# CONTRASTIVE LEARNING FOR SOURCE CODE WITH STRUCTURAL AND FUNCTIONAL PROPERTIES

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Pre-trained transformer models have recently shown promises for understanding the source code. Most existing works expect to understand code from the textual features and limited structural knowledge of code. However, the program functionalities sometimes cannot be fully revealed by the code sequence, even with structure information. Programs can contain very different tokens and structures while sharing the same functionality, but changing only one or a few code tokens can introduce unexpected or malicious program behaviors while preserving the syntax and most tokens. In this work, we present BOOST, a novel *self-supervised* model to focus pre-training based on the characteristics of source code. We first employ automated, structure-guided code transformation algorithms that generate (i.) functionally equivalent code that looks drastically different from the original one, and (ii.) textually and syntactically very similar code that is functionally distinct from the original. We train our model in a way that brings the functionally equivalent code closer and distinct code further through a *contrastive learning* objective. To encode the structure information, we introduce a new *node-type masked language model* objective that helps the model learn about structural context. We pre-train BOOST with a much smaller dataset than the state-of-the-art models, but our small models can still match or outperform these large models in code understanding and generation tasks.

## 1 INTRODUCTION

Large pre-trained models have been applied for source code and reported promising performance for code understanding and generation tasks. These models have successfully captured the code features by treating code as a sequence of text (Feng et al., 2020; Buratti et al., 2020; Kanade et al., 2020). Some works further explored the potentials of leveraging structural characteristics of programming languages (*e.g.,* abstract syntax tree and code graph) to understand code (Guo et al., 2021; Jiang et al., 2021).

However, even with a good understanding of code syntax (*i.e.,* tokens and structures), such pre-trained models can get confused when understanding code functionalities. For instance, two code fragments having identical functionality (a.k.a. semantic clones) but different syntax may not be recognized as similar by the existing models. Likewise, these models cannot distinguish between two code fragments that differ in functionalities but share 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 capture such subtle differences in the functionalities. In addition, existing pretraining techniques rely on huge volume of training corpus that is randomly selected. Given the main motivation of pretraining is to understand program's functionality (Ahmad et al., 2021), such a random selection of training data is not tailored for explicitly learning the functionalities.

To address these limitations, we present BOOST, a self-supervised pre-trained model that jointly learns code tokens, structures and functionalities. For understanding code tokens, we use a standard masked language model (MLM). To learn about the structural code properties, we propose a

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

new pre-training task, *node-type masked language model (NT-MLM)*, which embeds the local tree-based contexts together with the token-based contexts into each token. Finally, to capture functional properties, we carefully design the pre-training to recognize the structurally different yet functionally identical programs (*i.e.,* positive samples) as similar and to differentiate between structurally close but functionally different programs (*i.e.,* negative samples). We apply contrastive learning to bring the functionally similar code embeddings closer in the vector space and vice versa. We design structure-guided code transformation heuristics to automatically augment each training sample with one positive and one hard negative contrasts. Since the model *explicitly* learns to reason about a code *w.r.t.* its functional equivalent and different counterparts during pretraining, we expect BOOST to learn sufficient knowledge for downstream applications from limited amount of data, consequently saving computing resources.

We pre-train BOOST 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 code understanding and generation tasks: vulnerability detection, code clone detection and code summarization. Experiments show that our small models outperform baselines that are pre-trained on 20$\times$ larger datasets. Our analysis further illustrates that by adding carefully crafted positive and negative code samples and the new NT-MLM objective, the model better understands code during pre-training. The ablation study also reveals that pre-training our model with 10$\times$ larger datasets further improves the performance up to 8.2%.

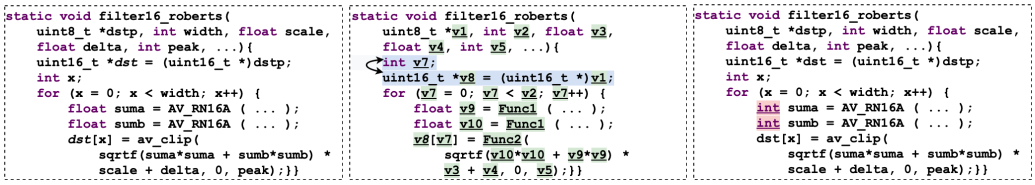
In summary, our major contributions are: 1) We design structure-guided code transformation heuristics to automatically augment training data without human labels. We introduce negative code samples (carefully crafted to inject real-world bugs) for the first time, to better learn code contrasts. 2) We propose a new pre-training task, NT-MLM, to embed structural context to each token embedding. 3) We develop BOOST, 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, BOOST matches or outperforms the state-of-the-art models on code understanding and generation tasks.

## 2 RELATED WORKS

**Token-based Pre-training for Code** Researchers have been passionate about pre-training transformer models (Vaswani et al., 2017; Devlin et al., 2019; Liu et al., 2019) for source code and reported promising results in code understanding and generation tasks (Feng et al., 2020; Pei et al., 2020; Kanade et al., 2020; Ahmad et al., 2021; Buratti et al., 2020). Such models are pre-trained with different token level objectives *e.g.,* 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 natual languages (Feng et al., 2020). While such pre-training objectives show promise in several source code understanding downstream tasks, they all ignore the syntactic and functional features of programming languages, which are crucial for understanding and reasoning about source code (Zhou et al., 2019; Chakraborty et al., 2021). In this work, we design BOOST to comprehensively understand code with three pre-training objectives, in a self-supervised style, to capture the textual, syntactic and functional aspects respectively.

**Modeling Code as Structured Data** Prior works aimed to understand the strict-defined structure of source code leveraging AST (Bui et al., 2021b; Alon et al., 2019; Chakraborty et al., 2020), control/data flow graphs (Hellendoorn et al., 2020; Allamanis et al., 2018; Yin et al., 2019; Dinella et al., 2020; Zhou et al., 2019; Chakraborty et al., 2021). While these techniques explicitly take the structure of the source code as input, other approaches aim at learning about source code through objective functions (Jiang et al., 2021; Guo et al., 2021). BOOST takes the advantage of both techniques. BOOST takes an additional input of AST node types into the model to capture code structure. BOOST also implicitly learns to reason about the structure by (a.) contrasting between syntactically and functionally equivalent and contradictory code examples, and (b.) learning to predict the AST node type of a token with node type masked language model (NT-MLM).

**Self-supervised Contrastive Learning** Self-supervised contrastive learning, originally proposed for computer vision (Chen et al., 2020), has gained a lot of 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 for the original samples and force the model to recognize




(a) Original Code (b) Functionally Equivalent Code (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 small variation from 1a where a potential integer overflow bug is injected.

such similarity from a batch of randomly selected samples. Therefore, how to build similar counterparts without human interference will significantly affect the model's overall quality. Corder (Bui et al., 2021a) leverages contrastive learning to understand similarity between a code and its functionally equivalent code. While Corder approach will help code similarity detection type of applications, their pretraining 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 code from unrelated code examples, not from variant of 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), BOOST learns both from equivalent code as positive contrast, and functionally different yet syntactically close code as *hard-negative* contrast. BOOST's heuristics for generating equivalent code (see Section 3.2) are inspired by source code obfucation Rozière et al. (2021). We also design heuristics (see section 3.1) to generate hard-negative samples by injecting small but crucial bugs in the original code.

## 3 DATA AUGMENTATION WITHOUT HUMAN LABELS

Goal of our pre-training is to identify the 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, for each original sample, we need a labeled positive and a negative example. Manually collecting them is expensive, especially at the scale of pre-training. Thus, 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 on 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 $x$. Note that, not all heuristics are applicable to all code samples; we decide on applicable heuristics based on the flow graph of original code. Figure 1 shows an example of the code transformation.

### 3.1 BUG INJECTION

For generating 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 number of Common Weakness Enumeration (CWE) types (Appendix A.1). While it is very difficult to guarantee that $x^-$ will exhibit vulnerability or security bug, our heuristics will force $x^-$ to exhibit different functionality than $x$.

**Misuse of Data Type.** Usage of a wrong data type can trigger several security flaws. For instance, using a smaller data type (*e.g.,* `short`) in place of a larger one (*e.g.,* `long`) may result in overflow bug (*e.g.,* CVE-2021-38094 (2021)). Such errors are very difficult to track since they are usually

exhibited in input extremities (*i.e.,* very large or very small values). For languages allowing implicit type casting, such incorrect type may even cause imprecision, resulting in the unpredictable behavior of the code. We intentionally change the data types in $x$ with wrong ones to inject potential bugs.

**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 bug in code (*e.g.,* CVE-2020-24020 (2020)). We introduce bug in the code by removing such small `if-statement` statement. In addition, we also inject bug 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 (*e.g.,* a `if` block surrounding number of statement), 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 a variable with another. 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 behavior, may even cause security flaw (*e.g.,* CVE-2019-12730 (2019)). We modify the original code by removing the initializer expression of variable. 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 bug in the code by randomly changing arguments of function call. 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 identifiers that are available in the dataset. For any variable renaming, we ensure that both the definition of the variable and subsequent usage(s) are renamed. 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).

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

## 4 BOOST

This section presents the model architecture, input representation and pre-training tasks. BOOST uses a 12-layered transformer encoder (Vaswani et al., 2017) model similar to BERT (Devlin et al.,

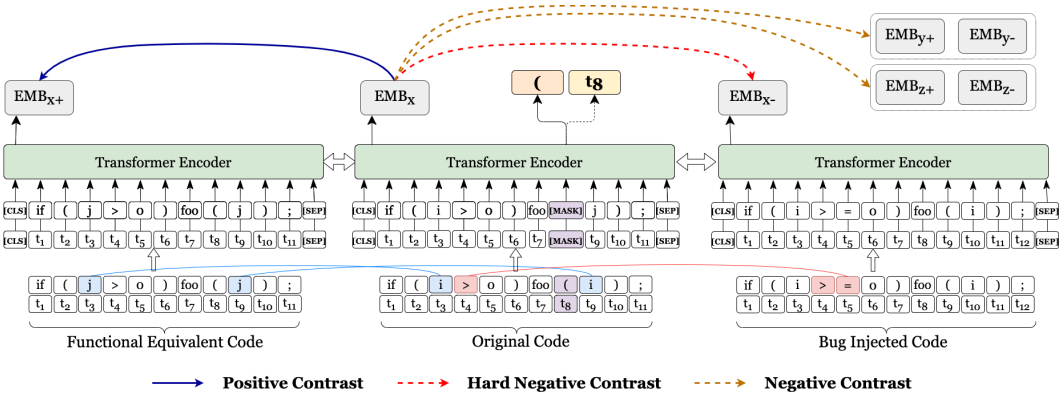



Figure 2: An illustration of BOOST pre-training with a minibatch of three. The original code and its node types will be randomly masked with $[MASK]$, and the final representation of masked tokens will be used to recover their source tokens and node types. The original code, say $x$, will also be transformed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

2019). We feed the model with both source code text and structure (AST) information (§4.1). We pretrain BOOST using three different pretraining tasks (§4.2). Figure 2 depicts an example workflow of BOOST. We randomly select tokens in the original sample and 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 learn to contrast them based on the code functionality.

### 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 & 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, BOOST 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.

**AST Node Types.** For every token in 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, for each token, we also extract its parent type ($pt$). Such parent type provides us with information about the context of a token. For instance, when parent type of an `idenfier` is `Function-Declarator`, we know that that identifier is a function name. In contrast, when the `identifier`'s parent is a `Binary Expression`, it should a variable. Consequently, we annotate each code sub-token $c_i$ with a type token $t = tt\#pt$. It is worth noting that, subtokens coming from the same original code token will all have the same node type. Therefore, we have the node type sequence for the code $T = \{[CLS], t_1, t_2, ..., t_k, [SEP]\}$, and BOOST 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 node types. BOOST 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

Our aim for pretraining BOOST is to train the model to learn robust representation of the source code. We aim training the BOOST to learn representation of source code based on (a.) textual context, (b.) syntax tree, and (c.) code functionality. In that spirit, we pretrain BOOST to optimize on three different objectives, *i.e.,* Masked Language Model (MLM), Node Type - Masked Language Model (NT-MLM), and Contrastive Learning (CLR).

For a given code $x$, 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 $x$ in a minibatch of input, we generate positive example $x^+$ and hard-negative example $x^-$ using the heuristics described in Section 3. We optimize CLR loss ( $\mathcal{L}_{CLR}$) (§4.2.3) on original code and its positive and hard-negative counterparts. The final loss function to optimize for pretraining BOOST is

$$\mathcal{L}(\theta) = \mathcal{L}_{MLM}(\theta) + \mathcal{L}_{NT-MLM}(\theta) + \mathcal{L}_{CLR}(\theta) \tag{1}$$

#### 4.2.1 MASKED LANGUAGE MODEL

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 NODE-TYPE MASKED LANGUAGE MODEL

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 shown in Figure 2, we flatten the ASTs as sequences and we expect the flattened trees can well preserve the local structure information (i.e., sub-trees containing terminal nodes), and existing work (Chakraborty et al., 2020) has empirically shown such potentials. To this end, we introduce node-type masked language model (NT-MLM). Given the corresponding node type sequence $T$ of source code $C$, we mask the node types $\{t_p | p \in loc_m\}$ with the special token $[MASK]$, and replace the node 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 node type will perform the same operation. NT-MLM will learn to recover the masked source code $\{t_i | i \in M\}$ given the transformer encoder's output $h_i$. We present the loss for NT-MLM as $\mathcal{L}_{NT-MLM} = \sum_{i \in M} -logP(t_i|h_i)$

#### 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 pairs 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 $\tau = 0.05$.

$$\mathcal{L}_{CLR} = -\log \frac{\mathbf{e}^{\text{sim}\left(\mathbf{h},\mathbf{h}^+\right)/\tau}}{\sum_{n=1}^{N}\left(\mathbf{e}^{\text{sim}\left(\mathbf{h},\mathbf{h}_n^+\right)/\tau} + \mathbf{e}^{\text{sim}\left(\mathbf{h},\mathbf{h}_n^-\right)/\tau}\right)} \tag{2}$$

## 5 EXPERIMENTS

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

### 5.1 PRE-TRAINING

**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 8.

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

### 5.2 VULNERABILITY DETECTION

Vulnerability detection is the task to the identify programs with security bugs: given a source code function, the model will predict 0 (benign) or 1 (vulnerable) as binary classification.

**Dataset and Metrics.** We consider two datasets for this 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 will bring 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 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.

Table 1: Vulnerability Detection Results on REVEAL Dataset.

| 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 (code) | 44.5 | 39.0 | 41.6 |
| CodeBERT | 44.6 | 45.8 | 45.2 |
| GraphCodeBERT | 47.9 | 43.9 | 45.8 |
| BOOST | | | |
| 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** |

CodeXGLUE presents another dataset of security vulnerabilities. It is a balanced dataset and has been frequently used by existing transformer-based models to evaluate their tools for the vulnerability detection task. To compare with these baselines, we use CodeXGLUE train/valid/test splits for training and testing. We use accuracy as the metrics.

**REVEAL.** Table 1 shows the results. We compare with four deep-learning-based vulnerability detection 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.

In our case, the best BOOST 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

Table 2: Results on CodeXGLUE dataset for vulnerability detection

| Model | Acc (%) |
|---|---|
| Transformer | 62.0 |
| RoBERTa (code) | 61.0 |
| CodeBERT | 62.1 |
| PLBART | 63.2 |
| CBERT | 63.6* |
| BOOST | |
| 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.

that BOOST can efficiently understand the code semantics and data dependencies from limited amount of data. Such understanding help identification of the vulnerable patterns. We also notice that hard negative samples (*i.e.,* buggy code contrasts) helps BOOST 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 4 to illustrate this situation.

**CodeXGLUE.** We consider four pre-trained models: RoBERTa (code), CodeBERT, PLBART and CBERT. The first three are pre-trained on much larger (up to 100 times) datasets than ours. However, even trained with small dataset, all three variations of BOOST again outperforms the larger models. Unlike REVEAL, CodeXGLUE does not have those challenging pairs of functions' buggy and patched version; thus the hard negative contrast in BOOST does not help the model much.

### 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. We use MAP@R as the metric for POJ-104 and precision/recall/f1 as the metric for BigCloneBench.

Table 3: Clone detection results on POJ104 and BigCloneBench

| Model | POJ104 | BigCloneBench | |
|---|---|---|---|
| | MAP@R | Prec.(%) | Rec.(%) |
| Transformer | 62.11 | - | - |
| MISIM-GNN | 82.45 | - | - |
| RoBERTa (code) | 76.67 | - | - |
| CodeBERT | 82.67 | 94.7 | 93.4* |
| GraphCodeBERT | - | 94.8 | **95.2*** |
| BOOST | | | |
| MLM+CLR$^{+}$ | 82.44 | 93.9 | 93.7 |
| MLM+CLR$^{\pm}$ | 82.73 | **95.1** | 93.3 |
| MLM+CLR$^{\pm}$+NT-MLM | **82.77** | 94.2 | 94.6 |

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

**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. From Table 3, all three variations of BOOST can outperform RoBERTa that only considers MLM for pre-traing. This well explains the effectiveness of our multi-task pre-training. Further, with hard negative contrast (HN) and NT-MLM, BOOST 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 of programs.

**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 CODE SUMMARIZATION

Table 4: Result of code summarization

| Model | BLEU-4 |
|---|---|
| Seq2Seq | 15.09 |
| Transformer | 16.26 |
| Roberta (code) | 16.47 |
| CodeBERT | 17.65 |
| BOOST | |
| MLM+CLR$^{+}$ | **17.89** |
| MLM+CLR$^{\pm}$ | 17.76 |
| MLM+CLR$^{\pm}$+NT-MLM | 17.78 |

Besides the sequence understanding tasks, we also hope to see the potentials of our model in sequence generation task, so we choose code summarization. For this task, the model will take the source code as input and generate the natural language description in a auto-regressive style with a decoder. Note that our pre-training do not involve any decoder and we initialize the decoder from scratch for this fine-tuning task.

**Dataset and Metric.** We consider the CodeXGLUE splits of CodeSearchNet (Husain et al., 2019) for this task. Since our model is trained with Java and C, we will only conduct experiments on the Java code summarization data. We use smoothed-BLEU-4 as the metric.

**Results.** We consider the baselines reported by CodeXGLUE: Seq2seq, Transformer, RoBERTa and CodeBERT. Table 4 shows that BOOST is outperforming all the baselines. Contrastive learning objective contributes the most to such an improvement since it helps the model learn the code functionality and consequently generate better summarization. From the result of BOOST variations, we conclude that the buggy contrast and tree structure have limited advantages for code summarization (Ahmad et al., 2020).

## 6 ANALYSIS

**Impacts of Augmented Samples and NT-MLM.** Language model perplexity is an important metric to evaluate the quality of pre-trained embeddings. To better understand how data augmentation and NT-MLM objective affect the pre-training quality, we keep evaluating the perplexity of BOOST's three variations (Section 5.1) on the held-out data during pre-training.

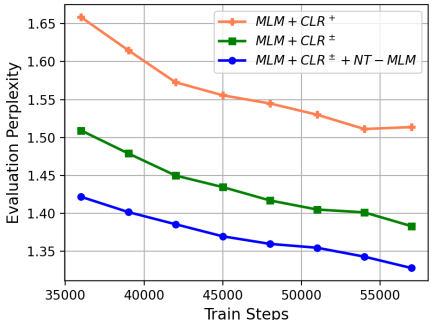



Figure 3: The evaluation perplexity of last five epochs for different BOOST variations.

We plot the last five epochs in Figure 3. As we explained in Figure 2, we only apply MLM to the original sample $x$ regardless of the existence of $(x^+, x^-)$, so it is fair to compare among three models. We can see that the model with hard negative samples keeps having lower perplexity than MLM+CLR$^+$model, and the model with node type information has even lower perplexity than both models that only consider source code tokens. This indicates that even if the models always come across the same sequences for MLM, learning the contrast of the hard negative pairs with tree-based context can further help the model understand the sequence.

Table 5: Results for the best baseline, small BOOST and medium BOOST for each downstream task. POJ-104 is for code clone task; VD-CXG is for CodeXGLUE vulnerability detection; VD-RV is for REVEAL vulnearbility detection

| Model | POJ-104 (MAP@R) | VD-CXG (Acc) | VD-RV (F1) |
|---|---|---|---|
| BOOST$_{small}$ | 82.8 | 63.8 | 46.4 |
| BOOST$_{medium}$ | **83.5** | **64.6** | **50.2** |
| Baseline$_{large}$ | 82.7 | 63.6 | 45.8 |

**Medium Pre-trained Model** As shown in Section 5, BOOST trained on a small dataset achieves comparable or even better performance than models pre-trained on large datasets in code understanding and generation tasks (Let's call this version BOOST$_{small}$). We further explore the benefits of pre-training using larger data. We pre-train a MEDIUM model, BOOST$_{medium}$, on our extended datasets with top-10,000 C-language Github repositories (13G). We evaluate BOOST$_{medium}$ on C-language tasks. The results are shown in Table 5. Increasing the pre-training dataset improves the performance of downstream tasks. Note that our medium dataset is still smaller than the large dataset of the baseline models (13G vs. 20G).

## 7 CONCLUSION

To sufficiently learn the code functionalities with unlabeled data, we first propose structure-guided code transformation algorithms to generate semantically equivalent code and inject bugs to programs without human efforts. Then we present BOOST, a pre-trained model that learns to identify code similarity and separate buggy programs from benign ones with the automatically generated code contrasts. We also introduce a new pre-training objective, NT-MLM, to encode the structural context to each token embedding. Our evaluation on code understanding and generation tasks reveals that BOOST pretrained with smaller dataset can still match the large models' performance and thus prove the effectiveness of our design.

## REFERENCES

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 4998–5007, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.449. URL https://aclanthology.org/2020.acl-main.449.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 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*, pp. 2655–2668, Online, June 2021. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2021.naacl-main.211.

Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. A systematic review on code clone detection. *IEEE Access*, 7:86121–86144, 2019. doi: 10.1109/ACCESS.2019.2918202.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=BJOFETxR-.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019. ISSN 2475-1421. doi: 10.1145/3290353. URL http://doi.acm.org/10.1145/3290353.

Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. SIGIR '21, pp. 511–521, New York, NY, USA, 2021a. Association for Computing Machinery. ISBN 9781450380379. doi: 10.1145/3404835.3462840. URL https://doi.org/10.1145/3404835.3462840.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Infercode: Self-supervised learning of code representations by predicting subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1186–1197. IEEE, 2021b.

Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. Exploring software naturalness through neural language models, 2020.

S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, pp. 1–1, 2020. doi: 10.1109/TSE.2020.3020502.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*, pp. 1–1, 2021. doi: 10.1109/TSE.2021.3087402.

Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Int. Res.*, 16(1):321–357, June 2002. ISSN 1076-9757.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 1597–1607. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/chen20j.html.

CVE-2019-12730. https://github.com/FFmpeg/FFmpeg/commit/9b4004c054964a49c7ba44583f4cee22486dd8f2, 2019.

CVE-2020-24020. https://github.com/FFmpeg/FFmpeg/commit/584f396132aa19d21bb1e38ad9a5d428869290cb, 2020.

CVE-2021-3449. https://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=fb9fa6b51defd48157eeb207f52181f735d96148, 2021.

CVE-2021-38094. https://github.com/FFmpeg/FFmpeg/commit/99f8d32129dd233d4eb2efa44678a0bc44869f23, 2021.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 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 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SJeqs6EFvB.

Yangruibo Ding, Baishakhi Ray, Devanbu Premkumar, and Vincent J. Hellendoorn. Patching as translation: the data and the metaphor. In *35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, 2020. doi: 10.1145/3324884.3416587. URL https://doi.org/10.1145/3324884.3416587.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://www.aclweb.org/anthology/2020.findings-emnlp.139.

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

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

John Giorgi, Osvald Nitski, Bo Wang, and Gary Bader. 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)*, pp. 879–895, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.72. URL https://aclanthology.org/2021.acl-long.72.

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. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=jLoC4ez43PZ.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=B1lnbRNtwr.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019. URL http://arxiv.org/abs/1909.09436.

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

Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. Treebert: A tree-based pre-trained model for programming language. *ArXiv*, abs/2105.12485, 2021.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 12-18 July 2020*, Proceedings of Machine Learning Research. PMLR, 2020.

Taku Kudo and John Richardson. 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*, pp. 66–71, Brussels, Belgium, November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-2012. URL https://aclanthology.org/D18-2012.

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

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

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019. URL http://arxiv.org/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. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

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

Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *CoRR*, abs/2012.08680, 2020. URL https://arxiv.org/abs/2012.08680.

Joshua David Robinson, Ching-Yao Chuang, Suvrit Sra, and Stefanie Jegelka. Contrastive learning with hard negative samples. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=CR1XOQ0UTh-.

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

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

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

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pp. 6000–6010, 2017. ISBN 9781510860964.

Zhuofeng Wu, Sinong Wang, Jiatao Gu, Madian Khabsa, Fei Sun, and Hao Ma. CLEAR: contrastive learning for sentence representation. *CoRR*, abs/2012.15466, 2020. URL https://arxiv.org/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. MISIM: an end-to-end neural code similarity system. *CoRR*, abs/2006.05265, 2020. URL https://arxiv.org/abs/2006.05265.

Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. Learning to represent edits. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=BJl6AjC5F7.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, pp. 10197–10207, 2019.

# 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.

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 |

## 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 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 |

### 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 8, we can see that our dataset is significantly smaller than the existing pre-trained models. For an ablation study (§ 6) 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.

Table 8: Comparison of pre-training dataset size between ours and other related work

| Dataset | Instance Count | Total Size |
|---|---|---|
| BOOST | | |
| 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 |

**Datasets for downstream tasks** We provide dataset details of our downstream tasks in Table 9. Noted that for POJ-104 (Mou et al., 2016), Table 9 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.

Table 9: 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 |
| Code Summarization | Husain et al. (2019) | Java | 164,923 | 5,183 | 10,955 |

### A.4 CONFIGURATION

BOOST is built based on a stack of 12 layers transformer encoder with 12 attention heads and 768 hidden sizes. 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. BOOST is trained with Java SMALL and C SMALL for 24 hours in total with two 32GB NVIDIA Tesla V100 GPUs, using 128 sequences

$\times$ 256 tokens and 64 sequences $\times$ 512 tokens. BOOST is also trained with C MEDIUM for 3 days, using 1024 sequences $\times$ 256 tokens and 512 sequences $\times$ 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 4 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, BOOST MLM+CLR_HN 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.

```
...
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 4: 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, BOOST MLM+CLR_HN model can correctly predict the buggy code as vulnerable, while MLM_CLR predicts it as benign.