Applying Contrastive Learning to Code Vulnerability Type Classification

Chen Ji[1], Su Yang[2], Hongyu Sun[3], Yuqing Zhang[1,2,3]*,
[1]Hangzhou Institute of Technology, Xidian University
[2]National Computer Network Intrusion Protection Center, University of Chinese Academy of Science
[3]School of Cyberspace Security, Hainan University
{jic, yangs, sunhy, zhangyq}@nipc.org.cn

Abstract

Vulnerability classification is a crucial task in software security analysis, essential for identifying and mitigating potential security risks. Learning-based methods often perform poorly due to the long-tail distribution of vulnerability classification datasets. Recent approaches try to address the problem but treat each CWE class in isolation, ignoring their relationships. This results in non-scalable code vector representations, causing significant performance drops when handling complex real-world vulnerabilities. We propose a hierarchical contrastive learning framework for multi-class code vulnerability type classification to bring vector representations of related CWEs closer together. To address the issue of class collapse and enhance model robustness, we mix self-supervised contrastive learning loss into our loss function. Additionally, we employ max-pooling to enable the model to handle longer vulnerability code inputs. Extensive experiments demonstrate that our proposed framework outperforms state-of-the-art methods by 2.97%−17.90% on accuracy and 0.98%−22.27% on weighted-F1, with even better performance on higher-quality datasets. We also utilize an ablation study to prove each component's contribution. These findings underscore the potential and advantages of our approach in the multi-class vulnerability classification task.

1 Introduction

Nowadays, software is ubiquitous in people's lives, permeating nearly every aspect of daily activities. However, as software systems continue to grow in scale and complexity, the types of vulnerabilities are becoming more diverse (Alaoui and Nfaoui, 2022). Common Weakness Enumeration (CWE) is a comprehensive vocabulary for describing and classifying weaknesses in software. Maintained by MITRE[1], its purpose is to aid security experts in identifying and addressing software security vulnerabilities. For example, the standardized terminology provided by CWE enables security experts to accurately report discovered vulnerabilities, offer specific repair recommendations, and determine the severity of vulnerabilities, which helps prioritize and address high-risk vulnerabilities first.

Classifying vulnerability code according to CWE is a challenging task that requires security experts to manually analyze the code and identify the specific types of vulnerabilities. While manual classification provides detailed analysis, it also has several significant drawbacks. First, it relies on the knowledge and experience of security experts. Second, manual analysis is typically slow, making it difficult to meet the demands of large-scale code reviews. For example, among the 28,902 Common Vulnerabilities and Exposures (CVE) entries newly published in 2023 in the National Vulnerability Database (NVD)[2], 4,113 cases still have unidentified types. This underscores the need for automated tools to quickly classify potential vulnerabilities.

Existing Approaches. Many methods have been proposed to classify software vulnerabilities. Traditional methods include static analysis (Lipp et al., 2022) or dynamic analysis (Zaddach et al., 2014; Chen et al., 2016; Zheng et al., 2023). However, they still face issues such as manual feature design, high false positives, and incomplete coverage. Given the significant success of deep learning methods in image recognition and natural language processing(NLP) tasks (Vaswani et al., 2017; Devlin et al., 2018), particularly in terms of automated feature extraction, handling complex patterns, and understanding code semantics, researchers are now exploring their application to the task of classifying code vulnerability types. Some researchers applied

*Corresponding author

[1]https://cwe.mitre.org
[2]https://nvd.nist.gov/vuln/data-feeds

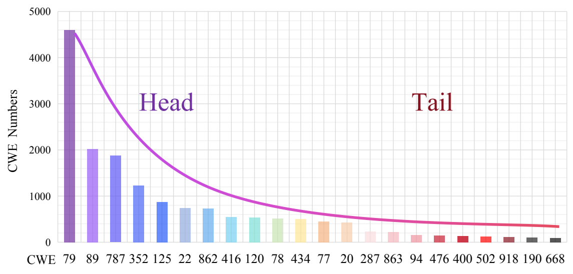

Figure 1: CWE type distribution of the newly published vulnerabilities by NVD in 2023, which follows a long-tailed distribution. (Top 22 CWEs selected)

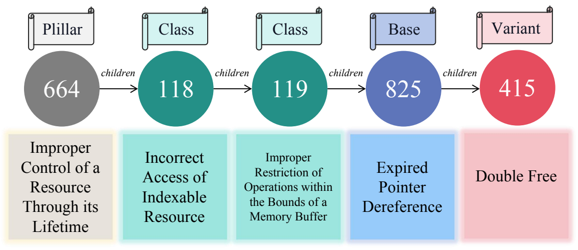
Pillar
Class
Class
Base
Variant
664
children
118
children
119
children
825
children
415
Improper Control of a Resource Through its Lifetime
Incorrect Access of Indexable Resource
Improper Restriction of Operations within the Bounds of a Memory Buffer
Expired Pointer Dereference
Double Free

Figure 2: The refinement chain of CWE-415 from higher to lower abstraction type, according to the amount of specific information in the CWE.

deep learning to software vulnerability detection tasks(Li et al., 2018; Zhou et al., 2019; Chakraborty et al., 2021; Li et al., 2021). However, these methods mostly focus on whether a given code contains vulnerabilities without providing extra information like vulnerability types. Therefore, security researchers still cannot obtain effective information about vulnerabilities from the above models.

Recently, several studies have been proposed (Das et al., 2021; Fu et al., 2023; Wen et al., 2024) to address this issue by leveraging learning-based methods to predict the types of vulnerabilities (i.e., CWE-ID). According to Figure 1, the distribution of CWE-IDs follows a long-tail pattern. Most vulnerabilities belong to a small subset of CWE-IDs, while most CWE-IDs have only a small portion of samples. Zhout et al. (2023) indicated that data with long-tail distributions can significantly impair the effectiveness of learning-based methods. Fu et al. (2023) introduces a transformer-based hierarchical distillation model to handle highly imbalanced CWE labels, while Wen et al. (2024) employs a GNN-based model with an adaptive re-weighting module to better predict the vulnerability types at the tail.

Challenges. Although the above methods have achieved success in the multi-class vulnerability classification tasks, some problems still remain. **Challenge 1:** These works usually target the long-tail distribution problem, but they all treat each CWE class as an isolated class and ignore the relationship between them. In a high-quality code vector space, not only should vectors of the same class be closer together, but vectors of similar classes should also exhibit proximity. This results in the code vector representation obtained by their learning not being scalable. When encountering complex vulnerabilities in real-world scenarios, the performance of the model significantly diminishes. (Ding et al., 2024). **Challenge 2:** Transformer-based models face input length limitation, while current work tends to truncate inputs arbitrarily (Fu and Tantithamthavorn, 2022; Fu et al., 2023). However, according to the dataset (Fan et al., 2020) they used, 73% of the vulnerability codes were longer than the limit their model could accept. This truncation can hinder the model's ability to effectively learn from vulnerabilities whose features exceed the input length limitation.

Solutions. In this paper, we propose a hierarchical contrastive learning framework, combined with geometric spread and max-pooling optimization, to address the above challenges. To address **challenge 1**, we use hierarchical contrastive learning to learn the hierarchical characteristics of CWEs. In the MITRE standard, CWE-IDs are organized hierarchically[3]. This structure is designed to facilitate research on CWEs. Figure 2 illustrates the hierarchical relationships. We use CWE-415 (Double Free) as the lowest level as an example. A precise CWE will have a parent CWE that is more abstract. The highest level of abstraction is called $Pillar$, in the example CWE-664. Then the lower abstraction is $Class$ and $Base$. $Variant$ is the most detailed and the lowest abstraction level. It is important to note that these abstract classes do not strictly adhere to a four-level hierarchy. For instance, a $Variant$ CWE-ID can be a child of a $Pillar$ and is not necessarily restricted to being a child of a $Base$. However, higher-level classes cannot be children of lower-level classes. In recent years, researchers have increasingly favored contrastive learning to achieve better vector representations (Khosla et al., 2020; Gao et al., 2021) in multi-class classification tasks. The central idea of contrastive learning is to obtain a sample representation and pull it closer to the representations of similar samples (positives)

[3]https://cwe.mitre.org/data/definitions/1000

while pushing it away from the representations of other samples (negatives). It aims to ensure that the representations themselves are meaningful rather than merely optimizing the loss function. However, according to Islam et al. (2021), supervised contrastive learning can lead to the problem of class collapse — when every sample from the same class has the same embedding. We use geometric spread, which combines self-supervised contrastive loss (He et al., 2020) with supcon loss (Khosla et al., 2020) in supervised contrastive learning to get a more transferable and robust representation. To address **challenge 2**, we employ max-pooling (Ding et al., 2020) to increase the length that the model can accept. By using max pooling, richer feature representations can be extracted from each data segment to make up for the length limitation.

Evaluation. We conducted extensive experimental evaluations of our framework. Similar to previous work, we performed experiments on the Big-Vul dataset (Fan et al., 2020). The results demonstrate that our framework improves accuracy by 2.97%−17.90% and weighted-F1 by 0.98%−22.27% compared to the baselines. To assess the performance of our framework in more realistic scenarios, we utilized a more recent and precise dataset, PrimeVul (Ding et al., 2024). On this dataset, our framework achieved an accuracy improvement of 4.98%−15.97% and a weighted-F1 improvement of 2.92%−18.94%. These results indicate that the model performs significantly better under near-realistic conditions.

In summary, our contributions are as follows:

- Based on the relationships between CWE types, we use hierarchical supervised contrastive learning to achieve a higher-quality code representation.
- We implement geometric spread by merging self-supervised contrastive loss with supervised contrastive learning to address the class-collapse problem, resulting in more robust representations.
- We use max-pooling to increase the input length, allowing it to learn features of vulnerabilities hidden beyond the transformer-based model's input length limitation.
- We conduct extensive experiments to demonstrate that our model achieves significant improvements compared to baseline methods and performs even better by using newer, higher-quality datasets.

2 Related Work

2.1 Contrastive Learning

Contrastive learning focuses on identifying common features among similar instances and distinguishing differences between dissimilar instances. Compared to general supervised learning, contrastive learning does not need to focus on intricate details of instances. Instead, it only needs to learn to distinguish data at an abstract semantic level in the feature space. Initially, contrastive learning employed a self-supervised approach (Wu et al., 2018; Oord et al., 2018; He et al., 2020), transforming the samples themselves to obtain positive samples, with other samples serving as negatives. Khosla et al. (2020) showed that supervised contrastive learning (where positives are from the same class labels) could be a better approach when it has labeled datasets. Graf et al. (2021) pointed out that supervised contrastive learning can lead to class collapse. Islam et al. (2021) combined supervised contrastive loss with self-supervised contrastive loss to achieve geometric spread, resulting in improved inferability. According to the characteristics of multi-label data, Guo et al. (2022); Wang et al. (2022) also used hierarchical contrast learning to solve the problem of multi-label classification.

Currently, contrastive learning is widely applied in computer vision (Chen et al., 2021) and natural language processing (Gao et al., 2021), and there is substantial work on its application in vulnerability detection (Du et al., 2022, 2023) and commit-level vulnerability type prediction (Pan et al., 2023; Zhou et al., 2023). However, no work has applied it to CWE classification yet.

2.2 Code Vulnerability Type Classification

The vulnerability classification task predicts the vulnerability type based on a given vulnerability statement or related information. Aota et al. (2020); Das et al. (2021) have focused on automatically classifying vulnerability types from textual descriptions, but these descriptions may not always be available. In the early stages of software development, security analysts typically only have access to the source code itself. Our main objective is to propose an end-to-end approach that relies solely on source code statements for vulnerability classification.

Recently, various deep learning methods have been proposed to address the problem of vulnerability classification (Fu and Tantithamthavorn, 2022; Hin et al., 2022). However, most of the existing work focuses on binary classification to predict whether source code is vulnerable or not. Our approach, in contrast, is a multi-class classification based on CWE categories. Fu et al. (2023) have introduced a transformer-based hierarchical distillation model to tackle the highly imbalanced vulnerability types. Wen et al. (2024) have employed a model based on Graph Neural Network with an adaptive re-weighting module. However, these approaches have not considered whether the vectors produced by the models are semantically meaningful, that is, whether similar types of samples are closer in the vector space.

3 Methodology

In this section, we present the details of our novel framework that uses hierarchical supervised contrastive learning to enhance the performance of a transformer-based model in vulnerability classification. The overview of the proposed method is illustrated in Figure 3 and the model architecture is detailed in Section 3.3.

3.1 Problem Formulation

We formalize vulnerability classification as a multi-class classification problem. Let $D = \{(x_i, y_i)\}_{i=1}^{N}$ denotes the set of vulnerability code including N code samples (x_i, y_i). In each sample, $x_i = \{w_i, ..., w_n\}$ denotes the i-th vulnerable function in the form of raw source code and $y_i \in \{0, 1\}^l$ denotes the corresponding label where l is the total number of labels. It aims to learn a mapping $f : x_i \mapsto y_i$ to predict the class of vulnerability.

3.2 Contrastive Loss Function

In supervised learning, cross-entropy loss is commonly employed to train model. In the context of multi-class scenarios, the cross-entropy loss is formulated as follows:

$$\mathcal{L}^{CE} = -\log \frac{\exp(p_i)}{\sum_{c=1}^{C} \exp(p_c)} \tag{1}$$

where C is the number of classes. However, as previously discussed, numerous studies have investigated the limitations of this loss function, particularly when dealing with data characterized by long-tailed distributions. This loss function may lead to poor generalization and instability under such conditions. A high-quality vector representation should bring vectors of similar classes closer together on the hypersphere. Motivated by this intuition, we use a supervised contrastive learning loss function hierarchically, tailored to the hierarchical characteristics of CWEs.

Contrastive learning derives positive and negative samples through certain proxy tasks. For instance, the seminal work of self-supervised contrastive learning InstDisc (Wu et al., 2018), assumes that each image represents a unique class, with all other images belonging to different classes. Subsequently, a model is used to extract features, and the contrastive learning loss function is applied to these features.

3.2.1 Self-Supervised Contrastive Loss

Contrastive learning was first widely used as unsupervised (self-supervised) learning in computer vision (He et al., 2020). Unsupervised contrastive learning first divides the training sample into batches of length n. Let one batch be defined as $\{x_i, y_i\}_{i=1...n}$. Then it performs data augmentation twice on each sample to obtain the augmented batch $\{x_i, y_i\}_{i=1...2n}$. Therefore, the batch size of each training is $2n$, which is called multiviewed batch.

In each multiviewed batch, let $j(i)$ be the index of the corresponding augmented sample originating from the same i-th source sample. Then the self-supervised contrastive loss can be described as follows:

$$\mathcal{L}^{\text{self}} = -\sum_{i \in I} \log \frac{\exp\left(z_i \cdot z_{j(i)}/\tau\right)}{\sum_{a \in A(i)} \exp\left(z_i \cdot z_a/\tau\right)} \tag{2}$$

Here z means the output of the model, τ is a scalar temperature parameter and $A(i) \equiv I \setminus \{i\}$. This loss function considers each sample as an *anchor*, its corresponding augmented sample as *positive*, and all other samples as *negative*.

3.2.2 Supervised Contrastive Loss

In order to make the features of similar images close to each other, we need to use class information to determine which samples belong to the same class. Therefore, *self-supervised* contrastive loss changes to *supervised* contrastive loss. The basis of supervised contrastive learning changed from "whether they come from the same sample" to "whether they belong to the same class." The

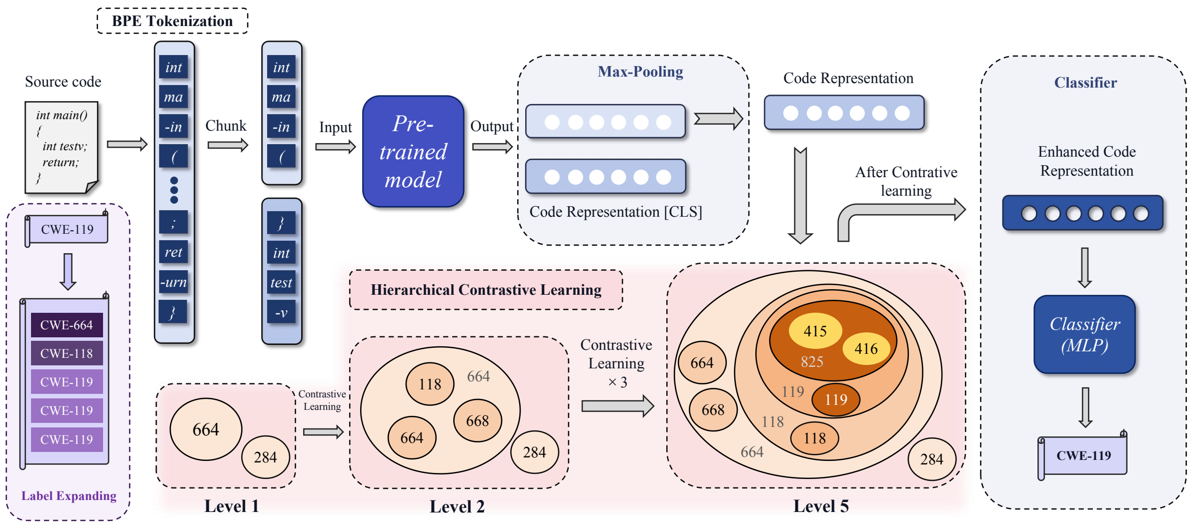

Figure 3: The architecture of our hierarchical contrastive learning framework.

loss function takes the following form:

$$\mathcal{L}_i^{\text{sup}} = \frac{-1}{2N_{\tilde{y}_i} - 1} \sum_{\substack{j=1 \\ j \neq i}}^{2N} \log \frac{\exp(z_i \cdot z_j / \tau)}{\sum_{\substack{k=1 \\ k \neq i}}^{2N} \exp(z_i \cdot z_k / \tau)}$$
$$\text{where } \tilde{y}_i = \tilde{y}_j \tag{3}$$

Here, loss will only be calculated when i and j have the same class. Therefore, samples that have the same class will have closer distances on the hypersphere.

3.3 Model Architecture

Now we present the architecture of our hierarchical contrastive learning framework. First, we expand the CWE labels in the dataset into five levels according to the MITRE standard, with the first level being the most abstract one. Then, we input the source code using Byte Pair Encoding (BPE) for subword tokenization. We employ max-pooling to allow the pretrained model, originally limited to 510 tokens, to process longer sequences. Once the source code is processed through the model to obtain representation vectors, we apply hierarchical contrastive learning to bring the representation vectors of similar classes closer within the hypersphere while pushing the vectors of unrelated classes further apart. Finally, after obtaining the enhanced code representation vector, it passes through the classification head to get the final CWE.

3.3.1 Label Expanding

Given that the CWE labels within the dataset span multiple abstract levels in the MITRE standard, we need a method to assign a set of five-level labels to each sample to facilitate hierarchical contrastive learning. Based on our research, the deepest CWE level of the MITRE standard reaches 5. For example, consider CWE-119: its parent is CWE-118, and its grandparent is CWE-664, which is at the most abstract level, *Pillar*. Thus, the five-level labels for CWE-119 would be {664, 118, 119, 119, 119}. Note that for the lower levels of the original label, the labels remain the same. This approach allows the model to progressively refine from abstract categories down to the specific current label.

3.3.2 Max-Pooling for Long-Text

Next, we input the source code into a pretrained model. Transformer-based models have a limitation on input length, and previous works (Fu and Tantithamthavorn, 2022; Fu et al., 2023; Li et al., 2018) have typically employed truncation methods. The main drawback of truncation is that it discards the extra part of the text, which can adversely affect the model's performance. Pooling, on the other hand, involves replacing the features of one text with the most significant features from that text, thereby receiving longer texts. Initially, we segment the long text into several chunks based on the model's input limitation and place them in the same batch. After obtaining the representation vectors from the model, we apply pooling to the vectors from the same group. Specifically, we use max-pooling to retain the most prominent features, resulting in a final vector that represents the long text.

3.3.3 Hierarchical Contrastive Learning

After obtaining the representation vectors, we first perform contrastive learning based on the first-level CWE labels for each sample. After several epochs, the model totally learns the first-level CWE classes, which do not exhibit a long-tail distribution. Next, the model undergoes contrastive learning with the second-level CWE labels, ensuring that samples with the same first-level label converge and separate mutually. Simultaneously, samples within the same first-level labels maintain a close distance. After five iterations, the model can separate all CWEs in the hyperspace using contrastive learning.

However, pure supervised contrastive learning can lead to the problem of class collapse, where the model learns to distinguish between classes but fails to differentiate between individual samples within a class. To address this issue, we incorporate the cross entropy Loss and self-supervised contrastive loss into the supervised contrastive loss, following the approach of Islam et al. (2021). Thus, the overall loss is a weighted average of these losses, formulated as follows:

$$\mathcal{L} = (1-\lambda-\mu)\mathcal{L}^{CE} + \lambda\mathcal{L}_i^{\text{sup}} + \mu\mathcal{L}^{\text{self}} \quad (4)$$

Through hierarchical contrastive learning, we obtain a higher-quality code vector representation. The representation is then fed into a classification head composed of a fully-connected layer. At last, the framework output the final CWE through softmax.

4 Experiments

4.1 Experimental Settings

4.1.1 Datasets

In current learning-based code vulnerability tasks, C/C++ have been major focuses due to their widespread use in system-level programming and the presence of well-known vulnerabilities like buffer overflows. So, similar to previous studies (Fu and Tantithamthavorn, 2022; Fu et al., 2023), we use the Big-Vul dataset (Fan et al., 2020) for experimental evaluation. Big-Vul extracts vulnerability statements from the Common Vulnerabilities and Exposures (CVE) database by mining 348 open-source GitHub projects. Each vulnerability statement is accompanied by detailed information, such as the CWE-ID, CVSS score (which describes the relative severity of software flaw vulnerabilities), and even the location of the vulnerability statement. Big-Vul comprises approximately 150,000 C/C++ functions, both vulnerable and non-vulnerable. Given that our task is the classification of vulnerability types, we derived a dataset of 8,782 vulnerable functions spanning 88 different CWE categories. However, according to recent research (Croft et al., 2023), popular vulnerability datasets, including Big-Vul, suffer from issues like poor data quality, high redundancy, and low label accuracy, which significantly impact the evaluation of models. Therefore, we utilize a higher-quality dataset, PrimeVul (Ding et al., 2024), to provide a more accurate assessment of model performance under real-world conditions. PrimeVul employs a novel data labeling technique, achieving label accuracy comparable to manual verification benchmarks. It aims to obtain a more accurate assessment of the performance of code pretrained models under real-world conditions.

4.1.2 Baselines

We will compare our approach with several large code pretrained language models, such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020) and CodeGPT (Lu et al., 2021), since our framework is based on these code language models. Additionally, we will compare our method with other vulnerability classification models, such as VulExplainer (Fu et al., 2023) and LIVABLE (Wen et al., 2024). VulExplainer introduces a transformer-based hierarchical distillation model to handle highly imbalanced CWE labels, while LIVABLE employs a GNN-based model with an adaptive re-weighting module. Furthermore, we will also compare our approach with models designed for binary vulnerability detection tasks, including Devign (Zhou et al., 2019) and ReGVD (Nguyen et al., 2022) to verify whether these methods can be transferred to the vulnerability classification task.

4.1.3 Hyperparameter Setting

We divided the dataset into training, validation, and test sets in an 8:1:1 ratio. For the hyperparameters of the baseline approaches, we followed the optimal settings specified by the original authors. Regarding our hierarchical contrastive learning framework, each level was trained for 300 epochs. Due to GPU memory limitations and to ensure that the batch size was not too small, we split the source code into two segments, each 512 tokens in length. These segments were then processed using max-pooling before being fed into the model, effectively doubling the model's length limit. For the propor-

Methods	Accuracy of 5 Levels (Big-Vul)					Weighted F1	Accuracy	Weighted F1
	Tier 1	Tier 2	Tier 3	Tier 4	**Tier 5**	**(Big-Vul)**	**(PrimeVul)**	**(PrimeVul)**
CodeBERT	71.26	68.55	66.08	64.56	63.19	43.07	48.98	28.54
GraphCodeBERT	70.01	69.87	65.73	63.04	62.27	62.74	45.77	35.90
CodeGPT	69.23	69.06	67.13	64.23	63.08	62.30	48.13	36.01
VulExplainer	72.21	70.55	69.16	66.85	66.09	62.93	53.14	38.32
LIVABLE	71.90	70.04	68.31	66.52	64.01	64.36	53.04	36.02
Devign	62.11	58.42	55.02	53.14	51.16	48.71	42.15	22.30
ReGVD	65.24	60.40	59.77	58.67	57.52	56.45	48.35	24.04
Ours (CodeBERT)	**75.76**	**73.94**	**72.81**	**70.91**	**69.06**	**65.34**	**58.12**	**41.24**
Ours (GraphCodeBERT)	73.26	71.12	71.08	69.42	67.13	62.94	56.60	38.07
Ours (CodeGPT)	72.13	70.14	69.87	68.16	66.43	63.86	54.35	40.98

Table 1: Results of our hierarchical contrastive learning method compared with the baselines. Tier 1-5 means the five-level CWE labels. Ours (CodeBERT) means we use CodeBERT to be the pretrained code language model.

Model	HCL	USCL	MP	Acc(B)	Acc(P)
CodeBERT	×	×	×	63.19	48.98
	✓	×	×	66.92	53.13
	✓	✓	×	68.31	57.10
	×	×	✓	63.24	49.42
	✓	×	✓	67.03	53.49
	✓	✓	✓	**69.06**	**58.12**

Table 2: Experimental results of ablation study. HCL denotes hierarchical contrastive learning. USCL denotes the extra unsupervised contrastive learning loss. MP denotes max-pooling to expand input length. Acc(B) denotes the accuracy in Big-Vul, and Acc(P) means the accuracy in PrimeVul.

tions of the components in the loss function, we set λ to 0.3 and μ to 0.2, ensuring that the cross-entropy loss accounted for 50% of the total loss. τ is assigned a value of 0.5. Our experiments were conducted on a server equipped with an NVIDIA RTX 3090 GPU with 24GB of RAM.

4.1.4 Evaluation Metric

We use Accuracy and the weighted F1 score to evaluate the model's performance. Accuracy represents the proportion of correctly classified instances among the total instances. Let TP (True Positive) represent the number of positive examples predicted by the model, and the actual label is also positive, so as FP (False Positive), TP(True Positive), and FN (False Negative). The Accuracy is expressed as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (5)$$

The F1 score is a commonly used metric for evaluating the predictive performance of a model in binary classification tasks, considering both precision ($P = \frac{TP}{TP+FP}$) and recall ($P = \frac{TP}{TP+FN}$). F1 score is the harmonic mean of precision and recall ($F1 = 2 \cdot \frac{PR}{P+R}$). The weighted F1 score calculates the F1 score for each class independently and then takes a weighted average based on the number of instances in each class. This approach addresses class imbalance issues and is defined as:

$$\text{Weighted-F1} = \sum_{i=1}^{N} w_i \cdot \text{F1}_i, \quad w_i = \frac{N_i}{N} \quad (6)$$

4.2 Results

Table 1 presents our experimental results. We applied the proposed hierarchical contrastive learning framework to three pretrained code language models and compared their performance with previous baseline methods on two datasets. The comparison metrics include Accuracy and weighted F1 Score. On the Big-Vul dataset, our hierarchical contrastive learning based on CodeBERT achieved a final accuracy of 69.06%, outperforming all baselines by $2.97\% - 17.90\%$. Additionally, we evaluated the accuracy at each level, finding that each level reached optimal performance. The weighted F1 score also reached the highest value of 65.34%, surpassing all baselines by $0.98\% - 22.27\%$.

Furthermore, we conducted experiments on the PrimeVul dataset, which is of higher quality. Since PrimeVul partially tackles the problems of poor data quality, low label accuracy, and high duplication rates found in Big-Vul, the model outcomes are less inflated by these issues and better represent real-world conditions. While the performance of all models decreased on this dataset, the accuracy of the framework applied to CodeBERT reached 58.12%, exceeding baselines by $4.98\% - 15.97\%$. The weighted F1 score was 41.24%, which was $2.92\% - 18.94\%$ higher than existing baselines, showing a more significant improvement compared to the results on Big-Vul.

4.3 Ablation Study

In this section, we conduct an ablation study to systematically evaluate the contribution of each component in our proposed model and to understand their impact on overall performance. As shown in Table 2, we performed ablation experiments on the three key components of the model: hierarchical contrastive learning, max-pooling, and self-supervised contrastive learning for class collapse optimization. It is important to note that self-supervised contrastive learning is built on top of hierarchical contrastive learning. We use accuracy as the metric to highlight changes in model performance.

Hierarchical Contrastive Learning: By cpmparing row 2 to 1 and row 5 to 4 in Table 2, it can be observed that adding hierarchical contrastive learning increases the accuracy on the Big-Vul and PrimeVul datasets by 3.73% and 4.15%, respectively, without max-pooling. With max-pooling, the increases are 3.79% and 4.07%, respectively, which is a significant improvement. This indicates that hierarchical contrastive learning greatly enhances the model's performance.

Max-Pooling: By comparing row 4 to 1, row 5 to 2, and row 6 and 3 in Table 2, it is evident that the use of max-pooling improves the average accuracy by 0.47%. This demonstrates that max-pooling also contributes positively to the model's effectiveness.

Class Collapse Optimization: By comparing row 3 to 2 and row 6 to 5 in Table 2, it can be seen that adding the self-supervised contrastive learning loss increases the accuracy by an average of 1.71% on the Big-Vul dataset and by 4.3% on the higher-quality PrimeVul dataset. This improvement suggests that self-supervised contrastive learning helps to better disperse code vectors, thereby avoiding the problem of class collapse.

Additionally, we tested the model's sensitivity to hyperparameters. As shown in Figure 4, we first determined the optimal value for the hyperparameter λ, which represents the proportion of the contrastive learning loss in the total loss function. We found that a value of 0.5 yields the highest accuracy, and other values do not significantly impact the model's performance. Since the hyperparameter μ is an enhancement based on the supervised

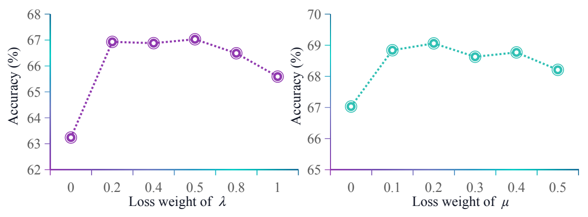

Figure 4: Model's sensitivity to loss weights λ and μ.

Prompt Setting	R(Acc)	T(Acc)	F(Acc)
Zero-shot	22%	25%	16%
Two-shot	34%	23%	27%
Chain-of-Thought	24%	30%	25%

Table 3: Experimental results of GPT-4o by using zero-shot prompting, two-shot prompting and chain-of-thought prompting. R(random) denotes 100 randomly selected samples from the entire dataset, T(true) represents 100 randomly selected samples correctly predicted by our model, and F(false) indicates 100 randomly selected samples that were incorrectly predicted.

contrastive learning loss, we only tested its effect in the range of 0 to 0.5. The results show that the value of μ between 0.1 and 0.5 does not greatly affect the performance of the model. The results indicate that our model is robust to hyperparameter settings.

The ablation study demonstrates that each component of our proposed model contributes to its performance. Hierarchical contrastive learning is particularly crucial, while self-supervised contrastive learning and max-pooling also play significant roles in enhancing model performance. We also verified that the model is not sensitive to the hyperparameter ratio of the contrastive learning loss.

4.4 Prompt Learning

Given the recent advancements in large language models, in this section, we compare our model with the popular large language model ChatGPT (GPT-4o) (OpenAI, 2023) to evaluate the effectiveness of zero-shot learning for the vulnerability classification task. we conducted experiments with three settings: zero-shot prompting, two-shot prompting and chain-of-thought prompting. We use the same samples as those in our paper. We selected 100 samples each. We used the following prompt(zero-shot): "*Please identify the CWE type of the following vulnerable function: {code}*". The other two prompts will show in the appendix.

As shown in Table 3, GPT-4o only successfully predicted 22%, 25% and 16% of the samples in the R, T and F sample sets by using zero-shot prompting. After the simple prompt design, the accuracy of LLM predictions did improve compared to zero-shot. However, the accuracy remains poor. The results show that the state-of-the-art language model, without fine-tuning specifically for the vulnerability classification task, does not perform well. This demonstrates that our model has a significant advantage in this specific task.

5 Conclusion

We propose a multi-class vulnerability type classification framework based on hierarchical contrastive learning to bring the vector representations of highly related CWEs closer together. Additionally, we employ geometric spread to address the class collapse problem and max-pooling to receive a longer input length, thereby enhancing the model's robustness. The results demonstrate that our hierarchical contrastive learning framework outperforms current baseline methods across various metrics. This improvement is primarily due to advancements in code representation provided by our method and these better quality vectors can be used in subsequent works.

Limitations

Finally, we outline the current limitations of our approach to guide future improvements. First, learning-based methods are highly dependent on the quality of the dataset. Existing vulnerability classification datasets still suffer from high error rates and significant redundancy, indicating an urgent need for the release of a high-quality dataset. Additionally, about 26% of the code samples exceed the 1024 token length limit we set, according to our statistics. So our method still struggles with these very long vulnerability codes, suggesting that a better solution should be designed to address the issue of handling long texts in transformer-based models. In the future, we plan to process source code in a way that effectively exposes information related to vulnerabilities, allowing large language models to better learn the relevant features. Also, we aim to utilize other information related to vulnerability code, such as CVE descriptions when available. By integrating multiple sources of information, it could be possible for the model to distinguish between different CWEs more accurately.

Acknowledgement

This work was supported by National Key Research and Development Program of China (No.2023YFB3106400, 2023QY1202), the National Natural Science Foundation of China (U2336203, U1836210), the Key Research and Development Science and Technology of Hainan Province (GHYF2022010), and Beijing Natural Science Foundation (L242015, 4242031).

References

Rokia Lamrani Alaoui and El Habib Nfaoui. 2022. Deep learning for vulnerability and attack detection on web applications: A systematic literature review. *Future Internet*, 14(4):118.

Masaki Aota, Hideaki Kanehara, Masaki Kubo, Noboru Murata, Bo Sun, and Takeshi Takahashi. 2020. Automation of vulnerability classification from its description using machine learning. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7. IEEE.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296.

Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, volume 1, pages 1–1.

Xinlei Chen, Saining Xie, and Kaiming He. 2021. An empirical study of training self-supervised vision transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9640–9649.

Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 121–133. IEEE.

Siddhartha Shankar Das, Edoardo Serra, Mahantesh Halappanavar, Alex Pothen, and Ehab Al-Shaer. 2021. V2w-bert: A framework for effective hierarchical multiclass classification of software vulnerabilities. In *2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–12. IEEE.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Ming Ding, Chang Zhou, Hongxia Yang, and Jie Tang. 2020. Cogltx: Applying bert to long texts. *Advances in Neural Information Processing Systems*, 33:12792–12804.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*.

Qianjin Du, Xiaohui Kuang, and Gang Zhao. 2022. Code vulnerability detection via nearest neighbor mechanism. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 6173–6178.

Qianjin Du, Shiji Zhou, Xiaohui Kuang, Gang Zhao, and Jidong Zhai. 2023. Joint geometrical and statistical domain adaptation for cross-domain code vulnerability detection. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Michael Fu, Van Nguyen, Chakkrit Kla Tantithamthavorn, Trung Le, and Dinh Phung. 2023. Vulexplainer: A transformer-based hierarchical distillation for explaining vulnerability types. *IEEE Transactions on Software Engineering*.

Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*.

Florian Graf, Christoph Hofer, Marc Niethammer, and Roland Kwitt. 2021. Dissecting supervised contrastive learning. In *International Conference on Machine Learning*, pages 3821–3830. PMLR.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Yuanfan Guo, Minghao Xu, Jiawen Li, Bingbing Ni, Xuanyu Zhu, Zhenbang Sun, and Yi Xu. 2022. Hcsc: Hierarchical contrastive selective coding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9706–9715.

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. 2020. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738.

David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607.

Ashraful Islam, Chun-Fu Richard Chen, Rameswar Panda, Leonid Karlinsky, Richard Radke, and Rogerio Feris. 2021. A broad study on the transferability of visual representations with contrastive learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 8845–8855.

Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *Advances in neural information processing systems*, 33:18661–18673.

Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 292–303.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*.

Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 544–555.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 178–182.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*.

OpenAI. 2023. Chatgpt: Optimizing language models for dialogue. Accessed: 2024-06-01.

Shengyi Pan, Lingfeng Bao, Xin Xia, David Lo, and Shanping Li. 2023. Fine-grained commit-level vulnerability type prediction by cwe tree structure. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 957–969. IEEE.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Zihan Wang, Peiyi Wang, Lianzhe Huang, Xin Sun, and Houfeng Wang. 2022. Incorporating hierarchy into text encoder: a contrastive learning approach for hierarchical text classification. *arXiv preprint arXiv:2203.03825*.

Xin-Cheng Wen, Cuiyun Gao, Feng Luo, Haoyu Wang, Ge Li, and Qing Liao. 2024. Livable: Exploring long-tailed classification of software vulnerability types. *IEEE Transactions on Software Engineering*.

Zhirong Wu, Yuanjun Xiong, Stella X Yu, and Dahua Lin. 2018. Unsupervised feature learning via non-parametric instance discrimination. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3733–3742.

Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. 2014. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, volume 14, pages 1–16.

Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. {FISHFUZZ}: Catch deeper bugs by throwing larger nets. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1343–1360.

Jiayuan Zhou, Michael Pacheco, Jinfu Chen, Xing Hu, Xin Xia, David Lo, and Ahmed E Hassan. 2023. Colefunda: Explainable silent vulnerability fix identification. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2565–2577. IEEE.

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. *Advances in neural information processing systems*, 32.

Xin Zhout, Kisub Kim, Bowen Xu, Jiakun Liu, Dong-Gyun Han, and David Lo. 2023. The devil is in the tails: How long-tailed code distributions impact large language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 40–52. IEEE.

A Appendix

The following is the two-shot prompting template:

You are a security expert that is good at static program analysis.

Please analyze the following code:
*static char *clean_path(char *path)*
{...}
Please indicate which CWE type this vulnerable code belongs to (Only reply with CWE-ID. Do not include any further information):
CWE-119

Please analyze the following code:
int64 ClientUsageTracker::GetCachedHostUsage
{...}
Please indicate which CWE type this vulnerable code belongs to (Only reply with CWE-ID. Do not include any further information):
CWE-118

Please analyze the following code:
<INSERT NEW CODE HERE>
Please indicate which CWE type this vulnerable code belongs to (Same as above):

The following is the chain-of-thought prompting template:

You are a security expert that is good at static program analysis.

Please analyze the following code:
<INSERT NEW CODE HERE>
Please indicate which CWE type this vulnerable code belongs to (Only reply with CWE-ID. Do not include any further information):

Let's think step-by-step.