LockDoc: Trace-Based Analysis of Locking in the Linux Kernel

Alexander Lochmann
Horst Schirmeier
Hendrik Borghorst
TU Dortmund
Department of Computer Science 12
Dortmund, Germany
first.lastname@tu-dortmund.de

Olaf Spinczyk
Osnabrück University
Department of Mathematics and Computer Science
Osnabrück, Germany
olaf.spinczyk@uni-osnabrueck.de

Abstract

For fine-grained synchronization of application and kernel threads, the Linux kernel provides a multitude of different locking mechanisms that are being used on various individually locked data structures. Understanding which locks are required in which order for a particular member variable of a kernel data structure has become truly difficult, even for Linux-kernel experts themselves.

In this paper we introduce LockDoc – an approach that, based on the analysis of execution traces of an instrumented Linux kernel, automatically deduces the most likely locking rule for all members of arbitrary kernel data structures. From these locking rules, LockDoc generates documentation that supports kernel developers and helps avoiding concurrency bugs. Additionally, the (very limited) existing documentation can be verified, and locking-rule violations – potential bugs in the kernel code – can be found.

Our results include generated locking rules for previously predominantly undocumented member variables of 11 different Linux-kernel data structures. Manually inspecting the scarce source-code documentation for five of these data structures reveals that only 53 percent of the variables with a documented locking rule are actually consistently accessed with the required locks held. This indicates possible documentation or synchronization bugs in the Linux kernel, of which one has already been confirmed by kernel experts.

ACM Reference Format:
Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. 2019. LockDoc: Trace-Based Analysis of Locking in the Linux Kernel. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3302424.3303948

1 Introduction

The Linux kernel was originally not designed for fine-grained synchronization of application and kernel threads accessing the global kernel state concurrently. However, the trend towards multi- and many-core platforms of the last two decades forced the kernel developers to first introduce the Big Kernel Lock [4] and later a whole zoo of mechanisms for various individually synchronized data structures [4, 19, 24].

The literature provides a holistic overview over the variety of synchronization primitives, and what contexts they can be used in. However, day-to-day kernel development requires concrete and reliable knowledge on which particular lock – or, sometimes, which *set* of locks in which locking order – must be acquired to access specific kernel data-structure elements.

This knowledge is currently hard to obtain: The Linux kernel does not provide a centralized documentation for such *locking rules*. In fact, this information is scattered across source-code comments in the kernel tree and only available for a subset of data-structure members at all. In the best case, locking rules can be found in the `include/` sub-tree in the header file defining the data structure in question. Other promising locations include the subsystem implementation that uses the particular data structure the most: e.g., locking information on `struct inode` can be found in the first source-code lines or in comments at the beginning of C functions in central files in `fs/`.

If an inexperienced developer is lucky enough to discover a locking rule for a particular data-structure member, they often soon realize that locking is documented – if at all – only informally and with inconsistent wording. In many instances, the source-code comments do not even spell out the name of the locking variable to be used. Instead they assume it to be implicitly evident – possibly because at the time of their writing there only existed one lock to come into consideration. Documented locking rules may also simply have been forgotten as the code evolved, and are consequently outdated – or simply had been wrong from day one.

The Linux source-code comments also indicate that – in all likelihood due to the aforementioned sorry state of locking-rule documentation – even seasoned kernel developers sometimes are not sure which locks were to be acquired for data-structure accesses. Consequently, some parts of the code take a "better safe than sorry" locking strategy instead of rectifying the situation; other parts may even completely lack essential lock acquisitions, opening chances for all sorts of synchronization issues (e.g. data races, dead- or livelocks).

In summary, locking rules are a) only partially and not well documented, b) the documentation is sometimes wrong

and inconsistent with the actual code, which c) can lead to and may very well already have led to locking-related bugs.

In this paper, we propose an approach to overcome these problems: Running in a virtual machine (VM), we trace data-structure accesses and lock acquisitions in an instrumented Linux kernel. From the reconstructed control flow we analyze observed lock combinations for read and write accesses to every data-structure member. We use the results of this analysis in three different ways: 1) We investigate whether the currently documented locking rules reflect what the code *actually* does – i.e. we look for *documentation bugs*, – 2) we fill in the *documentation gaps* by generating locking rules that do not exist yet (from the most likely of a set of locking-rule hypotheses), and 3) we look for memory accesses that do not adhere to the complete set of locking rules, which may be – and in part, according to kernel experts when presented with our data, are – *bugs in the code*.

To summarize, the contributions of this paper are:

- An approach to **record accesses to data-structure members and lock acquisitions** of an instrumented Linux kernel running in a virtualized environment (Sec. 5 and 6),
- a method to **infer locking-rule hypotheses** for each member based on such a recorded trace (Sec. 4, 5 and 6),
- and in-depth analyses based on the trace data and locking rules, including an **examination** (Sec. 7.3) and **extension** (Sec. 7.4) **of the rules documented** in the Linux kernel's source code, and an automated **localization of locking-rule violations** (Sec. 7.5) that potentially represent locking-related kernel bugs.

Sec. 2 gives an overview why locking is a complex issue especially in the Linux kernel. Sec. 3 discusses related work, and Sec. 8 concludes the paper.

2 Background

This section illustrates the relevance of locking in the Linux kernel by a quantitative analysis and discusses the complexity involved for kernel developers, potential bugs related to locking, and the state of the documentation.

2.1 Lock Usage

Locks are an essential mechanism used for the implementation of operating systems. While early UNIX systems only had to deal with interrupt synchronization on single-CPU machines, a modern Linux supports multi- and many-core hardware and has various different kinds of control flows executing in the kernel context concurrently. When Linux started to support multi-core hardware, synchronization was still simple: a single "Big Kernel Lock" [4]. That, however, turned out to be a bottleneck and over time more fine-grained locks of different kinds have been introduced.

Fig. 1 depicts the development of lock usage for various lock types. The X-axis denotes the major releases of the

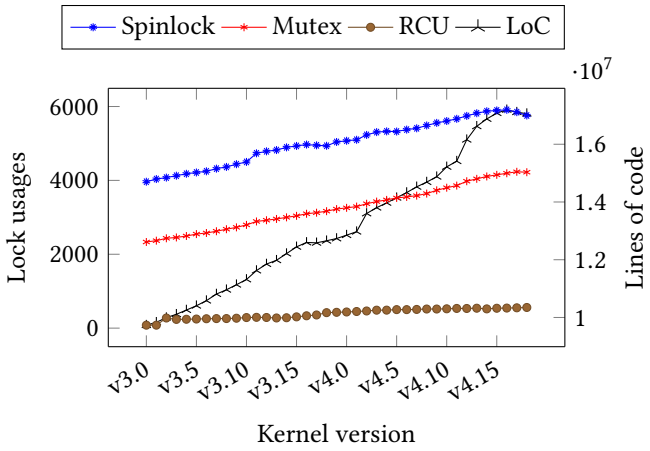

Figure 1. Increase of lock usage and lines of code (LoC) from Linux 3.0 to 4.18.

Linux kernel from version 3.0 up to 4.18 – a period of 7 years. The left Y-axis shows the number of calls to lock-related initialization functions in the source code for the most common lock types.

The number of used mutexes, for instance, has increased by about 81% in the given period. Despite the slight decrease during the last releases, Spinlock usage has increased by about 45% with an absolute number in the order of thousands and a linearly rising trend. The source code base itself increased by 73%, as is shown on the right Y-axis on Fig. 1.

2.2 Locking Complexity

The Linux kernel offers a variety of synchronization methods. Depending on the current and the potential opponent's execution context, which can both be a task, a bottom half, or an IRQ handler, the developer has to choose among various locking-related primitives, e.g. `spin_lock[_(bh|irq)]`, `read_seqbegin`, `write_seqlock[_(bh|irq)]`, `mutex_lock`, `down`, `local_(bh|irq)_disable`, and `preempt_disable` combined with the respective data structures [19]. Besides the contexts there are often other, even more subtle criteria to be considered to identify the most efficient primitive under the specific circumstances. For example, sequence locks (`read_seqbegin`, `write_seqlock*`) are more efficient than spinlocks if data races are unlikely. If it is known that the concurrent control flow must run on the same CPU as the current control flow, lightweight mechanisms such as `preempt_disable`, `local_bh_disable` , and `local_irq_disable` can be used. Depending on the expected duration for which the lock is going to be held, either blocking or non-blocking functions are more appropriate. Finally, there are subtle differences in the semantics of the locking mechanisms. For example, a *down* operation on a semaphore is not equivalent to a `mutex_lock`. Making matters even worse,

many of the primitives are available in reader/writer variants or special killable/interruptible flavors. There is also the alternative of choosing a lock-free data structure or a simple synchronization based on atomic operations.

Beyond the problem of synchronizing accesses to parts of *one* data structure, several of them can be deeply nested or linked by pointers in an arbitrary manner. Therefore, often a hierarchy of locks must be taken in the correct order to make sure that no race conditions or deadlocks are possible.

2.3 Potential for Locking-Related Bugs

Even experienced programmers can easily produce various kinds of bugs and performance issues by wrong lock usage. In the following we will describe a few simple examples.

Following a "better safe than sorry" strategy, it would make sense to acquire all locks that seem to be related to the data structure of interest. This might unintentionally limit parallelism and thus lead to performance degradation. On the other hand, if locking is too permissive, race conditions may occur leading to silent data corruption or system crashes.

When multiple locks are needed, the locking order becomes important. Here, a wrong order could result in a live- or deadlock depending on the lock type. That in turn can freeze the whole kernel.

Finally, it is not unlikely that programmers simply forget to release a lock that has been acquired. For example, in a function that returns early if some error situation is detected, the releasing operation might never be reached. The Linux source code contains about 10% more unlock calls than lock calls because of functions returning early.

2.4 Lock Documentation

In order to cope with the complexity of locking and considering the severity of locking-related bugs, detailed documentation of necessary locks to be acquired before using any kernel data structure would be needed to support kernel developers. However, we will show in the following that the current state of lock-related documentation in Linux is clearly insufficient.

The first location where a new developer would probably look for locking rules would be the header file in which the data structure is defined. In the case of `struct inode`, e.g., there is only a single comment of questionable usefulness in `include/linux/fs.h`:

```
spinlock_t i_lock; /* i_blocks, i_bytes, maybe i_size */
```

Another promising starting point is one of the central *C* files of a subsystem, e.g., `fs/inode.c`. The beginning of that file is shown in Fig. 2. To the best of our knowledge, these are the only two source code locations in the Linux kernel that tell something about locking rules for members of `struct inode`. The remaining information, if present, is distributed across C files under `fs/`, and the header files located in `include/`.

```
1  /*
2   * Inode locking rules:
3   *
4   * inode->i_lock protects:
5   *   inode->i_state, inode->i_hash, __iget()
6   * Inode LRU list locks protect:
7   *   inode->i_sb->s_inode_lru, inode->i_lru
8   * [...]
9   * inode_hash_lock protects:
10  *   inode_hashtable, inode->i_hash
11  * [...]
12  */
```

Figure 2. An extract of the locking rule documentation in `fs/inode.c`

Besides source code comments, the kernel comes with a variety of documentation in the `Documentation/` directory. One part is dedicated to filesystem-related locking[1]. This part, however, provides locking rules per use-case such as dropping an inode, mounting a filesystem, or comparing a dentry. It does not cover how to lock when using individual members in other situations.

In addition to the aforementioned diversity in location, it gets even worse for the wording of locking rules. Scanning the source files of the file-system subsystem in `fs/*.c` reveals a mixture of vocabulary being used in function comments. Different expressions are used to tell the developer which lock has to be acquired, e.g., "holds, "is held", or "to be grabbed". This complicates the automatic extraction of locking rules. The same applies for the name of a lock. Instead of using the variable name, e.g., *i_lock*, various names and descriptions are used. An example where the documentation and the code are telling different stories can be found in *fs/inode.c*: The *find_inode()* function, which traverses the inode list using member *i_hash*, should be called with "inode lock held". However, the *inode_hash_lock* is used when calling *find_inode()* from *iget5_locked()*. The documentation shown in Fig. 2, lines 4, 5, 9 and 10, advises the developer to use both locks. As a consequence, it is not clear to a new kernel developer which lock should be used by just reading the documentation. Finally, even the kernel developers are not sure which lock should be acquired. Fig. 3 shows a function comment which demonstrates that kernel developers do not always know the proper locking method. Moreover, the source code itself contains proofs for that: *"We don't actually know what locking is used at the lower level; but if it's a filesystem that supports quotas, it will be using i_lock as in inode_add_bytes()."*[2] In both cases the developers make best-effort assumptions.

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/Locking

[2] A comment found in `fsstack_copy_inode_size()` in `fs/stack.c`.

```
/*
 * inode_set_flags - atomically set some inode flags
 *
 * Note: the caller should be holding i_mutex, or
 * else be sure that they have exclusive access to
 * the inode structure (i.e., while the inode is
 * being instantiated).  The reason for the
 * cmpxchg() loop --- which wouldn't be necessary
 * if all code paths which modify i_flags actually
 * followed this rule, is that there is at least
 * one code path which doesn't today so we use
 * cmpxchg() out of an abundance of caution.
 *
 * In the long run, i_mutex is overkill, and we
 * should probably look at using the i_lock
 * spinlock to protect i_flags, and then make sure
 * it is so documented in include/linux/fs.h and
 * that all code follows the locking convention!!
 */
```

Figure 3. Documentation of function `inode_set_flags` in `fs/inode.c` (reformatted).

The kernel comes with a variety of documentation in `Documentation/`. One part is dedicated to filesystem-related locking[3]. This part however provides locking rules per use-case such as dropping an inode, mounting a filesystem, or comparing a dentry. It does not cover how to lock particular members. Since a new kernel developer needs explicit locking rules dedicated to specific members,

2.5 Summary

To sum it up, Linux kernel developers and especially developers who are not the original authors of the component they are maintaining have a very difficult job. Dealing with thousands of ubiquitous locks in the kernel source code of many different kinds is very complex, and the risk of inducing bugs is high. It is likely that a tremendous amount of effort is wasted by searching the code and internet forums for help and for debugging locking-related problems. More and better documentation for the locking requirements of all kernel data structures is urgently needed.

3 Related Work

The analysis of lock usage in multi-threaded software and in OS kernels in particular can either be performed ahead of time by static code analysis, at runtime, or ex-post based on an execution trace. We will now discuss related work in each of these areas.

3.1 Ahead-of-Time Analysis

Due to the dynamic nature of locking, static analysis can identify problems in many cases, but is not complete. Especially the use of pointers in C creates aliases for variables, which are hard to analyze [11]. This also affects inter-procedural control flow analysis as it might depend on arbitrary state whose value cannot be determined ahead of time.

Breuer et al. are able detect points of wrong spinlock usage in Linux kernel source code that might lead to a deadlock [5]. However, only one kind of locking mechanism is addressed and only simple coding errors are detected such as locking the same spinlock twice. Another example is the SLAM project [3], which follows a more general approach: It can check API usage rules in a flexible manner. This includes simple properties of correct locking, such as pair-wise usage of lock and unlock primitives. However, SLAM can only analyze sequential control flows and is unaware of different execution contexts such as threads, first-level and second-level interrupt handlers, etc. Another example is the work of Engler et al. [9, 10] who use templates to find bug patterns in Linux and OpenBSD via static code analysis, also including locking-related patterns. Engler's approach to assume that deviant behavior might indicate a bug was an important inspiration for LockDoc.

3.2 Runtime Analysis

Runtime analysis does not suffer from the pointer alias problem, but comes with performance loss and the limitation that only code that is actually executed can be analyzed.

Most modern general-purpose operating systems provide some means for lock debugging in order to support kernel developers in finding the root cause of race conditions, deadlocks, and other lock-related problems. For example, the Linux kernel contains an in-situ lock analysis mechanism called *lockdep* [23]. It aims at detecting deadlocks, livelocks, and lock inversion at runtime. Therefore, it tracks every lock access and incrementally creates a model of valid access patterns per *lock class*. A lock class is a set of locks that follow the same locking rules such as a lock in the inode struct. While each inode has its own lock instance, they are represented by only one lock class. The lockdep mechanism is in particular useful for detecting cyclic lock acquisitions and lock access from the wrong execution context. If a violation to the created model is detected, a warning is printed to the system's kernel log. FreeBSD comes with a similar mechanism called the *witness system* [22].

Besides these built-in mechanisms there is only little help for kernel developers to deal with the locking-related complexity (at runtime). Our literature study only revealed *Lockmeter* [6] and *HaLock* [13]. Both papers describe approaches for gathering statistics on lock usage, which is aimed at identifying performance bottlenecks. While Lockmeter is based on a Linux kernel patch, HaLock requires hardware support, namely a dedicated DIMM module, and thereby reduces the overhead.

Jeong et al. developed a complementary approach: They combine static and dynamic analysis in a tool called *Razzer* to find race bugs in the Linux kernel[14]. They determine potential races via static analysis, and use syscall fuzzing to

[3] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/Locking

generate a multi-threaded program that reproduces a race condition.

Outside the OS kernel there exists far more support for debugging lock usage. Examples for application-level lock profilers and locking-bug detectors are *Critical Lock Analysis* by Chen and Stenstrom [7], Eraser by Savage et al. [25], and Agarwal and Stoller's deadlock-potential checking algorithm [1]. Similarly, a deadlock-preventing approach is taken by Jula et al. [15] who provide *deadlock immunity* for multi-threaded applications.

3.3 Ex-Post Analysis

Trace-based ex-post analyses of OS kernel behavior has been performed by many researchers for various purposes. The approach shares most pros and cons with runtime analysis, but is more convenient for developers as the recorded execution traces can be easily archived and analyzed in arbitrary ways.

There are examples of intrusion detection tools based on system call traces of applications [17] and scheduler improvements based on context switch traces [2].

A more versatile approach has been presented by Matni et al. who use LTTng to instrument the Linux kernel in order to trace various kinds of events [21]. Finite state machine models are used to describe valid kernel behavior and deviations from that can be detected in the traces automatically. In one example the authors used their approach to check the rule that a thread must not block while it is holding a spinlock.

Other frameworks for obtaining kernel execution traces are *Pin* [20], which can instrument arbitrary x86 executables, and *Tralfamadore* [18], which executes a system in a VM environment and produces complete traces of all executed machine instructions. The latter could have been used as a basis for LockDoc's analyses, but we preferred to re-use our own framework.

3.4 Summary

Until now researchers seem to have focused on finding bugs in existing code and profiling to avoid bottlenecks. However, the presented approaches do not support kernel developers in *writing correct code* in the first place. Up-to-date documentation on where to use which lock is the key to achieve correct multi-threaded kernel code faster. The availability of locking-related documentation in Linux has been discussed in Sec. 2.4. In FreeBSD the situation is better, as the documentation is more complete and well-structured, but it still has to be maintained manually. To the best of our knowledge, no approach has so far automatically derived locking rules and revealed potential bugs. This paper fills that gap by linking the documentation with the reality of locking in the respective OS kernel using a trace-based approach.

Of all aforementioned approaches the work by Engler et al. [10] has probably the most resemblance to LockDoc.

```
1 lock(&sec_lock); // transaction a - start
2 seconds = seconds + 1;
3 if (seconds == 60) {
4     lock(&min_lock); // transaction b - start
5     seconds = 0;
6     minutes = minutes + 1;
7     unlock(&min_lock); // transaction b - end
8 }
9 unlock(&sec_lock); // transaction a - end
```

Figure 4. Implementation of a clock counter.

The main difference of Engler's contribution is that it is based on a static code analysis instead of trace analysis, which has advantages and disadvantages, and the lack of a comprehensive case study on the locking rules in Linux, which we provide here. Both approaches are complementary, as they look at the same problem from a different angle.

4 Approach and Challenges

Sec. 2 already stressed the importance of locking in modern operating systems as well as the consequences of faulty locking. Being able to automatically derive locking rules from execution traces could help to write correct code, check documentation, and even find bugs in existing kernel code. In this section, we now discuss the key assumptions behind the approach as well as challenges. For this purpose we first introduce a simple and contrived example.

4.1 Setting: A Shared 'Time' Data Structure

Fig. 4 shows code for incrementing the time, which is represented by the two shared variables `seconds` and `minutes`. In order to guarantee mutual exclusion, `seconds` is protected by the lock `sec_lock`. When `seconds` reaches 60, the lock `min_lock` is acquired as well and `minutes` is incremented. In a hypothetical execution trace the code fragment appears 1000 times. Besides this, the trace also contains one execution of a similar function with an important deviation: The developer forgot to acquire `min_lock` when `seconds` reaches 60 – a bug that could cause a race condition.

The relation of 1000 correct lock usages to one incorrect usage reflects a fundamental assumption of LockDoc: Locking-related bugs are rare in the systems we are targeting. If they were not, the respective system would suffer from frequent misbehavior, which is usually not the case. Most of the time the system uses the correct lock – or set of locks in the right order.

To automatically derive locking rules for the given example two steps are necessary: (1) Analyze which locks are being held for each interesting variable in the trace at the moment of an access and (2) derive locking hypotheses and select a winning hypothesis – the one which is most likely the locking rule for the variable.

Table 1. Accesses to `seconds` and `minutes` grouped by access type for one execution. Each entry lists the number of actually observed accesses (*Observed*), aggregated accesses (*Folded*), and cases where writes won over reads (*WoR = Write over Read*).

Variable	Access Type	Observed		Folded		WoR	
		a	b	a	b	a	b
`seconds`	r	2	0	1	0	0	0
	w	1	1	1	1	1	1
`minutes`	r	0	1	0	1	0	0
	w	0	1	0	1	0	1

4.2 Analysis of Locks per Variable

In order to explain the analysis of lock usage for the variables `seconds` and `minutes` (step 1), we first introduce how we use the term *transaction* (txn) in this context. A transaction is a – potentially interrupted – sequence of memory accesses in the execution trace with a fixed set of held locks. For now, an execution trace consists of variable accesses and lock operations. A transaction starts upon lock acquisition and ends when the same lock gets released (see comments in Fig. 4). Each access belongs to only one transaction. Thus, if another lock is acquired within a transaction, the following accesses are assigned to the new transaction. When the nested transaction ends by an operation that releases the second lock, the following accesses are assigned to the first transaction, because the set of held locks is the same again.

Now each variable access can be assigned unambiguously to a transaction and the corresponding set of held locks. The resulting 4-tuples, which consist of the number of the element in the trace, the name of the accessed variable, the access type (read or write), and the set of locks being held, form the set of observations that are used as positive or negative examples for the locking rule hypotheses.

Tab. 1 provides statistics on one execution of the transactions *a* and *b* from Fig. 4. The variable `seconds`, for example, is read two times in transaction *a* (column *Observed/a*). The distinction between reads and writes is necessary because different locking rules might apply. For locking rule derivation it is not relevant how often a variable is accessed within a transaction. Therefore, we create a binary matrix (see column *Folded*), which denotes whether there was at least one access of the respective variable in the transaction. Another important information for locking rule derivation is whether a transaction contains both a read and a write operation on the same variable. In this case it is unclear whether the locks were held because of the read or the write. As locking rules for write operations are typically more restrictive (more locks), we assume that combined read/write transactions are treated as write transactions – the reads are ignored in this case. The result is shown in the last column (*WoR = Write over Read*).

Table 2. Possible locking rules for writing variable `minutes` with their absolute and relative support (s_a and s_r).

ID	Locking Hypothesis	s_a	s_r
#0	*no lock needed*	17	100%
#1	`sec_lock`	17	100%
#2	`sec_lock` → `min_lock`	16	94.12%
#3	`min_lock`	16	94.12%
#4	`min_lock` → `sec_lock`	0	0%

4.3 Deriving Locking-Rule Hypotheses

Locking rule derivation for a variable (step 2) starts by enumerating all locking rule hypotheses. For example, Tab. 2 shows locking hypotheses for write operations on `minutes` (second column). The construction algorithm first finds all transactions in which `minutes` is written (based on column *WoR* in Tab. 1) and inserts the associated locks into a set of related lock objects. Then all subsets of these locks are enumerated in every possible order. The third column shows the absolute number of observed rule-complying accesses/transactions that are found in the trace – the *absolute support* s_a. It is important to note that even though transaction *b* has the associated locks `sec_lock` *and* `min_lock`, we also consider it possible that *only* `min_lock`, *only* `sec_lock`, or *no lock* have been actually needed for safely writing to `minutes`.[4] In the example, transaction *b* was executed 16 times (1000/60), which contributes to #0, #1, #2, and #3. Hypotheses #0 and #1 are also supported by the one faulty transaction that only acquired `sec_lock` before changing `minutes`. Column four shows the *relative support* s_r. This is the quotient of s_a and the number of transactions in which the variable was accessed – 17 in the example. Both metrics – s_a and s_r – can be used to quantify the confidence we can put in the hypothesis, as they grow with the number of observations "backing" it.

As a naïve approach for identifying the true locking rule, one could choose the hypothesis with the highest (absolute or relative) support if the relative support exceeds an empirically determined accept threshold t_{ac}, e.g. 90%. However, this leads to two problems: First, the "no lock needed" rule (#0) would always win, as no transaction is regarded as a counterexample. Second, even if a special treatment of the "no lock" hypothesis would fix the first issue, hypothesis #1 would dominate #2. This contradicts with the fact that in the example #2 is the correct locking rule for writes to `minutes`. The reason for the wrong rule being preferred is that the true locking rule is not a counterexample for the wrong alternative rule. Hence, the approach would be unable to detect such bugs.

To overcome this problem, we propose a different selection strategy: We assume that all hypotheses above the (high)

[4] If there were many other observations with only one of these locks held while accessing `minutes`, transaction *b* shall not be regarded as a counterexample.

accept threshold ($s_r \geq t_{ac}$) are related. For example, all observations that contributed to #2 also contributed to #1 and #3. In the described fault scenario the wrong rule would have a higher support value. Therefore, we choose the rule with the lowest support from this group ($t_{ac} \leq s_r \leq 1$). If – as for #2 and #3 – the support values are equal, the hypothesis with more locks is selected. As the "no lock" rule is always above the threshold, a result will always be found.

In summary: The LockDoc approach is to assume that locking in the target operating systems, such as Linux or similar, is done correctly most of the time. Thus, we can automatically derive the locking rules from an execution trace. Rare deviations from the normal behavior can be detected and might be a symptom of bugs.

5 Design: LockDoc for Linux

As an aid in improving locking in complex software systems – discussed on the example of the Linux kernel in Sec. 1 and 2 – LockDoc's goal is to improve locking-related documentation, and to uncover potential bugs. The basic idea has already been explained in Sec. 4.

This section discusses concrete prerequisites and assumptions of our approach when using it for Linux, dissects separate analysis phases, and describes how we harness the results to check preexisting (incomplete, and potentially wrong) locking rules, to fill in the documentation gaps with locking rules for currently undocumented data-structure members, and to look for potential locking bugs in the code.

5.1 LockDoc Workflow Overview

The LockDoc approach comprises three phases, as depicted in Fig. 5:

❶ The *monitoring/tracing phase* runs the target system in a VM environment, and monitors and records several types of events for a defined set of data structures: Dynamic memory allocations/deallocations, read/write accesses to memory occupied by such allocations, and lock acquisitions/releases. The output is a complete *trace* of these events during a run of a specific workload. During the following post processing, the *trace* is loaded into a relational database.

❷ The *locking-rule derivation phase* takes this trace as an input: It analyzes each recorded memory access, determines which data structure and member it addressed, and which locks were held during that access. From this information, the *locking-rule derivator* generates a set of locking-rule hypotheses, i.e. proposals for locking rules for each data-structure member. From these hypotheses, the tool picks the most likely one (see Sec. 4.3) for each member and writes it to a set of *generated locking rules*. However, LockDoc is currently limited to locking-rule derivation for individual members, which is also the scope the Linux locking documentation is limited to. It does not consider relations between members, e.g. for a consistent view both members a and b have to be accessed with lock c.

❸ The *analysis phase* takes the outputs of phases ❶ and ❷ and processes them further to aid the developer in dealing with the particular locking-related problems discussed in Sec. 1 and 2: The *locking-rule checker* takes the *officially documented* locking rules (e.g. from the Linux-kernel source code) and checks each rule whether it matches all – or at least the majority of – the observed events in the trace. The *documentation generator* takes the *generated* locking rules and generates human-readable output that could replace or extend the official documentation. The *rule-violation checker* helps identifying member accesses that do not adhere to the complete set of locking rules, and tracking them to the source-code line that caused them.

The following sections provide details on the design of the tools involved in each phase.

5.2 Target-System Monitoring/Tracing

Monitoring and tracing of dynamic-memory allocations/deallocations and acquisitions/releases of locks for a specific data type is easily accomplishable by instrumenting the target system's source code with logging functionality. However, logging arbitrary memory accesses to instances of the observed data structures is harder – especially for systems implemented in C and assembly.

Consequently, LockDoc uses a hybrid approach. The target system is instrumented to log heap allocations and lock operations. It then runs in a VM environment that is capable of recording memory accesses. In essence, this VM environment passes through events emitted by the source-code instrumentation to a trace file, and additionally logs accesses to memory that belongs to dynamic allocations of observed data structures during their lifetime.

5.3 Trace Post-Processing

For locking-rule derivation and the later analysis steps, the raw event trace is in phase ❶ transformed into a relational database with relations suitable for analysis queries. The database schema is presented (slightly simplified) in Fig. 6: Memory *accesses* go to one of many *allocations*, which in turn are instances of the observed *data_types*. Each data type – in the form of the *type_layout* – comprises a set of members with a size in bytes at a specific struct offset. In case one or more locks were held during a memory access, it is assigned a transaction (table *txns*) that refers to all held *locks* in locking order. Each lock, in turn, may be embedded in an *allocation*. Additionally, each memory access has an associated function-call *stack_trace* to determine the concrete context of an access with insufficient locking later on (e.g., when hunting for bugs in Sec. 7.5).

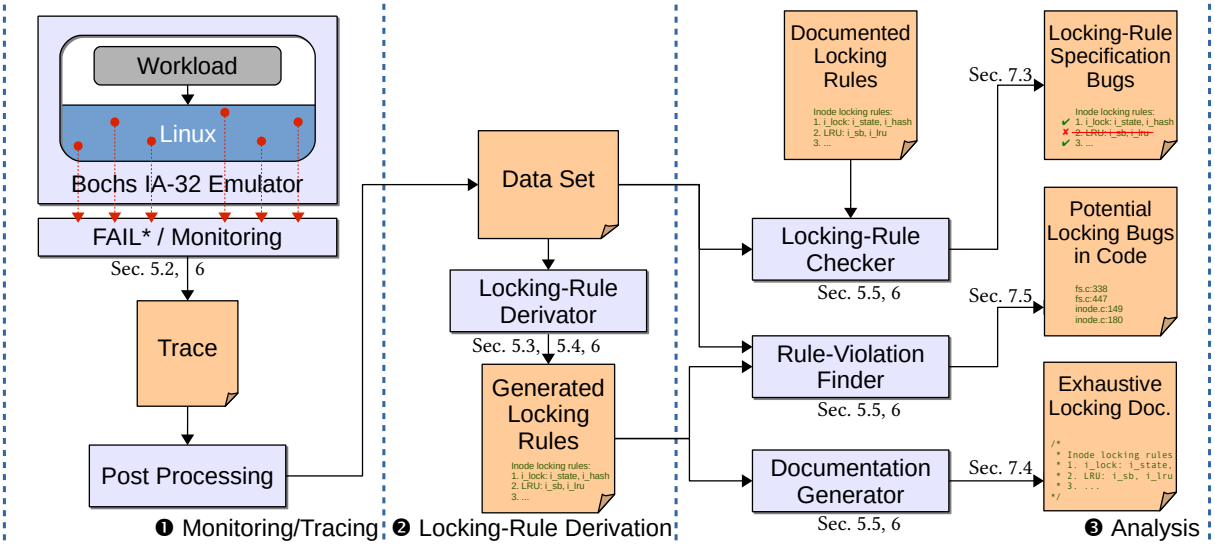

Figure 5. LockDoc overview: Based on a memory-access and lock acquisition trace from the Linux kernel (phase ❶), we infer the most probable locking rules for a specific set of data structures (❷). Using this information, we look for locking-rule documentation bugs, locking bugs in the code, and generate exhaustive locking documentation (❸).

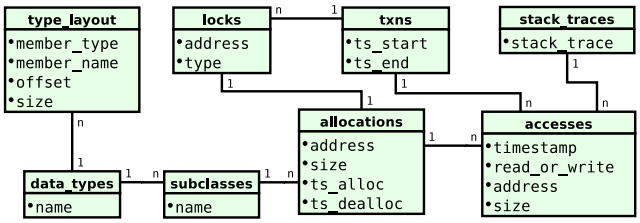

Figure 6. LockDoc database schema storing the recorded trace in a structured form, revolving around the central memory *accesses* table.

In the post-processing step, we also deal with three properties of real-world system software that go beyond the doctrinal locking theory outlined in the previous sections:

1. Although many system-programming languages – such as C – do not provide explicit language support for class inheritance, this OOP mechanism can be emulated. For example, the Linux kernel implements filesystem-specific subclassing of `struct inode` by means of struct nesting, function pointers, and the `i_private` member variable intended for filesystem-specific "private" use. It turns out that `inode` member accesses are also *synchronized* differently throughout Linux's filesystems; for example, the *proc* filesystem does not lock-protect some members because it only implements a subset of all filesystem operations. Consequently, we handle data-structure subclassing in LockDoc by recording each object's subclass in the monitoring/tracing phase (table *subclasses* in Fig. 6), allowing us to derive `struct inode` locking rules separately for, e.g., the proc and the ext4 filesystem.
2. Object initialization and teardown is usually done without taking otherwise necessary locks – deliberately violating locking rules – because such objects are not visible to concurrent control flows. Including member-variable accesses from these phases in the objects' lifetime would possibly lead to wrong conclusions: Locking-rule derivation might wrongly conclude that these members can be accessed without locks, or the rule-violation finder would mark these accesses as potential bugs. Consequently, we maintain a list of (de)initialization functions and filter out member accesses emitted in their contexts.
3. Additionally, we filter out memory accesses to members we define to be out of scope for our experiments (e.g., nested structures that are related to parts of the system we do not observe), or we know to need no locking at all: members defined with the `atomic_t` data type[5], or lock variables themselves. We also filter accesses to other (usually integer) members done via e.g. `atomic_read()`, as they are meant to explicitly bypass the "official" locking mechanisms for performance reasons.

5.4 Locking-Rule Derivation

Informally, a locking rule specifies a set of locks and a lock ordering required for a read or write access to a particular data-structure member. As discussed in Sec. 2.3, holding only a strict subset of the specified locks during an access creates potential for a data race, while taking locks in the wrong order can create a dead- or livelock. However, holding

[5] `atomic_t` members might theoretically also need locks if they are part of a larger critical section.

additional and potentially completely unrelated locks not specified in a locking rule is not harmful, and can be considered a common case when, e.g., conducting data-structure accesses in a nested context. If, for example, a locking rule specifies that locks a and b must be taken in the order $a \rightarrow b$, a memory access with the locks $a \rightarrow c \rightarrow b$ held – and taken in this order – *complies* with the locking rule: All locks in the locking rule are held, and a was taken before b.

The locking-rule derivation builds on the previously mentioned assumption that the observed system accesses data structures with the correct locks held at least *most* of the time. This means that for the searched locking rule, i. e. the correct set of locks in the right order for a specific data-structure member, there must be a high number of memory-access observations that comply with it.

Consequently, the *locking-rule derivator* (Fig. 5) exhaustively enumerates locking-rule hypotheses for each data-structure member, and determines the metric for each hypothesis's s_a and s_r.

As complex software systems such as the Linux kernel use a large number of fine-grained locks, it is obviously not possible to naïvely iterate over all possible lock combinations and determine their support in the memory-access observation trace. Instead, the locking-rule derivator iterates over all *observed* lock combinations (or, transactions), and generates hypotheses by enumerating all subsets of each combination. This ensures that all possible hypotheses with $s_a \geq 1$ are enumerated and quantified in a reasonable time frame.

5.5 Analysis Phase

The three analysis tools in phase ❸ build on results of the previous phases – the structured trace information in the database, and the generated locking rules – and puts them to practical use for the kernel developers.

For the *Locking-Rule Checker*, the officially documented locking rules – which are put to trial in this step – first need to be manually converted into LockDoc's internal locking-rule notation. The tool then determines the absolute and relative support s_a and s_r for each rule, and categorizes each rule as either *correct* ($s_r = 1$), *ambivalent* ($0 < s_r < 1$), or *incorrect* ($s_r = 0$). Ambivalent or incorrect rules are inconsistently or never followed in the code, and should be either reexamined by kernel developers or replaced by our generated rules.

The *Documentation Generator* takes the generated locking rules with sufficient s_r from the previous phase and generates human-readable documentation that can, e.g., replace currently documented but ambivalent/incorrect rules, or add new documentation for data-structure members that were not documented before.

The *Rule-Violation Finder*, in contrast to the Locking-Rule Checker, assumes the *generated* locking rules to be correct, and scans the event trace for member accesses that violate their associated rule. With the help of additionally recorded data and debug information, the developer is presented necessary information to track down this possible locking bug: the data-structure and member name, the locks that *should* have been held (i.e., the locking rule), the locks that *actually* were held, and the context the access originated from – source-code file and line, and a stack trace from that moment.

6 Implementation

This section provides implementation details on target-system monitoring, database import and filtering, the locking-rule derivator, and the analysis tools harnessing the resulting locking-rule set.

Target-System Monitoring For target-system monitoring, we use the FAIL* fault-injection framework [26] (v. 1.0.1) as a means to automatically run the target system in a virtualized, single-core environment (in our case the Bochs x86 emulator [16] in version 2.4.6), and to monitor and control it from a FAIL* *experiment description*. This monitoring/controlling experiment is implemented in 646 lines[6] of C++.

As described in Sec. 5.2, memory allocation/deallocation and lock-operation events in the target system are instrumented; these events are communicated through a virtual I/O port to the experiment implementation, where they are logged in the trace accompanied by a time stamp. Additionally, each allocation event for a data-structure we want to monitor enables a *MemoryAccessListener* via FAIL*'s API that ensures that future accesses to the data structure are logged as well. Vice versa, the corresponding deallocation disables this listener again.

Regarding the target system itself, few changes are necessary to monitor new Linux data types after the initial, manual effort of instrumenting the APIs for the different lock types: Just two function calls have to be added per data type, as Linux uses its own standard interfaces for allocation/deallocation of central data types. The chance of missing lock operations or allocations due to code ignoring these interfaces is rather low, as their usage is strongly enforced for code contributions – especially to central subsystems like the VFS layer.

To ensure no optimization takes place, i.e. eliding spinlocks[7], we built the Linux kernel with SMP support[8]. Moreover, the module support is disabled to get one statically linked binary.

Trace Post-Processing Trace post-processing, filtering and database import – as described in Sec. 5.3 – is implemented in 1490 lines of C++, and a small shell script that coordinates and automates the whole post-processing step. The script runs the C++ tool with parameters from a separate configuration file, including "black lists" of data-structure

[6] Effective lines of code (excl. empty lines/comments), counted with *cloc* [8].
[7] https://www.kernel.org/doc/htmldocs/kernel-locking/uniprocessor.html
[8] The configuration option `CONFIG_PREEMPT` is not set.

members and functions to be filtered out, and subsequently imports several generated CSV tables into a MariaDB database. The function black list comprises 99 entries for 9 different data types plus 58 globally ignored functions, e.g. `atomic_inc()`, whereas the data-structure member black list holds 30 entries.

Locking-Rule Derivator The locking-rule derivator is implemented in 612 lines of C++. It provides several command-line switches configuring the accept threshold t_{ac}, an additional cut-off threshold t_{co} limiting the output to locking-rule hypotheses with a minimum relative support, and several human- and machine-readable report modes.

Analysis-Phase Tools The locking-rule checker is implemented as a 150-line Python script that compares the manually extracted locking rules from the documentation, and a specific summary-mode output of the locking-rule derivator. The documentation generator is completely built into the locking-rule derivator and merely implements a different output format. The rule-violation finder is implemented as a parametrizable SQL statement (executed in MariaDB) and a post-processing script that converts static instruction addresses into human-readable source-code locations.

7 Evaluation

This section evaluates the LockDoc approach with several file-system related data structures in the Linux kernel. In the following we provide details on the evaluation setup, give an impression on the resource requirements of our tooling, and discuss results of locking-rule checking, documentation generation, and rule-violation finding.

7.1 Evaluation Setup

We used vanilla Linux 4.10 as the target system, running a custom set of benchmarks primarily intended to trigger file-system specific code paths in the kernel. We instrumented Linux manually by replacing several locking-related functions (*spinlock_t*, *rw_lock_t*, *semaphore*, *rw_semaphore*, *mutex* and *rcu*) with macros that additionally call a new logging function, which communicates context information – source-code file/line, calling function, lock-variable address, and name of the lock function – to the Fail*-based experiment environment. Additionally, we record lock/release events for synthetic *softirq* and *hardirq* locks. We also modified allocation/deallocation functions for 11 file-system related data structures (listed in the first column of Tab. 6), for example `alloc_inode()`, to log each allocation's start address, size, and data type. For `struct inode`, we additionally log the type of the backing filesystem to realize subclassing.

Note that some Linux-kernel data structures use `union` compounds containing differently named members (for example `i_pipe`, `i_bdev` etc. in the inode data structure) to save memory. To distinguish these members just by the logged memory address we "unrolled" these unions by embedding their contents in the encompassing struct, essentially giving them different offsets within the data type.

Table 3. Code coverage generated by our benchmark mix: Each line summarizes the coverage across all files that are located in the respective directory.

Directory	Line Coverage	Function Coverage
fs	30.95 % (5757/18598)	33.33 % (608/1824)
fs/ext4	31.52 % (4863/15429)	43.67 % (345/790)
fs/jbd2	43.29 % (1106/2555)	43.18 % (76/176)

In search of a benchmark that triggers many file-system related code paths and locking operations, we realized that currently no systematic coverage-testing benchmark suite exists for Linux. A (possibly automatically generated) statement- or path-coverage benchmark suite would be ideal for our purposes, but is currently subject to future work. Instead we resorted to a custom mix of benchmarks with the intention of emitting a wide variety of different system calls: a subset of tests from the Linux Test Project [12] (*fs-bench-test2*, creating files, changing owner/permission, and randomly accessing them; *fsstress*, randomly carrying out I/O operations on a directory tree; *fs_inod*, allocating/deallocating inodes), and our own test programs using pipes, creating/deleting symbolic links, and changing permissions.

The code coverage generated by our custom mix of benchmarks is shown in Tab. 3. We gathered the code coverage using GCOV with the Linux kernel[9]. Each line shows the coverage for all source code files that reside directly in the respective directory.

All evaluation experiments took place on a dual-socket Intel® Xeon® Gold 6152 machine, while the database server ran on a dual-socket Intel® Xeon® E5345.

7.2 Tracing and Locking-Rule Derivation

Monitoring and tracing of the target system/benchmark setup in Fail* took about 34 minutes (3-4 seconds without instrumentation) and produced a trace with about 27.4 million recorded events – 13 million locking operations, 14.4 million memory accesses (13.9 million remain after filtering), 33,606 allocations and 18,660 deallocations. The LockDoc database lists a total of 41,589 different locks, 821 of them statically allocated and 40,768 as part of dynamically allocated data structures. Filtering and database import ran for another 8 minutes, the query generating the locking-rule derivator input took 77 minutes, and locking-rule derivation itself finished in 3.02 s. Extraction of all counterexamples from the database took 172 minutes.

Table 4. Summary of validated locking rules: Each row shows how many locking rules are documented (#R), and how many of the corresponding members have not been observed (#No) and observed (#Ob). The last three columns denote the portion of correct ($s_r = 1$), ambivalent ($0 < s_r < 1$) and incorrect ($s_r = 0$) rules (cf. Sec. 5.5).

Data Type	#R	#No	#Ob	✓ (%)	~ (%)	✕ (%)
inode	14	3	11	18.18	45.45	36.36
journal_head	26	3	23	56.52	17.39	26.09
transaction_t	42	13	29	79.31	13.79	6.90
journal_t	38	8	30	56.67	33.33	10.00
dentry	22	0	22	27.27	63.64	9.09

Table 5. Overview of check rules for `struct inode`: ES indicates that the mentioned lock is embedded in the same data structure instance the member access goes to, while `inode_hash_lock` is a global lock.

Member	r/w	Locking Rule	s_r	OK?
i_bytes	w	ES(inode.i_lock)	100 %	✓
i_state	w	ES(inode.i_lock)	100 %	✓
i_hash	w	inode_hash_lock→ ES(inode.i_lock)	98.1 %	~
i_blocks	w	ES(inode.i_lock)	93.56 %	~
i_lru	r	ES(inode.i_lock)	50.6 %	~
i_lru	w	ES(inode.i_lock)	50.39 %	~
i_state	r	ES(inode.i_lock)	19.78 %	~
i_size	r	ES(inode.i_lock)	0 %	✕
i_hash	r	inode_hash_lock→ ES(inode.i_lock)	0 %	✕
i_blocks	r	ES(inode.i_lock)	0 %	✕
i_size	w	ES(inode.i_lock)	0 %	✕

7.3 Locking Rule Checking

To illustrate the practicality of our approach, we now present the results from comparing existing locking-rule documentation with the rules learned by LockDoc. For this experiment we chose five relatively well documented kernel data structures from the VFS layer, namely the `inode`, `dentry`, `journal_t` as well as `transaction_t` and `journal_head` structs from the filesystem-independent Journaling Block Device (JBD2) used by the OCFS2 and ext4 filesystems. The documentation was found in the first lines of `fs/inode.c` and `fs/dcache.c`, in `include/linux/journal-head.h`, in `include/linux/dcache.h` (line 83 ff.), and in `include/linux/jbd2.h` (around lines 543 and 795). Especially for the latter three structs almost all member variables are commented with locking requirements.

[9] https://www.kernel.org/doc/html/v4.10/dev-tools/gcov.html

Tab. 4 shows a summary of our results. In total the Linux documentation contains 142 locking rules (column #R), covering 71 members as we handle read and write accesses separately. Column #No contains the number of rules for which LockDoc did not produce any result, because the benchmark code did not trigger any access to the respective member variables. For the other cases (column #Ob) LockDoc could approve the documentation in 18.18%, 79.31%, 56.67%, 27.27%, and 56.52% of the rules, respectively (column ✓). An approval means that 100% of the observations followed the rule. The next column (~) shows the share of members for which the documented rule was not always followed. Finally, the column ✕ shows for how many members the rule was not obeyed at all. Tab. 5 shows an example of the detailed results for `struct inode`.

Kernel developers can learn from these results that either the documentation needs to be improved – or that the code does not follow the rules, which might have the aforementioned negative effects such as performance degradation or locking-related bugs. As there is no authoritative "ground truth" besides the cumulative knowledge of all involved developers, we cannot decide whether the documentation or the code is wrong without submitting patches and hoping for a thorough kernel-community review process.

Furthermore, the statistics can be used as an indicator for documentation quality. A visual inspection quickly shows that the structures `transaction_t` and `journal_t` are more thoroughly documented than `inode` and, hence, the LockDoc results are better. However, we identified three members of `transaction_t` that have been transformed from an `int` into an `atomic_t` without updating the respective locking documentation.

7.4 Locking Rule Mining

We now present our results regarding yet undocumented data-structure members. Our *generated* locking rules can be used to create new documentation for every data type listed in Tab. 6, which also summarizes the results: Column #M shows the number of members and #Bl how many of them are black-listed/filtered (see Sec. 5.3). #Rules shows how many locking rules with sufficient support ($s_r \geq t_{ac}$, with $t_{ac} = 0.9$) could be generated, and #Nl (= *No* Lock) counts the subset of #Rules indicating that accesses to their corresponding member does not need a lock at all.

Of course, decreasing the acceptance threshold below $t_{ac} = 0.9$ would allow us to accept even more locking-rule hypotheses that differ from "no lock", and in turn generate more documentation. We adopted this threshold from Engler et al., who successfully used $p_{correct} = 0.9$ in their statistical analysis on finding semantic bugs [10]. We discuss the selection of t_{ac} later in this section.

Tab. 6 shows that a certain number of members per data type is seemingly accessible without locks. The reason is ambivalent: a) The member can really be accessed without locks,

Table 6. Summary of mined locking rules for 11 data types and 10 subclasses of `struct inode`: the number of members (#M), the number of black-listed/filtered members (#Bl), the number of members we actually generated locking rules for (#Rules). #Nl counts the subset of #Rules indicating that the corresponding member does not need any lock at all (similarly divided into rules for read and write accesses).

Data Type	#M	#Bl	#Rules		#Nl	
			r	w	r	w
backing_dev_info	43	2	25	20	11	3
block_device	21	2	14	15	6	6
buffer_head	13	0	10	8	7	5
cdev	6	0	2	6	2	4
dentry	21	1	19	18	13	6
inode:anon_inodefs	65	5	11	2	8	0
inode:bdev	65	5	24	18	14	6
inode:debugfs	65	5	0	1	0	0
inode:devtmpfs	65	5	32	24	26	5
inode:ext4	65	5	45	30	36	4
inode:pipefs	65	5	30	7	29	3
inode:proc	65	5	33	10	31	2
inode:rootfs	65	5	38	19	35	3
inode:sockfs	65	5	19	3	17	0
inode:sysfs	65	5	30	14	26	1
inode:tmpfs	65	5	37	20	29	3
journal_head	15	0	13	12	6	0
journal_t	58	11	34	20	21	1
pipe_inode_info	16	1	13	7	4	0
super_block	56	3	35	8	21	2
transaction_t	27	1	20	16	9	1

or b) the correct hypothesis has low relative support. We have several conjectures regarding the reasons for locking rules with particularly low absolute or relative support:

- Low *absolute* support – with the extreme case of $s_a = 0$ – is relatively clearly caused by the benchmarks' inability to systematically trigger accesses to that particular data-structure member often enough. We believe this could be remedied with better benchmarks, especially ones that aim at improved kernel-code coverage.
- Low *relative* support, however, could mean several things: We missed instrumenting a part of the kernel code (and the *true* s_r is in fact a lot higher), our initial assumption that the code gets locking right at least *most* of the time does not hold (and we found a potential locking-related kernel bug, but cannot decide automatically what locking behavior would be correct), or we hit general limitations of our approach. Unfortunately, either requires deeper manual inspection of the observations and the related source code, and will involve interviewing domain experts. We intend to follow this road in future work.

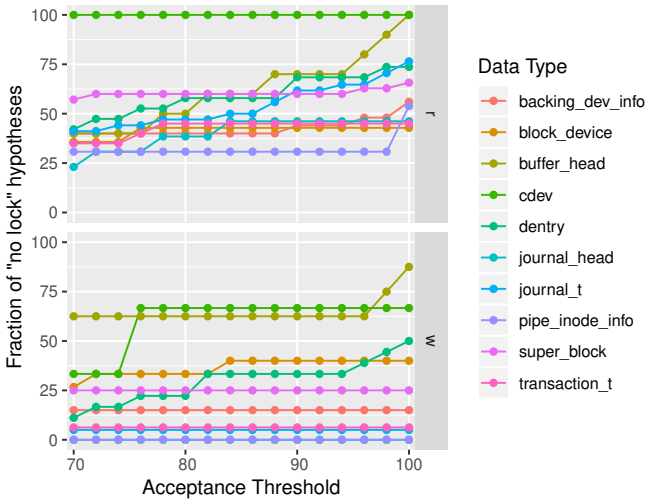

Figure 7. Fraction of "no lock" locking-rule hypotheses for different acceptance thresholds t_{ac}, separately plotted for each observed data type and read vs. write accesses.

One example that may fall into the "potential locking-related bug" category revolves around `inode.i_hash`, for which we generate a locking rule claiming only the global `inode_hash_lock` needs to be held to modify it. The kernel documentation claims `i_hash` is protected by the embedded `i_lock`; the kernel source code, however, seems to take *both* locks, for example in `__remove_inode_hash()`. Looking closer reveals that this function, which removes an inode from a doubly linked list, writes to `i_hash` of *three* inode instances: The to-be-removed one it holds the `i_lock` for, and its predecessor and successor in the list whose `i_lock` is *not* held. In effect this leads to numerous write accesses to `i_hash` without the corresponding `i_lock` held, allowing LockDoc to conclude that it is not needed for this operation – but contradicting both the documentation and parts of the kernel source code. This confusion can only be cleared up by a kernel expert.

As described in Sec. 4.3, t_{ac} divides the set of hypotheses in those being relevant and those considered to be noise. The choice of t_{ac}, therefore, impacts the outcome of our approach in several ways: Due to a higher (or lower) t_{ac} , a different hypothesis might be chosen as winner. That in turn leads to a different documentation. That again advises a kernel developer to use different locks. Finally, based on the winning hypothesis different locking rule violations are found. To sum it up, the value of t_{ac} is crucial to our approach.

To show the actual impact, we examined the number of "no lock" hypotheses. As stated above, LockDoc can end up selecting the "no lock" hypothesis over hypotheses with locks. Besides the fact that really no locks are needed to a access a specific member, the "no lock" hypothesis can be selected if no hypothesis exists with $t_{ac} \leq s_r$. Consequently, the choice of t_{ac} and the number of "no lock" rules directly

```
/*
 * inode locking rules:
 *
 * No locks needed for:
 *   i_data.a_ops, i_data.nrexceptional, i_rdev,
     i_data.gfp_mask, i_generation, i_security,
     i_nlink, i_flctx, i_size, i_atime, i_mtime,
     i_data., i_data.host, i_sb
 *
 * EO(wb.list_lock in backing_dev_info) protects:
 *   dirtied_when, i_io_list
 *
 * EO(i_rwsem in inode) protects:
 *   i_op, i_link, i_fop, i_acl, i_default_acl,
     i_private
 *
 * EO(s_umount in super_block) protects:
 *   i_data.writeback_index
 *
 * ES(i_rwsem in inode) protects:
 *   i_flags, i_uid, i_gid, i_version, i_ctime,
     i_size_seqcount
 * [...]
 */
```

Figure 8. An excerpt of the exemplarily generated locking rule documentation for `fs/inode.c`: EO ("**e**mbedded **o**ther") indicates a lock embedded in another object, ES ("**e**mbedded **s**ame") a lock embedded in the same object that holds the protected variable.

affect each other: A higher value of t_{ac} intuitively creates higher confidence in the winning hypothesis, but also increases the number of "no lock" rules – i.e., more hypotheses with locks are rejected. Fig. 7 depicts the fraction of "no lock" rules chosen as winner for $0.7 \leq t_{ac} \leq 1$ for ten different data types (excluding the 11 `inode` subclasses for clarity). For many of the data types the fraction levels off at 90% with a high s_r. For some data types, the fraction of "no lock" rules never reaches 100%. For only 50% of the members of `dentry` – for write accesses – the "no lock" rule is the winner having $t_{ac} = 1$, for example. That implies that the other half has a winning hypothesis with $s_r = 1$ providing a high confidence. However, t_{ac} needs some refinement in the future once the body of resulting hypotheses is manually reviewed and verified by kernel experts.

Based on the mined locking rules, we can than generate locking documentation to replace (or update) the existing documentation. Fig. 8 gives an example of what such documentation looks like. When accepted as a patch, it could update the existing documentation in `fs/inode.c` that has already been presented in Fig. 2.

7.5 Locking Rule Violations

For each generated rule from Sec. 7.4 that has a relative support $s_r < 1.0$, the rule-violation finder can locate memory-access events that violate this rule. Tab. 7 shows a summary of violations per data type with the number of rule-violating memory access events, the data-structure members involved, and the number of distinct contexts (source locations and associated stack traces, i.e. one location would be counted multiple times if it was reached from different functions) the violations occurred in.

Table 7. Summary of locking-rule violations: Number of violating memory-access events (total: 52,452 events at 986 contexts), associated members, and number of distinct contexts.

Data Type	Events	Members	Contexts
backing_dev_info	267	4	61
block_device	1	1	1
buffer_head	45325	4	635
cdev	0	0	0
dentry	749	5	58
inode:anon_inodefs	0	0	0
inode:bdev	5	2	5
inode:debugfs	0	0	0
inode:devtmpfs	29	2	10
inode:ext4	355	6	28
inode:pipefs	0	0	0
inode:proc	0	0	0
inode:rootfs	1720	5	42
inode:sockfs	0	0	0
inode:sysfs	57	1	21
inode:tmpfs	59	4	12
journal_head	0	0	0
journal_t	3845	7	99
pipe_inode_info	9	3	5
super_block	31	3	9
transaction_t	0	0	0

Table 8. Locking-rule violation examples: For each violation, LockDoc provides information in which context the access happened (stack-trace omitted), and which locks were – against the underlying locking rule – held at that time. Analogously to ES meaning "embedded in the same object", EO indicates embedding in another object.

Data Type/Member	Locks held	Location
inode:ext4.i_hash	inode_hash_lock→ EO(i_lock)	fs/inode.c:507
journal_t.j_committing_ transaction	EO(i_rwsem) → ES(j_state_lock)	fs/ext4/inode.c: 4685
dentry.d_subdirs	EO(i_rwsem)→rcu	fs/libfs.c:104

Tab. 8 gives an impression what information the rule-violation finder can present the developer as a starting point to look into a possible locking-related kernel bug: For example the `i_hash` access seems to add another case to the locking-rule mystery discussed in the previous subsection.

Note that not all 52,452 rule-violating memory access events (Tab. 7) are necessarily symptoms of *real* synchronization bugs. False positives can be caused by several issues:

- Linux-kernel code tends to deliberately violate its own locking rules for performance reasons. For example, word-sized variables are sometimes read without locks when consistency with other data is not relevant. Although we filter accesses to `atomic_t` variables and atomic memory accesses using specialized functions like `atomic_read()` (see Sec. 5.3), parts of the code make such accesses by other means.
- Similarly, parts of the kernel omit locking primitives when it is clear from the context that no concurrent accesses can happen. We already filter the most prevalent case – object initialization and teardown (see Sec. 5.3) – but know from feedback by domain experts that there exist more, data-structure specific contexts that allow skipping the locks.
- The noise caused by these issues, combined with a suboptimally chosen t_{ac} (see Sec. 7.4), can lead to LockDoc picking the wrong locking hypothesis; memory accesses violating such a wrong locking rule are also probably false positives.

Currently, false positives must be identified manually with the help of domain experts. Without a reliable "ground truth" and under a flexible interpretation of what's "correctly" synchronized, any attempts of estimating the false-positive rate are futile. Of the violations presented, one bug[10] has already been confirmed by a kernel developer: the member `i_flags` in `struct inode` is accessed without proper synchronization in one code path, as the developer comment in Fig. 3 already suspected.

8 Conclusions

In huge and highly optimized software projects such as the Linux kernel, where hundreds of developers with different levels of experience use and extend each other's code, up-to-date and complete documentation is essential. In the case of locking rules we found that learning them from examples in the running system is a promising approach to check existing documentation, generate new documentation, and to find rule violations – potential bugs. The LockDoc approach is by no means specific to the Linux kernel and could be applied to other projects with concurrent control flows and huge numbers of locks – today more the rule than the exception especially in the operating-systems domain.

As our next step, we intend to extend the still rather simplistic model behind our locking rules: a sequence of locks – global, embedded within the same object, or member of "some" other object – to be held before a member access. This model in particular does not yet capture object interrelations, which we believe might further improve result quality and allow deriving rules such as *"acquire lock L in the list head before accessing a member of a list element"*.

[10] https://lkml.org/lkml/2018/12/7/532
https://lkml.org/lkml/2018/12/14/277

Acknowledgments

The authors would like to thank the referees for their valuable comments and helpful suggestions. We would also like to thank Jan Kara who gave us valuable feedback. Part of the work on this paper has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A1, and grant no. SP 968/9-1.

References

[1] Rahul Agarwal and Scott D. Stoller. 2006. Run-time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD '06)*. ACM Press, Portland, Maine, USA, 51–60. https://doi.org/10.1145/1147403.1147413

[2] Mikael Åsberg, Johan Kraft, Thomas Nolte, and Shinpei Kato. 2010. A loadable task execution recorder for Linux. In *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*. Brussels, Belgium.

[3] Thomas Ball and Sriram K. Rajamani. 2002. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM Press, Portland, Oregon, 1–3. https://doi.org/10.1145/503272.503274

[4] Daniel Pierre Bovet and Marco Cesati. 2005. *Understanding The Linux Kernel* (3rd ed.). O'Reilly Media Inc.

[5] Peter T. Breuer and Marisol García Valls. 2004. Static Deadlock Detection in the Linux Kernel. In *Reliable Software Technologies – Ada-Europe 2004*, Albert Llamosí and Alfred Strohmeier (Eds.). Springer, 52–64. https://doi.org/10.1007/978-3-540-24841-5_4

[6] Ray Bryant and John Hawkes. 2000. Lockmeter: Highly-informative Instrumentation for Spin Locks in the Linux® Kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference – Volume 4 (ALS '00)*. USENIX Association, Atlanta, Georgia, 17–17.

[7] Guancheng Chen and Per Stenstrom. 2012. Critical Lock Analysis: Diagnosing Critical Section Bottlenecks in Multithreaded Applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Press, Salt Lake City, Utah, Article 71, 11 pages.

[8] Al Danial. [n. d.]. cloc: Count Lines of Code. https://github.com/AlDanial/cloc. Accessed: 2018-01-24.

[9] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM Press, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[10] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM Press, New York, NY, USA, 57–72. https://doi.org/10.1145/502034.502041

[11] Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*. ACM Press, New York, NY, USA, 54–61. https://doi.org/10.1145/379605.379665

[12] Cyril Hrubis et al. [n. d.]. Linux Test Project. https://github.com/linux-test-project/ltp. Accessed: 2018-01-24.

[13] Yongbing Huang, Zehan Cui, Licheng Chen, Wenli Zhang, Yungang Bao, and Mingyu Chen. 2012. HaLock: Hardware-assisted Lock Contention Detection in Multithreaded Applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM Press, Minneapolis, Minnesota, USA, 253–262. https://doi.org/10.1145/2370816.2370854

[14] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *Proceedings of the IEEE PressSymposium on Security and Privacy*. 279–293. https://doi.org/10.1109/SP.2019.00017

[15] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. 2008. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. In *Proceedings of the 8th Symposium on Operating System Design and Implementation (OSDI '08)*. USENIX Association, San Diego, CA, USA, 295–308.

[16] Kevin P. Lawton. 1996. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal* 1996, 29 (Sept. 1996), 7.

[17] Wenke Lee, Salvatore Stolfo, and Philip K. Chan. 1997. Learning Patterns from Unix Process Execution Traces for Intrusion Detection. In *Proceedings of the Workshop on AI Approaches to Fraud Detection and Risk Management (AAAI)*. AAAI Press, Providence, RI, USA. https://doi.org/10.7916/D8B56RF2

[18] Geoffrey Lefebvre, Brendan Cully, Christopher Head, Mark Spear, Norm Hutchinson, Mike Feeley, and Andrew Warfield. 2012. Execution Mining. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. ACM Press, London, England, UK, 145–158. https://doi.org/10.1145/2151024.2151044

[19] Robert Love. 2010. *Linux Kernel Development* (3rd ed.). Addison-Wesley, Boston, MA, USA.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM Press, Chicago, IL, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[21] Gabriel N. Matni and Michel R. Dagenais. 2011. Operating System Level Trace Analysis for Automated Problem Identification. *Open Cybernetics & Systemics Journal* 4 (2011), 45–52.

[22] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. 2014. *The Design and Implementation of the FreeBSD Operating System* (2nd ed.). Addison-Wesley Professional.

[23] Ingo Molnar and Arjen van de Ven. [n. d.]. Runtime locking correctness validator. https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt. Accessed: 2018-01-24.

[24] Rusty Russell. [n. d.]. Kernel Hacking Guides: Unreliable Guide To Locking. https://www.kernel.org/doc/html/v4.15-rc9/kernel-hacking/locking.html. Accessed: 2018-01-24.

[25] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transaction on Computer Systems* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[26] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. 2015. FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance. In *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*. IEEE Press, Piscataway, NJ, USA, 245–255. https://doi.org/10.1109/EDCC.2015.28