Compiler Optimization for Code Clone Detection

Savani Yash
Institute of Technology. Nirma University
Ahmedabad, India
19bce243@nirmauni.ac.in

Sakariya Gaurav
Institute of Technology. Nirma University
Ahmedabad, India
19bce233@nirmauni.ac.in

ABSTRACT

Goal: The key observation of our work is that the compiler optimizations can be used to smooth out source code level idiosyncrasies introduced by the developers, thus making the optimized programs, for the same task, similar in structure.

Motivation: We conducted experiments on the Google Code Jam dataset to demonstrate the effectiveness of our approach. The experimental results show that our technique can achieve up to 85% accuracy on the program classification task, which is an improvement of more than 25% over the source code level classification.

Challenges: In the paper, we propose a novel approach that leverages compiler optimizations to transform semantically similar code and detect similar programs. The similarity in structure can then be used to classify the programs. Finding similar code in software systems can guide several software engineering tasks such as code maintenance, program understanding, and code reuse.

Index term: code clones, compiler optimization, reverse engineering, code representation.

INTRODUCTION

The detection of code clones is a significant issue for the upkeep and development of software. Many methods have been investigated for clone detection, and they can be divided into two main categories: a) Static analysis: information extraction from the content of the code [1] and b) dynamic analysis: run-time program behavior based on clone detection. [2]. Code reuse, program comprehension, malware detection, and code maintenance are just a few of the many uses for code clone detection. In this study, we use compiler optimizations to categorized programs that are semantically similar. Compiler optimization is a series of adjustments made to a program by the compiler to create a binary that is semantically equivalent but consumes less system resources when executed. The underlying premise of our study is that compiler optimizations can be used to eliminate any quirks developers may have introduced at the source code level, resulting in structurally similar optimized code for the same task. The programs can then be categorized using this similarity.

- **Contributions**

In this paper, we answer three research questions:

• RQ1: Can compiler optimization be used to smooth out code level differences introduced by the developer?
• RQ2: Can the compiler optimized code be used to detect similarity? If yes, then which optimizations are optimal?
• RQ3: Can cross-optimization detect similar code?
The primary contributions of this paper are three-fold:

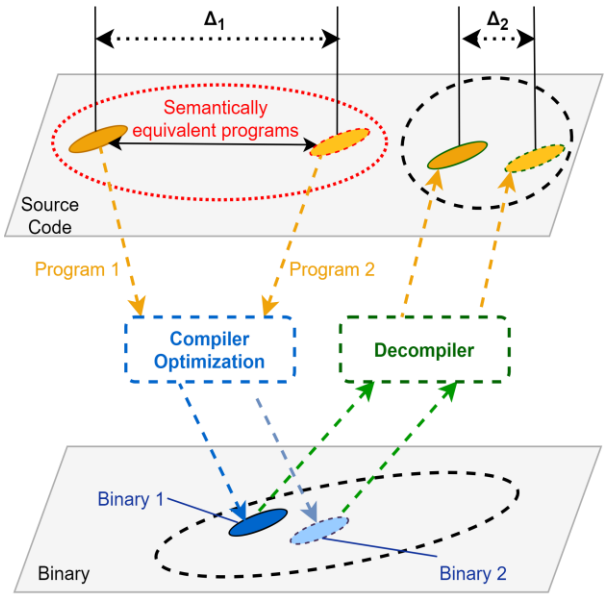


Fig. 1 Represents the distance between the vector representations of two semantically similar source code. 2 Represents the distance between the vector representations of the decompiled binaries of the source code.

• We present a novel technique for code clone based on compiler optimizations. Our approach can also be adapted to detect similar binaries without source code availability.
• We study the impact of different compiler optimization levels on code clone detection. We also perform experiments to investigate the impact of cross-optimization on clone detection.

To test the viability of our suggested strategy, we ran experiments on the 2008 Google Code Jam dataset. According to the experimental findings, our method can complete the classification task with an accuracy rate of up to 85%. We also research how well cross-compiler optimizations perform on the classification task. To the best of our knowledge, this is the first study investigating compiler-optimized decompiled code for tasks involving source-level code clone detection. Our research provides a general framework that can be used to address various problems, including malware detection and plagiarism detection, among others.

The remaining portions of the essay are structured as follows: The history of code clones, compiler optimizations, and code embedding is covered in Section II. The data set used in our study is discussed in Section III. In Section IV, the suggested framework is described. The findings of our study are presented in Section V, which is followed by a section on related work and a discussion. Section VIII talks about the constraints on our research. Section IX concludes by discussing the results and the next steps.

BACKGROUND

In this section, we discuss code clones, compiler optimizations, Ghidra, and code representation through code2vec.

1) **Code Clones**: Code clones are similar pieces/fragments of code that are either syntactically or behaviorally similar. In practice, programmers often use clones via copy/paste to support rapid software development. For a given code snippet, there can be several types of clones. Four types of code clones have been widely studied in literature [3], [4]: Type-1 (textual similarity), Type-2 (lexical, or token-based, similarity), Type-3 (syntactic similarity), and Type-4 (semantic similarity).

2) GCC Compiler Optimizations: GNU Compiler Collection (GCC) is the GNU compiler project which supports several high-level languages, such as C and C++. One core function of the compiler is to optimize the code for performance. Code optimization has several benefits; it allows reduced resource consumption, resulting in faster running machine code and lesser memory usage. The optimization is performed by doing transformations (optimizations) that can only be done at the assembly (machine) level for the target hardware. The GCC optimizer supports six pre-defined optimization levels: -O1, -O2, -O3, -Ofast, -Og, and -Os [5]. In this work, we utilize -O1, -O2, -O3 optimization levels.

3) Ghidra: Ghidra is a free, open-source reverse engineering tool developed by National Security Agency (NSA) [6]. It is a comprehensive and expandable framework covering the complete workflow of binary analysis. Ghidra is often used for the decomplication of executable binaries, and in this study, we use Ghidra's command-line analysis tool to reverse- engineer the decompiled code of compiler optimized C/C++ code binaries.

4) Code2vec: Code2vec [7] is a neural network architecture based on attention architecture for representing snippets of code as continuous distributed vectors or code embeddings. Originally trained on Java, Code2vec converts the source code into a set of paths using the code's underlying Abstract Syntax Tree (AST) and learns how to combine these paths using an attention mechanism. Code2vec then represents each function as a fixed-length code vector which is used to represent the different features of that function. Method embeddings generated by code2vec serve as a base for a large variety of applications and analyses such as author attribution, bug detection, and so on. It has been shown that the generated embeddings can be aggregated using several aggregation methods such as max, min, sum, mean, median, and standard deviation to obtain embeddings at a file-level [8]. We utilize median aggregation method to represent each program.

DATASET

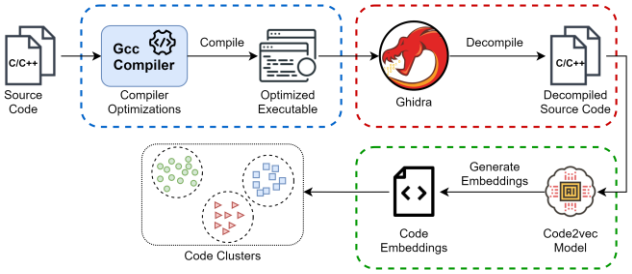
Code Embedding Dataset: Since the original code2vec model is trained on Java language, we trained a new code2vec model on C and C++ programs from top 1000 GitHub repositories. Because of memory limitations, we excluded the Linux repository.

Experiment Dataset: Google Code Jam (GCJ) is a yearly programming competition hosted by Google. In our study we used GCJ dataset from 2008 [9] provided on Github [10]. The competition has several rounds, each containing several problems to be solved by the participants worldwide. The diverse characteristics of the participant pool introduces diversity in the submissions for any given programming task. Participants are allowed to submit their programs in any language of their choice. In this study, however, we only consider C and C++ programs because of compiler restrictions.

The GCJ dataset can be further sub-divided into two types of programs: accepted solutions and non-accepted submissions. In this study, because of ground truth availability, we only use the accepted solutions. In our study, the submissions from 2008 GCJ were used to extract the code embedding, train, and test the classifiers. The 2008 data contained 8,524 solutions written in C/C++ with disproportionate distribution across different problems. For consistency, we consider six programming tasks with about 200 randomly sampled submissions. The total size of the dataset was 1,423. We further split the data into training and test set, containing 1,280 and 143 submissions, respectively. The programs were then compiled with three different optimization levels (see section II-4) and then decompiled using Ghidra [6] (see section II-3).

SYSTEM OVERVIEW

Four processes make up our method: code compilation employing compiler optimizations, code recompilation, generation of code embedding, and classification of the embedding into clusters of related code. The high-level overview of our pipeline is shown in Figure 2. To create an executable binary from a C/C++ program, we first compile the binary with one of the optimization flags. The source code is then extracted from the binary using Ghidra. The code2vec model is then used to extract the program's code embedding from the produced source code. A model used to categorize the programs is trained using the embedding. We go into further depth about each step in this section.



Overview of System

A. Compiling Binaries

We use the GCC compiler to generate the GSJ dataset's source programs' binaries in the first step. For every program, we generate three binaries corresponding to three optimization levels: O1, O2, and O3. These binaries are then decompiled using Ghidra reverse-engineering tool to get the source code. All binaries were compiled for x64 architecture.

B. Ghidra for decomplication

Our study uses Ghidra's command-line analysis tool to reverse-engineer the decompiled version of compiler optimized C/C++ code binaries. The study uses command-line analysis, also known as headless-analysis, since work requires several files to decompile at once, so it is feasible to use command-line analysis. We first import all binaries and then perform analysis to decompile them. The decompiled code is saved in separate files to generate code vector representations.

C. Code Embedding

Path extraction is crucial for obtaining code vectors from code2vec. Code2vec creates the AST first (Abstract Syntax Tree). The route-context is created by extracting the syntactic path between AST leaves. A path context's individual paths and leaf values are mapped to their corresponding real-valued vector representations, or embeddings. A single vector that represents that path-context is created by concatenating the three vectors of each context.

Notably, code2vec created code embedding for methods rather than for programs or files. We had to create a single code embedding for a single file that may include numerous functions since we wanted to investigate program level similarities rather than function level similarity. By combining the set of

method level embedding, as demonstrated in earlier study [8], we may obtain the file level embedding. Applying the aggregate technique column-wise. Max, Min, Mean, and Median are the basic aggregating functions that are employed. A mix of aggregation techniques may also be taken into account. In our investigation, median aggregation performed best, thus we used it to build the program level embedding.

1) **Model Training:** Since the base code2vec model is trained on Java language, we trained a new code2vec model on C and C++ programs from top 1000 Github repositories. Code2vec model generation is a two-step process: pre- processing and model training. For pre-processing step, we used the pre-processing script provided by code2vec c [11]. We set the maximum leaf node to be processed in the given method to 320. We pre-processed all C programs in the repositories; however, some files that did not match the maximum leaf node size training criteria were removed. The remaining 1.2 million programs were then used to train the code2vec model. The model was trained for 40 epochs.

2) **Feature Vector Extraction:** For getting the code embed- ding for the classification task, we used the newly trained code2vec model. We captured code-vector corresponding to each function in the program. Since we wanted to study program level similarity rather than function level similarity, we had to generate a single code embedding for a single file that might contain multiple functions. To generate one vector to represent a given program, we used median aggregation function following prior research [8], which showed that we can get the file/program level embedding by aggregating the set of method level embedding. These program level embed- ding were then used to train and test the classification models.

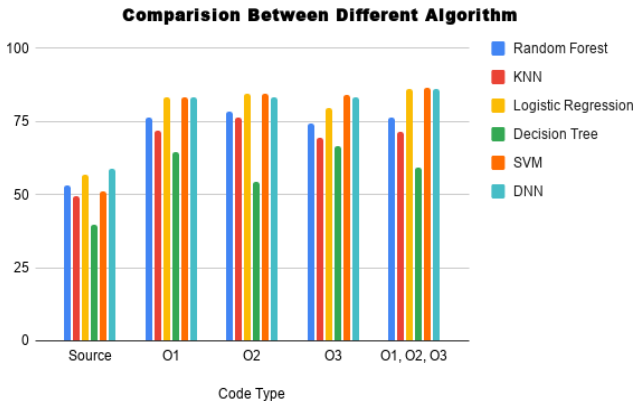
RESULTS

A. Experimental Setting

Five commercial machine learning techniques and one homemade DNN are used to train our models and show the viability of the suggested work. Five pre- built machine learning algorithms were trained categorization tasks: K-Nearest Neighbor (KNN), Random Forest (RF), and Support Decision Tree, Logistic Regression, and Vector Machines (SVM) (DT). A DNN was also trained. The DNN is made up of a SoftMax output layer, a hidden layer with 384 neurons with ReLu activation, and an input layer with 384 neurons matching to the 384 code2vec features (6 classes corresponding to 6 programming tasks).

B. Code Classification

The 2008 Google Code Jam dataset had six programming challenges, each with roughly 200 data points. The dataset, which included a total of 1,423 programs, was divided into a training set and a test set, each of which had 1,280 and 143 programs, respectively. Then, we retrieved the code vectors associated with each program and combined the vectors to obtain a representation of each program. The procedure was repeated for decompiled code with the O1, O2, and O3 degrees of optimization. In the end, we created four embedding datasets—original programs, O1 optimized, O2 optimized, and O3 optimized programs—for the classification job. We used all of the optimized program embeddings (O1, O2, and O3) in addition to the four datasets to evaluate the performance of the models.



We trained the models on the code vector representation of the programs. Since we had six different classes of programs in the dataset, we trained multi-class classifiers. Table I summarizes the results of the models. We observe the classification accuracy is highest in the models trained on O2 optimized programs. Furthermore, it can be seen that using the best models, the classifier can correctly classify up to 84.61% of the decompiled programs vs. only 58.74% of the source code. We also observe that DNN and SVM models perform similarly.

Optimization	Models											
	RF		KNN		LR		DT		SVM		DNN	
	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.	F1
Source	53.14%	52.00%	49.65%	50.71%	56.64%	56.68%	39.86%	42.03%	51.04%	51.35%	58.74%	59.08%
O1	76.22%	76.95%	72.02%	71.56%	83.21%	83.43%	64.33%	64.47%	83.21%	83.30%	83.21%	83.34%
O2	78.32%	78.56%	76.22%	76.49%	84.61%	84.85%	54.54%	55.45%	84.61%	84.96%	83.21%	83.29%
O3	74.12%	74.03%	69.23%	69.90%	79.72%	79.36%	66.43%	65.67%	83.91%	83.77%	83.21%	81.43%
O1,O2,O3	76.22%	76.93%	71.32%	71.12%	86.01%	86.13%	59.44%	61.90%	86.71%	87.07%	86.01%	84.94%

Performance comparison table shows the accuracy and F1-score achieved by each model

Figure summarizes the accuracy of all the models. We can observe that the accuracy is sub-optimal in the case of the developer written program (depicted as 'Source'); however, the accuracy significantly increases if we apply a compiler optimization. This increase in the accuracy is owed to the transformations performed by the compiler on the source pro- gram, which results in similar binaries being constructed from semantically similar programs written by different developers.

	Source	O1	O2	O3
Source	**51.04%**	11.88%	11.88%	11.88%
O1	22.37%	**83.21%**	65.03%	60.13%
O2	15.38%	72.72%	**84.61%**	68.53%
O3	18.88%	68.53%	74.82%	**83.91%**

Performance of cross optimizations (Accuracy)

We also ran a test to see how cross-compiler optimization affected the categorization task. On each dataset, we trained a single SVM model, and we tested the models using different datasets to assess their performance. Table II presents the experiment's findings in terms of accuracy. When evaluated on the same set of data that they were trained on, the models may be seen to perform best. This finding suggests that the compiler transformations produce somewhat different binaries at various levels of optimization such that the code embedding cannot detect the similarities.

The performance of the models trained on O1 and O3 optimized programs also exhibits a distinctive pattern. When compared to O3 dataset, the model's performance on the O2 dataset is superior for the model trained on O1 optimization. Similar to this, the model's performance on O2 over O1 is improved by O3 optimization training. The parallels between similar degrees of optimization are suggested by this pattern. The model displays the same pattern after being trained on the original source programs. Performance degrades when we depart from the original program, going from O1 to O2 to O3.

The t-SNE plots of the code embedding for each dataset are shown in Figure 4. It can be seen that it is more difficult to separate source code, but compiler-optimized decompiled program representation shows a large improvement and enables this. As we move from the O1 to the O2 level of optimization, the degree of separation rises. However, in O3, the separation slightly declines.

RELATED WORK

Code resemblance, especially type-4 (semantically identical) code clones, has been well researched in the literature [12]. Researchers discovered functionally related code in use through a user study [13]. While static token-based methods like SourcererCC[14] and CCFinder[15] have been studied, two additional methods for code clone detection have emerged as a result of technological advancements. We first describe dynamic approaches to machine learning, then approaches based on static features of the code.

A. ML for Code Clone Detection

Additionally, deep learning has been used to find code clones [16], [17], [18], and [19]. All four forms of code clones were found by researchers using either structure or identifiers [16]. The phrases in code fragments were mapped to vector representations so that terms used in similar ways translate to comparable vectors. This was a revolutionary code representation scheme that was the foundation of their approach. The model then picks up distinguishing properties for code fragments at various granularities. Another method, called Deep-six, quantifies the functional similarity of the code [17] by embedding the control flow and data flow graphs into a semantic matrix. By utilizing the structured, syntactic, and semantic information of the source code, HOLMES [18] (which depends on CFG and DFG) performs semantic code clone detection using program dependency graphs and graph neural networks.

B. Dynamic Analysis for Code Clone Detection

To find functionally equivalent code for freshly written methods without test cases, Tajima et al. [21] presented an approach. From techniques, authors first extract interface data and PDG. Following that, similarity detection is done using this information. By running the created test cases, Li et al. [22] developed a method based on automatic test case creation to find semantically similar API methods. If two procedures provide the same result for each of the created test cases, they are said to be comparable. SLACC is a cross-language clone detection method based on runtime behavior that was proposed by Mathew et al. [2]. It clusters code based on behavior using function I/O. To find similarities, authors produce 256 inputs for each function. Compared to dynamic techniques, our work is lightweight since we donot need to run the programs.

THREATS AND LIMITATIONS

Our strategy uses Code2vec embedding, which creates the vector representation using the program's ASTs. Because of the closeness in syntactic structure, code2vec generates very comparable ASTs for Type-I (textual similarity), Type-II (lexical similarity), and Type-III (syntactic similarity) clones. Type-IV code clones, however, differ in their syntactic structures and merely share behavioral similarities.

As a result, the Type-IV clones' underlying ASTs differ greatly, which affects their code2vec vector representation. Additionally, it has been demonstrated that code2vec significantly relies on variable names for prediction, making it vulnerable to adversarial assaults or mistakes [8].

The performance of our code2vec model may be enhanced by utilizing obfuscated training data because it was developed using the source code. Additionally, a number of methods for creating code embedding that make use of call graphs, ASTs, and other information from the code are documented in the literature [23].

Our approach in this study makes use of code2vec embedding. It's likely that other representations, such Asm2vec [20], produce more beneficial outcomes. We defer to further work on this.

CONCLUSION

In this research, we provide a strategy for detecting code clones based on compiler optimization. Our method depends on the compiler to eliminate the discrepancies in the source code that the developer inserted. We found that O2 optimization, when compared to O1, O2, and O3, produces the highest results for the classification job (84.61% accuracy and 84.96% F1-score). In comparison to the source code-based representation, our suggested technique produces an accuracy gain of more than 25%. We also looked into the effectiveness of cross compiler optimization for classification issues. Our findings imply that the optimizations produce noticeably distinct binary representations, making it challenging for the model to learn optimally. We intend to investigate program representation in the future to provide cross compiler optimization and enhance classification performance.

REFERENCES

[1] 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.

[2] 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.

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

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

[5] 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

[6] 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

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

[8] 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).

[9] 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
[10] V. Ka¨fer, S. Wagner, and R. Koschke, “Are there functionally similar code clones in practice?” in 2018 IEEE 12th International Workshop on Software Clones (IWSC), 2018, pp. 2–8.
[11] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, “Sourcer- ercc: Scaling code clone detection to big-code,” in 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 1157–1168.
[12] T. Kamiya, S. Kusumoto, and K. Inoue, “Ccfinder: a multilinguistic token-based code clone detection system for large scale source code,” IEEE Transactions on Software Engineering, vol. 28, no. 7, pp. 654–670, 2002.
[13] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 87–98.
[14] G. Zhao and J. Huang, “DeepSim: Deep Learning Code Functional Similarity,” in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 141–151. [Online]. Available: https://doi.org/10.1145/3236024.3236068
[15] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare, “Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks,” arXiv preprint arXiv:2011.11228, 2020.
[16] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, “FCCA: Hybrid Code Rep- resentation for Functional Clone Detection Using Attention Networks,” IEEE Transactions on Reliability, pp. 1–15, 2020.
[17] S. H. H. Ding, B. C. M. Fung, and P. Charland, “Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization,” in 2019 IEEE Symposium on Security and Privacy (SP), 2019, pp. 472–489.
[18] 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. 1–26