occur less frequently) is another challenge because authoritative information could only be obtained via intrusive memory tracing in production use. Existing approaches [8, 71] that find uncommon interleavings within a concurrent test are dependent on the process that generates the corpus (both the paired tests and their interleaving), and may not reflect frequency in a production setting.

Snowboard capitalizes on the insight that PMC rarity in a test corpus can approximate bug-prone behavior rarity in production. By not considering an interleaving at all—and assuming that the execution framework can trigger an interleaving that will exercise a PMC—we can focus on counting uncommon PMCs types (e.g., clusters, as defined above) to rank concurrency tests for execution. Although still not necessarily reflective of rarity in a production setting, this approximation captures a proxy feature of uncommon but possible PMCs, which may warrant testing.

3.2.4 Use PMCs as Scheduling Hints. Snowboard executes a test by inducing thread yields at instructions where a PMC access is about to happen or just happened. This approach focuses the scheduling exploration only on the instructions that affect the PMC and thus encourages shared memory accesses without exhaustively searching all interleavings of the two threads under test.

4 Snowboard Architecture

Snowboard employs a pipelined architecture involving four major stages, as summarized in Figure 2. Snowboard first executes a corpus of kernel sequential tests and profiles their execution, starting from a reproducible and consistent kernel state (§4.1). Afterward, it gathers all profiled shared memory accesses from every sequential test and identifies PMCs in the kernel by selectively examining pairs of write and read accesses that touch common memory addresses (§4.2). Snowboard then prunes and prioritizes the testing of uncommon PMCs using a set of heuristic clustering strategies (§4.3). Finally, Snowboard executes generated concurrent tests, exploring interleavings that target the associated PMC (§4.4).

4.1 Sequential Test Generation and Profiling

Snowboard requires an external tool that generates sequential tests. Any high-quality test generator based on fuzzing, static analysis, or heuristics would do. Snowboard uses a coverage metric exported by the generator (e.g., edge coverage) to select a subset of the generated tests that provide high coverage but low overlap of exercised behaviors.

After generating a comprehensive set of distinct sequential tests, Snowboard dynamically profiles each test by recording a memory trace of the corresponding sequential kernel execution, collecting memory accesses—address range accessed, type of access, value read/written—and corresponding instruction addresses.

If tests were executed from arbitrary kernel states, memory traces collected in this fashion would be of limited use. To reduce such non-determinism, Snowboard runs all sequential tests from the same, *fixed initial kernel state*. In particular, Snowboard profiles sequential tests from a virtual machine snapshot that is taken after the target kernel boots and launches two test executor processes that run on two different vCPUs. Any further kernel configuration or setup specific to a test is considered part of the test itself, rather than encoded in the initial kernel state; in that sense, a much broader set of kernel states are reachable before some sensitive sequence of system calls are executed. However, given an upper limit on sequential test length, some initial kernel states may not be reachable in this fashion; in such cases, Snowboard can grow the number of initial kernel states it utilizes to increase diversity.

Using a snapshot has two advantages. First, *profiling* every sequential test from the same state allows Snowboard to reason about the PMCs of different tests, identifying potential memory-access overlap. Second, Snowboard uses the same starting state to execute generated *concurrent* tests. In most cases, this means that the two threads under test will access overlapping memory areas, exercising the PMC under an appropriate interleaving.

Note that some PMCs will not be exercised under *any* interleaving when the two relevant threads are executed concurrently. For instance, if the writer properly protects a buffer from concurrent access before writing into it, the reader may select a different buffer to read from (e.g., by retrieving the front of a queue of buffer pointers). However, by constructing tests that *might* exercise the PMC, we encourage unsafe concurrent behaviors to arise if they exist.

4.1.1 Implementation Details. Our current implementation generates sequential tests using Syzkaller [31], a state-of-the-art feedback-based kernel fuzzing tool. Instead of using

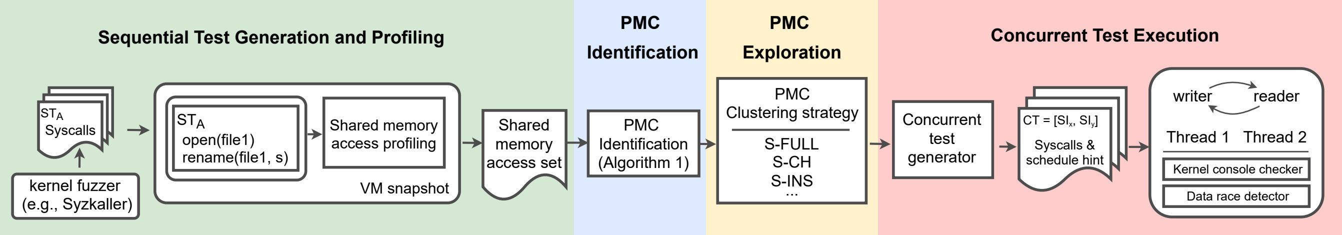


Figure 2. Design overview of Snowboard.

every single test produced by the sequential test generator, which typically yields a very large number of redundant tests produced through random mutations, Snowboard uses the edge coverage metric, exported by Syzkaller, to select tests.

Snowboard emulates the guest machine using a customized hypervisor. During the sequential test execution, Snowboard first records every guest memory access and then uses the `CR3` register to filter out accesses made by other irrelevant threads. To identify potential memory accesses to shared memory space, Snowboard makes the standard assumption [21, 57, 95] that only non-stack accesses are potentially *shared*.

Snowboard leverages the ESP register to prune memory accesses that are not deemed shared. Snowboard computes the kernel stack range of the target thread by reading the ESP register. For example, in Linux kernel x86, each kernel thread has a fixed size of 8KB (2 physical pages) for the stack region, and the stack is 8KB aligned. Thus, Snowboard can compute the kernel stack range $[ESP \land \neg(STACK\_SIZE - 1), (ESP \land \neg(STACK\_SIZE - 1) + STACK\_SIZE]$. A similar approach is used by the Linux kernel function `current_thread_info()` to identify the stack region. Excluding these memory accesses from the analysis avoids predicting PMCs that are destined not to happen across kernel stack memory accesses and increases the overall scalability of the PMC analysis (§4.2).

We implemented a user-space test suite running in the guest machine that executes sequential tests generated by Syzkaller, and communicates with the Snowboard hypervisor for testing actions via hypercalls: starting sequential or concurrent tests and transferring test data and test results between the host and the guest. Furthermore, we implement a memory-access analyzer inside the hypervisor to profile the target kernel thread.

4.2 PMC Identification

Snowboard identifies PMCs by analyzing the memory accesses of sequential tests. Snowboard first gathers all shared memory accesses profiled from every sequential test and indexes them by the memory range they access. Then, for all pairs of reads and writes with overlapping memory ranges, it designates a pair as a PMC if the values written to and read from the shared memory range differ.

Algorithm 1 PMC identification.

Input: $\mathcal{T}$: Profiled sequential tests.
Output: C: All PMCs along with the tests exhibiting them.

```
 1: ▷ Index all sequential tests.
 2: $\mathcal{A} = \emptyset$                                  ▷ Tests indexed by memory range.
 3: for t ∈ $\mathcal{T}$ do
 4:     for a ∈ t.accesses do
 5:         $\mathcal{A}$.insert($a_{mem}$, $a_{access\_type}$, $a_{val}$, $a_{instr}$, t)
 6: ▷ Scan memory-range overlaps and identify PMCs.
 7: $C = \emptyset$                                  ▷ PMCs indexed by access features.
 8: for o ∈ $\mathcal{A}$.read_write_overlaps() do
 9:     read_value = project_value(o.$read_{mem}$, o.$read_{val}$, o.range)
10:     write_value = project_value(o.$write_{mem}$, o.$write_{val}$, o.range)
11:     if read_value ≠ write_value then
12:         read_key = (o.$read_{mem}$, o.$read_{instr}$, o.$read_{val}$)
13:         write_key = (o.$write_{mem}$, o.$write_{instr}$, o.$write_{val}$)
14:         PMC = (read_key, write_key)
15:         $C$[PMC].append(o.$read_{test}$, o.$write_{test}$)
```

Algorithm 1 describes this process in pseudocode. Lines 1–5 index the memory accesses of all sequential tests and then lines 6–15 find and filter overlaps. Indexing iterates over all tests and all accesses in each test and puts each test in an index structure, recording the test itself, the memory range accessed, the access type (read/write), the value read or written, and the instruction address. Note that a single test may appear in multiple entries in this structure, if it incurs multiple memory accesses. This index is then queried for read and write access pairs with overlapping memory ranges—we capture this lookup with the `read_write_overlaps` method on line 8. Every overlap returned contains a read test and its relevant read access, and a write test and its relevant write access. The corresponding value read/written is projected to the overlapping memory range (o.range) on Lines 9 and 10. If the projected read/write values differ, we classify this as a PMC, and store it indexed by the memory ranges, instruction addresses, and values of both the read and write accesses. Multiple pairs of read/write tests may map to the same PMC key.

4.2.1 Implementation Details. Locating overlapped access pairs requires a nested scan over all possible pairs in the naïve case, which scales quadratically with our sequential test corpus size.

Snowboard uses an ordered nested index to implement the $\mathcal{A}$ structure in Algorithm 1. The outer index is ordered by start address. Given a start address, a nested index orders accesses by range length. Finally, given a specific range, accesses are indexed by the operation instruction address.

Although more sophisticated structures exist for efficient interval searches, given the size of our corpus and the target usage—scanning and locating *all* pairs with overlap—this approach is efficient in space and computation.

4.3 PMC Selection

Due to the high kernel complexity, PMC identification typically generates a large number[3] of PMCs. As testing all of them is not feasible—each test needs to set up a VM, load the fixed kernel snapshot, execute with various interleavings, store findings, repeat—a systematic and efficient search algorithm that selects PMCs for testing is needed. Our approach is based on our two guiding insights (§3.2): a) cluster *approximately equivalent* PMCs by some clustering criterion and b) choose an exemplar PMC from each cluster, from the least to the most common cluster. Intuitively, this strategy avoids running multiple tests that are likely to lead to the same kind of buggy or benign behavior—hence the clustering—and favors tests from smaller clusters. PMCs from smaller clusters could be regarded as uncommon among all predicted PMCs, so exercising them is likely to trigger behaviors not often seen in production, or not well tested.

Clustering of PMCs is done using a *clustering strategy*, which consists of a *clustering key*, and a *filter*. PMCs with the same clustering key belong in the same cluster under the strategy, but some clusters may be altogether discarded by the filter. Both keys and filters are expressed in terms of PMC features; the features we collect (Algorithm 1) are the reading/writing instruction addresses ($\text{ins}_{r/w}$), the read/written values ($\text{value}_{r/w}$), the read/written memory-range start addresses ($\text{addr}_{r/w}$), and the read/written memory-range lengths in bytes ($\text{byte}_{r/w}$); recall that memory ranges overlap by PMC construction, but may not be identical.

We define 8 heuristic clustering strategies, supported by intuition gleaned from our inspection of many concurrency bugs. While a clustering key that constructs large clusters reduces the size of the search space, it may misrepresent two PMCs as equivalent, dismissing one of them, even though it might uncover distinct misbehaviors. Note also that some clustering strategies throw PMCs away, so they might best be combined with others that partition the search space. We evaluate the effectiveness of each in §5.3.2. We describe them in detail next and define them formally in Table 1.

Clustering strategy	Clustering Key / [*Filter Predicate*]
S-FULL	(ins_w, addr_w, byte_w, value_w, ins_r, addr_r, byte_r, value_r) / [*True*]
S-CH	(ins_w, addr_w, byte_w, ins_r, addr_r, byte_r) / [*True*]
S-CH NULL	(ins_w, addr_w, byte_w, ins_r, addr_r, byte_r) / [$value_w$=*0*]
S-CH UNALIGNED	(ins_w, addr_w, byte_w, ins_r, addr_r, byte_r) / [*(*$addr_r$ *!=* $addr_w$ *or* $byte_r$ *!=* $byte_w$*)*]
S-CH DOUBLE	(ins_w, addr_w, byte_w, ins_r, addr_r, byte_r) / [*df_leader*]
S-INS	($\text{ins}_{w/r}$) / [*True*]
S-INS-PAIR	(ins_w, ins_r) / [*True*]
S-MEM	(addr_w, byte_w, addr_r, byte_r) / [*True*]

Table 1. The PMC clustering strategies we consider. Each is expressed in terms of a clustering key, and a filter. Both keys and filters refer to PMC features: instruction ins, memory-range start address addr, memory-range length byte, read/written value, and the special `df_leader` boolean indicating the first of a double-fetch read access (§4.3).

S-FULL: Full Communication. *S-FULL* considers all PMC features to cluster PMCs, which means only identical PMCs will be assigned to the same cluster. This predicate is a baseline since it yields the largest number of clusters and PMCs to test, and is thus the costliest.

S-CH: Channel. During the development of Snowboard, we observed that many PMCs share the same instructions and memory ranges, but have different read/written values. This situation is common when shared objects are pointers or counters that are accessed frequently with different values. Often these PMCs only cause (at most) one bug. Therefore the *Channel* strategy uses all features except the access values.

S-CH-NULL: Object Nullification Channel. A limitation of *S-CH* is that it considers all values equivalent. However, some values have special meanings. For example, bugs often arise when a writer zeroes out a shared object and the reader dereferences this value as an address. Thus, we treat the all-zero write value as a special case in the *S-CH-NULL* strategy.

S-CH-UNALIGNED: Unaligned Channel. This is another special case of *S-CH*, where the write and read access ranges differ (start from different addresses or have different lengths). Bugs of this kind occur because the reader fetches partially updated data, breaking object invariants.

S-CH-DOUBLE: Double Fetch Channel. Double-fetch vulnerabilities often cause kernel concurrency bugs [87, 93]. They lead to a time-of-check-to-time-of-use bug that occurs, for instance, when the kernel reads a given user-space data

[3] We identified over 169 billion PMCs in Linux kernel 5.12-rc3.

location first to verify access, and then again to use the user-space object, assuming that the two values read are identical. A concurrent update between the two reads leads to severe bugs such as privilege escalation.

Snowboard introduces `df_leader`, a special boolean PMC feature, to capture double fetches. During sequential test analysis, it sets this feature when it finds that two read accesses by different instructions occur sequentially with no intervening write access of the same memory region, and the values read are identical. The feature is set on the first of the two read accesses.

S-INS: Instruction. An unsynchronized access, whether a write or read, can cause a bug regardless of its counterpart instruction. For example, an unsynchronized write may clobber the reads of multiple read instructions. This strategy pair (one for reads and one for writes) clusters solely on the instruction address.

S-INS-PAIR: Instruction Pair. Extending on the intuition above, in some cases, a specific write-read instruction pair is the sole bug cause. For example, a lock-protected writer may communicate with several lock-protected readers, but only a single unprotected reader helps it cause a bug.

S-MEM: Memory region. Shared memory objects are stored at varying memory addresses. PMCs that communicate over the same memory area may have the same effect on the kernel, benign or buggy. For example, performance counters in the kernel are often not synchronized, as developers chose performance over strong semantics [21]. This strategy assumes each overlapped memory region holds a unique kernel shared object, which is reasonable since Snowboard always uses the same fixed kernel state.

Given a clustering strategy choice, Snowboard clusters all PMCs, counts the cardinality of each cluster, and then selects the exemplar to test from each cluster, from the least populous—less common—to the most populous cluster. Note that this approach can be applied iteratively: Choose predicate A, test one exemplar from each A-cluster, then choose predicate B, test one exemplar from each B-cluster excluding those tested before, etc. Furthermore, it is possible to use one strategy to subdivide large clusters produced by another.

4.4 Concurrent Test Execution

During test execution, one PMC is chosen from each cluster in uncommon-to-common cluster order. A PMC may correspond to multiple test pairs (recall Line 15 in Algorithm 1); one pair is chosen among them at random, to construct a concurrent test.

A Snowboard concurrent test differs from traditional ones in that it includes a *scheduling hint*: a PMC designating a write and read memory access in the respective writer/reader threads that should be explored. The execution framework performs multiple actual executions of the pair of tests exploring interleavings relevant to the PMC.

Algorithm 2 Execution exploration.

Input: $\mathcal{C}$: PMCs identified in Algorithm 1.
Output: $\mathcal{R}$: Tests that triggered the bug detectors.

```
for cluster ∈ ordered_pmc_clusters(C, STRATEGY) do
    pmc = draw_from_cluster(cluster, random)
    flags = ∅
    for trial ∈ NUMBER_OF_TRIALS do
        random.seed(SEED + trial)                    ▷ Always same randomness in trial.
        current_pmcs.add(pmc)
        last_access[reader] = last_access[writer] = None
        resume_snapshot()
        while !test_end() do                         ▷ Execution loop for single trial.
            if !is_live(current_thread) then
                switch = True
            if switch then
                yield()
            switch = False
            for access ∈ execute(next_ins) do        ▷ Monitor memory accesses.
                if pmc_access_coming(access, flags) then
                    switch = random()
                if performed_pmc_access(access) then
                    previous_access = last_access[current_thread]
                    flags.add(previous_access)
                    switch = random()
                last_access[current_thread] = access
                accesses.add(access)                 ▷ Instruction finished.
            if is_bug() then
                R.record(scheduling_decisions, accesses, pmc)
        incidental_pmcs = find_new_pmc(accesses.write, accesses.read)    ▷ Trial done.
        current_pmcs.add(random_choice(incidental_pmcs))
```

The scheduling component of the execution framework attempts to satisfy multiple goals: 1) trigger the PMC, 2) also *do not* trigger the PMC!, 3) obtain meaningful executions (e.g., avoid deadlocks or livelocks), 4) opportunistically explore other co-incident PMCs that may be observed during execution of the test pair, to amortize the execution cost towards covering more PMCs (similarly to other work [27, 92]). We present pseudocode for the testing loop in Algorithm 2, which we describe below.

To achieve its goals, Snowboard uses some execution primitives: `yield` switches execution from the reader to the writer or vice versa (ignoring other kernel threads that may be running at the time), `is_live` heuristically detects if a thread is making progress, rather than being blocked, say due to a deadlock, `performed_pmc_access` is an alert from the execution framework indicating that the current instruction made a PMC memory access (read or write by the corresponding thread), `pmc_access_coming` is a similar execution alert indicating that the current instruction *is likely to make a PMC memory access* (see below), and `is_bug` detects if a bug has been triggered. Some randomness is also involved, so a pseudo-random number generator makes non-deterministic decisions.

`pmc_access_coming` bears some explanation: Snowboard observes the last access right before a PMC access and remembers it for future trials of the concurrent test as *flags*, markers that tell us that the PMC access is about to be performed by the current thread. Snowboard uses *flags* to infer whether a PMC memory access is to be executed. Once Snowboard observes the execution of any memory accesses from

ID	Summary	Kernel version	Subsystem	Type	Status	Input
#1	BUG: unable to handle page fault for address	5.3.10	include/linux/	DR	**Fixed** [90]	Distinct
#2	EXT4-fs error: swap_inode_boot_loader: ... checksum invalid	5.3.10/5.12-rc3	fs/ext4/	AV	**Harmful**	Duplicate
#3	EXT4-fs error: ext4_ext_check_inode: ... invalid magic	5.3.10	fs/ext4/	AV	Reported	Duplicate
#4	Blk_update_request: IO error	5.3.10/	fs/	AV	**Harmful**	Distinct
#5	Data race: blkdev_ioctl() / generic_fadvise()	5.3.10	block/, mm/	DR	**Harmful**	Distinct
#6	Data race: do_mpage_readpage() / set_blocksize()	5.3.10	fs/	DR	Reported	Distinct
#7	Data race: rawv6_send_hdrinc() / __dev_set_mtu()	5.3.10	net/	DR	**Harmful**	Distinct
#8	Data race: packet_getname() / e1000_set_mac()	5.3.10	net/	DR	**Harmful**	Distinct
#9	Data race: dev_ifsioc_locked() / eth_commit_mac_addr_change()	5.3.10	net/	DR	**Fixed** [86]	Distinct
#10	Data race: fib6_get_cookie_safe() / fib6_clean_node()	5.3.10	net/	DR	Benign	Distinct
#11	BUG: Kernel NULL pointer dereference	5.12-rc3	fs/configfs	DR	**Fixed** [29]	Distinct
#12	BUG: kernel NULL pointer dereference	5.12-rc3	net/l2tp	OV	**Fixed** [30]	Distinct
#13	Data race: cache_alloc_refill() / free_block()	5.12-rc3	mm/	DR	Benign	Duplicate
#14	Data race: tty_port_open() / uart_do_autoconfig()	5.12-rc3	driver/tty/	DR	**Harmful**	Distinct
#15	Data race: snd_ctl_elem_add()	5.12-rc3	sound/core	DR	**Fixed** [41]	Distinct
#16	Data race: tcp_set_default_congestion_control / tcp_set_congestion_control()	5.12-rc3	net/ipv4	DR	Benign	Distinct
#17	Data race: fanout_demux_rollover() / __fanout_unlink()	5.12-rc3	net/packet	DR	**Fixed** [19]	Distinct

Table 2. Testing results by Snowboard, which include 14 concurrency bugs and 3 benign data races. DR denotes "data race", OV denotes "order violation", and AV denotes "atomicity violation" [62]. Concurrent tests may comprise 2 distinct sequential tests ("distinct") or 2 identical sequential tests ("duplicate"). Bugs confirmed as harmful are in **bold type**.

the *flags* set, Snowboard may choose to switch thread execution. Such a switch, from a thread that is going to execute a PMC memory access (e.g., a write) to another thread, can help explore various meaningful interleavings. It is possible that the future execution changes (e.g., a PMC access did not happen right after a memory access from *flags*). However, Snowboard can still notice that a PMC access is just made using `performed_pmc_access` and non-deterministically reschedule thread execution after that PMC access happens.

Also, Snowboard utilizes fine-grained execution control of the kernel threads under test, and ensures only one executes at all times, to enforce a controlled sequential schedule between them.

To start a trial execution, Snowboard restores a checkpoint with the same fixed initial kernel state used during profiling (§4.1). It executes the two tests as separate user-space processes—recall we do not allow the user-space portions of tests to share memory. Right before every instruction, the scheduler may switch to the other test thread, depending on the previous instruction. This switch decision always happens after a memory access.

A non-deterministic decision to switch threads occurs 1) after `performed_pmc_access` is triggered, 2) after `pmc_access_coming` is triggered, and 3) after `is_live` indicates a liveness issue. Snowboard applies non-deterministic interleaving exploration to the target PMC because some PMC accesses are executed several times in one execution. However, it is possible that only one of them will expose the bug (e.g., only an access unprotected by a lock causes the bug, and not all locking primitives or conventions are known ahead of time). Non-deterministically exploring such PMC accesses would hopefully reach the inconsistency-inducing PMC and expose the concurrency bug.

During each trial, Snowboard collects *all* memory accesses. At the end of the trial, Snowboard checks if there is a different PMC whose read and write appears in the accesses of the just-concluded trial. If so, that PMC is added into the set of PMCs under test, and in subsequent trials, `performed_pmc_access` will trigger for the new PMC's accesses as well.

Note that our current design does not perform feedback-based exploration; this is an area for future work (§6).

4.4.1 Implementation Details.

We implemented the execution primitives in a customized QEMU emulator [6] based on SKI, as it already provides the `yield` primitive. It segregates reader/writer threads in separate vCPUs, and only executes one vCPU at a time, enforcing the desired interleaving schedule among them.

Motivated by SKI, `is_live` is implemented by observing the thread execution with some common low-liveness characteristics, including constantly fetching the same memory area, executing `HALT/PAUSE` instructions and having executed a threshold amount of instructions.

The hypervisor performs tracing of every kernel memory access instruction, to enable `performed_pmc_access` and `pmc_access_coming` implementations. In both cases, the features of the current memory access (access type, memory range, value, instruction address) are compared to a set of interesting features. For the former primitive, each access is checked for membership to the accesses in `current_PMCs`, while for the latter, checked against the accesses in `flags`. To reduce false positives, we exclude from memory tracking those accesses that touch kernel stacks (as in §4.1.1).

We implement is_bug by capturing guest-kernel console output, as well as the output of the runtime race detector provided by SKI. Furthermore, to improve the diagnosis, we built post-mortem analysis tools that verify that a data race is caused by an identified PMC and its kernel source code information.

We integrate the execution platform with a lightweight distributed queue [18] so that concurrent tests can be distributed in a cloud platform.

5 Evaluation

We evaluate Snowboard on two recent kernel releases. In the process, Snowboard was able to find new concurrency bugs. In addition, this section analyzes the effectiveness of PMC identification in predicting actual memory communications, evaluates PMC selection, and measures Snowboard performance.

5.1 Experimental Setup

We conduct real-world tests by applying Snowboard to a recent stable Linux kernel version (5.3.10) and a release-candidate version (5.12-rc3). We use the former for a focused search, while we use the latter—assumed to be perhaps more buggy—for a wider search with more clustering strategies.

We use machines of three types. Machine A is an AMD EPYC 7302P with 256GB of memory; machine B is a Google Cloud Platform (GCP) VM with 30 E2 vCPUs [32]; and machine C is a 64-E2-vCPU VM with 512GB of memory.

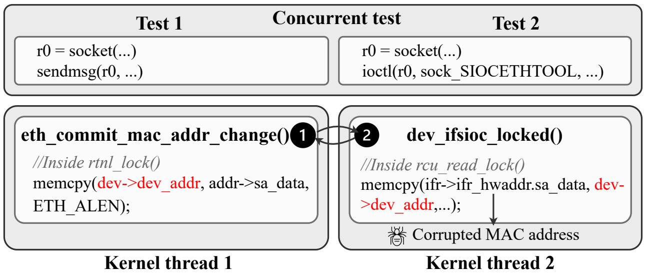
For version 5.3.10, we profile and generate tests on machine A, but run concurrent tests on 10 machine Bs. All clustering strategies combined are used. For version 5.12-rc3, we deploy 11 separate Snowboard instances, one per clustering strategy, doing profiling and test generation on one machine C (for all instances), and testing on 10 machine Bs each §5.3.1.

We measured the performance of profiling, PMC identification, and test generation on machine C and 10 machine Bs, and concurrent test execution on machine B. Every PMC was explored with at most 64 trials.

The new code for Snowboard consists of about 4500 LoC of Python, C, C++, and Bash scripts.

5.2 Finding New Concurrency Bugs

After testing, Snowboard returned tests that had triggered one of the stock bug detectors (e.g., the DataCollider data race bug detector). Data race detectors report data races regardless of whether they are harmful or benign to the kernel. In contrast, Snowboard should only report issues that are likely to be harmful. To prune benign detected data races, we ranked those by frequency, and manually inspected over 100 of the highest-ranked ones. We spent roughly 80 person-hours total on manual inspection and reproduction.



```
r0 = socket(...)
sendmsg(r0, ...)
```

```
r0 = socket(...)
ioctl(r0, sock_SIOCETHTOOL, ...)
```

```
eth_commit_mac_addr_change()
//Inside rtnl_lock()
memcpy(dev->dev_addr, addr->sa_data,
ETH_ALEN);
```

```
dev_ifsioc_locked()
//Inside rcu_read_lock()
memcpy(ifr->ifr_hwaddr.sa_data, dev-
>dev_addr,...);
```

Figure 3. A harmful data race in net/ subsystem found by Snowboard. Due to this bug, the kernel can send a partially updated MAC address to the user. #9 in Table 2.

We arrived at 17 cases that we deemed real bugs (Table 2). Of those 12 were confirmed to be new kernel concurrency bugs and 3 benign data races. Some bugs found could have serious impact on the system by causing kernel panics and filesystem errors. We reported these bugs to kernel developers; they confirmed 12 of those as new, real bugs, and they have fixed 6 of those, as of this writing. During our interaction with developers, we noticed that many bugs were fixed very quickly; they were on average patched within 1.8 days, even when the bug required a lot of code to fix, which, we speculate, is because these bugs were serious. For example, bug #1 in Table 2, which likely affects all kernel code that uses the rhashtable data structure, was fixed within 2 days with 2 patches that changed around 100 LoC kernel code.

Considering the intensive level of continuous scrutiny that the Linux kernel receives [66], we believe these results demonstrate high effectiveness at finding hard kernel concurrency bugs.

Next, we analyze three bugs found by Snowboard and discuss why Snowboard is able to discover them.

Case 1: A data race Concurrency Bug (#9). As shown in Figure 3, a harmful data race found by Snowboard arises between kernel functions eth_commit_mac_addr_change() and dev_ifsioc_locked(). The former (writer), changes the MAC address stored in dev->dev_addr while the latter (reader) reads the address from dev->dev_addr and later sends it back to the user as requested. When their accesses to the shared kernel object interleave, the reader may read a partially-updated MAC address. This corrupted MAC address is then sent to the user without further check.

Our analysis reveals that the eth_commit_mac_addr_change() and dev_ifsioc_locked() functions both execute while holding locks. However, mutual exclusion is not guaranteed because the functions use different locks. Interestingly, the patch submitted by the developers to fix our reported bug changed the locking scheme on the reader side, which had not been changed for over 10 years until we reported this bug, showing that Snowboard uncovers bugs even in mature components.

The challenge in exposing this data race was finding the right user-space code snippets that execute these two functions concurrently. Snowboard composed two sequential tests chosen by Syzkaller because it detected that, individually, the two tests used `memcpy()` to read/write different values to the same address (the `dev->dev_addr` object).

Case 2: A Non-data Race Network Concurrency Bug (#12). This bug leads to a null-pointer dereference in the network stack, and causes a kernel panic (it was the bug illustrated in Figure 1). This bug was exposed under a relatively small subset of interleavings, where the reader side executes both `pppol2tp_connect()` and `l2tp_xmit_core()` after `tunnel` is registered by `l2tp_tunnel_register()` but the tunnel socket (`tunnel->sock`) is not yet initialized.

Our analysis shows that the `tunnel` ID being looked up in `pppol2tp_connect()` is determined by an argument in the `connect()` syscall, which is supplied by the user process. This finding indicates that this bug could be an easy-to-exploit vulnerability. Attacks could trigger this bug as a denial-of-service attack by creating a massive number of user processes requesting the same `tunnel` ID, adding an instance to the ways the kernel can be attacked by a denial-of-service attack. One process, which is running ahead of the rest, will cause the kernel to create a `tunnel` object, and then the others would fetch the newly allocated `tunnel` object and some of them might dereference the `sock` field of the object before it is initialized in `l2tp_tunnel_register()`.

Although concurrency bugs often arise due to data races, they also occur when there are no data races involved (i.e., memory accesses are synchronized), as this bug confirms. Finding non-data-race concurrency bugs is typically more challenging because we cannot rely on data race detectors to identify potentially brittle code regions.

In this case, `l2tp_tunnel_register()` and `l2tp_tunnel_get()` both implement the tunnel registration and application using the standard RCU synchronization protocol, where the writer acquires a lock for updating shared objects and the reader reads optimistically, but safely with an RCU reader lock [14]. An analysis of the kernel commit history reveals that this bug was introduced into the kernel 3 years before this writing, in a patch that fixed another concurrency bug.

Case 3: Conditionals with Omitted Operands (#1). This bug is caused by incorrect assumptions made by kernel developers about a GCC extension to the C conditional (ternary) operator [28]: GCC allows the omission of the second operand ("`x?:y`" has generally the same semantics as "`x?x:y`"). However, if `x` has *side effects*, the terse form has the benefit of not causing side effects twice, when `x` is true.

In this case, developers wrongly assumed that the read access in the ternary conditional would be performed only once. However, depending on optimization, the compiler can emit instructions that perform the read twice, since memory reads are not generally considered side effects in C.

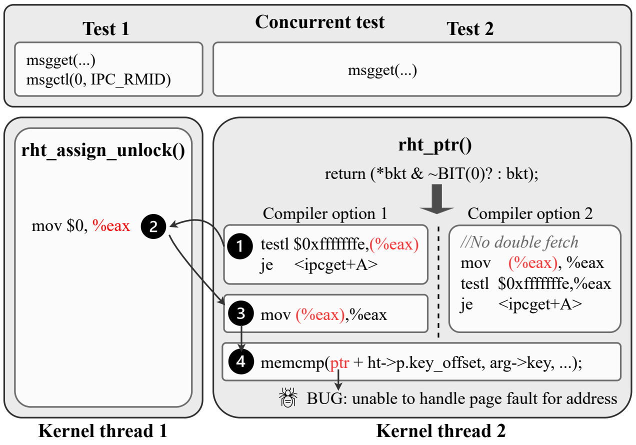


Figure 4. A harmful data race in the `rhashtable` data structure found by Snowboard. Compiler option 1 is "`gcc -O1 -fno-tree-dominator-opts -fno-tree-fre`" and option 2 is "`gcc -O2`". System-call pairs that share `rhashtable`-type data can run into kernel panics. #1 in Table 2.

Figure 4 shows the bug occurs when a write zeroes the value of the shared object (`obj`, referenced by `bkt`), between the two duplicate reads in the ternary operator. In this case, the interleaving vulnerability window is extremely narrow—a single assembly instruction—hence hard for a tool to find at random. When the writer successfully zeroes out the shared memory between duplicate reads, the reader dereferences a null address, causing a kernel panic. Since this is a bug in the `rhashtable` library, any system-call pair that uses it to communicate is affected. For example, this bug can cause kernel panics when system calls `msgctl()` and `msgget()` or `socket()` and `sendmsg()` are executed concurrently.

This bug is particularly insidious because, in its source code, only one read access is explicit, and bug effects can be masked by the default compiler optimizations.

5.3 PMC Identification and Clustering Strategies

Snowboard relies on PMC clustering to prioritize concurrent test generation. Thus, its efficiency of test exploration hinges on the clustering strategies. This section analyzes the effectiveness of each strategy individually.

5.3.1 Clustering Strategy Comparison. To gain a systematic view on the effectiveness of each PMC clustering strategy, we apply each strategy individually on Linux 5.10-rc3 for a period of a week. As §5.1 mentions, we launch 11 instances of Snowboard and configure each to use a unique strategy. Every instance runs from scratch independently to test Linux 5.10-rc3 with the same computing resources for the same amount of time (one week).

In total, 11 unique strategies are evaluated. First, we evaluate the 8 clustering strategies in §4.4. Second, to analyze the impact of ordering the cluster by cardinality, we evaluate Random *S-INS-PAIR*. Compared with *S-INS-PAIR*, which

Clustering strategy	Exemplar PMCs	Tested PMCs	Issues found (days taken to find)
S-FULL	169130631.4K	737.1K	#13 (0.1)
S-CH	36131.8K	146.5K	#13 (4.75)
S-CH-NULL	7457.9K	234.9K	#13 (1.1)
S-CH-UNALIGNED	13681.7K	147.4K	#13 (0.5)
S-CH-DOUBLE	2676.0K	105.5K	#13 (0.3)
S-INS	15.9K	15.9K	**#2** (0.2), #13 (0.3), **#15** (0.1), #16 (0.1)
S-INS-PAIR	738.5K	286.0K	**#11** (4.3), **#12** (6.8), #13 (0.2), **#14** (2.5), **#15** (6.1), #16 (3.2), **#17** (1.2)
S-MEM	2708.1K	235.5K	#13 (0.4)
Random S-INS-PAIR	738.5K	249.5K	**#2** (5.1), #13 (0.1), **#14** (1.1), **#15** (6.5), #16 (2.1), **#17** (3.2)
Random pairing	NA	779.2K (tests)	**#2** (2), #13 (0.4)
Duplicate pairing	NA	831.9K (tests)	#13 (0.2)

Table 3. Testing results on Linux kernel 5.12-rc3 by each concurrent test generation method. "Exemplar PMCs " shows the number of exemplar PMCs as well as the number of clusters according to each strategy ("NA" indicates that no exemplar PMCs are selected by the generation method). "Tested PMCs " shows the number of tested PMCs under each strategy. For *Random pairing* and *Duplicate pairing*, it shows the number of concurrent tests tested. #XX in "Issues found" refers to the bugs listed in Table 2. Bugs confirmed as harmful are shown in **bold type**.

selects exemplar PMCs in order from the smallest to the largest cluster, *Random S-INS-PAIR* randomizes cluster order, but still executes a random exemplar from each. Third, two baseline approaches to generating concurrent tests are evaluated [27, 31]: *Random pairing* randomly selects two kernel sequential tests and combines them as a concurrent test. *Duplicate pairing* randomly generates a concurrent test that consists of two identical kernel sequential tests.

Table 3 presents testing statistics for these strategies. Different PMC strategies affect the number of clusters and, therefore, exemplar PMCs. First, the number of PMCs in *S-FULL* is clearly astronomical; after the test periods, even though the strategy performed the second-highest number of tests, it still was unfocused and found just the most commonly found bug. Thus, more aggressive clustering seems crucial.

Next, we notice that certain strategies (e.g., *S-CH-DOUBLE*) have fewer tested PMCs than others. By inspecting and comparing concurrent tests chosen by each strategy, we find that some (e.g., *S-CH-DOUBLE*) usually consist of 1 or 2 heavy sequential tests, e.g., those contain the `mount()` system call. This may be because the memory accesses selected by such strategies tend to be profiled from heavy sequential tests, reducing the testing throughput.

Interestingly, strategies that tested the most PMCs did not find the most bugs. First, the benign data race #13 is found by all strategies, even the two baseline ones. We believe this is because this data race exists in the memory subsystem, so it can be unmasked by any concurrent tests that request kernel memory. *Random pairing* is also able to find bug #2, which many of the Snowboard strategies did not find, but we chalk that up to the randomness of aimless search.

Second, we find that *S-INS-PAIR*, *S-INS* and *Random S-INS-PAIR* found more bugs than the rest, which indicates that clustering by instruction collects similar behaviors that can be covered with a single PMC, leading to broader exploration, and more bugs found.

Third, although *Random instruction pair* and *instruction pair* are both able to expose several kernel concurrency bugs, *instruction pair* discovered more bugs and in general found bugs more quickly. We conclude that prioritizing the test of uncommon instruction-pair clusters leads to higher behavior coverage per test, than the alternative. Although applied differently, this finding is consistent with the use of instruction-pair coverage to guide search in Krace [92], and composes powerfully with our other techniques.

5.3.2 PMC Identification.

Since identified PMCs are only a hint that actual memory channels will be exercised by a concurrent combination of sequential tests, we evaluate here how often the hint was borne out by the test, which we measure as the PMC *accuracy*: the number of PMC tests that actually exercised the memory channel between the writer and the reader (in at least one of the trials), divided by the total number of PMC tests.

After profiling the Syzkaller sequential tests for Linux 5.12-rc3, Snowboard identified 169.1B PMCs. After testing the kernel for a week, 3743.1K concurrent inputs were tested (in several trials each), of which 784.9K (22%) actually exercised predicted PMCs. Among all tested concurrent inputs, 2153.5K were generated based on predicted PMCs (prioritized by different strategies) while the rest were generated by *Random pairing* or *Duplicate pairing*, which do not involve any PMC analysis. Thus, the precision (i.e., true positive rate) of the PMC identification is about 36% (784.9K out of 2153.5K).

We identify two reasons for mispredictions, i.e., PMCs that could not lead to actual data flow over the channel: the two threads allocated and accessed a buffer, and each ended up with its own private buffer when running concurrently (because the allocator gave each a separate buffer, as intended);

the concurrent execution led a thread to a different control flow, perhaps due to an earlier, different exercised PMC.

Although mispredictions may happen, SNOWBOARD does not produce any false positive bug reports because SNOWBOARD tests PMCs dynamically using generated concurrent inputs and it only raises an alarm when it observes issues in concurrent execution.

The ability of SNOWBOARD to find hard bugs even with a 36% precision when inducing memory channels suggests that further improvements in preparing a kernel state with more pre-allocated objects and thus, less runtime allocation, PMC filtering, and more targeted exploration of interleavings to force the PMC channel, could further boost its success at uncovering hard-to-find bugs, given a fixed time/test budget.

5.4 Performance

Profiling the execution of 129,876 sequential tests generated by Syzkaller takes around 40 hours on machine C (§5.1). After profiling the tests, SNOWBOARD collects all shared accesses, identifies PMCs, and clusters PMCs in under 80 hours on 10 machine Bs. The major computation in this stage involves clustering PMCs according to *S-FULL*, which requires storing all unique PMCs on disk and sorting them by frequency. The effectiveness of *S-FULL* suggests that this is not time well-spent. Removing *S-FULL* from the battery of strategies completes all clustering in under 5 hours on machine C. Finally, SNOWBOARD generates concurrent tests at a throughput higher than 1000 tests per second, which is significantly higher than the test execution throughput.

We study the execution throughput and compare it with SKI's, by randomly selecting 10,000 concurrent tests generated by *Random S-INS-PAIR* and executing them with SNOWBOARD and SKI. SNOWBOARD achieves slightly higher performance than SKI (193.8 vs 170.3 executions/minute). After inspecting several concurrent-test execution traces, we find this is due to SKI's execution of more vCPUs switches than SNOWBOARD: SKI yields thread execution whenever it observes the write or read instruction involved in a PMC (regardless of memory targets), while SNOWBOARD only reschedules execution when it observes a precise PMC write or read access.

Importantly, SNOWBOARD can expose concurrency bugs much faster than SKI. We execute all 9 concurrent tests that found bugs in Linux 5.3.10 with SNOWBOARD and SKI. SKI requires 84 times more interleavings than SNOWBOARD on average to expose the concurrency bug (826.29 interleavings/test on average for SKI, versus only 9.76 interleavings/test for SNOWBOARD). Since SNOWBOARD uses SKI for its fine-grained scheduling control, its advantage comes solely from its use of PMCs as scheduling hints and the scheduling algorithm (Algorithm 2). In contrast, SKI on its own has to consider all potential shared memory accesses, and randomly select a few to explore.

6 Discussion

Testing Thread Count. Although the vast majority of concurrency bugs can be discovered with two testing threads that interleave with each other [62], some only occur with three or more threads running in parallel. As the input space dimension becomes cubic or even higher, finding a concurrent test that exposes these intricate bugs becomes even more challenging. SNOWBOARD should apply to input spaces of more dimensions, e.g., with PMCs of 1 shared write with 2 reads, or PMC chains. Found high-dimension real-world bugs should motivate such future extensions.

Hardware Input & Hardware-specific bugs. SNOWBOARD exercises certain virtualized hardware through associated system calls. However, hardware drivers also receive input from the device itself. Thus, generating concurrent tests to the device is also important for finding concurrency bugs in hardware. As devices usually have diverse input specifications, SNOWBOARD could leverage existing hardware-specific input generation methods [72] to generate device-specific, sequential test corpora.

In addition, since SNOWBOARD always serializes instructions from two threads when exploring interleavings, it cannot expose concurrency bugs that only happen in weak memory models [2]. Finding such concurrency bugs usually requires specific approaches [9, 40] and SNOWBOARD currently does not target these bugs.

Bug Diagnosis and Deterministic Reproduction. The SNOWBOARD PMC approach also assists debugging, which is particularly valuable for diagnosing concurrency bugs. Identifying the problematic interleaving is typically very challenging when debugging kernel concurrency bugs due to the plethora of interleavings. Concurrent tests generated by SNOWBOARD allow developers to refer to the PMC channel to understand a possible cause. In addition, SNOWBOARD has the benefit of providing a reliable environment to replicate bugs once they are found. Although our implementation does not reproduce the wall clock, in all cases we evaluated, SNOWBOARD was able to reproduce found bugs.

Generality. This work focuses on the Linux kernel because it is a critical system, and there is vibrant interest from the research community in higher kernel robustness. The SNOWBOARD approach should also apply to other binaries with a well-defined API, available sequential test corpora, and a reasonable definition of bug detection (e.g., deadlock detection, exceptions, crashes). Most importantly, PMCs are predictive as long as individual operations touch relatively small swaths of memory; operations that touch big objects, e.g., a DBMS updating an in-memory index, would make PMC identification imprecise, reducing utility.

7 Related Work

Kernel Testing. Manually generated kernel tests have been shown to be effective [4, 51, 58], but require significant developer effort and do not cover all corner cases. Randomness-based testing systems, such as Syzkaller [31], Moonshine [70], and HFL [50], have become effective at testing complex systems, such as kernels, through feedback mechanisms that guide test mutations. This enables them to generate tests that explore complex states and deep paths, despite the complex system call semantics and dependencies [15, 35, 56]. Snowboard uses Syzkaller for initial sequential test generation, generalizes to any sequential test-generation mechanism.

Kernel Concurrency Testing. Razzer [43] and Krace [92], the closest work to Snowboard (§2.1), are testing frameworks that target data races in the kernel or in filesystems. Razzer uses static analysis to identify possible data races and relies on fuzzing to generate concurrent inputs that test these data races. Krace proposes feedback-based fuzzing techniques for multi-threaded kernel input generation with the help of a new coverage metric. By contrast, Snowboard is not data-race specific. It generates concurrent tests by identifying PMCs between two threads, and therefore tackles all interleaving-dependent bugs. Also, it identifies PMCs using a common kernel state, which leads to lower false positive rates than static-data race detectors [20, 85], which are notoriously prone to false positives.

DataCollider [21] detects data races in the kernel by randomly scheduling memory accesses and SKI [27] provides systematic instruction-schedule exploration. Instead, Snowboard focuses its interleaving exploration on a specific scheduling hint—the PMC—thus exposing bugs triggered by that PMC with fewer trials. Nevertheless, Snowboard is in general orthogonal to DataCollider and SKI and it could be used to generate concurrent tests as input to them.

Schedule Space Exploration. Concurrent testing requires a schedule exploration approach. Traditionally, developers employed stress testing to cause schedule diversity during testing but more effective techniques have been proposed, which aim to explore the exponential interleaving space better. Several techniques use noise generators [21, 71], typically by injecting sleeps or breakpoints. To reason about interleavings, other techniques propose more systematic algorithms that implement testing schedulers [24]. For instance, CHESS [69] and PCT [8] provide theoretical foundations to reason about schedules and propose schedule exploration algorithms for user-space applications. Snowboard employs a scheduling algorithm based on SKI [27] and PCT [8], but customizes the exploration to use PMCs as hints, which dramatically narrows the search space.

Input Space Exploration. Prior work [75–77] attempts to generate concurrent tests for applications (e.g., Java libraries) by analyzing sequential tests. However, these approaches usually entail heavy analysis on both the target application and the execution trace, thus they do not scale to large applications or the kernel. For instance, some require comprehensive lockset analysis on the execution trace, but as more fine-grained and optimistic locking protocols are used [14, 48], lockset analysis suffers from high false positive rate [17, 78, 79, 92]. The focus of Snowboard is the kernel, which has a complex interface and large size, making it impractical to use standard analysis techniques. In particular, Snowboard does not require static analysis, which generally fails to reason about complex code with extensive aliasing.

Bug Oracles. Bug oracles check whether the program satisfies some aspect of the specification [7, 82, 83, 91]. Simple oracles detect crashes, kernel panics, or hangs; we leverage such oracles in Snowboard. There are other techniques that check for atomicity violations [12, 22, 25, 61, 63, 65, 71] and abnormal communication [26, 64, 97] that belie suspicious executions. Data race detectors [1, 13, 23, 38, 47, 52, 78, 84, 85, 96] fall into the oracle category. As discussed, data races are only associated with one class of concurrency bug and not all data races are bugs, especially in the kernel [21]. Bug oracles are generally orthogonal to the input/schedule space exploration problem, which is the focus of Snowboard.

8 Conclusion

This work introduces Snowboard, a framework to generate effective kernel concurrent tests. Snowboard observes the execution behavior of kernel sequential tests and uses observed memory accesses to identify PMCs, a hint meant to predict actual memory channels during concurrent execution. Among those, it decides which PMCs to turn into tests by clustering them under various strategies. An exhaustive study provides evidence that testing one PMC for every unique pair of potentially communicating instructions, from least to most often observed in the corpus, leads to the most effective (and productive!) exploration of recent Linux kernels. Snowboard has found 14 new kernel concurrency bugs so far, some critical, and some persistent after years of exhaustive kernel testing by the open-source community.

Acknowledgments

We are thankful for the insightful feedback provided by the anonymous reviewers and our shepherd Emery Berger. We are also grateful to Martin Maas for early feedback on this work.

References

[1] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. 2021. Kard: Lightweight Data Race Detection with per-Thread Memory Protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 647–660. https://doi.org/10.1145/3445814.3446727

[2] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-Ups: Concurrency in the Linux Kernel. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 405–418. https://doi.org/10.1145/3173162.3177156

[3] Darrell Anderson. 2002. *Fstress: A Flexible Network File Service Benchmark*. Technical Report.

[4] Linux Kernel Archives. [n.d.]. Linux Kernel Selftests. https://www.kernel.org/doc/Documentation/kselftest.txt Accessed: 7 May 2021.

[5] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 255–268. https://www.usenix.org/conference/atc19/presentation/bai

[6] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA, 41. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator

[7] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: A Complete and Automatic Linearizability Checker. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 330–340. https://doi.org/10.1145/1806596.1806634

[8] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. *SIGARCH Comput. Archit. News* 38, 1 (March 2010), 167–178. https://doi.org/10.1145/1735970.1736040

[9] Jacob Burnim, Koushik Sen, and Christos Stergiou. 2011. Testing Concurrent Programs on Relaxed Memory Models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 122–132. https://doi.org/10.1145/2001420.2001436

[10] Pablo Carvalho, Rommel Cruz, Lucia M A Drummond, Cristiana Bentes, Esteban Clua, Edson Cataldo, and Leandro A J Marzulo. 2020. Kernel concurrency opportunities based on GPU benchmarks characterization. *Cluster Computing* 23, 1 (2020), 177–188. https://doi.org/10.1007/s10586-018-02901-1

[11] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1416–1432. https://doi.org/10.1109/SP40000.2020.00061

[12] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. 2008. JPredictor: A Predictive Runtime Analysis Tool for Java. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/1368088.1368119

[13] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 258–269. https://doi.org/10.1145/512529.512560

[14] The Kernel Development Community. 2020. Linux Rcu Documentation. http://blog.foool.net/wp-content/uploads/linuxdocs/RCU.pdf

[15] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3133956.3134069

[16] James Darvell. 2015. urgent-kernel-patch-ubuntu. https://www.linuxjournal.com/content/urgent-kernel-patch-ubuntu

[17] Pantazis Deligiannis, Alastair F. Donaldson, and Zvonimir Rakamarić. 2015. Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 166–177. https://doi.org/10.1109/ASE.2015.30

[18] Vincent Driessen. [n.d.]. Redis queue. https://python-rq.org Accessed: 7 May 2021.

[19] Eric Dumazet. 2021. net/packet: remove data races in fanout operations. https://github.com/torvalds/linux/commit/94f633ea8ade8418634d152ad0931133338226f6

[20] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[21] John Erickson, Madan Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *Operating System Design and Implementation (OSDI'10)* (operating system design and implementation (osdi'10) ed.). USENIX. https://www.microsoft.com/en-us/research/publication/effective-data-race-detection-for-the-kernel/

[22] Cormac Flanagan and Stephen N Freund. 2004. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 256–267. https://doi.org/10.1145/964001.964023

[23] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[24] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. 2011. Finding Complex Concurrency Bugs in Large Multi-Threaded Applications. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 215–228. https://doi.org/10.1145/1966445.1966465

[25] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. 2011. Finding Complex Concurrency Bugs in Large Multi-Threaded Applications. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 215–228. https://doi.org/10.1145/1966445.1966465

[26] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. 2010. A study of the internal and external effects of concurrency bugs. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. 221–230. https://doi.org/10.1109/DSN.2010.5544315

[27] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule

Exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 415–431. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/fonseca

[28] GNU. [n.d.]. Conditionals with Omitted Operands. https://gcc.gnu.org/onlinedocs/gcc/Conditionals.html Accessed: 7 May 2021.

[29] Sishuai Gong. 2021. configfs: fix a race in configfs_lookup(). https://github.com/torvalds/linux/commit/c42dd069be8dfc9b2239a5c89e73bbd08ab35de0

[30] Sishuai Gong. 2021. net: fix a concurrency bug in l2tp_tunnel_register(). https://github.com/torvalds/linux/commit/69e16d01d1de4f1249869de342915f608feb55d5

[31] Google. 2015. Syzkaller-kernel fuzzer. https://github.com/google/syzkaller

[32] Google. 2019. Introducing E2, new cost-optimized general purpose VMs for Google Compute Engine. https://cloud.google.com/blog/products/compute/google-compute-engine-gets-new-e2-vm-machine-types

[33] gregkh. 2012. Patch "ext4: fix crash when accessing /proc/mounts concurrently" has been added to the 3.6-stable tree. https://www.mail-archive.com/stable@vger.kernel.org/msg19380.html

[34] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[35] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-Based Fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2345–2358. https://doi.org/10.1145/3133956.3134103

[36] Red Hat. 2015. Panic due to race condition between iput() and invalidate_inodes(), kernel BUG at fs/inode.c. https://access.redhat.com/solutions/1593553

[37] Red Hat. 2017. CVE-2017-17712 kernel: Race condition in raw_sendmsg function allows denial-of-service or kernel addresses leak. https://bugzilla.redhat.com/show_bug.cgi?id=1526427

[38] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. 2014. Race Detection for Event-Driven Mobile Applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 326–336. https://doi.org/10.1145/2594291.2594330

[39] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. 2016. An Evolutionary Study of Linux Memory Management for Fun and Profit. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 465–478. https://www.usenix.org/conference/atc16/technical-sessions/presentation/huang

[40] Mohammad Majharul Islam and Abdullah Muzahid. 2013. Characterizing Real World Bugs Causing Sequential Consistency Violations. In *5th USENIX Workshop on Hot Topics in Parallelism (HotPar 13)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/hotpar13/workshop-program/presentation/islam

[41] Takashi Iwai. 2021. ALSA: control: Fix racy management of user ctl memory size account. https://patches.linaro.org/patch/421808/

[42] Joab Jackson. 2012. Nasdaq's Facebook glitch came from 'race conditions'. https://www.computerworld.com/article/2504676/nasdaq-s-facebook-glitch-came-from--race-conditions-.html

[43] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. *Proceedings - IEEE Symposium on Security and Privacy* 2019-May (2019), 754–768. https://doi.org/10.1109/SP.2019.00017

[44] Dave Jones. 2012. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity

[45] Daniel Jordan. 2018. ktask: multithread CPU-intensive kernel work. http://lkml.iu.edu/hypermail/linux/kernel/1811.0/03370.html

[46] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware Blocking Synchronization Primitives. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 603–615. https://www.usenix.org/conference/atc17/technical-sessions/presentation/kashyap

[47] Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced Data Race Detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 406–422. https://doi.org/10.1145/2517349.2522736

[48] Linux Kernel. [n.d.]. Sequence counters and sequential locks. https://www.kernel.org/doc/html/latest/locking/seqlock.html Accessed: 7 May 2021.

[49] Michael Kerrisk. [n.d.]. syscalls(2) — Linux manual page. https://man7.org/linux/man-pages/man2/syscalls.2.html Accessed: 7 May 2021.

[50] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/

[51] KUnit. [n.d.]. KUnit - Unit Testing for the Linux Kernel. https://kunit.dev/third_party/kernel/docs/ Accessed: 7 May 2021.

[52] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. 2011. Pervasive Detection of Process Races in Deployed Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 353–367. https://doi.org/10.1145/2043556.2043589

[53] Michael Larabel. 2019. The Linux Kernel Enters 2020 At 27.8 Million Lines In Git But With Less Developers For 2019. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019

[54] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. 2021. ExpRace: Exploiting Kernel Races through Raising Interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2363–2380. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-yoochan

[55] Michel Lespinasse. 2020. Fine grained MM locking. https://patchwork.kernel.org/project/linux-mm/cover/20200224203057.162467-1-walken@google.com/

[56] Hongliang Liang, Yixiu Chen, Zhuosi Xie, and Zhiyi Liang. 2020. X-AFL: A Kernel Fuzzer Combining Passive and Active Fuzzing. In *Proceedings of the 13th European Workshop on Systems Security* (Heraklion, Greece) *(EuroSec '20)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3380786.3391400

[57] Qianyu Liu, Naijie Gu, and Junjie Su. 2019. Method for Reducing Overhead of Shared Memory Access Instrumentation. In *Proceedings of the 3rd International Conference on Computer Science and Application Engineering* (Sanya, China) *(CSAE 2019)*. Association for Computing Machinery, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/3331453.3361323

[58] LTP. 2012. Linux test project. https://linux-test-project.github.io

[59] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A Study of Linux File System Evolution. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*. USENIX Association, San Jose, CA, 31–44. https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu

[60] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. 2007. A Study of Interleaving Coverage Criteria. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers* (Dubrovnik, Croatia) *(ESEC-FSE companion '07)*. Association for Computing Machinery, New York, NY, USA, 533–536. https://doi.org/10.1145/1295014.1295034

[61] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. 2007. MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 103–116. https://doi.org/10.1145/1294261.1294272

[62] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[63] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1168857.1168864

[64] Brandon Lucia and Luis Ceze. 2009. Finding concurrency bugs with context-aware communication graphs. In *42st Annual IEEE/ACM International Symposium on Microarchitecture MICRO-42 2009, December 12-16, 2009, New York, New York, USA*, David H. Albonesi, Margaret Martonosi, David I. August, and José F. Martínez (Eds.). ACM, 553–563. https://doi.org/10.1145/1669112.1669181

[65] Brandon Lucia, Joseph Devietti, Luis Ceze, and Karin Strauss. 2009. Atom-Aid: Detecting and Surviving Atomicity Violations. *IEEE Micro* 29, 1 (2009), 73–83. https://doi.org/10.1109/MM.2009.1

[66] LWN. 2018. Introducing the syzbot dashboard. https://lwn.net/Articles/749910/

[67] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1007–1024. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry

[68] Paul McKenney. 2019. The RCU API, 2019 edition. https://lwn.net/Articles/777036/

[69] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 267–280. https://dl.acm.org/doi/10.5555/1855741.1855760

[70] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 729–743. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[71] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[72] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2559–2575. https://www.usenix.org/conference/usenixsecurity20/presentation/peng

[73] Bo Qiao, Oliver Reiche, Jürgen Teich, and Frank Hannig. 2020. Unveiling Kernel Concurrency in Multiresolution Filters on GPUs with an Image Processing DSL. In *Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit* (San Diego, California) *(GPGPU '20)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/3366428.3380773

[74] Rapid7. 2011. Linux PolicyKit Race Condition Privilege Escalation. https://www.rapid7.com/db/modules/exploit/linux/local/pkexec/

[75] Malavika Samak and Murali Krishna Ramanathan. 2014. Multithreaded Test Synthesis for Deadlock Detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 473–489. https://doi.org/10.1145/2660193.2660238

[76] Malavika Samak and Murali Krishna Ramanathan. 2015. Synthesizing Tests for Detecting Atomicity Violations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 131–142. https://doi.org/10.1145/2786805.2786874

[77] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. 2015. Synthesizing Racy Tests. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 175–185. https://doi.org/10.1145/2737924.2737998

[78] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multi-threaded programs. *SIGOPS Oper. Syst. Rev.* 31, 5 (1997), 27–37. https://doi.org/10.1145/269005.266641

[79] Justin Seyster, Prabakar Radhakrishnan, Samriti Katoch, Abhinav Duggal, Scott D. Stoller, and Erez Zadok. 2011. Redflag: A Framework for Analysis of Kernel-Level Concurrency. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I* (Melbourne, Australia) *(ICA3PP'11)*. Springer-Verlag, Berlin, Heidelberg, 66–79. https://dl.acm.org/doi/10.5555/2075416.2075425

[80] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. 2010. Do I Use the Wrong Definition? DeFuse: Definition-Use Invariants for Detecting Concurrency and Sequential Bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 160–174. https://doi.org/10.1145/1869459.1869474

[81] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching Concurrent Data Structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 136–148. https://doi.org/10.1145/1375581.1375599

[82] Viktor Vafeiadis. 2010. Automatically Proving Linearizability. In *Proceedings of the 22nd International Conference on Computer Aided Verification* (Edinburgh, UK) *(CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

[83] Martin Vechev, Eran Yahav, and Greta Yorsh. 2009. Experience with Model Checking Linearizability. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software* (Grenoble, France). Springer-Verlag, Berlin, Heidelberg, 261–278. https://doi.org/10.1007/978-3-642-02652-2_21

[84] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. Detecting and Surviving Data Races Using Complementary Schedules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 369–384. https://doi.org/10.1145/2043556.2043590

[85] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. 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* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 205–214. https://doi.org/10.1145/1287624.1287654

[86] Cong Wang. 2021. net: fix dev_ifsioc_locked() race condition. https://github.com/torvalds/linux/commit/3b23a32a63219f51a5298bc55a65ecee866e79d0

[87] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. 2017. How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1–16. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-pengfei

[88] Robert Watson. 2007. Before & After Under The Giant Lock. https://lists.freebsd.org/pipermail/freebsd-hackers/2007-November/022368.html

[89] Wikipedia contributors. 2020. Therac-25 — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Therac-25&oldid=992942654.

[90] Herbert Xu. 2020. rhashtable: Fix unprotected RCU dereference in __rht_ptr. https://github.com/torvalds/linux/commit/1748f6a2cbc4694523f16da1c892b59861045b9d

[91] Min Xu, Rastislav Bodík, and Mark D. Hill. 2005. A Serializability Violation Detector for Shared-Memory Server Programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/1065010.1065013

[92] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1643–1660. https://doi.org/10.1109/SP40000.2020.00078

[93] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*. 661–678. https://doi.org/10.1109/SP.2018.00017

[94] Cheer-Sun D. Yang, Amie L. Souter, and Lori L. Pollock. 1998. All-Du-Path Coverage for Parallel Programs. *SIGSOFT Softw. Eng. Notes* 23, 2 (March 1998), 153–162. https://doi.org/10.1145/271775.271804

[95] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: A Coverage-Driven Testing Tool for Multithreaded Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 485–502. https://doi.org/10.1145/2384616.2384651

[96] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom) *(SOSP '05)*. Association for Computing Machinery, New York, NY, USA, 221–234. https://doi.org/10.1145/1095810.1095832

[97] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 251–264. https://doi.org/10.1145/1950365.1950395

[98] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. *On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications*. Association for Computing Machinery, New York, NY, USA, 540–555. https://doi.org/10.1145/3447786.3456258