ArKanjo: a tool for detecting function-level Code Duplication in the Linux Kernel

Luan Arcanjo
Universidade de São Paulo, Brazil
luanicaro@usp.br

David Tadokoro
Universidade de São Paulo, Brazil
davidbtadokoro@usp.br

Paulo Meirelles
Universidade de São Paulo, Brazil
paulormm@ime.usp.br

***Abstract*—The Linux kernel massive scale (+28 M LoC, +20 K contributors) presents unique maintenance challenges. Surprisingly, code duplication remains a persistent issue in the kernel codebase, which can hinder its evolution and patching. Academic approaches often focus on pairwise comparison of code artifacts, which are not directly applied to comprehensive codebase analyses. Other existing free software[1] tools explored in practice frequently suffer from limited functionality, such as primitive textual matching, prove too narrow in scope, or fail to deliver effective results on complex, large-scale codebases. Existing solutions generally fail to address the Linux kernel-specific needs: (1) scalability to handle its size, (2) actionable results for developers, and (3) integration with kernel development workflows. This paper presents *ArKanjo*, a novel command-line tool for Linux kernel maintenance designed to detect and analyze function-level duplications. Released under the MIT license, ArKanjo employs a two-stage architecture consisting of a Preprocessor and a Query Responder that separates computationally intensive analysis from efficient querying for duplications within large codebases. Pivotal advantages of ArKanjo over existing solutions include: (1) preprocessing that enables rapid queries without redundant analysis; and (2) prioritization of duplicates that impact maintainability, such as copied buggy logic. We evaluate ArKanjo against real-world duplication cases in recent kernel versions, demonstrating its effectiveness in identifying problematic clones that generic tools often overlook. By identifying well-defined, manageable duplication instances, ArKanjo effectively lowers the barrier for new contributors, a capability evidenced by its role in guiding students to make their first code improvements to the kernel. ArKanjo offers immediate value to kernel maintainers and serves as a replicable model for clone detection in other large-scale free software projects.**

I. Introduction

The Linux kernel is a foundational free software project critical to a significant portion of the world digital infrastructure. Maintaining the kernel is an enormous undertaking involving over 28 million lines of code and contributions from over twenty thousand developers. Within this context, code duplication persists as a significant challenge, a known harmful practice that negatively affects code readability and increases the likelihood of introducing bugs [1], [2]. This issue is particularly acute in kernel device drivers, which comprise over 66% of the source code [3]. For instance, maintainers of the AMD Display driver have specifically highlighted code duplication as a significant impediment to their work.

[1]In this paper, we use the term *Free Software* interchangeably with *Open Source Software*.

Detecting code duplication, or code clones, has been a research subject for decades [4]. The literature provides a standard taxonomy, classifying clones into four types based on their degree of similarity, from identical copies (Type-1) to semantically equivalent but textually different fragments (Type-4) [5]. Various detection methodologies have been developed, including textual, token-based, tree-based, and graph-based approaches [5]. These have culminated in state-of-the-art techniques, such as the graph-based work by Liu et al. [6].

However, the primary focus of such academic work remains on determining whether a given pair of code artifacts is a duplicate rather than providing a scalable method to scan an entire codebase for actionable results. Conversely, existing free software tools explored in practice, often lack this sophistication, relying instead on primitive textual matching that is insufficient for the complexity and scale of the kernel.

To fill this gap, this research proposes a new approach embodied in *ArKanjo*, a tool designed specifically to identify and facilitate the mitigation of function-level code duplications within the Linux kernel. We employ multiple analytical methods and ethnographic studies to test it.

II. ArKanjo Tool

ArKanjo, our proposed tool, is a Command Line Interface (CLI) application designed to help developers identify code clone duplication at the function level. Released under the MIT license, ArKanjo is available at *github.com/arkanjo-tool/arkanjo*.

A. Architecture

The tool operates in two main parts, as illustrated in Figure 1: the **Preprocessor** and the **Query Responder**. The **Preprocessor** performs heavy computations to find code duplications across a codebase and produces artifacts to store duplication-related information in a structured form. The **Query Responder** consumes the artifacts produced by the Preprocessor to execute the tool functionalities as requested. This design allows the tool to perform the heavy and time-consuming steps only once per codebase, enabling fast performance for multiple queries on the same codebase.

The Preprocessor contains two main components: the Function Breaker and the Code Duplication Finder. The workflow of the Preprocessor is as follows: the Function Parser receives the input codebase, extracts the functions along with metadata,

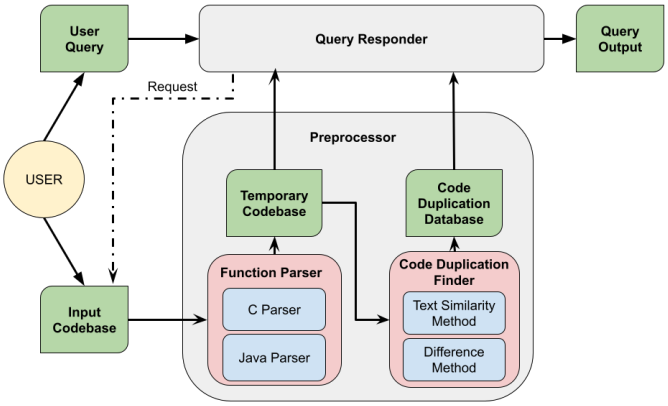

Fig. 1. Architecture diagram with the tool components

and creates a temporary codebase where each extracted function becomes a new code file. The Code Duplication Finder then iterates over every pair of files in the temporary codebase, checks whether they are code clones, and, if so, stores the result in the Code Duplication Database. This database is a text file that records each code duplication as a triple <**function1, function2, similarity**>, where **function1** and **function2** are the duplicated functions, and **similarity** is the score returned by the duplication detection method used in the Code Duplication Finder.

The Query Responder uses the temporary codebase and the Code Duplication Database to extract information about duplicated functions based on the user's request. If the user executes the Query Responder without previously executing the Preprocessor, the Query Responder automatically calls the Preprocessor to generate the required artifacts.

The Function Parser receives the input codebase and transforms it into the temporary codebase. It iterates through every source code file written in a supported programming language and uses a language-specific extractor to isolate each function. For each extracted function, two new source code files and a metadata file are created in the temporary codebase, represented by the pair <**file name, function name**>, the source code of the function, and the function metadata (in practice, this is a duplication of the original codebase, structured as a parsed representation). One of the new files contains the function body, and the other contains the function signature. The metadata file includes additional relevant information, such as the function name, the line where the signature starts, and the line where the function body ends. Currently, the supported programming languages are C and Java, although Java support is limited.

The Code Duplication Finder iterates through every pair of source code files in the temporary codebase, each representing a function from the input codebase. For each pair, a code duplication detection method computes a similarity score. If the similarity is greater than or equal to the minimum threshold (provided by the user during preprocessing), the pair is stored in the Code Duplication Database along with the similarity score.

We implemented two duplication detection methods in the tool, which users can choose between. The first method is based on text similarity, and the second is a more straightforward approach based on the number of identical lines. The experiments and tests in this research were conducted using the text similarity method.

1) Code Duplication Methods Used: For the text similarity method, we treat the source code files as plain text and apply the TF-IDF vector embedding method implemented by the Gensim library [7], then compute cosine similarity as the similarity metric. We chose this method for its reported performance, programming language independence, and the fact that it does not require compilable code, an important aspect given that the temporary codebase does not contain complete code artifacts.

We implemented a more straightforward approach for the difference method, considering the number of exactly equal lines between two functions. For two functions, *function1* and *function2*, we compute the similarity as the ratio of duplicated lines to the total number of lines across both functions. This method is considerably simpler and more explainable than the text similarity method. The metric is defined by the proportion of common lines between the two functions. We use the *diff* command built into the Linux environment to compute the number of equal lines.

B. Functionalities

We propose three main functionalities for the user in the tool: Duplication Explorer, Function Information, and Duplication Report. Each functionality accepts specific parameters to perform its operations.

```
→ arkanjo git:(main) ✗ ./exec ex -l 5 -c T -p bios -s 90
It was found a total of 119 pair of duplicate functions in the codebase.
 Which the first 5 can be found below.
--------------------
Functions find: dc/bios/command_table.c::transmitter_control_v3 AND dc/b
ios/command_table.c::transmitter_control_v2 , TOTAL NUMBER LINES IN FUNC
TIONS: 133
Functions find: dc/bios/command_table.c::transmitter_control_v3 AND dc/b
ios/command_table.c::transmitter_control_v4 , TOTAL NUMBER LINES IN FUNC
TIONS: 133
Functions find: dc/bios/command_table.c::transmitter_control_v4 AND dc/b
ios/command_table.c::transmitter_control_v2 , TOTAL NUMBER LINES IN FUNC
TIONS: 122
Functions find: dc/bios/bios_parser.c::get_embedded_panel_info_v1_2 AND
dc/bios/bios_parser.c::get_embedded_panel_info_v1_3 , TOTAL NUMBER LINES
 IN FUNCTIONS: 118
Functions find: dc/bios/bios_parser.c::update_slot_layout_info AND dc/bi
os/bios_parser2.c::update_slot_layout_info , TOTAL NUMBER LINES IN FUNCT
IONS: 100
```

Fig. 2. Example of the *Duplication Explorer* functionality.

The Duplication Explorer is the primary functionality of our tool, designed to present the user with pairs of duplicated functions identified by the tool. We implement optional filters to support more complex queries. Figure 2 demonstrates an example of this functionality in use.

The Function Information functionality provides detailed information about a specific function. It receives a target function from the user and returns information such as the relative

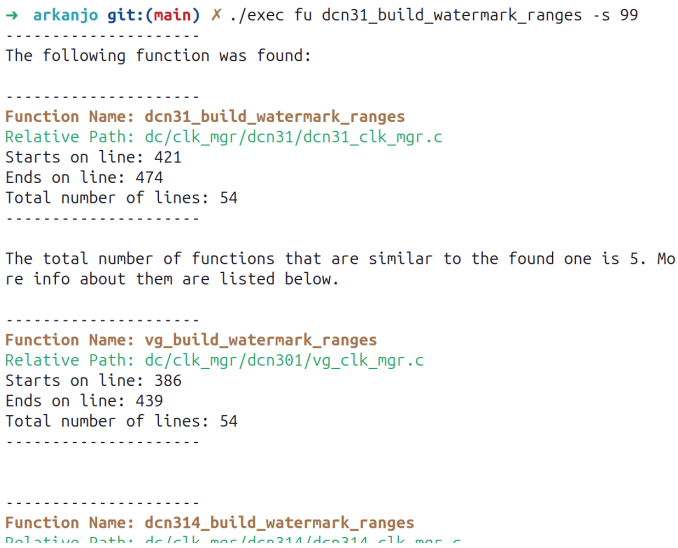

Fig. 3. Example of the *Function Information* functionality.

path, function name, and line numbers where the function is defined. Additionally, it provides similar information for every function that duplicates the given function. Figure 3 demonstrates an example of this functionality in use.

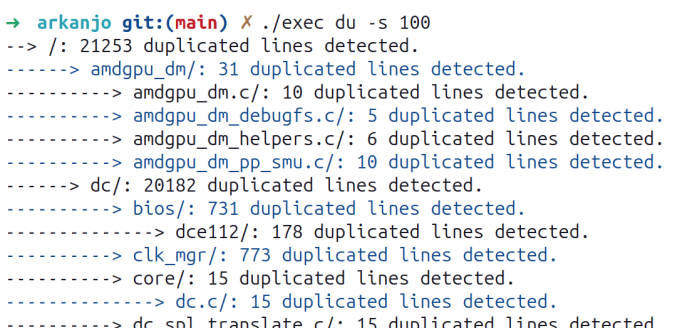

Fig. 4. Example of the *Duplication Report* functionality.

The Duplication Report functionality provides an overview of duplicated code within the input codebase. It calculates the number of duplicated lines per folder in the codebase and presents the information to the user in a readable format, as demonstrated in Figure 4.

III. Methods

We applied two independent evaluation approaches to validate the capabilities of the ArKanjo tool in finding code duplications using the text similarity detection method. The first method assesses the tool using a literature-based approach, comparing it against the BigCloneBench dataset [8]. The second method consists of an empirical analysis of a subset of functions within the AMD Display driver that our tool identified as duplicates.

To validate the tool against the BigCloneBench dataset [8], we followed the methodology presented by Liu et al. [6]. We sampled 20,000 pairs from each clone type and added 20,000 non-duplicate pairs as negative samples. We applied the same sampling strategy to our tool to ensure a fair comparison.

Unlike state-of-the-art methods, our tool does not distinguish between clone types. Additionally, it identifies duplications at the function level, whereas most existing tools operate at the file level. Therefore, we adapted the metric calculation method for evaluation purposes. Specifically, we considered every pair of functions with a similarity metric equal to or greater than a threshold X as duplicates and marked the corresponding file pairs. A correctly identified duplication pair is counted as accurate for its clone type, while an incorrectly inferred pair is considered incorrect across all clone types. To understand the impact of varying the similarity threshold, we evaluated our tool using different threshold values: 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100%. We then analyzed and discussed the results.

For the empirical analysis, we randomly sampled function pairs identified by the tool as duplicates. For each similarity threshold X (30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100%), we randomly selected ten function pairs with a similarity close to X, allowing for a 1% deviation.

We conducted a multi-part ethnographic study to assess whether the duplications found by the ArKanjo tool were actionable. In the second semester of 2024 and the first semester of 2025, respectively, the main author, conducting a first validation, and 12 student groups (19 undergraduate and 7 graduate students) from the Free Software Development course at the University of São Paulo acted as new contributors to the Linux kernel in a broader experiment, used the tool to identify duplications within the Industrial I/O (IIO) subsystem and the AMD Display driver subsystem refactored the code to address the issues, and submitted their proposed fixes as patches to the community maintainers. The students documented their refactoring approaches and experiences interacting with the kernel community throughout this process.

IV. Results

In this section, we assess the ArKanjo tool effectiveness in detecting code duplications through benchmark comparison and empirical analysis. We also describe findings from an ethnographic study involving real contributions to the Linux kernel, highlighting the tool practical utility and limitations in developer workflows.

A. Tool Evaluation

Table I shows the results obtained by our tool on the BigCloneBench dataset, and Table II presents the results from the empirical analysis. The outcomes from both the BigCloneBench evaluation and our empirical method indicate that our tool performs well in detecting Type-1 and Type-2 code clone duplications, which are sufficient to reveal propagated issues like copied buggy logic. However, it struggles with more complex clone types. A similarity threshold between 80% and 100% yields favorable results in both methods.

Nevertheless, a discrepancy emerges when detecting non-duplicate pairs between the two methods. In the Big-

TABLE I
RECALL OF THE ARKANJO TOOL IN THE BIGCLONEBENCH DATASET.

Similarity Threshould	**T1**	**T2**	ST3	MT3	WT3/ T4	**False**
100%	100%	5%	6%	0%	0%	100%
90%	100%	85%	26%	0%	0%	100%
80%	100%	87%	37%	1%	0%	100%
70%	100%	88%	44%	2%	0%	100%
60%	100%	89%	49%	4%	0%	100%
50%	100%	90%	64%	6%	0%	100%
40%	100%	90%	68%	9%	0%	100%
30%	100%	98%	74%	13%	0%	100%

TABLE II
RESULTS OF THE ARKANJO TOOL ANALYZING THE AMD DISPLAY DRIVER.

Similarity range	**T1**	**T2**	T3	T4	**False**	**Success Ratio**
99% - 100%	9	0	0	1	0	**100%**
89% - 91%	0	8	0	1	1	**90%**
79% - 81%	0	3	2	3	2	**80%**
69% - 71%	0	3	1	1	5	**50%**
59% - 61%	0	0	0	2	8	**20%**
49% - 51%	0	0	0	0	10	**0%**
39% - 41%	0	0	0	0	10	**0%**
29% - 31%	0	0	0	0	10	**0%**

CloneBench evaluation, we found no false positives (i.e., negative samples inferred as duplications), whereas the empirical analysis revealed a higher percentage of false positives. We propose two potential explanations for this discrepancy.

The first explanation concerns the limitations and known issues with BigCloneBench, as highlighted by Krinke and Ragkhitwetsagul [9]. The second explanation relates to the nature of the AMD Display driver, where code artifacts often share semantic meaning. In contrast, BigCloneBench comprises self-contained code artifacts with minimal or no shared semantics. Since our tool relies on a text-based code clone detection approach, it may naturally perform worse on the AMD Display driver than on the BigCloneBench dataset.

While we did not conduct a formal performance benchmark, we measured the Preprocessor's execution time on a machine equipped with a Ryzen 5700X processor and 32GB of RAM. Across 5 runs on the AMD Display driver, the worst-case execution time for the Preprocessor was 8 minutes and 56 seconds. On the IIO subsystem, the worst-case time over 5 runs was 9 minutes and 41 seconds. In contrast, the Query Responder processed requests in under 2 seconds in both cases, demonstrating the efficiency of the two-stage architecture.

B. Ethnographic Study

Using the ArKanjo tool, the main author and 23 students (divided into 11 groups) identified and refactored duplicated code in the AMD Display driver and the Industrial I/O (IIO) subsystems, submitting their work as patches to the community maintainers. This effort resulted in 12 patch submissions: One was from the main author to the AMD Display driver, and 11 were from the student groups, with 10 targeting the IIO subsystem and one aimed at the AMD Display driver. After a lengthy review process, the main author's patch was successfully accepted and merged into the kernel.

The students' efforts demonstrated that newcomers could use the tool to contribute effectively. Of the 11 students' patches, 6 were accepted, 4 were rejected, and 1 required follow-up fixes based on maintainer feedback. To remove the duplicated code, 7 student groups used the *Parameterize Method*, and 2 used the *Extract Method* [10], both of which are straightforward refactoring strategies. These results support that ArKanjo identifies actionable duplications and that newcomers can successfully contribute patches to the kernel.

While this research includes successfully merged patches by the main author and student groups, a significant portion of the students' contributions encountered notable challenges. These difficulties often stemmed from maintainer feedback indicating that, although the proposed changes reduced duplication, they were rejected or required substantial revision due to concerns about code readability, increased abstraction, or the specific context of the code. Additional challenges included technical complexities, such as C macros in configuration files and extended patch review timelines. These findings highlight that while ArKanjo effectively identifies code duplications, a successful contribution also requires navigating trade-offs between deduplication and other factors, such as contextual appropriateness and maintainer expectations.

V. CONCLUSION

This paper presented ArKanjo, a novel tool designed to detect effectively and facilitate the refactoring of function-level code duplication within the Linux kernel. Evaluations demonstrated that ArKanjo is highly effective at identifying Type-1 and Type-2 clones, which are often indicative of propagated issues such as copied buggy logic. We validated the practical value of the tool through an ethnographic study in which the main author and student groups used ArKanjo to identify actionable duplications. This effort led to 13 patch submissions, including 1 from the main author and 6 from student groups that were successfully merged into the kernel [2], collectively removing 987 lines of duplicated code. While some refactoring efforts were declined due to maintainer priorities, ArKanjo successfully lowered the barrier for new contributors to make meaningful improvements. The tool provides immediate value for kernel maintenance and serves as a model for clone detection in other large-scale projects.

ACKNOWLEDGMENT

This study was financed, in part, by CAPES (Finance Code 001), the University of São Paulo – USP (Proc. 22.1.9345.1.2) and the São Paulo Research Foundation – FAPESP with the São Paulo State Data Analysis System Foundation – SEADE (Proc. 2023/18026-8), Brazil.

[2]Reference for the accepted patches: *github.com/arkanjo-tool/arkanjo/blob/main/PATCHES.md*.

REFERENCES

[1] W. Hordijk, M. L. Ponisio, and R. Wieringa, "Harmfulness of code duplication: a structured review of the evidence," p. 88–97, 2009.

[2] 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

[3] M. Schmitt, "Linux kernel device driver testing." São Paulo : Instituto de Matemática e Estatística, Universidade de São Paulo, 2022.

[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] R. Řehůřek, P. Sojka *et al.*, "Gensim—statistical semantics in python," *Retrieved from genism. org*, 2011.

[8] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 476–480.

[9] J. Krinke and C. Ragkhitwetsagul, "Bigclonebench considered harmful for machine learning," in *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, 2022, pp. 1–7.

[10] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.