When Do You Repeat Yourself? Voices from the Trenches of Linux Kernel Maintainers on Code Duplication

Luan Arcanjo, David Tadokoro, Marcelo Spessoto, Rafael Passos, Paulo Meirelles
Free Software Competence Center
Institute of Mathematics, Statistics, and Computer Science Universidade de São Paulo, Brazil
{marcelomspessoto,luanicaro,davidbtadokoro}@usp.br; rpassos,paulormm}@ime.usp.br

***Abstract*—The vast scale and continuous evolution of the Linux kernel make its maintenance a complex undertaking, where code duplication remains a persistent challenge that can hinder development and introduce bugs. This paper presents a multimethod ethnographic study investigating the socio-technical dynamics of contributing to the Linux kernel by reducing identified code duplication. To support this study, we developed a command-line utility tool for detecting function-level duplications, offering a concrete entry point for new contributors. Our ethnographic approach, including complete participant observation and participant-as-observer analyses, demonstrated that addressing duplications is viable for lowering the contribution barrier. This point is evidenced by 8 of 13 (62%) accepted contributions (patches) from 24 newcomers (16 undergraduate and 8 graduate students) in the Linux kernel project, collectively removing 1,397 lines of duplicated code. Nevertheless, the study reveals a more intriguing and complex reality beyond mindlessly removing clones. An analysis of maintainer feedback on both accepted and rejected contributions highlights a nuanced understanding of technical debt due to code duplication within the Linux kernel community, where the benefits of eliminating duplications are carefully weighed against factors such as readability, the introduction of new abstractions, and the specific code context.**

I. Introduction

Hundreds or thousands of developers working together on a software product is a complex and demanding task. Similar or even exact copies are a frequent and often unavoidable outcome. Excessive duplication can negatively impact the maintainability of a project, as, when it occurs, any change to a repeated code segment (to fix a bug, for instance) requires replication across all other copies. Developers must, therefore, remain constantly vigilant to synchronize different parts of the codebase, which is a highly error-prone yet seemingly simple responsibility. Regarded as a major source of technical debt, removing or refactoring duplicated code demands great care to preserve the original behavior, with minor mistakes possibly leading to serious problems in the future. A widespread dogma among developers is the *Don't Repeat Yourself* (DRY) principle [1], which asserts that to develop reliable and maintainable software, "every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

The Linux kernel is a foundational Free/Libre and Open Source Software (FLOSS) project, a vital component of the world's digital infrastructure. Considering only lines of code (no comments or blank lines) of strictly programming languages, in version 6.17, the main repository has more than 27 million lines of code and involved contributions from over twenty thousand developers. Within this context, code duplication remains a significant challenge, reducing the code readability and increasing the risk of introducing bugs [2], [3]. This issue is particularly prominent in kernel device drivers, which account for over 66% of the Linux kernel source code.

The detection of code duplication, or *code clones*, has been a research topic for decades [4]. The literature offers a widely accepted taxonomy that classifies clones into four types based on their similarity, from exact copies (Type-1) to semantically equivalent but syntactically different fragments (Type-4) [5]. Several detection techniques have emerged, including textual, token-based, tree-based, and graph-based approaches [5]. These efforts have culminated in state-of-the-art techniques, such as the graph-based approach proposed by Liu et al. [6].

To address code duplication in the Linux kernel, we developed a command-line tool capable of detecting and analyzing function-level duplications, called Arkanjo[1]. We conducted a multimethod ethnographic study leveraging the tool capabilities to identify duplications and submit contributions, directly or indirectly, to the Linux kernel, aiming to remove redundant code. Through interactions with maintainers, this study gained insights into how the Linux community perceives the code quality related to code clones. Contrary to the DRY principle, our findings indicate that, in some contexts, maintainers can tolerate duplications for better readability, performance increases, or even to avoid integration overhead.

Ethnography is a well-established qualitative method for understanding people, their cultures, and work practices [7]. It provides insights into community members' values, beliefs, and practices [8]. In this study, we first employed a complete participant observation approach [9], actively contributing by performing deduplications while documenting the process. We then conducted a participant-as-observer study [9] with students resolving duplications identified by the tool. Beyond demonstrating that the tool lowered the entry barrier for new contributors, these studies yielded intriguing observations on kernel Linux deduplications. This paper describes the afore-

[1] github.com/arkanjo-tool/arkanjo

mentioned studies, along with our visions and reflections from diving into the Linux kernel development process.

II. The Linux Kernel

The Linux kernel is a complex project organized into subsystems, such as the process scheduler, memory management, and device drivers. Each subsystem typically has a dedicated maintainer or a team of maintainers overseeing its development and managing contributions. The development process is coordinated through Git (also designed by Linus Torvald, focused on the Linux kernel development), and contributions are formatted as patches, text documents that outline the differences between two source code versions. These patches are then submitted to the relevant Linux mailing lists for public review and discussion.

This study focuses on two of these subsystems, selected to represent different contexts for contribution. We explore the *AMD Display drivers*, chosen for their importance within the hardware ecosystem, which is responsible for natively enabling AMD GPU functionality in Linux. This subsystem is substantial, with over 391,000 lines of code across more than 1,089 files. We also investigate the *Industrial Input/Output (IIO)* subsystem, often considered an accessible entry point for new contributors. The IIO subsystem provides support for devices that perform analog-to-digital or digital-to-analog conversions[2] and contains over 281,772 lines of code in more than 755 files.

III. Research Strategy

Preliminarily, we conducted an initial exploratory survey of existing static analysis tools, including PMD, Duplo, Simian, and CodeClimate. While these tools offer established detection capabilities, we observed practical limitations when applying them to the scale and specificities of the Linux kernel. Many solutions were either restricted by commercial licensing, relied exclusively on token-based matching, or, in the case of FLOSS alternatives like the *duplicate-code-detection-tool*, were limited to file-scope comparisons, thereby missing intra-file duplications. To overcome these specific constraints and streamline our analysis, we developed a custom command-line tool, Arkanjo, adapted to detect function-level redundancy within large C codebases.

The tool, released under the LGPLv3, employs a standard text similarity method based on TF-IDF (Term Frequency-Inverse Document Frequency) vector embeddings [REF ArKanjo DebConf]. By computing cosine similarity between C functions, the tool addresses the "same-file" blind spots of previous file-scope parsers. The decision to implement this specific instrumentation was driven by the need for a simple, lightweight CLI that prioritizes maintainability, allowing the research focus to remain on the qualitative analysis of the code and subsequent community interactions, rather than on the tool itself.

To support the iterative nature of the study, the tool utilizes a two-stage architecture that decouples the computationally intensive parsing from data retrieval. We measured the preprocessor's execution time on a machine equipped with a Ryzen 5700X processor and 32GB of RAM. Across multiple runs on complex subsystems, such as the AMD Display driver and IIO, the worst-case indexing time was approximately 9 minutes and 41 seconds. In contrast, the query responder processed requests in under 2 seconds. These tests confirmed that the tool was sufficiently efficient to enable our ethnographic study, providing rapid access to analysis data without introducing significant overhead to the workflow.

Using Arkanjo, we proceeded to assess the viability of reducing the identified duplications through an ethnographic study using a participant observation methodology [9]. In the first phase, adopting the role of *complete participant* [9], we engaged directly with the Linux kernel community by submitting deduplication patches to the AMD Display driver. In the second phase, shifting to the role of *participant-as-observer* [9], we introduced a similar activity within a university course, where students were tasked with proposing patches for the AMD Display or IIO subsystems. Throughout this process, we provided guidance and mentorship to the students as they interacted with the community.

A. Complete Participant Observation Study

To investigate whether we could use the tool to find ways to contribute to the Linux kernel, we first conducted a complete participant observation experiment, where the main author (a graduate student) acted as a first-time contributor to the AMD Display driver, collecting artifacts of our experience in the process.

We executed the tool in the AMD Display driver codebase and manually analyzed the biggest duplications regarding the number of lines, finding one pair that we judged promising to try to deduplicate. We approached big duplications as proof of concept to gauge the tool's potential for enabling significant impacts.

Given the duplicated function pair, we observed that the files in the functions contained multiple other duplications. Thus, we proposed a simple systematic approach to mitigate all the functions duplicated in the context, not just the duplicated function pairs. After approaching the duplications with a systematic strategy, we sent a patch to the AMD development mailing list for the feedback of maintainers, while documenting the process, interactions, and impressions (available in our replication package[3]).

In C programming, communication between source files is achieved by creating header files that specify libraries [10]. Since the Linux kernel is primarily written in C, our systematic strategy consisted of eliminating code duplications through consolidating the duplicated code into a single library, which would replace instances of duplication across the codebase. For each duplicated function pair, for each function in the pair, we used the tool to locate all duplications of the function in the codebase, as it could have been duplicated more times than

[2]https://kernel.org/doc/html/v6.17/driver-api/iio/intro.html

[3]https://doi.org/10.5281/zenodo.17503684

the two occurrences detected initially. This strategy resulted in a collection of duplicated functions and corresponding code files.

To identify shared code more effectively, we extended this approach to search for other common functions across all collection files. We then applied specific refactoring methods to each shared function. If the functions were identical across files, we removed the duplicates and created a single function in the library. If modifications existed, we applied case-specific refactoring.

B. Participant-as-Observer Observation Study

To broaden our study and observations, we adopted a participant-as-observer approach, involving students as FLOSS project newcomers to make practical contributions to the Linux kernel. This approach formed the core activity of a university course, in which we guided students to remove code duplications in the Linux kernel as one possible pathway for contributing to the project in the first phase of the course [omitted].

The 2025 course offering had 37 students (25 undergraduate and 12 graduate), who were asked to form groups of two or three members, but graduate students could work individually. To assist students in this task, we (a professor and three more experienced graduate students) prepared several alternatives for contributing to the Linux kernel, including simpler contributions such as fixing coding style issues. In this context, we specified two options for removing duplications in the Linux kernel.

Deduplication Option 1*:* We ran the tool on the IIO subsystem to generate a list of duplicated function pairs, then curated it to highlight actionable cases for students. Specifically, we (1) kept only pairs within the same file, since cross-driver duplications could involve multiple maintainers or distinct interfaces; (2) removed very short duplications; and (3) manually ranked the remaining pairs, adding annotations to guide students toward entries likely to be accepted. In this option, students were tasked with selecting a recommended entry, devising a way to remove the duplication, and submitting their patches to the IIO mailing list for review.

Deduplication Option 2*:* We offered an experience similar to our initial complete participant observation, where students managed the entire workflow: running the tool, analyzing results to identify potential deduplications, creating a patch, and submitting it to the driver's maintainers. This option provided freedom of choice but made the students' task more complex.

Of the students who chose to work on the tool-related tasks, 23 (16 undergraduate and 7 graduate) formed 11 groups to pursue the first alternative. One graduate student opted for the second alternative. Notably, none of these students had prior experience contributing to the Linux kernel, making this their first attempt to submit a patch as newcomers to the project.

It is important to stress that, **no matter the chosen approach, we supported all groups by providing review cycles on their contributions before submission**, to prepare students for the interactions with the maintainers and alleviate rough, but natural, newcomers' mistakes. After the submission, we helped students understand the feedback from maintainers and gave directions on how to move forward. This close, yet indirect, participation through intimate mentoring characterizes the second phase as our participant-as-observer observation.

For this second phase, our primary data source was the mailing list threads, where all development occurred (also available in our replication package). Using blog posts, the groups documented their experiences submitting a patch to the Linux kernel [omitted], and we also leveraged those to complement our observations.

IV. Results

In the **complete participant observation** study (first phase), we chose one function pair found on the AMD Display Driver to approach and apply the systematic approach. We reached a point where we had an idea of removing the duplications found, but we could not complete the refactoring to a state capable of submitting as a patch to the driver. We found that the configuration files on the AMD Display driver are designed to be imported at compilation time using *#define* macros. The configuration files' design choice makes refactoring duplications on generic approaches trickier, as refactoring code that depends on these files requires significant modification of the configuration files' design and deep knowledge of the codebase, which we, as first-time contributors, did not have. Thus, we opted not to continue investigating this deduplication. We found that the duplicated function from the code reduction existed in five code files, and two other functions were duplicated in those five. All functions across the files were exactly equal, so there was no need to apply the refactoring methods presented in the literature. The refactoring was resumed to create a generic library and fix the compilation targets.

At this stage, we considered the deduplication proposal good enough to submit it to the AMD Display driver as a patch. We sent the first patch on August 9, 2024, and a resend followed on October 9, 2024, due to a lack of initial feedback. Nevertheless, for the first two versions, the maintainers' response requested minor changes for coding style, license use, and alignment with existing conventions. In the third version, the feedback was to move the new generic library functions to an existing file, instead of creating a new one. Then, the fourth version was accepted and integrated into the kernel on February 25, 2025.

Since we suffered a delay in the review process, we directly contacted a maintainer to understand why. From this conversation, we understand that one reason for the delay was that duplicated code enhances the independence of GPU driver code. This approach allows developers to make changes to a specific GPU without needing to test compatibility with others, which helps save a significant amount of time and effort. This example of hardware variations, also mentioned by Kapser and Godfrey [11], leads us to the primary conclusion of our complete participant observation:

Takeaway T0: *Driver developers do not necessarily view duplicated code negatively, which contradicts standard software engineering practices.*

Building on this insight, we realized that the perception identified in takeaway T0 might not be limited to AMD Display alone. To deepen our understanding, we examined other contributions regarding code duplication reduction and analyzed how different maintainers responded to such patches. This broader analysis aimed to capture diverse maintainer perspectives and contextual factors influencing their decisions.

TABLE I
SUMMARY OF NEWCOMER DEDUPLICATION CONTRIBUTIONS.

GID	SS	SML	Patch Status	Takeaway	Diff
0	AMD	100%	Accepted (v3)	T0	+114/-520
1	IIO	100%	Dropped (v2)	T1, T2	+7/-14
2	IIO	100%	Dropped (v1)	T1	+23/-27
3	IIO	100%	Accepted (v1)	None	+12/-30
4	IIO	100%	Accepted (v3)	T1	+2/-7
5	IIO	90%	Accepted (v1)	None	+17/-70
6	IIO	90%	Dropped (v1)	T3	+95/-92
7	IIO	90%	Accepted (v3)	None	+20/-36
8	IIO	90%	Accepted (v4)	T4	+20/-16
9	IIO	90%	Accepted (v6)	T4	+149/-243
10	IIO	90%	Dropped (v1)	T1, T4	+32/-58
11	IIO	100%	Dropped (v1)	None	+2/-12
	AMD	90%	Accepted (v2)	None	+90/-489

Table I summarizes all the contributions sent by the main author and student groups (GID) [4]. It contains the information about the chosen subsystem (SS) and the similarity threshold (SML). Each entry also has the patch status, takeaways that may have surfaced from the interaction, and a code differential corresponding to the final patch versions before it was dropped or accepted into the code base. The code differential for a set of patches corresponds to the sum of added and removed lines of each patch.

We derived four takeaways from our **participant-as-observer observation**. We substantiate these takeaways with quotes from the maintainers that originated during the review process[5].

Groups 1 and 4 worked on similar duplications using the extract method on predicate functions (functions that return a boolean) that, essentially, were the negation of one another. Group 4 initially used macros, but, after feedback from maintainers, both solutions converged toward keeping one of the functions while modifying the other to return the negation of the former. Both cases suffered substantial mutations, even though they were correct in their first iterations, and these changes were primarily driven by the feedback of maintainers regarding readability. For Group 1, a maintainer said that *"[...] the naming as `_reg_check()` is not helpful as it doesn't indicate anything specific is being checked."*, while for Group 4, the feedback explicitly highlighted the precedence of readability over deduplications: *"I think the old code is more readable than hiding the values in a macro even if it is duplicating a few lines of code."*.

Even though the deduplication of Group 2 was not similar to that of Group 1 and Group 4 in terms of implementation (it used the parametized method), it also brought forth the readability factor: *"In my view this isn't a significant enough reduction to justify the more complex code."*.

These experiences revealed a recurring concern among maintainers beyond functional correctness or redundancy reduction. Instead, it reflected a preference for code clarity and comprehension. From these consistent insights that originated from multiple review interactions, we formulated our first takeaway:

Takeaway T1 (Readability): *Maintainers prioritized readability over removing duplications in many different contexts. In their view, going against the DRY principle was worth it if it resulted in easier-to-read and understand code.*

Still focusing on Group 1, maintainers questioned the merit of the proposed patch. While the previous interactions highlighted how maintainers often favor readability over strict code deduplication, this case added a new dimension to our understanding: the *cost-benefit* evaluation of proposed changes. A maintainer pointed out that the effort required to propagate such modifications after acceptance (i.e., during the upstreaming process) could outweigh the perceived benefits: "*[...] such patches might not worth it since the proposed improvement is very small (and questionable) while the upstreaming process still requires some effort.*". From this feedback, we crafted our second takeaway:

Takeaway T2 (Integration Overhead): *Maintainers may evaluate the trade-off between the impact of contributions (not only deduplications) and the struggle to integrate them into the complex Linux project.*

Groups 3, 6, and 10 used the parameterized method by introducing a new struct used as a parameter to a generic function. Notwithstanding that the same maintainer reviewed all three contributions, only Group 3 had its patch accepted. Our understanding was that this change, in particular, acted on a simple iterative algorithm that (colaterally) also fixed a bug, which favored its straightforward acceptance.

Although removing duplications does not necessarily mean reducing lines of code, the patch from Group 6 resulted in a positive net change in lines of code, which the maintainer pointed out as a drawback. Still, the real reason for the rejection was that the deduplication scattered parts of a complex code that resided close together before: *"Wrapping this up doesn't provide any real advantage, requiring as it does the reviewer to look at this function AND where the value is set rather than seeing them in one place."*. Needing to jump to a function definition (to see its behavior) or a function

[4]Since the first author was also a newcomer, the first phase experience was included as Group 0.

[5]We **do not claim** these quotes fully represent the maintainers' stance on code duplication.

call (to see its concrete parameters) to understand code is a cognitively demanding task for developers [12]. Merging these informations, we derived our third takeaway:

> **Takeaway T3 (Cognitive Load):** *Maintainers can justify duplications if they avoid having to jump back and forth in the code when parsing through the file.*

The reason for rejecting the patch from Group 10 involved the deduplication lowering readability, as the maintainer was *"[...] not sure the code reduction is sufficiently to cover the resulting loss of readability."*, reinforcing the takeaway T1.

In the interaction of Group 10, there was also a hint that performance could be considered when evaluating the worth of a deduplication, whilst reaffirming that readability was a priority in that context: *"I'd be slightly interested to see the optimized output of the two approaches, but this is far from a high performance path so we care a lot more about readability here."*. Nonetheless, this notion surfaced aggressively with Group 10, where the contribution could completely remove both duplicated functions using the inline method by leveraging helpers; however, one of the functions had a call that, if eliminated, would result in a significant performance impact: *"Note that is not an appropriate change for the large reads though [...] This is the one case were* `spi_write_then_read()` *is probably not appropriate due to the large buffers that are potentially involved."*. These interactions, coupled with the low-level development context of Linux and driver development, which is usually related to devices with limited resources, produced our fourth and final takeaway:

> **Takeaway T4 (Performance):** *Interactions with maintainers hinted that duplications can be forgiven if they improve performance in contexts where resources are limited.*

Groups 7 and 8 used the extract method using helpers to remove duplicated code in the function's body. Group 7 refined its patch from the maintainers' feedback, resulting in a smooth acceptance process that did not produce any insights. Group 8 followed a similar course, but generated an interaction reinforcing the takeaway T4. In the feedback of version 1, a maintainer claimed that *"Even though there is less code repetition, we now have an extra comparison [...]"*, with another maintainer corroborating this in version three by saying: *"There is always a balance/trade-off between modularity and execution speed. I agree with* `anonymous-maintainer`*'s reply in the first patch [...]"*, regardless of the fact that the patch was eventually accepted in version four.

We did not explore the contributions of Group 5 and Group 11 in detail, as they did not result in (or influence) any insights regarding these fascinating perspectives from the Linux community that are dissonant with the apparently unquestionable DRY principle. However, they were still meaningful as they produced impactful contributions that reduced a lot of duplicated code. Especially in Group 11, where a single student executed the whole workflow we did in the first phase, and managed to remove four hundred net lines of code.

In this sense, we discuss another aspect of importance stemming from this work: using the developed tool to create deduplication opportunities in the Linux kernel can be an excellent introduction for newcomers.

To illustrate this point, we display some statistics concerning the 13 contributions contemplated in this paper:

- Collectively, the patches proposed the insertion of 585 lines and the removal of 1626 lines.
- 8 out of 13 contributions were accepted, resulting in a success rate of 62%.
- For accepted contributions, the code diff value was +255/-1152, averaging +31.9/-144.0 additions/removals.
- For rejected contributions, the code diff value was +181/-231, averaging +36.2/-46.2 additions/removals.

V. Concluding Remarks

This ethnographic study, combining complete participant and participant-as-observer observations, provided a realistic view of the opportunities and challenges in removing code duplications detected by the tool. First-time Linux kernel contributors could effectively use the tool to identify duplicated functions and submit meaningful patches.

Throughout this process, we observed that maintainers do not necessarily regard code duplication as inherently harmful. Instead, their decisions reflect a nuanced and context-dependent understanding of software quality. As evidenced by our takeaways, readability often outweighed strict adherence to the DRY principle (T1), while the effort required to propagate changes also influenced maintainers' acceptance decisions (T2). Moreover, avoiding cognitive overhead when navigating the code (T3) and preserving performance in resource-constrained environments (T4) emerged as additional justifications for tolerating certain forms of duplication.

Overall, these findings reveal that, within the Linux kernel community, the evaluation of code duplication extends beyond technical redundancy. Maintainability, clarity, and practical trade-offs play a decisive role in shaping maintainers' judgments, suggesting that the DRY principle is sometimes less valuable than keeping the code understandable, efficient, and integrated smoothly into a large and evolving system.

Replication Package

All anonymized threads containing the entire review process and interactions with maintainers are available in this Zenodo repository: https://doi.org/10.5281/zenodo.17503684.

To avoid violating the double-anonymized review process, the anonymous repository of the developed tool is available at https://anonymous.4open.science/r/anonymous-repo-tech-debt/. The tool website link and omitted references will be added in the camera-ready version of the paper.

REFERENCES

[1] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[2] W. Hordijk, M. L. Ponisio, and R. Wieringa, "Harmfulness of code duplication-a structured review of the evidence," in *13th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. BCS Learning & Development, 2009.

[3] K. Hotta, Y. Sasaki, Y. Sano, Y. Higo, and S. Kusumoto, "An empirical study on the impact of duplicate code," *Adv. Soft. Eng.*, vol. 2012, 1 2012. [Online]. Available: https://doi.org/10.1155/2012/938296

[4] H. T. Jankowitz, "Detecting plagiarism in student pascale programs," *The computer journal*, vol. 31, no. 1, pp. 1–8, 1988.

[5] C.-F. Chen, A. Zain, and K.-Q. Zhou, "Definition, approaches, and analysis of code duplication detection (2006–2020): a critical review," *Neural Computing and Applications*, vol. 34, pp. 1–31, 08 2022.

[6] J. Liu, J. Zeng, X. Wang, and Z. Liang, "Learning graph-based code representations for source-level functional similarity detection," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 345–357.

[7] W. A. Edmonds and T. D. Kennedy, *An applied guide to research designs: Quantitative, qualitative, and mixed methods*. Sage Publications, 2016.

[8] H. Sharp, Y. Dittrich, and C. R. De Souza, "The role of ethnographic studies in empirical software engineering," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 786–804, 2016.

[9] R. L. Gold, "Roles in sociological field observation," *Social Forces*, vol. 36, pp. 217–223, 1958.

[10] B. W. Kernighan and D. M. Ritchie, *The C programming language*. prentice-Hall, 1988.

[11] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," in *2006 13th Working Conference on Reverse Engineering*, 2006, pp. 19–28.

[12] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: Association for Computing Machinery, 2006, p. 23–34. [Online]. Available: https://doi.org/10.1145/1181775.1181779