OFence: Pairing Barriers to Find Concurrency Bugs in the Linux Kernel

Baptiste Lepers, Josselin Giet, Julia Lawall, Willy Zwaenepoel

OFence: Pairing Barriers to Find Concurrency Bugs in the Linux Kernel

Baptiste Lepers
University of Neuchâtel

Josselin Giet
ENS

Julia Lawall
Inria

Willy Zwaenepoel
University of Sydney

Abstract

Knowing which functions may execute concurrently is key to finding concurrency-related bugs. Existing tools infer the possibility of concurrency using dynamic analysis or by pairing functions that use the same locks. Code that relies on more relaxed concurrency controls is, by and large, out of the reach of existing concurrency-related bug-tracking tools.

In this paper, we propose a new heuristic to automatically infer the possibility of concurrency in lockless code that relies on memory barriers (memory fences) for correctness, a task made complex by the fact that barriers do not have a unique identifier and do not have a clearly delimited scope.

To infer the possibility of concurrency between barriers, we propose a novel heuristic based on matching objects accessed before and after barriers. Our approach is based on the observation that barriers work in pairs: if a write memory barrier orders writes to a set of objects, then there should be a read barrier that orders reads to the same set of objects. This pairing strategy allows us to infer which barriers are meant to run concurrently and, in turn, check the code surrounding the barriers for concurrency-related bugs. As a example of a type of concurrency bug, we focus on bugs related to the incorrect placement of reads or writes relative to barriers. When we detect incorrect read or write placements in the code, we automatically produce a patch to fix them.

We evaluate our heuristic on the Linux kernel. Our analysis runs in 8 minutes. We fixed 12 incorrect ordering constraints that could have resulted in hard-to-debug data corruption or kernel crashes. The patches have been merged in the mainline kernel. None of the bugs could have been found using existing static analysis heuristics.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Concurrent programming structures**.

Keywords: static analysis, memory barrier, kernel

EuroSys '23, May 8–12, 2023, Rome, Italy

ACM Reference Format:
Baptiste Lepers, Josselin Giet, Julia Lawall, and Willy Zwaenepoel. 2023. OFence: Pairing Barriers to Find Concurrency Bugs in the Linux Kernel. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3552326.3567504

1 Introduction

Many software tools exist to help detect bugs in concurrent code. Tools vary, but they share a common first step: finding functions that may run concurrently. A whole body of work has been dedicated to infer possible concurrency using dynamic analysis [14, 19] or by using heuristics to pair locks (locksets) [6, 8, 11, 22]. To the best of our knowledge, code that relies on more relaxed concurrency controls is, by and large, out of the scope of existing tools.

A key observation we make in this paper is that lockless concurrent code frequently relies on memory barriers (memory fences) for correctness. Barriers are used to prevent compilers and CPUs from issuing memory accesses out-of-order, an optimization that may cause unpredictable behavior in concurrent lockless code. In the Linux 5.15 kernel, more than 2000 functions contain memory barriers and over 6000 use kernel APIs that rely on barriers for correctness (e.g., RCU).

A simple but common example of code that relies on barriers is the lockless initialization of a structure. A writer thread initializes a data structure, issues a write barrier, and sets a flag to indicate that the initialization is complete. On the reading side, a thread concurrently checks the flag. If the flag is set, the thread issues a read barrier and the structure is read. Otherwise, the reader thread aborts. The barriers are used to ensure that the writer sets the flag *after* initializing the structure and that the reader checks the flag *before* reading the structure. Without either the read or write barrier, the compiler and CPUs could reorder memory accesses, resulting in the reader reading a partially initialized data structure.

In this paper we propose to use barriers to infer possible concurrency between functions. We present a novel method to automatically discover which pairs of barriers are meant to run concurrently, a task made complex by the fact that a barrier neither indicates which functions it may run concurrently with, nor indicates which memory accesses it orders. Analyzing barriers is notably harder than analyzing lock-based code because, unlike locks, barriers do not have a unique identifier and do not have a clearly delimited scope.

Our work relies on the observation that the memory accesses surrounding barriers are related: one barrier orders the writes to a set of memory locations while the other barrier orders the reads to the same memory locations. We thus pair barriers by matching the objects that are accessed around barriers. Because these objects may be accessed using different variable names in different functions, we propose to uniquely identify them by using a combination of data types and structures' field names. We show that this heuristic allows pairing barriers with a low false positive rate.

Pairing barriers allows us to infer possible concurrency between functions and to find concurrency-related bugs. As an example, we focus on tracking bugs related to the incorrect placement of loads and stores relative to barriers: the stores before a write barrier are expected to match the loads after a read barrier, and vice versa. If not, the barriers provide no ordering guarantee. When we detect inconsistent read and write placements, we automatically produce a patch that reorders the code to fix the bugs. In some situations, we detect that memory accesses would be correctly ordered even without the barrier and we issue a patch that removes the unneeded barrier. The generated patches describe why memory accesses are misplaced, or why the barrier is unneeded. The patches are thus easy to understand and to check for correctness.

We have implemented these ideas in a tool called OFence to be used on the Linux kernel. The tool uses static analysis on kernel code to pair barriers and detect possible deviations from correct barrier usage. We have found 12 ordering bugs in the Linux kernel that could all have resulted in hard-to-debug data corruption or segmentation faults. As of the writing of the paper, 8 patches have been merged in the Linux kernel; the remaining 4 patches correctly fix ordering issues related to barriers, but also reveal other concurrency issues in the code that require fixes that go beyond the scope of this paper. We also detected 53 unneeded barriers.

The number of bugs that we fixed by pairing barriers compares favourably with the number of bugs found using the more conventional lock pairing heuristics (e.g., RacerX [8] fixed 3 races in the Linux kernel and DataCollider [11] fixed 12 races in the Windows kernel). None of the bugs we fixed could have been found using existing tools. Pairing barriers to infer concurrency is thus a useful addition to existing concurrency-detection techniques.

In summary, the contributions of this paper relate to lockless concurrent code and are as follows:

- The first heuristic to infer concurrency in lockless code that relies on barriers for correctness.
- A method for discovering and eliminating concurrency bugs resulting from the incorrect placement of reads or writes relative to barriers.
- A tool implementing this heuristic and method in the Linux kernel.
- The discovery and elimination of 12 concurrency bugs in the Linux kernel and, as a byproduct of our analysis, the elimination of 53 unnecessary barriers in the Linux kernel. None of the bugs could have been found by existing tools.

The rest of this paper is organized as follows. Section 2 explains how barriers are used to enforce ordering constraints. Section 3 presents the main intuitions behind the design of OFence and the high level techniques that allow us to pair barriers. Section 4 presents the implementation of OFence. Section 5 presents the ordering constraints that we check and Section 6 evaluates OFence on the Linux kernel. Section 7 discusses possible extensions of OFence, Section 8 presents the related work, and Section 9 concludes.

2 Background on Memory Barriers

Memory barriers are used to enforce memory access orderings at the compiler and CPU level: memory accesses issued prior a barrier are guaranteed to be performed before memory accesses issued after the barrier (illustrated in Figure 1). When developing lockless code, developers rely on barriers to ensure that shared data structures are read and written following the code order.

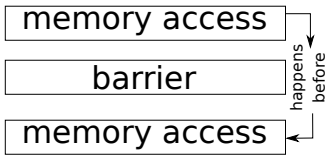

Figure 1. A barrier orders memory accesses done by a single thread.

Pair of barriers. Barriers typically work in pairs, as illustrated in Figure 2. $Thread_1$ writes two variables, `a` and `b`, which are concurrently read by $Thread_2$. The barriers are used to limit the observable states: if $Thread_2$ reads the newest value of `b` , then it will also read the newest value of `a`. Limiting the observable states is frequently used in concurrent code to ensure that reader threads only read initialized data. For instance, in Listing 1, the writer function initializes a data structure and then sets the init flag to true. A reader function concurrently checks the data structure, and only reads its content after checking if it has been initialized. Without either barrier, the compiler and the CPU could reorder the code, resulting in the reader reading partially initialized data.

Inconsistent orderings. Barriers only enforce ordering constraints if the values written *before* the first barrier are read *after* the second barrier, and if the values written *after* the first barrier are read *before* the second barrier. Figure 3 illustrates an incorrect usage of barriers that offers no ordering guarantee: `a` is read *and* written before the barriers

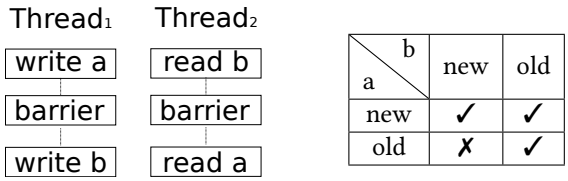

Figure 2. Barriers work in pairs. If *thread*$_2$ reads the new value of b then it is guaranteed also read the new value of a.

Listing 1 Example of barrier usage. The reader checks the flag *before* reading the structure's content, and the write sets the flag *after* initializing the structure.

```
1  void reader(struct my_struct *a) {
2      if(!a->init)
3          return;
4      read_barrier();
5      f(a->y);
6  }
7  void writer(struct my_struct *b) {
8      b->y = ...;
9      write_barrier();
10     b->init = 1;
11 }
```

and b is read *and* written after the barriers. Depending on the interleaving of the threads, it is possible to read the new value of a and the old value of b (Figure 3(a)) or the old value of a and the new value of b (Figure 3(b)). The barriers thus provide no constraint on the observable values of a and b. In the remainder of the paper, we refer to the ordering constraints of Figure 3 as being *inconsistent*.

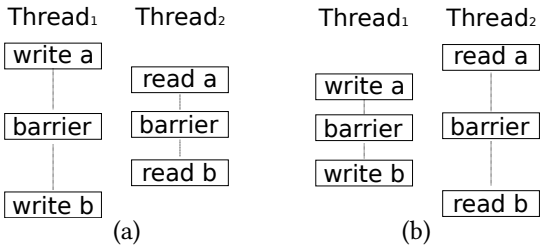

Figure 3. If reads and writes are performed on the same side of the barriers, then no ordering constraint is enforced: the reader can both (a) read the new value of a and the old value of b and (b) the old value of a and the new value of b.

3 Design

In this section we explain the intuitions behind the design of OFence. OFence is based on two main observations.

First, the presence of a barrier in a function is a good indicator that the function is meant to run concurrently with another function. Indeed, enforcing ordering constraints is only useful on objects that can be read and written concurrently. Furthermore, to enforce these constraints, barriers have to work in pairs, as explained in Section 2. We thus infer the possibility of concurrency by pairing barriers that order the same objects.

Second, writing concurrent lockless code is hard, and developers tend to keep ordered memory accesses close to the barriers. The closer a memory access is performed to a barrier, the higher the probability that the memory access is intended to be ordered by the barrier. When a barrier is paired with multiple barriers, we filter pairings based on a notion of distance.

Finding objects used in multiple functions: One of the main difficulties of finding common objects is aliasing: developers may use different variable names to refer to the same memory location. So, instead of relying on variable names, we rely on data types and field names to distinguish objects.

Concurrent code mainly orders reads and writes to shared structures. For every access to a structure field, we build a tuple `(typeof(struct), nameof(field))`. In Listing 1, the `reader` function reads the field `x` of a structure of type `my_struct`. The writer function also writes the field `x` of a structure of type `my_struct`. Because the tuple (`my_struct`, `x`) is accessed by two functions, we refer to it as a *shared object*.

Pairing: We pair two functions if they both have a barrier and if they have at least two common shared objects. The shared objects must be ordered by a barrier in at least one of the two functions (i.e., one object is before the barrier and the other object is after the barrier in one function). A function can be paired with multiple functions as long as all functions order the same shared objects.

In some situations, the read barrier may be implicit. For instance, when a function initializes a structure, issues a write barrier, and then issues an IPC to wake up another thread, the other thread is guaranteed to read an initialized structure, even without issuing a read barrier – we consider that the IPC acts as an implicit barrier. When we detect such a pattern, we leave the writer function unpaired.

4 Implementation

OFence is built on top of Smatch, a static analysis tool for C code [3]. Smatch is designed to track the most common programming errors in the Linux kernel, such as use after free or array overflow. OFence adds 2100 lines of code to Smatch. The code of OFence is available at *https://github.com/BLepers/OFence*.

4.1 Finding barriers

The first step of our analysis finds barriers in the Linux kernel source code. As shown in Table 1, the kernel currently offers eight primitives to explicitly order reads and writes. OFence analyzes all files that contain memory barriers.

Table 1. Barriers used by Linux

Primitive	Description
smp_rmb()	Orders reads
smp_wmb()	Orders writes
smp_mb()	Orders reads and writes
smp_store_mb(&a, v)	Write + smp_mb
smp_store_release(&a, v)	smp_mb + write
smp_load_acquire(&a)	Read + smp_mb
smp_mb__before_atomic()	Barrier before atomic_*()
smp_mb__after_atomic()	Barrier after atomic_*()

Memory barriers (smp_rmb, smp_wmb, smp_mb). Memory barriers are used to force the compiler and CPU to order memory accesses. A read memory barrier (`smp_rmb`) ensures that all the *reads* appearing before the barrier happen before all the *reads* appearing after the barrier. No guarantee is provided on writes. Similarly, a write barrier (`smp_wmb`) only orders writes, no guarantee is provided on reads. A call to `smp_mb` orders both reads and writes. Memory barriers are expensive and kernel developers try to use them only when necessary.

Acquire, release (smp_load_acquire, smp_store_release): To ease the development of concurrent code, the kernel provides functions that issue both a compiler barrier and a memory barrier. The `smp_load_acquire` function reads a variable and then performs a `smp_mb` memory barrier. The `smp_store_release` function performs a `smp_mb` barrier and then writes the variable.

Barriers before/after atomics: The kernel offers more than 400 primitives to perform atomic operations on integers (e.g., `atomic_inc`, `atomic_inc_unless`, ...). Some atomic operations act as memory barriers but some do not (i.e., the CPU is allowed to reorder the atomic operation with other reads and writes). To turn an atomic operation into a barrier, kernel developers use the `smp_mb__before_atomic` and `smp_mb__after_atomic` functions. These functions compile as no-ops on most architectures, but are necessary to enforce order on some weak-memory architectures (e.g., Alpha).

4.2 Finding shared objects and pairing barriers

The second step of the analysis consists in finding shared objects. To find the objects accessed by a function, OFence relies on the control flow graphs (CFGs) produced by Smatch. Smatch generates a CFG per function. Each CFG branches through the code of a function and recursively through the code of all called functions defined in the same file or in an included header.

To avoid analyzing code that is completely unrelated to the barrier, and to reduce the cost of the analysis, we currently bound the exploration of shared objects at function boundaries. In theory, a barrier may order all memory accesses performed prior to the barrier with all memory accesses performed after the barrier but we found that developers rarely rely on the effect of a barrier across function calls. As a first approximation, we thus consider that a barrier may order the memory accesses of the function in which it occurs, as well as its immediate caller functions and callee functions. We also bound the effect of memory barriers at other memory barriers and at atomic operations that have a barrier semantics. For example, if a function contains multiple barriers, then we consider that the first barrier only orders memory accesses up to the second barrier.

For large functions, we currently limit the exploration to within 5 statements of a write memory barrier and 50 statements of a read barrier. We chose these numbers based on the manual observation of common barrier patterns – concurrent writes tend to be close to write barriers, concurrent reads are more spread out. The accuracy of the analysis is not strongly dependent on the exact values. Each shared object found in the exploration is assigned a distance value, which corresponds to the number of statements that separates it from the barrier.

Pairing using shared objects: We build a tuple `(typeof(struct), nameof(field))` for every memory access performed within the analyzed statements, and only keep the tuples that are present in at least two functions (*shared objects*). We use this information to pair functions. Figure 4 illustrates our approach.

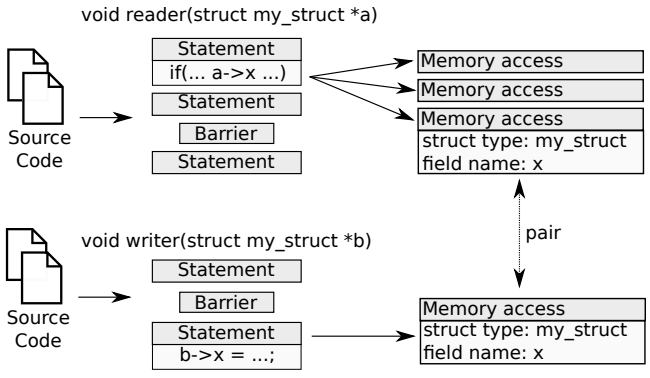

Figure 4. We extract memory accesses from the CFG produced by Smatch and pair functions based on shared objects.

The pairing is performed from the point of view of the write barriers. A write barrier is paired with another barrier if they are preceded or followed by at least two common shared objects. The shared objects must be ordered by at

least one of the two barriers (i.e., one object must be accessed before one barrier while the other must be accessed after that barrier). Algorithm 1 summarizes our approach. We first build a hashmap that associates shared objects with read barriers. Then, for each write barrier, we find barriers that have at least two shared objects in common. When multiple matches are found, we only keep the pairing whose shared objects are closest to the barriers.

Once we have paired two barriers, we build a list of all shared objects $S_0 ... S_n$ that they have in common. If another barrier also has the $S_0 ... S_n$ objects in its list of shared objects, we add the barrier to the pairing. As a consequence, it is possible for a pairing to contain more than two barriers. Such pairings happen when a writer function may run concurrently with multiple reader functions, or when the code relies on a succession of barriers to enforce complex ordering constraints.

Special case: implicit barriers. When a write barrier is followed by an interprocess communication (IPC) call, we consider that the IPC call acts as an implicit read barrier. To distinguish IPC calls from regular function calls, we maintain a list of wake up functions used in the kernel (e.g., `smp_call_function_many`). Note that maintaining a list of functions to detect patterns is common in static analysis – for instance to detect use-after-free bugs many static analyzers maintain lists of functions used to perform allocations and to free memory. If the wake up call is closer to the barrier than the shared objects found using Algorithm 1, then we consider that the barrier exists to order the wake up call with respect to prior memory accesses and we do not try to pair it with another function.

5 Statically Analysing Ordering Constraints

The pairings produced by OFence allow inferring the possibility of concurrency between pairs of functions that use barriers. In this paper, we choose to focus on checking ordering constraints and on tracking unneeded barriers. We distinguish three cases that cover all usages of barriers: unpaired barriers, barriers paired with a single barrier, and barriers paired with multiple barriers. In each case, we explain deviations from correct ordering constraints and how we fix these deviations.

5.1 Deviations on unpaired barrier

When OFence leaves a barrier unpaired, we check if the code would offer the same ordering guarantees without the barrier. We consider that a barrier is unneeded when it is immediately followed by another barrier or by a function that offers barrier semantics.

In the Linux kernel, many functions offer barrier semantics; Table 2 presents a few of these functions. Some atomic operations act as memory barriers while others do not. The same is true for operations that only change a bit in a bitfield. In Linux, the rule of thumb is that atomic and bit operations that return no value do not act as barriers while functions that return a usable value act as barriers. Similarly, some functions that wake up threads contain barriers while others do not. Because kernel developers are often unaware of the semantics of these functions, they sometimes issue a barrier prior to calling a function that already has a barrier semantics.

When OFence matches a barrier prior to a function that already has barrier semantics, OFence issues a patch that removes the barrier and replaces it with a comment. The comment explains that the function call acts as a barrier and documents which prior reads or writes are meant to be ordered by the barrier.

5.2 Deviations when a write barrier is paired with a single read barrier

A write barrier paired with a single other read barrier corresponds to the textbook usage of barriers. The pair of barriers offers ordering guarantees if the writes performed before the write barrier are read after the read barrier, and if the writes performed after the write barrier are read before the read barrier (see Section 2). Any other ordering is inconsistent. In practice, this simple rule can be broken in multiple ways that we detail in this section.

Deviation #1: Misplaced memory access. A misplaced memory access happens when the same shared object is both read and written on the same side of both the read and write barriers. When OFence detects a misplaced memory access, it issues a patch that moves the memory access to the correct side of the barrier. The patch is biased towards the correctness of the writer: we always move the read. The bias comes from the experimental observation that readers tend to contain more bugs than writers. Indeed, shared objects tend to be further away from the barrier in the reader than in the writer, which results in ordering constraints being more often overlooked in readers than in writers.

Deviation #2: Wrong type of barriers. CPUs usually offer different barriers for reads and for writes. We consider that a read barrier should be replaced by a write barrier when it only orders writes. Likewise, a write barrier should be replaced by a read barrier when it only orders reads. When OFence detects a wrong type of barrier, it issues a patch that replaces the barrier with a barrier of the correct type.

Deviation #3: Repeated reads. A repeated read corresponds to a variable correctly read before a read barrier, and then re-read. Listing 2 presents an incorrect re-read of a variable. At the beginning of the function, the `a->field` flag is checked and the function returns if it has a bit of `FLAG_VALUE` set. The `a->field` flag is then re-read as an argument of a function call. If the flag can be modified concurrently, the called function has no guarantee about its value. This, at

Algorithm 1 Pseudo code of the pairing algorithm; we pair barriers using common shared objects and a distance metric.

```
pairings = [];

// Create a "shared object" -> barriers hash
obj_to_barriers = {}
foreach barrier b in kernel_barriers {
  foreach shared_obj o in b->objs {
     obj_to_barriers[o.name].push(b);
  }
}

// For each write barrier, find a barrier
// sharing at least 2 shared objects
foreach barrier b in kernel_barriers {
  continue if(!b->write_barrier);
  (min_w, match) = (inf, NULL);
  foreach (o1,o2) in make_pairs(b->objs) {
    my_weight = o1.distance * o2.distance;
    (pair_weight, pair) = get_pair(b, o1, o2);
    weight = my_weight*pair_weight;
    if(weight < min_w &&
     (b.orders(o1, o2) || pair.orders(o1, o2))
      (min_w, match) = (weight, pair);
  }
  if(match) {
    b.pairings.push({pair:match, w:min_w});
    match.pairings.push({pair:b, w:min_w});
  }
}
// If a barrier exists in multiple pairings,
// keep the pairing with the lowest weight
foreach barrier b in kernel_barriers {
   best = b.pairings.sortBy(w).first();
   others = b.pairings.remove(best);
   foreach(others o)
     o.pairings.remove(b);
   b.pairings = best;
}
// Build pairings array
foreach barrier b in kernel_barriers {
  if(b.pairings && !b.paired) {
    pairings.push([b,b.pairings.first().pair]);
    b.paired = true;
  }
}

// Find unpaired barriers that share
// objects with existing pairs of barriers
// and add them in the pairings
foreach pairing p in pairings {
  barrier1 = p[0];
  barrier2 = p[1];
  common = barrier1.objs.keep(barrier2.objs);
  foreach barrier b in kernel_barriers {
    if(!b.paired && b.objs.contains(common)){
      p.push(b);
    }
  }
}

// Find other barriers surrounded by o1 and o2
// Return the barrier with the lowest weight
fun get_pair(b, o1, o2) {
  // all barriers surrounded by o1 and o2
  b1 = obj_to_barriers[o1.name];
  b2 = obj_to_barriers[o2.name];
  // keep barriers surrounded by both o1 and o2
  matches = b1.intersect(b2);
  // excluding the barrier we pair with
  matches.remove(b);

  // return the match with the lowest weight
  match = matches.sortBy(
         b => b.find(o1).dst * b.find(o2).dst
    ).first();
  w = match.find(o1).dst * match.find(o2).dst
  return (weight, match);
}

// Get pairs of objects from the list of
// objects surrounding a barrier
fun make_pairs(obj_list) {
  for i in 0..obj_list.length {
    for j in i+1..obj_list.length {
      if(obj_list[i].name != obj_list[j].name)
        yield (obj_list[i], obj_list[j]);
    }
  }
}
```

least, questions the need for the check. We thus consider this variation as erroneous. When OFence detects a repeated read, it issues a patch that reuses the initially read value.

5.3 Writer paired with multiple readers and writers

OFence sometimes pairs a write barrier with multiple read and write barriers. While uncommon, these pairings reveal more complex usages of barriers and more complex ordering constraints than those presented previously. Checking the correctness of all possible multi-writer/multi-reader patterns is outside of the scope of this paper. We limit the analysis to the most common multi-writer/multi-reader pattern used in the Linux kernel, illustrated in Figure 5.

In Figure 5, four barriers work in coordination. The writer thread increments a shared object S_0, writes multiple shared objects, and then increments S_0 again. The reader reads S_0,

Listing 2 Re-reading a variable that may be written concurrently may cause unexpected results. In the code, `a->field` might have the `FLAG_VALUE` bit set on line 4 despite the check performed on line 2.

```
1  void reader(struct type *a) {
2      if(a->field & FLAG_VALUE)
3          return;
4      subfunction(a->field, ...);
5      read_barrier();
6  }
```

reads other shared objects, and then checks if S_0 has changed since its initial read. Such a pattern is used to check if the reader reads consistent values: S_0 is the version number of an object, the reader reads the version number prior to reading the object's content, and then re-reads the version at the end to ensure that the object did not change while it was read. In the kernel, this pattern is frequently implemented using the `seqcount` interface, as exemplified in Listing 3. The `get_counters` function of the ARP subsystem reads a per-cpu counter to ensure that the `bcnt` and `pcnt` values are not concurrently modified in the `do_add_counters` function.

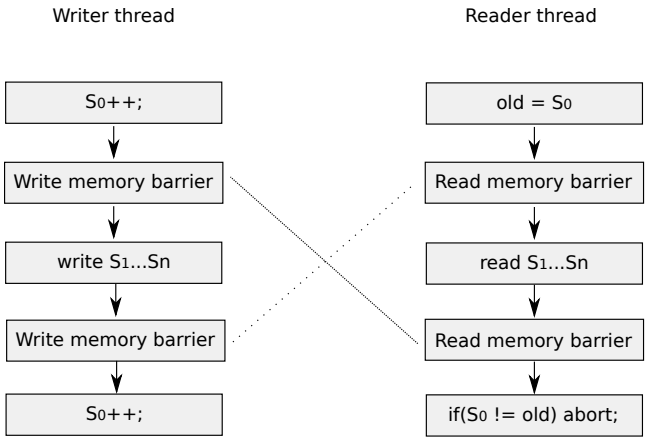

Figure 5. Double pairing. S_0...S_n are shared objects. S_0 usually represents the version of an object that can be read and written concurrently.

Listing 3 Code of the ARP subsystem that relies on four barriers for ordering memory accesses (the barriers are called by the `seqcount` functions, in blue). The reader reads a version number and re-checks the version after reading the `bcnt` and `pcnt` counters. The pattern is used by the reader to detect concurrent writes.

```
1  void get_counters(...) {
2    for_each_possible_cpu(cpu) {
3      seqcount_t *s = &per_cpu(xt_recseq, cpu);
4      xt_entry_foreach(i, t->entries, t->size) {
5        ...
6        do {
7          v = read_seqcount_begin(s);
8          bcnt = tmp->bcnt;
9          pcnt = tmp->pcnt;
10       } while(read_seqcount_retry(s, v));
11     }
12   }
13 }
14
15 int do_add_counters(...) {
16   a = xt_write_recseq_begin();
17   ...
18   xt_entry_foreach(i, p->entries, p->size) {
19     ...
20     ADD_COUNTER(t,paddc[i].bcnt,paddc[i].pcnt);
21   }
22   xt_write_recseq_end(a);
23 }
```

In this example, OFence correctly infers the possibility of concurrency between the four barriers, but the ordering constraints are more subtle than in the previous examples. The barriers work in duo: the first write barrier provides ordering constraints that are used by the second read barrier, and the second write barrier provides ordering constraints used by the first read barrier.

Deviations. The deviations of the multi-writer, multi-reader patterns are the same as those of the single-writer single-reader pattern, but must be checked per duo of barriers instead of being checked between all barriers. For instance, to find misplaced memory accesses OFence looks for values written before the *first* write barrier and read before the

Table 2. Functions that have subtle semantics – some behave as barriers, while others do not.

Primitive	Compiler barrier	Memory barrier	Description
atomic_inc(&a)	✗	✗	Not a barrier on some architectures
atomic_inc_and_test(&a)	✓	✓	Always a barrier
set_bit(&a)	✗	✗	Not a barrier
test_and_set_bit(&a)	✓	✓	Always a barrier
wake_up_process(&task)	✓	✓	Always a barrier

second read barrier. The exploration stops when encountering a memory barrier: when looking for misplaced memory accesses before the *second* read barrier, the backward exploration stops at the *first* read barrier.

5.4 Finding and patching bugs

To find the deviations mentioned above, and to produce patches, we rely on the CFG produced by Smatch. When walking a CFG, we track local variables that point to shared objects. Checking the orderings and fixing them is easy to perform automatically, but may require manual intervention to fix styling issues.

On top of patching the code, the patch documents which shared objects were used to pair the barriers and the type of constraint that was fixed. The patch also briefly explains why the original code was erroneous (e.g., "`(structX, fieldY)` was read before the barrier and incorrectly re-read after the barrier").

6 Evaluation

The evaluation aims at answering the following questions:

- Is OFence effective at finding barrier misuses and at producing patches?
- What is the coverage and the false positive ratio of OFence?

6.1 Setup

We first executed OFence on the Linux 5.11 kernel and have continued to monitor barrier usage in the Linux kernel since. We compile the kernel using an unmodified Ubuntu kernel configuration file. We currently analyze 614 files out of a total of 669 files that contain memory barriers in the kernel (the 55 unchecked files belong to modules that are not compiled by the Ubuntu kernel configuration file). The full analysis of the kernel takes 8 minutes on a 16-core desktop machine with 32GB of memory. Updating the analysis after modifying a single file takes less than 30 seconds for all but 2 driver files, for which it takes 50 seconds. These numbers show that OFence is sufficiently efficient to become part of the standard kernel development toolchain.

6.2 Fixing incorrect code

Using OFence, we found and fixed 12 bugs in functions that rely on barriers for correctness. The bugs are summarized in Table 3. Eight fixes move misplaced memory accesses (e.g., a read placed on the wrong side of a barrier). We also fixed 3 incorrect re-reads of variables and fixed the type of one memory barrier. Looking at the commit history, we found that most bugs were introduced when refactoring the code or adding new functionalities. Unlike traditional race reports that force developers to reason about the race and its consequences, our patches clearly explain why the ordering is incorrect and have been accepted within 24 hours of their submission without any debate.

Table 3. Breakdown of the bugs and suboptimal patterns found in the kernel

Description	Number of patches
Misplaced memory access	8
Racy variable re-read after the read barrier	3
Read barrier used instead of a write barrier	1

Example of a misplaced memory access: Patch 1 fixes a misplaced memory access in the remote procedure call (RPC) interface of Linux. OFence pairs the read and write barriers of `xprt_complete_rqst` and `call_decode` because of their shared objects. OFence then detects that `req->rq_reply_bytes_recvd`, which is used as a flag in `xprt_complete_rqst` (line 4), is incorrectly read after performing a barrier. Reading the flag after the barrier provides no ordering guarantees: the CPU could prefetch `req->rq_private_buf.len` (line 13) before reading the value of the flag (line 11), possibly prefetching a value that has not yet been initialized. The bug could have resulted in the userland reading seemingly random values from the kernel instead of the messages sent via RPC. The fix moves the memory access before the barrier. The patch was accepted in Linux 5.12 and backported to all maintained versions of the Linux kernel (commit f8f7e0fb22b2e75be55f2f0c13e229e75b0eac07).

Patch 1 OFence moves the barrier after the checking of `req->rq_reply_bytes_recvd` to preserve the ordering guarantees offered by `xprt_complete_rqst`.

```
1  void xprt_complete_rqst(...) {
2     req->rq_private_buf.len = ....;
3     smp_wmb();
4     req->rq_reply_bytes_recvd = copied;
5  }
6
7  static void call_decode(...) {
8  +  if (!req->rq_reply_bytes_recvd)
9  +     goto out;
10    smp_rmb();
11 -  if (!req->rq_reply_bytes_recvd)
12 -     goto out;
13    req->rq_rcv_buf.len = req->rq_private_buf.len;
14 }
```

Example of an incorrect re-read prior to a barrier: Patch 2 fixes an incorrect re-read in the event interface of Linux. The `perf_event_addr_filters_apply` function reads the task associated with an event (line 3) and performs

Patch 2 OFence prevents re-reading a racy variable that was used in a condition.

```
1  static void perf_event_addr_filters_apply(...){
2      struct task_struct *task =
3          READ_ONCE(event->ctx->task);
4      ...
5      if (task == TASK_TOMBSTONE)
6          return;
7      ...
8      if (ifh->nr_file_filters)
9  -         mm = get_task_mm(event->ctx->task);
10 +         mm = get_task_mm(task);
11     ...
12     smp_mb();
13 }
```

checks on that task. The task is then re-read later in the function, voiding the guarantees of the prior checks. The bug could have caused the kernel to access a non-existent thread's metadata, resulting in data corruption or kernel crashes. OFence patches the access by re-using the previously read value. The patch was accepted in Linux 5.15 and backported to all maintained versions of the Linux kernel (commit b89a05b21f46150ac10a962aa50109250b56b03b) .

Example of a re-read and misplaced memory access: Patch 3 fixes an incorrect barrier usage found in the socket interface. OFence notices that `reuse->num_socks` is incremented just after a memory barrier in `reuseport_add_sock` (line 3). The write barrier is automatically paired by OFence with a read barrier in `reuseport_select_sock` because of the shared objects. OFence notices that `reuse->num_socks` is read on the wrong side of the barrier on line 12. Re-reading `reuse->num_socks` after the barrier could result in accessing indexes of `reuse->socks` that have not been initialized (on line 15). The fix consists in reusing the value of `reuse->num_socks` that was correctly read before the barrier (on line 9 of Patch 3), fixing both the ordering and re-read problems. The bug could have caused the kernel to read an array out-of-bounds, ultimately resulting in a crash or in data corruption (`reuse->socks` on Line 15 could be NULL or point to a random kernel location in the original code). The patch was accepted in Linux 5.12 and backported to all maintained versions of the Linux kernel (commit fd2ddef043592e7de80af53f47fa46fd3573086e).

6.3 Removing unneeded barriers

We removed 53 unneeded barriers in the Linux kernel. Unneeded barriers were mostly found in the "single barrier" pattern where barriers are followed by a wake up function that already offers barrier semantics.

Patch 3 OFence prevents re-reading a racy variable after a barrier. OFence re-uses the value that was correctly read before the barrier.

```
1  int reuseport_add_sock(...) {
2   reuse->socks[reuse->num_socks] = sk;
3   smp_wmb();
4   reuse->num_socks++;
5  }
6
7  struct sock *reuseport_select_sock(...) {
8   int socks = READ_ONCE(reuse->num_socks);
9   if (likely(socks)) {
10    smp_rmb();
11    ...
12    while(i < ...) {
13 -    if (i >= reuse->num_socks)
14 +    if (i >= socks)
15      ... reuse->socks[i];
16    }
17 }
```

Example of an unneeded barrier: Patch 4 presents an excerpt from the I/O subsystem of Linux. The `rq_qos_wake_function` function uses a barrier before calling `wake_up_process`. OFence knows, for each wake up function, if the function contains a barrier or not and `wake_up_process` does. The extra barrier of `rq_qos_wake_function` is not necessary and can be removed.

Patch 4 OFence removes an unneeded barrier in the I/O subsystem of Linux (`wake_up_process` already contains a barrier).

```
1  static int rq_qos_wake_function(...) {
2     data->got_token = true;
3  -  smp_wmb();
4     wake_up_process(data->task);
5  }
```

6.4 Pairings, false positives and coverage

We experimentally found that most shared objects used to perform the pairings are within 5 statements of write barriers (see Figure 6). Exploring more statements does not significantly increase the number of pairings, but results in a slightly higher number of incorrect pairings. Reads are more spread out (see Figure 7). Just as for writes, most pairings are found using shared objects located close to the barriers, but bugs tend to happen on reads located further away from the barriers. For instance, the incorrect re-read presented in Patch 3 happens 26 statements after the read barrier.

False positives ratio: The number of incorrect pairings and patches is very small. We manually reviewed all the

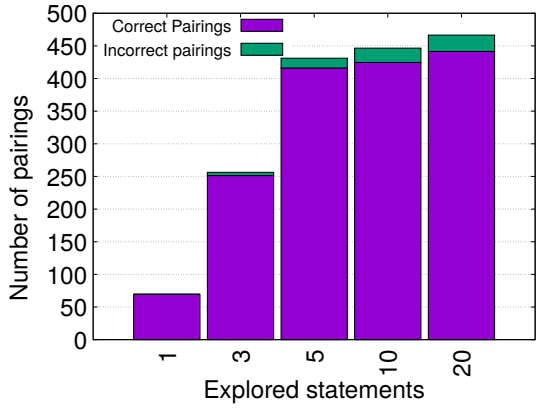

Figure 6. Number of pairings found depending on the number of analyzed statements before and after the write barriers. Most shared objects used in the pairings are within five statements of the write barrier.

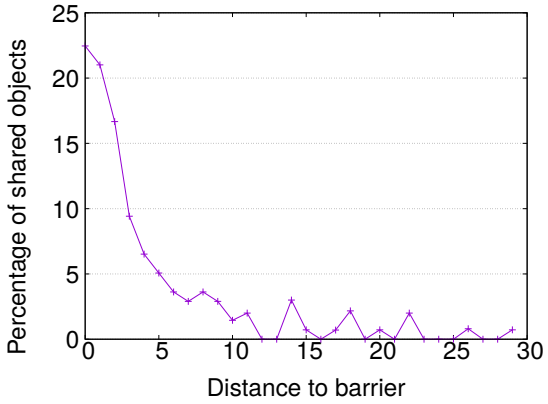

Figure 7. Distance between read barriers and read shared objects.

pairings and patches produced by OFence and found 15 incorrect pairings and 12 incorrect patches among the 456 pairs of barriers that OFence analyzed. All incorrect pairings were easy to detect: OFence paired functions by matching generic types (e.g., the list type used throughout the kernel), but the paired functions had otherwise no logic or shared objects in common.

OFence currently has a 50% false positive ratio (12 incorrect patches and 12 bugs fixed). The false positive ratio of OFence compares favorably with that of previous tools aiming at finding concurrency bugs in the Linux kernel. For instance, 90% of the data races reported by DataCollider [11] are benign. We explain the low number of false positives as follows: we only look for code that marginally deviates from correct barrier patterns. It is unlikely that OFence pairs and patches functions that look like they may perform concurrent accesses but where concurrency is not possible. Indeed, the presence of barriers and shared objects is a strong indicator that the functions are meant to run concurrently.

The main source of false positives when patching code comes from a few functions that break the assumptions made by OFence. Listing 4 presents an example of the bnx2x driver. The `bnx2_sp_event` function guarantees that the `sp_state` field always has at least one of its bits set: the `BNX2X_..._ACK` bit is set *before* clearing the `BNX2X_..._PENDING` bit (the ordering is guaranteed using a call to `smp_wmb()` on line 4 of Listing 4). In this context, the write memory barrier is used to order two writes to the same variable, which breaks our assumptions about variables being accessed either before or after a barrier. OFence produces a patch that moves the accesses after the barrier. Note that OFence still correctly pairs the `bnx2_sp_event` function with its reader function, the code is thus easy to check, and the false positive is easy to detect by simply looking at the way readers access the `sp_state` field.

Listing 4 OFence incorrectly interprets the ordering constraints on `bp->sp_state` because the variable is written on both sides of the barrier.

```
1  void bnx2x_sp_event(...) {
2      ...
3      set_bit(BNX2X_..._ACK, &bp->sp_state);
4      smp_wmb();
5      clear_bit(BNX2X_..._PENDING,&bp->sp_state);
6      ...
7  }
```

Coverage: OFence identifies 456 pairs of barriers in the 614 analyzed files, which accounts for approximately 50% of the barriers present in the files that we analyze.

Our current coverage is explained by the conservative choices we made during pairing. We focused on pairing barriers with other barriers and IPC calls, and most of the barriers we did not pair are intended to run concurrently with code that uses locks or atomic instructions. Looking at the history of bug fixes in the kernel, we inferred that such code rarely suffers from inconsistent ordering or unneeded barriers, and was thus outside the scope of this work. The pairing heuristic of OFence could be extended to pair barriers with atomic operations.

Smatch relies on the kernel Makefiles to compile and check files. As a result, Smatch only verifies the code for a specific configuration of the kernel (e.g., we used the default Ubuntu x86_64 kernel configuration file), thus OFence can only check the corresponding sub-part of the kernel. The more configurations are tested, the more barrier issues will be discovered and patched.

7 Discussion

In this section we discuss extensions to OFence and discuss their interest to the Linux kernel and the static analysis community.

Extending OFence to protect against compiler optimizations: On top of re-ordering memory accesses, a compiler is allowed to modify the way a program accesses memory. For instance, a compiler might fuse a single load with other loads to limit the number of memory accesses, or it might split a single load into multiple smaller loads to make better use of available registers [5, 24]. When a variable is read and written concurrently, the observed results might be unreliable and hard to interpret (e.g., a 64b variable may contain 32b of the old value and 32b of the new value).

The kernel currently offers two annotations to prevent compiler optimizations on concurrent variables: `READ_ONCE` and `WRITE_ONCE` [17]. These annotations are also used by some data race detectors [14] to avoid reporting races on memory accesses that are meant to be racy. Despite calls to use these annotations on all concurrent accesses [24], few kernel functions use `READ_ONCE` and `WRITE_ONCE`.

We have extended OFence to find the memory accesses that need to be annotated with `READ_ONCE` and `WRITE_ONCE`, and to produce patches that add the missing annotations. First, we find barriers that correctly order reads and writes to shared variables. Then, we annotate the reads and writes performed to the shared objects that are accessed concurrently. We are currently in the process of submitting these patches to the Linux kernel community.

Patch 5 presents an example of a patch that adds missing annotations. OFence pairs the `__pollwake` and `poll_schedule_timeout` functions because they have common shared objects. OFence annotates the racy accesses in both functions.

Patch 5 OFence adds the missing `READ_ONCE` and `WRITE_ONCE`.

```
1  static int __pollwake(...) {
2    // Paired with [barrier] in poll_schedule_...
3    smp_wmb();
4  -  pwq->triggered = 1;
5  +  WRITE_ONCE(pwq->triggered, 1);
6    return ...;
7  }
8
9  static int poll_schedule_timeout(...) {
10 -  if (!pwq->triggered)
11 +  if (!READ_ONCE(pwq->triggered))
12     rc = schedule_hrtimeout_range(...);
13   // The following smp_store_mb [is] the
14   // counterpart rmb of the wmb in pollwake()
15   smp_store_mb(...); // equivalent to smp_mb
16 }
```

Extending OFence to find missing barriers: We currently only look for misplaced barriers. We do not analyze code that does not contain barriers and thus do not look for missing barriers.

In our experience, looking for missing barriers leads to a high number of false positives. For instance, a structure might be initialized in isolation, and then modified concurrently. The initialization does not require any barrier if it is not racy. Figuring out if code is meant to run in isolation or concurrently is, in general, a hard problem – the presence of barriers indicates that code is meant to be racy, but the absence of barriers does not give any information.

`READ_ONCE` and `WRITE_ONCE` could be used as an indication that the code is meant to be racy and could be used to find missing barriers. However, we found that most of the code that uses `READ_ONCE` and `WRITE_ONCE` without a memory barrier does so to prevent the compiler from optimizing the code but does not require ordering memory accesses for correctness. The presence of multiple `READ_ONCE` annotations usually indicates that the developer wants to make sure that the data is read from memory, but cannot be used to predict a missing barrier in the general case.

8 Related Work

Race detection. Data races have long been recognized as a serious problem, and there have been many works in this area. We highlight a few that illustrate the main research directions. These all mainly focus on the challenge of figuring out what code can actually be executed concurrently, even if it is not possible to observe a particular concurrent execution.

Due to the need for accurate information about what locks are held at the time of an access to a shared variable, many early approaches relied on dynamic analysis. Mellor-Crummey proposed to record a history, making it possible to find previous accesses to a given shared variable and compare the held locks to the locks held at the time of a given access, amounting to inferring a happens-before relation [19]. Such an approach has a high runtime overhead for instrumenting the code and collecting the lock and access information. Choi et al. [4] improved on this approach by using static analysis to identify probable shared variable accesses and avoid redundancy. Still, these approaches are dependent on the scheduler, which determines whether accesses can be determined to be concurrent in a given run. Eraser [22] proposed lockset analysis, which for each shared variable keeps track of the intersection of the locks ever held at any access; a race is reported if this set ever becomes empty. This approach requires collecting less information than a happens-before approach and is independent of the scheduling that occurs on a particular run.

Dynamic analysis is limited by the ability of the test cases to trigger the execution of the various parts of the code base. Much of the work on race detection for the Linux kernel has

focused on static analysis. An early effort was RacerX [8] that collects locksets via static analysis. To avoid false positives, RacerX uses heuristics for ranking the results, for example using the presence of locks within a control-flow path as an indication of possible concurrency. RacerX also includes heuristics to detect shared variables that are used to collect statistics and assignments to 0 and 1, for which races may be considered to be benign. Deligiannis [6] propose a static lockset-based approach that generates a sequential approximation of a pair of driver entry points and then uses existing verifiers to check generated verification conditions. They observe that many of the races they find involve global counters, and are likely benign. DCUAF [1] uses a statistical analysis to propagate information about the possible concurrent execution of driver entry points across all of the drivers of a particular class. The approach has been applied to find races that lead to use-after-free bugs. RacerD [2, 12] relies on hints from Java programs to understand which functions may run concurrently and only reports races that have a high probability of occurring.

Previous works [13, 20, 26] rely on record and replay to compare the output of the program given different interleavings of memory accesses. Races that have no influence on the program output are classified as benign (e.g., races on statistics counters). The patterns we study have different behaviors and different outputs depending on the interleaving of memory accesses and would be classified as harmful by these works, regardless of the correctness of the enforced memory orderings.

DataCollider [11] is a dynamic data race detector that focuses on reporting races that have a high probability of happening. DataCollider inserts a small pause after an access to a shared variable. A race is only reported if a concurrent access is actually observed within this time. The current approach used in the Linux kernel, KCSAN [7], is based on this idea. DataCollider and KCSAN avoid reporting races on statistics counters and on "safe flags updates" (reading a flag bit in a memory location while another thread updates a different bit in the same memory location).

To the best of our knowledge, no race detector can successfully infer concurrency between two functions that rely on barriers, or check ordering constraints enforced by barriers. Code surrounding barriers is either always reported as erroneous, or ignored.

Use of specifications and comments. iComment [23] and aComment [23] recognize that inferring concurrency by analysing the code is hard. They propose strategies for using natural language processing techniques to infer concurrency specifications from comments and function names. We have also found the comments around barriers to be useful in determining the intent of a particular use of a barrier and, when possible, have used them verify the correctness of the pairings performed by OFence. Unfortunately, currently less than 20% of the barriers in the Linux kernel are commented, and most comments provide little information about the intent of the barrier.

Checking programming rules. Engler *et al.* [9] proposes compiler extensions to check implicit rules of systems. Developers document their rules using templates of the form "Always do X before/after Y". Using these rules to document the orderings enforced by a barrier would simplify the analysis of barrier code. Engler *et al.* [10] proposed a methodology based on statistics to check if the above mentioned rules hold on a large code base. PR-Miner [16] continued this work to automatically extract programming rules such as "a lock is followed by an unlock" and to deduce correlations between two variables (e.g., an array's data and the variable that indicates the number of elements in the array). Coccinelle [15, 21] provides a developer friendly notation, based on patches, to automatically check common programming mistakes such as dereferencing a NULL pointer in a block of code where the pointer is known to be NULL. The pairing heuristic of OFence could be used by these tools to check concurrency rules on code ordered by barriers.

Patterns of concurrency bugs. MUVI [18] focuses on a specific kind of concurrency bug, which occurs when two variables are expected to have correlated values. MUVI detects such pairs of variables and describes how to extend existing race detection strategies to detect this issue. MUVI only considers locks for synchronization.

Xiong *et al.* [25] identify ad hoc synchronization as synchronization implemented without the benefit of well known synchronization primitives such as lock/unlock and cond_wait/cond_signal. Typically, such synchronization is implemented as a loop that spins until a variable has an expected value. They find that ad hoc synchronization is common in user programs, that it is error prone, and that it leads race detection tools to report false positives.

9 Conclusion

In this paper we have proposed a static analysis heuristic to infer concurrency by pairing barriers. We used concurrency information to check the ordering of memory accesses. We have demonstrated the usefulness of the method on the Linux kernel. We fixed 12 incorrect memory orderings that could have caused serious crashes and removed 53 unneeded barriers.

Acknowledgements. We would like to thank our shepherd, Bart Coppens, and the anonymous reviewers for all their helpful comments and suggestions. This work was supported in part by the Australian Research Council Grant DP210101984.

References

[1] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *USENIX Annual Technical Conference (USENIX ATC 19)*, pages 255–268, 2019.

[2] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. RacerD: compositional static race detection. *Proceedings of the ACM on Programming Languages*, (OOPSLA), 2018.

[3] Neil Brown. Smatch: pluggable static analysis for C. "https://lwn.net/Articles/691882/", 2016.

[4] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, 2002.

[5] Many contributors. Who's afraid of a big bad optimizing compiler? "https://lwn.net/Articles/793253/", 2019.

[6] Pantazis Deligiannis, Alastair F Donaldson, and Zvonimir Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177. IEEE, 2015.

[7] Marco Elver. Add Kernel Concurrency Sanitizer (KCSAN). "https://lwn.net/Articles/804813/", 2019.

[8] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. *Proceedings of SOSP*, 37(5):237–252, 2003.

[9] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *4th Symposium on Operating System Design and Implementation (OSDI)*, pages 1–16. USENIX Association, 2000.

[10] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating System Principles (SOSP)*, pages 57–72, 2001.

[11] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *OSDI*, pages 1–16, 2010.

[12] Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. A true positives theorem for a static race detector. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[13] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. In *Architectural Support for Programming Languages and Operating Systems ASPLOS*, pages 185–198, 2012.

[14] KCSAN. data-race in do_select / pollwake. "https://syzkaller.appspot.com/bug?id=99a1275e3959868a4057f3f85e8f6908d21934f6", 2021.

[15] Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the Linux kernel. In *USENIX Annual Technical Conference*, pages 601–614, 2018.

[16] Zhenmin Li and Yuanyuan Zhou. PR-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*, pages 306–315, 2005.

[17] Linux. Access Marking. "https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/memory-model/Documentation/access-marking.txt", 2021.

[18] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A Popa, and Yuanyuan Zhou. 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*, pages 103–116, 2007.

[19] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 24–33. IEEE, 1991.

[20] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–31, 2007.

[21] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys*, pages 247–260, 2008.

[22] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[23] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 11–20. IEEE, 2011.

[24] Linus Torvalds. rcu_read_lock lost its compiler barrier. "https://lore.kernel.org/netdev/CAHk-=wj2t+GK+DGQ7Xy6U7zMf72e7Jkxn4_-kGyfH3WFEoH+YQ@mail.gmail.com/", 2019.

[25] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *OSDI*, pages 163–176, 2010.

[26] Lu Zhang and Chao Wang. RClassify: classifying race conditions in web applications via deterministic replay. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 278–288. IEEE, 2017.