COCL: An Intelligent Framework for Enhancing Deep Learning-based Vulnerability Detection

Wenxuan Li, Shihan Dou, Yueming Wu, Chenxi Li, and Yang Liu

***Abstract*— Due to the powerful feature extraction capability of *deep learning* (DL), many recent studies have used it to conduct source code vulnerability analysis. However, although it has good performance on artificial datasets, it does not perform satisfactorily on the real-world vulnerabilities with higher complexity. In this paper, we introduce contrastive curriculum learning into DL-based vulnerability detection to find a suitable boundary to distinguish vulnerabilities from normal codes. Contrastive learning can be used to reduce the difference between different vulnerabilities while amplifying the difference between vulnerabilities and normal codes. To make the training phase of contrastive learning more intelligent, we apply curriculum learning to mimic the way humans acquire knowledge, which means that the model will learn simple samples first and then increase the difficulty of training samples. Specifically, we implement an intelligent framework (*i.e., COCL*) that can enhance the detection effect of existing DL-based vulnerability detectors. To verify the capability of *COCL*, we select four state-of-the-art DL-based vulnerability detectors (*i.e., AutoVulTC*, *VulDeePecker*, *BenchSG*, and *Devign*) as our base models. The experimental results show that using *COCL* can bring an improvement of 8.1% to the F1 scores of these models on a real-world vulnerability dataset.**

***Index Terms*— Vulnerability Detection, Deep Learning, Contrastive Learning, Curriculum Learning**

I. INTRODUCTION

In recent years, the greatly increased complexity of computer systems makes both possibility and diversity of software vulnerabilities greatly increased, the wide application of open source libraries and sharing devices also make software vulnerabilities' propagation speed increase dramatically. According to a recent report[1], the number of vulnerabilities released by the National Vulnerability Database (NVD) has risen for four consecutive years. As a matter of fact, software vulnerability detection is designed to discover weak snippets in source code and prevent attackers from exploiting these snippets to gain unauthorized access to the system. If the source code can be detected before launching, the potential damages caused by vulnerabilities may be avoided. Therefore, software vulnerability detection has great significance to the security of the software system.

Generally, there are mainly two categories of source code vulnerability detection technology, code-similarity-based approaches [1] and pattern-based methods [2]. Code-similarity-based approaches can only figure out vulnerabilities caused by code cloning. To this end, these detectors calculate the similarity between object code and known vulnerabilities, and report the object source codes which have a similarity score higher than the threshold, these source codes are more likely to have a cloned vulnerability. Since they are designed to discover similar vulnerabilities, they cannot find new vulnerabilities and have limitations in practical applications. To detect more new vulnerabilities, pattern-based techniques are proposed. For pattern-based methods, learning algorithms take patterns produced by professional or machines as inputs and output the vulnerabilities detection results. At the early stage of pattern-based vulnerability detection [3], [4], the only way to get vulnerability patterns is handmaking by experts, which is laborious and lacking generalization. Fortunately, the emergence of deep learning (DL) makes up for this defect well.

Deep learning can extract features from source code through iterative learning, that are more complex and representative than those handmade. This advantage promotes researchers [5]–[7] using deep learning to automatically learn high-latitude features with strong generalization and relevance to the context of source code. For example, *VulDeePecker* [8] first breaks the source code into program slices, and adopts a *bidirectional long short-term memory* (BiLSTM) neural network to detect vulnerabilities. *Funded* [9] processes the source code through complex procedural analysis into an enhanced *abstract syntax tree* (AST), and then uses a *graph neural network* (GNN) to train a classifier. Although these above methods have an ideal performance on the Software Assurance Reference Dataset (SARD), the effect is not satisfactory on real-world vulnerability datasets. This happens mainly because the major purpose of SARD is academic research, and the simple synthetic vulnerabilities are sufficient, while the real-world vulnerabilities have higher complexity and diversity, making it difficult to be distinguished [10].

To improve the ability of existing methods on detecting real-world vulnerabilities, we propose to use *Contrastive Learning* in DL-based vulnerability detection. Contrastive learning can gather homogeneous features and distinguish non-homogeneous features in feature space as far as possible. Its goal is to maximize the similarity between positive samples and minimize the similarity between negative samples. However, in the training procedures, contrastive learning simply presents samples in a random order. But the fact is that, for a special model, different training samples have different

W. Li and S. Dou are with Fudan University, China; Y. Wu and Y. Liu are with Nanyang Technological University, Singapore; C. Li is with China University of Geosciences, China.

Y. Wu is the corresponding author, Email: wuyueming21@gmail.com.

[1]https://nvd.nist.gov/.

learning difficulties. For example, some negative samples may be similar to positive samples, making it hard for the model to distinguish them. In other words, these negative samples have high learning difficulties. Coincidentally, there is a similar situation in the process of human ingestion of knowledge, and our solution is not learning difficult knowledge at first, but starting from simple samples and gradually join samples with higher difficulty. The phased learning from primary school to university is the application of this method. In this way, we can understand what we learned better. In practice, certain studies [11] have proved that a model can get better learning effect if it can have a phase training procedure. They formally define this approach as *Curriculum Learning*. Its basic idea is to imitate the way humans learn, and gradually add difficult samples during the training process to give the model more buffer space for understanding.

In this paper, we introduce *Contrastive Curriculum Learning* into DL-based vulnerability detection. Specifically, we implement two different difficulty calculators to sort our training samples. One is to use the source code complexity as the learning difficulty of the sample, and the other is to calculate the similarity between the original sample and the corresponding negative sample as the learning difficulty of the negative sample. We implement a novel framework (*i.e., COCL*) that can enhance the detection ability of DL-based vulnerability detectors. To examine the ability of *COCL*, we choose four state-of-the-art DL-based vulnerability detectors (*i.e., AutoVulTC* [7], *VulDeePecker* [8], *BenchSG* [12], and *Devign* [5]) as our base models. According to the experimental results, we find that introducing *COCL* during training phase can make these detectors discover 9.6% more real-world vulnerabilities on average.

In summary, the contributions of this paper are:

- We introduce *Contrastive Curriculum Learning* into DL-based vulnerability detection to improve their detection ability.
- We implement *COCL*, a model-agnostic framework that can enhance the effectiveness of DL-based vulnerability detection models.
- We use a real-world vulnerability dataset and four state-of-the-art DL-based vulnerability detectors (*i.e., AutoVulTC*, *VulDeePecker*, *BenchSG*, and *Devign*) for the evaluations. Experimental report shows that the use of *COCL* can make them detect 9.6% more vulnerabilities on average.

Paper organization. The remainder of the paper is organized as follows. Section II describes the related work. Section III introduces our system. Section IV reports the experimental results. Section V concludes the present paper.

II. RELATED WORK

A. Vulnerability Detection

Generally, there are mainly two categories of source code vulnerability detection technology: code-similarity-based approaches and pattern-based techniques. Code-similarity-based vulnerability detection methods only detect cloned vulnerabilities, they use different code similarity analysis algorithms based on different data structures (*e.g.,* tree-based [13], token-based [14], and graph-based [15]). In the early stage of pattern-based methods [2], vulnerability patterns are usually defined by human experts which is laborious and lacks generalization. To achieve more intelligent vulnerability analysis, researchers [5], [7], [8], [16] begin to use deep learning to detect vulnerabilities. For example, *VulDeePecker* [8] first breaks the source code into program slices, and adopts a *bidirectional long short-term memory* (BiLSTM) neural network to detect vulnerabilities using the way it treats text. *DeepWukong* [16] processes program semantics into a program dependency graph, then based on the program interest points, the dependency graph is splitted into multiple subgraphs. Finally, a *graph neural network* (GNN) takes these subgraphs to train a detector.

B. Contrastive Learning

The idea of contrastive loss was generated by [17] in 2006, instead of matching samples to fixed targets, it maximize or minimize the similarities of representation pairs for optimization. Then Mikolov *et al.* [18] introduce the first contrastive loss NCE in 2013, and used it for *natural language processing* (NLP). In practice, contrastive learning has been used in many other domains and is at the core of several recent works. For example, [19] combines contrastive and predictive coding to improve the accuracy of image classification. [20] designs a lightweight self-supervised representation learning model of audio based on contrastive learning and reinforcement learning. Contrastive learning was used mostly for unsupervised or self-supervised represent learning field, however, [21] finds that label information can be used to enhance the training effect with contrastive loss and proposes a supervised contrastive loss form. Therefore, in this paper, we adopt supervised contrastive learning to improve the accuracy of vulnerability detection in source code.

C. Curriculum Learning

Referring to the human learning process, Elman [22] first studies the effect of learning order on model's effectiveness, and from his experimental results, he finds that if the training samples were sorted from simple to complex, the model could learn more grammar than those random sorted. In 2009, Bengio *et al.* [23] officially name this training method as *curriculum learning* (CL) and prove that curriculum learning improves both the study effectiveness and the convergence speed of the model. Today, curriculum learning has been widely used in many fields [24]–[26]. For example, in the computer vision field, [25] adopts CL to enhance the DNN's training on large-scale weakly supervised images in the web. In the medical field, [24] uses reinforcement learning and CL to predict where the emergency patients will be admitted after diagnosis in a hospital. In the translation field, [26] exploits uncertainty-aware CL to improve neural machine translation's translation quality and convergence.

III. METHOD

In this section, we introduce our proposed framework namely *COCL* that can improve the detection ability of DL-based vulnerability detectors.

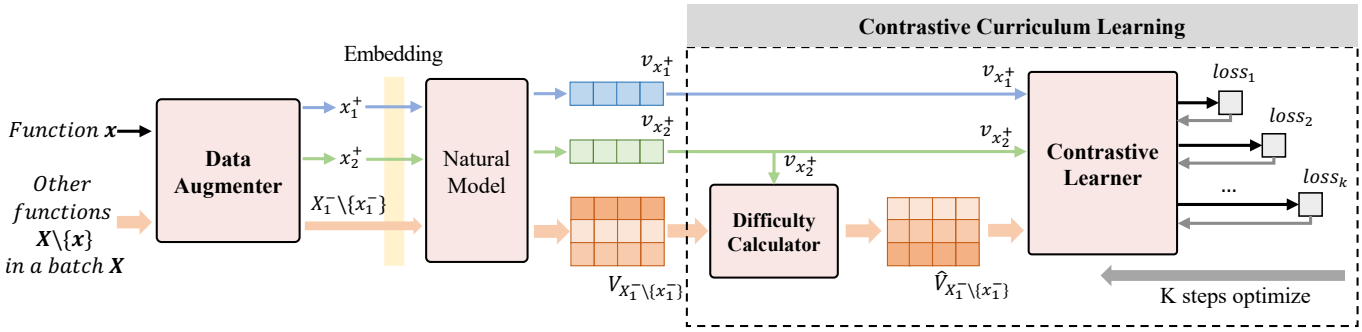


Fig. 1: System overview of *COCL*. Here, $Xnfxg$ is the complementary set of fxg to X, and $x^+;x$ is the positive and negative augmented function of anchor function x, respectively.

A. Overview

As show in Figure 1, *COCL* can be divided into three parts: *Data Augmenter*, *Difficulty Calculator*, and *Contrastive Learner*.

Data Augmenter: For raw data, data augmenter applies semantic-equivalent code transformations to generate corresponding data variants which are used to construct the positive pairs required for contrastive learning.

Difficulty Calculator: For training dataset, difficulty calculator calculates the learning difficulty for each training sample which are required for curriculum learning.

Contrastive Learner: For a batch of data, contrastive learner first sorts the negative samples by difficulty, then divides them into several levels and combines them with positive samples to complete contrastive learning.

During the training process, the Data Augmenter first generates augmented data for each data point, and the processed embeddings enter the feature learning process of the Natural Model. According to the difficulty ranking of the samples calculated by the Difficulty Calculator, negative samples are sequentially introduced in increasing numbers, along with their corresponding positive samples, in order of increasing difficulty, to participate in the contrastive learning process during the k-step optimization.

B. Data Augmenter

Contrastive learning needs data augmentation technology to produce positive samples (or positive pairs). Data augmenter only exploits the data sample itself to build new data samples without any other information, and the new data sample has the same label as the original sample. This technology has been widely used in the machine learning field to add more new samples into training progress as the number of labeled samples is limited, thus improving the model's generalization. Data augmentation has different implementation details on different types of data. For example, images can use rotation, cropping, and grayscale adjustment, but graph-type data should use node-dropping or edges-removal. In DL-based vulnerability detection, the data type is source code. Source code has both semantic and structural information, but we only need to keep its semantic information. Therefore, our data augmentation skill will change the structure of some code snippets but produce semantic-equivalent variants.

We have two expectations for data augmentation techniques we will use. First, the trained model should be resilient to common code modifications without losing semantic invariance. Some DL-based vulnerability detection techniques apply abstraction like mapping user-defined variable names to symbolic names (*e.g.,* VAR) to the source code, which brings a normalization effect. Second, the data augmentation techniques should work well with some code analysis methods, which are usually used to distill the program semantics into different code representation (*e.g.,* AST and CFG), these meaningful representations cooperate with different network structures (*e.g.,* GNN) can improve the training effect of the model. Moreover, we observe that when implementing an algorithm, the code of the same functionality may have different implementation details. Therefore, we design some code conversion rules that can produce semantic-equivalent code variants with different control flow structures.

We summarize the code transformation rules into a set of atomic operations. Thses operations can be categorized into three types: function substitution, code split, and junk code.

1) *Function Substitution*: Using keywords with the same functionality to substitute. For example, we can use for-loop to do the same thing as while-loop or use nested if-else to replace if-elif-else.
2) *Code Split*: Splitting the complex decision statements into several simple decision statements.
3) *Junk Code*: Adding junk code that makes no sense, like some true or false statements.

TABLE I: Atomic operations: function substitution (loop)

Type	For	while	do-while
loop	Body A for(i=0;i¡10;i++)*f* Body A *g*	Body A i=0;while(i¡10)*f* Body A i++;*g*	i=0; do*f* Body A *g* while(i¡10);

TABLE II: Atomic operations: function substitution (condition)

Type	if-elif-else	if-else
condition	if(x¡a) Body A else if(x¡b) Body B else Body C	if(x¡a) Body A else*f* if(x¡b) Body B else Body C *g*

For a given program, we first traverse the whole source code and find out all the statements that satisfy the code transformation rules. After collecting all the code statements that can be converted, we perform code transformations on

TABLE III: Atomic operations: code split

Type	complex	simple

condition	if(x¡a & x ¿b)f
	Body A
	g
		if(x¡a)f
		if(x¿b)f
		Body A g g

TABLE IV: Atomic operations: junk code

Type	original	added junk code
In	Body A	if(True)f Body A g

Out	while(x¡a)f
	Body A
	g
		while(x¡a)f
		Body A
		if(False) Break; g

these code statements according to the transformation rules. Here, we transform all the code statements that satisfy transformation rules, and when a statement satisfies two or more transformation rules, we will use all the satisfied rules in turn to produce multiple semantic invariants to provide richer semantic possibilities.

As Figure 1 shows, we first input a function x into *Data Augmenter* and collect the positive pairs (x_1^+, x_2^+), then input the other functions Xnx in the same batch to obtain the corresponding augmented functions $X_1\ nx_1$. These augmented functions go through the embedding layer, then the model gets embeddings as input, processes, and extracts the features.

C. Difficulty Calculator

Because the ability of a model to learn different samples is different, we need to choose a reasonable way to evaluate the learning difficulty of samples and distinguish them. Here, we introduce two different methods to assess the learning difficulty of a function. One is a static-based method, that is, we directly calculate the complexity of the source code and use it as the learning difficulty of the corresponding training samples. The lower the code complexity, the simpler the sample. The other is a dynamic-based method, that is, we compute the similarity between negative samples and positive pairs from the point of view of their feature vectors. The lower the sample similarity, the simpler the sample.

1) Static-based Difficulty Calculator

Static-based Difficulty Calculator directly analyzes source codes to calculate the learning difficulty of samples. The samples with simpler source code are considered easier to learn, while those samples which have complex source code are difficult samples.

The prior work [27] has shown the overall complexity of a program can be expressed in terms of program maintainability. A complex program is harder to maintain than an easy program. It is the reason why we leverage *Maintainability Index* to be a standard of source code complexity, which also represents the learning difficulty of the samples. *Maintainability Index* [28] is a metric mostly derived from practice, developed by Hewlett-Packard and its software teams[2]. It mainly contains three indicators: 1) Lines of Code, 2) Cyclomatic Complexity [29], and 3) Halstead Volume [30]. According to certain previous works [28], the calculation formula of *Maintainability Index* is as follows:

$$MI = 171 \quad 5.2\ InV_{LC} \quad 0.23\ InV_{CC} \quad 16.2\ InV_{HV} \quad (1)$$

Note that V_{LC} represents the number of Lines of Code, a straightforward metric used to measure the complexity of codes. By excluding comments and blank lines, the remaining code can be equated to the size of the program. Huge programs are more complex than tiny programs. V_{CC} represents the Cyclomatic Complexity, that is, the number of judgment statements contained in the code. Programs that make more decisions are more complex than those straightforward. V_{HV} represents the Halstead Volume, its value is related to the number of variables contained in the program and the frequency of the according calls. Large volume means complex program.

In brief, lower *Maintainability Index* (MI) means more complex code and higher learning difficulty.

2) Dynamic-based Difficulty Calculator

Dynamic-based Difficulty Calculator compares the similarity between the negative samples and the corresponding positive samples to calculate the learning difficulty of negative samples. Here, in a multi-viewed batch of data, we have the anchor embedding z_i and K negative embeddings fz_kg, and we define a similarity metric function $sim(\ )$ to calculate the similarity of negative pair $fz_i; z_kg$, that is $sim(z_i; z_k)\ 2\ \mathbb{R}$. Specifically, we choose one most common similarity function (*i.e.,* cosine similarity) to commence our similarity calculation.

$$sim(z_i; z_k) = \frac{z_i \quad z_k}{jz_ij \quad jz_kj} \quad (2)$$

After obtaining the similarity scores between all negative samples and a positive sample in a batch, the learning difficulty is directly linked to the similarity, that is, we regard samples with lower similarity as simple samples, samples with higher similarity as difficult samples.

D. Contrastive Learner

1) Contrastive Learning

Contrastive self-supervised learning techniques learn sufficiently representative representations by finding and encoding what makes two samples the same or different. These techniques only use information from data itself rather than labels, thus becoming popular when labeled data can be expensive. The motivation of contrastive learning is that humans not only learn from positive signals but also benefit from correcting bad behavior. Therefore, it cares about both the similarity and dissimilarity among data, unlike cross-entropy only concerns the similarity between the model's outputs and labels. Formally speaking, contrastive learning aims to get an encoder to produce data features with more uniqueness while not losing essence. For any data sample x (referred as anchor sample) and its feature $f(x)$, it intends to complete a task as equation (3).

$$\mathrm{SIM}(f(x); f(x^+)) >> \mathrm{SIM}(f(x); f(x\ )) \quad (3)$$

Here, x^+ is a data sample similar to x (referred to as a positive sample), typically a data-augmented version of x, and x is a data sample dissimilar to x (referred to as a negative sample). $f(x^+); f(x\ )$ is the feature of x^+, x separately, and SIM( ) is a metric to measure the similarity between two

[2]https://www.codegrip.tech/productivity/a-simple-understanding-of-code-complexity.

features. The underlying mechanism of contrastive learning aims to minimize the distance between anchor samples and positive samples in the embedding space, while maximizing the distance between anchor samples and negative samples. This implies that the similarity between anchor and positive samples should be significantly greater than the similarity between anchor and negative samples.

Since the excellent classification effects of contrastive learning, it flourishes in the self-supervised representations learning field. Although the original intention of contrastive learning is to save the cost of labeling data and use data's information to classify, its role is more than that. From another angle, if we can combine the enhanced effect of contrastive learning on classification with label information, the effects can be further enhanced in the supervised representations learning field. In fact, there are already some studies [21] on the supervised contrastive learning area. Through their empirical results, they observe that supervised contrastive learning can achieve better effectiveness than self-supervised contrastive learning. When the supervised learning effect has been improved by the enhancement of contrastive learning, contrastive learning is also leveraging label information to alleviate the wrong clustering rate caused by overfitting. Therefore, we adopt the loss function of *Supervised Contrastive Learning* (SupCon) [21] as our loss function.

SupCon calculates contrastive loss in batches. For every input batch of the training data, SupCon first obtains two copies of the batch by using data augmentation twice and combines these two augmented batches to form a multi-viewed batch. For each anchor sample x_i with label l_i, the positive samples are all samples labeled l_i in the same batch and the negative samples are the remaining samples whose labels are different from l_i. Within a multi-viewed batch of data, let $i \ 2 \ I \quad f1;:::;2Ng$ be the index of an arbitrary augmented sample, the loss takes the following form.

$$L_{out}^{sup} = \sum_{i2I} \frac{1}{jP(i)j} \sum_{p2P(i)} log \frac{exp(z_i \ z_p= \ )}{\sum_{a2A(i)} exp(z_i \ z_a= \ )} \quad (4)$$

Here, z_i represents the embedding of data x_i and $2 \ R^+$ is a scalar temperature parameter. $A(i) \quad I=fig$ and $P(i)$ $fp \ 2 \ A(i) : l_p = l_ig$ is the set of indices of all positives in the batch apart from i, $jP(i)j$ is its cardinality.

The equation (4) uses multiple positive and negative samples per batch, and brings more information advantages to contrastive learning. It uses the positive normalization factor(*i.e.,* $\frac{1}{jP(i)j}$) to remove bias present in multiple positives samples and preserve the summation over negatives in the denominator to increase performance.

SupCon just treats all the negative samples consistently, however, the learning difficulties of samples are different. If we can grade and join them into the training progress step by step, the model may find a better way to distinguish features. To tackle the issue, we introduce curriculum learning into contrastive learning.

2) Curriculum Learning

As aforementioned, when contrastive learning selects negative samples, it just treats all the samples as the same and uses a random strategy to sample. But in our humans' daily life, we never treat all the learning objects to the same degree. To better understand the role of curriculum learning, we give a simple example. When studying mathematics, we do not start with advanced mathematics, but first learn addition, subtraction, multiplication, and division in elementary school, calculus in high school, and finally advanced mathematics in college. This intuition also applies to machine learning. In this part, we illustrate the way we introduce the human-learning manner into the model's training process to enhance the effect of contrastive learning. In brief, we combine curriculum learning with contrastive learning to improve the form of the loss function (*i.e.,* equation (4)).

According to the process shown in Figure 1, after getting the feature vectors $V_{X_1^- nx_1^-}$ extracted by the Natural Model, we input these features into *Difficulty Calculator* to obtain the difficulties, and finally have the sorted feature vectors $\hat{V}_{X_1^- nx_1^-}$. Then we use these sorted vectors as input to the *Contrastive Learner* to complete our optimization process. In each epoch, we perform a total of K steps optimizations, and each optimization process takes a certain proportion of negative samples with lower learning difficulties, both the difficulties and the number of negative samples increase as the optimization process progresses. The loss function of step k is the form of the equation.

$$L_{COCL;k} = \quad \sum_{i2I} L_{k;i}^{sup} + c \quad L_s \quad (5)$$

Here, $L_s = \quad \sum_{i2I} logy_i \quad M(z_i)$ is the cross-entropy loss, y_i is the label of sample z_i and $M(z_i)$ means the logit of z_i. c is a scalar value that determines whether to join L_s into the training process, only when $k = K$, c is equal to 1, and the rest of steps c is 0. $L_{k;i}^{sup}$ represents the advanced form of equation (4) in k-th step, which we describe in equation (6) and is its weight.

$$L_{k;i}^{sup} = \sum_{i2I} \frac{1}{jP(i)j} \sum_{p2P(i)} log \frac{exp(z_i \ z_p= \ )}{\sum_{a2A(k;i)} exp(z_i \ z_a= \ )} \quad (6)$$

where $A(k;i) = p(I=fig;k)$

The only difference between equation (6) and equation (4) is $A(k;i)$, it is fixed to a certain sample x_i in equation (4) while related to steps in equation (6). Here, $p(\ ;k)$ is a pacing function to schedule how the negative samples are introduced to the training procedure. Since the samples in batch I have been sorted in increasing order, the sample with index j is the j-th east sample in batch I. Therefore, the result of $p(N;k)$ can be explained as $fz_1;z_2;:::z_{N \ \frac{k}{K}}g$, a linear relationship with step k. More specifically, if we have the total optimization steps to be 4, then in step 1 we would have the first 25% easiest samples and in step 3 we have the first 75%. In fact, the pacing function has a variety of forms like logarithmic and quadratic, but sometimes complexity doesn't mean better. A linear function is not only simple but also powerful, which has been proven in [21].

After completing contrastive curriculum learning, we can obtain an enhanced version of an original DL-based vulnerability detector. In other words, the output of *COCL* is an

enhanced vulnerability detector.

IV. EVALUATIONS

Since the main components of *COCL* are *Contrastive Learning* module and *Curriculum Learning* module, we focus on examining their effectiveness in this section.

A. Experimental Setup

1) Dataset

We choose a real-world vulnerability dataset [5] which is widely used for vulnerability detection jobs. The main sources of the dataset are two open source software, qemu[3] and ffmpeg[4]. It contains 14,858 normal functions (10,070 from qemu and 4,788 from ffmpeg) and 12,460 Vulnerable functions (7,479 from qemu and 4,981 from ffmpeg), which took 600 man-hours of a professional security team to build it.

2) Selected Vulnerability Detectors

To enable a more comprehensive evaluation of *COCL*, we select four state-of-the-art DL-based vulnerability detection methods using different code representations respectively. These four detectors are as follows: ***AutoVulTC*** [7] is a vulnerability detector based on token. It first uses lexical analysis to obtain tokens of a program, then inputs these tokens into a *convolutional neural network* (CNN) to train the detection model; ***VulDeePecker*** [8] is a vulnerability detector based on program slices. It first uses static analysis to extract the program slices, then processes them with a *bidirectional long short-term memory* (BLSTM) network to train the detection model; ***BenchSG*** [12] is a vulnerability detector based on statement. It regards a program's statements as sentences and inputs these "sentences" into a *bidirectional gated recurrent unit* (BiGRU) directly to train the detection model; ***Devign*** [5] is a vulnerability detector based on graph. It first processes a program into various forms, including *abstract syntactic tree* (AST), *control flow graph* (CFG), *data flow graph* (DFG), and *natural code sequence* (NSC). Then it inputs all types of data into a *graph neural network* (GNN) to train the detection model.

3) Implementations

We implement *COCL* in Python using Pytorch. We run experiments on a machine with NVIDIA GeForce GTX 2080Ti GPU and 32 cores of CPU. Limited by memory, the batch size we experimented with is 32. For our dataset, we first divide the whole dataset into ten subsets and combine eight of them into the training set, one for validation and one for testing.

4) Metrics

We use four widely used metrics to measure the performance. $Accuracy = \frac{TP+TN}{TP+FP+TN+FN}$, $Recall = \frac{TP}{TP+FN}$, $Precision = \frac{TP}{TP+FP}$, and $F1 = 2\ \frac{Precision\ Recall}{Precision+Recall}$. Note that *True Positive* (TP) denotes the number of correctly predicted vulnerable samples, *True Negative* (TN) denotes the number of correctly predicted normal samples, *False Positive* (FP) denotes the number of incorrectly classified vulnerable samples, and *False Negative* (FN) denotes the number of incorrectly classified normal samples.

[3]https://www.qemu.org.
[4]https://www.ffmpeg.org.

TABLE V: Experimental results of *COCL* with static-based and dynamic-based *Difficulty Calculators*. DC denotes *Difficulty Calculator*, SupCon denotes *Supervised Contrastive Learning*.

AutoVulTC	Baseline	SupCon	COCL	
DC	None	None	Static	Dynamic
Accuracy	0.458	0.537	**0.582**	0.573
Recall	0.510	0.532	**0.574**	0.545
Precision	0.416	0.508	**0.540**	0.533
F1	0.458	0.520	**0.557**	0.539
VulDeePecker	Baseline	SupCon	Static	Dynamic
Accuracy	0.532	0.597	0.600	**0.611**
Recall	0.577	0.643	**0.671**	0.669
Precision	0.472	0.549	0.550	**0.561**
F1	0.519	0.592	0.605	**0.611**
BenchSG	Baseline	SupCon	Static	Dynamic
Accuracy	0.537	0.59	0.606	**0.609**
Recall	0.592	0.651	**0.666**	0.665
Precision	0.488	0.542	0.557	**0.560**
F1	0.535	0.591	0.606	**0.608**
Devign	Baseline	SupCon	Static	Dynamic
Accuracy	0.561	0.572	0.576	**0.587**
Recall	0.572	0.619	**0.723**	0.667
Precision	0.534	0.536	**0.541**	0.539
F1	0.553	0.575	**0.619**	0.596

B. Contrastive Learning

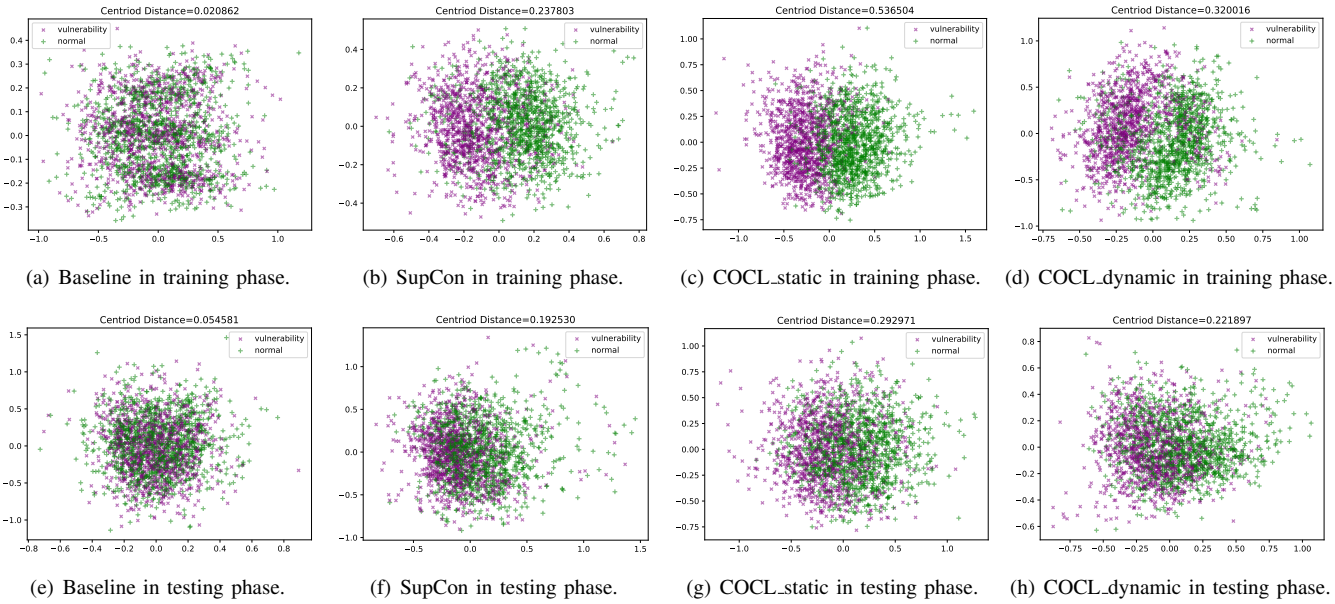
In this part, we conduct experiments to check the ability of contrastive learning on DL-based vulnerability detection.

TABLE VI: The improvements (%) introduced by *co*ntrastive *le*arning (Average$_{cole}$) on DL-based vulnerability detection.

Models	Accuracy	Recall	Precision	F1
AutoVulTC	7.9	2.2	9.2	6.2
VulDeePecker	6.5	6.6	7.7	7.3
BenchSG	5.3	5.9	5.4	5.6
Devign	1.1	4.7	0.2	2.2
Average$_{cole}$	5.2	4.9	5.6	5.3

Table V shows the relevant results. Specifically, we present the detection effectiveness of DL-based models (*i.e., AutoVulTC*, *VulDeePecker*, *BenchSG*, and *Devign*) with contrastive loss and without contrastive loss (only cross-entropy loss). If a model does not use contrastive learning to train itself, we refer to it as *Baseline* in our experiments. Similarly, if we perform contrastive learning to train a model, we refer to it as *SupCon* in our experiments. Through the results in Table V, we can see that the use of contrastive learning can indeed improve the detection effectiveness of all four DL-based vulnerability detectors. For example, the F1 of *AutoVulTC* is only 45.8% while can increase to 52% if we apply contrastive learning to complete the training phase. For *VulDeePecker*, the recall of *Baseline* is 57.7%. However, after using contrastive learning to train *VulDeePecker*, the recall can increase to 64.3%. In other words, using contrastive learning can make *VulDeePecker* detect 6.6% more vulnerabilities.

To show more clearly the improvement introduced by contrastive learning, we add another table to present the results. As shown in Table VI, all metrics are boosted for all models (*i.e., AutoVulTC*, *VulDeePecker*, *BenchSG*, and *Devign*), which means that contrastive learning can indeed enhance their vulnerability detection capabilities. On average,



(a) Baseline in training phase. (b) SupCon in training phase. (c) COCL_static in training phase. (d) COCL_dynamic in training phase.

(e) Baseline in testing phase. (f) SupCon in testing phase. (g) COCL_static in testing phase. (h) COCL_dynamic in testing phase.

Fig. 2: The visualization of source code features generated by *AutoVulTC* on a real-world vunlerability dataset.

the improvements of Accuracy, Recall, Precision, and F1 are 5.2%, 4.9%, 5.6%, and 5.3%, respectively. Such results demonstrate the effectiveness of contrastive learning on DL-based vulnerability detection.

C. Curriculum Learning

In practice, traditional contrastive learning does not consider the difficulty of each sample during the training phase. It treats all the samples equally and samples them by using a random strategy. To make our training phase more intelligent, we apply curriculum learning to assist contrastive learning. In particular, we take into account the model's different abilities to understand different samples. To be more precise, we change the way that contrastive learning is used to select negative samples, which are usually random. The type of contrastive learning we choose is one positive pair to multiple negative samples, so we sort the negative samples according to their learning difficulty, and let the model start learning from simple samples, then gradually transition to those more difficult samples.

TABLE VII: The improvements (%) introduced by *cu*rriculum *le*arning (Average$_{cule}$) on DL-based vulnerability detection. A: Accuracy, R: Recall, P: Precision.

Models	COCL_static				COCL_dynamic			
	A	R	P	F1	A	R	P	F1
AutoVulTC	4.5	4.2	3.2	3.7	3.6	1.3	2.5	1.9
VulDeePecker	0.3	2.8	0.1	1.3	1.4	2.6	1.2	1.9
BenchSG	1.6	1.4	1.5	1.5	1.9	1.4	1.8	1.7
Devign	0.4	10.4	0.5	4.4	1.5	4.8	0.3	2.1
Average$_{cule}$	1.7	4.7	1.3	2.7	2.1	2.5	1.5	1.9
Average$_{cole+cule}$	6.9	9.6	6.9	8.1	7.3	7.4	7.1	7.2

In this part, we present the experimental results after joining curriculum learning and analyze the improvements introduced by curriculum learning. We uniformly select the pace step size of two for simplicity and train four DL-based vulnerability detectors (*i.e., AutoVulTC*, *VulDeePecker*, *BenchSG*, and *Devign*) with two types of difficulty calculators (*i.e.,* static-based calculator and dynamic-based calculator). Table V illustrates the detection results of *AutoVulTC*, *VulDeePecker*, *BenchSG*, and *Devign* after combining curriculum learning with contrastive learning during training.

For static-based difficulty calculator, we consider code complexity to be equivalent to sample learning difficulty. Samples with high code complexity will be regarded as difficult samples. From the results in Table VII, we see on all metrics for all models, curriculum learning can bring an improvement to contrastive learning. For F1, it improves 3.7% on *AutoVulTC*, 1.3% on *VulDeePecker*, 1.5% on *BenchSG*, and 4.4% on *Devign*, respectively. Such results indicate that curriculum learning can boost the detection effectiveness of contrastive learning. More detailed improvements are described in Table VII. For dynamic-based difficulty calculator, we obtain the difficulty by computing the similarity between a positive sample and other negative samples in a batch. Samples with high similarity will be treated as difficult samples. Through Table VII, it can be seen that the detection effectiveness can also be enhanced when we adopt dynamic-based difficulty calculator. For example, for Recall, we can maintain an average of 2.5% improvement on four DL-based vulnerability detectors (*i.e., AutoVulTC*, *VulDeePecker*, *BenchSG*. In one word, no matter which difficulty calculator is adopted, the detection ability of four models can be boosted.

We also show the improvements introduced by contrastive curriculum learning (*i.e.,* Average$_{cole+cule}$), that is, the improvements between *Baseline* model and model trained with contrastive curriculum learning. For example, for *Devign*, the Recall of *Baseline* model is 57.2% while it can be increased to 72.3% after we perform contrastive curriculum learning to finish the training process. The improvement is 15.1%, which means that using *COCL* allows *Devign* to detect 15.1% more vulnerabilities. In the case of *VulDeePecker*, the *Baseline* model only achieves an F1 of 51.9%. However, after applying *COCL*, the F1 can be improved to 61.1%, resulting in an increase of 9.2%. Moreover, when *COCL* uses static difficulty

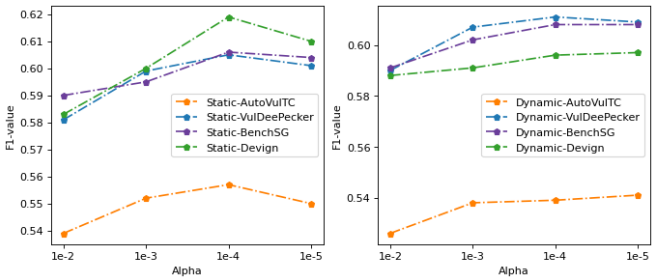


Fig. 3: Sensitivity analysis of hyperparamter alpha.

calculator to rank the samples, the average Recall and F1 can achieve an improvement of 9.6% and 8.1%, respectively. These results suggest that *COCL* is an effective framework that can enhance the ability of DL-based vulnerability detection.

To further understand why *COCL* can enhance DL-based vulnerability detection, we use PCA[5] to visualize the distribution of feature vectors generated by *AutoVulTC*, *VulDeePecker*, *BenchSG*, and *Devign*, respectively. Due to space limitations, we only show the results of *AutoVulTC* in Figure 2 including the visualization of feature vectors in the training and testing phases. Through the figure, we see that *Baseline* model exhibits a significant degree of overlap in the feature space between vulnerabilities and normal codes. After we introduce contrastive learning into *AutoVulTC*, the centroid distance is increased. It is because contrastive learning can cluster the homogeneous feature vectors and separate the non-homogeneous feature vectors as much as possible in the feature space. Furthermore, when curriculum learning is combined with contrastive learning, the centroid distance is also increasing. Such result indicates that nesting curriculum learning into contrastive learning can improve the effect of clustering and separation, thus further improve the detection performance.

D. Sensitivity Analysis

In this part, we investigate the impact of different loss weight in equation (5). More specifically, we adjust within the range of 1e-2 to 1e-5 and record the corresponding F1 values of *COCL*. The experimental results are shown in Figure 3. When the value of varies within the aforementioned range, the static sorting *COCL* produces a fluctuation of 0.016 to 0.036 in F1 values, while the dynamic sorting *COCL* produces a fluctuation of 0.009 to 0.021. The range of fluctuations under both methods is within 6%, indicating that the tolerance of equation (5) for fluctuations in is relatively high, and good detection performance could be obtained within a certain range of values. In our paper, we finally choose 1e-4 since both static and dynamic sorting *COCL* can achieve higher F1 values when is 1e-4.

V. CONCLUSION

In this paper, we introduce *Contrastive Curriculum Learning* into deep learning-based vulnerability detection. Specifically, we implement a framework (*i.e., COCL*) and select four state-of-the-art vulnerability detection methods (*i.e., AutoVulTC* [7], *VulDeePecker* [8], *BenchSG* [12], and *Devign* [5]) to evaluate *COCL* on a real-world vulnerability dataset. Through the results, we find that using *COCL* enables them to detect 9.6% more vulnerabilities on average. Especially for *Devign*, the Recall can be increased from 57.2% to 72.3%, which means that the use of *COCL* makes *Devign* detect 15.1% more vulnerabilities. Such results imply that *COCL* is an effective framework that can indeed enhance the performance of existing deep learning-based vulnerability detectors.

[5]https://en.wikipedia.org/wiki/Principal_component_analysis.

ACKNOWLEDGEMENTS

This research is supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-RP-2020-019), and NRF Investigatorship NRF-NRFI06-2020-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 595–614.

[2] "Checkmarx," https://www.checkmarx.com/, 2022.

[3] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE transactions on software engineering*, vol. 37, no. 6, pp. 772–787, 2010.

[4] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, 2010, pp. 1–8.

[5] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[6] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," *arXiv preprint arXiv:1708.02368*, 2017.

[7] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[8] Z. Li, D. Zou, S. Xu, X. Ou, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018, pp. 1–15.

[9] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2020.

[10] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[11] K. A. Krueger and P. Dayan, "Flexible shaping: How learning in small steps helps," *Cognition*, vol. 110, no. 3, pp. 380–394, 2009.

[12] G. Lin, W. Xiao, J. Zhang, and Y. Xiang, "Deep learning-based vulnerable function detection: A benchmark," in *International Conference on Information and Communications Security*. Springer, 2019, pp. 219–232.

[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 96–105.

[14] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, 2016, pp. 1157–1168.