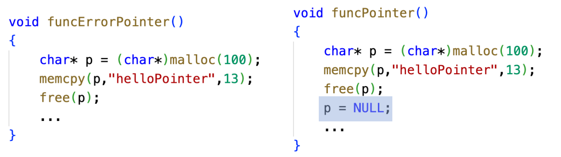
Code Vulnerability Detection via Nearest Neighbor Mechanism

Qianjin Du[1]* **Xiaohui Kuang**[2] **Gang Zhao**[2]*
[1]Department of Computer Science and Technology, Tsinghua University
[2]National Key Laboratory of Science and Technology on Information System Security
dqj20@mails.tsinghua.edu.cn
xhkuang@bupt.edu.cn; bisezhaog@163.com

Abstract

Code vulnerability detection is a fundamental and challenging task in the software security field. Existing research works aim to learn semantic information from the source code by utilizing NLP technologies. However, in vulnerability detection tasks, some vulnerable samples are very similar to non-vulnerable samples, which are difficult to identify. To address this issue and improve detection performance, we introduce the k-nearest neighbor mechanism which retrieves multiple neighbor samples and utilizes label information of retrieved neighbor samples to provide help for model predictions. Besides, we use supervised contrastive learning to make the model learn the discriminative representation and ensure that label information of retrieved neighbor samples is as consistent as possible with the label information of testing samples. Extensive experiments show that our method can achieve obvious performance improvements compared to baseline models.



```
void funcErrorPointer()
{
    char* p = (char*)malloc(100);
    memcpy(p,"helloPointer",13);
    free(p);
    ...
}
```

(a) Vulnerable instance

```
void funcPointer()
{
    char* p = (char*)malloc(100);
    memcpy(p,"helloPointer",13);
    free(p);
    p = NULL;
    ...
}
```

(b) Non-vulnerable instance

Figure 1: Illustrations of the vulnerable and non-vulnerable code snippets.

1 Introduction

Code vulnerabilities generally denote security bugs or weaknesses in software. The presence of vulnerabilities in software is an inevitable problem because of design flaws of the program language and faults caused by programmers. Traditional detection methods, such as static methods (Kim et al., 2017; Roy et al., 2009) or dynamic methods (Serebryany, 2015; Cadar et al., 2008; Sen et al., 2005), generally require a lot of human labor to summarize vulnerability rules or massive computational resources to trigger potential vulnerabilities by executing programs. To improve detection efficiency, there are multiple efforts (Li et al., 2018; Zhou et al., 2019; Feng et al., 2020) have attempted to introduce deep learning and NLP techniques for automated vulnerability detection. Deep learning-based detectors can extract semantic features from the source code and automatically learn potential vulnerability patterns. Li et al. (2018) proposed a vulnerability detection framework named VulDeePecker, which used a binary classifier based on LSTM to detect whether a piece of code is vulnerable or not. In order to learn comprehensive program semantics to characterize vulnerabilities of high diversity and complexity in the source code, Zhou et al. (2019) utilized the graph neural network to extract code semantic representations. Inspired by the big success of pre-training models on numerous natural language processing (NLP) tasks (Liu et al., 2019; Raffel et al., 2020), code pre-training models for programming languages, such as CodeBert (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2020), were proposed and applied to a variety of code-related tasks such as code search, code completion, and code vulnerability detection, etc. Although the above methods have achieved remarkable improvements, they still easily generate a large number of false predictions because of the relatively large classification uncertainty.

In vulnerability detection tasks, vulnerabilities are sometimes subtle flaws. Fig.1 illustrates the comparison between the vulnerable and non-vulnerable code snippets. We can find that the difference between vulnerable and non-vulnerable samples is very small. This means that some vulnerable and non-vulnerable samples are close to

*Corresponding authors.

Findings of the Association for Computational Linguistics: EMNLP 2022, pages 6173–6178
December 7-11, 2022 ©2022 Association for Computational Linguistics

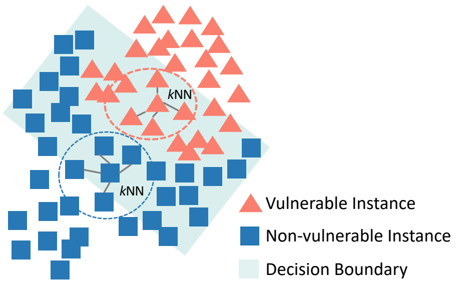


Figure 2: The illustration of vulnerable and non-vulnerable samples in the representational space.

the classifier boundary, which has the large classification uncertainty and easily results in false positive/negative predictions (as depicted in Fig.2). To alleviate this problem and improve classification performance, we argue that fully exploiting the label information of samples with the same class can assist the model to classify samples more accurately. That is, when predicting the label of a sample, the model can obtain a good guideline from its neighbor samples. Motivated by this obervation, we present a novel code vulnerability detection method by introducing the k-nearest neighbor mechanism, which could fully utilize the label information of neighbor samples to perform more accurate vulnerability classification. Specifically, it firstly retrieves multiple neighbor samples according to the representation of the sample and then integrate their labels into the final prediction of the model. Furthermore, in order to retrieve more similar samples and make the label information of retrieved neighbor samples more effective, we introduce supervised contrastive learning (SCL) to learn discriminative code representations. Supervised contrastive learning (SCL) could pull samples with the same class closer to each other and push away samples with the different classes. In short, our contributions are summarized as follows:

- We propose a novel vulnerability detection method by introducing the k-nearest neighbor mechanism, which fully utilizes the label information of neighbor samples to assist the model to conduct prediction.

- We introduce supervised contrastive learning to learn more discriminative code representations, so as to retrieve more similar neighbor samples to improve detection performance.

- We conduct extensive experiments to demonstrate the effectiveness of the proposed method. Experimental results show that our method achieves universal and significant performance improvements compared to baseline methods.

2 Related Work

2.1 Code Vulnerability

To alleviate the reliance on the intense labor of security experts and improve detection efficiency, recent research works (Lin et al., 2017; Grieco et al., 2016; Li et al., 2018; Zou et al., 2019) have attempted to introduce deep learning techniques for vulnerability detection. There are multiple research works in pinpointing vulnerabilities at different levels of granularity (e.g., program (Grieco et al., 2016), package (Neuhaus and Zimmermann, 2009), file (Shin et al., 2010), function (Yamaguchi et al., 2014; Zhou et al., 2019)) by combining deep learning with static bug detection.

2.2 K-Nearest Neighbor

Some research works have applied kNN to NLP in recent years. Khandelwal et al. (2019) performed an accurate natural language model by introducing kNN, which directly queries training examples at test time. Then Khandelwal et al. (2020) proposed a machine translation framework based on kNN, which predicts tokens with the nearest neighbor classifier over a large datastore of cached examples. Muktafin et al. (2021) used kNN and TF-IDF algorithm to perform sentiments analysis. Riza et al. (2019) utilized kNN to perform a question generator system. Wang et al. (2022) proposed a k-nearest neighbor mechanism for multi-label text classification. Besides, they designed a multi-label contrastive learning objective that makes the model aware of the k-nearest neighbor mechanism.

3 Methodology

In this section, we will elaborate the details of the proposed method. The overview of the proposed method is illustrated in Fig.3.

3.1 Problem Formulation

Let a sample be defined as (x_i, y_i), where x_i represents a code sample and $y_i \in \{0, 1\}$ (where 1: vulnerable code and 0: non-vulnerable code). The goal of vulnerability detection is to learn a mapping from x_i to y_i, $f : x_i \mapsto y_i$ to predict whether a code sample is vulnerable or not. The prediction

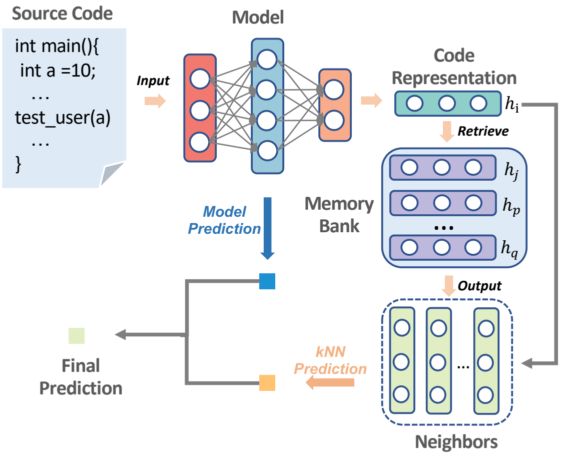


Figure 3: The framework of the proposed method.

function f can be learned by minimizing the loss function below:

$$\min \sum_{i=1}^{n} \mathcal{L}(f(x_i), y_i) \tag{1}$$

where $\mathcal{L}(\cdot)$ is the cross-entropy loss function.

3.2 K-Nearest Neighbor for Vulnerability Detection

As mentioned before, to further improve detection performance, we propose a novel detection method based on the k-nearest neighbor mechanism, which utilizes the label information of neighbor samples to assist the model to make predictions. The proposed method consists of two components: the construction of the memory bank for storing sample representations and the prediction based on the k-nearest neighbor mechanism.

The construction of the memory bank Given an code sample (x_i, y_i), h_i denotes the latent feature representation outputted by the vulnerability detection model. The memory bank is the data structure that is used to store the latent representation and the corresponding label of each training sample. Formally, the memory bank is built by: $MB = \{(h_1, y_1), ..., (h_n, y_n)\}$.

Prediction based on the k-nearest neighbor mechanism Given a code sample x_i, the latent representation of x_i can be obtained from the model: $h_i = f(x_i)$. Then the representation h_i will be used as the key to query the memory bank to obtain the k-nearest neighbor samples by calculating the distance of each sample pairs (h_i, h_j): $h_{kNN} = \{(h_j, y_j)\}_{j=1}^{k}$, where h_j denotes the representation of a sample stored in the memory bank MB. The prediction based on the k-nearest neighbor mechanism is formulated by:

$$y_{kNN} = \sum_{j=1}^{k} \alpha_j y_j \tag{2}$$

where α_j denotes the weight of the j_{th} neighbor sample. In our work, α_j is calculated by:

$$\alpha_j = \frac{exp(D(h_i, h_j))}{\sum_o exp(D(h_i, h_o))} \tag{3}$$

where $D(\cdot)$ denotes the euclidean distance of sample pairs (h_i, h_j).

The final prediction is formulated as follows:

$$\hat{y} = \gamma y_{Model} + \delta y_{kNN} \tag{4}$$

where y_{Model} denotes the model prediction and γ and δ represent the weights of model prediction and kNN prediction respectively.

3.3 Supervised Contrastive Learning

As mentioned before, some vulnerable code text is similar to the non-vulnerable code text, which is easy to cause labels of the retrieved samples to be different from the label of the testing sample. That is, for a non-vulnerable code sample, quite a few of its retrieved neighbor samples may be vulnerable, which cannot provide effective help and even misleads the final prediction. To alleviate this problem, supervised contrastive learning (SCL) (Khosla et al., 2020) is introduced to learn more discriminative code representations.

Supervised contrastive learning aims to pull samples belonging to the same class together in the representational space and push away samples from different classes. In this way, it makes vulnerable samples closer to each other in the representational space so that the label information of retrieved samples can provide more effective help for the model prediction (Khosla et al., 2020). Within a mini-batch $B = \{x_1, ...x_M\}$ consisting of M samples, let $P(i)$ be the index of all the other samples whose classes are the same as the sample i and $A(i)$ be the index of all samples except the sample i. The supervised contrastive learning can be calculated as:

$$\mathcal{L}_{sup} = \sum_{i\ \in B} \frac{-1}{|P(i)|} \sum_{p \in P(i)} \log \frac{exp(\boldsymbol{z_i} \cdot \boldsymbol{z_p}/\tau)}{\sum\limits_{a \in A(i)} exp(\boldsymbol{z_i} \cdot \boldsymbol{z_a}/\tau)} \tag{5}$$

where τ is the contrastive learning temperature and $\boldsymbol{z_i} = f(x_i)$ denotes the latent representation.

The overall loss function can be formulated as: $\mathcal{L} = \mathcal{L}_{CE} + \lambda\mathcal{L}_{sup}$, where $\mathcal{L}_{CE}$ denotes the cross entropy loss function.

4 Experiments

4.1 Experimental Setup

Our experiments are implemented based on the open-source deep learning framework Pytorch (Paszke et al., 2019). The total training epochs for all models is 50 with a batch size of 64. We optimize the model weights using SGD with an initial learning rate of 0.01. As for the hyper-parameters of our proposed method, γ is set to 0.7 and δ is set to 0.3. λ is set to 0.2. We conduct experiments on the published code vulnerability dataset QEMU+FFmpeg (Zhou et al., 2019), which collects real-world vulnerabilities from GitHub repositories. The labelling is done based on commit messages and domain experts. Statistics on the QEMU+FFmpeg dataset are summarized in Table 1. Following the previous work (Zhou et al., 2019), we adopt the F1-score as our evaluation metric.

Total Samples	Vulnerabilities	Non-Vulnerabilities
21,020	9,116	11,904

Table 1: Statistics on the QEMU+FFmpeg dataset.

4.2 Baseline

In our experiments, adopted baselines are listed as follows:

Devign (Zhou et al., 2019) uses a gated graph convolutional networks to extract the semantic information of the source code and finally utilizes a 1-D CNN-based pooling ("Conv") to make a prediction.

CodeBERT (Feng et al., 2020) is a powerful pretrained model for programming language, which is trained in six programming languages.

GraphCodeBERT (Guo et al., 2020) is a new pretrained programming language model, extending CodeBERT to consider the inherent structure of code data flow into the training objective.

4.3 Results

The experimental results are reported in Tab.2. We can find that our proposed method achieves universal and obvious performance improvements compared to baselines. The CodeBERT using our proposed prediction method achieves the best performance on FFmpeg, obtaining an F1-score gain of 2.69% compared to the original CodeBERT model. The GraphCodeBERT using our proposed method achieves an F1-score of 72.47% on QEMU, surpassing the original GraphCodeBERT model by 2.46%.

Models	FFmpeg	QEMU
Devign	62.18	65.74
Devign + Ours	64.60	67.81
CodeBERT	67.46	69.20
CodeBERT + Ours	**70.15**	71.52
GraphCodeBERT	67.05	70.01
GraphCodeBERT + Ours	69.43	**72.47**

Table 2: The final performance on QEMU+FFmpeg dataset.

Models	FFmpeg	QEMU
Devign	62.18	65.74
Devign + SCL	62.78	66.52
Devign + kNN	64.14	67.30
Devign + kNN + SCL	64.60	67.81
CodeBERT	67.46	69.20
CodeBERT + SCL	68.38	70.08
CodeBERT + kNN	69.32	70.96
CodeBERT + kNN + SCL	70.15	71.52
GraphCodeBERT	67.05	70.01
GraphCodeBERT + SCL	68.01	70.96
GraphCodeBERT + kNN	68.66	71.92
GraphCodeBERT + kNN + SCL	69.43	72.47

Table 3: F1-score of the ablation studies. kNN indicates the proposed k-nearest neighbor mechanism and SCL indicates contrastive learning.

4.4 Ablation Study

In our proposed method, we introduce the k-nearest neighbor mechanism to fully utilize label information of neighbor samples and use supervised contrastive learning to learn more discriminative code representations. In this section, we conduct experiments to verify the effectiveness of the above components respectively. The results of ablation studies are reported in Tab.3. We can find that both the k-nearest neighbor mechanism and supervised contrastive learning achieve performance improvements individually. The k-nearest neighbor mechanism can bring an average gain of 1.68%. In addition, compared to the individual k-nearest neighbor mechanism, the k-nearest neighbor mechanism using supervised contrastive learning can achieve better classification performance, demonstrating that introducing supervised contrastive learning can learn more discriminative code representations to better assist the kNN prediction.

4.5 Conclusion

In this paper, we propose a novel code vulnerability detection method based on the k-nearest neighbor mechanism, which fully utilizes the label information of neighbor samples to assist the model to predict code vulnerabilities. Moreover, to ensure that the label information of retrieved neighbor samples can provide more effective help for model predictions, we introduce supervised contrastive learning to make vulnerable samples closer to each other in the representational space. Finally, we conduct extensive experiments to verify the effectiveness of our proposed method.

Limitations

Here we summarize the limitations for further discussion and investigation of the community. Our proposed k-nearest neighbor mechanism requires manually setting the optimal k^* to achieve the best classification performance, which is inefficient. A better solution is to design an adaptive k-nearest neighbor mechanism which could adaptively seek optimal neighbor samples according to the distribution information of training samples.

References

Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224.

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*.

Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 85–96.

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*.

Urvashi Khandelwal, Angela Fan, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Nearest neighbor machine translation. *arXiv preprint arXiv:2010.00710*.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2019. Generalization through memorization: Nearest neighbor language models. *arXiv preprint arXiv:1911.00172*.

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.

Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614. IEEE.

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*.

Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. 2017. Poster: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2539–2541.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Elik Hari Muktafin, Pramono Pramono, and Kusrini Kusrini. 2021. Sentiments analysis of customer satisfaction in public services using k-nearest neighbors algorithm and natural language processing approach. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, 19(1):146–154.

Stephan Neuhaus and Thomas Zimmermann. 2009. The beauty and the beast: Vulnerabilities in red hat's packages. In *USENIX annual technical conference*, pages 527–538.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67.

Lala Septem Riza, Anita Dyah Pertiwi, Eka Fitrajaya Rahman, Munir Munir, and Cep Ubad Abdullah. 2019. Question generator system of sentence completion in toefl using nlp and k-nearest neighbor. *Indonesian Journal of Science and Technology*, 4(2):294–311.

Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272.

Kostya Serebryany. 2015. libfuzzer–a library for coverage-guided fuzz testing. *LLVM project*.

Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering*, 37(6):772–787.

Ran Wang, Xinyu Dai, et al. 2022. Contrastive learning-enhanced nearest neighbor mechanism for multi-label text classification. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 672–679.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE.

Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks.

Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μvuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*.