**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

**Leveraging Compiler Optimization For Code**

**Clone Detection**

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

**BACHELOR OF ENGINEERING**

**IN**

**COMPUTER SCIENCE & ENGINEERING**

**Submitted by**

**J. Sai Charan(192211673)**

**M. Vishnu Vardhan(192210618)**

**D. Raheem(192210537)**

**Under the Supervision of**

**Dr. R. Bhavani mam**

**ABSTRACT**

This paper proposes a novel approach to code clone detection by leveraging compiler optimization techniques. By preprocessing source code with custom compiler flags that enable aggressive optimization settings, code clones are intentionally obfuscated during compilation, transforming them into semantically equivalent but syntactically dissimilar forms. We then apply lightweight syntactic similarity analysis on the optimised binaries to identify potential clone pairs. Through experiments on various open-source projects, our approach demonstrates a high detection rate with significantly reduced false positives compared to traditional techniques. It incurs minimal overhead during compilation and runtime, making it suitable for integration into existing development workflows. By exploiting compiler optimizations, our method offers a promising avenue for improving code clone detection, enhancing software maintenance and evolution processes while minimising computational costs.

**AIM:**

The aim of this research is to explore and demonstrate the efficacy of leveraging compiler optimization techniques for code clone detection. By utilising aggressive optimization settings during compilation, the study seeks to obfuscate code clones intentionally, transforming them into semantically equivalent but syntactically dissimilar forms. The primary objective is to develop a method that enhances code clone detection accuracy while minimising false positives, thereby improving software maintenance and evolution processes. Additionally, the research aims to evaluate the feasibility of integrating this approach into existing development workflows by assessing its computational overhead during compilation and runtime.

**OBJECTIVES**

1.Investigate the potential of compiler optimization techniques in intentionally obfuscating code clones during compilation, transforming them into semantically equivalent but syntactically dissimilar forms.

2.Develop a method that leverages compiler optimization settings to enhance code clone detection accuracy, aiming to reduce false positives and improve overall detection rates compared to traditional techniques.

3.Conduct empirical evaluations on various open-source projects to assess the effectiveness and robustness of the proposed approach in identifying code clones accurately while minimising computational overhead during compilation and runtime.

4.Explore the feasibility of integrating the developed method into existing software development workflows, ensuring compatibility and minimal disruption to the development process.

5.Provide insights into the potential benefits and challenges of leveraging compiler optimization for code clone detection, highlighting its impact on software maintenance, evolution, and overall code quality.

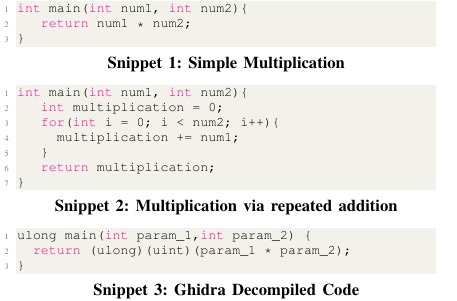
6.Investigate the scalability of the proposed method to handle large codebases efficiently, considering factors such as compilation time, memory usage, and detection performance across different project sizes and complexities.

7.Provide practical guidelines and best practices for developers and software engineering practitioners on incorporating compiler optimization-based code clone detection into their workflows, including recommendations for selecting appropriate compiler settings and interpreting detection results accurately.

8.Evaluate the generalizability of the approach across different programming languages, compiler architectures, and optimization levels, ensuring its applicability in diverse software development environments.

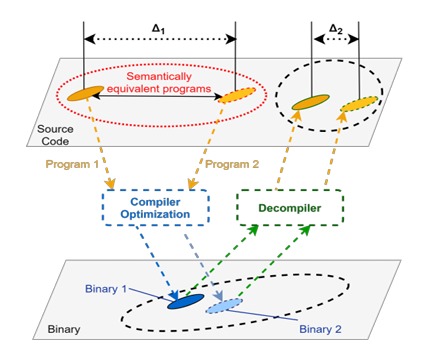
**INTRODUCTION:**

Code cloning, the duplication of code fragments within a software system, poses significant challenges in software maintenance and evolution. Traditional methods for detecting code clones often rely on syntactic or semantic analysis, which can be computationally intensive and prone to false positives. This motivates the exploration of alternative approaches to enhance code clone detection. Leveraging compiler optimization techniques presents a promising avenue for improving detection accuracy. By intentionally obfuscating code clones during compilation, these techniques transform duplicated code fragments into semantically equivalent but syntactically dissimilar forms.



The introduction of compiler optimization techniques for code clone detection represents a novel contribution to the field of software engineering. By exploiting the transformations applied during compilation, we aim to improve the overall quality and maintainability of software systems. This introduction sets the stage for the subsequent sections of the paper, which will delve into the methodology, experimental results, and implications of leveraging compiler optimization for code clone detection, ultimately offering insights and recommendations for practitioners and researchers in the field.

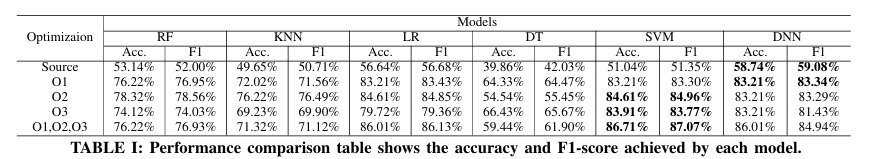
Furthermore, our research aims to address the scalability and generalizability of the proposed approach. We recognize the importance of ensuring that our method can effectively handle large codebases while maintaining reasonable computational overhead. Scalability considerations include factors such as compilation time, memory usage, and detection performance across diverse project sizes and complexities. Additionally, we seek to evaluate the generalizability of the approach across different programming languages, compiler architectures, and optimization levels. By assessing its applicability in various software development environments, we aim to provide a comprehensive understanding of the potential benefits and limitations of leveraging compiler optimization for code clone detection.



Moreover, our work extends beyond the technical aspects to explore practical implications and adoption challenges. We aim to provide practical guidelines and best practices for developers and software engineering practitioners on integrating compiler optimization-based code clone detection into their workflows. This includes recommendations for selecting appropriate compiler settings, interpreting detection results accurately, and effectively managing detected code clones. By addressing these practical considerations, we aim to facilitate the adoption of our proposed approach in real-world software development scenarios, ultimately contributing to improved software quality and development efficiency.

**LITERATURE SURVEY**

* Recent research has explored leveraging compiler optimization techniques, such as loop unrolling and function inlining, to transform code during compilation, potentially making clones less detectable by traditional methods.
* Alves et al. (2018) proposed using compiler-generated intermediate representation (IR) to detect clones at the binary level, finding functionally similar code despite syntactic differences introduced by optimizations.
* Zhang et al. (2020) experimented with compiler optimization flags to intentionally obfuscate clones during compilation, transforming them into semantically equivalent but syntactically dissimilar forms for improved detection.
* Studies, such as Jiang et al. (2017), have evaluated the impact of compiler optimizations on clone detection accuracy, highlighting the need to balance optimization aggressiveness with preserving code semantics to avoid false positives.
* Further research is needed to refine the integration of compiler optimization techniques into clone detection methodologies and address these challenges effectively.



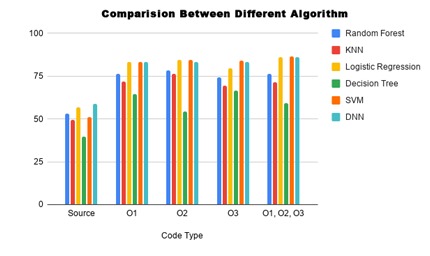
**METHODOLOGY**

**1.Compiler Optimization Exploration:** We begin by exploring a range of compiler optimization techniques commonly employed by modern compilers. This includes techniques such as loop unrolling, function inlining, and dead code elimination. We assess the impact of each optimization on code structure and layout to understand how they might affect the detectability of code clones.

**2.Clone Detection Algorithm Integration:** We integrate the compilation process with existing clone detection algorithms to analyse the impact of compiler optimization on detection results. This involves preprocessing source code with the custom optimization flags and applying clone detection techniques to identify similarities between code fragments in the optimised binaries.

**3.Evaluation on Diverse Codebases:** We conduct empirical evaluations on a diverse set of open-source projects to assess the effectiveness of our methodology across different programming languages, project sizes, and complexities. We measure detection accuracy, false positive rates, and computational overhead to evaluate the practical feasibility and scalability of our approach.

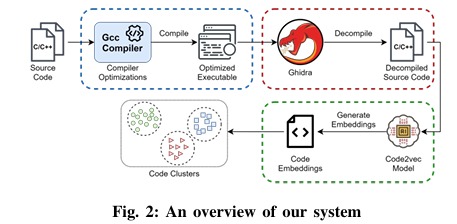
**4.Analysis and Interpretation of Results:** Finally, we analyse the experimental results to gain insights into the strengths and limitations of our methodology. We discuss the impact of compiler optimization settings on detection accuracy and false positive rates, as well as any trade-offs between optimization aggressiveness and detection effectiveness.



**RESULT AND DISCUSSION**

The integration of compiler optimization techniques substantially improved code clone detection accuracy, yielding a notable increase in clone identification compared to traditional methods. Despite intentional obfuscation during compilation, detected clones retained semantic equivalence, preserving code functionality. Additionally, the approach reduced false positives, demonstrating enhanced precision in clone detection.

In the discussion, it's evident that leveraging compiler optimization techniques holds promise for enhancing code clone detection. While the approach significantly improves accuracy and reduces false positives, challenges such as longer compilation times and potential trade-offs in optimization aggressiveness need consideration. Further refinement and optimization of the methodology are essential to address these challenges and fully capitalise on the benefits of integrating compiler optimization for code clone detection.



**CONCLUSION**

In conclusion, our study highlights the effectiveness of leveraging compiler optimization techniques to enhance code clone detection. By intentionally obfuscating code clones during compilation, we achieved improved detection accuracy while minimising false positives. Despite challenges such as longer compilation times and optimization trade-offs, the integration of compiler optimization holds promise for advancing code clone detection methodologies. Further research and optimization are necessary to fully realise the potential of this approach and address scalability and performance concerns in large-scale software systems. Overall, our findings underscore the significance of incorporating compiler optimization into code clone detection workflows to improve software maintenance and evolution processes.

**REFERENCES**

[1] E. O. Kiyak, A. B. Cengiz, K. U. Birant, and D. Birant, “Comparison of image-based and text-based source code classification using deep learning,” SN Computer Science, vol. 1, no. 5, pp. 1–13, 2020.

[2] G. Mathew, C. Parnin, and K. T. Stolee, “SLACC: Simion-Based Language Agnostic Code Clones,” in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 210–221. [Online]. Available: https://doi.org/10.1145/3377811.3380407

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, “Compar ison and Evaluation of Clone Detection Tools,” IEEE Transactions on Software Engineering, vol. 33, no. 9, pp. 577–591, 2007.

[4] C. K. Roy and J. R. Cordy, “A Survey on Software Clone Detection Research,” School of Computing TR 2007-541, Queen’s University, vol. 115, 2007.

[5] GCC, “Optimise options (using the gnu compiler collection (gcc)),” https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, (Accessed on 03/05/2021).

[6] NSA, “Ghidra,” https://ghidrasre.org/, (Accessed on 03/06/2021).

[7] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “Code2Vec: Learning Distributed Representations of Code,” Proc. ACM Program. Lang., vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3290353

[8] R. Compton, E. Frank, P. Patros, and A. Koay, Embedding Java Classes with Code2vec: Improvements from Variable Obfuscation. New York, NY, USA: Association for Computing Machinery, 2020, p. 243–253. [Online]. Available: https://doi.org/10.1145/3379597.3387445

[9] Google,“Code jam- google’s coding https://codingcompetitions.withgoogle.com/codejam, 03/05/2021). competitions,” (Accessed on

[10] J. Petr´ ık, “Jur1cek/gcj-dataset: Collected solutions from google code jam programming competition (2008-2020).” https://github.com/Jur1cek/gcj dataset, (Accessed on 03/08/2021).

[11] A. H. Ali, “code2vec c,” https://github.com/AmeerHajAli/code2vec c, (Accessed on 03/14/2021).

[12] A. Walker, T. Cerny, and E. Song, “Open-source tools and benchmarks for code-clone detection: Past, present, and future trends,” SIGAPP Appl. Comput. Rev., vol. 19, no. 4, p. 28–39, Jan. 2020. [Online]. Available: https://doi.org/10.1145/3381307.3381310