Leveraging Compiler Optimization for Code Clone Detection

Shirish Singh
Dept. of Computer Science
Columbia University, USA
shirish@cs.columbia.edu

Harshit Singhal
Dept. of Computer Science & Engineering
The LNMIIT, India
18ucc159@lnmiit.ac.in

Bharavi Mishra
Dept. of Computer Science & Engineering
The LNMIIT, India
bharavi@lnmiit.ac.in

***Abstract*—Finding similar code in software systems can guide several software engineering tasks such as code maintenance, program understanding, and code reuse. Similar code detection has been actively studied in the past. In the paper, we propose a novel approach that leverages compiler optimizations to transform semantically similar code and detect similar programs. 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. The similarity in structure can then be used to classify the programs. We conducted experiments on the Google CodeJam 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.**

***Index Terms*—code clones, compiler optimization, reverse-engineering, code representation**

I. Introduction

Code clone detection is an important problem for software maintenance and evolution. Several approaches have been studied for clone detection, which can be subdivided into two broad categories: a) Static analysis: extraction information from the code content [1] and b) Dynamic analysis: clone detection based run-time program behavior. [2]. Applications of code clone detection are manifold, such as code maintenance, program understanding, malware detection, and code reuse. In this work, we rely on compiler optimizations to classify the semantically similar programs. Compiler optimization is a sequence of transformations performed by the compiler on a program to produce a semantically equivalent binary that uses fewer system resources for its execution. Our study's main idea relies on the hypothesis that the compiler optimizations can be used to remove any source code level idiosyncrasies introduced by the developers, thus making the optimized code, for the same task, structurally similar. This similarity can then be used to classify the programs.

A. Motivating Example

Code snippet 1 and 2 represent the multiplication functionality using two different methods: simple multiplication and multiplication by addition. After compiling the snippets with O3 compiler optimization and then decompiling the binaries through Ghidra, we observe that the resultant code is the same, shown in code snippet 3.

```
int main(int num1, int num2){
   return num1 * num2;
}
```

Snippet 1: Simple Multiplication

```
int main(int num1, int num2){
   int multiplication = 0;
   for(int i = 0; i < num2; i++){
     multiplication += num1;
   }
   return multiplication;
}
```

Snippet 2: Multiplication via repeated addition

```
ulong main(int param_1,int param_2) {
  return (ulong)(uint)(param_1 * param_2);
}
```

Snippet 3: Ghidra Decompiled Code

As seen in the above code snippets, the main idea behind our study relies on the key observation that the compiler transforms the code for optimized performance, and two semantically similar codes can yield very similar (often overlapping) optimized binaries. Hence, we leverage these compiler optimizations to reduce the differences in the source code introduced by the developers. Figure 1 depicts the key idea of our work. Two semantically similar developer-written programs can have a large distance ($\triangle_1$) when represented as vectors. After compiling (with optimizations) and then decompiling, we observe that the effective distance between the decompiled programs ($\triangle_2$) reduces significantly to deem them similar. The net change ($\triangle_1 - \triangle_2$) in the distances between the original programs and the decompiled programs results in significant improvements in the clone detection task.

B. 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:

DOI reference number: 10.18293/SEKE2021-032

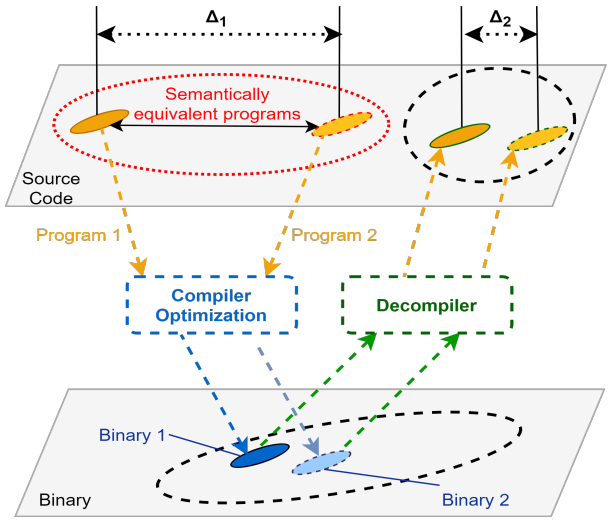


Fig. 1: $\triangle_1$ represents the distance between the vector representations of two semantically similar source code. $\triangle_2$ represents the distance between the vector representations of the decompiled binaries of the source code.

- 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.

We conducted experiments on the Google Code Jam dataset from 2008 to demonstrate the effectiveness of our proposed approach. The experimental results show that our technique can achieve accuracy up to 85% on the classification task. We also study the effectiveness of cross compiler optimizations on the classification task. To the best of our knowledge, this is the first work exploring compiler optimized decompiled code for code clone detection tasks at the source code level. Our work is a general framework that can be adapted to solve other challenges such as malware detection, plagiarism detection, etc.

The remainder of the paper is organized as follows: Section II explains the background on code clones, compiler optimizations, and code embedding. Section III discusses the data-set used in our study. Section IV describes the proposed framework. Section V presents the results of our study, followed by a discussion section and a section on related work. Section VIII discusses the limitations of our work. Finally, section IX discusses the conclusion and future work.

II. 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 decompilation 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.

III. 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 CodeJam (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).

IV. System Overview

Our approach involves four steps: a) Code compilation using compiler optimizations, b) Code decompilation, c) Generation of code embedding, and d) Classifying the embedding in to clusters of similar code. Figure 2 shows the high level overview of our pipeline. Given a C/C++ program, we first compile the binary using one of the optimization flags to generate a binary executable. Then we use Ghidra to reverse engineer the binary to retrieve the source code. The generated source code is then fed to the code2vec model to retrieve the code embedding for the program. The embedding is used to train a model to classify the programs. In this section, we discuss all the steps in further detail.

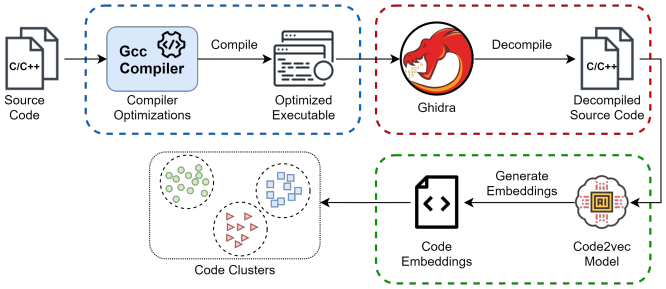


Fig. 2: An overview of our system

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 decompilation

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

The main component of getting code vectors from code2vec is *path extraction*. Code2vec first constructs the AST (Abstract Syntax Tree). Then the syntactic path between AST leaves are extracted, which form the *path-context*. Each path and leaf-values of a path-context is mapped to its corresponding real-valued vector representation, or its *embedding*. Then, the three vectors of each context are concatenated to a single vector that represents that *path-context*.

It is important to note that code2vec generated code embedding for methods as opposed to programs or files. 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. As shown in prior research [8], we can get the file level embedding by aggregating the set of method level embedding. The aggregation method is applied column-wise. The base aggregation functions used are *max*, *min*, *mean* and *median*. A combination of aggregation methods can also be considered. In our study, *median* aggregation worked best, so we constructed the program level embedding using median aggregation.

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 embedding 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 embedding were then used to train and test the classification models.

V. Evaluation and Results

A. Experimental Setting

We use five off-the-shelf machine learning algorithms and one customized DNN to train our models and demonstrate the effectiveness of the proposed work. We trained five off-the-shelf machine learning algorithms for classification tasks: Random Forest (RF), K-Nearest Neighbour (KNN), Support

Optimizaion	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%

TABLE I: Performance comparison table shows the accuracy and F1-score achieved by each model.

Vector Machines (SVM), Logistic Regression (LR), and Decision Tree (DT). We also trained a DNN. The DNN consists of an input layer (containing 384 neurons corresponding to the 384 code2vec features), one hidden layer (384 neurons and ReLu activation), and a softmax output layer (6 classes corresponding to 6 programming tasks).

We ran our training and testing scripts on a Dell XPS 8930, with Intel i5-9600K 6-core 64GB RAM, running Ubuntu 18.04 and Python 3.7.3. To evaluate the performance of the model, we use four metrics, namely, accuracy, precision, recall, and F1-score. Because of space limitations, we only report accuracy and F1-score.

B. Code Classification

We selected six programming tasks from the 2008 Google Code Jam dataset, each having about 200 data points. The dataset's total size was 1,423 programs, which was split into training and test set containing 1,280 and 143 programs, respectively. We then extracted code vectors corresponding to each program and aggregated the vectors to get a program-level representation. The process was repeated for decompiled code for different optimization levels O1, O2, O3. Ultimately, we compiled four embedding datasets for the classification task: original programs, O1 optimized, O2 optimized, and O3 optimized programs. In addition to the four datasets, we also merged all the optimized program embeddings (O1, O2, and O3) to test the performance of the models.

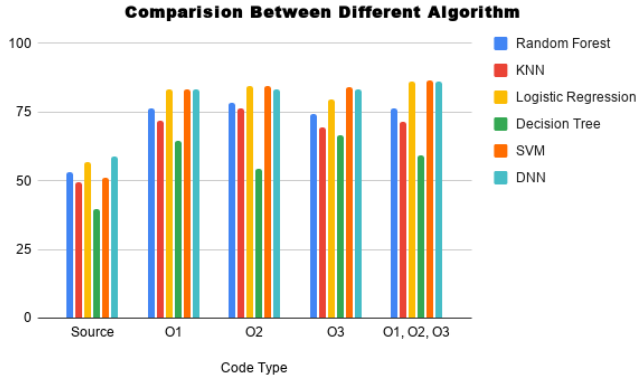


Fig. 3: Performance comparison chart compares the accuracy of each model on the optimization datasets.

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. The models collectively trained on O1, O2, and O3 optimizations outperform other models trained on single optimizations, with significant margins.

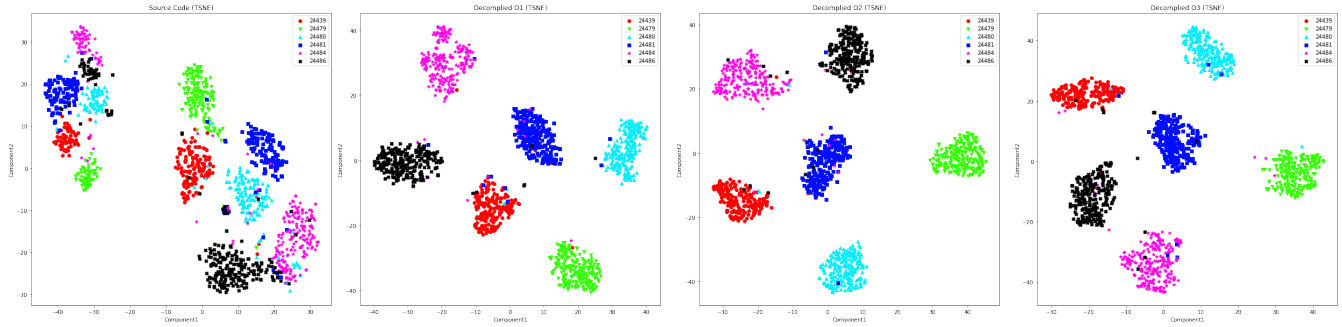
Figure 3 summarizes the accuracy of all the models. We can observe that the accuracy is sub-optimal in the case of the developer written program (depicted as 'Source'); however, the accuracy significantly increases if we apply a compiler optimization. This increase in the accuracy is owed to the transformations performed by the compiler on the source program, which results in similar binaries being constructed from semantically similar programs written by different developers.

Train Set \ Test Set	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%**

TABLE II: Performance of cross optimizations (Accuracy)

We also conducted an experiment to study the impact of cross-compiler optimization on the classification task. We trained one SVM model on each dataset and evaluated the models by testing on other datasets. Table II summarizes the results of the experiment in terms of accuracy. It can be observed that the models perform the best when tested on the same set of data it was trained on. This observation implies that the compiler transformations, under different optimization levels, produce binaries that are somewhat different from each other such that the code embedding cannot capture the similarities. Moreover, we see a unique pattern in the performance of the models trained on O1 and O3 optimized programs. For the model trained on O1 optimization, we see that the performance of the model is better on O2 dataset as compared with O3. Similarly, the performance of the model trained on O3 optimization is better on O2 as compared to O1. This pattern suggests some similarities between close levels of optimizations. The model trained on the original source programs exhibits the same pattern. As we move away from the original program, from O1 to O2 to O3, we observe a degradation of performance.

Figure 4 depicts the t-SNE plots of the code embedding for each dataset. It can be observed that source code is harder to separate, but compiler-optimized decompiled program representation shows significant improvement and allows the data points to separate. The level of separation increases as we increase the optimization level from O1 to O2. However, the separation decreases slightly in O3.



(a) Source Code **(b) Decompiled O1** **(c) Decompiled O2** **(d) Decompiled O3**

Fig. 4: t-SNE of code vectors for different optimization levels. Different colors represent different problems of the GCJ dataset. We can observe that optimization O2 segregates the data with a clear distinction between the clusters.

VI. Discussion

RQ1: Can compiler optimization be used to smooth out source code level differences introduced by the developer?

Through the code classification study, we observed that compiler optimized code has significantly higher accuracy when compared to the original developer written source code. Since the code2vec representation relies on the AST of a given method, the compiler optimizations transform semantically similar high-level developer written code to programs with similarly structured ASTs. We also observed in the motivating example, that two semantically similar code snippets can often result in similar binary.

RQ1: We observe that it is possible to smooth out source code level differences by using compiler optimizations. In particular, optimization techniques can transform, two separate but similar code snippets, to produce similar output.

RQ2: Can the compiler optimized code be used to detect similarity? If yes, then which optimizations are optimal?

Through the experiments, we demonstrated that the classifier trained on compiler optimized programs embedding can out perform classifier training on the original source code embedding. In our study, we also observed that O2 optimization performs the best among all optimizations. We hypothesize that this phenomenon occurs because several O3 optimization flags transforms the loops of the program, thereby changing the AST of the method, which in turn impacts the code vector embedding of code2vec model.

RQ2: We show that the compiler optimization O2 performs optimal transformations for the classification task.

RQ3: Can cross-optimization detect similar code?

We performed a study to learn about the impact of different optimizations on each other. We trained a DNN model on each optimization level and tested against the other optimizations. Table II summarizes the performance on cross compiler optimizations. We observe that each model learns to classify the programs with the same optimization they were trained upon. However, the models fail to adopt and perform sub-optimally on other optimizations. This outcome can be partially be attributed to vector embedding used to represent the code. We hypothesize that using a more comprehensive code representation can lead to improvement in classification task on cross compiler optimization (see section VIII).

RQ3: Cross compiler optimizations are not effective in detecting similar code.

VII. Related Work

Code similarity has been extensively studied in literature [12], particularly type-4 (semantically similar) code clones. Through a user study, researchers showed that functionally similar code exists in practice [13]. While static token based approaches such as SourcererCC[14] and CCFinder[15] have been studied, advances in computing has paved the path for two other approaches to code clone detection. We first outline machine learning approaches based on static features of the code, followed by dynamic approaches.

A. ML for Code Clone Detection

Deep learning has also been applied for detecting code clones [16], [17], [18], [19]. Researchers used both structure or identifiers to detect all four types of code clones [16]. Their technique relied on a novel code representation scheme: the terms in code fragments were mapped to vector representations such that terms used in similar ways map to similar vectors. Then the model learns discriminating features for code fragments at different levels of granularity. DeepSim is another approach that measures code functional similarity [17] by encoding control flow and data flow graphs into a semantic matrix. Another similar approach, HOLMES [18] (that relies on CFG and DFG), performs semantic code clone detection using program dependency graphs and graph neural networks by leveraging the structured, syntactic, and semantic information of the source code. FCCA [19] uses hybrid code representation by combining unstructured (code as sequential tokens) and structured (ASTs and CFGs) information of the code. Authors then train a deep-learning model with attention. Asm2vec [20] is a binary clone detection system that uses vector representation of assembly functions to detect clones.

B. Dynamic Analysis for Code Clone Detection

Tajima et al. [21] proposed to detect functionally similar code for newly created methods that do not have test cases. Authors first extract interface information and PDG from methods. Then this information is used for similarity detection. Li et al. [22] proposed a technique based on automatic test case generation to search semantically equivalent API methods by running the generated test cases. They consider two methods to be similar if the methods generate the same output on each of the generated test cases. Mathew et al. [2] proposed SLACC, a cross-language clone detection based on runtime behavior. It uses function I/O to cluster code based on its behavior. Authors generate 256 inputs per function to find similarity. Compared to dynamic techniques, our work is lightweight since we do not need to run the programs.

VIII. Threats and Limitations

Our approach relies on Code2vec embedding that utilizes ASTs of the program to generate the vector representation. For Type-I (textual similarity), Type-II (lexical similarity), and Type-III (syntactic similarity) clones, code2vec produces significantly similar ASTs because of similarity in syntactic structure. However, Type-IV code clones are only behaviorally similar; they have different syntactic structures. Hence, the underlying ASTs of Type-IV clones are significantly different, leading to different code2vec vector representation. Moreover, code2vec has been shown to rely heavily on variable names for prediction, causing it to be fooled by typos or adversarial attacks [8]. Our code2vec model was trained on the source code, and using an obfuscated version of the training data can potentially improve the performance. Moreover, there are several techniques in the literature to generate code embedding that utilize call graphs, ASTs, and other data from the code [23]. In this work, our framework relies on code2vec embedding. It is possible that other representations, such as Asm2vec [20], yield better results. We leave this to future work.

IX. Conclusion

In this paper, we propose compiler optimization based code clone detection technique. Our approach relies on the compiler to smooth out the differences in the source code introduced by the developer. We observed that O2 optimization yields the best performance (84.61% accuracy and 84.96% F1-score) for the classification task among O1, O2, and O3. Our proposed approach yields an improvement of more than 25% accuracy over the source code based representation. Furthermore, we investigated the utility of cross compiler optimization for classification problem. Our results suggest that the optimizations yield significantly different binaries making it difficult for the model to learn optimally. In the future, we plan to study program representation to accommodate cross compiler optimization and improve classification performance.

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, "Comparison 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, "Optimize 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 competitions," https://codingcompetitions.withgoogle.com/codejam, (Accessed on 03/05/2021).

[10] J. Petrík, "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

[13] V. Käfer, 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.

[14] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 1157–1168.

[15] 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.

[16] 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.

[17] 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

[18] 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.

[19] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, "FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks," *IEEE Transactions on Reliability*, pp. 1–15, 2020.

[20] 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.

[21] R. Tajima, M. Nagura, and S. Takada, "Detecting functionally similar code within the same project," in *2018 IEEE 12th International Workshop on Software Clones (IWSC)*, 2018, pp. 51–57.

[22] G. Li, H. Liu, Y. Jiang, and J. Jin, "Test-Based Clone Detection: an Initial Try on Semantically Equivalent Methods," *IEEE Access*, vol. 6, pp. 77 643–77 655, 2018.

[23] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," *arXiv preprint arXiv:1904.03061*, 2019.