Redflag: A Framework for Analysis of Kernel-Level Concurrency

Appears in the proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'11)

Justin Seyster, Prabakar Radhakrishnan, Samriti Katoch, Abhinav Duggal, Scott D. Stoller, and Erez Zadok

Department of Computer Science, Stony Brook University

Abstract. Although sophisticated runtime bug detection tools exist to root out several kinds of concurrency errors, they cannot easily be used at the kernel level. Our *Redflag* framework and system seeks to bring these essential techniques to the Linux kernel by addressing issues faced by other tools. First, other tools typically examine every potentially concurrent memory access, which is infeasible in the kernel because of the overhead it would introduce. Redflag minimizes overhead by using offline analysis together with an efficient in-line logging system and by supporting targeted configurable logging of specific kernel components and data structures. Targeted analysis reduces overhead and avoids presenting developers with error reports for components they are not responsible for. Second, other tools do not take into account some of the synchronization patterns found in the kernel, resulting in false positives. We explore two algorithms for detecting concurrency errors: one for race conditions and another for atomicity violations; we enhanced them to take into account some specifics of synchronization in the kernel. In particular, we introduce Lexical Object Availability (LOA) analysis to deal with multi-stage escape and other complex order-enforcing synchronization. We evaluate the effectiveness and performance of Redflag on two file systems and a video driver.

1 Introduction

As the kernel underlies all of a system's concurrency, it is the most important front for eliminating concurrency errors. In order to design a highly reliable operating system, developers need tools to find concurrency errors before they cause real problems in production systems. Understanding concurrency in the kernel is difficult. Unlike many user-level applications, almost the entire kernel runs in a multi-threaded context, and much of it is written by experts who rely on intricate synchronization techniques.

Runtime analysis is a powerful and flexible approach to detection of concurrency errors. We designed the *Redflag* framework and system with the goal of airlifting this approach to the kernel front lines. Redflag takes its name from stock car and formula racing, where officials signal with a red flag to end a race. It has two main parts:

1. *Fast Kernel Logging* uses compiler plug-ins to provide *modular* instrumentation that targets specific data structures in specific kernel subsystems for logging. It

   reserves an in-memory buffer to log operations on the targeted data structures with the best possible performance.
2. The *offline Redflag analysis* tool performs post-mortem analyses on the resulting logs. Offline analysis reduces runtime overhead and allows any number of analysis algorithms to be applied to the logs.

Currently, Redflag implements two kinds of concurrency analyses: *Lockset* [15] analysis for data races and *block-based* [19] analysis for atomicity violations. We developed several enhancements to improve the accuracy of these algorithms, including *Lexical Object Availability* (LOA) analysis, which eliminates false positives caused by complicated initialization code. We also augmented Lockset to support Read-Copy-Update (RCU) [12] synchronization, a synchronization tool new to the Linux kernel.

The paper is organized as follows. Section 2 describes our system. Section 3 presents experimental results. Section 4 discusses related work. Section 5 concludes and discusses future work.

2 Design

2.1 Instrumentation and Logging

Redflag inserts targeted instrumentation using a suite of GCC compiler plug-ins that we developed specifically for Redflag. Plug-ins are a recent GCC feature that we contributed to the development of. Compiler plug-ins execute during compilation and have direct access to GCC's intermediate representation of the code [2]. Redflag's GCC plug-ins search for relevant operations and instrument them with function calls that serve as hooks into Redflag's logging system.

Redflag currently logs four types of operations: (1) Field access: read from or write to a field in a `struct`; (2) Synchronization: acquire/release operation on a lock or wait/signal operation on a condition variable; (3) Memory allocation: creation of a kernel object, necessary for tracking memory reuse (Redflag can also track deallocations, if desired); (4) System call (syscall) boundary: syscall entrance/exit (used for atomicity checking).

When compiling the kernel with the Redflag plug-ins, the developer provides a list of `struct`s to target for instrumentation. Field accesses and lock acquire/release operations are instrumented only if they operate on a targeted `struct`. A lock acquire/release operation is considered to operate on a `struct` if the lock it accesses is a field within that `struct`. Some locks in the kernel are not members of any `struct`: these global locks can be directly targeted by name.

To minimize runtime overhead, and to allow logging in contexts where potentially blocking I/O operations are not permitted (e.g., in interrupt handlers or while holding a spinlock), Redflag stores logged information in a lock-free in-memory buffer. I/O is deferred until logging is complete.

When logging is finished, a backend thread empties the buffer and writes the records to disk. With 1GB of memory allocated for the buffer, it is possible to log 7 million events, which was enough to provide useful results for all our analyses.

2.2 Lockset Algorithm

Lockset is a well known algorithm for detecting *data races* that result from variable accesses that are not correctly protected by locks. Our Lockset implementation is based on Eraser [15]. A *data race* occurs when two accesses to the same variable, at least one of them a write, can execute together without intervening synchronization. Not all data races are bugs. A data race is *benign* when it does not affect the program's correctness.

Lockset maintains a *candidate set* of locks for each monitored variable. The candidate lockset represents the locks that have consistently protected the variable. A variable with an empty candidate lockset is potentially involved in a race. Before the first access to a variable, its candidate lockset is the set of all possible locks. The algorithm tracks the current lockset for each thread. Each lock-acquire event adds a lock to its thread's lockset. The corresponding release removes the lock.

When an access to a variable is processed, the variable's candidate lockset is refined by intersecting it with the thread's current lockset. In other words, the algorithm sets the variable's candidate lockset to be the set of locks that were held for *every* access to the variable. When a candidate lockset becomes empty, the algorithm revisits every previous access to the same variable, and if no common locks protected both the current access and that previous one, we report the pair as a potential data race.

Redflag produces at most one report for each pair of lines in the source code, so the developer does not need to examine multiple reports for the same race. Each report contains every stack trace that led to the race for both lines of code and the list of locks that were held at each access.

Beyond the basic algorithm described above, there are several common refinements that eliminate false positives (false alarms) due to pairs of accesses that do not share locks but cannot occur concurrently for other reasons.

Variable initialization. When a thread allocates a new object, no other thread has access to that object. until the thread stores the new object's address in globally accessible memory. Most initialization routines in the kernel exploit this to avoid the cost of locking during initialization. As a result, most accesses during initialization appear to be data races to the basic Lockset algorithm.

The Eraser algorithm solves this problem by tracking which threads access variables to determine when each variable become shared by multiple threads [15]. We implement a variant of this idea: when a variable is accessed by more than one thread or accessed while holding a lock, it is considered shared. Accesses to a variable before its first shared access are marked as thread local, and Lockset ignores them.

Memory reuse. When a region of memory is freed, allocating new data structures in the same memory can cause false positives in Lockset, because variables are identified by their location in memory. Eraser solves this problem by reinitializing the candidate lockset for every memory location in a newly allocated region [15]. Redflag also logs calls to allocation functions, so that it can similarly account for reuse.

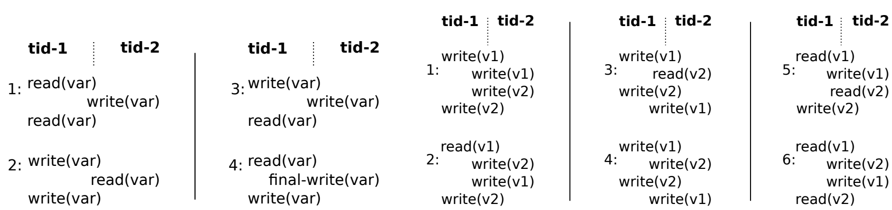
2.3 Block-Based Algorithms

Redflag includes two variants of Wang and Stoller's block-based algorithm [18, 19]. These algorithms check for *atomicity*, which is similar to serializability of database transactions and provides a stronger guarantee than freedom from data races. Two atomic functions executing in parallel always produce the same result as if they executed in sequence, one after the other.

When checking atomicity for the kernel, system calls provide a natural unit of atomicity. By default, we check atomicity for each syscall execution. Not all syscalls need to be atomic, so Redflag provides a simple mechanism to specify smaller atomic regions (see Section 2.5).

We implemented two variants of the block-based algorithm: a single-variable variant that detects violations involving just one variable and a two-variable variant that detects violations involving more than one variable.

The single-variable block-based algorithm decomposes each syscall execution into a set of *blocks*, which represent sequential accesses to a variable. Each block includes two accesses to the same variable in the same thread, as well as the list of locks that were held for the duration of the block (i.e., all locks acquired before the first access and not released until after the second access). The algorithm then checks each block, searching all other threads for any access to the block's variable that might interleave with the block in an unserializable way. An access can interleave a block if it is made without holding any of the block's locks, and the interleaving is unserializable if it matches any of the patterns in Figure 1(a).



(a) Single variable (b) Double variable

Fig. 1. The illegal interleavings in the single- and double-variable block-based algorithms [19]. Note that a final write is the last write to a variable during the execution of an atomic region.

The two-variable block-based algorithm also begins by decomposing each syscall execution into blocks. A two-variable block comprises two accesses to *different variables* in the same thread and syscall execution. The algorithm searches for pairs of blocks in different threads that can interleave illegally. Each block includes enough information about which locks were held, acquired, or released during its execution to determine which interleavings are possible. Figure 1(b) shows the six illegal interleavings for the two-variable block-based algorithm; Wang and Stoller give details of the locking information saved for each block [19].

Together, these two variants are sufficient to determine whether any two syscalls in a trace can violate each other's atomicity [19]. In other words, these algorithms can detect atomicity violations involving any number of variables.

Analogues of the Lockset refinements in Section 2.2 are used in the block-based algorithm to eliminate false positives due to variable initialization and memory re-use.

2.4 Algorithm Enhancements

The kernel is a highly concurrent environment and uses several different styles of synchronization. Among these, we found some that were not addressed by previous work on detecting concurrency violations. This section discusses two new synchronization methods that Redflag handles: multi-stage escape and RCU.

Multi-stage escape. As explained in Section 2.2, objects within their initialization phases are effectively protected against concurrent access, because other threads do not have access to them. However, an object's accessibility to other threads is not necessarily binary. An object may be available to a limited set of functions during a secondary initialization phase and then become available to a wider set of functions when that phase completes. During the secondary initialization, some concurrent accesses are possible, but the initialization code is still protected against interleaving with many functions. We call this phenomenon *multi-stage escape*. As an example, inode objects go through two stages of escape. First, after a short first-stage initialization, the inode gets placed on a master inode list in the file system's superblock. File-system–specific code performs a second initialization and then assigns the inode to a dentry.

The block-based algorithm reported illegal interleavings between accesses in the second-stage initialization and syscalls that operate on files, like `read()` and `write()`. These interleavings are not possible, however, because file syscalls *always* access inodes through a dentry. Before an object is assigned to a dentry—its second escape—the second-stage initialization code is protected against concurrent accesses from any file syscalls. Interleavings are possible with functions that traverse the superblock's inode list, such as the writeback thread, but they do not result in atomicity violations, because they were designed to interleave correctly with second-stage initialization.

To avoid reporting these kinds of false interleavings, we introduce *Lexical Object Availability* (LOA) analysis, which produces a relation on field accesses for each targeted `struct`. Intuitively, the LOA relation encodes observed ordering among lines of code. We use these orderings to infer when an object becomes unavailable to a region of code, marking the end of an initialization phase.

In the inode example, any access from a file syscall serves as evidence that first- and second-stage initialization are finished, meaning that accesses from those initialization routines are no longer possible. Accesses from the writeback thread are weaker evidence, showing that first-stage initialization is finished.

The LOA algorithm first divides the log file into sub-traces. Each sub-trace contains all accesses to one particular instance o of a targeted `struct` S. For each sub-trace, which is for some instance of some `struct` S, the algorithm adds an entry for a pair of statements in the LOA relation for S when it observes that one of the statements occurred after the other in a different thread in that sub-trace. Specifically, for a `struct`

S and read/write statements a and b, (a, b) is included in LOA_S iff there exists a sub-trace for an instance of `struct` S containing events e_a and e_b such that:

1. e_a is performed by statement a, and e_b is performed by statement b, and
2. e_a occurs before e_b in the sub-trace, and
3. e_a and e_b occur in different threads.

We modified the block-based algorithm to report an atomicity violation only if the interleaving statements that caused the violation are allowed to interleave by their LOA relation. For an event produced by statement b to interleave a block produced by statements a and c, the LOA relation must contain the pairs (a, b) and (b, c). Otherwise, the algorithm considers the interleaving impossible.

Returning to the inode example, consider a and c to be statements from the secondary initialization stage and b to be a statement in a function called by the `read` syscall. Because statement b cannot access the inode until after secondary initialization is done, (b, c) cannot be in LOA_{inode}, the LOA relation for inodes.

We also added LOA analysis to the Lockset algorithm: it reports that two statements a and b can race only if both (a, b) and (b, a) are in the LOA relation for the `struct` that a and b access.

Although we designed LOA analysis specifically for multi-stage escape, it can also infer other kinds of order-enforcing synchronization. For example, we found that the kernel sometimes uses condition variables to protect against certain operations to inodes that are in a startup state, which lasts longer than its initialization. We constructed the happened-before relation [8] to determine which potential interleavings were precluded by condition variables, but all such interleavings were already filtered by LOA. LOA analysis can also infer *destruction* phases, when objects typically return to being exclusive to one thread.

Because LOA filters interleavings based on the observed order of events, it can cause false negatives (i.e., it can eliminate warnings corresponding to actual errors). The common technique of filtering based on when variables become shared (see Section 2.2) has the same problem: if a variable becomes globally accessible but is not promptly accessed by another thread, neither technique recognizes that such an access is possible. Dynamic escape analysis addresses this problem by determining precisely when an object becomes globally accessible [19], but it accounts for only one level of escape.

Syscall interleavings. Engler and Ashcraft observed that dependencies on data prevent some kinds of syscalls from interleaving [4]. For example, a `write` operation on a file never executes in parallel with an `open` operation on the same file, because userspace programs have no way to call `write` before `open` finishes.

These dependencies are actually a kind of multi-stage escape. The return from `open` is an escape for the file object, which then becomes available to other syscalls, such as `write`. For functions that are called only from one syscall, our LOA analysis already rules out impossible interleavings between syscalls with this kind of dependency.

However, when a function is reused in several syscalls, the LOA relation, as described above, cannot distinguish executions of the same statement that were executed in different syscalls. As a result, if LOA analysis sees that an interleaving in a shared function is possible between one pair of syscalls, it will believe that the interleaving is possible between any pair of syscalls.

To overcome this problem, we augment the LOA relation to contain entries of the form $((syscall, statement), (syscall, statement))$. As a result, LOA analysis treats a function called from different syscalls as separate functions. Statements that do not execute in a syscall are instead paired with the name of the kernel thread they execute in. The augmented LOA relations can express dependencies caused by both multi-stage escape during initialization and dependencies among syscalls.

RCU. Read-Copy Update (RCU) synchronization is a recent addition to the Linux kernel that allows very efficient read access to shared variables [12]. A typical RCU-write first copies the protected data structure, modifies the local copy, and then replaces the pointer to the original copy with a pointer to the updated copy. RCU synchronization does not protect against lost updates, so writers must use their own locking. A reader needs only to surround read-side critical sections with `rcu_read_lock()` and `rcu_read_unlock()`, which ensure that the shared data structure does not get freed during the critical section.

We extended Lockset to test for correctness of RCU use. When a thread enters a read-side critical section by calling `rcu_read_lock()`, our implementation adds a virtual RCU lock to the thread's lockset. We do not report a data race between a read and a write if the read access has the virtual RCU lock in its lockset. However, conflicting writes to an RCU-protected variable will still produce a data race report.

2.5 Filtering False Positives and Benign Warnings

```
/* [Thread 1] */                   /* [Thread 2] */
spin_lock(inode->lock);
inode->i_state |= I_SYNC;
spin_unlock(inode->lock);
                                   spin_lock(inode->lock);
                                   if (inode->i_state & I_CLEAR) {
                                      /* ... */
                                   }
                                   spin_unlock(inode->lock);
spin_lock(inode->lock);
inode->i_state &= ~I_SYNC;
spin_unlock(inode->lock);
```

Fig. 2. An interleaving that appears to violate the atomicity of the `i_state` field. However, this is a false alarm, because the two threads access different bits of the field.

Bit-level granularity We found that many false positives in the block-based algorithms were caused by *flag variables*, like the `i_state` field in Figure 2, which group several boolean values into one integer variable. Because several flags are stored in the same variable, an access to any individual flag appears to access all flags in the variable. Erickson et al. observed this same pattern in the Windows 7 kernel and account for it in their DataCollider race detector [5].

Figure 2 shows an example of an interleaving that the single-variable block-based algorithm would report as a violation. The two bitwise assignments in thread 1 both write to the `i_state` field. These two writes form a block between which the conditional in thread 2 can interleave; this is one of the illegal patterns shown in Figure 1(a). However, there is no atomicity problem, because thread 1 writes only the `I_SYNC` bit, and thread 2 reads only the `I_CLEAR` bit.

We eliminate such false positives by modifying the block-based algorithms to treat any variable that is sometimes accessed using bitwise operators as 64 individual variables (on 64-bit systems). Our analysis still detects interleavings between bitwise accesses to individual flags and accesses that involve the whole variable.

Idempotent operations An operation is *idempotent* if, when it is executed multiple times on the same variable, only the first execution changes the variable's value. For example, setting a bit in a flag variable is an idempotent operation. When two threads execute an idempotent operation, the order of these operations does not matter, so atomicity violations involving them are false positives. The user can annotate lines that perform idempotent operations. Our algorithms filter out warnings that involve only these lines.

Choosing atomic regions We found that many atomicity violations initially reported by the block-based algorithms are benign: the syscalls involved are not atomic, but are not required to be atomic. For example, the `btrfs_file_write()` function in the Btrfs file system loops through each page that it needs to write. The body of the loop, which writes one page, should be atomic, but the entire function does not need to be.

Redflag lets the user break up atomic regions by marking lines of code as *fenceposts*. A fencepost ends the current atomic region and starts a new one. For example, placing a fencepost at the beginning of the page-write loop in `btrfs_file_write()` prevents Redflag from reporting atomicity violations spanning two iterations of the loop. Fenceposts provide a simple way for developers to express expectations about atomicity.

To facilitate fencepost placement, Redflag determines which lines of code, if marked as fenceposts, would filter the most atomicity violations. Any line of code that executes in the same thread as a block between the first and last operations of the block (see Section 2.3 for a description of blocks) can serve as a fencepost that filters all violations involving that block. After the block-based analysis produces a list of atomicity violations with corresponding blocks, fencepost inference proceeds by greedily choosing the fencepost that will filter the most violations, removing these violations from its list, and repeating until no violations remain. The result is a list of potential fenceposts sorted by the number of violations they filter. The user can examine these candidate fenceposts to see whether they lie on the boundaries of logical atomic regions in the code.

3 Evaluation

To evaluate Redflag's accuracy and performance, we exercised it on three kernel components: Btrfs, Wrapfs, and Noveau. Btrfs is a complex in-development on-disk file system. Wrapfs is a pass-through stackable file system that serves as a stackable file system template. Because of the interdependencies between stackable file systems and the underlying virtual file system (VFS), we instrumented all VFS data structures along with Wrapfs's data structures. We exercised Btrfs and Wrapfs with Racer [16], a workload designed to test a variety of file-system system calls concurrently. Nouveau is a video driver for Nvidia video cards. We exercised Nouveau by playing a video and running several instances of `glxgears`, a simple 3D OpenGL example.

Lockset results. Lockset revealed two confirmed locking bugs in Wrapfs. The first bug results from an unprotected access to a field in the `file struct`, which is a VFS data structure instrumented in our Wrapfs tests. A Lockset report shows that parallel calls to the `write` syscall can access the `pos` field simultaneously. Investigating this race, we found an article describing a bug resulting from it: parallel writes to a file may write their data to the same location in a file, in violation of POSIX requirements [3]. Proposed fixes carry an undesirable performance cost, so this bug remains.

The second bug is in Wrapfs itself. The `wrapfs_setattr` function copies a data structure from the wrapped file system (the *lower inode*) to a Wrapfs data structure (the *upper inode*) but does not lock either inode, resulting in several Lockset reports. We discovered that file truncate operations call the `wrapfs_setattr` function after modifying the lower inode. If a truncate operation's call to `wrapfs_setattr` races with another call to `wrapfs_setattr`, the updates to the lower inode from the truncate can sometimes be lost in the upper inode. We confirmed this bug with Wrapfs developers.

Lockset detected numerous benign races: 8 in Btrfs, and 45 in Wrapfs. In addition, it detected benign races involving the `stat` syscall in Wrapfs, which copies file metadata from an inode to a user process without locking the inode. The unprotected copy can race with operations that update the inode, causing `stat` to return inconsistent (partially updated) results. This behavior is well known to Linux developers, who consider it preferable to the cost of locking [1], so we filter out the 29 reports involving `stat`.

Lockset produced some false positives due to untraced locks: 2 for Wrapfs, and 11 for Noveau. These false positives are due to variable accesses protected by locks external to the traced `struct`s. These reports can be eliminated by telling Redflag to trace those locks.

Block-based algorithms results. Table 1 summarizes the results of the block-based algorithms. We omitted four `struct`s in Btrfs from the analysis, because they are modified frequently and are not expected to update atomically for an entire syscall. The two-variable block-based algorithm is compute- and memory-intensive, so we applied it to only part of the Btrfs and Wrapfs logs.

For Wrapfs, the `wrapfs_setattr` bug described above causes atomicity violations as well as races; these are counted in the "setattr" column. The results for Wrapfs do not count 86 reports for the file system that Wrapfs was stacked on top of (Btrfs in our test). These reports were produced because we told Redflag to instrument all accesses to targeted VFS structures, but they are not relevant to Wrapfs development.

For Wrapfs, the unprotected reads by `stat` described above cause two-variable atomicity violations, which are counted in the "stat" column. These reads do not cause single-variable atomicity violations, because inconsistent results from `stat` involve

	setattr		stat		atime		useless read		counting		struct granularity		untraced lock		other	
Btrfs					5				61	6	2				40	
Wrapfs	34	6		14	43										2	
Nouveau							1				21	2		1		

Table 1. Summary of results of block-based algorithms. From left to right, the columns show: reports caused by `wrapfs_setattr`, reports caused by `touch_atime`, reports caused by reads with no effect, reports involving counting variables, reports caused by coarse-grained reporting of `struct` accesses, and reports that do not fall into the preceding categories. Each column has two sub-columns, with results for the single-variable and two-variable algorithms, respectively. Empty cells represent zero.

multiple inode fields, some read before an update by a concurrent operation on the file, and some read afterwards.

For Noveau, the report in the "Untraced lock" column involves variables protected by the Big Kernel Lock (BKL), which we track.

The "counting" column counts reports whose write accesses are increments or decrements (e.g., accesses to reference count variables). Typically, these reports can be ignored, because the order in which increments and decrements execute does not matter—the result is the same. Our plug-ins mark counting operations in the log, so Redflag can automatically classify reports of this type.

The "struct granularity" column counts reports involving `struct`s whose fields are grouped together by Redflag's logging. Accesses to a `struct` that is *not* targeted get logged when the non-targeted `struct` is a field of some `struct` that is targeted and the access is made through the targeted `struct`. However, all the fields in the non-targeted `struct` are labeled as accesses to the field in the targeted `struct`, so they are treated as accesses to a single variable. This can cause false positives, in the same way that bit-level operations can (*cf.* Section 2.5). These false positives can be eliminated by adding the non-targeted struct to the list of targeted structs.

Filtering. Table 2 shows how many reports were filtered from the results of the single-variable block-based algorithm (which produced the most reports) by manually chosen fenceposts, bit-level granularity, and LOA analysis. The "unfiltered" column shows the number of reports not filtered by any of these techniques. We used fewer than ten manually chosen fenceposts each for Btrfs and Wrapfs. Choosing these fenceposts took only a few hours of work. We did not use fenceposts for our analysis of Nouveau because we found that entire Nouveau syscalls are atomic.

LOA analysis is the most effective among these filters. Only a few `struct`s in each of the modules we tested go through a multi-stage escape, but those `struct`s are widely accessed. It is clear from the number of false positives removed that a technique like LOA analysis is necessary to cope with the complicated initialization procedures in systems code.

	Fenceposts	Bit-level granularity	LOA	Unfiltered
Btrfs	44	0	159	108
Wrapfs	81	6	215	79
Nouveau	-	2	70	22

Table 2. Number of false positives filtered out by various techniques.

Some reports filtered by LOA analysis may be actual atomicity violations, as discussed in Section 2.4. This happened with a bug in Btrfs' inode initialization that we discovered during our experiments. The Btrfs file creation function initializes the new inode's file operations vector just after the inode is linked to a dentry. This linking is the inode's second stage of escape, as discussed Section 2.4. When the dentry link makes the new inode globally available, there is a very narrow window during which another thread can open the inode while the inode's file operations vector is still empty. This bug is detected by the single-variable block-based algorithm, but the report is filtered out by LOA analysis. LOA analysis will determine that the empty operations vector is available to the `open` syscall only if an open occurs during this window in the logged execution, which is unlikely. Dynamic escape analysis correctly recognizes the possible interleaving in any execution, but has other drawbacks, because it accounts for only one level of escape. In particular, the bug can be fixed by moving the file operations vector initialization earlier in the function: before the inode is linked to a dentry, but still after the inode's first escape. Dynamic escape analysis would still consider the interleaving possible, resulting in a false positive.

We tested the fencepost inference algorithm in Section 2.5 on Btrfs. We limited it to placing fenceposts in Btrfs functions (not, e.g., library functions called from Btrfs functions). The algorithm produced a useful list of candidate fenceposts. For example, the first fencepost on the list is just before the function that serializes an inode, which is reasonable because operations that flush multiple inodes to disk are not generally designed to provide an atomicity guarantee across all their inode operations.

Performance. To evaluate the performance of our instrumentation and logging, we measured overhead with a micro-benchmark that stresses the logging system by constantly writing to a targeted file system. For this experiment, we stored the file system on a RAM disk to ensure that I/O costs did not hide overhead. This experiment was run on a computer with two 2.8GHz single-core Intel Xeon processors. The instrumentation targeted Btrfs running as part of the 2.6.36-rc3 Linux kernel. We measured an overhead of 2.44$\times$ for an instrumented kernel without logging, and 2.65$\times$ with logging turned on. The additional overhead from logging includes storing event data, copying the call stack, and reserving buffer space using atomic memory operations.

Schedule sensitivity of LOA. Although LOA is very effective at removing false positives, it is sensitive to the observed ordering of events, potentially resulting in false negatives, as discussed in Section 2.4. We evaluated LOA's sensitivity to event orderings by repeating a workload under different configurations: single-core, dual-core, quad-core, and single-core with kernel preemption disabled. We then analyzed the logs with the single-variable block-based algorithm. The analysis results were quite stable across these different configurations, even though they generate different schedules. The biggest difference is that the non-preemptible log misses 13 of the 201 violations found in the quad-core log. There were only three violations unique to just one log.

4 Related Work

A number of techniques, both runtime and static, exist for tracking down difficult concurrency errors. This section discusses tools from several categories: runtime race detectors, static analyzers, model checkers, and runtime atomicity checkers.

Runtime race detection Our Lockset algorithm is based on the Eraser algorithm [15]. Several other variants of Lockset exist, implemented for a variety of languages. LOA analysis is the main distinguishing feature of our version. Some features of other race detectors could be integrated into Redflag, for example, the use of sampling to reduce overhead, at the cost of possibly missing some errors, as in LiteRace [11].

Microsoft Research's DataCollider [5] is the only other runtime data race detector that has been applied to an OS kernel, to the best of our knowledge. Specifically, it has been applied to several modules in the Windows kernel and detected numerous races. It detects actual data races when they occur, in contrast to Lockset-based algorithms that analyze synchronization to detect possible races. At runtime, DataCollider pauses a thread about to perform a memory access and then uses hardware watchpoints to intercept conflicting accesses that occur within the pause interval. This approach produces no false positives but may take longer to find races and may miss races that happen only rarely. DataCollider uses sampling to reduce overhead.

Static analysis Static analysis tools, typically based on the Lockset approach of finding variables that lack a consistent locking discipline, have uncovered races even in some large systems. For example, RacerX [4] and RELAY [17] found data races in the Linux kernel. Static race detection tools generally produce many false positives, due to the well-known difficulties of analyzing aliasing, function pointers, calling context, etc.

Static analysis of atomicity has been studied (e.g., [7, 14]) but not applied to large systems software. Generally, these analyses check whether the code follows certain safe synchronization patterns.

Runtime atomicity checking. To the best of our knowledge, we are the first to apply a runtime atomicity checker to components of an OS kernel. Although we used the block-based algorithms, other runtime techniques for checking atomicity and similar properties could be adapted to work on Redflag's logs. Atomicity checkers based on Lipton's reduction theorem [9, 6, 19] are computationally much cheaper than the block-based algorithms, because they check a simpler condition that is sufficient but not necessary for ensuring atomicity. As a result, however, they usually produce more false positives.

AVIO [10] and CTrigger [13] use heuristics to infer programmers' expectations about atomicity, and then check for violations thereof (i.e., atomicity violations). An important difference from our work is that the block-based algorithm reports potential and actual atomicity violations, while AVIO and CTrigger report only actual atomicity violations (i.e., atomicity violations that manifest in the monitored run). They actively perturb the schedule to increase the likelihood that atomicity bugs will manifest during testing. Also, they do not detect atomicity violations involving multiple variables. As a result, they are computationally cheaper and produce fewer false positives, but they are more schedule-sensitive and may miss bugs that the block-based algorithms would report. Their implementations use binary instrumentation and are not integrated with the compiler, so it would be difficult to target their analysis to specific data structures.

5 Conclusions

We have described the design of Redflag and shown that it can successfully detect data races and atomicity violations in components of the Linux kernel. To the best of our knowledge, Redflag is the first runtime race detector applied to the Linux kernel, and the first runtime atomicity detector for any OS kernel.

Redflag's runtime analyses are designed to detect potential concurrency problems even if actual errors occur only in rare schedules not seen during testing. The analyses are based on well-known algorithms but contain a number of extensions that significantly improve accuracy, such as LOA analysis. Although the cost of thorough logging can be high, we have shown that Redflag's performance is sufficient to capture traces that exercise many system calls and execution paths.

Future work We plan to extend Redflag with dynamic escape analysis and active analysis (i.e., schedule perturbation) and experiment with the interaction between these techniques and LOA analysis. We also plan to extend Redflag with an analysis that identifies where memory barriers are needed. Memory barriers, which are usually necessary only in low-level systems code, prevent memory operation reorderings that would otherwise be allowed by the weak (not sequentially consistent) memory models used in modern compilers and processors. Another direction for future work is to apply Redflag for performance improvement of concurrent code. By examining locking and access patterns in execution logs, Redflag could identify critical sections that can employ double-checked locking and data structures that would benefit from RCU use. We plan to release the entire Redflag framework and tools publicly under an open source license.

Acknowledgements Research supported in part by NFS grants CNS-0509230 and CNS-0831298, AFOSR grant FA0550-09-1-0481, and ONR grant N00014-07-1-0928.

References

1. BACIK, J. Possible race in btrfs, 2010. `http://article.gmane.org/gmane.comp.file-systems.btrfs/5243/`.
2. CALLANAN, S., DEAN, D. J., AND ZADOK, E. Extending GCC with modular GIMPLE optimizations. In *Proceedings of the 2007 GCC Developers' Summit* (Ottawa, Canada, July 2007).

3. CORBET, J. write(), thread safety, and POSIX. `http://lwn.net/Articles/180387/`.
4. ENGLER, D., AND ASHCRAFT, K. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003), ACM Press, pp. 237–252.
5. ERICKSON, J., MUSUVATHI, M., BURCKHARDT, S., AND OLYNYK, K. Effective data-race detection for the kernel. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Berkeley, CA, USA, 2010), USENIX Association.
6. FLANAGAN, C., AND FREUND, S. N. Atomizer: A dynamic atomicity checker for multi-threaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2004), ACM, pp. 256–267.
7. FLANAGAN, C., AND QADEER, S. A type and effect system for atomicity. In *Proc. ACM SIGPLAN Conference on Programming Language Design and IMPLEMENTATION (PLDI)* (2003), ACM Press, pp. 338–349.
8. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978), 558–565.
9. LIPTON, R. J. Reduction: A method of proving properties of parallel programs. *Commun. ACM 18*, 12 (1975), 717–721.
10. LU, S., TUCEK, J., QIN, F., AND ZHOU, Y. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM, pp. 37–48.
11. MARINO, D., MUSUVATHI, M., AND NARAYANASAMY, S. LiteRace: Effective sampling for lightweight data-race detection. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), ACM, pp. 134–143.
12. MCKENNEY, P. E. *What is RCU?*, 2005. `http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.33.y.git;a=blob;f=Documentation/RCU/whatisRCU.txt`.
13. PARK, S., LU, S., AND ZHOU, Y. Ctrigger: exposing atomicity violation bugs from their hiding places. In *Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009), ACM, pp. 25–36.
14. SASTURKAR, A., AGARWAL, R., WANG, L., AND STOLLER, S. D. Automated type-based analaysis of data races and atomicity. In *Proceedings of the Tenth ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming* (June 2005).
15. SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. ERASER: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems 15*, 4 (1997), 391–411.
16. SUBRATA MODAK. Linux Test Project (LTP), 2009. `http://ltp.sourceforge.net/`.
17. VOUNG, J. W., JHALA, R., AND LERNER, S. RELAY: static race detection on millions of lines of code. In *FSE '07: Proceedings of the 6th ESEC/SIGSOFT International Symposium on Foundations of Software Engineering* (2007), ACM, pp. 205–214.
18. WANG, L., AND STOLLER, S. D. Run-time analysis for atomicity. In *Proceedings of the Third Workshop on Runtime Verification (RV)* (July 2003), vol. 89(2) of *Electronic Notes in Theoretical Computer Science*, Elsevier.
19. WANG, L., AND STOLLER, S. D. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng. 32*, 2 (2006), 93–110.