Unleashing the Power of Compiler Intermediate Representation to Enhance Neural Program Embeddings

Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang*
The Hong Kong University of Science and Technology
Hong Kong SAR
{zligo,pmaab,hwangdz,shuaiw}@cse.ust.hk

Qiyi Tang, Sen Nie, Shi Wu
Tencent Security Keen Lab
China
{dodgetang,snie,shiwu}@tencent.com

ABSTRACT

Neural program embeddings have demonstrated considerable promise in a range of program analysis tasks, including clone identification, program repair, code completion, and program synthesis. However, most existing methods generate neural program embeddings directly from the program source codes, by learning from features such as tokens, abstract syntax trees, and control flow graphs.

This paper takes a fresh look at how to improve program embeddings by leveraging compiler intermediate representation (IR). We first demonstrate simple yet highly effective methods for enhancing embedding quality by training embedding models alongside source code and LLVM IR generated by *default* optimization levels (e.g., `-O2`). We then introduce IRGen, a framework based on genetic algorithms (GA), to identify (near-)optimal sequences of optimization flags that can significantly improve embedding quality.

We use IRGen to find optimal sequences of LLVM optimization flags by performing GA on source code datasets. We then extend a popular code embedding model, `CodeCMR`, by adding a new objective based on triplet loss to enable a joint learning over source code and LLVM IR. We benchmark the quality of embedding using a representative downstream application, code clone detection. When `CodeCMR` was trained with source code and LLVM IRs optimized by findings of IRGen, the embedding quality was significantly improved, outperforming the state-of-the-art model, `CodeBERT`, which was trained only with source code. Our augmented `CodeCMR` also outperformed `CodeCMR` trained over source code and IR optimized with default optimization levels. We investigate the properties of optimization flags that increase embedding quality, demonstrate IRGen's generalization in boosting other embedding models, and establish IRGen's use in settings with extremely limited training data. Our research and findings demonstrate that a straightforward addition to modern neural code embedding models can provide a highly effective enhancement.

*Corresponding author

ACM Reference Format:
Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang and Qiyi Tang, Sen Nie, Shi Wu. 2022. Unleashing the Power of Compiler Intermediate Representation to Enhance Neural Program Embeddings. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510217

1 INTRODUCTION

Recent developments in deep neural networks (DNNs) have delivered advancements in computer vision (CV) and natural language processing (NLP) applications. We have noticed lately an increase in interest in using DNNs to solve a variety of software engineering (SE) problems, including software repair [37, 89], program synthesis [14, 54, 71], reverse engineering [78], malware analysis [15], and program analysis [80, 103]. Similar to how DNNs understand discrete natural language text, nearly all neural SE applications require computing numeric and continuous representations over software, which are referred to as program embeddings or embedding vectors.

The common procedure for generating code embeddings is to process a program's source code directly, extracting token sequences, statements, or abstract syntax trees (ASTs) to learn program representations [9, 11, 17, 37, 66]. Although some preliminary approaches have attempted to extract semantics-level code signatures, such approaches are limited by use of semantic features that are too coarse-grained [73], low code coverage (due to dynamic analysis) [89], or limited scalability [90]. To date, learning from code syntactic and structural information has remained the dominant approach in this field, and as previous work has argued [27, 82, 88, 90], the use of features at this relatively "shallow" level is likely to degrade learning quality and produce embeddings with low robustness.

For some CV and NLP tasks, *data augmentation* has been proposed as a tool to improve the quality of learned embedding representations [30, 79]. These approaches typically increase the amount of training data by adding slightly modified copies of already existing data or by creating new pieces of synthetic data from existing data. Thus, embedding models can be trained on larger numbers of data samples, resulting in higher-quality embedding representations. Previous research has shown the value of data augmentation approaches in increasing embedding quality [29, 53, 57, 64].

This work investigates using compiler intermediate representations (IR) to augment code embedding. Modern compilers include numerous optimization flags that can seamlessly convert a piece of source code into a range of semantically identical but syntactically distinct IR codes. From a comprehensive standpoint, we argue that our technique can boost program embedding on two fundamental levels. **First**, the translation of a single piece of source code into several variants of IR code with the same functionality significantly increases the diversity of available training data. As previously noted, such augmented data can commonly improve the quality of learned embeddings. **Second**, although programs with the same functionality may appear syntactically distinct as source code, they are likely to become more similar after pruning and rearrangement by optimizations. This alleviates the difficulties imposed by syntax changes, as the optimizations regulate syntactic characteristics.

We begin by illustrating that using default compiler optimization levels, such as -O2 of LLVM, can produce IR code that significantly improves the embedding quality of a popular embedding model, CodeCMR [100], and outperforms the state-of-the-art (SOTA) model, CodeBERT [31], trained on source code alone. However, despite the promising potential in this "misuse" of compiler optimizations, the high number of available optimization flags and the consequently large search space present a challenge for identifying well-performing optimization sequences to augment embedding models.

We propose IRGen, a framework that uses genetic algorithms (GA) to search for (near-)optimal optimization sequences for generation of IR code to augment program embedding models. Compiler optimization flags are typically combined to generate machine instructions with high speed or small size. In contrast, IRGen targets optimization sequences, generating IR code that is *structurally similar* to the input source code. This prevents over-simplification of the IR code, which is undesirable in our task since overly-simplified IR often becomes less "expressive." Additionally, to maximize learning efficiency, we limit overuse of out-of-vocabulary (OOV) terms (our definition of OOV follows ncc [16]; see Sec. 4.3).

We present a simple yet unified extension, through triplet loss [94], to enable embedding models to learn from source code and LLVM IR. For evaluation, we used IRGen to analyze IRs generated from the POJ-104 [66] and GCJ [72] datasets, which include a total of 299,880 C/C++ programs. After 143 to 963 CPU hours of search (we use a desktop computer to run IRGen), IRGen was able to form optimization sequences with high fitness scores from the 196 optimization flags available in the x86 LLVM framework (ver. 11.1.0). To evaluate the quality of embedding, we setup a representative downstream task, code clone detection. When CodeCMR was trained with IR code generated by the identified optimization sequences, embedding quality (in terms of code clone detection accuracy) significantly improved by an average of 11.66% (peaking at 15.46%), outperforming the SOTA model CodeBERT trained with only source code (for 12.02%) or CodeCMR jointly trained with source code and IR emitted by default optimizations (for 5.94%). We also demonstrate that IRGen is general to augment other neural embedding models and show that IRGen can almost *double* the quality of learned embeddings in situations with limited data (e.g., 1% of training data available). We characterize optimization flags selected by IRGen and summarize our findings. This work can help users take use of compiler optimization, an *out-of-the-box* amplifier, to improve embedding quality. In summary, our contributions are as follows:

- We advocate the use of compiler optimizations for software embedding augmentation. Deliberately-optimized IR code can principally improve the quality of learned program embeddings by extending model training datasets and normalizing syntactic features with modest cost.
- We build IRGen, a practical tool that uses GA algorithms to iteratively form (near-)optimal optimization sequences. Additionally, we present a simple yet general extension over modern code embedding models to enable joint learning over source code and IR.
- Our evaluation demonstrates highly promising results, with our augmented model significantly outperforming SOTA models. We further demonstrate the generalization of IRGen and its merit in augmenting very limited training data. IRGen is released at [1].

2 PRELIMINARY

Neural code embedding, as in Fig. 1, converts discrete source code to numerical and continuous embedding vectors, with the end goal of facilitating a variety of learning-based program analysis. We introduce program representations in Sec. 2.1. We examine alternative model designs in Sec. 2.2 and the concept of data augmentation for neural (software) embedding in Sec. 2.3.

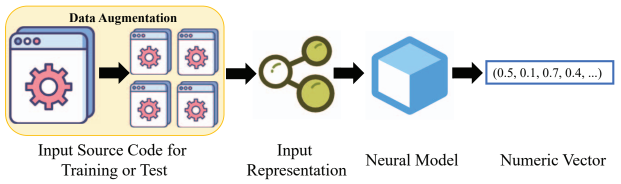

Figure 1: Common neural program embedding pipeline.

2.1 Input Representation

Code can be expressed as text and processed using existing NLP models. However, it would be costly and likely ineffective because programming languages usually contain a wealth of explicit and sophisticated structural information that is difficult for NLP models to comprehend [66]. Therefore, modern code embedding models often learn program embeddings using code structural representations which are informative. For instance, the abstract syntax tree (AST) is used to represent code fragments for program embeddings. Once a code snippet's AST has been generated, there are several methods for extracting discrete symbols (e.g., AST nodes) for use in the subsequent learning process. For example, code2vec [11] and code2seq [10] extract a collection of paths from AST to form embeddings, as discussed below.

Control flow graphs (CFG) are also used to form input representation, especially when analyzing assembly code. Two representative tools, asm2vec [27] and BinaryAI [99], construct CFGs over assembly code and combine basic block-level embeddings into the program's overall embedding. Recent research [8, 16, 36] has explored the use of hybrid representations that incorporate data from different layers. For instance, ncc [16] extracts a so-called contextual flow graph first, which subsumes information from both control flow graph and data flow graph.

2.2 Neural Model Learning Procedure

NLP models are generally designed to process infinite sequences of tokens, whereas software is structured. Hence, the neural code embedding learning process can be divided into two broad categories: 1) decomposing program (structural) representations (e.g., AST or CFG) into one or multiple token sequences that are then processed by NLP models; and 2) attempting to initiate an "end-to-end" procedure for directly learning structural representations using advanced neural models like graph neural networks (GNN).

CodeBERT is a large-scale SOTA code embedding model that primarily learns from token-level software representations. It is inspired by BERT [26], a famous bidirectional natural language embedding model. CodeBERT constructs learning objectives using both masked language modeling (MLM) and replacement token detection. Using these objectives, it is trained to predict tokens that have been randomly masked out of the inputs until saturation accuracy is reached. Another model, asm2vec [27], uses MLMs, particularly an extended PV-DM model [52], to embed x86 instructions at the token level. Token sequences can be extracted from tree or graph representations. For example, code2vec [11] and code2seq [9] break AST into paths, transform paths to embeddings using LSTM [42], and finally aggregate path-level embeddings to produce the AST's embedding. The structure-based traversal method [43] converts ASTs into structured sequences.

TBCNN [67], Great [41] and BinaryAI [99] leverage advanced models, such as GNNs, to directly process program structural representations. BinaryAI [99], for example, uses standard GNNs to propagate and aggregate basic block embeddings into CFG embeddings. Besides CFGs, neural models can create structures with richer information. ncc [16] forms a contextual flow graph with control- and data-flow information. Each node in the contextual flow graph contains a list of LLVM IR statements, which ncc then transforms into vectors. It further uses a GNN to aggregate the node embeddings into an embedding of the entire program. As with ncc, MISIM begins by constructing a novel context-aware semantic structure (CASS) from collections of program syntax- and structure-level properties. It then converts CASS into embedding vectors using GNNs. It outperforms prior AST-based embedding tools, including code2vec and code2seq [9].

2.3 Data Augmentation

Images can be rotated while retaining their "meaning" (e.g., via affine transformations [104]). Similarly, we can replace words in natural language sentences with their synonyms, which should not impair linguistic semantics. Data augmentation leverages these observations to create transformation rules that can enlarge model training data.

It is worth noting a conventional technique, namely feature engineering [105], can generally help data science and machine learning tasks. Feature engineering facilitates to eliminate redundant data that can reduce overfitting and increase accuracy. Nevertheless, in the era of deep learning, it gradually becomes less desirable to manually "pick useful features," given that we need to frequently deal with high-dimensional data like image, text, video, and software. How to pick useful features is often obscure when learning from those complex high-dimensional data. In fact, it has been demonstrated that data augmentation generally and notably improves deep learning model performance and robustness, and it has been frequently employed as a routine technique to enhance modern deep learning models in a variety of domains [60, 61, 70, 75, 77, 91, 93, 101, 102, 104].

Table 1: MAP scores of CodeCMR on POJ-104 [66] for different input setup.

Setup	MAP(%)	Setup	MAP(%)
Source	76.39	Source + LLVM IR -O2	84.29
Source + LLVM IR -O0	82.90	Source + LLVM IR -O3	84.21
Source + LLVM IR -O1	83.37	Source + LLVM IR -Os	83.81
Source + LLVM IR Optimized by Optimization Sequences Found by IRGen			89.18

Standard data augmentation approaches, however, are *not* directly applicable to enhance program embeddings. Augmenting neural program embeddings is challenging and under-explored. Due to the synthetic and semantic constraints of programming languages, arbitrary augmentation can easily break a well-formed program. This paper explores bringing data augmentation to source code. In particular, we advocate employing compiler optimizations to turn a same piece of source code into semantically identical but syntactically diverse IR code. Note that we do not need to "reinventing the wheel" to develop extra semantics-preserving source code transformations [44]. Instead, we demonstrate how a mature compiler can facilitate effective data augmentation simply by exploiting optimizations developed over decades by compiler engineers.

3 MOTIVATION

The LLVM compiler architecture supports hundreds of optimization passes, each of which mutates the compiler IR in a unique way. To make compiler optimization more accessible to users, the LLVM framework offers several optimization bundles that a user can specify for compilation, for example, -O2, -O3, and -Os. The first two bundles combine optimization passes for fast code execution, whereas -Os aims to generate the smallest executable possible. Our preliminary study shows that by incorporating optimized IR code into embedding learning, the embedding quality can be substantially enhanced. This section describes our preliminary finding, which serves as an impetus for the subsequently explored research.

Learning Over Source Code. We use POJ-104 [66], a commonly used dataset containing 44,912 C programs written for 104 tasks. This dataset is split into three program sets: one for training, one for validation, and one for testing (see Sec. 6). We trained CodeCMR [100], one popular code embedding tool, on the training split, and then perform multi-label classification on the testing split. CodeCMR generates code embeddings by first converting source code to a character sequence and then computing character-level embeddings. The embeddings are fed to a stack of Pyramid Convolutional Neural Network (DPCNN) [45], in which an average pooling layer constructs the program's embedding. DPCNN has been shown as powerful at embedding programs. In our evaluation on POJ-104, we observe promising accuracy in terms of MAP [68] score, as shown in Source of Table 1. MAP is a commonly used metrics in this field, and a higher MAP score indicates a greater quality of code embeddings. As will be shown in evaluation (Table 4), this result is comparable to those obtained using the SOTA model, CodeBERT.

Findings. Despite the decent results, we find that CodeCMR fails to group quite a number of POJ-104 programs who belong to the same class. The POJ-104 C code and corresponding LLVM IR are too lengthy to fit in the paper; we present some very readable cases at [3] and summarize our key observations below.

C/C++ programs implementing the same functionality can exhibit distinct syntactic appearance. For instance, at [3], we present a case where two programs, p_1 and p_2, are implementing the same "days between dates" task. We find that p_1 uses one `switch` statement, whereas p_2 uses a sequence of `if` statements. Further, p_1 uses many local variables to encode #days in each month, while those information in p_2 are hardcoded in constants. This way, p_1 and p_2, differ from both control- and data-flow perspectives.

Nevertheless, we find that the LLVM IR code compiled from these two programs are much closer in both control- and data-flows. Let l_1 and l_2 (see [3]) be two LLVM IR programs compiled from p_1 and p_2 and optimized with optimization level -O3. We find that l_1 and l_2 preserves most of the structure of the source code. More crucially, l_1 and l_2 both use a LLVM IR `switch` statement to encode the control structures. Data usage is also regulated, where both local variables in p_1 and the constants in p_2 become integers hardcoded in IR statements. The induced IR programs l_1 and l_2 are (visually) very similar, revealing the true semantics-level equivalence of p_1 and p_2. We thus suspect that `CodeCMR` is indeed hampered by too flexible code representation in C programs. In other words, it is shown as demanding to explore extracting more robust features from C/C++ code to enhance the learning quality.

Learning over Code Structure or Semantics. As previously stated, `CodeCMR` learns on the character (token) sequence. This indicates that `CodeCMR` is less resilient against changes at the syntactic level. Graph-level embeddings might be more robust to token-level changes, given their reliance on the rich structural information contained in the program. Nonetheless, in real-life code samples, many changes can also occur at the graph level, and as shown in Sec. 6, representative graph-level embedding models also perform poorly on diverse and large-scale datasets, such as POJ-104.

Some readers may wonder if learning directly from *code semantics*, such as input-output behaviors captured by dynamic analysis [88, 90], is possible. While dynamic analysis can precisely describe code behaviors (on the covered paths), it suffers from low coverage. Symbolic execution (SE) [20] is used to include a greater amount of program activity in applications such as code similarity analysis [59]. Nonetheless, SE is inherently inefficient, where trade-offs are made to launch SE over real-world software [19, 85].

Learning over IR Code. This paper advocates using compiler IR to extend model train dataset and enhance code embedding models. However, we do *not* suggest learning solely from IR for two reasons. First, compiler optimizations such as static single-assignment (SSA) [25] result in LLVM IR codes that typically have ten times as many lines of code (LOC) as the corresponding C source code. This provides a significant impediment to training embedding models. In our preliminary study, we find that training embedding models using LLVM IR code alone resulted in significantly inferior performance across multiple tasks and datasets. Second, when outputting IR code, the LLVM compiler prunes or "obfuscates" certain source code features such as string and variable names. Note that variable names and constants are generally crucial to improving embedding quality. Similarly, in LLVM IR code, callsites, particularly to standard libraries like `glibc`, are often modified. For example, the callsite statement in `set<int> row; row.insert(x);` would be converted to a complex function name with additional prefixes. Notably, we should avoid tweaking with or disabling certain "annoying" optimization passes (for example, the SSA pass), as many optimization flags assume the existence of other flags.

Learning Over Source Code and IR Code. We extended `CodeCMR` to process IR code emitted by `clang`. We augmented the frontend of `CodeCMR` to process LLVM IRs. We also extended the learning objectives by requiring `CodeCMR` to minimize the distance between the source code and corresponding IR using triplet loss (see Sec. 4.4). We compiled each test program in the POJ-104 training set into LLVM IR to train the `CodeCMR`, and then benchmark the MAP score using the same setting.

As seen in Table 1, using LLVM IR in the learning process significantly improved embedding performance. For instance, when compiling the source code into LLVM IR with negligible optimization (`-O0`), the joint learning enhanced the MAP score by approximately 6%. Note that jointly training over source code and IR (`-O0`) has already outperformed the SOTA model, `CodeBERT` (82.7%). More importantly, it is seen that compiler optimizations can notably improve `CodeCMR`'s performance. We observe that as compared with `-O0`, using optimization levels `-O2`, `-O3`, and `-Os` produces MAP scores greater than 84%.

We regard the above findings as encouraging and intuitive: they demonstrate the possibility and benefit of learning jointly from source code and IR code (which are more regulated) rather than from source code alone. In evaluation (Sec. 6.3), we discuss optimization flags further with case studies to reveal that they can effectively regulate code generation patterns, remove superfluous code fragments, and generate more consistent IR code in the presence of syntactic or structural changes in the source code. We therefore summarize the key findings of this early investigation as follows:

> Launching a joint training using both program source code and corresponding IR can notably improve the embedding quality.

Limitation of Standard Optimization Levels. Despite the encouraging results, we note that these default optimization levels are selected by compiler engineers with *different* focus, e.g., producing smallest possible executable or fast execution. However, we explore a different angle, where optimizations are "misused" to generate LLVM IR code to *augment program neuron embedding*. In that regard, it is possible to suspect the inadequacy of utilizing simply the default optimization levels. For instance, certain CPU flags in `-Os` and `-O3` are aggressive in shrinking IR code (e.g., `-aggressive-instcombine`), which, might not be proper since embedding models generally prefer more "expressive" inputs. In evaluation, we find that aggressive flags such as `-aggressive-instcombine` are not picked by IRGen. We also find that optimization flags should be adaptive to source code of different complexity, whereas default optimization levels are fixed. Sec. 6.3 compares optimization flags selected by IRGen when analyzing different datasets.

We introduce IRGen, an automated framework to determine (near-)optimal sequences of optimization flags for each particular dataset. To compare with standard optimization levels, the final row of Table 1 presents the improved performance of `CodeCMR` when using IRGen-selected optimization flags. These results show that IR code optimized using IRGen-formed optimization sequence significantly improved the accuracy.

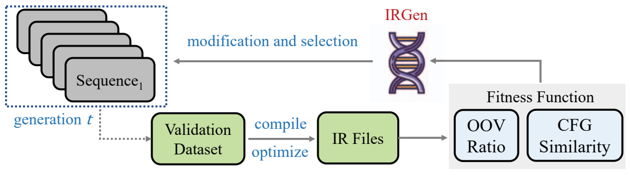

Figure 2: The workflow of IRGen.

4 DESIGN OF IRGEN

Fig. 2 depicts IRGen's workflow. Given a dataset of C source code, we first form a validation dataset P where fitness scores are computed from (see Sec. 4.3). Our observation shows that the size of P does not need to be large (otherwise the GA procedure becomes slow). For the current implementation, we randomly select 5% programs from the training dataset (POJ-104 or GCJ; see details in Sec. 6) of `CodeCMR`.

IRGen initializes the first generation of optimization sequences (Sec. 4.1), and then launches the GA-based search to iteratively modify and select sequences giving high fitness scores (Sec. 4.2). The GA process is repeated N times until termination (N is 800 currently). After termination, we select the K sequences with the top-K highest fitness scores across the entire GA search. Using these optimization sequences, each piece of C/C++ code in the training data can be compiled into K syntactically distinct pieces of LLVM IR code. The resulting augmented code dataset can be used to empower neural embedding models, by incorporating triplet loss as an extra learning objective (Sec. 4.4).

Application Scope. This research mainly focuses on the LLVM compiler framework given its popularity. LLVM provides a total of 196 optimization flags that are applicable on x86 platforms. IRGen traverses the entire search space to identify sequences expected to improve embedding quality. IRGen's iterative method is *orthogonal* to the LLVM framework and can therefore be extended to support other compilers, such as `gcc` (to manipulate its GIMPLE IR), without incurring additional technical difficulties.

The current implementation of IRGen primarily enhances C/C++ code embedding. C/C++ programs can be compiled into LLVM IR and optimized accordingly. Moreover, most security related embedding downstream applications (e.g., CVE search [27]) concern C/C++ programs. Nevertheless, we clarify that code embedding has its wide application on other languages such as Java/Python. It is worth noting that IRGen relies on a rich set of compiler optimizations to generate diverse IR code. Java/Python compilers/interpreters provide fewer optimization passes, and they leave many optimizations at runtime. As a result, the search space for IRGen to explore would be much smaller. We also have a concern on the expressiveness of Java/Python bytecode in comparison with LLVM IR. Their bytecode seems very succinct, potentially undermining the SOTA embedding models. Overall, we leave it as one future work to explore extending IRGen to enhance Java/Python embedding.

IRGen's present implementation does not consider the order of optimizations in a sequence. We also assume each flag can only be used once. This enables a realistic and efficient design when GA is used; similar design decisions are also made in relevant works [55, 74]. Taking orders or repeated flags into account would notably enlarge the search space and enhance the complexity of IRGen. We reserve the possibility of using metaheuristic algorithms with potentially greater capacity, such as deep reinforcement learning [65], for future work. See Sec. 7 for further discussion.

Design Focus. IRGen's GA-based pipeline was inspired by literatures in search-based software engineering, particularly using GA for code testing, debugging, maintenance, and hardening [5, 33, 63, 69, 74, 92]. Sec. 8 further reviews existing studies. Our evaluation will show that the GA method, when combined with our well-designed fitness function, is sufficiently good at selecting well-performing optimization sequences. Further enhancement may incorporate other learning-based techniques; see Sec. 7.

4.1 Genetic Representation

Following common GA practice, we represent each optimization sequence as a one-dimensional vector $v = (f_1, f_2, \ldots, f_L)$, where L is the total number of optimization flags offered by LLVM for x86 platforms. Each f_i is a binary number (0/1), denoting whether the corresponding flag, c_i, is enabled or not on sequence i. As standard setup, we initialize M instances of vector v, by randomly setting elements in a vector v as 1. These randomly initialized sets, referred to as a "population" in GA terminology, provide a starting point to launch generations of evolution. Here, M is set as 20.

4.2 Modification and Selection

At each generation t, we employ two standard genetic operators, Crossover and Mutation, to manipulate all 20 vectors in the population. Given two "parent" vectors, v_1 and v_2, two offsprings are generated using k-point crossover: k cross-points are randomly selected on v_1 and v_2, and the content marked by each pair of cross-points is swapped between them. Here, k is set as 2, and the chance of each potential crossover is set as 0.4. We also use flip bit mutation, another common method, to diversify vectors in the population. We randomly mutate 1% of bits in vector v. After these mutations, the population size remains unchanged (20 vectors), but some vectors are modified. Each vector is assessed using the fitness function defined in Sec. 4.3. All mutated and unmutated vectors are then passed into a standard roulette wheel selection (RWS) module, where the chance for selecting a vector is proportional to its fitness score. This way, a vector with a higher fitness score is more likely to be selected into the next generation. The RWS procedure is repeated 20 times to prepare 20 vectors for generation $t + 1$.

4.3 Fitness Function

Given a vector, v, denoting a sequence of optimization flags, fitness function $\mathcal{F}$ yields a fitness score as an estimation of v's merit. Specifically, for each v, we compile every program p in the validation dataset P using optimizations specified in v to produce IR programs $l \in L$. For a C program p and its compiled IR l, we first compute the following fitness score:

$$Fitness\_Score_{p,l} = sim_G \times \frac{unk\_rate_0}{unk\_rate_l}$$

where sim_G denotes the graph-level similarity between the l and p. The value of unk_rate_0 denotes the number of #OOV cases found in IR code l_0 when compiling p with `-O0` (i.e., the baseline), and unk_rate_l stands for #OOV cases found in l. Then, $\mathcal{F}$ is acquired by averaging the above fitness score for all programs $p \in P$.

Graph Similarity. The graph similarity metric quantifies the similarity between the original source code and the compiled IR code at the CFG level. This provides a high-level assessment of the created IR code's quality. More importantly, this condition prevents excessive reduction of the code by the compiler optimizations, ensuring that the IR code reasonably preserves the original source code's structure-level features.

We tentatively assessed three graph similarity computation methods: 1) kernel methods, 2) graph embedding [35, 97], and 3) tree edit distance. Graph embedding methods often require to fine-tune a large number of hyper-parameters which is generally challenging. We also find that tree edit distance algorithms had limited capacity to process the very complicated CFGs created for our test cases. IRGen's present implementation therefore uses a classic and widely-used kernel method, shortest-path kernels [18], to quantify the structural distances between source code p and its optimized IR code l. Overall, kernel methods, including shortest-path kernels, denote a set of methods originated from statistical learning theory to support pattern analysis [21]. Kernel methods are shown to be effective in various tasks such as classification and regression. For our scenario, we feed the employed kernel function with a pair of CFG derived from source code p and the corresponding IR l, and the kernel function returns a score sim_G.

OOV Ratio. In embedding learning, OOVs represent tokens that are rarely observed and are not part of the typical token vocabulary. We clarify that we follow `ncc` [16] to define vocabulary. Particularly, our vocabulary denotes a bag of IR statements, and therefore, IR code is represented by a list of statement embeddings. Accordingly, "OOV" in our context denotes a new statement never occurring in the baseline vocabulary. Such new statements correspond to a special embedding noted as "[unknown]" in our implementation, degrading the learning quality.

A high #OOV is discouraged in the fitness function. That is, we leverage the OOV ratio to punish an optimization sequence if it results in an IR code with too many OOV cases. To this end, a "baseline" is first computed, recording the #OOV encountered in IR code generated by compiling p with `-O0`. Then, given optimization sequence v, we count the #OOV cases identified in its optimized IR code l, and compute the relative OOV ratio.

We clarify that it is possible to avoid token-level OOV issue by leveraging sub-tokenization techniques like BPE [46]. Given that said, in the current setting, an IR statement is represented by a single embedding vector, whereas BPE represents a statement by multiple vectors of sub-tokens. The extra overhead due to multiple vectors is seen as acceptable for source code but unaffordable for IR code, which is orders of magnitude longer. In fact, our preliminary study explored using BPE: we report that BPE would result in 16× and 30× longer vectors on our test datasets, POJ-104 [66] and GCJ [72].

4.4 Learning Multiple IRs using Triplet Loss

Instead of keeping single sequence with the highest fitness score, IRGen retains the top-K sequences from each generation, as ranked by their fitness scores. We find that it is beneficial to perform augmentation with multiple LLVM IR codes generated by the top-K optimization sequences (see results in Sec. 5). Given the GA procedure, these top-K sequences will evidently share some overlapping optimization flags. However, we find that when a source program is compiled into K LLVM IR programs using these top-K sequences, these K IR programs are still distinct (see cases at [2]), although they share regulated code structures that are correlated with the reference source code. Hence, we anticipate that the augmented dataset will be diverse, which has been generally shown to be useful in enhancing embedding learning quality [50, 56, 96]. K denotes a hyper-parameter of IRGen. We benchmark the accuracy changes in terms of different K in Sec. 5.

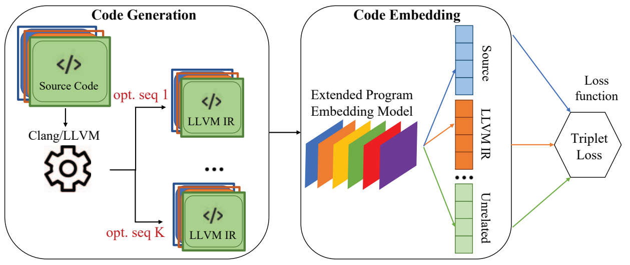

Figure 3: Learning from IR code with Triplet Loss.

Fig. 3 depicts an efficient and general extension over program embedding models to subsume multiple IR code. As expected, we first extend a code embedding model M to process LLVM IR. Then, we employ a popular learning objective, namely triplet loss [94], as the loss function of M. The triplet, which consists of a positive sample, a negative sample, and an anchor, is used as the input for triplet loss. An anchor is also a positive sample, which is initially closer to some negative samples than it is to some positive samples. The anchor-positive pairs are pulled closer during training, whereas the anchor-negative pairs are pushed apart. In our setting, a positive sample represents a program p, anchor represents IR code produced from p, and negative samples represent other unrelated source code.

Note that M is not necessarily `CodeCMR`. Other non-trivial source code embedding models can serve M in this pipeline; see our evaluation on generalization in Sec. 6.4. Further, while we adopt Fig. 3 to enhance M, we clarify that there may exist other augmentation pipelines. We provide proposals of other pipelines at [4] for information of interested audiences.

5 IMPLEMENTATION

IRGen is written primarily in Python with about 9K lines of code. This primarily includes our GA pipeline (Sec. 4) and extension of `CodeCMR` (see below). IRGen is based on LLVM version 11.1.0 [51]. We also tentatively tested LLVM version 7.0, which works smoothly. IRGen is built in a fully automated and "out-of-the-box" manner. Users only need to configure IRGen with the path of their LLVM toolchain. We release IRGen and data (e.g., augmented models) at [1]. Our results can be reproduced using our released artifacts. We pledge to keep IRGen up to date to support future study.

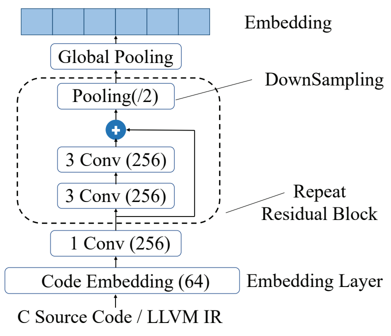

Figure 4: The main structure of CodeCMR.

Enhancing `CodeCMR`. As stated in Sec. 3, IRGen is currently implemented with `CodeCMR` [100], which is a SOTA code embedding model that has been thoroughly tested on real-world C/C++ programs. We find that its code is straightforward to use and of high quality. We emphasize that IRGen is orthogonal to the particular code embedding models used. We assess the generalization of IRGen using another embedding tool, `ncc`, which directly computes embeddings from LLVM IR; see Sec. 6.4. We extended the official version of `CodeCMR` to jointly compute embeddings using C source code and LLVM IR code. We also implement a C/C++ parser based on treesitter [86] and a LLVM IR parser (extended from `ncc`), as we need to compare distance of C/C++ and IR code using kernel methods. Fig. 4 depicts the main network structure of our extended `CodeCMR`. `CodeCMR` is a variant of DPCNN, which has been shown to efficiently represent *long-range associations* in text. As shown in Fig. 4, the key building block, a word-level convolutional neural network (CNN), can be duplicated until the model is sufficiently deep to capture global input text representations. Given that (IR) programs are typically lengthy and contain extensive global information, `CodeCMR` exhibits promising accuracy.

Table 2: Augmentation using K collections of optimized IR.

	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
MAP	86.34	88.03	88.96	89.71	90.16	**91.12**	90.00

Tuning K. Recalling that IRGen generates K collections of IR codes by returning the top-K sequences, we now compare the effect of K on learning quality. We ran our experiments using `CodeCMR` trained on POJ-104 and measured the MAP accuracy for different K in Table 2. Overall, although increasing K can continuously extend the training data, the learning accuracy reached its peak value when K=6. We interpret these results to confirm another important (and intuitive) observation:

Aligned with data augmentation on natural language or image models, involving multiple diverse IR code collections in training datasets augments the learning quality of code embedding.

We provide samples on [2] to illustrate how source code can be compiled into K pieces of diverse IR code. In our evaluation (Sec. 6), we chose K=6 as the default option. However, we clarify that the best value of K, as a hyper-parameter, can be influenced by both the specific dataset and the neural embedding model. We therefore recommend users to tune K for their specific learning task.

6 EVALUATION

Our evaluation aims to answer the following research questions: **RQ1**: How does `CodeCMR`, after enhanced by IRGen, perform in comparison to other relevant works on code clone detection? **RQ2**: How accurate is the genetic algorithm (GA) adopted by IRGen? **RQ3**: What are the most important optimization flags and their characteristics? Does the optimal sequence of flags change on different datasets? **RQ4**: What is the generalization of IRGen with respect to other models and different learning algorithms? **RQ5**: Can IRGen still achieve promising augmentation when only limited source code samples are available? Before reporting the evaluation results, we first discuss the evaluation setup as follows.

Dataset. We used the POJ-104 [66] and GCJ [72] datasets for our evaluations. Table 3 reports the summary statistics of these two datasets. As mentioned in Sec. 3, the POJ-104 dataset contains 44,912 C/C++ programs that implement entry-level programming assignments for 104 different tasks (e.g., merge sort and two sum). The Google Code Jam (GCJ) is an international programming competition that has been run by Google since 2008. The GCJ dataset contains the source code from solutions to GCJ programming challenges. The GCJ dataset is commonly used and contains 260,901 C/C++ programs. Compared to POJ, the GCJ files are longer and more numerous. We find that GCJ files contain complex usage of C macros. As described later in the evaluation, we found that the more lengthy GCJ code and its relatively complex code structures had notable impacts on the optimization sequences selected by the GA procedure. For both datasets, we used the default setting to split them for training and testing. We did not use their validation splits. For each dataset, we used IRGen to select the top-K optimization sequences retained by the GA process. We then compiled each C source code in the training datasets into K pieces of LLVM IR code to extend the training datasets.

Our method requires that the training codes be *compilable*. We indeed explored some other datasets such as Devign [106]. However, we found that many of its cases cannot be compiled. Fixing these issues would have required considerable manual effort. Another practical concern is *cost*; as soon reported in **Cost**, training `CodeCMR` on GCJ already takes over 90 hours on *16 Tesla V100 GPU cards*. Considering larger datasets is out of scope for this research project.

Table 3: Statistics of the dataset used in evaluation.

Split	GCJ	POJ-104
Classes in training data	237	64
Programs in training data	238,106	28,103
Classes in test data	31	24
Programs in test data	22,795	10,876
Programs with macro	80,432	10
Average lines of C code	71.19	35.97
Average lines of LLVM IR code	1659.50	238.51

Table 4: Accuracy of all (augmented) models. For each metrics, we mark best models when training with C source code. We also mark the best models when training with C code and LLVM IR code optimized following different schemes. IRGen denotes training CodeCMR using source code and six collections of LLVM IR optimized by sequences formed by IRGen.

Method	GCJ		POJ-104	
	MAP@R(%)	AP(%)	MAP@R(%)	AP(%)
code2vec	7.76 (-0.79/+0.88)	17.95 (-1.24/+1.76)	1.90 (-0.43/+0.38)	5.30 (-0.80/+0.60)
code2seq	11.67 (-1.98/+1.73)	23.09 (-3.24/+2.49)	3.12 (-0.45/+0.67)	6.43 (-0.37/+0.48)
ncc	17.26 (-1.11/+0.57)	31.56 (-1.11/+1.46)	39.95 (-2.29/+1.64)	50.42 (-2.98/+1.61)
ncc-w/o-inst2vec	34.88 (-5.72/+7.63)	56.12 (-7.63/+9.96)	54.19 (-3.18/+3.52)	62.75 (-5.49/+4.42)
Aroma-Dot	29.08	42.47	52.08	45.99
Aroma-Cos	29.67	36.21	55.12	55.40
CodeCMR	64.86(-1.49/+0.72)	98.52(-0.16/+0.12)	76.39(-0.55/+1.30)	77.18(-2.95/+1.92)
MISIM-GNN	74.90(-1.15/+0.64)	92.15(-0.97/+0.7)	82.45 (-0.61/+0.40)	82.00 (-2.77/+1.65)
CodeBERT	68.95(-0.91/+0.37)	81.34(-1.29/+0.36)	82.67(-0.42/+0.33)	85.73(-1.14/+2.13)
CodeCMR-O0	81.08(-1.03/+0.58)	96.31(-0.34/+1.11)	82.90(-1.24/+0.97)	84.95(-2.53/+1.03)
CodeCMR-O1	83.87(-0.77/+0.24)	97.10(-0.27/+0.54)	83.37(-0.97/+0.31)	86.61(-1.35/+0.78)
CodeCMR-O2	82.60(-0.81/+0.19)	96.28(-0.57/+0.27)	84.29(-1.24/+0.53)	85.96(-1.18/+0.91)
CodeCMR-O3	82.67(-1.13/+0.69)	96.77(-0.44/+0.64)	84.21(-0.43/+0.98)	85.06(-0.83/+0.39)
CodeCMR-Os	85.17(-0.24/+0.38)	98.02(-0.31/+0.13)	83.81(-0.93/+0.24)	85.07(-0.72/+1.21)
CodeCMR-demix	84.93(-1.44/+0.73)	98.02(-0.49/+0.31)	85.14(-0.71/+0.76)	88.58(-0.93/+0.44)
IRGen	86.48(-1.13/+1.57)	99.94(-0.07/+0.02)	89.18(-0.33/+0.61)	93.24(-0.21/+0.09)

Baseline Models. To compare with CodeCMR augmented by IRGen, we configure seven embedding models, including CodeBERT [31], code2vec [11], code2seq [9], ncc [16], Aroma [58], CodeCMR and MISIM [98]. CodeCMR was introduced in Sec. 3. CodeBERT, ncc, and MISIM were introduced in Sec. 2.2.

ncc is a unique and extensible code embedding framework that learns directly from LLVM IR code. As expected, ncc can be augmented with LLVM IR optimized by IRGen (see Sec. 6.4). When using ncc, we assessed two variants, ncc and ncc-w/o-inst2vec. The latter model omits the standard ins2vec model [16] for IR statement-level embedding and instead uses a joint learning approach to simultaneously compute and fine-tune the statement and graph-level embeddings. For MISIM, we leveraged its provided variant, referred to as MISIM-GNN, that leverages GNNs in the learning pipeline and has been shown to outperform other MISIM variants.

Aroma was released by Facebook to facilitate high-speed query matching from a database of millions of code samples. Aroma does *not* perform neural embedding but instead contains a set of conventional code matching techniques (pruning, clustering, etc.). We selected Aroma for comparison because it is a SOTA production tool that also features code clustering and similarity analysis. Hence, Aroma and neural embedding tools can be compared on an equivalent basis, demonstrating the strength of SOTA neural embedding tools, particularly after augmentation using IRGen. The official codebase of Aroma provides two variants, Aroma-Dot and Aroma-Cos. We benchmarked both variants.

Cost. Our learning and testing for GA were conducted on a desktop machine with two Intel Core(TM) i7-8700 CPU and 16GB RAM. The machine was running Ubuntu 18.04. IRGen takes averaged 143.52 and 963.41 CPU hours to finish all 800 iterations of GA procedure for POJ-104 and GCJ, respectively. Despite the high CPU hours, we clarify that the wall-clock time can be largely reduced via parallelism. We explored to re-run the GA procedure on a 64-core CPU server. We report that it takes about 25 wall-clock hours for POJ-104, and about 81 wall-clock hours for GCJ. Setting this parallelism changes about 60 LOC in IRGen; see our codebase at [1]. When needed, it is also possible to optimize GA with subsampling for extremely large datasets.

Training embedding models are usually *very costly*. We employ a GPU server for training for all involved models. The server has two Intel(R) Xeon(R) Platinum 8255C CPUs operating at 2.50GHz, 384 GB of memory and 16 NVIDIA Tesla V100 GPU, each with 32GB RAM. The learning rate is 0.001 and the repeat number of residual blocks is 11; other settings of our extended CodeCMR are the same with the standard CodeCMR setting. In total, over 120 epochs took approximately 15.9 and 91.3 hours for POJ-104 and GCJ, respectively.

6.1 Accuracy of IRGen

We first answer **RQ1** using Table 4. For neural embedding models, we launch each experiments for three times and report the average, as well as the minimum and maximum scores in parentheses. Table 4 reports the evaluation results of baseline models in lines 3–11. In accordance with our research motivation (Sec. 3), we also report results using CodeCMR augmented with IR code optimized by standard optimization levels (-O0, -O1, -O2, -O3, -Os). CodeCMR-demix represents training CodeCMR by using source code and five sets of IR compiled by all five default optimization levels. The last row in Sec. 3 reports the performance metrics for CodeCMR augmented by six collections of LLVM IRs optimized using sequences generated by IRGen. For both the POJ-104 and GCJ datasets, in addition to MAP, we used AP [13] as the metric. Both metrics are commonly used in relevant research to assess performance of embedding models. AP stands for Average Precision, a method combines recall and precision for ranked retrieval results. For both metrics, a higher score indicates better performance.

Our results show that modern embedding models, including CodeBERT, CodeCMR, and MISIM-GNN, can largely outperform conventional code matching tools such as Aroma-Dot and Aroma-Cos. When learning over C source code, we found that CodeBERT was the best performing model for the POJ-104 dataset, whereas MISIM-GNN delivered the best performance for the GCJ dataset. In contrast, CodeCMR performed less well than either of the SOTA models across all of the evaluated metrics. code2vec and code2seq shows relatively lower accuracy compared with others. Since we run each evaluation three times, we find that their accuracy scores are unstable. Such observation is also consistently reported in previous works [98]. Nevertheless, even the "peak" accuracy scores of them are still much lower than that of the SOTA models.

When learning from IR optimized using standard optimization levels, CodeCMR outperformed SOTA model MAP scores by more than 10% on the GCJ dataset. Evaluation of this form of CodeCMR training on the POJ-104 dataset showed consistently promising enhancement relative to the SOTA models in most cases. Also, comparing with augmenting CodeCMR with one collection of optimized IR code, the CodeCMR-demix setting shows (slightly) better performance, particularly for the POJ-104 setting. This also reveals the strength of training with multiple diverse sets of IR code.

We found that CodeCMR, when augmented by findings of IRGen (the last row of Table 4), constantly and notably outperformed all the other settings. We interpret the evaluation results as highly encouraging, showing that IRGen can generate high-quality LLVM IR code that enhances CodeCMR to significantly outperform the SOTA models (CodeBERT and MISIM-GNN) on all metrics. Again, we note that IRGen is not limited to enhance CodeCMR: we present evaluation of enhancing ncc in Sec. 6.4.

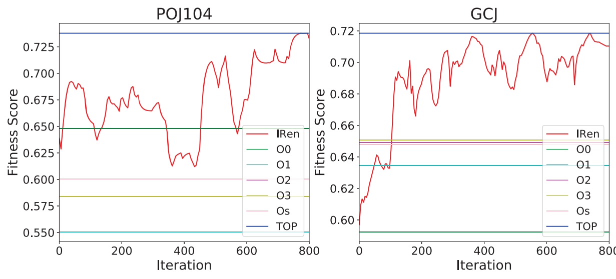

Figure 5: Smoothed fitness score increases over 800 iterations.

6.2 Fitness Function

RQ2 assesses the efficiency of our fitness function. Fig. 5 reports the fitness score increases from all 800 IRGen iterations across each GA campaign. The test cases, despite their diverse functionality, manifested encouraging and consistent trends during optimization searching. The fitness scores kept increasing and were seen to reach saturation performance after around 410 to 600 iterations. We interpret that under the guidance of our fitness function, IRGen can find well-performing sequences for both datasets.

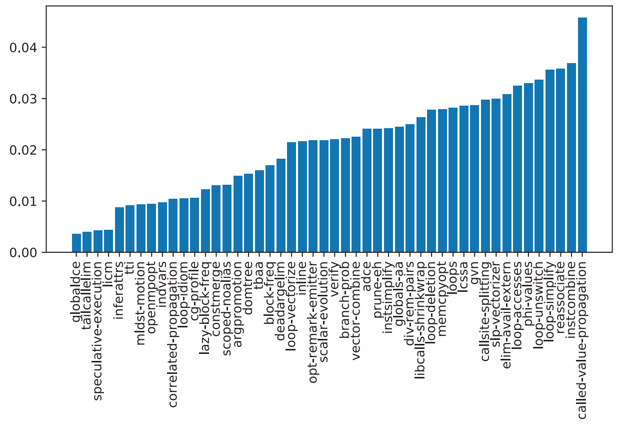

Figure 6: Ordered contributions of each optimization flag.

6.3 Potency of Optimization Flags

This section answers **RQ3** by measuring the potency of selected optimization flags (selected flags are fully listed at [1]). We report that for the POJ-104 dataset, the top-1 sequence S having the highest fitness score contains 49 flags. To measure their contribution, we first train CodeCMR using C source code and LLVM IR optimized using sequence S and record the baseline accuracy as *acc*. Then, we iteratively discard one optimization flag f from S and measure the augmentation effectiveness of using the remaining sequence with 48 flags. The accuracy drop reveals the contribution of flag f.

Fig. 6 orders the contribution of each flag in S. Overall, we interpret that no "dominating" optimization flags are found in this evaluation. In other words, we interpret that all these 49 flags manifest reasonable contribution to the model augmentation, and the top-10 flags contributes in total 34.38%. We thus make our first important observation w.r.t. **RQ3**:

> Instead of identifying one or few *dominating flags* that significantly contribute to enhancing code embedding, it is rather the formed sequence of optimization flags that is important.

This evaluation shows that a sequence of flags works together to produce high-quality IR, instead of one or a few "franchise players" that can largely outperform other flags. In other words, the GA process conducted by IRGen is *critical* to this research, because it offers a general way to construct such a sequence with modest cost.

We now consider the characteristics of the ten highest potency flags. We put these flags into three categories as follows:

Simplify an IR Statement. Optimization flags, including -dce, -early-cse, -reassociate, -bdce and -loop-deletion, simplify IR statements via various data flow or control flow analysis methods. For instance, -early-cse regulates IR statements by eliminating common subexpression eliminations, and -reassociate reassociates commutative expressions to support better constant propagation. In all, these optimization can make two syntactically distinct pieces of source code more similar in IR.

Make IR Statement Sequences Closer to Source Code. Flags, including `-mem2reg`, `-instcombine`, and `-dse`, can simplify the compiled IR code, making it *more similar to the source code.* For instance, `-mem2reg` promotes memory references to be register references. This prunes verbose memory usage in IR code and generates IR code similar with source code in terms of memory reference. Other flags, such as `-instcombine` and `-dse`, combine and simplify instructions to form fewer and more succinct instruction sequences that are generally closer to source code.

Simplify CFG. Optimization flags, including `-break-crit-edges`, `-simplifycfg`, and `-loop-rotate`, perform more holistic transformations to simplify the IR code CFGs. For instance, `-simplifycfg` performs dead code elimination and basic block merging by eliminating useless basic blocks and their associated LLVM PHI nodes. By regulating CFGs, these optimizations deliver similar IRs from two semantically similar but syntactically different source codes.

Our analysis identified numerous optimization flags that significantly improved the training IRs for embedding learning. This is intuitive, given that they launch transformation from different granularities. More importantly, we make the following observation to characterize important optimization flags:

Optimization passes that *simplify and regulate IR code*, either at the IR statement level or the CFG level, are generally desirable.

Many of these flags are often employed as cleanup passes to run after compiler inter- or intra-procedural optimizations.

Optimization Passes on GCJ

We further analyze the top-1 optimization sequences found by IRGen for the GCJ dataset. This top-1 sequence contains 50 flags. Given that the top-1 sequence found over POJ-104 contains 49 flags, we further measure the agreement of these two sequences by counting the number of flags appeared in both sequences. These two sequences agree on 28 flags. The top-3 flags in the POJ-104 sequence all exist in the intersection set, and five of the top-10 flags in the POJ-104 sequence exist in the intersection set. With respect to the (dis)agreement, we conduct a manual investigation and summarize our findings as follows:

Agreement. We found that these 29 overlapping flags primarily serve the purpose of simplifying and regulating IR code in different ways. For instance, `-reassociate` and `-deadargelim` simplify IR statements and delete dead arguments. The rest overlapping flags are used as utilities for other passes (e.g., `-lcssa` serves loop-oriented optimizations) or for analysis purposes (e.g., `-block-freq`). Overall, we interpret that code cleanup and regulation are generally applicable to enhancement of learning quality.

Disagreement. Given that POJ-104 test cases are relatively succinct, we find that IRGen tended not to select flags that focus on shrinking the code size. In contrast, GCJ contains much lengthy C/C++ code, whose derived LLVM IR code is even more lengthy. Hence, IRGen adaptively prioritizes more flags to reduce the size. Overall, we find that whether IRGen inclines to "shrink" code size is dataset-dependent. IR compiled from POJ is generally shorter than that of GCJ; therefore, it has fewer OOV issues and the need for shrinking is less frequent. GCJ has lengthy IR and more OOV IR statements; it is demanding to shrink IR to avoid OOV. In addition, GCJ features more floating number related programming tasks, and accordingly, floating number related flags, such as `-float2int`, are involved to turn floating numbers into integers and effectively reduce the #OOV cases. In contrast, POJ-104 dataset does not primarily involve floating number-related computations.

Overall, we interpret these results as promising: we found that over half of the optimization flags selected by IRGen (29; $^{29}/_{49}$ = 59.2% of all flags selected over POJ-104) were selected across two datasets of different complexity without using any pre-knowledge or in-depth program analysis. These overlapping flags further highlight the importance of cleaning up and regulating IR code to make neural embedding models more robust. Moreover, the 42.9% disagreement, to some extent, shows that IRGen enables the selection of more diverse flags adaptive to different datasets.

Table 5: Augment `ncc` over the POJ-104 dataset.

Model	MAP@R(%)
ncc	39.95
ncc-random	40.34
ncc-IRGen	56.07
ncc-w/o-inst2vec	54.19
ncc-w/o-inst2vec-random	55.10
ncc-w/o-inst2vec-IRGen	60.46

6.4 Generalization

As aforementioned, augmentation (including the fitness function; see Sec. 4.3) delivered by IRGen is *independent* from particular embedding model design. To answer **RQ4**, we demonstrate the generalization of IRGen by augmenting another popular neural embedding model, `ncc`. As previously described, `ncc` performs embedding over LLVM IR code. Therefore, we did not need to change the implementation of `ncc`. The `ncc` augmentation evaluation results are reported in Table 5. To compare with optimization sequences formed by IRGen, we also prepare a "baseline", denoting a sequence containing randomly selected 49 optimization flags. These two baseline results are reported in third and sixth rows.

As expected, augmentation notably improved the quality of `ncc` and `ncc-w/o-inst2vec`. Particularly, the MAP score of the latter model is improved to 60.46%, which is even higher than the scores achieved by two variants of Aroma. In contrast, we find that two random schemes show negligible enhancement. Overall, this evaluation indicates that IRGen delivers general augmentation for neural code embedding models, without consideration of the specific model designs.

6.5 Augmentation with Small Training Data

RQ5 considers a scenario where program embedding is inhibited by a limited amount of training data. We argue that this is a common situation, such as in vulnerability analysis where only limited vulnerable code samples are available. In these situations, we anticipate the feasibility and usefulness of extending training dataset and augmenting embedding models using optimized IR code.

Fig. 7 presents the evaluation results of augmenting a small training dataset. Specifically, we randomly selected 1% of the C source

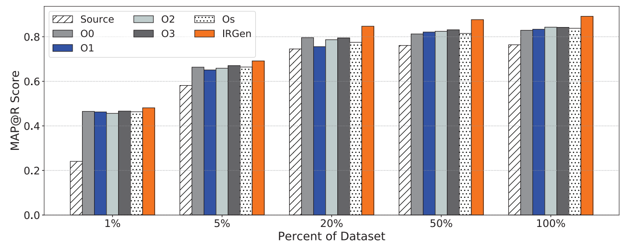

Figure 7: Performance with different size of training data.

codes from the POJ-104 training data to train CodeCMR and measured the resulting embedding accuracy, which was quite low (the first bar in Fig. 7; MAP = 24.08%). However, after using different standard optimization levels and optimization sequences selected by IRGen, the MAP accuracy increased to over 40%, almost *doubling* the original MAP score. In addition to an extreme 1% sample test, we also randomly selected 5%, 20%, and 50% of the POJ-104 training data and re-trained CodeCMR. As shown in Fig. 7, we further augmented the model for each subset by using the same standard optimization flags and flags selected by IRGen. We consistently achieved promising augmentation results. In particular, optimization flags found by IRGen outperformed all other augmentation schemes in all of these small training data settings. These intuitive and highly promising evaluation results indicate that IR code can be leveraged to significantly and reliably augment code embedding, particularly when only very limited data are available. Comparing with standard optimization levels, optimization sequences selected by IRGen can result in even higher enhancement.

7 DISCUSSION

Generalizability of Downstream Applications. This work focuses on a representative downstream task of code embedding — code clone detection. To clarify the generalizability behind "code clone": program embeddings can be used as the basis of a variety of downstream tasks like malware clustering, vulnerability detection, and code plagiarism detection. Holistically, many of these applications are based on deciding two software's *similarity*. Therefore, we view "code clone" detection as a core basis to assess those applications. Nevertheless, the augmentation pipeline of IRGen is generally *orthogonal* to a particular downstream task. We leave it as one future work to benchmark the augmentation capability offered by IRGen toward other important downstream applications, such as vulnerability detection and program repair. We envision to have consistently promising observations.

Conflicts Between Optimization Flags. gcc documents a set of constraints between optimization passes [32] wherein using two conflicting passes can result in compilation errors. However, to our knowledge, the LLVM compiler framework does not explicitly document any "conflicting" flags. We are also not aware of any compilation errors caused by using two conflicting LLVM passes. In cases where one flag has conflicts with other flags, such information can be explicitly encoded as constraints to validate generated optimization sequences.

Other Learning Methodologies. Production compiler frameworks like gcc and LLVM provide considerable optimization flags, forming a large search space in our research scenario. IRGen constitutes a GA-based approach to search for (near-)optimal optimization sequences. Our empirical evaluation shows that the proposed learning process is *sufficient* to identify high-performing optimization sequences. We note that there are more advanced (evolutionary) optimization algorithms available. In particular, the two fitness objectives could have been optimized separately (i.e., with multi-objective optimization). Also, from a holistic view, searching for optimization sequences is a Markov Decision Process (MDP). Complex MDPs (e.g., auto-driving) can be likely addressed with reinforcement learning (RL) techniques. Future work may explore using advanced deep RL models, which have achieved prominent success in solving real-world challenges in autonomous driving [28, 84] and video games [65].

8 RELATED WORK

We reviewed program embedding from various perspective in Sec. 2. The development of IRGen was inspired by existing works in search-based software engineering [39, 40] and search-based iterative compilation techniques [12, 22, 24, 48, 49]. In general, many tasks in software engineering require exploration of (optimal) solutions under a range of constraints and trade-offs between resources and requirements. To this end, metaheuristic algorithms, such as local search, simulated annealing (SA) [47], genetic algorithms (GAs) [95], and hill climbing (HC) [76] are frequently used to address these challenges. Typical applications include testing and debugging [38, 62, 63], verification [5–7, 23, 34], maintenance [69, 81], and software hardening [33, 33, 83].

A line of relevant and actively-developed research augments software obfuscation by combining obfuscation passes. Liu et al. [55] search for a sequence of obfuscation passes to maximize obfuscation effectiveness (and thus make software more secure). Amoeba [92] empirically demonstrated that combining obfuscation passes, though enhancing obfuscation potency, often carries high costs. Wang et al. [87] trained a reinforcement learning model to explore optimal obfuscation combinations by taking both cost and effectiveness into account. BinTuner [74] uses a guided stochastic algorithm to explore how combinations of compiler optimization passes can obfuscate software.

9 CONCLUSION

Existing neural program embedding methods are generally limited to processing of program source code. We present simple, yet effective, strategies to improve embedding quality by augmenting training datasets with compiler-generated IR code. In addition to use of default compiler optimization levels, we present IRGen, a search-based framework to find customized optimization sequences that produce IRs that can substantially improve the quality of learned embeddings. In evaluation, these models outperformed others trained using only source code or IR generated with default optimization combinations. Our study provides insights and guidance for users aiming to generate higher quality code embeddings.

ACKNOWLEDGEMENTS

This work was supported in part by CCF-Tencent Open Research Fund. We are grateful to the anonymous reviewers for their valuable comments.

REFERENCES

[1] [n.d.]. IRGen Website. https://sites.google.com/view/irgen.
[2] [n.d.]. One Source Code Compiled into Diverse IR Programs. https://sites.google.com/view/irgen/main-page/diverse-ir-code-example.
[3] [n.d.]. POJ104 Code Sample and LLVM IR. https://sites.google.com/view/irgen/main-page/motivation-code-example.
[4] [n.d.]. Proposals of Other Pipelines. https://sites.google.com/view/irgen/main-page/proposals-of-other-pipelines.
[5] Enrique Alba and Francisco Chicano. 2007. ACOhg: Dealing with huge graphs. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. 10–17.
[6] Enrique Alba and Francisco Chicano. 2007. Ant colony optimization for model checking. In *International Conference on Computer Aided Systems Theory*. Springer, 523–530.
[7] Enrique Alba and Francisco Chicano. 2007. Finding safety errors with ACO. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. 1066–1073.
[8] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.
[9] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).
[10] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices* 53, 4 (2018), 404–419.
[11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019).
[12] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.
[13] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. 1999. *Modern Information Retrieval*. ACM Press / Addison-Wesley.
[14] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. In *Proceedings of 4th International Conference on Learning Representations*.
[15] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Advances in Neural Information Processing Systems*. 3585–3597.
[16] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. *Advances in Neural Information Processing Systems* 31 (2018), 3585–3597.
[17] Sahil Bhatia and Rishabh Singh. 2016. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129* (2016).
[18] Karsten M Borgwardt and Hans-Peter Kriegel. 2005. Shortest-path kernels on graphs. In *Fifth IEEE international conference on data mining (ICDM'05)*. IEEE, 8–pp.
[19] Cristian Cadar. 2015. Targeted program transformations for symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 906–909.
[20] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, Vol. 8.
[21] Colin Campbell. 2002. Kernel methods: a survey of current techniques. *Neurocomputing* 48, 1-4 (2002), 63–84.
[22] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. 2010. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 448–459.
[23] Francisco Chicano and Enrique Alba. 2008. Finding liveness errors with ACO. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. IEEE, 2997–3004.
[24] Keith D Cooper, Devika Subramanian, and Linda Torczon. 2002. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing* 23, 1 (2002), 7–22.
[25] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.
[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018).
[27] S. H. Ding, B. M. Fung, and P. Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *IEEE S&P*.
[28] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An open urban driving simulator. In *Conference on robot learning*. PMLR, 1–16.
[29] Alhussein Fawzi, Horst Samulowitz, Deepak Turaga, and Pascal Frossard. 2016. Adaptive data augmentation for image classification. In *2016 IEEE international conference on image processing (ICIP)*. Ieee, 3688–3692.
[30] Steven Y Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. 2021. A survey of data augmentation approaches for nlp. *arXiv preprint arXiv:2105.03075* (2021).
[31] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[32] GCC. 2021. Options That Control Optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.
[33] Shadi Ghaith and Mel O Cinnéide. 2012. Improving software security using search-based refactoring. In *International Symposium on Search Based Software Engineering*. Springer, 121–135.
[34] Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 174–186.
[35] Palash Goyal and Emilio Ferrara. 2018. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems* 151 (2018), 78–94.
[36] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.
[37] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.
[38] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–12.
[39] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.
[40] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 1–61.
[41] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2019. Global relational models of source code. In *International conference on learning representations*.
[42] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[43] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.
[44] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive Code Representation Learning. In *EMNLP*. 5954–5971.
[45] Rie Johnson and Tong Zhang. 2017. Deep pyramid convolutional neural networks for text categorization. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 562–570.
[46] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *ICSE*. IEEE, 1073–1085.
[47] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.
[48] Peter MW Knijnenburg, Toru Kisuki, and Michael FP O'Boyle. 2001. Iterative compilation. In *International Workshop on Embedded Computer Systems*. Springer, 171–187.
[49] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. 2004. Fast searches for effective optimization phase sequences. *ACM SIGPLAN Notices* 39, 6 (2004), 171–182.
[50] Varun Kumar, Ashutosh Choudhary, and Eunah Cho. 2020. Data Augmentation using Pre-trained Transformer Models. *CoRR* abs/2003.02245 (2020).
[51] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 75–86.
[52] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. PMLR, 1188–1196.

[53] Zhenhao Li and Lucia Specia. 2019. Improving neural machine translation robustness via data augmentation: Beyond back translation. *arXiv preprint arXiv:1910.03009* (2019).
[54] Chen Liang, Jonathan Berant, Quoc Le, Kenneth D Forbus, and Ni Lao. 2017. Neural Symbolic Machines: Learning Semantic Parsers on Freebase with Weak Supervision. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 23–33.
[55] Han Liu, Chengnian Sun, Zhendong Su, Yu Jiang, Ming Gu, and Jiaguang Sun. 2017. Stochastic optimization of program obfuscation. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 221–231.
[56] Ruibo Liu, Guangxuan Xu, Chenyan Jia, Weicheng Ma, Lili Wang, and Soroush Vosoughi. 2020. Data Boost: Text Data Augmentation Through Reinforcement Learning Guided Conditional Generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*. Association for Computational Linguistics, 9031–9041.
[57] Raphael Gontijo Lopes, Dong Yin, Ben Poole, Justin Gilmer, and Ekin D Cubuk. 2019. Improving robustness without sacrificing accuracy with patch gaussian augmentation. *arXiv preprint arXiv:1906.02611* (2019).
[58] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.
[59] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In *FSE*.
[60] Pingchuan Ma and Shuai Wang. 2022. MT-Teql: Evaluating and Augmenting Neural NLIDB on Real-world Linguistic and Schema Variations. In *PVLDB*.
[61] Pingchuan Ma, Shuai Wang, and Jin Liu. 2020. Metamorphic Testing and Certified Mitigation of Fairness Violations in NLP Models. In *IJCAI*. 458–465.
[62] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.
[63] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 153–163.
[64] Junghyun Min, R Thomas McCoy, Dipanjan Das, Emily Pitler, and Tal Linzen. 2020. Syntactic data augmentation increases robustness to inference heuristics. *arXiv preprint arXiv:2004.11999* (2020).
[65] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
[66] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*.
[67] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. (2016).
[68] Kevin Musgrave, Serge J. Belongie, and Ser-Nam Lim. 2020. A Metric Learning Reality Check. In *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XXV (Lecture Notes in Computer Science, Vol. 12370)*. Springer, 681–699.
[69] Mark O'Keeffe and Mel O Cinnéide. 2008. Search-based refactoring for software maintenance. *Journal of Systems and Software* 81, 4 (2008), 502–516.
[70] Qi Pang, Yuanyuan Yuan, and Shuai Wang. 2021. MDPFuzzer: Finding Crash-Triggering State Sequences in Models Solving the Markov Decision Process. *arXiv preprint arXiv:2112.02807* (2021).
[71] Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-Symbolic Program Synthesis.
[72] Juraj Petrík. 2020. Google Code Jam programming competition. https://github.com/Jur1cek/gcj-dataset.
[73] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. In *ICML*.
[74] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the hidden power of compiler optimization on binary code difference: an empirical study *(PLDI)*.
[75] Marco Tulio Ribeiro, Carlos Guestrin, and Sameer Singh. 2019. Are red roses red? evaluating consistency of question-answering models. In *ACL*. 6174–6184.
[76] Bart Selman and Carla P Gomes. 2006. Hill-climbing search. *Encyclopedia of cognitive science* 81 (2006), 82.
[77] Ramprasaath R Selvaraju, Purva Tendulkar, Devi Parikh, Eric Horvitz, Marco Tulio Ribeiro, Besmira Nushi, and Ece Kamar. 2020. SQuINTing at VQA Models: Introspecting VQA Models With Sub-Questions. In *CVPR*.
[78] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *In Proceedings of the 24th USENIX Security Symposium*. 611–626.
[79] Connor Shorten and Taghi M Khoshgoftaar. 2019. A survey on image data augmentation for deep learning. *Journal of Big Data* 6, 1 (2019), 1–48.
[80] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification.. In *Advances in Neural Information Processing Systems (NeurIPS)*.
[81] Chris Simons, Jeremy Singer, and David R White. 2015. Search-based refactoring: Metrics are not enough. In *International Symposium on Search Based Software Engineering*. Springer, 47–61.
[82] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: Value-flow-based precise code embedding. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.
[83] Julian Thomé, Alessandra Gorla, and Andreas Zeller. 2014. Search-based security testing of web applications. In *Proceedings of the 7th International Workshop on Search-Based Software Testing*. 5–14.
[84] Marin Toromanoff, Emilie Wirbel, and Fabien Moutarde. 2020. End-to-end model-free reinforcement learning for urban driving using implicit affordances. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 7153–7162.
[85] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*. 350–360.
[86] treesitter-release [n.d.]. Tree-sitter. https://github.com/tree-sitter/tree-sitter. https://tree-sitter.github.io/tree-sitter/.
[87] Huaijin Wang, Shuai Wang, Dongpeng Xu, Xiangyu Zhang, and Xiao Liu. 2020. Generating Effective Software Obfuscation Sequences with Reinforcement Learning. *IEEE Transactions on Dependable and Secure Computing* (2020).
[88] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic neural program embedding for program repair. In *ICLR*.
[89] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embeddings for Program Repair. In *6th International Conference on Learning Representations*.
[90] Ke Wang and Zhendong Su. 2019. Learning blended, precise semantic program embeddings. In *POPL*.
[91] Shuai Wang and Zhendong Su. 2020. Metamorphic Object Insertion for Testing Object Detection Systems. In *ASE*.
[92] Shuai Wang, Pei Wang, and Dinghao Wu. 2017. Composite software diversification *(ICSME)*.
[93] Jason Wei and Kai Zou. 2019. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv preprint arXiv:1901.11196* (2019).
[94] Kilian Q Weinberger and Lawrence K Saul. 2009. Distance metric learning for large margin nearest neighbor classification. *Journal of machine learning research* 10, 2 (2009).
[95] Darrell Whitley. 1994. A genetic algorithm tutorial. *Statistics and computing* 4, 2 (1994), 65–85.
[96] Cameron R. Wolfe and Keld T. Lundgaard. 2019. Data Augmentation for Deep Transfer Learning. *CoRR* abs/1912.00772 (2019).
[97] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.
[98] Fangke Ye, Shengtian Zhou, Anand Venkat, Ryan Marucs, Nesime Tatbul, Jesmin Jahan Tithi, Paul Petersen, Timothy Mattson, Tim Kraska, Pradeep Dubey, et al. 2020. MISIM: An end-to-end neural code similarity system. *arXiv preprint arXiv:2006.05265* (2020).
[99] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. (2020).
[100] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. CodeCMR: Cross-Modal Retrieval For Function-Level Binary Source Code Matching. *Advances in Neural Information Processing Systems* 33 (2020).
[101] Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2021. Enhancing Deep Neural Networks Testing by Traversing Data Manifold. *arXiv preprint arXiv:2112.01956* (2021).
[102] Yuanyuan Yuan, Shuai Wang, Mingyue Jiang, and Tsong Yueh Chen. 2021. Perception Matters: Detecting Perception Failures of VQA Models Using Metamorphic Testing *(CVPR)*.
[103] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William E. Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. 2018. Neural Guided Constraint Logic Programming for Program Synthesis.
[104] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. *Advances in neural information processing systems* 28 (2015), 649–657.
[105] Alice Zheng and Amanda Casari. 2018. *Feature engineering for machine learning: principles and techniques for data scientists*. " O'Reilly Media, Inc.".
[106] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 10197–10207.