# Towards Learning (Dis)-Similarity of Source Code from Program Contrasts

**Yangruibo Ding**[§], **Luca Buratti**[†], **Saurabh Pujar**[†], **Alessandro Morari**[†],
**Baishakhi Ray**[§], **and Saikat Chakraborty**[§]
[§] Columbia University
[†] IBM Research
[§]{yrbding, rayb, saikatc}@cs.columbia.edu
[†]{luca.buratti1, saurabh.pujar}@ibm.com, amorari@us.ibm.com

## Abstract

Understanding the functional (dis)-similarity of source code is significant for code modeling tasks such as software vulnerability and code clone detection. We present DISCO (*DIS-similarity of COde*), a novel self-supervised model focusing on identifying (dis)similar functionalities of source code. Different from existing works, our approach does not require a huge amount of randomly collected datasets. Rather, we design structure-guided code transformation algorithms to generate synthetic code clones and inject real-world security bugs, augmenting the collected datasets in a targeted way. We propose to pre-train the Transformer model with such automatically generated program contrasts to better identify similar code in the wild and differentiate vulnerable programs from benign ones. To better capture the structural features of source code, we propose a new cloze objective to encode the local tree-based context (*e.g.,* parents or sibling nodes). We pre-train our model with a much smaller dataset, the size of which is only 5% of the state-of-the-art models' training datasets, to illustrate the effectiveness of our data augmentation and the pre-training approach. The evaluation shows that, even with much less data, DISCO can still outperform the state-of-the-art models in vulnerability and code clone detection tasks.

## 1 Introduction

Understanding the functional similarity/dissimilarity of source code is at the core of several code modeling tasks such as software vulnerability and code clone detection, which are important for software maintenance (Kim et al., 2017; Li et al., 2016). Existing pre-trained Transformer models (Guo et al., 2021; Feng et al., 2020; Ahmad et al., 2021) show promises for understanding code syntax (*i.e.,* tokens and structures). However, they still get confused when trying to identify functional (dis)-similarities. For instance, syntax-based models can embed two code fragments with identical functionality but very different tokens and structures as distinct vectors and fail to identify them as semantically similar. Likewise, these models cannot distinguish between two code fragments that differ in functionalities but share a close syntactic resemblance. For example, consider an if statement `if(len(buf) < N)` checking buffer length before accessing the buffer. Keeping the rest of the program the same, if we simply replace the token '$<$' with '$\leq$,' the modification can potentially trigger security vulnerability, *e.g.,* buffer overflow bug[1]. It is challenging for existing pre-training techniques to tell apart such subtle differences in the functionalities.

In addition, existing pre-training techniques rely on a huge volume of training corpus that is randomly selected. For fine-tuning tasks like code clone detection or vulnerability detection, such random selection of training data is never tailored to teach the model about code functionalities.

To address these limitations, we present DISCO, a self-supervised pre-trained model that jointly learns the general representations of source code and specific functional features for identifying source code similarity/dis-similarity. Similar to state-of-the-art pre-trained Transformer models (Devlin et al., 2019; Liu et al., 2019), we apply the standard masked language model (MLM) to capture the token features of source code. To learn about the structural code properties, we propose a new auxiliary pre-training task that consumes additional inputs of local tree-based contexts (*e.g.,* parent or sibling nodes in abstract syntax trees) and embeds such structural context, together with the token-based contexts, into each token representation. On top of such well-learned general code representations, we further incorporate prior knowledge of code clones and vulnerable programs into the pre-training to help the model learn the functional (dis)-similarity. We design structure-guided

[1]https://en.wikipedia.org/wiki/Buffer_overflow

code transformation heuristics to automatically augment each training sample with one synthetic code clone (*i.e.,* positive samples) that is structurally different yet functionally identical and one vulnerable contrast (*i.e.,* hard negative samples) that is syntactically similar but injected with security bugs. During the pre-training, DISCO learns to bring similar programs closer in the vector space and differentiate the benign code from its vulnerable contrast, using a contrastive learning objective. Since we augment the dataset in a more targeted way than existing works and the model *explicitly* learns to reason about a code *w.r.t.* its functional equivalent and different counterparts during pre-training, DISCO can learn sufficient knowledge for downstream applications from a limited amount of data, consequently saving computing resources. In particular, we evaluate DISCO for clone detection and vulnerability detection, as the knowledge of similar/dissimilar code fragments is at the core of these tasks.

To this end, we pre-train DISCO on a *small* dataset, with only 865 MB of C code and 992 MB Java code from 100 most popular GitHub repositories, and evaluate the model on four different datasets for vulnerability and code clone detection. Experiments show that our small models outperform baselines that are pre-trained on 20× larger datasets. The ablation study (§5.4) also reveals that pre-training our model with 10× larger datasets further improves the performance up to 8.2%, outperforming state-of-the-art models by 1% for identifying code clones and up to 9.6% for vulnerability detection, even if our dataset is still smaller.

In summary, our contributions are: 1) We design structure-guided code transformation heuristics to automatically augment training data to integrate prior knowledge of vulnerability and clone detection without human labels. 2) We propose a new pre-training task to embed structural context to each token embedding. 3) We develop DISCO, a self-supervised pre-training technique that jointly and efficiently learns the textual, structural, and functional properties of code. Even though pre-trained with significantly less data, DISCO matches or outperforms the state-of-the-art models on code clone and vulnerability detection.

## 2 Related Works

**Pre-training for Source Code.** Researchers have been passionate about pre-training Transformer models (Vaswani et al., 2017) for source code with two categories: encoder-only and encoder-decoder (Ahmad et al., 2021; Wang et al., 2021; Rozière et al., 2021; Phan et al., 2021). Our work focuses on pre-training encoder-only Transformer models to understand code. Existing models are pre-trained with different token level objectives, such as masked language model (MLM) (Kanade et al., 2020; Buratti et al., 2020), next sentence prediction (NSP) (Kanade et al., 2020), replaced token detection, and bi-modal learning between source code and natural languages (Feng et al., 2020). However, these approaches ignore the underlying structural information to fully understand the syntax and semantics of programming languages. Recently, more works aimed to understand the strict-defined structure of source code leveraging abstract syntax tree (AST) (Zügner et al., 2021; Jiang et al., 2021), control/data flow graphs (CFG/DFG) (Guo et al., 2021). DISCO leverages code structures differently from existing works in two ways: (a.) with AST/CFG/DFG, we automatically generate program contrasts to augment the datasets targeting specific downstream tasks. (b.) DISCO takes an additional input of local AST context, and we propose a new cloze task to embed local structural information into each token representation.

**Self-supervised Contrastive Learning.** Self-supervised contrastive learning, originally proposed for computer vision (Chen et al., 2020), has gained much interest in language processing (Giorgi et al., 2021; Wu et al., 2020; Gao et al., 2021). The common practice of self-supervised contrastive learning is building similar counterparts, without human interference, for the original samples and forcing the model to recognize such similarity from a batch of randomly selected samples. Corder (Bui et al., 2021) leverages contrastive learning to understand the similarity between a program and its functionally equivalent code. While Corder approach will help code similarity detection type of applications, their pre-training does not learn to differentiate syntactically very close, but functionally different programs. Such differentiation is crucial for models to work well for bug detection (Ding et al., 2020). ContraCode (Jain et al., 2020) also leverages contrastive learning. However, they generate negative contrast for a program from unrelated code examples, not from variants of the same code. They also do not encode the structural information into the code as we do. Inspired by the empirical findings that hard negative image and text samples are beneficial for contrastive learning (Gao et al., 2021; Robinson et al., 2021), DISCO learns both from equivalent code as the positive contrast, and functionally different yet syntactically close code as the *hard-negative* contrast. We generate hard-negative samples by injecting small but crucial bugs in the original code (§3.1).

## 3 Data Augmentation Without Human Labels

Our pre-training aims to identify similar programs that can be structurally different (positive sample) and differentiate the buggy programs (negative sample) that share structural resemblances with the benign ones. Thus, we need a labeled positive and a negative example for each original sample. Manually collecting them is expensive, especially at the scale of pre-training. To this end, we design code transformation heuristics to automatically generate such positive and negative samples so that the transformation can be applied to any amount of programs without human efforts.

We first represent a code sample as Abstract Syntax Tree (AST), and build a control/data flow graph from the AST. The code transformation heuristics are then applied to this graph. For every original code sample ($x$), we apply semantic preserving transformation heuristics (§3.2) to generate a positive sample ($x^+$) and a bug injection heuristics (§3.1) to generate a hard-negative code example ($x^-$). We design the heuristics in a way that makes $x^+$ be the functional equivalent or semantic clone of $x$ and $x^-$ be the buggy/noisy version of $x$. Noted that not all heuristics are applicable to all code samples; we decide on applicable heuristics based on the flow graph of the original code. Figure 1 shows an example of the code transformation.

### 3.1 Bug Injection

To generate a hard negative sample ($x^-$) from a given code ($x$), we define six categories of bug injection heuristics. Here our goal is to maintain maximum token-level similarity to the original code, so that the model can learn to analyze source code beyond token-level similarity. These heuristics are inspired by the buggy code patterns from a wide range of Common Weakness Enumeration (CWE) types (Appendix A.1). While it is challenging to guarantee that $x^-$ will exhibit vulnerability or security bug, our heuristics will force $x^-$ to exhibit different functionality than $x$. Compared with a concurrent work from Allamanis et al. (2021), our methods are significantly different. First, we focus on concrete types of security bugs that have been identified by the security experts, while they mainly target regular bugs. Second, our scope is not only bug detection but clone detection as well, and we apply contrastive learning to differentiate the code functionalities of code clones and vulnerabilities.

**Misuse of Data Type.** Usage of the wrong data type can trigger several security flaws. For instance, using a smaller data type (*e.g.,* `short`) to replace a larger one (*e.g.,* `long`) may result in an overflow bug (*e.g.,* CVE-2021-38094 (2021)). Such errors are complicated to track since they are usually exhibited in input extremities (*i.e.,* very large or very small values). For languages allowing implicit typecasting, such an incorrect type may even cause imprecision, resulting in the unpredictable behavior of the code. We intentionally change the data types in $x$ to inject potential bugs, while ensuring the code can still be compiled (*e.g.,* we will not replace `int` with `char`).

**Misuse of Pointer.** Incorrect pointer usage is a major security concern. Accessing uninitialized pointers may lead to unpredictable behavior. A `NULL` pointer or freed pointer could lead to Null Pointer Dereferencing vulnerability (*e.g.,* CVE-2021-3449 (2021)). To inject such bugs, we randomly remove the initialization expression during pointer declaration, or set some pointers to `NULL`.

**Change of Conditional Statements.** Programmers usually check necessary preconditions using `if-statement` before doing any safety-critical operation. For instance, before accessing an array with an index, a programmer may add a condition checking the validity of the index. Lack of such checks can lead to buffer-overflow bugs in code (*e.g.,* CVE-2020-24020 (2020)). We introduce bugs in the code by removing such small `if-statements`. In addition, we also inject bugs by modifying randomly selected arithmetic conditions— replace the comparison operator ($<$, $>$, $\leq$, $\geq$, $==$, $!=$) with another operator, to inject potential out-of-bound access, forcing the program to deviate from its original behavior.

**Misuse of Variables.** When there are multiple variables present in a code scope, incorrect use of variables may lead to erroneous behavior of the program. Such errors are known as VARMISUSE bug (Allamanis et al., 2018). We induce code with such bugs by replacing one variable with another.

```
static void filter16_roberts(
    uint8_t *dstp, int width, float scale,
    float delta, int peak, ...){
    uint16_t *dst = (uint16_t *)dstp;
    int x;
    for (x = 0; x < width; x++) {
        float suma = AV_RN16A ( ... );
        float sumb = AV_RN16A ( ... );
        dst[x] = av_clip(
            sqrtf(suma*suma + sumb*sumb) *
            scale + delta, 0, peak);}}
```

(a) Original Code

```
static void filter16_roberts(
    uint8_t *v1, int v2, float v3,
    float v4, int v5, ...){
    int v7;
    uint16_t *v8 = (uint16_t *)v1;
    for (v7 = 0; v7 < v2; v7++) {
        float v9 = Func1 ( ... );
        float v10 = Func1 ( ... );
        v8[v7] = Func2(
            sqrtf(v10*v10 + v9*v9) *
            v3 + v4, 0, v5);}}
```

(b) Functionally Equivalent Code

```
static void filter16_roberts(
    uint8_t *dstp, int width, float scale,
    float delta, int peak, ...){
    uint16_t *dst = (uint16_t *)dstp;
    int x;
    for (x = 0; x < width; x++) {
        int suma = AV_RN16A ( ... );
        int sumb = AV_RN16A ( ... );
        dst[x] = av_clip(
            sqrtf(suma*suma + sumb*sumb) *
            scale + delta, 0, peak);}}
```

(c) Bug Injected Code

Figure 1: An example illustrating data augmentation. 1a shows the original code that is adapted from the CVE-2021-38094 patch. 1b shows functionality equivalent code of 1a where the original code is transformed by renaming and statements permutation. 1c shows a variation from 1a where a potential integer overflow bug is injected.

To keep the resultant code compilable, we perform scope analysis on the AST and replace a variable with another variable reachable in the same scope.

**Misuse of Values.** Uninitialized variables or variables with wrong values may alter the program behaviors and consequently cause security flaws (*e.g.,* CVE-2019-12730 (2019)). We modify the original code by removing the initializer expression of some variables. In addition, to induce the code with `divide-by-zero` vulnerability, we identify the potential divisor variables from the flow graph and forcefully assign zero values to them immediately before the division.

**Change of Function Calls.** We induce bugs in the code by randomly changing arguments of function calls. For a randomly selected function call, we add, remove, swap, or assign `NULL` value to arguments, forcing the code to behave unexpectedly.

### 3.2 Similar Code Generation

To generate positive samples ($x^+$) from a given code, we use three different heuristics. In this case, our goal is to generate functionally equivalent code while inducing maximum textual difference. These heuristics are inspired by code clone literature (Funaro et al., 2010; Sheneamer et al., 2018).

**Variable Renaming.** Variable renaming is a typical code cloning strategy and frequently happens during software development (Ain et al., 2019). To generate such a variant of the original code, we either (a.) rename a variable in the code with a random identifier name or (b.) with an abstract name such as `VAR_i` (Rozière et al., 2021). While choosing random identifier names, we only select available identifiers in the dataset. We ensure that both the definition of the variable and subsequent usage(s) are renamed for any variable renaming. We also ensure that a name is not used to rename more than one variable.

**Function Renaming.** We rename function calls with abstract names like `FUNC_i`. By doing this, we make more tokens different compared with the original code but keep the same syntax and semantics. We do not rename library calls for the code (*e.g.,* `memcpy()` in C).

Noted that even if tokens like `VAR_i` and `FUNC_i` are rare in normal code, the model will not bias towards identifying samples with these tokens as positive samples. The reason is that, as shown in Figure 2, $x^+, y^+$ and $z^+$ all potentially have these abstract tokens, but the model learns to move $EMB_x$ closer to $EMB_{x^+}$ and further from $EMB_{y^+}$ and $EMB_{z^+}$, regardless of the existence of abstract tokens.

**Statement Permutation.** The relative order among the program statements that are independent of each other can be changed without altering the code functionality. More specifically, we focus on the variable declaration or initialization statements. We first conduct the 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.

## 4 DISCO

This section presents the model architecture, input representation, and pre-training tasks. DISCO uses a 12-layered Transformer encoder model similar to BERT. We feed the model with both source code text and structure (AST) information (§4.1). We pre-train DISCO using three different pre-training tasks (§4.2). Figure 2 depicts an example workflow of DISCO. We randomly select tokens in the original sample, mask them and their node types, and then use the embedding of these masks to predict them back. We further extract the sequence embeddings within a minibatch and contrast them based on the code functionality.

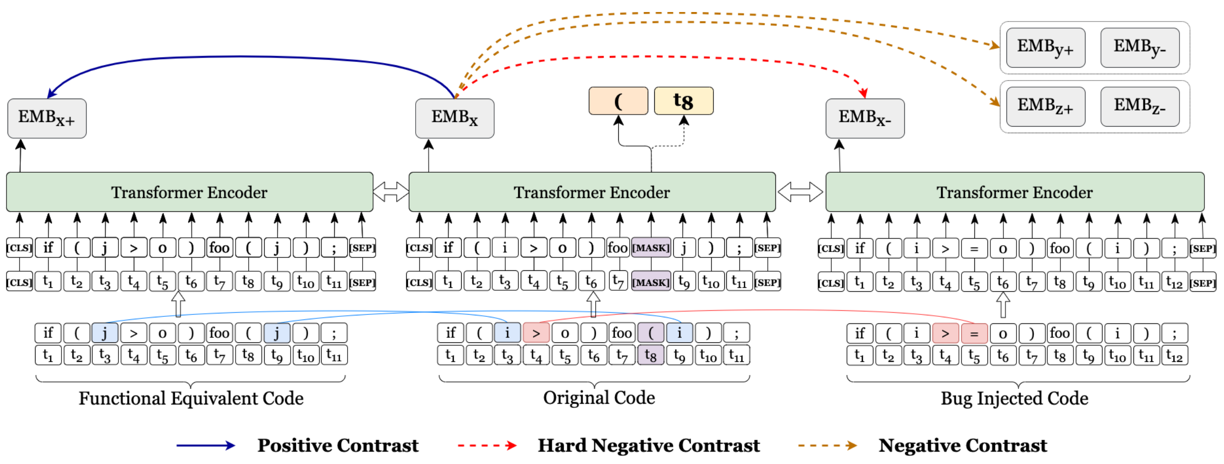


Figure 2: An illustration of DISCO pre-training with a minibatch of three. The original code and its node types will be randomly masked with $[MASK]$, and the final representation of masked tokens will be used to recover their source tokens and node types. The original code, say $x$, will also be transformed to build $(x, x^+, x^-)$. Then the triplet will be fed into the **same** Transformer encoder and get the embedding of each sequence with $[CLS]$ tokens for contrastive learning.

### 4.1 Input Representation

**Source Code.** Given a program ($x$), we apply a lexical analyzer to tokenize it based on the language grammar and flatten the program as a token sequence ($x_1 x_2 ... x_m$, where $x_i$ is i$^{th}$ token in the code). We further train a sentencepiece (Kudo and Richardson, 2018) tokenizer based on such flattened code token sequences with vocabulary size 20,000. We use this tokenizer to divide the source code tokens into subtokens. We prepend the subtoken sequence with a special token `[CLS]` and append with a special token `[SEP]`. Finally, DISCO converts the pre-processed code sequence $C = \{[CLS], c_1, c_2, ..., c_k, [SEP]\}$ to vectors $V^{src} = \{v^{src}_{[CLS]}, v^{src}_1, v^{src}_2, ..., v^{src}_k, v^{src}_{[SEP]}\}$ with a token embedding layer.

**Local AST Types.** For every token in the input code, we extract the node type ($tt$) from the syntax tree. Since such types are all terminal node types (*e.g.,* keyword, identifier, punctuation), we do not get enough information about the structure only with these types. In order to add more information about the tree, we also extract its parent type ($pt$) for each token. Such parent type provides us with information about the structural context of a token. For instance, when the parent type of an `identifier` is `Function-Declarator`, we know that the identifier is a function name. In contrast, when the `identifier`'s parent is a `Binary Expression`, it should be a variable. Consequently, we annotate each code subtoken $c_i$ with a local AST-type token $t = tt\#pt$. It is worth noting that sub-tokens coming from the same code token will all have the same type. Therefore, we have the AST-type sequence for the code $T = \{[CLS], t_1, t_2, ..., t_k, [SEP]\}$, and DISCO converts it as vectors $V^{type} = \{v^{type}_{[CLS]}, v^{type}_1, v^{type}_2, ..., v^{type}_k, v^{type}_{[SEP]}\}$ with a type embedding layer. Appendix Table 7 shows an example of code tokens and their AST types. DISCO generates token representation $v_i$ of subtoken $c_i$ as a sum of token embedding $v^{src}_i$ and type embedding $v^{type}_i$. Thus, $V = V^{src} + V^{type}$.

### 4.2 Pre-training

We aim to pre-train the DISCO to learn the representation of source code based on (a.) token-based context, (b.) AST-based context, and (c.) code functionality. In that spirit, we pre-train DISCO to optimize on three different objectives, *i.e.,* masked language model (MLM), local AST node type-MLM (NT-MLM), and Contrastive Learning (CLR).

For a given program $x$, we first embed the tokens and node-types to vectors $V = \{v_{[CLS]}, v_1, ..., v_{[SEP]}\}$. We optimize MLM loss ($\mathcal{L}_{MLM}$) (§4.2.1) and NT-MLM loss ($\mathcal{L}_{NT-MLM}$) (§4.2.2) based on $x$. These two loss functions learn about the textual and syntactic context of source code. For every code sample $x$ in a minibatch of input, we generate a positive example $x^+$ and a hard-negative example $x^-$ using the heuristics described in Section 3. We optimize CLR loss ( $\mathcal{L}_{CLR}$) (§4.2.3) considering the original code and its positive and hard-negative counterparts. The final loss function to optimize for pre-training DISCO is

$$\mathcal{L}(\theta) = \mathcal{L}_{MLM}(\theta) + \mathcal{L}_{NT-MLM}(\theta) + \mathcal{L}_{CLR}(\theta)$$

#### 4.2.1 Encoding Token-based Context

We apply the standard masked language model to the original code ($x$). Given a source code sequence $C$, we randomly choose 15% of tokens and replace them with a special token $[MASK]$ for 80% of the time and a random token for 10% of the time and leave the rest 10% unchanged. We record the indices of masked token as $loc_m$, replaced token as $loc_r$ and unchanged tokens as $loc_u$ for node-type MLM. We define the union of these indices as $M = loc_m \cup loc_r \cup loc_u$. MLM will learn to recover the masked source code $\{c_i | i \in M\}$ given the Transformer encoder's output $h_i$. We present the loss for MLM as $\mathcal{L}_{MLM} = \sum_{i \in M} -logP(c_i|h_i)$

#### 4.2.2 Encoding AST-based Context

Token-based MLM re-builds the token using its surrounding tokens and successfully encodes the contextual information into each token representation. Motivated by MLM, we propose the tree-based context-aware pre-training task, to encode the structural context, such as parent, sibling, and children nodes. As we have shown in Figure 2, we flatten the ASTs as sequences and we expect the flattened trees can preserve the local structure information (i.e., sub-trees containing terminal nodes), and existing work (Chakraborty et al., 2020; Hellendoorn et al., 2020) has empirically shown such potentials. To this end, we introduce AST node-type masked language model (NT-MLM). Given the corresponding AST-type sequence $T$ of source code $C$, we mask the AST types $\{t_p | p \in loc_m\}$ with the special token $[MASK]$, and replace the AST types $\{t_q | q \in loc_r\}$ with random tokens. Specifically, by doing this, we make sure that if a source code token is chosen to be masked or replaced, its corresponding AST type will perform the same operation. NT-MLM will learn to recover the masked AST type $\{t_i | i \in M\}$ given the Transformer encoder's output $h_i$. We present the loss for NT-MLM as $\mathcal{L}_{NT-MLM} = \sum_{i \in M} -logP(t_i|h_i)$

A recent work, CodeT5 (Wang et al., 2021), proposes to predict token type as well. However, our new objective is different from them in both high-level designs and the detailed implementation. First, their objective only predicts one single token type: identifiers, while our approach predicts all possible AST types. Also, we do not only consider the AST node type of tokens, but also include their AST parents to embed the local sub-tree context (§4.1). Second, CodeT5 implements the identifier tagging task as a binary classification (0/1) for each token, while our NT-MLM reconstructs the local ASTs out of hundreds of distinct types.

#### 4.2.3 Contrastive Learning

We adopt contrastive learning to focus on the functional characteristics of code. With the structure-guided code transformation algorithms in Section 3, we are able to generate a positive sample ($x^+$ in Figure 2) and a hard negative sample ($x^-$ in Figure 2) for each program in the dataset. More specifically, we have a minibatch of $N$ programs, and for each program, we extract the sequence representation from the Transformer outputs $\mathbf{h} = h_{[CLS]}$. We will augment every sequence in the minibatch with positive and negative samples, and then the minibatch is extended to N triplets of $(\mathbf{h}, \mathbf{h}^+, \mathbf{h}^-)$. We refer to the contrastive loss with hard negative samples from Gao et al. (2021) and we adapt it to our scope as follows. We use cosine similarity as the $sim()$ function and $\tau$ is the temperature parameter to scale the loss, and we use $\tau = 0.05$.

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

We also consider to pre-train the model with only positive counterparts as a variation. In such a case, the minibatch will contain N pairs of $(\mathbf{h}, \mathbf{h}^+)$ and the loss is computed as

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

## 5 Experiments

In this section, we will explain our experimental settings and report the results. We evaluate our model on vulnerability and code clone detection.

### 5.1 Experimental Settings

**Data.** We collect our pre-training corpus from open-source C and Java 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 ".java" and ".c" files for Java and C repositories respectively, and we further remove comments and empty lines from these files. The corresponding datasets for Java and C are of size of 992MB and 865MB, respectively. Our datasets are significantly *smaller* than existing pre-training models (Feng et al., 2020; Ahmad et al., 2021; Guo et al., 2021). For example, while CodeBERT and GraphCodeBERT are trained on 20GB data, we used an order of magnitude less data. Details of our datasets and the comparison can be found in Appendix Table 5.

**Models.** To study the different design choices, we train four variations of DISCO. (i) **MLM+CLR$^{\pm}$+NT-MLM** is trained by all three tasks with hard negative samples. (ii) **MLM+CLR$^{\pm}$**. The input of this model only considers the source code sequence and ignores the AST-type sequence. This model helps us understand the impact of NT-MLM. (iii) **MLM+CLR$^{+}$**. This variant evaluates the effectiveness of hard negative code samples, by contrasting its performance with MLM+CLR$^{\pm}$. (iv) **MLM**. This is the baseline trained with only MLM objective. We provide detailed model configuration in Appendix A.4 to ensure the reproducibility.

**Baselines.** We consider two types of baselines: encoder-only pre-trained Transformers and existing deep-learning tools designed for code clone and vulnerability detection. We do not consider encoder-decoder pre-trained Transformers as baselines, since such generative models always need much more pre-training data and training steps to converge, so it is unfair to compare our model with them. For example, PLBART uses 576G source code for pre-training, while we only use less than 1G. Based on the data size. As future work, we plan to pre-train the model on much larger datasets.

## 5.2 Vulnerability Detection (VD)

VD is the task to identify security bugs: given a source code function, the model predicts 0 (benign) or 1 (vulnerable) as binary classification.

**Dataset and Metrics.** We consider two datasets for VD task: REVEAL (Chakraborty et al., 2021) and CodeXGLUE (Lu et al., 2021; Zhou et al., 2019). In the real-world scenario, vulnerable programs are always rare compared to the normal ones, and Chakraborty et al. (2021) have shown such imbalanced ratio brings challenges for deep-learning models to pinpoint the bugs. To imitate the real-world scenario, they collect REVEAL dataset from Chromium (open-source project of Chrome) and Linux Debian Kernel, which keeps the ratio of vulnerable to benign programs to be roughly 1:10. Following Chakraborty et al. (2021), we consider precision, recall and F1 as the metrics.

CodeXGLUE presents another dataset of security vulnerabilities. It is less real-world than REVEAL, since it a balanced dataset, but it has been frequently used by existing Transformer-based models to evaluate their tools for VD task. To compare with these baselines, we use CodeXGLUE train/valid/test splits for training and testing. We use accuracy as the metric, following the design of the benchmark.

**REVEAL.** Table 1 shows the results. We compare with four deep-learning-based VD tools. VulDeePecker (Li et al., 2018b) and SySeVR (Li et al., 2018a) apply program slices and sequence-based RNN/CNN to learn the vulnerable patterns. Devign (Zhou et al., 2019) uses graph-based neural networks (GNN) to learn the data dependencies of program. REVEAL (Chakraborty et al., 2021) applies GNN + SMOTE (Chawla et al., 2002) + triplet loss during training to handle the imbalanced distribution. We also consider pre-trained RoBERTa, CodeBERT and GraphCodeBERT, and a 12-Layer Transformer model trained from scratch.

Table 1: Vulnerability Detection Results on REVEAL.

| Model | Prec. (%) | Rec. (%) | F1 (%) |
|---|---|---|---|
| VulDeePecker | 17.7 | 13.9 | 15.7 |
| SySeVR | 24.5 | 40.1 | 30.3 |
| Devign | 34.6 | 26.7 | 29.9 |
| REVEAL | 30.8 | **60.9** | 41.3 |
| Transformer | 41.6 | 45.3 | 43.4 |
| RoBERTa | 44.5 | 39.0 | 41.6 |
| CodeBERT | 44.6 | 45.8 | 45.2 |
| GraphCodeBERT | 47.9 | 43.9 | 45.8 |
| DISCO | | | |
| MLM | 45.5 | 31.0 | 36.9 |
| MLM+CLR$^{+}$ | 38.6 | 47.7 | 42.6 |
| MLM+CLR$^{\pm}$ | 39.4 | 50.5 | 44.2 |
| MLM+CLR$^{\pm}$+NT-MLM | **48.3** | 44.6 | **46.4** |

Table 2: Results on CodeXGLUE for vulnerability detection

| Model | Acc (%) |
|---|---|
| Transformer | 62.0 |
| RoBERTa | 61.0 |
| CodeBERT | 62.1 |
| GraphCodeBERT | 63.2 |
| C-BERT | 63.6* |
| DISCO | |
| MLM | 61.8 |
| MLM+CLR$^{+}$ | **64.4** |
| MLM+CLR$^{\pm}$ | 63.6 |
| MLM+CLR$^{\pm}$+NT-MLM | 63.8 |

*We take this result from Buratti et al. (2020). They did not use CodeXGLUE splits, so the test data can be different with other baselines.

In our case, the best DISCO variation with contrastive learning and NT-MLM objective outperforms all the baselines, including the graph-based approaches and models pre-trained with larger datasets. This empirically proves that DISCO can efficiently understand the code semantics and data dependencies from limited amount of data, helping the identification of the vulnerable patterns. We notice that hard negative samples (*i.e.,* buggy code contrasts) helps DISCO improve the performance. The reason is that REVEAL contains thousands of (buggy version, fixed version) pairs for the same function. Two functions in such a pair are different by only one or a few tokens. Such real-world challenges align well with our automatically generated buggy code, and pre-training with these examples teaches the model better distinguish the buggy code from the benign ones. We provide an example in Appendix Figure 3 to illustrate this.

**CodeXGLUE.** We consider four pre-trained models: RoBERTa, CodeBERT, GraphCodeBERT and C-BERT. The first three are pre-trained on much larger datasets than ours. However, even trained with small dataset, three variations of DISCO outperforms the baselines. Unlike REVEAL, CodeXGLUE does not have those challenging pairs of functions' buggy and patched version; thus the hard negative contrast in DISCO does not help the model much.

Table 3: Clone detection on POJ104 and BigCloneBench

| Model | POJ104 | BigCloneBench | | |
|---|---|---|---|---|
| | MAP@R | Prec.(%) | Rec.(%) | F1(%) |
| Transformer | 62.11 | - | - | - |
| MISIM-GNN | 82.45 | - | - | - |
| RoBERTa | 76.67 | - | - | - |
| CodeBERT* | 82.67 | 94.7 | 93.4 | 94.1 |
| GraphCodeBERT* | - | 94.8 | **95.2** | **95.0** |
| DISCO | | | | |
| MLM | **83.32** | 93.4 | 93.8 | 93.6 |
| MLM+CLR$^+$ | 82.44 | 93.9 | 93.7 | 93.8 |
| MLM+CLR$^\pm$ | 82.73 | **95.1** | 93.3 | 94.2 |
| MLM+CLR$^\pm$+NT-MLM | 82.77 | 94.2 | 94.6 | 94.4 |

*The authors of both works fixed bugs in their evaluation tool and updated the results in their Github repositories. We directly take their latest results and use their latest evaluation tool for fair comparisons.

## 5.3 Clone Detection

Clone detection aims to identify the programs with similar functionality. It also can help detecting security vulnerabilities—given a known vulnerability, we can scan the code base with clone detector and check for similar code snippets.

**Dataset and Metrics.** We consider POJ-104 (Mou et al., 2016) and BigCloneBench (Svajlenko et al., 2014) as the evaluation datasets. We again strictly follow the CodeXGLUE train/dev/test splits for experiments. Following CodeXGLUE's design, we use MAP@R as the metric for POJ-104 and precision/recall/F1 as the metric for BigCloneBench.

**POJ-104.** We consider three pre-trained models, one graph-based model (Ye et al., 2020) and one 12-layer Transformer model trained from scratch as baselines. Table 3 shows that, with hard negative contrast and NT-MLM, DISCO outperforms all baselines including CodeBERT, which is pre-trained on much larger datasets. This highlights the significance of learning the code contrasts together with syntactical information to better capture the functional similarities. Interestingly, we notice that DISCO-MLM performs the best among all variations. This indicates that our current positive heuristics might not align with all the clone patterns in this benchmark. As future work, we will propose more code transformation rules to imitate more real-world clone patterns.

**BigCloneBench.** Our best model achieves slightly better precision than the baselines indicating that our designs with contrastive learning and structure information can compensate the loss brought by less data. However, our recall is slightly worse than GraphCodeBERT, since they are pre-trained on large datasets with code graph. We conclude that enlarging our Java pre-training dataset is necessary for code clone detection and we regard this as future work.

## 5.4 Medium Pre-trained Model

As shown in Section 5, DISCO trained on a small dataset achieves comparable or even better performance than models pre-trained on large datasets in vulnerability and clone detection (Let's call this version DISCO$_{small}$). We further explore the benefits of pre-training using larger data. We pre-train a MEDIUM model, DISCO$_{medium}$, on our extended datasets with more C-language Github repositories (13G). Note that our medium dataset is still smaller than the large dataset of the baseline models (13G vs. 20G). We evaluate DISCO$_{medium}$ on C-language tasks. The results are shown in Table 4. Increasing the pre-training dataset improves the performance of downstream tasks, outperforming the best baselines' results.

Table 4: Results for the best baselines, small- and medium-DISCO. POJ-104 is for code clone task; VD-CXG and VD-RV are VD tasks for CodeXGLUE and REVEAL datasets.

| Model | POJ-104 (MAP@R) | VD-CXG (Acc) | VD-RV (F1) |
|---|---|---|---|
| DISCO$_{small}$ | 82.8 | 63.8 | 46.4 |
| DISCO$_{medium}$ | **83.8** | **64.6** | **50.2** |
| Baseline$_{large}$ | 82.7 | 63.6 | 45.8 |

## 6 Conclusion

In this work, we present DISCO, a self-supervised contrastive learning framework to both learn the general representations of source code and specific characteristics of vulnerability and code clone detections. Our evaluation reveals that DISCO pre-trained with smaller dataset can still outperform the large models' performance and thus prove the effectiveness of our design.

## Acknowledgements

We would appreciate the insightful feedback and comments from the anonymous reviewers. This work was partially done when Yangruibo Ding was an intern at IBM Research. This work is also supported in part by NSF grants CCF-2107405, CCF-1845893, IIS-2040961, and IBM.

## Ethical Considerations

The main goal of DISCO is to generate functionality-aware code embeddings, producing similar representations for code clones and differentiating security bugs from the benign programs. Our data is collected from either the open-source projects, respecting corresponding licences' restrictions, or publicly available benchmarks. Meanwhile, throughout the paper we make sure to summarize the paper's main claims. We also discussed DISCO's limitation and potential future work for clone detection in Section 5.3. We report our model configurations and experiment details in Appendix A.4.

## References

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*, pages 2655–2668, Online. Association for Computational Linguistics.

Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A systematic review on code clone detection. *IEEE Access*, 7:86121–86144.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *International Conference on Learning Representations*.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. In *NeurIPS*.

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*, page 511–521.

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.

S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, pages 1–1.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*, pages 1–1.

Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357.

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*, pages 1597–1607. PMLR.

CVE-2019-12730. 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-12730.

CVE-2020-24020. 2020. https://nvd.nist.gov/vuln/detail/CVE-2020-24020.

CVE-2021-3449. 2021. https://nvd.nist.gov/vuln/detail/CVE-2021-3449.

CVE-2021-38094. 2021. https://nvd.nist.gov/vuln/detail/CVE-2021-38094.

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*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

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*, ASE '20.

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*, pages 1536–1547, Online. Association for Computational Linguistics.

Marco Funaro, Daniele Braga, Alessandro Campi, and Carlo Ghezzi. 2010. A hybrid approach (syntactic and textual) to clone detection. In *Proceedings of the 4th International Workshop on Software Clones*, pages 79–80.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple contrastive learning of sentence embeddings. In *Empirical Methods in Natural Language Processing (EMNLP)*.

John Giorgi, Osvald Nitski, Bo Wang, and Gary Bader. 2021. DeCLUTR: Deep contrastive learning for unsupervised textual representations. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 879–895, Online. Association for Computational Linguistics.

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*.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code. In *International Conference on Learning Representations*.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *arXiv preprint*.

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.

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

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)*, pages 595–614.

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*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

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*, ACSAC '16, page 201–213.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2018a. Sysevr: A framework for using deep learning to detect software vulnerabilities.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018b. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'2018)*.

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.

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.

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*, pages 1287–1293.

Long N. Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James T. Anibal, Alec Peltekian, and Yanfang Ye. 2021. Cotext: Multi-task learning with code-text transformer. *CoRR*, abs/2105.08645.

Joshua David Robinson, Ching-Yao Chuang, Suvrit Sra, and Stefanie Jegelka. 2021. Contrastive learning with hard negative samples. In *International Conference on Learning Representations*.

Baptiste Rozière, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A deobfuscation pre-training objective for programming languages. *CoRR*, abs/2102.07492.

Abdullah Sheneamer, Swarup Roy, and Jugal Kalita. 2018. A detection framework for semantic code clones and obfuscated code. *Expert Systems with Applications*, 97:405–420.

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*, pages 476–480. IEEE.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010.

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*.

Zhuofeng Wu, Sinong Wang, Jiatao Gu, Madian Khabsa, Fei Sun, and Hao Ma. 2020. CLEAR: contrastive learning for sentence representation. *CoRR*, abs/2012.15466.

Fangke Ye, Shengtian Zhou, Anand Venkat, Ryan Marcus, Nesime Tatbul, Jesmin Jahan Tithi, Paul Petersen, Timothy G. Mattson, Tim Kraska, Pradeep Dubey, Vivek Sarkar, and Justin Gottschlich. 2020. MISIM: an end-to-end neural code similarity system. *CoRR*, abs/2006.05265.

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*, pages 10197–10207.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations (ICLR)*.

## A Appendix

### A.1 Bug Injection Heuristics and Common Weakness Enumeration Types

Our automated bug injection heuristics are motivated by the real-world security bugs that are always small but hazardous. We empirically conclude the frequently happened vulnerable patterns based on the concrete CWE types. Table 6 shows that each of our operation is relating to several CWE types. We inject all these security issues automatically and ask model to distinguish them with the benign samples.

### A.2 Node Type Details

We parse the source code into ASTs and extract the node type and parent node type for each token. Table 7 shows an example after parsing. We can see that, with the parent node type, each token can be well embedded with its local structure contexts. Considering two tokens that are distant from each other: `if` and `else`. With only node types, we just know these two tokens are keywords, but with parent node type, we can easily know that they are from the same `if-statement` and they are siblings in the AST.

Table 5: Comparison of pre-training dataset size between ours and other related work

| Dataset | Instance Count | Total Size |
|---|---|---|
| DISCO | | |
| Java SMALL | 187 K | 992 MB |
| C SMALL | 64 K | 865 MB |
| C MEDIUM | 860 K | 12 GB |
| CodeBERT | 8.5 M | 20 GB |
| GraphCodeBERT | 2.3 M | 20 GB |
| CuBERT | 7.4 M | - |
| DOBF | - | 45 GB |
| PLBART | - | 576 GB |

### A.3 Dataset

**Pre-training** We collect our dataset from C and Java Github repositories. Our main dataset is the combination of Java SMALL and C SMALL. From Table 5, we can see that our dataset is significantly smaller than the existing pre-trained models. For an ablation study (§ 5.4) with enlarged dataset, we collect a MEDIUM dataset of C language. We have seen the improvement using such larger dataset, but even MEDIUM dataset is still much smaller than other datasets.

**Datasets for downstream tasks** We provide dataset details of our downstream tasks in Table 8. Noted that for POJ-104 (Mou et al., 2016), Table 8 only shows the number of code samples, and we follow the design of CodeXGLUE that build positive and negative pairs during the minibatch generation. The amount of pairs for training is much larger than the number of samples.

### A.4 Configuration

DISCO is built based on a stack of 12 layers transformer encoder with 12 attention heads and 768 hidden sizes. The model is implemented with PyTorch-1.9.0 and Huggingface-transformer-4.12.3 [2]. Longer sequences are disproportionately expensive so we follow the original BERT design by pre-training the model with short sequence length for first 70% steps and long sequence length for the rest 30% steps to learn the positional embeddings. DISCO is trained with Java SMALL and C SMALL for 24 hours in total with two 32GB NVIDIA Tesla V100 GPUs, using batch size of 128 with max sequence length of 256 tokens and batch

[2] `https://github.com/huggingface/transformers/tree/c439752482759c94784e11a87dcbf08ce69dccf3`

Table 6: Common Weakness Enumeration (CWE) types covered by our bug injection heuristic

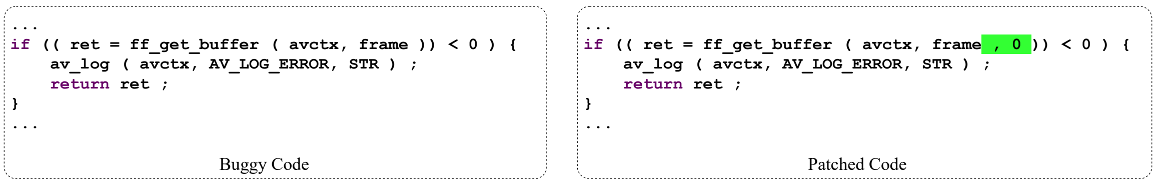
| Operation | Potential CWE types |
|---|---|
| Misuse of Data Type | CWE-190: Integer overflow<br>CWE-191: Integer Underflow<br>CWE-680: Integer Overflow to Buffer Overflow<br>CWE-686: Function Call With Incorrect Argument Type<br>CWE-704: Incorrect Type Conversion or Cast<br>CWE-843: Access of Resource Using Incompatible Type |
| Misuse of Pointer | CWE-476: NULL Pointer Dereference<br>CWE-824: Access of Uninitialized Pointer<br>CWE-825: Expired Pointer Dereference |
| Change of Conditional Statements | CWE-120: Buffer Overflow<br>CWE-121: Stack-based Buffer overflow<br>CWE-122: Heap-based Buffer overflow<br>CWE-124: Buffer Underflow<br>CWE-125: Out-of-bounds Read<br>CWE-126: Buffer Over-read<br>CWE-129: Improper Validation of Array Index<br>CWE-787: Out-of-bounds Write<br>CWE-788: Access of Memory Location After End of Buffer<br>CWE-823: Use of Out-of-range Pointer Offset |
| Misuse of Values | CWE-369: Divide By Zero<br>CWE-456: Missing Initialization of a Variable<br>CWE-457: Use of Uninitialized Variable<br>CWE-908: Use of Uninitialized Resource |
| Change of Function Calls | CWE-683: Function Call With Incorrect Order of Arguments<br>CWE-685: Function Call With Incorrect Number of Arguments<br>CWE-686: Function Call With Incorrect Argument Type<br>CWE-687: Function Call With Incorrectly Specified Argument Value<br>CWE-688: Function Call With Incorrect Variable or Reference |

Table 7: Examples for tokens and their AST node types

| **token** | **node type** | **parent node type** | **token** | **node type** | **parent node type** |
|---|---:|---:|---|---:|---:|
| int | type | func_definition | ) | punctuation | parenthesized_expr |
| foo | identifier | func_declarator | return | keyword | return_stmt |
| ( | punctuation | param_list | ( | punctuation | parenthesized_expr |
| int | type | parameter_declaration | 1 | number_literal | parenthesized_expr |
| bar | identifier | parameter_declaration | ) | punctuation | parenthesized_expr |
| ) | punctuation | parameter_list | ; | punctuation | return_stmt |
| { | punctuation | compount_stmt | else | keyword | if_stmt |
| if | keyword | if_stmt | return | keyword | return_stmt |
| ( | punctuation | parenthesized_expr | ( | punctuation | parenthesized_expr |
| bar | identifier | binary_expr | 0 | number_literal | parenthesized_expr |
| < | operator | binary_expr | ) | number_literal | parenthesized_expr |
| 5 | number_literal | binary_expr | ; | punctuation | return_stmt |
| | | | } | punctuation | compount_stmt |

Table 8: Details of downstream tasks datasets.

| Task | Dataset | Language | Train | Valid | Test |
|---|---|---|---|---|---|
| Vulnerability Detection | Chakraborty et al. (2021) | C/C++ | 15,867 | 2,268 | 4,535 |
| | Zhou et al. (2019) | C/C++ | 21,854 | 2,732 | 2,732 |
| Clone Detection | Mou et al. (2016) | C/C++ | 32,000 | 8,000 | 12,000 |
| | Svajlenko et al. (2014) | Java | 901,028 | 415,416 | 415,416 |


```
...
if (( ret = ff_get_buffer ( avctx, frame )) < 0 ) {
    av_log ( avctx, AV_LOG_ERROR, STR ) ;
    return ret ;
}
...
```
Buggy Code

```
...
if (( ret = ff_get_buffer ( avctx, frame , 0 )) < 0 ) {
    av_log ( avctx, AV_LOG_ERROR, STR ) ;
    return ret ;
}
...
```
Patched Code

Figure 3: An example in REVEAL dataset. The patched code happens to be in the train split and the buggy code is in the test split. During inference, DISCO MLM+CLR$^{\pm}$ model can correctly predict the buggy code as vulnerable, while MLM+CLR$^{+}$predicts it as benign.

size of 64 sequences with max sequence length 512 tokens. DISCO is also trained with C MEDIUM for 3 days, using batch size of 1024 sequences with max sequence length of 256 tokens and batch size of 512 sequences and max sequence length of 512 tokens. We use the Adam optimizer and 1e-4 as the pre-training learning rate. For fine-tuning tasks, we use batch size of 8 and the learning of 8e-6. We train the model with train split and evaluate the model during the training using validation split. We pick the model with best validation performance for testing.

## A.5 Case Study

We studied the model performance on REVEAL dataset for vulnerability detection. Figure 3 shows two samples inside REVEAL. We can recognize that they are from the same program. We further checked the details of these two example and we found the code on the left is a buggy version, and it is fixed by adding an argument of value 0 to the function call. This real-world situation actually matches our "Change of Function Calls" (§ 3.1) bug injection operation. In the REVEAL dataset, the patched code is in the training corpus while the buggy one is in the test split. Interestingly, during the inference, DISCO MLM+CLR$^{\pm}$ can correctly predict the buggyiess while MLM+CLR$^{+}$fails. This empirically prove our bug injected samples can help the model identify small but siginicant real-world vulnerabilities.