CONCORD: Clone-Aware Contrastive Learning for Source Code

Yangruibo Ding
Columbia University
New York, NY, USA

Saikat Chakraborty
Microsoft Research
Redmond, WA, USA

Luca Buratti
IBM Research
Yorktown Heights, NY, USA

Saurabh Pujar
IBM Research
Yorktown Heights, NY, USA

Alessandro Morari
IBM Research
Yorktown Heights, NY, USA

Gail Kaiser
Columbia University
New York, NY, USA

Baishakhi Ray
Columbia University
New York, NY, USA

ABSTRACT

Deep Learning (DL) models to analyze source code have shown immense promise during the past few years. More recently, self-supervised pre-training has gained traction for learning generic code representations valuable for many downstream SE tasks, such as clone and bug detection.

While previous work successfully learned from different code abstractions (*e.g.,* token, AST, graph), we argue that it is also essential to factor in how developers code day-to-day for learning general-purpose representation. On the one hand, human developers tend to write repetitive programs referencing existing code snippets from the current codebase or online resources (*e.g.,* Stack Overflow website) rather than implementing functions from scratch; such behaviors result in a vast number of code clones. In contrast, a deviant clone by mistake might trigger malicious program behaviors.

Thus, as a proxy to incorporate developers' coding behavior into the pre-training scheme, we propose to include code clones and their deviants. In particular, we propose CONCORD, a self-supervised pre-training strategy to place benign clones closer in the representation space while moving deviants further apart. We show that CONCORD's clone-aware pre-training drastically reduces the need for expensive pre-training resources while improving the performance of downstream SE tasks. We also empirically demonstrate that CONCORD can improve existing pre-trained models to learn better representations that consequently become more efficient in both identifying semantically equivalent programs and differentiating buggy from non-buggy code.

CCS CONCEPTS

• **Software and its engineering** → *Language features*; • **Computing methodologies** → **Knowledge representation and reasoning**.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598035

KEYWORDS

Source Code Pre-training, Code Clone, Bug Detection

ACM Reference Format:
Yangruibo Ding, Saikat Chakraborty, Luca Buratti, Saurabh Pujar, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. 2023. CONCORD: Clone-Aware Contrastive Learning for Source Code. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598035

1 INTRODUCTION

Self-supervised pre-training with BERT-like models [17, 19, 55, 70] (*i.e.,* a stack of Transformer-encoder layers) has achieved impressive success on many Software Engineering (SE) Tasks [7, 20, 30, 37, 45, 75]. The main advantage of these pre-trained models is that they do not require manual labels or active supervision. Instead, by leveraging huge existing code corpora (*e.g.,* GitHub), these models try to capture the statistical properties of source code, and use these correlations to "estimate" the code properties. Such self-supervised pre-training aims to embed the learned estimation to code representations and consequently assists with various downstream SE tasks like clone detection, bug finding, etc. during fine-tuning.

The early work in this line directly transplants models and pre-training strategies from the Natural Language Processing (NLP) field to large code corpora [7, 30, 45]. For example, Feng et al. [30] propose CodeBERT, one of the pioneers of pre-trained code models. They pre-train a BERT-like model on NL-PL pairs with two token-based objectives: masked language model (MLM) and replaced token detection. Later researchers integrated structural properties of code into the pre-training to better understand code syntax. For example, GraphCodeBERT [37] uses structural information such as abstract syntax trees (ASTs) [1] and data dependency graphs.

Limitations of Existing Work. Since previous work on source code modeling mainly focuses on lexical or syntactic properties of code (token, AST, and graph), they successfully learn the statistical properties at the granularity of language constructs. However, as pointed out by Hindle et al. [38] in their seminal paper "On the Naturalness of Software", as well as many years ago by Donald Knuth in "Literate programming" [50], the art of coding goes beyond

[1] GraphCodeBERT does not explicitly take ASTs as input, but its data flow graph is built on ASTs.

```
TFStatus Eval(TfContext*
context, TfNode* node){
  TfIntArray* output_shape=...;
  const int lookup_rank=...;
  TF_LITE_ENSURE(context,...);
  int k=0;
  size_t embedding_size=1;
  size_t lookup_size=1;
  for(int i=0;i<lookup_rank-1;i++,k++)
  {
    const size_t dim=...;
    lookup_size *= dim;
    output_shape->data[k]=dim;
  }
}
```

(a) Original Code

```
TFStatus model_evaluation(
TfContext* ctxt, TfNode* nd){
  TfIntArray* rand_name =...;
  const int rank=...;
  TF_LITE_ENSURE(ctxt,...);
  size_t lookup_sz=1;
  int z=0;
  size_t emb_sz=1;
  int x=0;
  while(x<rank-1){
    const size_t dim=...;
    lookup_sz *= dim;
    rand_name->data[z]=dim;
    x+=1;
    z+=1;}}
```

(b) Code Clone

```
TFStatus Eval(TfContext*
context, TfNode* node){
  TfeIntArray* output_shape=...;
  const int lookup_rank=...;
  TF_LITE_ENSURE(context, ...);
  int k = 0;
  int embedding_size=1;
  int lookup_size=1;
  for(int i=0;i<=lookup_rank-1;i++,k++)
  {
    const int dim=...;
    lookup_size *= dim;
    output_shape->data[k]=dim;
  }
}
```

(c) Buggy Clone-deviant

Figure 1: Motivation example: 1a is the original code; 1b is the clone of 1a; 1c is the deviant clone of 1a that accidentally introduces security bugs. This example is adapted from CVE-2022-23559 [18] of Tensorflow [25] project.

using programming constructs – it is also a human experience where developers follow some day-to-day coding practices.

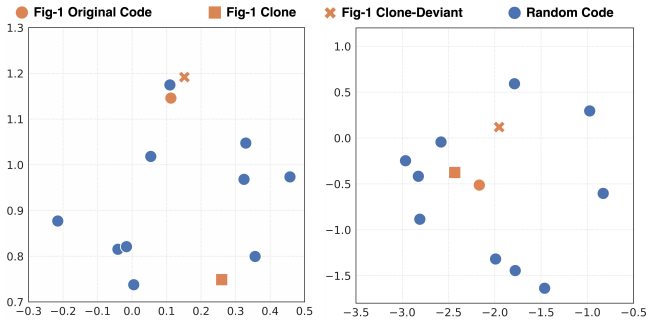
For example, developers tend to introduce code clones, often by common copy-paste practices, rather than implementing functions from scratch [5, 22, 52, 53]. Developers need to then adapt the clone to the new scope, such as reassigning the identifiers meaningful names [50] and adjusting control-/data-flow. Unfortunately, the introduction of subtle human errors is nearly unavoidable under such adaptation—*e.g.,* wrong identifiers [16, 53] and inconsistent control / data flows [43, 66].

Ignorance of common coding behaviors and likely human errors makes existing models inaccurately estimate code semantics in certain cases. Figure 1 shows such an example collected from a real-world project: 1a and 1b have identical functionalities, but they do not share similar tokens or structures since developers refactor it quite a bit (marked in green) after cloning. On the other hand, 1a and 1c are syntactically very similar, yet the subtle differences in type and comparison operator (marked in red) can potentially trigger integer overflow and out-of-array access issues in the latter.

Such examples can occur due to developers' common coding behaviors – copy-pasting and accidentally introducing bugs, respectively. We refer to the former as *code clones* (a.k.a. code variants) and the latter as *clone deviants*. Ideally, the pre-trained code model should well-capture the functional similarity of programs [20] rather than only textual overlaps, and encode 1a to be closer to 1b than 1c in the representation space.

To better understand whether existing code models encode programs as expected, we visualize the code embeddings of {1a, 1b, 1c} generated by GraphCodeBERT in Figure 2a. Disappointingly, as we see in Figure 2a, GraphCodeBERT ignores the functional equivalence of code, where the clone is far away from the original code. Also, it relies too much on syntactic similarities to represent programs, leading to the deviant and some random code in the wild being encoded closer to the original code. Other syntax-based pre-trained models operate similarly. In contrast, Figure 2b shows the desired embedding where original and its clone are closer in the representation space rather than its deviant.

In this paper, we aim to address the limitation of existing pre-trained code models by incorporating common coding behaviors



(a) Syntax-based Code Model **(b) Improved by CONCORD**

Figure 2: Visualization using Principle Component Analysis (PCA) [39] of Figure 1's code representations generated by GraphCodeBERT before (2a) and after CONCORD's improvement (2b).

directly into the pre-training framework with a focus on developers' code cloning practices. However, designing such a tool is challenging because:

i. *Lack of clone variants.* To expose diverse cloning practices commonly performed by developers, we need to train a model with various clone variants for each original code sample. However, manually collecting such data is almost impossible, especially at the scale of pre-training. Similarly, collecting deviants of clones is also non-trivial.

ii. *Learning clone functionalities.* CONCORD needs to learn that although two code snippets may not be structurally similar, they still may be functionally equivalent (*e.g.,* Figures 1a and 1b). Similarly, clone deviants can structurally resemble each other but functionally deviate (*e.g.,* Figures 1a and 1c).

Our Approach. To address the first challenge, we propose a novel clone-aware data augmentation scheme. We design multiple code transformation heuristics, imitating human developers' cloning behaviors. Using these heuristics, we generate two variants of each program in the dataset: a) Type-1, Type-2, Type-3, and Type-4 clone, as defined by [67, 68, 74]—these are essentially a near-miss or semantic clone with equivalent functionality and b) a clone deviant with contradictory functionality (by injecting subtle

bugs into the original program). To address the second challenge, we then learn clone functionalities with the augmented dataset. We encode the program and its clone variants with similar embeddings and differentiate the buggy deviants with very distinct embeddings. Concretely, we pre-train a language model, CONCORD, with a *contrastive learning* (CLR) objective [14, 15, 31] to maximize the similarity between the original code and its clone and minimize the agreement between the original code and its clone deviant.

Besides the CLR objective, CONCORD also learns the token representation of the original code using a masked language model (MLM) and the structural code properties with a new tree-structure prediction objective (LTSP) that can embed structural contexts into each token representation. Thus CONCORD is designed to learn statistical properties of PL constructs as well as developers' cloning behaviors in a single framework.

Results. We pre-train CONCORD with only 1.55 Gigabytes (GB) code[2] in a multi-lingual setting with C, C++, and Java for only 40k steps with batch size of 2048 samples during the first phase and 3k steps with batch size of 512 samples during the second phase (details in § 3.2). In contrast, CodeBERT and GraphCodeBERT are pre-trained for 100k-200k steps with batch size of 2048 samples on 20GB data.

Our results show that CONCORD achieves significantly better performance in clone detection and bug detection even with much cheaper training expense: CONCORD achieves 91.5% MAP@R on CodeXGLUE POJ-104 [57, 61] for clone detection, outperforming the best baseline by 5.5%, and reports the best F1 on three different bug detection benchmarks. We also explore CONCORD's extendability by adapting our approach on existing pre-trained code models. Our evaluation empirically shows that (1) CONCORD effectively improves existing code models' performance from 82.7% to 89.3% in MAP@R for clone detection, from 53.1% to 60.6% in F1 for bug detection, and from 67.6% to 69.7% in MRR for code search; and (2) CONCORD enhances these models' code representation with semantic-aware signals, pulling the clone towards the original code representation and pushing the deviant and irrelevant programs away from it. Figure 2b visualizes this enhancement.

To summarize, the main contributions of this paper are:

(1) We design a data-augmentation technique to automatically synthesize near-miss clones [67, 74], semantic clones and clone-deviants emulating a daily developer practice. Our evaluation reveals that CONCORD's data augmentation is more effective and controllable than the state-of-the-art deep-learning-based augmentation strategy.
(2) We propose CONCORD, a clone-aware pre-training framework, to effectively encode program semantics into code representations. Our evaluation shows that CONCORD outperforms existing code models in downstream SE tasks while requiring significantly less training resources (data and step size).
(3) We adapt our CONCORD approach to existing pre-trained code models and successfully improve their performance and code representation quality.
(4) We release CONCORD pre-trained model, data and code at: https://github.com/ARiSE-Lab/CONCORD_ISSTA_23.

[2]The size is measured when the code is raw text.

2 OVERVIEW

CONCORD is constructed based on BERT [19]. Figure 3 shows the overview of CONCORD. CONCORD contains three main stages: (1) data augmentation, (2) pre-training to learn code representations, and (3) task-specific fine-tuning.

Stage-1: Data Augmentation. The goal of our data augmentation is to pair each original sample with a clone as the semantically equivalent counterpart and a clone-deviant as the buggy counterpart, so we design heuristics to augment the dataset by transforming the original code into these counterparts. The generated clone is syntactically distinct from the original sample while preserving the semantics. The generated clone-deviant shares most tokens and a similar structure to the original sample but is injected with bugs. Such augmentation forces the model to contrast code semantics rather than only syntactic properties of code during training.

Stage-2: Pre-training to learn code representations. With augmented clones and clone-deviants, we pre-train CONCORD to learn code representations. We propose two-phase pre-training to comprehensively capture code properties, from general and statistical perspectives to structural and semantic features.

Phase-I applies the standard masked language model training (MLM) [19, 30, 37, 55], where we randomly mask code tokens and train the model to predict them back. This phase teaches the model to generally understand code so that it can pick the correct token from the vocabulary given the code context. Based on such a generic model, Phase-II conducts multi-task contrastive learning to better capture the code structures and semantics. Specifically, we introduce two more objectives besides MLM: local tree-structure prediction (LTSP) and contrastive learning (CLR). LTSP trains the model to construct the abstract syntax tree (AST) given the source code, empowering it with structural knowledge of programming languages. CLR trains the model to generate code representations according to the program semantics. This objective optimizes the model toward encoding code clones with similar representations and pushing buggy deviants far from the benign code in the representation space.

Such a two-phase pre-training also leaves more flexibility for CONCORD, as we can adapt the multi-task contrastive learning on top of the existing code models, extending CONCORD to distinct tasks, and modalities. We will study such extensibility in § 5.4.

Stage-3: Fine-tuning for Downstream Tasks. Finally, we load the pre-trained model as the encoder for the task-specific inputs and keep optimizing the encoder with task-specific objectives (*e.g.,* cross-entropy for classification) during the fine-tuning.

3 CONCORD

3.1 Stage-I: Data Augmentation

To augment a code sample, we first represent it as AST and transform the AST with applicable heuristics to generate its clones and clone-deviants.

3.1.1 Generating Clones. Real-world code clones are commonly classified into four categories [67, 68, 74]. Type-1 defines the exact clones that programs only differ in white space and comments. Type-2 defines the clones that are syntactically similar but may contain distinct identifier names, types, and literals. Type-3 clones

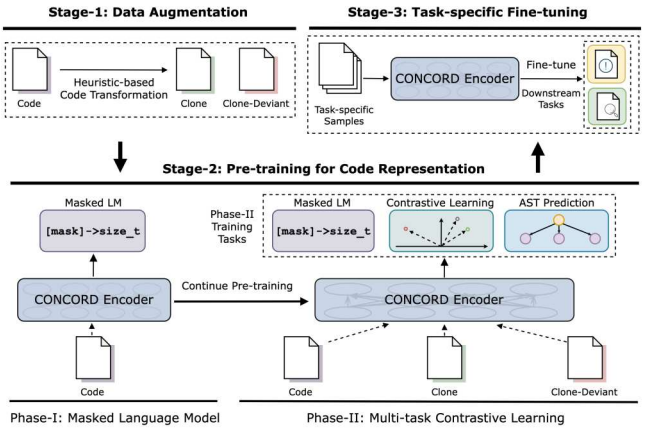


Figure 3: Overview of CONCORD.

are still syntactically similar but consider more complicated patterns than Type-2, such as statement-level modification. Type-4 defines the semantic clones that programs are functionally equivalent but might not be syntactically similar. In this work, we design heuristics to generate clones, imitating the features of Type-2/3/4 clones.

① **Identifier Renaming.** Motivated by Type-2 clone, we design identifier (variable or function name) renaming rules to generate clones. We imitate the developers' renaming behaviors with semantic-preserving strategies: (1) If the name is a single character, such as `i`, we rename it with another single-character variable, such as `j`. (2) If the name is camel case or snake case, we will split the name into sub-words by underscores or capitals and reconstruct the name by permuting these sub-words or randomly removing part of them: for example, if we have a name called `client_server`, we rename it as `server_client` or `client`.

We also consider the names with more randomness. We collect a vocabulary of all possible identifier names from our datasets, and to generate clones, we will rename the identifiers in the original program with randomly selected names from this vocabulary.

We apply the semantic-preserving first and if not successful, we apply the random renaming. We also make sure that our renaming does *not* change the execution behaviors of original programs.

② **Statement Rewriting.** We design rewriting rules to synthesize Type-3 clones, imitating the developers' behaviors of writing functionally equivalent statements with varied patterns [33–35]. We rewrite statements based on the three most frequent patterns: (1) We transform the conditional/ternary operators into If-Else statements. For example, we rewrite "`y = (x != 0) ? 2/x : 0;`" as "`if (x != 0) {y = 2/x;} else {y = 0;}`". (2) We rewrite the increment/decrement statements into other equivalent formats. For example, we rewrite "`y = x++;`" as "`y = x; x = x + 1;`". (3) We mirror the comparison statements without changing the control flow. For example, "`if (x > y)`" will be rewritten as "`if (y < x)`".

③ **Block Rewriting.** To introduce more complicated Type-3 clones, we also rewrite code blocks with two main categories: (1) Loops rewriting. We design transformations to rewrite for-loop(s) as while-loop(s) and vice versa. (2) If-Else block swapping. For example, given the program "`if (a < b) {A} else {B}`", we will rewrite it as "`if (a >= b) {B} else {A}`".

④ **Dead Code Insertion.** We create unreachable code blocks such as `if (False) {BLOCK}` and `while (2 < 0) {BLOCK}` and inject such blocks into the original code. For the statements in the `BLOCK`, we randomly pick sequential statements from the original programs and replicate them at the `BLOCK` part.

⑤ **Declaration/Initialization Statements Permutation.** To implement this, we first conduct dependency analysis to identify a set of local variables that do not depend on other values for initialization. Then we move their declaration statements to the beginning of the function and permute them. For example, "`int x; int y = 0;`" will be rewritten as "`int y = 0; int x;`".

We randomly pick one or several applicable transformations to generate the clone while maintaining the program behaviors

3.1.2 Generating Clone-deviants. Clone-deviants imitate the situation that code clones accidentally introduce flaws that maliciously change the program behaviors while sharing most tokens and similar syntax with the original, benign code. We design heuristics to generate such deviants based on the observations from a wide number of reported, real-world bug patterns [3, 12, 20, 21, 44, 46, 59, 60].

① **Operator Bugs.** We randomly replace the comparison operator with another of the same type to change the control flow and replace arithmetic operator to trigger unexpected program behaviors.

② **Data-type Bugs.** Wrong data types can trigger several security flaws (*e.g.,* integer overflow). We intentionally change the data types in the original code to inject such bugs, while ensuring the new code can still be compiled.

③ **Variable Bugs.** We induce such bugs with two approaches: (1) we perform scope analysis on the AST and replace a variable with another unexpected variable reachable in the same scope. (2) we remove the initialization expression of variables.

④ **Value Bugs.** We inject bugs by replacing a Boolean value with its opponent and an arithmetic value with random numbers.

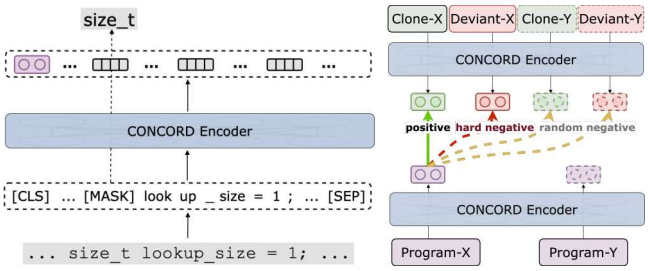
⑤ **Pointer Bugs.** To inject such bugs, we randomly remove the initialization expression during pointer declaration or set some pointers to `NULL`.

⑥ **Statement Bugs.** We randomly remove small condition-checking statements/blocks, which are typically used to check necessary preconditions before doing critical operations (*e.g.,* checking the index's validity before accessing an array)

⑦ **Function-call Bugs.** For a randomly selected function call, we add, remove, swap or assign `NULL` value to arguments, forcing the code to behave unexpectedly.

3.2 Stage-II: Pre-Training

3.2.1 Input Representation. As shown in Figure 4a, given the program, we parse it and flatten it as a sequence of code tokens $S = \{s_1, ..., s_m\}$. To alleviate the out-of-vocabulary concern of programming languages [46], we train a SentencePiece [51] subword tokenizer based on such flattened code token sequences with vocabulary size of 50,000. We use this tokenizer to divide m source code tokens into k sub-tokens ($m \leq k$). Similar to BERT, we prepend the special token `[CLS]` and append the special token `[SEP]` to the sub-token sequence $C = \{[CLS], c_1, ..., c_k, [SEP]\}$. Finally, CONCORD converts the pre-processed code sequence to vectors $V = \{v_{[CLS]}, v_1, ..., v_k, v_{[SEP]}\}$ with a token embedding layer.



(a) Input Representation & MLM (b) Contrastive Learning

Figure 4: Details of Pre-training Tasks

3.2.2 Phase-I: Learning Code Texts. In Phase-I, we pre-train the model with masked language model (MLM) [19] to capture the *naturalness* [38, 65] of code text (Figure 4a). Concretely, given the code sequence C, we randomly choose 15% of tokens (*e.g.,* `size_t` in 4a) and replace them with a special token `[MASK]`. We denote the set of masked token index as M, and the masked tokens as $\{c_m|m \in M\}$. The model will learn to encode the surrounding contexts (*e.g.,* `lookup_size=1` in 4a) of `[MASK]`s into their hidden states output by the model $\{h_m|m \in M\}$, and reconstruct the masked tokens conditioned on them. We compute the negative log likelihood of the masked tokens as the loss for this phase.

$$\mathcal{L}_{MLM} = \sum_{m \in M} -logP(c_m|h_m) \tag{1}$$

Equation 1 optimizes the model to minimize this negative loss during training, which maximizes the likelihood of original tokens before masking, given their surrounding contexts. It guides the model weights to fit the source code distribution in the wild. Consequently, the trained model will be able to produce code representations following such general distribution.

3.2.3 Phase-II: Learning Code Structures and Semantics. In Phase-II, we load the model weights from Phase-I and continue to learn the syntactic and semantic perspectives of programs.

Local Tree Structure Prediction (LTSP). To learn the code structure, we propose LTSP, teaching the model to predict the local ASTs given the code text. Concretely, we assign every code token a local AST label (Figure 5), $tt\#pt$, comprising of the type of the corresponding terminal node (tt) (*e.g.,* keyword, identifier), and the type of the immediate parent node (pt)(*e.g.,* for-statement, declaration). For example, in Figure 5, the token `size_t` is a primitive type in the variable initialization statement, so it will have a AST label of *primitive_type#variable_initialization*. All sub-tokens of a token will share the same label. Essentially, we are encoding the information of a 2-layer sub-tree into the AST-Label, and with such labels, the model can comprehensively capture the local dependencies, such as the connection with parent, children, and sibling nodes. We parse our dataset and exhaustively build the AST-Label vocabulary with all possible labels.

Formally, we pre-define the AST-Label sequence by parsing the code sequence, $T = \{[CLS], t_1, t_2, ..., t_k, [SEP]\}$, and we use this sequence as the ground-truth of the LTSP task. Similar to Phase-I, the model input is just the source code sequence (C), and the Transformer encoder will output a representation h_i for $c_i \in C$, and

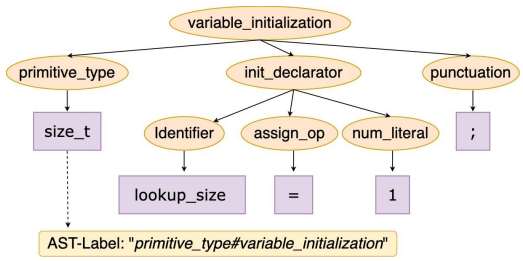


Figure 5: Building AST-Label for LTSP.

we train the model to predict the local AST type for *every* token based on h_i. We present the loss for LTSP as

$$\mathcal{L}_{LTSP} = \sum_{i} -logP(t_i|h_i) \tag{2}$$

Compared with other structure-based models [20, 37, 42], CONCORD has the advantage of not requiring structure input during the pre-traing and fine-tuning, but is aware of code structures and language grammars.

Contrastive Learning w/ Clones and Deviants. Contrastive learning has been proven to be effective in learning the semantic similarity of source code [6, 20, 36, 41, 75]. It is realized by optimization functions that maximize the representation similarity among semantically equivalent programs and enlarge the distances among semantically distinct or irrelevant code. We apply the contrastive learning objective to encode the clone-aware semantic signals into the code representations.

As shown in Figure 4b, for each program in the dataset (e.g., Program-X), the data augmentation generator (§3.1) creates a clone (Clone-X) as the positive counterpart and a clone-deviant (Deviant-X) as the negative counterpart. For each batch (Program-X, Y is one batch), CONCORD builds one positive pair (green arrow), one hard-negative pair (red, dashed arrow) and several random negative pairs using other in-batch samples (yellow, dashed arrows). Given all these pairs, we train the model to maximize the cosine similarity of the positive pair's representations and minimize the similarity of negative pairs. Formally, we have a minibatch of N programs, and for each program, we use the encoder output of `[CLS]` token to represent the whole sequence: $\mathbf{z} = h_{[CLS]}$. With the data augmentation, the minibatch is extended to N triplets of $(\mathbf{z}, \mathbf{z}^+, \mathbf{z}^-)$, where $\mathbf{z}^+$ corresponds to the generated clone, and $\mathbf{z}^-$ corresponds to the clone-deviant. We refer to the contrastive loss with hard negative samples from Gao et al. [31] and we adapt it to our scope as follows.

$$\mathcal{L}_{CLR} = -\log \frac{\mathbf{e}^{\text{sim}(\mathbf{z},\mathbf{z}^+)/\tau}}{\sum_{n=1}^{N} \left(\mathbf{e}^{\text{sim}(\mathbf{z},\mathbf{z}_n^+)/\tau} + \mathbf{e}^{\text{sim}(\mathbf{z},\mathbf{z}_n^-)/\tau}\right)} \tag{3}$$

In equation 3, we use cosine similarity as the $sim()$ function and τ is a parameter to scale the loss, and similar to [31] we use $\tau = 0.05$.

Similar to existing works [30, 37], we keep learning the code text using MLM during the second phase of pre-training, together with LTSP and contrastive loss. Therefore, the final loss function to optimize Phase-II pre-training is as follows, where $\lambda_1 = 1.0$, $\lambda_2 = 0.1$, $\lambda_3 = 1.0$ respectively.

$$\mathcal{L}(\theta) = \lambda_1 \cdot \mathcal{L}_{MLM}(\theta) + \lambda_2 \cdot \mathcal{L}_{LTSP}(\theta) + \lambda_3 \cdot \mathcal{L}_{CLR}(\theta) \tag{4}$$

3.3 Stage-III: Fine-Tuning

We apply the standard transfer learning strategy to the pre-trained model for concrete downstream tasks: we load the pre-trained model as the encoder to generate generic code representations and keep optimizing the model with supervised fine-tuning. We consider semantic-clone detection and bug detection as CONCORD's main applications.

Semantic Clone Detection. Detecting semantic clones is significant for software maintenance [48, 54] yet very challenging in practice, since the token overlap among semantic clones is quite limited and syntactic structures are not similar as well. This task evaluates the model's capacity of retrieving semantic clones: given a program as query and a set of random programs as candidates, the model needs to identify the semantic clones of the query out of thousands of candidates.

Bug Detection. Human errors are the main causes for software flaws. For example, when developers adapt clones into a new scope, small errors, such as wrong identifier names and operators, are accidentally introduced [53, 66], and such bugs are only a few tokens away from the benign version. Similarly, it is challenging for models to identify bug-fixing patches as benign, if developers only repair a few tokens [21]. Without attending to such human behaviors, models struggle with false positives and false negatives [47, 63]. CONCORD alleviates such concerns by incorporating these behaviors into the pre-training, synthesizing clone-deviants as the hard negative samples and forcing the model to differentiate bugs and benign code that are syntactically similar.

We evaluate the model's capacity of detecting software bugs at function-level. Specifically, given a code function as input, the model needs to classify it as buggy (positive) or benign (negative).

4 EXPERIMENTAL SETUP

4.1 Pre-Training Dataset

We collect our pre-training corpus from open-source GitHub projects. We rank Github repositories by the number of stars and focus on the most popular ones. After filtering out forks from existing repositories, we collect the dataset for each language from top-100 repositories. We only consider the ".c", ".cpp" and ".java" files for C, C++, and Java repositories respectively. Similar to the existing datasets [40], we extract the code functions/methods from the code files. The raw datasets for C, C++, and Java are of size 662MB, 330MB, and 556MB respectively. We use the state-of-the-art, multi-lingual AST parser, Tree-sitter [73] to parse the source code.

4.2 Model Configuration

CONCORD applies a standard $\text{BERT}_{\text{BASE}}$ architecture [19] with 12 layers of Transformer-encoder, and each layer has 12 attention heads and the hidden dimension is 768. The maximum sequence length is 512 BPE tokens, and the longer sequence will be truncated. As samples in our datasets are function-level code, truncation does not frequently happen. Our experiments are conducted on 2 × 24GB NVIDIA GeForce RTX-3090 GPUs. For Phase-I, we pre-train CONCORD for 40k batches with a batch size of 2048, and we use the learning rate of 5e-4. 1e-4 to 5e-4 is the common range for learning rates of MLM-based code models [7, 20, 30, 37, 45], and we follow [30] to select 5e-4. For phase-II, we further pre-train CONCORD for 3k batches with a batch size of 512, and we use the learning rate of 5e-5. For all the fine-tuning tasks, CONCORD uses the learning rate of 8e-6. Learning rates typically decrease for later phases [20, 30, 37], so CONCORD follows the same design. We use Adam optimizer [49] with the linear learning rate decay. CONCORD is implemented mainly with Pytorch [23] and Huggingface [28] libraries.

4.3 Evaluation Datasets & Metrics

Semantic Clone Datasets. We consider two datasets for the semantic clone detection: CodeXGLUE-POJ104 [57, 61] and CodeNet-Java250 [64]. CodeXGLUE-POJ104 contains 104 programming challenges, and each has 500 C/C++ solutions submitted by different students. CodeXGLUE [57] reconstruct it as a public benchmark by splitting the dataset into Train (64 challenges), Dev (16 challenges), and Test (24 challenges) sets, making sure that there is no overlapped challenges between any two sets. CodeNet-Java250 contains 250 Java programming challenges from online judge websites, and each has 300 solutions from programmers. It splits the datasets into Train (125 challenges), Valid (62 challenges), and Test (63 challenges) without overlapped challenges. The detailed statistics of these two datasets can be found in Table 1.

We notice that some existing works are evaluated on CodeXGLUE-BigCloneBench [72]. We do not use this benchmark, because we find that the labels are rather noisy and inaccurate [3,4].

Metrics of Clone Detection. Both CodeXGLUE [57] and CodeNet [64] are using MAP@R (Mean Average Precision @ R)[5], so we follow such a design. Average precision at R is a common metric to evaluate the quality of information retrieval; it measures the average precision scores of a set of the top-R clone candidates presented in response to a query program. The "R" for CodeXGLUE is 499 as it has 500 solutions for each challenge, and we do not consider the code itself as a clone, and for CodeNet is 299, since it has 300 solutions for each problem.

Table 1: Details of downstream tasks datasets.

Task	Dataset	Lang.	Train	Valid	Test
Clone Detection	CXG-POJ104	C/C++	32,000	8,000	12,000
	CN-Java250	Java	37,500	18,600	18,900
Bug Detection	REVEAL	C/C++	15,867	2,268	4,535
	D2A	C/C++	4,644	597	619
	CXG-Devign	C/C++	21,854	2,732	2,732

Bug Datasets. We choose three public datasets: REVEAL (RV) [12], D2A [78], and CodeXGLUE-Devign (CXG-DV) [57, 79]. Chakraborty *et al.* curated REVEAL, imitating the real-world scenario that bugs are always rare compared to the normal programs, so it ends up with the ratio of the buggy to benign samples being roughly 1:10. D2A is a balanced dataset focusing on bug-fixing commits and annotates the previous version of modified functions as buggy and the fixed version as benign. CodeXGLUE-Devign is another balanced dataset introduced by Zhou et al. [79], and CodeXGLUE reconstructs the

[3]Corresponding GitHub issue: https://github.com/microsoft/CodeXGLUE/issues/93
[4]Corresponding GitHub issue: https://github.com/microsoft/CodeXGLUE/issues/99
[5]https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval)#Mean_average_precision

dataset as a public benchmark, fixing the train/valid/test splits so that all models can be evaluated with the same splits.

Metrics of Bug Detection. REVEAL is a super imbalanced dataset, so following the design of the original paper, we use Precision, Recall, and F1 as the evaluation metrics. D2A and Devign are balanced datasets, so we use Accuracy and F1.

5 EVALUATION

CONCORD aims to learn more meaningful code representations by incorporating common coding behaviors directly into the pre-training framework, focusing on code clones and bugs introduced by these behaviors. In this section, we present the evaluation results and analysis. In particular, we ask the following five RQs:

- **RQ1:** How effective is CONCORD *w.r.t.* state-of-the-art baselines for (1) clone detection and (2) bug finding?
- **RQ2:** How effective is CONCORD's data augmentation *w.r.t.* state-of-the-art deep-learning-based data augmentation?
- **RQ3:** Can CONCORD's LTSP pre-training objective help to learn better code presentations with code structures?
- **RQ4:** Can CONCORD's pre-training improve existing code models for downstream tasks?
- **RQ5:** Can CONCORD learn more meaningful representations and better identify semantic similarity of programs than existing code models?

5.1 RQ1. Comparing CONCORD to Baselines

5.1.1 RQ1-A. Semantic Clone Detection. We present the baselines and results of semantic clone detection in this section.

Baselines. We choose the best-performing pre-trained models reported by CodeXGLUE-POJ104 [57] : RoBERTa [55], CodeBERT [30], and GraphCodeBERT [37]. These pre-trained models have already been proven to be more effective than previous work [57], such as SourcererCC [69] and Aroma [58], so we no more include these older approaches in our results. We also consider two contrastive-learning-based code models: Corder [6][6] and DISCO [20][7]. We directly compare the originally reported results from the paper (*e.g.,* Corder) or the benchmark (*e.g.,* CodeBERT and RoBERTa), if available. Otherwise, we will fine-tune the baselines.

Results. The results are shown in Table 2. CONCORD achieves **91.5%** MAP@R for CodeXGLUE-POJ104 and **86.5%** for CodeNet-Java250. Even if pre-trained with a very small dataset (7.5% size of GraphCodeBERT's dataset), CONCORD is still a clear winner with a significant margin. Comparing CONCORD with syntax-based baselines (*i.e.,* CodeBERT, GraphCodeBERT, and RoBERTa), the results reveal the effectiveness of contrastive learning: learning to contrast code functionalities can improve the model's performance while reducing the training cost. Interestingly, we found that CONCORD can also outperform other contrastive-learning-based approaches, which empirically proves the effectiveness of CONCORD's data augmentation and multi-task pre-training. We will conceptually discuss the different design options among these contrastive-learning-based code models in Related Work (§ 6)

[6]Corder has several variants, and we reported Corder-Transformer results according to the original paper, as CONCORD applies the same model architecture.

[7]For a fair comparison, we re-implement and pre-train DISCO with the same setting as CONCORD (*e.g.,* library versions). The evaluation shows the new implementation slightly improves the original version [20] with acceptable error bound.

Table 2: MAP@R (%) Results of Semantic-clone Detection

Models	Data Size	CodeXGlue-POJ104	CodeNet-J250
Corder*	1.9 GB	72.0	-
DISCO	1.8 GB	82.5	76.5
Corder*	9.3 GB	84.1	-
RoBERTa	160 GB	76.7	75.5
CodeBERT	20 GB	82.7	81.1
GraphCodeBERT	20 GB	86.7	84.3
CONCORD	**1.5 GB**	**91.5**	**86.5**

*Corder applies a slightly different setting from the commonly used CodeXGLUE benchmark on POJ-104: it randomly samples 50 programs for each problem in POJ-104, resulting in 5,200 samples, and reports the mean average precision (MAP). In the strict Corder's setup, CONCORD reports 90.4 MAP in POJ-104.

5.1.2 RQ1-B.Bug Detection. We present the baselines and results of bug detection in this section.

Baselines. We compare CONCORD with Transformer-based pre-trained models containing a similar number of Transformer layers (12 layers in total), since Transformer models with more layers always significantly outperform those with fewer parameters [19, 27, 55, 76]. Thus, we consider either 12-Layer Transformer-Encoder models or 6-Layer Transformer-Encoder-Decoder models.Again, for released pre-trained models, we conduct full experiments on all three benchmarks, and for the others, we directly take their reported results in the original paper.

Table 3: Results of Bug Detection.

Model	Data Size	RV			D2A		CXG-DV	
		P.	R.	F1.	Acc.	F1.	Acc.	F1.
DISCO	1.8 GB	47.9	46.4	47.2	60.2	57.9	**64.2**	**58.5**
CodeBERT	20 GB	47.0	47.7	47.3	59.2	63.6	63.4	53.1
GraphCodeBERT	20 GB	**55.9**	39.9	46.6	61.0	66.1	62.9	56.3
PLBART	576 GB	44.9	41.6	43.2	57.0	61.2	62.5	57.9
CodeT5	>20 GB	47.1	46.7	46.9	58.9	56.1	62.8	58.3
CONCORD	**1.5 GB**	47.8	**49.3**	**48.6**	**62.1**	**67.1**	63.7	58.3

Results. From Table 3, we can see that, pre-trained with clone-aware signals, CONCORD is effective at reducing the false positives and false negatives: CONCORD reported better F1 for all three benchmarks than those pre-trained models focusing on code syntax. In particular, when all the baselines are reporting many false negatives in REVEAL, due to the rareness of positive samples in it, CONCORD achieves significantly higher recall than competing models, even if these baselines are pre-trained with significantly more samples. This result empirically reveals that CONCORD's augmented deviants help the model reduce the confusion when differentiating buggy from benign code, even if they are sometimes syntactically similar. Interestingly, we notice DISCO [20] performs slightly better in the CodeXGLUE benchmark, which also demonstrates the effectiveness of contrastive learning for code. However, CONCORD outperforms DISCO in all other bug detection benchmarks as well as the clone detection tasks, proving CONCORD is in general more effective at learning code semantics than DISCO.

Result-1: Even if pre-trained with **significantly less** data, CONCORD **outperforms** the state-of-the-art baselines that are not trained with the clone awareness in downstream tasks.

5.2 RQ2. Effectiveness of CONCORD's Data Augmentation

Data augmentation is the key to the success of contrastive learning models [14, 31]. It integrates strong bias regarding the semantic similarity of data into the model, since, during training, the model learns to maximize/minimize the similarity of the original sample with its positive/negative counterparts, which are completely generated by the pre-defined data augmentation strategy. In this section, we compare CONCORD's data augmentation with the state-of-the-art deep-learning-based approach.

The state-of-the-art deep-learning-based data augmentation [31, 36] for code relies on the model's randomness to generate positive/negative counterparts implicitly, which is difficult to control and interfere with domain knowledge. Differently, CONCORD proposes to explicitly augment code datasets with carefully crafted heuristics (§ 3.1), by imitating the developers' behaviors of cloning existing code.

Baseline and Setup. SimCSE [31] is the state-of-the-art contrastive learning framework for natural languages, and it has been proven to be effective in learning better code representations [36]. SimCSE leverages "dropout" [71], a deep-learning technique initially proposed to avoid model over-fitting, to generate positive samples. Concretely, the dropout mechanism randomly disables certain neurons in the neural network following Bernoulli distribution, and the randomness of each neuron is independently seeded. Therefore, SimCSE passes the *same* sample twice through the model and with dropout, it will get two slightly different embeddings, which will be regarded as semantic equivalent pair. SimCSE builds negative samples using randomly sampled data points within the same batch.

To conduct the comparison, we augment the original dataset with SimCSE data augmentation rather than CONCORD's, and retrain the model with the SimCSE-augmented dataset. We report results from the original CONCORD and CONCORD-SimCSE (*i.e.,* SimCSE for short in Table 4) on downstream tasks.

Table 4: Performance of CONCORD with SimCSE-augmented dataset and CONCORD-augmented dataset.

Task	Clone Det.		Bug Detection						
Dataset	P104	J250	RV			D2A		CXG-DV	
Metric	MAP@R		P.	R.	F1	Acc	F1	Acc	F1
SimCSE	88.5	**86.5**	**50.9**	46.5	**48.6**	61.3	67.0	63.6	56.3
CONCORD	**91.5**	**86.5**	47.8	**49.3**	**48.6**	**62.1**	**67.1**	**63.7**	**58.3**

Results and Analysis. The results are shown in Table 4. CONCORD performs marginally better than CONCORD-SimCSE. Besides, CONCORD's augmentation is easily controlled by heuristic designs while SimCSE completely relies on dropout randomness. CONCORD's clone-aware augmentation is a proof-of-concept of integrating developers' cloning behaviors into code representation and using the same philosophy, we could propose heuristics that align with other human requests, while SimCSE does not have such flexibility.

Result-2: CONCORD's data augmentation proposes carefully crafted heuristics to imitate developers' cloning patterns and bugs, and it reports comparable performance with state-of-the-art deep-learning-based data augmentation in clone detection and bug finding tasks.

5.3 RQ3. Effectiveness of CONCORD's LTSP Pre-Training Objective

CONCORD proposes a new pre-training objective, LTSP, to guide the model to learn the code syntax during the pre-training. In this RQ, we study the effectiveness of this new pre-training objective.

Setup. To conduct a strict comparison, we pre-train a CONCORD variant by removing the LSTP objective but keeping all other settings the same as the main model. As we mentioned in § 3.2.3, LTSP objective does not require any parsing or pre-processing on the source code during the fine-tuning for downstream tasks. Therefore, we evaluate CONCORD-without-LTSP using exactly the same fine-tuning data and strategies as discussed in § 5.1 and compare its performance with the main model.

Table 5: The comparison of CONCORD's performance between with and without LTSP objective during pre-training.

Task	Clone Det.		Bug Detection						
Dataset	P104	J250	RV			D2A		CXG-DV	
Metric	MAP@R		P.	R.	F1	Acc	F1	Acc	F1
w/o LTSP	91.3	86.1	**48.7**	46.9	47.8	60.0	56.1	63.4	56.1
w/ LTSP	**91.5**	**86.5**	47.8	**49.3**	**48.6**	**62.1**	**67.1**	**63.7**	**58.3**

Results. In Table 5, we conclude that removing LTSP hurts the model's performance in general, degrading clone retrieval performance slightly and F1 of bug detection significantly. The results empirically reveal the necessity of learning code structures for better code representations and the effectiveness of our proposed LTSP objective in learning such information.

Result-3: CONCORD's LTSP pre-training objective effectively improves the model's performance in downstream tasks by guiding the model to learn code structures.

5.4 RQ4. Applying CONCORD to Existing Pre-Trained Code Models

As we introduced in §3.2.3, CONCORD's two-phase pre-training strategy leaves the flexibility of replacing CONCORD's first phase with other BERT-like pre-trained code models. In this RQ, we explore CONCORD's extensibility by applying it to existing syntax-based code models. We expect that CONCORD is able to improve the performance of pre-trained code models, and meanwhile, these existing models can help CONCORD to extend to more tasks.

Setup. We choose the two most popular models for experiments: CodeBERT and GraphCodeBERT. Specifically, we load the pre-trained weights from CodeBERT and GraphCodeBERT to initialize the Transformer-encoder layers within CONCORD architecture, and further train these models with CONCORD's multi-task second phase. We name these variations as CONCORD-CB (initialized with CodeBERT) and CONCORD-GCB (initialized with GraphCodeBERT). We fine-tune these models on the same downstream tasks discussed in § 5.1.

Another benefit of using CodeBERT and GraphCodeBERT is that they are pre-trained with bi-modal datasets, where natural

language and code both exist, so that we could extend CONCORD to bi-modal downstream tasks with these models.

Extending CONCORD to Bi-modal Task: Code Search. Code search is the task of retrieving programs that match the natural language description, from tens of thousands of candidates. It requires the model's capacity of capturing semantic similarity of texts (between natural languages and code) rather than only syntactic similarity. Empowered by the bi-modal pre-trained models, we expect CONCORD to perform well in code search, as its semantic-aware contrastive learning aligns well with this task.

CodeSearchNet [40] is the most popular benchmark for code search. It pairs each function with a natural language description, relating to the code comments. CodeSeachNet does not have datasets for C and C++, so we choose to evaluate CONCORD on its Java dataset. Again, we take the reconstructed benchmark from CodeXGLUE, which has 164,923 / 5,183 / 10,955 samples for train / valid / test splits respectively. We follow the benchmark's design, using MRR (mean reciprocal rank) as the evaluation metric, and use the originally reported scores[8] for comparison.

Results. Table 6 summarizes the comparison between the existing pre-trained models and their CONCORD enhanced variants. We could see that CONCORD variants win on all tasks with a clear margin. These results empirically prove that CONCORD can significantly improve the performance of existing code models. Also, the code search result also reveals that CONCORD can be extended to multi-modal tasks and perform well by loading a multi-modal pre-trained model.

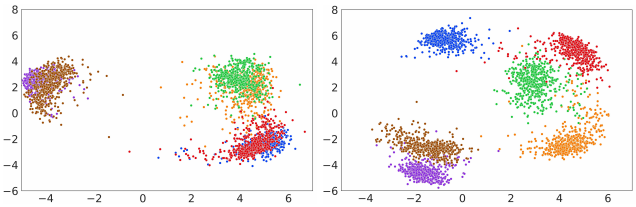
Table 6: Results of applying CONCORD to existing pre-trained code models.

Task	Clone Det.		Bug Detection					Search
Dataset	P104	J250	RV	D2A		CXG-DV		CSNet
Metric	MAP@R		F1	Acc	F1	Acc	F1	MRR
CodeBERT	82.7	81.1	47.3	59.2	63.6	63.4	53.1	67.6
CONCORD-CB	**89.3**	**85.1**	**48.7**	**61.5**	**65.5**	**64.6**	**60.6**	**69.7**
GraphCodeBERT	86.7	84.3	46.6	61.0	66.1	62.9	56.3	69.1
CONCORD-GCB	**91.6**	**84.8**	**47.2**	**62.3**	**70.0**	**64.2**	**60.1**	**70.5**

Result-4: CONCORD framework is **flexible** to be adapted with existing pre-trained code models and **improve their performance** in downstream tasks.

5.5 RQ5. Semantic-Aware Code Representations

When we further study and visualize the code representations, we found that, even after fine-tuning, existing code models still struggle to encode programs based on semantic similarity. We take six coding challenges from the test split of POJ-104 dataset, and generate these samples' representations using the models *fine-tuned* on the POJ-104 to study their distributions. For better comparison and visualization, we use principle component analysis (PCA) [39] to reduce representations' dimensions and plot the 2-d data points in Figure 6: 6a is generated by CodeBERT, and 6b is generated by CONCORD-CodeBERT, and each color represents one coding challenge. Clearly, CodeBERT's representations of distinct coding challenges significant overlap, which makes it difficult for the model to retrieve the semantic clones and tend to make mistakes. In contrast, CONCORD-CodeBERT's representations have clear boundaries among different clusters, so retrieving clones of the same code challenging becomes efficient.



(a) Existing Code Model (b) CONCORD

Figure 6: Visualization of data points in clone detection using PCA. Different colors represent distinct coding challenges in POJ-104

Motivated by this finding, we study the general quality of code representations of existing pre-trained code models and their enhanced variant with CONCORD. We design experiments to compare the similarity of code representations between the original code and its clone / clone-deviant / random code. Ideally, the model with decent awareness of code semantics should generate code representations following the order of Sim[Original, Clone] > Sim[Original, Deviant] > Sim[Original, RandomCode], where Sim[] is the cosine similarity of two vectors, since the clone is semantically equivalent to the original code, and clone-deviant is buggy but textually more similar than other random code in the wild.

Setup. We randomly sampled 10,000 samples from the held-out dataset to ensure the programs are from real developers, and CONCORD models have never seen these code during training. Then we run CONCORD's data augmentation tool to pair each sample, x, with a clone, x^+, and a deviant, x^-. We end up with 10k triplets of (x, x^+, x^-) as the dataset of this section. Then we encode every program in the dataset with different pre-trained models we are studying. For the convenience of discussion, we define the set of original programs as $X = \{x_i | x_i \text{ is original code}\}$, where $i \in [0, 10,000)$, and augmented programs as $\hat{X} = \{\hat{x}_j | \hat{x}_j \text{ is clone or deviant}\}$, where $j \in [0, 20,000)$, since one original program is augmented by two counterparts (clone and deviants). Finally, we exhaustively compute the cosine similarity of every possible pair of $\{x_i, \hat{x}_j\}$. All the experiments are conducted with the *zero-shot* setting.

Metrics. We use two metrics to evaluate this RQ. First, we compute average pair-wise similarity to measure the similarity of original code and its clone, deviant and other random code within the same dataset. Specifically, we compute

$$\text{Avg. Clone Pair Similarity} = \mathbb{E}_{x \in X} SIM(x, x^+)$$
$$\text{Avg. Deviant Pair Similarity} = \mathbb{E}_{x \in X} SIM(x, x^-)$$
$$\text{Avg. Random Pair Similarity} = \mathbb{E}_{x \in X} \mathbb{E}_{\hat{x} \in \hat{X}/\{x^+, x^-\}} SIM(x, \hat{x})$$

Second, we evaluate the models' capacity of retrieving semantically similar code using the learned code representations. For each $x \in X$, we retrieve its Top-1 similar code from $\hat{X}$, and we check how often the retrieved code is x's clone (*i.e.,* x^+), deviant (*i.e.,* x^-), and irrelevant code respectively.

[8]Reference: https://github.com/microsoft/CodeBERT/tree/master/GraphCodeBERT/codesearch

Table 7: Code representation similarities between the original code and clone / clone-deviant / irrelevant code. Pair-wise Similarity is compared horizontally (↔) within the same model; Top-1 similar is compared vertically (↕) across models.

Model	Avg. Pair-wise Similarity (↔)			Top-1 Similar @ 20K (↕)		
	Clone	Deviant	Rand.	Clone	Deviant	Rand.
CodeBERT	99.3	**99.8**	97.5	12.1	85.8	2.1
CONCORD-CB	**95.5**	25.8	2.9	**97.1**	**1.0**	**1.9**
GraphCodeBERT	90.5	**97.2**	72.9	9.2	87.4	3.4
CONCORD-GCB	**95.7**	22.7	2.2	**97.6**	**0.8**	**1.6**

Results. Results are in Table 7. Unfortunately, both CodeBERT and GraphCodeBERT assign higher similarity score to (original, deviant) pairs than (original, clone) pairs, and purely MLM-based CodeBERT assigns very high scores even to (original, random) pairs. These results show that existing code models have a weak sense of semantic similarity of source code, as they always encode syntactically similar code as closer representations. On the contrary, after training these models with CONCORD's approach, their representations become more aware of semantic similarity: both CONCORD-CodeBERT and CONCORD-GraphCodeBERT regard (original, deviant) pairs as less similar than (original, clone) pairs, and give (original, random) pairs the least similarity scores. Similarly, syntax-based models tend to wrongly retrieve the deviant as the top-1 similar program in most cases, while CONCORD can always pinpoint the semantically equivalent programs from tens of thousands of candidates.

Result-5: CONCORD effectively improves syntax-based models to learn better code representations for identifying semantic similarity.

6 RELATED WORK

Self-supervised Pre-training for Code. Researchers have been passionate about pre-training Transformer models for source code. There are three main architectures for existing models: Encoder-only [6, 7, 20, 30, 37, 45, 75], Decoder-only [4, 26, 77], and Encoder-decoder [1, 11, 29, 36, 62]. Encoder-only models are commonly pre-trained with cloze tasks (*e.g.,* masked language model) and sequence understanding tasks (*e.g.,* next statement prediction). Decoder-only models are mostly trained with autoregressive, left-to-right language model (LM) Encoder-Decoder models are pre-trained with different tasks including denoising autoencoding to reconstruct the wrongly permuted tokens [1], predicting missing identifiers [76], recovering method names [62], etc. In recent years, with the rapid development of computing devices, such as GPUs and TPUs, researcher also shed light on the incredible power of extremely large Transformer models (up to hundreds of billions of parameters) for understanding and generating code [4, 26, 29, 32].

Contrastive Learning for Code. Most recently, self-supervised contrastive learning has gained a lot of interest in learning source code representations [6, 13, 20, 36, 41, 56, 75]. Contrastive learning models for source code typically include two steps: (1) augmenting datasets with semantically equivalent programs as positive samples and contradictory programs as negative samples. (2) learning to maximize the vector similarity of equivalent samples and minimize the similarity of contradictory samples. Ding *et al.* proposes DISCO [20] that generates the functionally equivalent code by renaming identifiers and permuting independent statements, and involves small security bugs as hard-negative samples. Besides the contrastive learning objective, DISCO also introduces NT-MLM to capture the code syntax. Corder [6] designs semantically preserving AST transformations to produce positive samples. ContraCode [41] uses the compiler to conduct the source-to-source compilation, *a.k.a.,*transpilation, which is originally for code optimization and obfuscation, to generate positive counterparts for JavaScript. We will compare CONCORD with these state-of-the-art contrastive learning code models in § 7.

7 DISCUSSION

In this section, we discuss and compare CONCORD with several most relevant code models using contrastive learning: DISCO [20], Corder [6], and ContraCode [41]. The comparison will be explained with respect to (1) data augmentation and (2) pre-training strategy.

DISCO. For data augmentation, DISCO's oversimplified data augmentation approach hurts its performance. Its positive heuristics for generating functionally equivalent code focus on renaming variables and functions with abstract names like "VAR_0" and "FUNC_0", which are rare in real programs, resulting in unnatural clones. In contrast, CONCORD incorporates real developers' cloning patterns into pre-training through several clone generation rules, inspired by patterns drawn from the existing studies [11, 33–35, 67, 74] and pre-defined clone types (Type- 1,2,3,4). As a result, Table 2 and Table 3 show that, in the same evaluation setup, CONCORD outperforms DISCO with a clear margin.

Beyond downstream tasks, we also compare the impacts of DISCO's and CONCORD's data augmentation on code representations quality, regarding identifying semantic similarity. We augment the original dataset with DISCO data augmentation, and train CONCORD on top of it (called CCD-DISCO). We reuse the zero-shot setup discussed in § 5.5 that first encodes the program with pre-trained models, and then computes the average of Sim[Original, Clone] and retrieves the top-1 similar code. Table 8 shows that, due to the oversimplified and unnatural transformations used to generate clones, CCD-DISCO has significantly lower cosine similarity between the representation of the original code and the clone, and a higher chance of failure in retrieving clones with the zero-shot setting when compared to CONCORD.

Table 8: Comparison of the quality of code representations in identifying code similarity between CCD-DISCO and CONCORD. ↑ indicates that a larger value represents a better representation, while ↓ indicates that a smaller value represents a better representation.

Model	Avg. Similarity with Clone (↑)	Top-1 Similar @ 20K		
		Clone (↑)	Deviant (↓)	Rand. (↓)
CCD-DISCO	71.7	64.4	3.7	31.8
CONCORD	**94.8**	**95.7**	**2.2**	**2.1**

DISCO's pre-training strategy also has limitations compared to CONCORD. First, it couples the multi-task pre-training into one single phase. This makes it less flexible to leverage large pre-trained code models, and also empirically less effective as the randomly initialized model struggles to learn multiple perspectives of source code at once. As a comparison, CONCORD proposes multi-phase

pre-training that first learns the general perspective of code text and then specializes to learn the syntax and semantics during the second phase. Second, DISCO pre-trains a mono-lingual model for each programming language (PL) that fails to unify the common knowledge across distinct PLs and degrades the quality of the learned code representations [24]. Training mono-lingual models separately also wastes training resources. In contrast, CONCORD is a multi-lingual model that learns comprehensive code representations across distinct PLs with cheaper training costs. Third, CONCORD's LTSP is by design more capable of encoding code structures than DISCO's NT-MLM, as the former predicts local ASTs for the whole program while the latter only reconstructs the node type for the masked token.

Corder. For data augmentation, Corder proposes AST transformations to synthesize semantic preserving samples. However, Corder does not consider any types of hard negative samples that are guaranteed to behave differently from the original code. This overlook might weaken the model's capacity in contrasting the semantic similarity of code, especially in differentiating the benign and buggy samples that are textually similar (*e.g.,* Figure 1a *vs.* Figure 1c). In contrast, CONCORD enhances the learning with clone-deviants as hard negative samples, which include bugs that maliciously change the original program behaviors. During pre-training, Corder takes AST-based intermediate representation as input and ignores learning the source code text directly. Such ignorance could make the model less capable of understanding the rich semantics underneath the source code text, such as variable/function names and comments, which are the main resources to expose developers' intentions during coding [8–10], and consequently, degrade the quality of learned code representations.

ContraCode. ContraCode smartly leverages the off-the-shelf compiler to generate optimized or obfuscated programs as semantic preserving samples. While the optimized and obfuscated code provides precise and formal semantics [8], they tend to be unnatural, introducing data structures and variable names that are not commonly used in human-written programs. Existing studies have argued that such formal but unnatural programs are less favorable to human developers [9, 10] and obstruct the code models' learning [11]. Also, ContraCode does not generate semantically contradicting programs as hard negative samples. In contrast, CONCORD imitates the developers' cloning patterns to augment the dataset with clones and clone-deviants, better aligning with human-written programs. For pre-training, ContraCode does not learn code syntax, such as ASTs, which might hurts the model's performance. ContraCode performs poorly in our evaluation datasets (*e.g.,* its best variant only reports 65.6 MAP@R on POJ-104) due to the above limitations as well as other practical restrictions, such as the model size being too small to compete with other baselines we discussed.

8 THREATS TO VALIDITY

We argued that incorporating real developers' coding patterns into pre-training helps to learn better, generic code representations. As a proof of concept, we choose clone-related coding behaviors as our main focus, since code clones are happening all the time in daily development. However, there are still other interesting patterns, such as how developers name variables/functions [2, 13], that will help deep-learning models understand source code and can be integrated into the pre-training. Also, our data augmentation only generate the variants of the same programming language as the original code, and our model might have limited capacity in detecting cross-lingual semantic clones.

Our data augmentation is trying to imitate the developers' cloning behaviors and have designed multiple transformation rules, but they may not cover all the clone patterns, as programming is a very personal activity [50], and different persons can implement the same function with drastically distinct algorithms. It is also very difficult to guarantee that clone-deviants will exhibit malicious behaviors, but we expect the model to be sensitive to unexpected changes as they are highly likely to introduce software flaws. In addition, during contrastive learning, we consider in-batch samples as random negative samples, since we assume that samples inside the same batch do not have similar functionalities. This assumption might not hold for 100% cases. However, because the batch is randomly sampled from millions of programs, it should be very rare that the in-batch programs share the same functionalities.

Another limitation is that CONCORD is pre-trained with multiple objectives, such that weights of different loss functions might have impacts on the quality of pre-training [31]. We design the weights to be $\lambda_1 = 1.0$, $\lambda_2 = 0.1$, $\lambda_3 = 1.0$ based on the intuition that giving high weights to LTSP might force the model to focus too much on code syntax and diminish the impacts of semantic-aware contrastive learning. We also conducted experiments with $\lambda_2 = 0.1$, 0.5, and 1.0, and $\lambda_2 = 0.1$ reports the best performance on downstream tasks but their difference is not significant: *e.g.,* $\lambda_2 = 0.1$ reports 91.5/86.5 MAP@R in the clone detection datasets, while $\lambda_2 = 0.5$ reports 91.5/86.4 MAP@R, and $\lambda_2 = 1.0$ reports 91.3/86.3 MAP@R, respectively. Pre-training code models is expensive, so we did not exhaustively search for the best weights for each objective. Other weights might improve CONCORD's performance.

9 CONCLUSION

In this paper, we incorporate into pre-training a common developer practice, copy/paste, to improve the quality and efficiency of learning code representations. We first introduce an automated tool to augment the code datasets with both semantic clones and buggy clone-deviants. With these augmented datasets, we pre-train CONCORD with MLM, LTSP, and CLR objectives. Our evaluation reveals the effectiveness and efficiency of our approach, showing that even with much cheaper training expenses, CONCORD still outperforms SOTA code models. In addition, CONCORD's approach can easily be applied to existing code models to improve their code representation quality.

ACKNOWLEDGMENTS

We appreciate all the anonymous reviewers for their thoughtful feedback and suggestions to improve this work.

This work was supported in part by an IBM Ph.D. Fellowship, an IBM Faculty Award, NSF grants CCF-1815494, CCF-210740, CCF-1845893, IIS-2221943, and DARPA/NIWC Pacific N66001-21-C-4018. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect those of the US Government, NSF, DARPA, or IBM.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668. https://www.aclweb.org/anthology/2021.naacl-main.211

[2] Toufique Ahmed and Prem Devanbu. 2022. Multilingual training for Software Engineering. abs/2112.02043 (2022).

[3] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-Supervised Bug Detection and Repair. In *NeurIPS*.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[5] B. S. Baker. 1995. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Second Working Conference on Reverse Engineering (WCRE '95)*. IEEE Computer Society, USA, 86.

[6] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations. In *SIGIR '21* (Virtual Event, Canada). 511–521. https://doi.org/10.1145/3404835.3462840

[7] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. 2020. Exploring Software Naturalness through Neural Language Models. arXiv:2006.12641 [cs.CL]

[8] Casey Casalnuovo, Earl T Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. 2020. A theory of dual channel constraints. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 25–28.

[9] Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. 2020. Do programmers prefer predictable expressions in code? *Cognitive science* 44, 12 (2020), e12921.

[10] Casey Casalnuovo, E Morgan, and P Devanbu. 2020. Does surprisal predict code comprehension difficulty. In *Proceedings of the 42nd Annual Meeting of the Cognitive Science Society*. Cognitive Science Society Toronto, Canada.

[11] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar Devanbu, and Baishakhi Ray. 2022. NatGen: Generative pre-training by" Naturalizing" source code. In *2022 The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM.

[12] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep Learning based Vulnerability Detection: Are We There Yet. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3087402

[13] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. 2021. VarCLR: Variable Semantic Representation Pre-training via Contrastive Learning. *CoRR* abs/2112.02650 (2021). arXiv:2112.02650 https://arxiv.org/abs/2112.02650

[14] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A Simple Framework for Contrastive Learning of Visual Representations. In *Proceedings of the 37th International Conference on Machine Learning*. PMLR, 1597–1607. https://proceedings.mlr.press/v119/chen20j.html

[15] S. Chopra, R. Hadsell, and Y. LeCun. 2005. Learning a similarity metric discriminatively, with application to face verification. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, Vol. 1. 539–546 vol. 1. https://doi.org/10.1109/CVPR.2005.202

[16] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. *SIGOPS Oper. Syst. Rev.* 35, 5 (oct 2001), 73–88. https://doi.org/10.1145/502059.502042

[17] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *ICLR*. https://openreview.net/pdf?id=r1xMH1BtvB

[18] CVE-2022-23559. 2022. https://nvd.nist.gov/vuln/detail/CVE-2022-23559.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[20] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray, and Saikat Chakraborty. 2022. Towards Learning (Dis)-Similarity of Source Code from Program Contrasts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 6300–6312. https://doi.org/10.18653/v1/2022.acl-long.436

[21] Yangruibo Ding, Baishakhi Ray, Devanbu Premkumar, and Vincent J. Hellendoorn. 2020. Patching as Translation: the Data and the Metaphor. In *35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. https://doi.org/10.1145/3324884.3416587

[22] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*. IEEE Computer Society, USA, 109.

[23] Adam Paszke *et al.*. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*.

[24] Ben Athiwaratkun *et al.*. 2023. Multi-lingual Evaluation of Code Generation Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Bo7eeXm6An8

[25] Martín Abadi *et al.*. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[26] Mark Chen *et al.*. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[27] Tom B. Brown *et al.*. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[28] Thomas Wolf *et al.*. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[29] Yujia Li *et al.*. 2022. Competition-Level Code Generation with AlphaCode. *ArXiv* abs/2203.07814 (2022).

[30] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[31] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. In *Empirical Methods in Natural Language Processing (EMNLP)*.

[32] GitHub. 2021. GitHub Copilot: Your AI Pair Programmer. https://copilot.github.com/.

[33] Dan Gopstein, Anne-Laure Fayard, Sven Apel, and Justin Cappos. 2020. Thinking Aloud about Confusing Code: A Qualitative Investigation of Program Comprehension and Atoms of Confusion. In *In Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. ACM. https://doi.org/10.1145/3368089.3409714

[34] Dan Gopstein, Jake Iannacone, Yu Yan, Lois Anne Delong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. 2017. Understanding Misunderstandings in Source Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 129–139.

[35] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. 2018. Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild. In *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 281–291. https://doi.org/10.1145/3196398.3196432

[36] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. https://doi.org/10.48550/ARXIV.2203.03850

[37] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCode{BERT}: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

[38] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 837–847.

[39] Harold Hotelling. 1936. Relations Between Two Sets of Variates. *Biometrika* 28, 3/4 (1936), 321–377. http://www.jstor.org/stable/2333955

[40] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). arXiv:1909.09436 http://arxiv.org/abs/1909.09436

[41] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez, and Ion Stoica. 2020. Contrastive Code Representation Learning. *arXiv preprint* (2020).

[42] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. TreeBERT: A Tree-Based Pre-Trained Model for Programming Language. *ArXiv* abs/2105.12485 (2021).

[43] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do Code Clones Matter?. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 485–495. https://doi.org/10.1109/ICSE.2009.5070547

[44] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*

(San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[45] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *ICML 2020*.

[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 *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1073–1085.

[47] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin B. Clement, and Neel Sundaresan. 2022. Learning to Reduce False Positives in Analytic Bug Detectors. (2022).

[48] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[49] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2015).

[50] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (may 1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[51] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Brussels, Belgium, 66–71. https://doi.org/10.18653/v1/D18-2012

[52] Jingyue Li and Michael D. Ernst. 2012. CBCD: Cloned Buggy Code Detector. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 310–320.

[53] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/osdi-04/cp-miner-tool-finding-copy-paste-and-related-bugs-operating-system-code

[54] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. 201–213. https://doi.org/10.1145/2991079.2991102

[55] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[56] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. https://doi.org/10.48550/ARXIV.2203.07722

[57] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[58] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code Recommendation via Structural Code Search. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 152 (oct 2019), 28 pages. https://doi.org/10.1145/3360578

[59] MITRE. 2020. *Common Weakness Enumeration*. https://cwe.mitre.org/data/index.html.

[60] MITRE. 2022. *Common Vulnerabilities and Exposures*. https://cve.mitre.org.

[61] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. 1287–1293.

[62] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. *CoRR* abs/2201.01549 (2022). arXiv:2201.01549 https://arxiv.org/abs/2201.01549

[63] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (oct 2018), 25 pages. https://doi.org/10.1145/3276517

[64] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *CoRR* abs/2105.12655 (2021). arXiv:2105.12655 https://arxiv.org/abs/2105.12655

[65] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 428–439. https://doi.org/10.1145/2884781.2884848

[66] Baishakhi Ray, Miryung Kim, Suzette Person, and Neha Rungta. 2013. Detecting and characterizing semantic inconsistencies in ported code. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 367–377. https://doi.org/10.1109/ASE.2013.6693095

[67] Chanchal K. Roy. 2009. Detection and analysis of near-miss software clones. In *2009 IEEE International Conference on Software Maintenance*. 447–450. https://doi.org/10.1109/ICSM.2009.5306301

[68] Vaibhav Saini, Farima Farmahinifarahani, Hitesh Sajnani, and Cristina Lopes. 2021. *Oreo: Scaling Clone Detection Beyond Near-Miss Clones*. Springer Singapore, Singapore, 63–74. https://doi.org/10.1007/978-981-16-1927-4_5

[69] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 1157–1168. https://doi.org/10.1145/2884781.2884877

[70] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. https://doi.org/10.48550/ARXIV.1910.01108

[71] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 56 (2014), 1929–1958. http://jmlr.org/papers/v15/srivastava14a.html

[72] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 476–480.

[73] Tree-sitter. 2022. *Tree-sitter*. https://github.com/tree-sitter/tree-sitter.

[74] Md. Sharif Uddin, Chanchal K. Roy, Kevin A. Schneider, and Abram Hindle. 2011. On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems. In *2011 18th Working Conference on Reverse Engineering*. 13–22. https://doi.org/10.1109/WCRE.2011.12

[75] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation. https://doi.org/10.48550/ARXIV.2108.04556

[76] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*.

[77] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. *arXiv preprint arXiv:2202.13169* (2022).

[78] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 111–120. https://doi.org/10.1109/ICSE-SEIP52600.2021.00020

[79] Yaqin Zhou, Shangqing Liu, Jingkai 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*. 10197–10207.

Received 2023-02-16; accepted 2023-05-03