Beastie In For Checkup: Analyzing FreeBSD with LockDoc

Alexander Lochmann
TU Dortmund
Department of
Computer Science 6
Dortmund, Germany

Horst Schirmeier
TU Dortmund
Department of
Computer Science 12
Dortmund, Germany

ABSTRACT

LockDoc is an approach to extract locking rules for kernel data structures, based on a dynamic execution trace. The recorded trace can e.g. be used to verify existing locking documentation. LockDoc results for Linux indicated that only 53 % of all examined data types were accessed consistently with their respective locking documentation [5]: Linux systematically elides locks for performance reasons, and the existing documentation is partially outdated or inconsistent. Without a solid "ground truth", it is impossible to reliably attribute LockDoc's findings to bugs in Linux, or to issues with the LockDoc approach itself.

Therefore, in this paper we present results from applying LockDoc to a much more straightforwardly and "cleanly" implemented operating system: FreeBSD offers sophisticated locking documentation – e.g. for many data structures, each individual field is annotated with a precise locking rule. We report that, for four centrally documented data types, FreeBSD adheres to the documented locking rules in 72.4 % of all dynamic data-structure accesses. Investigating the remaining rule-violating accesses, we already triggered two commits for the FreeBSD kernel fixing unprotected accesses, and nudge this value to 73.6 %.

KEYWORDS

Synchronization bugs, Locking documentation, Trace-based, Locking-rule extraction

1 INTRODUCTION

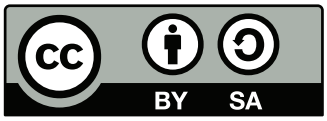
The increasing trend to SMP machines over the past two decades pushed operating systems, such as Linux or FreeBSD, towards a more and more fine-grained synchronization on the granularity of kernel subsystems and even portions of single data structures [1, 2, 6, 7]. Fine-grained locking, however, is error-prone and has led to numerous synchronization bugs in the past. This risk can be reduced by a consistent and sound locking documentation, which is not always given.

LockDoc [5] is a trace-based approach aiming at alleviating this situation. From a recording of lock operations and data-structure accesses in a running OS kernel under stress, it derives *locking rules* – a concrete, ordered sequence of locks that must be taken before accessing a specific data-structure element. The derived locking rules can consequently be used to locate synchronization bugs, and to validate or generate locking documentation.

In previous work [5], we validated the locking documentation of Linux. Our results showed that only 53 % of all examined data types are accessed with the documented locks held. According to the Linux kernel developers, this has several plausible causes: 1) Word-sized variables can be accessed without locks. 2) If no concurrency takes place – known statically, from a whole-system perspective – locks can be elided. 3) Likewise, no lock is needed if consistency is not required in a particular code location, e.g. a simple NULL-pointer check. However, without a precise and up-to-date locking documentation as a "ground truth", it is hard to reliably attribute LockDoc's findings to bugs in Linux, or to issues with the LockDoc approach itself.

Consequently, in this work we investigate a much more straightforwardly and "cleanly" implemented operating system: FreeBSD offers a sophisticated locking documentation. As exemplified in Listing 1, data type definitions in FreeBSD are often prefixed with a comment introducing the relevant locks, assigning each lock an individual letter that is referenced in the actual type definition: The letter *m*, for example, is used to refer to the *mount point interlock*. It is used in line 7 in Listing 2 to assign a lock to the data-structure element `v_nmntvnodes`. The comment in Listing 1 also contains more information regarding locking order (line 17 ff.).

However, the documentation is still imperfect: It informally *describes* locks instead of explicitly naming them. A reference counter is sometimes mentioned in the text, and sometimes modeled as a dedicated lock in the documentation. Nonetheless, the locking documentation looks promising. We



FGBS '21, September 21–22, 2021, Trondheim, Norwegen

https://doi.org/10.18420/fgbs2021h-04

```
/*
 * Reading or writing any of these items requires
 * holding the appropriate lock.
 *
 * Lock reference:
 *  c - namecache mutex
 *  i - interlock
 *  l - mp mnt_listmtx or freelist mutex
 *  I - updated with atomics, 0->1 and 1->0 transitions
 *      with interlock held
 *  m - mount point interlock
 *  p - pollinfo lock
 *  u - Only a reference to the vnode is needed to
 *      read.
 *  v - vnode lock
 *
 * Vnodes may be found on many lists.  The general way
 * to deal with operating on a vnode that is on a list
 * is:
 *  1) Lock the list and find the vnode.
 *  2) Lock interlock so that the vnode does not go
 *     away.
 *  3) Unlock the list to avoid lock order reversals.
 *  4) vget with LK_INTERLOCK and check for ENOENT, or
 *  5) Check for DOOMED if the vnode lock is not
 *     required.
 *  6) Perform your operation, then vput().
 */
```

Listing 1: Reformatted excerpt of FreeBSD's locking documentation for `struct vnode` (`sys/sys/vnode.h`).

```
struct vnode {
  // [...]
  /*
   * Filesystem instance stuff
   */
  struct mount *v_mount; /* u ptr to vfs we are in */
  TAILQ_ENTRY(vnode) v_nmntvnodes; /* m vnodes for
      mount point */
  // [...]
};
```

Listing 2: A slightly reformatted excerpt of `struct vnode` from the FreeBSD kernel (`sys/sys/vnode.h`): Each element is annotated with a letter corresponding to a certain lock.

therefore applied LockDoc to FreeBSD, and performed the same analysis as for Linux [5]. In this paper, we show parts of those results leading to the following contributions:

- An analysis of the locking documentation for four data types of FreeBSD's virtual filesystem.
- An examination of those rules that seem not fully implemented by the source code. This resulted in two commits fixing unprotected accesses in the FreeBSD kernel.

2 THE LOCKDOC APPROACH

The basic idea behind LockDoc is to automatically learn about synchronization rules in a complex software system solely by observing its dynamic behavior: Source-code annotations may be deceptive or wrong, locking documentation outdated or unavailable. In contrast, the actual running system represents a kind of ground-truth, manifested in machine code – especially in the case of well-tested and widely used operating-system kernels like Linux or FreeBSD.

We observe the system's dynamic behavior using two complementary mechanisms:

- We instrument the target system, 1) to record lock acquisition and release operations, and 2) to notice memory allocation and deallocation operations for the data types we are interested in.
- We run the target system in a full-system emulator that records all memory accesses to instances of these data types.

In the full-fledged, original LockDoc approach [5], we harness the resulting, combined trace for *deriving probable locking rules*. We use these derived rules

(1) for *matching them against available documentation* to reveal documentation bugs,
(2) for *scanning the trace for rule-violating memory accesses* to reveal synchronization bugs,
(3) and for *generating documentation*.

2.1 LockDoc for FreeBSD: Kernel Instrumentation

As FreeBSD's locking documentation – at least for some well-documented data structures – is particularly elaborate and precise, we modified the approach to not scan the trace for memory accesses violating *automatically derived* locking rules: Instead, we formalize the *documented* locking rules in a way compatible to LockDoc's rule-violation checker, and scan the trace for accesses violating those rules (Fig. 1).

Since both LockDoc and FreeBSD's *Witness* system – a built-in dynamic lock-order monitor aiming at detecting certain types of synchronization bugs [1, 7], similar to Linux's *lockdep* [3, 8] – use the same lock model, instrumenting the *Witness* system provides tracing data on all software-level locking mechanisms supported by FreeBSD. Additionally, we instrumented `intr_disable()` / `intr_restore()` to record instances of disabling and enabling CPU-local interrupts – in principle a hardware "lock". To get notice of data-structure allocations and deallocations, we instrument data-structure specific allocator functions, such as `getnewvnode()` (which is called to instantiate a `vnode` structure).

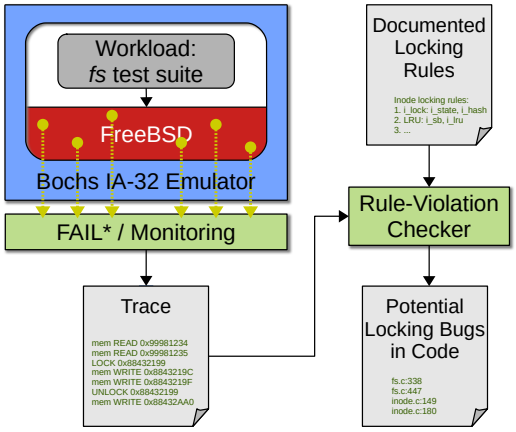


Figure 1: LockDoc workflow excerpt: Based on a memory-access and lock-acquisition *trace* observed in a full-system emulator, and a manually formalized list of *documented locking rules*, LockDoc's *rule-violation checker* detects memory accesses that do not adhere to the documentation. In a perfect world, those accesses are all either indications of synchronization – or locking documentation – bugs.

2.2 Memory-Access Tracing

LockDoc uses the FAIL* fault-injection and monitoring framework [9] as an event-logging communication endpoint for all in-kernel instrumentation (see Fig. 1). Within the experiment flow controlling the FreeBSD monitoring, we additionally record all dynamic memory accesses to data-structure instances under observation. In a post-processing step, we filter out accesses to elements marked as *atomic* in the source code, and also those originating in a list of known initialization or atomic-access functions.

2.3 Locking-Bug Localization

With the trace data and the formalized locking rules in place, LockDoc's rule-violation checker can scan through the trace and look for accesses that do not adhere to the documented locking rule. Once it finds such an access, it harnesses debug information to pinpoint the exact source-code location of the violating access, to provide a stack trace indicating the call history that led to this access, and to give the developer specifics on which locks were not held or taken in the wrong order.

3 EVALUATION

In this section, we evaluate LockDoc on FreeBSD. We describe the evaluation setup (Sec. 3.1) and present results validating the existing locking documentation (Sec. 3.2).

3.1 Setup

For our analysis, we used an i386 FreeBSD 13.0 kernel[1]. We instrumented 4 virtual-filesystem data types that are accompanied by comprehensive locking documentation, namely `struct vnode`, `struct bufobj`, `struct mount`, and `struct buf`. We recorded lock operations for the following 8 different lock types: *hardirq*, *lockmgr*, *rm*, *rw*, *sleepable rm*, *sleep mutex*, *spin mutex*, and *sx*.

We chose the *fs*[2] test suite from the *Linux Test Project* [4] as the workload for triggering locking primitives. It took 26.43 hours (real runtime in a virtual machine: 20.22 minutes) to run the benchmark, and traced its execution via FAIL* [9]. This run produced 1.52 billion events. The complete post processing took 14.42 hours. All experiments took place on a Intel® Xeon® E5-1620 processor.

3.2 Locking Rule Checking

Analogous to results from our previous work on Linux [5], we validated the locking documentation for the aforementioned 4 data types. This documentation is located in the FreeBSD *src* tree in `sys/sys/{vnode.h,buf.h,mount.h}` line 79 ff., line 94 ff., and line 195 ff., respectively. Note that `struct bufobj` (`sys/sys/bufobj.h` line 94 ff.) is embedded in `struct vnode` and not explicitly listed in Tab. 1, which gives a summary of our results: For each data type, the number of documented locking rules (column *#R*) is given. A locking rule refers to a tuple of data type, member, and access type[3]. The following columns denote the number of rules that have at least one observation (column *#Ob*), and those not having been observed at all (column *#No*). We categorize the observed members into rules that have a relative support of 100 % (column ✓), a relative support below 100 % but above 0 % (column ~), or have no observation at all (column ✗). We were able to confirm the locking documentation for 72.6 % (`vnode`), 71.43 % (`buf`), and 74.19 % (`mount`) of the rules. Overall, the members are accessed consistently with their documentation in 72.41 % of the cases.

The locking rule for only *one* tuple of data type, element, and access type was not found in our data set (column ✗): The trace did not contain any read accesses to element `b_error` of `struct buf` – this element seems only to be read in conditions not triggered by the workload we chose.

A closer inspection of the rules with $0.9 \leq s_r \leq 1$ revealed two real locking bugs: For both tuples, writing `b_vflags` of `struct buf` and reading `b_blkno` of `struct buf`, the documented locking rules had only relative supports of 97.3 %

[1]Based on Git commit *2134e85bc1b02389b462c2c9995af98ca0bf7213*.
[2]We used Git tag *20190115* from the LTP repository.
[3]Since we differentiate between read and write accesses, dividing the number of rules by two gives the number of documented members.

Data Type	#R	#No	#Ob	✓(%)	~ (%)	✗(%)
vnode	82	9	73	72.60	27.40	0.00
mount	38	7	31	74.19	25.81	0.00
buf	80	10	70	71.43	27.14	1.43

Table 1: 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). One locking rule refers to the locking order for a tuple of data type, element, and access type (read/write). The last three columns denote the portion of correct ($s_r = 1$), ambivalent ($0 < s_r < 1$) and incorrect ($s_r = 0$) rules.

and 96.2 %, respectively. Zooming in on the numerous memory accesses violating the documented locking rules, we could identify one responsible code location for each of the two cases. Confronted with our findings, the FreeBSD developers confirmed and fixed[4] both bugs. Applying those fixes and re-running our experiment would push the percentage of approved rules to 73.6 %. However, these tuples are 2 out of 11 that have a relative support between 90 % and 100 %; the remaining 9 tuples are false positives. The causes overlap with those for Linux (see Sec. 1): Some accesses are domain-specifically known from a whole-system perspective not to be racy[5], and some NULL-pointer checks are not guarded. However, we identified one new cause for false positives that very likely also applies to Linux and other large-scale software systems: According to the documentation[6], `struct inode` attributes may be changed after *either* acquiring the vnode lock exclusively, *or* after acquiring the vnode lock in *shared mode* and taking the vnode interlock – both variants can be used equally. LockDoc is currently not equipped for this kind of locking-pattern alternative, which we intend to remedy in future work.

Taking samples of rules with $s_r < 0.9$ revealed another interesting case, attributable to the same shortcoming in LockDoc: For `b_qindex` and `b_subqueue` of `struct buf`, the relative support is split across two locks. The documentation states: *"Protected by the buf queue lock"*[7]. However, there are *two* buf queue locks in the addressed data type `struct bufdomain`[8]: `bd_dirtyq.bq_lock` and `bd_subq.bq_lock`. Depending on the buf queue in use – `bd_subq` or `bd_dirtyq`, – the respective lock is used. It is therefore not possible for either rule to reach 100 % relative support.

3.3 Discussion

Our results indicate that FreeBSD indeed presents itself more "cleanly" with respect to locking documentation and implementation. Together with an overall much lower complexity than Linux, and consequently results that were much easier to manually inspect than in earlier work [5], this study on FreeBSD allowed us to uncover a fundamental limitation of LockDoc, which we intend to address in future work.

Sec. 3.2 perfectly demonstrated LockDoc's ability to pinpoint locking-rule violations: For both tuples, LockDoc could provide the exact source-code location where the access happened, together with a stack trace leading to that access, and the locks that were actually held (and at what exact code locations they were acquired). This information was sufficient to fix the unguarded write to `b_vflags`. For the unprotected read of `b_blkno`, we were additionally able to tell the developers where exactly the lock was released that should have guarded the access. With that information, the developers were able to also fix the unguarded read of `b_blkno`.

Finding those two bugs was based on an approach different from LockDoc's original workflow: Normally, LockDoc uses its own *derived* locking rules to find accesses violating them. Here, we used the *documented* locking rules as the ground truth to scan our data set for counterexamples. For systems with a sound ground truth, this approach seems promising for future work.

4 CONCLUSIONS

In this paper, we applied LockDoc to FreeBSD to investigate whether its superior locking documentation and comparably lower complexity than Linux provides new insights on our approach. This resulted in a higher rate of approved documented locking rules: 72.4 % of all evaluated data-structure elements are accessed according to the documentation. Our analysis also showed that, at least for systems with a soundly documented locking "ground truth", looking for bugs using the *documented* instead of the *derived* locking rules seems beneficial: This approach revealed two locking bugs in the latest FreeBSD release, which have since been fixed by the developers. In the process, we also uncovered a limitation in LockDoc's internal lock model, which we intend to remedy in future work.

[4] https://github.com/freebsd/freebsd-src/commit/e3d67595
https://github.com/freebsd/freebsd-src/commit/5cc82c56

[5] https://lists.freebsd.org/archives/freebsd-fs/2021-August/000371.html

[6] cf. `sys/ufs/ufs/inode.h`, line 74 ff.

[7] cf. `sys/sys/buf.h`, line 96.

[8] cf. `sys/kern/vfs_bio.c`, line 117.

REFERENCES

[1] John H. Baldwin. 2002. Locking in the Multithreaded FreeBSD Kernel. In *Proceedings of the BSDCon '02 Conference on File and Storage Technologies*. Cathedral Hill Hotel, San Francisco, California, USA, 27–36. https://www.usenix.org/legacy/events/bsdcon/full_papers/baldwin/baldwin_html/

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

[3] Jonathan Corbet. 2006. The kernel lock validator. https://lwn.net/Articles/185666/. Accessed: 2020-05-28.

[4] Cyril Hrubis et al. [n. d.]. Linux Test Project. https://github.com/linux-test-project/ltp. Accessed: 2020-08-20.

[5] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. 2019. LockDoc: Trace-Based Analysis of Locking in the Linux Kernel. In *Proceedings of the 14th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '19)*. ACM Press, New York, NY, USA. https://doi.org/10.1145/3302424.3303948

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

[7] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. 2014. *The Design and Implementation of the FreeBSD Operating System* (2. ed. ed.). Addison-Wesley, Upper Saddle River, NJ.

[8] Byungchul Park. 2016. Enhancing lockdep with crossrelease. https://lwn.net/Articles/709849/. Accessed: 2020-05-28.

[9] 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