Improving Linux-Kernel Tests for LockDoc with Feedback-driven Fuzzing

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

Robin Thunig
TU Dortmund
Department of
Computer Science 12
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 from a dynamic execution trace recorded while the system is under a benchmark load. These locking rules can e.g. be used to locate synchronization bugs. For high rule precision and thorough bug finding, the approach heavily depends on the choice of benchmarks: They must trigger the execution of as much code as possible in the kernel subsystem relevant for the targeted data structures. However, existing test suites such as those provided by the *Linux Test Project* (LTP) only achieve – in the case of LTP – about 35 percent basic-block coverage for the VFS subsystem, which is the relevant subsystem when extracting locking rules for filesystem-related data structures.

In this article, we discuss how to complement the LTP suites to improve the code coverage for our LockDoc scenario. We repurpose *syzkaller* – a coverage-guided fuzzer with the goal to validate the robustness of kernel APIs – to 1) *not* aim for kernel crashes, and to 2) maximize code coverage for a specific kernel subsystem. Thereby, we generate new benchmark programs that can be run in addition to the LTP, and increase VFS basic-block coverage by 26.1 percent.

KEYWORDS

Test Generation, Kernel Test Coverage, Basic-Block Coverage, syzkaller, Linux Test Project, kcov

1 INTRODUCTION

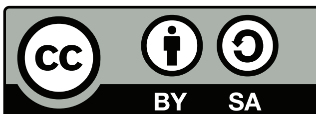
Over a period of more than a decade, the Linux kernel underwent a transformation from Linux 2.0's coarse-grained and sturdy Big Kernel Lock [1] to more and more fine-grained synchronization on the granularity of kernel subsystems and even single data structures [1, 11, 14]. While reducing lock contention and scaling better to multi- and many-core platforms, fine-grained locking is error-prone and has led to numerous synchronization bugs in the past. This situation is exacerbated by incomplete, inconsistent and in parts faulty locking documentation.

Our LockDoc approach [10] addresses these issues by tracing locking patterns and data-structure accesses in a running Linux kernel under load, and deriving locking rules – i.e., which locks in which particular order must be taken to access a specific data-structure element – from this trace. The derived locking rules can consequently be used to validate or generate documentation, and to locate synchronization bugs. However, locking-rule quality and the capability to find bugs also in remote parts of the kernel heavily depends on how the system is put under load – i.e. on the choice of benchmark programs that trigger the execution of kernel code by invoking system calls. Focusing on the Linux kernel's *Virtual File System* (VFS) subsystem in our LockDoc study [10], we relied on a filesystem-specific subset of the *Linux Test Project* (LTP) [4, 7] benchmark suites to provoke lock operations for and accesses to VFS data structures.

However, when we actually measure basic-block test coverage in Linux with kcov [18] while running all LTP suites that even remotely seem to be related to VFS, our results indicate that only about 35 percent of the kernel's basic blocks associated with the VFS subsystem are actually executed. Although it is not reasonable to expect 100 percent coverage – e.g., depending on the kernel configuration there may be several filesystems compiled in that are not associated with an actual mount point on the test system – there is certainly room for improvement.

In this article, we propose an approach to increase kernel-code coverage for LockDoc: We repurpose *syzkaller* [17], a coverage-guided fuzzer with the goal to validate the robustness of kernel APIs, to not aim for kernel crashes but only for basic-block coverage in a particular kernel subsystem. The thereby generated new benchmark programs can be run in addition to the LTP suites, and increase VFS basic-block coverage by 26.1 percent from 34.7 percent (LTP only) to 43.8 percent (LTP + generated benchmarks).

To summarize, the contributions of this article are:



Except as otherwise noted, this paper is licenced under the Creative Commons Attribution-Share Alike 4.0 International Licence.

FGBS '20, September 24–25, 2020, Aaachen, Germany
© 2020 Copyright held by the authors.
https://doi.org/10.18420/fgbs2020h-01

- A quantitative analysis of the LTP's capability to trigger code execution in Linux's VFS subsystem (Sec. 3).
- An approach to repurpose a coverage-guided kernel fuzzer to generate benchmark programs that target a particular kernel subsystem and do not crash the system (Sec. 3).
- An evaluation demonstrating that the combination of LTP and generated benchmark programs covers significantly more VFS basic blocks than LTP alone (Sec. 4).

Sec. 2 discusses related work, and Sec. 5 concludes the paper.

2 RELATED WORK

Linux-kernel test coverage has been a research topic since Linux's very early days. Iyer [5] analyzes the LTP's coverage of Linux 2.4 kernel code with GCOV, and reports that parts of the `fs/` kernel source-code subtree have line-coverage values between 0.0 (especially for most of the actual filesystem implementations) and 29.5 percent (for the generic, filesystem-agnostic part). Larson [8] distills detailed HTML reports from GCOV results obtained from Linux 2.5, reports that the LTP covers about 90 percent of all kernel basic blocks that are executed by a much larger benchmark corpus, and concludes that coverage results should drive further LTP development. Yoshioka [19] describes a Linux regression test framework named *crackerjack* and the accompanying branch-coverage test tool *btrax*, and demonstrates coverage advantages over LTP. The *Lachesis* approach [2] by Claudi and Dragoni provides a test suite focusing on real-time extensions for Linux.

While OS-kernel fuzzing approaches date back to at least 1991 with Le's *tsys* [9] or to 2006 with *trinity* by Jones et al. [6], modern kernel-fuzzing approaches like Vyukov's *syzkaller* [17] are coverage-guided. For example, Nossum and Casasnovas [13] port the prominent user-mode fuzzer AFL to the kernel and uncover filesystem bugs. *DIFUZE* by Corina et al. [3] fuzzes kernel drivers to detect bugs, and is aided by static analysis that determines the necessary input structure. Schumilo et al.'s *kernel AFL* (kAFL) [15] is a target-OS agnostic fuzzing approach based on a hypervisor and hardware support in the form of Intel's *processor trace* feature. Shi et al. [16] demonstrate the practical application of existing kernel-fuzzing tools to several Linux versions, and provide an overview of several other kernel-fuzzing approaches.

To the best of our knowledge, the approach described in this paper is the first to repurpose coverage-guided kernel fuzzing to generate benchmarks that complement existing test suites.

3 APPROACH

The quality of LockDoc's results heavily depends on the number of observed lock operations and data-structure accesses [10]. Due to static-analysis limitations (e.g. pointer aliasing), it is generally infeasible to statically identify all code locations that make such accesses. For similar reasons it is generally infeasible to statically determine all contexts from which these code locations are called, and consequently, which locks are possibly held when the data-structure accesses are made. LockDoc therefore resorts to dynamic analysis, i.e. the observation of the running kernel under a benchmark load. As LockDoc focuses on the VFS subsystem to determine locking rules e.g. for the *inode* data structure, the problem at hand therefore is to find and run benchmarks that maximize kernel-code coverage for this particular subsystem.

The literature proposes several metrics for code coverage, e.g., path, branch or line coverage [12, 20]. For this paper, we chose basic-block coverage over line coverage, because it better captures the actual fraction of the code that is being executed. One line of code, for example, can be mapped to several basic blocks. Hence, having one particular line covered does not necessarily mean all basic blocks are covered. Inversely, covering all basic blocks means all lines of code that have been compiled are covered.

For the Linux kernel, there is already a good starting point for executing a large fraction of kernel code: the *Linux Test Project* (LTP) [4, 7], of which we already used a VFS-related subset for earlier work on LockDoc [10]. LTP's aim is to *"validate the reliability, robustness, and stability of Linux"* [4]. It consists of several individual tests that are grouped into test suites. Each of these suites targets a particular subsystem or a particular kernel functionality such as the IPC mechanism or the VFS subsystem. The scope of the tests ranges from stress testing to regression testing. Since those tests are manually composed, only a limited subset of each system call's parameter space can be covered, resulting in a limited amount of code coverage.

The VFS-related LTP test suites generate a basic-block coverage of 34.7 percent for the VFS subsystem. We determined the coverage for each test individually of the following test suites[1]: *dio*, *fcntl-locktests*, *filecaps*, *fs*, *fs_ext4*, *fs_perms_simple*, *fsx*, *io*, and *syscalls*. As the *fs_readonly* suite only runs a subset of *fs* on a read-only mounted filesystem, we skipped it. Tab. 1 shows the number of VFS basic blocks covered by each suite.

The numbers indicate that there is still room for coverage improvement – and, hence, higher LockDoc precision and bug-finding effectiveness. It turns out that *fuzzing* (see Sec. 2) in its coverage-guided feedback variant already has

[1] We use git tag *20190115* of the LTP repository.

Test Suite	# Tests	Covered VFS BBs	(%)
dio	30	8312	11.0%
fcntl-locktests	1	2420	3.2%
filecaps	1	2518	3.3%
fs	65	17 495	23.2%
fs_ext4	4	13 081	17.3%
fs_perms_simple	18	5081	6.7%
fsx	1	6572	8.7%
io	2	6817	9.0%
syscalls	1181	24 217	32.1%
Total	1303	26 229	34.7%

Table 1: Covered basic blocks for each LTP test suite in the VFS subsystem. In total, our Linux 4.10 kernel build consists of 342,732 basic blocks, and the VFS subsystem of 75,531 basic blocks, respectively.

```
int main(void)
{
  syscall(__NR_mmap, 0x1ffff000, 0x1000, 0, 0x32,
       -1, 0);
  syscall(__NR_mmap, 0x20000000, 0x1000000, 7, 0
     x32, -1, 0);
  syscall(__NR_mmap, 0x21000000, 0x1000, 0, 0x32,
       -1, 0);

  *(uint32_t*)0x20002480 = 0x20000340;
  memcpy((void*)0x20000340, "\x12", 1);
  *(uint32_t*)0x20002484 = 1;
  *(uint32_t*)0x20002488 = 0;
  syz_read_part_table(0, 1, 0x20002480);
  return 0;
}
```

Listing 1: An excerpt of a program generated by fuzzing the Linux Kernel using Syzkaller [17].

coverage maximization as one of its main goals. A coverage-guided kernel fuzzer that recently came to fame is Vyukov's *syzkaller* [17], which fuzzes the Linux kernel by randomly generating user programs that use the system-call interface. For each generated program (see an example in Listing 1), syzkaller determines the resulting basic-block coverage. Only programs that cover at least one new basic block are stored in the database. syzkaller's objective is to trigger kernel bugs, and to minimize the program that triggered the bug.

We modified syzkaller to 1) ignore programs triggering a bug, and to 2) only store programs that increase the coverage in the VFS subsystem. We furthermore disabled a set of system calls that are not related to the VFS to improve fuzzing speed. The resulting programs are intended to cover more code, and thus cover more memory accesses, which in turn can be used by LockDoc. Note that these generated benchmark programs cannot (directly) be used as regression tests for the kernel, as they do not make any explicit output that can be used to determine test success.

4 EVALUATION

In this section, we first present our evaluation setup in Sec. 4.1, and then show our results in Sec. 4.2.

4.1 Setup

We conduct our experiments on an x86 64-bit Linux Kernel 4.10. The kernel is built without module support, and uses a minimal kernel configuration: Network support is active as well as the essential drivers for the root filesystem and for running in a paravirtualized *QEMU*-based virtual machine. To record the executed basic blocks, we enabled a kernel feature called kcov [18], which was initially introduced by syzkaller [17].

The syzkaller modifications[2] mentioned in Sec. 3 include disabling 191 system calls that are not related to VFS, e.g. those for process control, memory operations, or network operations. Based on an example given by syzkaller, we implemented a library that records the executed kernel basic blocks during execution of an arbitrary program in 307 lines of *C++* code. Since the library hooks into the program under test via the *LD_PRELOAD* mechanism, we collect covered basic blocks for a complete process hierarchy. We used this library to gather the results presented in Sec. 3.

Whether one basic block belongs to the VFS subsystem or not is determined using *addr2line*[3] on the Linux-kernel ELF image: It converts an address to one or more kernel source-file names. Due to function inlining, it may return more than one source file for a single address. If a source file matches the following regular expression, a basic block is considered to belong to the VFS subsystem: `/fs/|/mm/|fs\.h|mm\.h`. We also include the `mm` directory and header files containing `mm.h`, because the file-I/O code is located there such as `mm/readahead.c` or `mm/page-writeback.c`.

4.2 Results

During its 65-hour run, syzkaller generated 2278 programs that created a basic-block coverage of 10.0 percent *for the*

[2] Our modifications are based on git commit *056be1b9c8d0c6942412dea4a4a104978a0a9311*.

[3] https://sourceware.org/binutils/docs/binutils/addr2line.html

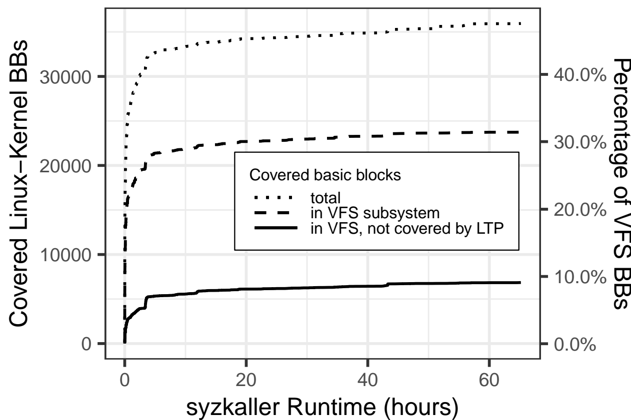


Figure 1: Development of the Linux-kernel basic-block coverage for the programs generated by syzkaller. The dotted line shows the number of covered basic blocks out of all 342,732 kernel BBs; the dashed line shows the fraction of the 75,531 basic blocks of the VFS subsystem. The solid line shows the number of VFS basic blocks covered by syzkaller-generated programs that are not already covered by LTP.

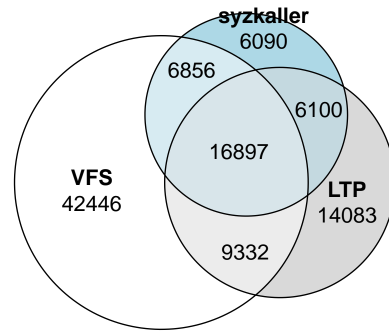


Figure 2: Set relations between VFS-subsystem basic blocks, and basic blocks covered respectively by syzkaller's generated programs and by LTP's test suites.

whole kernel and 31.4 percent *for the VFS subsystem*. Moreover, it covers 9.1 percent of VFS basic blocks that are *not already covered by LTP*. Fig. 1 shows the development of these three basic-block coverage numbers during syzkaller's run: The basic-block coverage increases quickly after the start, and slowly levels off afterwards. The Y axis on the left-hand side of the plot shows the absolute amount of basic blocks covered, the Y axis on the right side the percentage of VFS basic blocks.

The relation between basic blocks belonging to the VFS subsystem, and those covered by syzkaller's generated programs and by LTP's test suites, is displayed in the area-equivalent Venn diagram in Fig. 2. The absolute numbers shown for each area intersection are the absolute number of basic blocks shared among all intersecting basic-block sets.

To summarize, our results indicate that combining LTP and syzkaller's programs to one workload can significantly improve the overall code coverage for the VFS subsystem by 9.1 percentage points from 34.7 to 43.8 percent.

5 CONCLUSIONS

In this article, we showed that using LTP as the only benchmark source for LockDoc yields limited kernel-code coverage, as it only covers 35 percent of basic blocks for the VFS subsystem. We repurposed syzkaller to generate programs that complement LTP to achieve better coverage, with the future-work goal to improve LockDoc's precision and bug-finding effectiveness. We were able to increase the VFS basic-block coverage by 26.1 percent by combining LTP and syzkaller.

As our next steps, we plan to optimize the resulting benchmark suite in terms of runtime with near-zero coverage loss. We want to discard tests or programs that do not add new coverage or incur too much runtime.

REFERENCES

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

[2] Andrea Claudi and Aldo Franco Dragoni. 2011. Testing Linux-based real-time systems: Lachesis. In *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*.

[3] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM Press, 2123–2138.

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

[5] Manoj Iyer. 2002. Analysis of Linux Test Project's kernel code coverage. http://kernel.poly.ro/2.4/kernel%20docs/kernel_coverage.pdf.

[6] David Jones et al. 2006. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity. Accessed: 2020-08-20.

[7] Paul Larson. 2002. Testing Linux with the Linux Test Project. In *Proceedings of the Ottawa Linux Symposium*. 265–273.

[8] Paul Larson, Nigel Hinds, Rajan Ravindran, and Hubertus Franke. 2003. Improving the Linux Test Project with kernel code coverage analysis. In *Proceedings of the Ottawa Linux Symposium*. 275–289.

[9] Tin Le. 1991. tsys. https://groups.google.com/g/alt.sources/c/V_B37EtnWKQ/m/NztsljVYV84J. Accessed: 2020-08-20.

[10] 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

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

[12] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3rd ed.). Wiley Publishing.

[13] Vegard Nossum and Quentin Casasnovas. 2016. Filesystem fuzzing with American Fuzzy Lop. In *Proceedings of the Linux Storage and Filesystems Conference (VAULT)*. USENIX Association.

[14] 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.

[15] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, Vancouver, BC, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[16] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. 2019. Industry Practice of Coverage-Guided Enterprise Linux Kernel Fuzzing. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM Press, New York, NY, USA, 986–995. https://doi.org/10.1145/3338906.3340460

[17] Dmitry Vyukov. 2015. Syzkaller: An unsupervised, coverage-guided kernel fuzzer. https://github.com/google/syzkaller. Accessed: 2020-08-20.

[18] Dmitry Vyukov. 2016. kcov: code coverage for fuzzing. https://www.kernel.org/doc/html/latest/dev-tools/kcov.html. Accessed: 2020-08-20.

[19] Hiro Yoshioka. 2007. Regression Test Framework and Kernel Execution Coverage. In *Proceedings of the Linux Symposium*. 285–296.

[20] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.* 29, 4 (Dec. 1997), 366–427. https://doi.org/10.1145/267580.267590