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

Luan Ícaro Pinto Arcanjo
Universidade de São Paulo
São Paulo, SP Brazil 05508-220
Email: luanicaro@usp.br

Paulo Meirelles
Universidade de São Paulo
São Paulo, SP Brazil 05508-220
Email: paulormm@ime.usp.br

David Tadokoro
Universidade de São Paulo
São Paulo, SP Brazil 05508-220
Email: davidbtadokoro@usp.br

Abstract—The Linux kernel's massive scale (+28 M LoC, +20 K contributors) presents unique maintenance challenges. Surprisingly, Code duplication remains a persistent issue in the kernel's codebase, which could hinder its evolution and patching. Academic approaches often focus on pairwise comparison of code artifacts, not directly applied for comprehensive codebase analyses. Other existing free software 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's 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) optimization for C codebases with kernel-specific patterns; (2) preprocessing that enables rapid queries without redundant analysis; and (3) 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/Libre/Open Source Software (FLOSS) project, critical to a significant portion of the world's digital infrastructure. Maintaining the kernel is an enormous undertaking, involving more than 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 the kernel's 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.

The detection of code duplication, or code clones, has been a subject of research for decades (Jankowitz, 1988). 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) [4]. Various detection methodologies have been developed, including textual, token-based, tree-based, and graph-based approaches [4]. These have culminated in state-of-the-art techniques, such as the graph-based work by Liu et al [5]. However, the primary focus of such academic work remains on determining if a given pair of code artifacts are duplicates, 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, such as relying 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. To test the tool, we employ multiple analytical methods and ethnographic studies.

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 the url: https://github.com/LipArcanjo/arkanjo

A. Architecture

The tool operates in two main parts: the Preprocessor and the Query Responder. The Preprocessor is responsible for performing heavy computations to find the code duplications across a codebase and produce artifacts to consult information related to the duplications in a structured form. The Query Responder consumes the artifacts produced by the Preprocessor to execute the tool functionalities as requested. This choice enables the

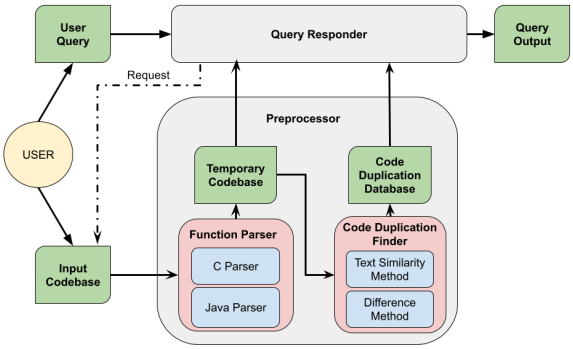

Fig. 1. Architecture diagram demonstrating the relationship between the tool components

tool to concentrate on the heavy and slow parts that can be executed only once per codebase, thus allowing a fast performance for multiple queries related to a single codebase.

The Preprocessor contains two main components, the Function Breaker and the Code Duplication Finder. The flow of how the Preprocessor works is as follow: Function Parser receives the codebase the user is interested in, extracts the functions of the codebase along with metadata, and creates a new temporary codebase where the functions extracted become new code files. The Code Duplication Finder iterates over every pair of files in the Temporary Codebase, checks if they are code clones, and, if so, saves them in the Code Duplication Database, which is a text file that stores every code duplication as a triple <function1,function2,similarity>, where function1 and function2 are the functions that are duplicates of each other, and similarity is the metric given by the code duplication detection method utilized in the Code Duplication Finder. Figure 1 illustrates the tool architecture.

The Query Responder consumes the Temporary Codebase and the Code Duplication Database to extract duplicated functions-related information per user request. If the user executes the Query Responder without executing the Preprocessor, the Query Responder calls the Preprocessor to create the required artifacts.

The Function Parser receives the input codebase and transforms it into the temporary codebase. The Function Parser iterates through every source code file from the programming language it supports and uses a specific programming language function extractor to extract every function in the file. For each function extracted, 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 file of the function, and the proper function. The first new source code file contains the function's body from the function it represents, while the other new source code file contains the function's signature. The metadata file contains additional relevant information about the function, such as the function name, the line where the function signature starts in the source code file, and the line where the function's body ends in the source code file. The programming languages supported at the moment are C and Java, although Java has limited support.

The Code Duplication Finder iterates through every pair of source code files in the temporary codebase, representing functions from the input codebase. For each pair of files, we execute a code duplication detection method that computes a metric measuring how similar the pair of files is, which we refer to as similarity. If the similarity is greater than or equal to the minimum similarity threshold (a parameter provided by the user during the Preprocessor), we store this pair of functions along with its similarity in the Code Duplication Database. We implemented two methods for code duplication detection in the tool, which the user can choose to use. The first method is based on text similarity, and the second is simpler and based on the number of equal lines. The experiments and tests in this research were done using the text similarity method.

1) Code Duplication Methods Used: For the text similarity method, we treat the source code files as text and apply the TF-IDF vector embedding method implemented by the Gensim library [6], then compute cosine similarity as the similarity metric. This method was chosen for its claimed performance, programming language independence, and the fact that it does not require compilable code, which is expected in the temporary codebase, as it does not contain complete code artifacts.

For the difference method, we implemented a method that considered 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 between the two functions by the total number of lines in both functions. This code duplication method is considerably simpler and more explainable than the text similarity method, as the metric is the ratio of common lines between the two functions. To compute the number of equal lines, we use the diff command built-in in the Linux environment.

B. Functionalities

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

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 facilitate more complex queries. Figure 2 demonstrates an example of this functionality in use.

The Function Information functionality provides detailed information about a specific function. The functionality receives a target function from the user and returns information such as the relative path, function name, and line numbers where the function is defined. Additionally,

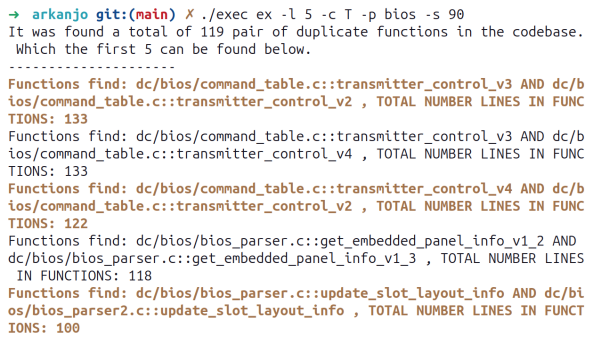

Fig. 2. Example of the Duplication Explorer functionality of the ArKanjo tool.

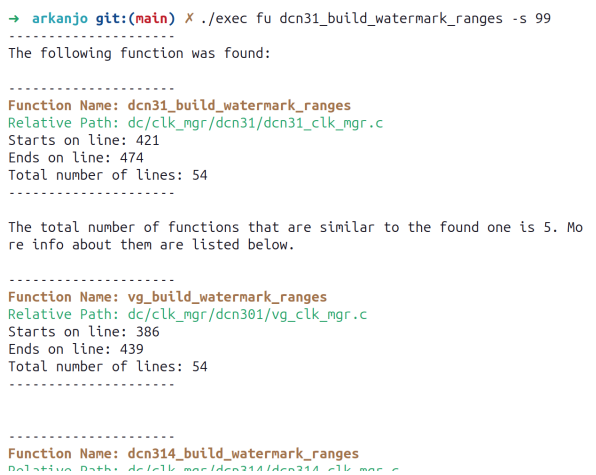

Fig. 3. Example of the Function Information functionality of the ArKanjo tool.

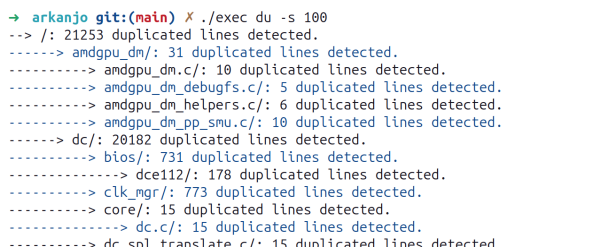

Fig. 4. Example of the Duplication Report functionality of the ArKanjo tool.

it provides similar information for every function that duplicates the given function. Figure 3 demonstrates an example of this functionality in use.

The Duplication Report functionality provides an overview of duplicated code within the input codebase. This functionality calculates the number of duplicated lines per folder in the codebase and presents the information to the user in a readable format. Figure 4 demonstrates an example of this functionality in use.

III. Methods

To validate the ArKanjo tool capabilities to find code duplications within the text similarity detection method used, we applied two independent methods. The first method evaluates the proposed tool using an approach from the literature, comparing it against the BigCloneBench dataset (Svajlenko et al., 2014). The second method is 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 [7], we followed the methodology presentedy by Liu et al. [5], we sampled 20,000 pairs from each clone type, adding 20,000 non-duplicate pairs as negative samples. We applied the same sampling approach to our tool to ensure a fair evaluation. Unlike state-of-art methods, our tool does not distinguish between clone types and identifies duplications at the function level rather than the file level, which is typical in state-of-the-art tools. 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 counts 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.

To assess if the duplications found by the ArKanjo tool were actionable, the research conducted a multi-part ethnographic study. In this study, the author and 12 student groups from the Free Software Development course at University of São Paulo acted as new contributors to the Linux kernel. They used the tool to identify duplications within the Industrial I/O (IIO) subsytem and AMD Display driver subsystems, refactored the code to address the issues, and submitted their proposed fixes as patches to the community maintainers. Throughout this process, the students documented their refactoring approaches and their experiences interacting with the kernel community.

IV. Results

A. Tool Evaluation

Table I shows the results obtained by our tool on the BigCLoneBench dataset and table II shows the results obtained in the empricial analysis. The BigCloneBench results and our empirical methods indicate that our tool performs well in detecting Type-1 and Type-2 code clone

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 I
Recall of the ArKanjo tool in the BigCloneBench dataset varying minimum similarity threshould.

Similarity range	T1	T2	T3	T4	False	Success Percentage
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%

TABLE II
Results of the ArKanjo tool in the empirical analyses in the AMD Display driver

duplications and can thus reveal propagated issues like copied buggy logic, but it struggles with more complex types. A similarity threshold between 80% and 100% yields favorable results in both methods.

However, there is a discrepancy in the detection of non-duplication pairs between the two methods. In the BigCloneBench evaluation, we found no false positives (negative samples inferred as duplications), while the empirical analysis revealed a higher percentage of false positives. We propose two potential explanations for this discrepancy.

The first reason may be the limitations and known issues with BigCloneBench, as highlighted by Krinke and Ragkhitwetsagul [8]. The second reason could relate to the nature of the AMD Display driver, where each code artifact shares semantic meaning. In contrast, BigCloneBench comprises self-contained code artifacts without shared semantics. Given that 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.

B. Ethnographic Study

Using the ArKanjo tool, the author and 12 student 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 13 total patch submissions: one from the author to the AMD Display driver and 12 from the student groups, with 11 targeting the IIO subsystem and one for the AMD Display driver. The author's patch, which removed over 500 lines of duplicated code, was successfully accepted and merged into the kernel after a lengthy review process.

The student efforts showed that newcomers could use the tool to make effective contributions. Of the 12 student patches, five were accepted, four were refused, and three required follow-up fixes based on maintainer feedback. To remove the duplicated code, seven student groups used the Parameterize Method and two used the Extract Method [9], which are simple refactoring methods. This result corroborates that ArKanjo finds actionable duplications and that newcomers can successfully contribute patches to the kernel.

While this research included a successfully merged patch by the author and students, a notable portion of student efforts encountered significant hurdles. These difficulties often stemmed from maintainer feedback where proposed changes, though reducing duplication, were rejected or required substantial rework due to concerns about code readability, added abstraction, or the specific context of the code. Technical complexities, such as C macros in configuration files, and lengthy patch review times also contributed to the challenges. This highlights that while ArKanjo effectively identifies duplications, a successful contribution involves a trade-off between code deduplication and other factors like the specific code's context and maintainer priorities.

V. Conclusion

In conclusion, this paper presented ArKanjo, a novel tool designed to effectively detect 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 like copied buggy logic. The practical value of the tool was confirmed through an ethnographic study where the author and student groups used ArKanjo to identify actionable duplications. This effort led to 13 patch submissions, including one from the author that removed over 500 lines of code and was merged into the kernel, alongside three successful student contributions. While some refactoring efforts were declined due to maintainer priorities, ArKanjo successfully lowers 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.

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] 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.
[5] 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.
[6] R. Řehůřek, P. Sojka et al., "Gensim—statistical semantics in python," Retrieved from genism. org, 2011.
[7] 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.
[8] 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.
[9] M. Fowler, Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.