Linux Kernel Audit Subsystem Code Coverage Analysis

Ricardo Robaina
Core Kernel Engineering. Red Hat
rrobaina@redhat.com

Rafael Aquini
Core Kernel Engineering. Red Hat
raquini@redhat.com

***Abstract*—Linux-based Operating Systems are widely used in different domains of today's industry. One of its key components, the Linux kernel, is a complex project in constant evolution that requires consistent testing diligence to ensure the expected behavior as the code base evolves. The audit subsystem is responsible for providing a secure log framework for the kernel, and its correct operation is paramount for meeting security standards. This paper proposes a new coverage metric, named PCII. It presents a code coverage analysis of the audit subsystem, exploring its overall coverage, individual function coverage, and the relationship between individual test cases and source code files. The results reveal that about 61% of the subsystem code is currently covered by its test suite. Moreover, it demonstrates that 51 functions are completely uncovered, which represents the majority of the uncovered parts of the audit subsystem. Lastly, the relationship between individual test cases and audit-related source code files was mapped and can be used to improve test infrastructure.**

1. Introduction

The GNU/Linux Operating System (OS) is widely used in innumerable domains of today's industry, from banking and telecommunication applications to supercomputers and embedded systems. In recent years, the demand for its use in safety-related systems is growing [3]. The Linux kernel is considered one of the main components of these systems, where it is responsible for managing hardware devices and offering essential services to the upper layers of the computing system [13]. The kernel is considered a large and complex software due to its non-deterministic nature, the breadth of supported hardware, and the number of features it offers [2] [1].

The fast pace at which kernel development progresses and, consequently, the increasing complexity of its subsystems require consistent testing diligence over time to ensure the expected behavior as the code base evolves. A commonly used strategy in this context is regression testing [13], which consists of validating that a code change does not affect existing functionalities or introduce malfunctions in the overall system. However, the success of a regression test process is coupled to the quality of the test cases within the test suite used. A robust set of tests, designed to cover most of the target code, helps discover issues during the testing process and reduces the odds of hitting bugs related to untested code paths upon deployment.

Code coverage analysis provides a method to evaluate the quality of a test suite, as it helps identify parts of the target code that are not covered by its tests, thus exposing untested areas of the system [5]. The data provided by such an analysis can, among other things, guide the test case maintenance process, aiming to improve it by filling gaps in the test suite. Two frequently used techniques are statement and branch coverage analysis. The first breaks the target code into basic blocks, called statements, and counts how many times each statement was exercised during a test execution. Branch coverage, on the other hand, focuses on the frequency of the possible conditional code paths the code execution can take during its execution [5]. Despite the known limitations of these two techniques, pursuing a 100% coverage rate is a good testing practice and a real requirement of safety standards, such as IEC 61508 [2] [11].

The audit subsystem [6] provides a secure log framework for the Linux kernel, allowing security-relevant events to be recorded. Thus, its correct operation is paramount to meeting the security standards of production systems. The audit-testsuite project [7] hosts the official regression test suite for the audit subsystem, is widely used by quality engineering organizations in the industry, and is the recommended test suite for upstream audit-related contributions. However, in recent years, the audit-testsuite project has not been evolving at the same pace as the Linux kernel audit subsystem itself. For example, of the 20 issues currently opened in the project, half are requests for enhancements (RFCs) filed more than ten years ago without any progress.

This paper presents a statement code coverage analysis of the Linux kernel audit subsystem provided by the audit-testsuite project. As part of the research process, it was sought to answer the following three questions:

1) What is the current coverage provided by the audit test suite to the audit subsystem?
2) What are the main coverage gaps of the test suite, if any?
3) What is the relationship between individual test cases and audit-related source code files?

This paper is organized as follows. Section 2 presents a brief overview of the test coverage tool used in this study. The next Section details the methodology followed

through the data-collecting process. The results obtained are discussed in Section 4. Finally, the last part of this paper concludes our findings and details planned future work.

2. GCOV Code Coverage Tool Overview

GCOV is a tool that allows developers to collect code coverage data from programs compiled with the GNU Compiler (GCC) [4]. This coverage tool achieves its goals by instrumenting the target code and measuring the extent to which it has been exercised after a test execution has occurred. It provides several coverage statistics and highlights untested regions of code. Its use provides excellent value, considering that the data obtained from it is helpful to evaluate the effectiveness of a test suite or benchmark and can be used in the process of improving the quality of the target software. GCOV offers both statement and branch coverage data gathering.

The Linux kernel supports GCOV profiling [12], which enables the collection of code coverage-related data from a running kernel. To do so, the kernel needs to be compiled with the following configurations enabled:

- *CONFIG_DEBUG_FS*
- *CONFIG_GCOV_KERNEL*
- *CONFIG_GCOV_FORMAT_AUTODETECT*
- *CONFIG_GCOV_PROFILE_ALL*

In this article [5], Paul Larson and contributors detail the GCOV internals and the challenges of enabling it in the Linux kernel. Although outside the scope of this study, being aware of the following features of a kernel compiled with GCOV support is valuable to understanding the present study:

1) The internal files GCOV manipulates to generate coverage metrics (*.gcda* and *.gcno*) are updated on the fly and exposed via the pseudo filesystem *sysfs* at */sys/kernel/debug/gcov*.
2) A global reset interface is available at */sys/kernel/debug/gcov/reset*, which allows the GCOV counters to be reset, or zeroed out, from a userspace call. Once a user triggers the reset, all coverage data accumulated since the last reset is completely erased.

Among other capabilities, the GCOV utility offers options to collect coverage data on a per-file or per-function basis. Both modes return the percentage of lines executed and the number of executable lines in the source code file or the function, respectively. Regardless of the mode chosen, the tool will create a *.gcov*-extension file as a result. This file contains the same source code as the targeted file, annotated with coverage data.

The Code Snippet 1 is an example of a *.gcov* file generated by the GCOV utility. All the lines annotated with a Natural Number (N) indicate that the line is executable and it has been exercised *N* times since the last data reset; The lines annotated with -, on the other hand, are non-executable, thus were not instrumented by GCOV and are not considered in the coverage metrics. Lastly, *####* mark executable lines that have not been executed yet. Thus, those lines indicate the gaps in terms of coverage provided by a given test.

```
    1:   18:main (void)
    -:   19:{
  ...
    1:   32:  if (total != 45)
#####:   33:    printf ("Failure\n");
    -:   34:  else
    1:   35:    printf ("Success\n");
    1:   36:  return 0;
    -:   37:}
```

Code Snippet 1: Example of a .gcov extension file that illustrates the annotations made by GCOV on the target source.

In addition to the GCOV output, only executable lines are considered throughout this paper. Thus, *number of lines* is interchangeable with *number of executable lines*. In this sense, the total of executable lines of a source code will be referred to as *source size (ss)* and of a function as *function size (fs)*. The percentage of coverage of a file will be referred to as *file coverage (sc)*, while the total coverage of a given function will be referred to as *function coverage (fc)*.

3. Research Design and Methods

3.1. Software Environment

The results explored in this study were collected from an x86_64 architecture KVM-based virtual machine running the Fedora 41 operating system. The specific versions of key components in the environment are:

- **kernel version:** 6.13.9
- **auditd version:** 4.0.3
- **audit-testsuite version:** HEAD at commit 6f8c12d
- **GCOV version:** 14.2.1

The kernel was compiled using the standard Fedora configuration file for the chosen CPU architecture (*kernel-x86_64-fedora.config*), with the additional enablement of *CONFIG_GCOV_KERNEL* and *CONFIG_GCOV_PROFILE_ALL* kernel configs.

3.2. Scope of the Study

The audit subsystem intersects with many other subsystems of the Linux kernel; therefore, its code is spread across different kernel source code files. However, the scope of the study was limited only to the files that implement the core auditing functionality. The source files considered are the following:

- *audit.c*
- *auditfilter.c*
- *auditsc.c*
- *audit_fsnotify.c*
- *audit_tree.c*
- *audit_watch.c*

3.3. Data Collection Method

Once the test machine was set up, the following process was followed to collect the data:

1) Clean GCOV data;
2) Run the audit-testsuite;
3) Collect coverage data.

The steps described above were automated using bash scripts to avoid error-prone manual tasks and ensure a consistent, reproducible data-gathering process, thereby reducing variance caused by user interaction with the OS. The first script, called *get-cvg-file.sh*, collects coverage data of selected source files and returns a list of filenames and their respective overall coverage, referred to as source coverage (*sc*). The script *get-cvg-func.sh*, on the other hand, collects function-related coverage data; its output contains both the coverage (*fc*) and size (*fs*) of each function from a source file, in addition, it also returns the source file size (*ss*).

Lastly, the data collected through the steps mentioned above were processed using the statistical language R [10], and the results obtained are presented in the next section. Both the scripts aforementioned and the raw data collected by them, as well as the R scripts used to process that data, are available in this public repository [8].

4. Results And Discussion

4.1. Audit Subsystem Overall Coverage

By following the steps exposed in the methodology, the code coverage data related to all the source code files studied was collected. To accomplish this, after cleaning the gcov counters, a complete execution of the audit-testsuite was triggered, and then per-file coverage data was collected. The initial analysis of the code coverage data showed that, on average, 61.08% of the audit subsystem code is covered by its test suite.

Figure 1 presents a bar graph that reveals the coverage provided by the full test suite on each source file. A closer look reveals that the files with higher coverage rates are *audit.c*, *auditfilter.c* and *audit_watch.c*, ranging between 67% and 72% coverage. The files *audit_fsnotify.c* and *auditsc.c*, on the other hand, have coverage rates close to 55%. Lastly, *audit_tree.c* is the file with the lowest coverage rate in the sample, only 46% of its code is covered by the test suite.

Looking at it from a different angle, however, these results indicate that approximately 38.92% of the audit subsystem code is not covered by the test suite. Thus, it is fair to state that, although the test suite covers a large portion of the subsystem code, there is still significant room for improvement.

The results presented above expose that there are, in fact, untested code areas within the audit subsystem. However, it does not reveal the main gaps in coverage. To find this answer, a per-function code coverage analysis was performed, and it is described in the following section.

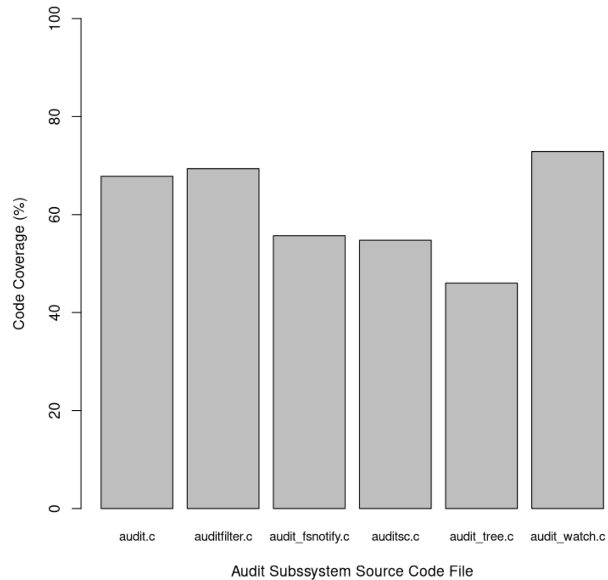

Figure 1. Bar Chart of Percentual Audit Subsystem Code Coverage Provided by the Test Suite

4.2. Audit Subsystem Function Coverage

To complement the results discussed in the previous section and to identify gaps in test suite coverage, an analysis of the functions' coverage was performed. In the same fashion as the overall coverage analysis, after cleaning the coverage counters, the full test suite was executed. However, this time, per-function data was collected instead.

The histogram plotted in Figure 2 illustrates the obtained results. The bars in the histogram represent the frequency distribution of all 295 functions presented in the audit files by coverage ranges. With the data organizing manner, it is possible to highlight the following:

- 171 functions, representing 57.96% of the total, have high coverage rates (higher than 80% coverage). Of those, 120 have full coverage.
- 51 functions are covered between 50% and 80% by the test suite, which represents 17.28% of the function set studied.
- In the neighborhood of 24% of functions have coverage lower than 50%. Notice that the great bulk of these functions are zero-coverage functions; out of the 73 functions in this range, 58 have a zero coverage rate.

The results obtained in this analysis reveal that the majority of functions have high coverage rates, and that nearly one-fifth of the subsystem is not covered by any test case in the test suite. This high number of functions with high coverage indicates that the existing tests in the suite already provide good coverage for the functions they exercise, and there is no need to update the majority of the

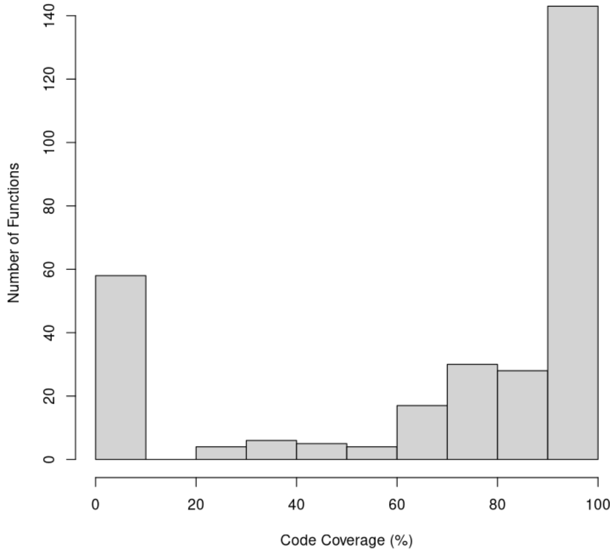

Figure 2. Histogram of Audit Subsystem Functions Distributed by Coverage Rates

existing tests. However, it also highlights the need for new test creation to address functions with lower coverage rates, particularly those with zero coverage.

Based on the facts mentioned earlier, it would be tempting to assume that the main gap in coverage provided by the audit-test suite originates from the lack of coverage in several subsystem functions. However, the coverage data does not provide enough insight into how the lack of coverage of one function affects the overall coverage of the source file in which it is located. Thus, this metric alone is insufficient to determine in detail where the gaps lie. The following section addresses this problem.

4.3. Potential Coverage Increase Index (PCII)

The results observed indicate that the test suite does not cover a considerable number of functions. Although creating test cases for those functions is key to ensuring the correct functioning of the subsystem, it might not necessarily result in a substantial increase in its overall coverage. This occurs because the relationship between the size of a function and its source file, in addition to the current coverage rate of the function, must be considered to compute the proportional impact of a function on the overall file coverage rate. Based on that, in cases where the function size is small compared to the size of the source code, creating a full coverage test case for that function would not result in a substantial increase in the overall coverage of a source file. For this reason, depending on the goal, prioritizing test creation solely based on coverage metrics may not be the most optimal approach.

To be able to calculate how much coverage increase would be achieved once a given function gets fully covered, the Potential Coverage Increase Index (PCII) was created, and it is defined by the equation below:

$$PCII_{ss,fs,fc} = (100 - fc) \times \frac{ss}{fs}$$

Where: ss is the source size, fs is the function size and fc is the function coverage.

This equation takes into consideration both the proportional size of the function and its current coverage rate to calculate the index. The result of this equation for a particular function indicates how much of the lack of coverage of the source file is related to that function. In other words, it indicates how much coverage would be gained as soon as a full coverage test case targeting that function is developed. Given the nature of the equation, functions with 100% coverage will result in a PCII score of zero, and the PCII summation of all functions within a source file is complementary to its overall coverage.

Table 1 shows a sample of *audit.c* functions with their respective coverage rates and PCII values, sorted by PCII score. By observing this example, it becomes clear that the highest potential for increasing the coverage of the particular source file lies in the function *audit_receive_message*. The PCII score of this function indicates that once this function is fully covered, the overall coverage of *audit.c* will increase around 6.2%. Notice that the test suite already covers 67% of it, so the 6.2% potential coverage increase, in this case, is solely related to the 33% uncovered statements of the function. On the opposite side of this sample, the zero-coverage function *audit_log_path_denied* has the potential of increasing only 0.8% of its source file.

TABLE 1. SAMPLE OF AUDIT.C FUNCTIONS SORTED BY PCII SCORE

Function	Coverage (%)	PCII Score
audit_receive_msg	64	6.28492
audit_init	0	2.30447
audit_set_feature	0	2.09497
audit_set_loginuid	41.18	1.39656
audit_log_n_string	67.39	1.04753
kauditd_thread	62.5	1.04749
audit_log_task_info	75	0.907821
audit_enable	0	0.907821
audit_log_path_denied	0	0.837989

The PCII has the potential to be utilized as an additional metric to guide the decision-making process when prioritizing the maintenance of a test suite, as it reveals in detail where coverage gaps are located. The following section presents the results of the PCII-based analysis of the audit-test suite project.

4.4. Detailing the Audit Subsystem Coverage Gaps Through PCII Analysis

To get more details about how the coverage gap is distributed in each source file studied, the PCII metric was

used. First, all functions were divided into subgroups based on coverage ranges, and then the total PCII score for each subgroup was calculated. The results were organized into a stacked bar chart presented in Figure 3.

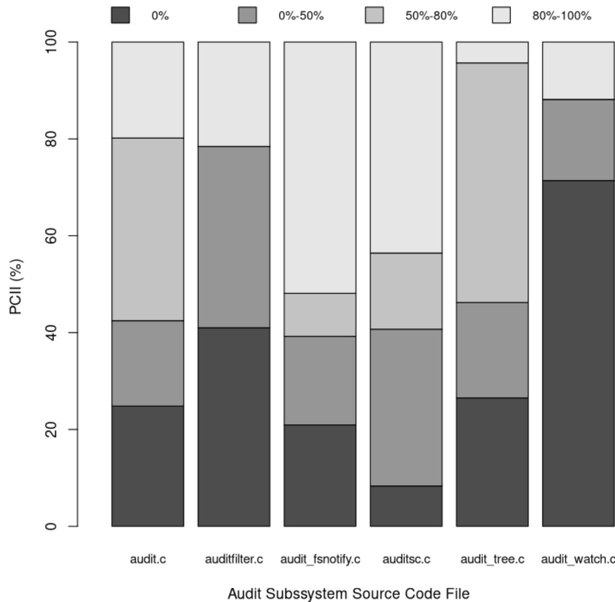

Figure 3. Stacked Bar Chart of Audit Subsystem Functions PCII Score Distribution Within Coverage Ranges

With the data arranged in this manner, it is clear to see where the coverage gaps are located and that they vary from one source file to another. However, combining the results of all the files reveals that, on average, most of the remaining coverage (32.16%) is related to the zero-coverage functions. The functions with low coverage rates (between 0% and 50% coverage) are responsible for 23.69% of the remaining coverage. 18.63% of the remaining coverage can be achieved once functions with reasonable coverage rates (between 50% and 80% coverage) are fully implemented. Lastly, about one-quarter of the remaining coverage is hidden in the gaps of functions considered to have high coverage rates (greater than 80%).

On average, zero and low coverage functions represent the most critical gap of the audit test-suite, since around 55% of the remaining coverage is related to these functions. Although well-covered already, the uncovered pieces of code in functions with high coverage rates represent a considerable amount of the remaining coverage and can also be considered a relevant gap in the test suite. These results suggest that creating new tests for zero-coverage functions, followed by improving the tests related to the low- and high-coverage ones, represents the bigger opportunities to enhance the audit test suite.

4.5. Individual Test Case Coverage Analysis

The audit-testsuite project currently contains 24 test cases, each designed to test a specific domain of the audit subsystem code. However, there is no evidence or documentation about each portion of the subsystem code that is meant to be exercised by a given test case, apart from the commit logs. Although not central for the effectiveness of the tests, this kind of information has associated value. It could potentially be used to make the execution of tests more targeted, thus optimizing automated testing, especially through Continuous Integration and Continuous Delivery/Deployment (CI/CD) pipelines.

Seeking to answer the last research question, the following method was used to identify which test cases are related to each source file of the audit subsystem. The data was collected similarly to Section 4.1 with one key difference: instead of running the full test suite at once, the GCOV counters were reset and the data was collected in between the execution of each test case within the test suite. The Table 2 presents the results obtained.

Each row of Table 2 is dedicated to a test case, and the columns show the percentage coverage rate provided by it on every source file studied. With the data presented in a tabular format, it is possible to identify that the files *audit.c*, *audit_filter.c*, and *auditsc.c* are exercised by all test cases within the test suite. For these files, a complete execution of the test suite is required upon a code change, since the coverage provided by the suite is spread among all its tests.

The files *audit_tree.c*, *audit_whatch.c*, on the other hand, are exercised only by a set of test cases. Notice that the former is just significantly exercised by the tests *file_permission* and *syscalls_file*. The latter is exercised substantially by a disjoint set of six test cases. Notice that running just the subset of test cases related to the specific source file is sufficient to test it; thus, the complete execution of the test suite is unnecessary in this case.

Lastly, it is easy to notice at a glance that the test case named *exec_name* is the only case in the entire test suite that, in fact, tests *audit_fsnotify.c* code. In this particular case, almost all the time spent on a complete execution of the audit-testsuite would not be testing this file at all.

The results above identify the relationship between test cases of the audit-testsuite project and audit subsystem source files. Although the files *audit.c*, *audit_filter.c*, and *auditsc.c* are deeply coupled with all the cases of the suite, and a complete execution of it should be triggered to test any changes impacting these files. For the other files, the developers could leverage this mapping information to make test executions more targeted by selecting only test cases of interest. Running only the test cases that need to be run upon a change set has the potential to optimize resources, such as hardware infrastructure and developer time.

The growing impact of time execution and resource consumption is a common concern when improving a test suite. However, applying the approach described above opens room for the creation and execution of new test cases aimed at testing currently untested areas of code, without

TABLE 2. RESULTS OF INDIVIDUAL TEST CASE COVERAGE PROVIDED TO AUDIT FILES

Test Case	Audit Source code file					
	audit.c	**auditfilter.c**	**audit_fsnotify.c**	**auditsc.c**	**audit_tree.c**	**audit_watch.c**
amcast_joinpart	35.94	4.32	0	16.54	0	0
backlog_wait_time_actual_reset	57.64	39	0	32.43	1.66	0
bpf	54.72	41.16	0	33.42	1.66	0
exec_execve	40.47	41.7	0	38.87	1.66	0
exec_name	38.58	50.47	55.68	31.83	1.66	4.46
fanotify	37.17	50.34	0	36.35	1.66	65.43
file_create	38.49	46.15	0	35.15	1.66	46.84
file_delete	38.49	46.15	0	34.68	1.66	46.84
file_permission	38.68	45.88	0	37.81	44.7	0
file_rename	38.49	46.15	0	35.28	1.66	46.84
filter_exclude	38.4	60.59	0	35.28	1.66	66.54
filter_saddr_fam	55.57	38.73	0	29.17	1.66	0
filter_sessionid	38.49	53.98	0	35.95	1.66	65.43
io_uring	54.81	44.13	0	39.2	1.66	0
login_tty	37.55	26.59	0	18.01	0	0
lost_reset	60.19	35.49	0	29.63	1.66	0
netfilter_pkt	33.77	26.59	0	18.8	0	0
signal	55.57	45.88	0	35.35	1.66	0
syscalls_file	38.68	46.96	0	39.67	46.03	0
syscall_module	38.49	44.26	0	35.95	1.66	0
syscall_socketcall	38.49	41.7	0	29.63	1.66	0
time_change	38.87	44.94	0	34.75	1.66	0
user_msg	24.34	4.32	0	11.5	0	0

necessarily increasing testing costs. One limitation, however, is that the mapping between source files and test cases tends to become outdated as both the subsystem and test suite code bases evolve. Keeping the mapping reliable over time requires its timely update, which can be seen as an adoption challenge.

4.6. Research Limitations Encountered

Throughout the development of this research, limitations were noticed in both the data collection method and the tooling used. The following two subsections are meant to discuss those.

4.6.1. Data Collection Method Limitations. There are some pieces of code in the Linux kernel that just run during the initialization phase of the OS. After that point in time, these pieces cannot be exercised. For instance, the function *audit_init*, listed in Table 1, is called by the initialization routine *postcore_initcall()* and thus cannot be executed after the reboot. Furthermore, this function is annotated as *__init*, which means its memory area is freed and reused by the kernel once the initialization process finishes.

Since the method used in this research foresees a GCOV data reset at the beginning of each data collection, all the initialization-only related functions are reported as zero-coverage ones. In practice, these zero-coverage functions represent false positives, as the coverage data collected does not account for their execution during the boot process. Out of the fifty-eight zero-coverage functions found, seven are false positives related to this limitation.

To overcome this limitation, a study related to the coverage provided by the initialization process itself should be considered, as a complement to the methodology proposed by this paper.

4.6.2. GCOV-related Limitations. Although we consider GCOV an effective tool for collecting coverage data of a running kernel, the following unexpected behavior in some data samples was eventually detected:

1) Inline functions originating from other source code files that are called in the analyzed source file are reported as functions of it. This added noise to the per-function coverage percentages reported by the tool.
2) It was observed that some functions were reported with coverage greater than 100% in per-function reports. A data cleanup was done before the data analysis in these cases.

The GCOV documentation [4] recommends disabling compiler optimizations to ensure it performs as expected; however, this instruction has not been followed throughout this study. Our goal was to analyze the subsystem as closely as possible, as a production kernel operates. The behavior, as mentioned earlier, may be related to the compiler optimizations.

Both the method and tooling limitations exposed above can be overcome by analyzing the data in conjunction with the code context. It was noticed that the optimal value of a coverage analysis process is obtained when the coverage metrics and the annotations of a *.gcov* file is studied in conjunction with the subsystem code and test suite contexts. These facts suggest that coverage metrics often automatically provided by third-party tooling should be taken with a grain of salt. Such data offer limited value without deep analysis, and relying on them might lead to misleading conclusions about the actual coverage provided by a given test suite.

5. Conclusions and Future Work

In conclusion, this article provides an overview of the current coverage state of the Linux kernel audit subsystem, revealing that although most of it is covered by the test suite, approximately 38% of the code remains untested. This represents a significant room for improvement.

It is highlighted that the main gap of the audit-testsuite project is primarily related to the lack of tests for functions with a low coverage rate within the subsystem. Additionally, it was noted that 51[1] functions of the subsystem are currently not being tested by the test suite.

The analysis of the data collected throughout this research led us to develop the PCII index, which can be explored as an additional metric in the coverage analysis process and in the prioritization of testing maintenance.

The relationship between individual test cases and source code files was also explored, and the results show which files are exercised by each test case. This information can potentially be used for optimizing resources by making automated tests more targeted.

Lastly, this study verifies that GCOV is a viable alternative for collecting coverage data from a running Linux kernel. Furthermore, the limitations highlighted can be overcome through contextual analysis and serve as valid knowledge for future related work.

Based on the findings of this research, the next steps are:

1) Propose a series of patches upstream containing both new test cases and enhancements to existing ones, aiming to improve the overall quality of the audit test-suite;
2) Propose a tool to suggest a set of test cases to be executed based on a kernel patch, leveraging the test/source file mapping presented by this paper;
3) Once the audit test-suite project achieves higher statement coverage rates, conduct a branch coverage analysis;
4) Correlate PCII with McCabe's cyclomatic complexity [9] metrics and verify if they can be used in conjunction to improve the testing and maintenance process.

1. accounting for false positives

References

[1] Imanol Allende et al. "Estimation of Linux Kernel Execution Path Uncertainty for Safety Software Test Coverage". In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 1446–1451. DOI: 10.23919/DATE51398.2021.9473951.

[2] Imanol Allende et al. "Statistical Test Coverage for Linux-Based Next-Generation Autonomous Safety-Related Systems". In: *IEEE Access* 9 (2021), pp. 106065–106078. DOI: 10.1109/ACCESS.2021.3100125.

[3] Imanol Allende et al. "Towards Linux based safety systems—A statistical approach for software execution path coverage". In: *Journal of Systems Architecture* 116 (2021), p. 102047. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2021.102047. URL: https://www.sciencedirect.com/science/article/pii/S1383762121000436.

[4] *gcov — a Test Coverage Program*. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html (visited on 06/10/2025).

[5] Paul Larson et al. "Improving the Linux Test Project with kernel code coverage analysis". In: *Proceedings of the Linux Symposium*. 2003, pp. 1–12.

[6] *Linux Audit Kernel Mirror*. URL: https://github.com/linux-audit/audit-kernel (visited on 06/10/2025).

[7] *Linux Audit Test Suite*. URL: https://github.com/linux-audit/audit-testsuite (visited on 06/10/2025).

[8] *Linux Code Coverage Repository*. URL: https://github.com/rprobaina/linux-code-coverage (visited on 06/13/2025).

[9] T.J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[10] R R Core Team, R Core Team, et al. *R: A language and environment for statistical computing*. 2020.

[11] Bo Sun, Yachao Shao, and Chao Chen. "Study on the Automated Unit Testing Solution on the Linux Platform". In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 2019, pp. 358–361. DOI: 10.1109/QRS-C.2019.00073.

[12] *Using gcov with the Linux kernel*. URL: https://www.kernel.org/doc/html/v4.14/dev-tools/gcov.html (visited on 06/10/2025).

[13] Haichi Wang et al. "An empirical study of test case prioritization on the Linux Kernel". In: *Automated Software Engg.* 32.2 (May 2025). ISSN: 0928-8910. DOI: 10.1007/s10515-025-00522-8. URL: https://doi.org/10.1007/s10515-025-00522-8.