One-for-All Does Not Work! Enhancing Vulnerability Detection by Mixture-of-Experts (MoE)

XU YANG, University of Manitoba, Canada
SHAOWEI WANG, University of Manitoba, Canada
JIAYUAN ZHOU, Huawei, Canada
WENHAN ZHU, Huawei, Canada

Deep Learning-based Vulnerability Detection (DLVD) techniques have garnered significant interest due to their ability to automatically learn vulnerability patterns from previously compromised code. Despite the notable accuracy demonstrated by pioneering tools, the broader application of DLVD methods in real-world scenarios is hindered by significant challenges. A primary issue is the "one-for-all" design, where a single model is trained to handle all types of vulnerabilities. This approach fails to capture the patterns of different vulnerability types, resulting in suboptimal performance, particularly for less common vulnerabilities that are often underrepresented in training datasets. To address these challenges, we propose MoEVD, which adopts the Mixture-of-Experts (MoE) framework for vulnerability detection. MoEVD decomposes vulnerability detection into two tasks: CWE type classification and CWE-specific vulnerability detection. By splitting the task, in vulnerability detection, MoEVD allows specific experts to handle distinct types of vulnerabilities instead of handling all vulnerabilities within one model. Our results show that MoEVD achieves an F1-score of 0.44, significantly outperforming all studied state-of-the-art (SOTA) baselines by at least 12.8%. MoEVD excels across almost all CWE types, improving recall over the best SOTA baseline by 9% to 77.8%. Notably, MoEVD does not sacrifice performance on long-tailed CWE types; instead, its MoE design enhances performance (F1-score) on these by at least 7.3%, addressing long-tailed issues effectively.

CCS Concepts: • **Software and its engineering**;

Additional Key Words and Phrases: Vulnerability Detection, Deep Learning, Mixture-of-Experts (MoE)

ACM Reference Format:
Xu Yang, Shaowei Wang, Jiayuan Zhou, and Wenhan Zhu. 2025. One-for-All Does Not Work! Enhancing Vulnerability Detection by Mixture-of-Experts (MoE). *Proc. ACM Softw. Eng.* 2, FSE, Article FSE021 (July 2025), 19 pages. https://doi.org/10.1145/3715736

1 Introduction

Vulnerability detection is a critical task in software engineering, aiming to identify security weaknesses that could be exploited by malicious entities. Deep learning-based vulnerability detection (DLVD) techniques have shown impressive results in academic settings [5, 20, 24, 26, 35, 36, 42].

However, a recent study [41] reveals a significant challenge that prevents deep learning models from being adopted in industry settings. The challenge is the **one-for-all design limitation**. Existing DLVD techniques follow the "one-for-all" design, where a single model is trained to predict vulnerabilities across all types of code [5, 16, 20, 24–26, 35, 50]. While this design simplifies the

Authors' Contact Information: Xu Yang, University of Manitoba, Winnipeg, Canada, yangx4@myumanitoba.ca; Shaowei Wang, University of Manitoba, Winnipeg, Canada, shaowei.wang@umanitoba.ca; Jiayuan Zhou, Huawei, Toronto, Canada, jiayuanzhou1@acm.org; Wenhan Zhu, Huawei, Waterloo, Canada, wenhanzhu1@acm.org.

ACM 2994-970X/2025/7-ARTFSE021
https://doi.org/10.1145/3715736

development and deployment process, it does not align with the diverse and specific needs of real-world applications. In practice, large organizations often encounter various types of vulnerabilities, each with unique characteristics and implications [41]. A one-for-all model may fail to adequately address the nuances of different vulnerability types and lack the capability of detecting certain types of vulnerabilities.

One significant issue stemming from the one-for-all design is the uneven and long-tailed distribution of different Common Weakness Enumeration (CWE) [7] types in real-world codebases. Certain types of vulnerabilities, such as those listed in the CWE Top 25 [29], are more prevalent and thus receive more attention in both research and industry. However, numerous other vulnerability types are less common and often underrepresented in training datasets. This imbalance leads to a situation where DLVD techniques are highly effective at detecting common vulnerabilities but struggle with rare ones. In practical terms, this means that while a model might excel in identifying frequent issues, its performance on less common, yet potentially critical vulnerabilities is subpar. This uneven performance can undermine the overall effectiveness of DLVD techniques, leaving some vulnerabilities undetected and increasing the risk of security breaches. For instance, Figure 1 presents the performance of LineVul [16], a SOTA approach that is a single model trained to detect all types of vulnerabilities, on different CWE types compared with experts using the same model trained on each specific CWE type. As we can see, LineVul performs poorly on less frequent CWE types with F1-score ranging from 0.016 to 0.268. A similar trend can be observed on other one-for-all models as well.

The Mixture-of-Experts (MoE) framework [21, 47] is a promising solution proposed by previous research to resolve the limitations of the one-for-all design. The MoE framework addresses this limitation by first splitting the input space into sub-spaces, and then leveraging multiple specialized models (i.e., experts), trained to handle each sub-space. Instead of relying on a single model to handle the entire input space, MoE includes a router that dynamically assigns the most appropriate expert for handling a given input.

To address the challenges mentioned above, we propose MoEVD, which leverages the Mixture-of-Experts (MoE) framework for vulnerability detection. Two key challenges need to be resolved to effectively leverage MoE in this context. First, how to split the input space effectively, so that each expert could focus on one type or a set of related vulnerabilities. Second, how to design a router that can effectively assign the input to the appropriate experts for handling the vulnerability detection task. To address the first challenge, we split the input code according to Common Weakness Enumeration (CWE) types [7], a widely used classification system for software vulnerabilities. This splitting mechanism allows each expert to focus on learning patterns and detecting vulnerabilities specific to a particular type of CWE or a subset of CWE types. More specifically, to construct the expert for each CWE type, we fine-tune a pre-trained model (e.g., CodeBERT [13]) to predict if a piece of code is vulnerable. To address the second challenge, we train a router to dynamically select the appropriate experts for a given input code by formulating this task as a CWE type multi-class classification task. Essentially, MoEVD decomposes the task of binary vulnerability detection into a CWE type classification task and a CWE-specific vulnerability detection task so that specific experts can be trained to handle a specific set of vulnerabilities instead of handling all types of vulnerabilities within one model.

We performed an extensive evaluation on a widely used benchmark dataset BigVul [11]. Our results show that MoEVD achieves an F1-score of 0.44, significantly outperforming all studied state-of-the-art (SOTA) baselines by at least 12.8%. MoEVD outperforms the best SOTA baseline across almost all CWE types in terms of F1-score and recall. For instance, MoEVD improves the recall of the best SOTA baseline by a range from 9% to 77.8%. More importantly, MoEVD does not sacrifice performance in long-tailed CWE types. On the contrary, benefiting from the MoE design

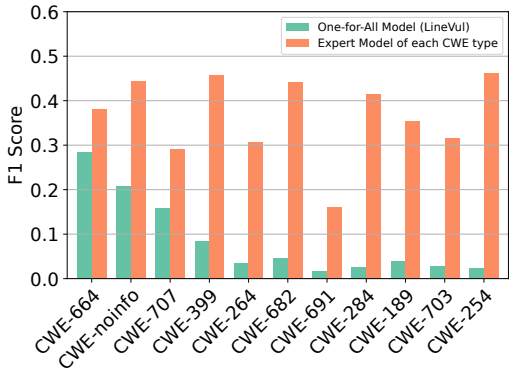

Fig. 1. The performance of one-for-all Model (LineVul) vs. expert model trained on each specific CWE type across CWE types.

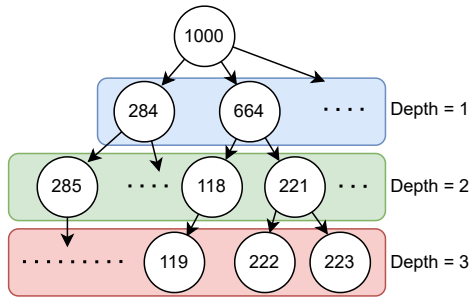

Fig. 2. Hierarchy structure of CWE types. The arrows present the parent-child relationship between two CWE types. For instance, CWE-664 is the parent of CWE-221.

that enables the model to learn various vulnerability patterns individually, MoEVD improves the performance on vulnerabilities of long-tailed CWE types. For instance, MoEVD mitigates long-tailed issues by improving the F1-score of SOTA baselines on the long-tailed CWEs by at least 7.3%.

Our contributions are summarized below:

- We are the first to adapt the Mixture-of-Experts (MoE) framework to enhance vulnerability detection and our extensive evaluation demonstrates its effectiveness, showing improvements across all CWE types.
- MoEVD enhances the detection of rare CWE types of vulnerabilities that are overlooked by existing one-for-all models and makes it more suitable for real-world applications.
- To facilitate future research, we have made all the datasets, results, and code used in this study openly available in our replication package [1].

2 Background and Motivation

In this section, we introduce the background of our study, focusing on the pre-trained models for code representation Learning, CWE hierarchy, and Mixture-of-Experts (MoE) framework.

2.1 Pre-trained Model for Code Representation Learning

Deep learning-based vulnerability detection techniques rely on code representation learning to capture the syntactic and semantic properties of the code [24, 46, 50]. These techniques transform code snippets into vectors, which can be effectively processed by classification models. Recently, several pre-trained models tailored for programming languages have emerged and demonstrated promising performance in code-related tasks, such as CodeBERT [13], GraphCodeBERT [18], and UniXcoder [17]. Those models are based on transformer architecture, which has proven effective in code-related tasks. For instance, CodeBERT is built upon the foundations of BERT [8] through further pre-training on natural language (NL) - programming language (PL) pairs. Its design and training methods enable CodeBERT to understand not only the relationship between natural language and code but also the semantics of the source code. These pre-trained models have demonstrated SOTA performance in vulnerability detection [16, 30]. In this study, we use pre-trained models as the foundation for our experts and router by fine-tuning them to adapt to

downstream tasks. Specifically, we add a binary classification head to a pre-trained model for vulnerability detection and a multi-class classification head for CWE type multi-class classification.

2.2 Common Weakness Enumeration

Common Weakness Enumeration(CWE) is a category system for weaknesses and vulnerabilities. CWE follows a hierarchical structure, as illustrated in Figure 2, where each CWE type has ancestors and related successors [7]. For instance, CWE-664 (Improper Control of a Resource Through its Lifetime) has two children, CWE-118 (Incorrect Access of Indexable Resource ('Range Error')) and CWE-221 (Information Loss or Omission). Typically, the children CWE types under a parent CWE type are related and share similar vulnerability patterns. A parent CWE is a more general category encompassing its child CWE types. For vulnerabilities where the details are unknown or unspecified, the NVD assigns a placeholder value, "CWE-noinfo," when it is unable to categorize a CVE under a specific CWE entry.

2.3 Mixture-of-Experts (MoE)

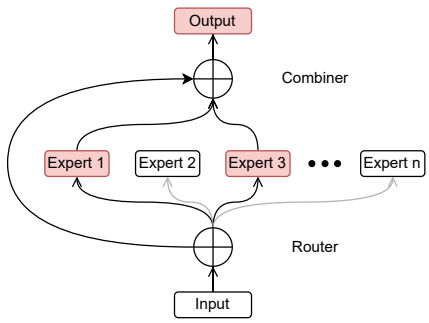

Fig. 3. An Example Design of Mixture-of-Experts (MoE).

2.3.1 Mixture-of-Experts Framework Design. Mixture-of-Experts (MoE) is a combinatory approach that leverages individual learning techniques as experts specializing in distinct sub-spaces of the input space [47]. MoE, proposed by Jacobs et al. in 1991 [21], uses the idea of dividing the input space into sub-spaces, trains experts within each sub-space, and combines the knowledge of experts using a router, also called gating function. The core idea behind MoE is to select suitable experts for a particular input using a router. Typically, MoE contains the following three components:

- **Experts:** These are individual models or neural networks trained to specialize in different parts of the input space or in different tasks. Experts can be of various types, such as linear models, decision trees, or deep neural networks.
- **Router (Gating function):** The router is a model that determines which expert or combination of experts should handle a particular input. It is typically implemented as a neural network and can be trained alongside the experts. Usually, not all experts are employed for inference; the router selects the most appropriate subset of experts for each input.
- **Combiner:** The combiner takes the outputs of the experts, weighted by the probabilities or weights assigned by the router, and produces the final output.

Let $\theta = \{\theta_g, \theta_e\}$ represent the set of parameters of MoE, where θ_g represent the parameters for the router, and θ_e denotes the parameters of experts. Given an input vector X and an output vector

Y, the final probability of P can be formulated as:

$$P(Y|X,\theta) = Aggregation_{i=1}^{N}(P(Y,i|X,\theta)) \quad (1)$$

$$= Aggregation_{i=1}^{N}(g(i|X,\theta_g)P(Y|i,X,\theta_e^i)) \quad (2)$$

, where N donotes the number of experts, $g(i|X,\theta_g)$ denotes the probability of selecting the i-th expert to handle input data X, $P(Y|i,X,\theta_e^i)$ is the probability of Y generated by the i-th expert given X, and θ_e^i is the parameters for i-th expert. The function $Aggregation$ aggregates results from each expert. The training objective for the MoE framework is to train both θ_g and θ_e or only θ_g.

Figure 3 presents a typical design of MoE. The input is first fed to the router and the router decides the most appropriate experts to handle the input. In this case, Expert 1 and Expert 3 are selected. The results returned by those two experts are aggregated in the combiner and output as the final results.

2.3.2 Vulnerability Detection with Mixture-of-Experts. The motivation for our approach stems from the significant limitations of the prevalent "one-for-all" design in existing deep learning-based vulnerability detection techniques. Those techniques train a single model to handle all types of vulnerabilities, which often fails to capture the unique patterns and characteristics of different vulnerability types. This is particularly problematic for rare vulnerabilities that are underrepresented in training datasets, leading to low performance and increased security risk.

In contrast, the MoE framework allows for specialization, where each expert focuses on detecting vulnerabilities specific to a particular CWE type. This inherent partitioning aligns well with the MoE approach, where each expert can be dedicated to a specific CWE type. By leveraging the MoE framework, our goal is to improve the overall accuracy and efficiency of vulnerability detection, making it more suitable for real-world applications where diverse and specific vulnerabilities need to be addressed effectively.

3 Methodology

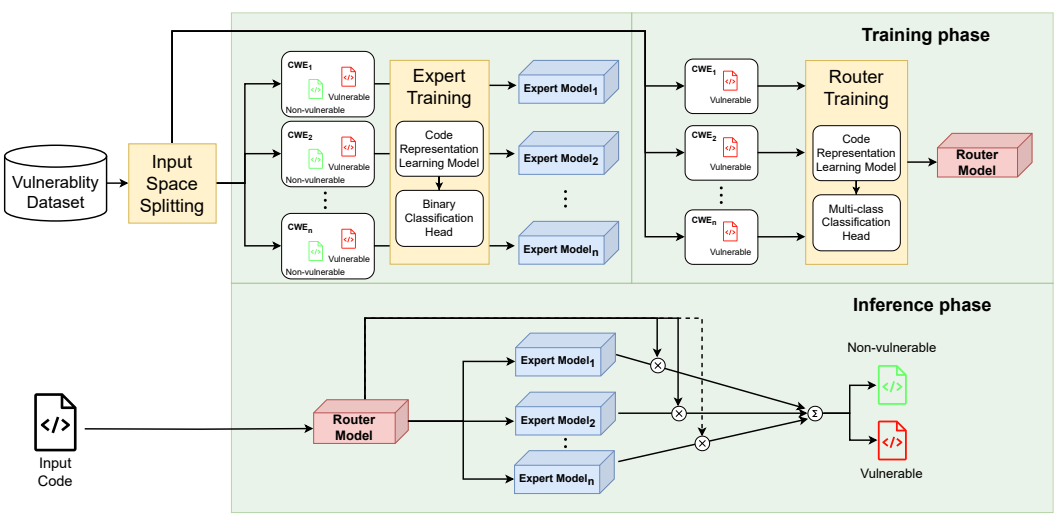

Fig. 4. The pipeline of MoEVD.

In this section, we present the design and implementation of MoEVD. Figure 4 illustrates the overall pipeline of MoEVD, which leverages the Mixture-of-Experts (MoE) framework to identify

and detect vulnerabilities with diverse patterns. During the training phase, we focus on training two key components within the MoE framework: the experts and the router. Each expert is specialized and trained to handle a specific CWE type (or combination) of vulnerabilities. Meanwhile, the router is trained using a CWE type classification task, enabling it to recognize the characteristics of specific CWE types and direct the input code to the appropriate experts. At inference time, the router assigns the input code to the most appropriate experts based on its characteristics. The selected experts then process the input code to predict vulnerabilities, and their outputs are aggregated to produce the final prediction. Essentially, MoEVD decomposes the task of binary vulnerability detection task into a CWE type classification task and a CWE-specific vulnerability detection task so that specific experts can be trained to handle a specific set of vulnerabilities instead of handling all types of vulnerabilities within one model.

3.1 Splitting Input Space Based on CWE Types

The Mixture-of-Experts (MoE) framework provides a promising solution to the limitations of one-in-all models. To effectively leverage the MoE framework, it is crucial to split the input space into sub-spaces, allowing each expert to handle a specific sub-space. This division enables the specialization of experts, where each expert is trained to focus on and master the nuances of its designated sub-space, leading to improved performance in their respective areas.

A common classification system to classify vulnerabilities is Common Weakness Enumeration (CWE) types, each representing a distinct category of software weaknesses, ranging from buffer overflows and injection flaws to improper authentication and information leakage [7]. These vulnerabilities of different CWE types exhibit different and often complex patterns that require specialized knowledge to detect effectively [2, 15].

To leverage the MoE framework, we split the input space of vulnerabilities into sub-spaces based on their CWE types. However, it is infeasible to train an expert for each specific CWE type directly for several reasons. First, there are too many CWE types (e.g., the BigVul dataset [11] contains 88 CWE types). Training an expert for each individual CWE type would make the number of experts excessively large and incur significant overhead in expert training. Second, the distribution of different CWE types of vulnerabilities follows a long-tailed pattern, meaning some CWE vulnerability types are rare and have only a few vulnerable instances [49]. It is challenging to train a robust expert on such rare CWE types.

To address the aforementioned challenges, we construct a CWE tree to present the hierarchical relationships and select the top-level CWE types as the basis for input space splitting. As introduced in Section2, CWE follows a hierarchical structure that naturally clusters CWEs with similar vulnerability patterns, and experts trained on the parent CWE types are capable of handling their child CWE types as well.

In this study, we begin by selecting the top-level CWE types (e.g., depth=1) as the basis for input space splitting. However, we found that several CWE types (e.g., CWE-388 and CWE-320) do not have sufficient instances. For example, in the BigVul vulnerability detection dataset, these types have less than 100 instances. For these cases, we aggregate all such top-level CWE types into a single category, named CWE-agg. This aggregation ensures that each CWE type has a sufficient number of instances for effective training while reducing the overall model complexity. We ended up with 12 CWE types. In other words, we need to train one expert for each of those CWE types with 12 experts in total.

3.2 Training Phase

As introduced in Section 2, an MoE framework typically has three components: experts, router, and combiner. To reduce the training complexity, we train the experts and the router separately.

Below, we elaborate on the process of training experts and the router, and aggregating the output of experts into a final prediction.

3.2.1 Expert Training. Vulnerabilities of the same CWE type share similar characteristics. Therefore, for each CWE type, we train a specific expert model. For a given CWE type CWE_i, our goal is to train an expert model (Expert Model$_i$) to predict the probability of an input code c being vulnerable vul, i.e., $\hat{y} = P(vul|CWE_i, c, \theta_e^i)$, where θ_e^i are the parameters for Expert Model$_i$. Assume we have a training dataset $\mathcal{D}_{CWE_i} = \{(c_j, y_j)\}_{j=1}^{N_{CWE_i}}$ for the CWE type CWE_i, where c_j is an input code instance and y_j is the corresponding ground truth label (i.e., vulnerable or non-vulnerable). The overall loss for the expert model of CWE type CWE_i over the entire dataset can be defined as the cross-entropy loss:

$$\mathcal{L}_{CWE_i}(\mathcal{D}_{CWE_i}, \theta_e^i) = -\frac{1}{N_{CWE_i}} \sum_{j=1}^{N_{CWE_i}} y_j \log(\hat{y}_j) \tag{3}$$

To construct the training data $\mathcal{D}_{CWE_i}$, we consider the vulnerable code of the specific CWE type (i.e., CWE_i) as positive labels, while treating all other code (including those labeled as other CWE types and all non-vulnerable code) as negative. This data construction strategy allows each expert to specialize in detecting vulnerabilities related to its specific CWE type, providing a more focused and accurate detection model for each type. This strategy is a key aspect of MoEVD, distinguishing it from other vulnerability detection models that classify code simply as vulnerable or non-vulnerable without considering the specific CWE type (see more discussion in Section 6.1).

We construct experts using pre-trained models (e.g., CodeBERT) as the backbone and fine-tune an individual expert for each CWE type by following previous studies [16, 35, 46]. We initialize each expert model with the pre-trained weights and add a binary classification head. Then we optimize the model parameters θ_e by minimizing the overall loss function $\mathcal{L}_{CWE_i}(\mathcal{D}_{CWE_i}, \theta_e^i)$.

3.2.2 Router Training. The goal of this step is to train a router that can distinguish the code characteristics of different CWE types, so that the appropriate experts can be selected during routing. We formulate the task of router training as a CWE type multi-class classification task. Given an input code c, the task is to classify its CWE type, $CWE_i \in \{CWE_1, CWE_2, \ldots, CWE_i, \ldots\}^N$, where N is the number of CWE types. The training objective is to learn a model to predict the probability of the code c being categorized as a specific CWE type CWE_i, $g_i(CWE_i|c, \theta_g)$. To make the router proficient in differentiating between the features of different CWE types and recognizing the nuances of various vulnerabilities, we do not include non-vulnerable code in the training dataset. Let $\mathcal{D}_R = \{(c_j, t_j)\}_{j=1}^{N_R}$ be the training dataset for the router, where c_j is an input code instance and t_j is the corresponding CWE type label, N_R is the size of the training data for the router. The probability of code c_j being classified as the ith class CWE_i by the router model g is given by:

$$\hat{t}_{ji} = g(CWE_i|c_j, \theta_g)$$

Note that the CWE types suffer from long-tail issues [49], meaning CWE types are highly imbalanced. To mitigate this issue, we use Focal Loss [27], defined as:

$$\mathcal{L}_{\text{focal}}(p_t) = -\alpha_t (1 - p_t)^\gamma \log(p_t) \tag{4}$$

where p_t is the predicted probability for the true class. α_t is a balancing factor for class t which is used to balance the importance of different classes. α_t can be set higher for minority classes to ensure they contribute more to the loss, helping the model pay more attention to underrepresented classes. γ is a focusing parameter to down-weight easy examples and focus on hard examples.

We adopt Focal Loss to our multi-class classification task and the overall focal loss for the router model over the entire dataset $\mathcal{D}_R$ can be defined as:

$$\mathcal{L}_R(\mathcal{D}_R, \theta_g) = -\frac{1}{N_R} \sum_{j=1}^{N_R} \sum_{i=1}^{N} \alpha_i (1 - \hat{t}_{ji})^{\gamma} t_{ji} \log(\hat{t}_{ji}) \tag{5}$$

where N is the number of CWE types, α_i is the balancing factor for class CWE_i, γ is the focusing parameter, t_{ji} is the ground truth label for code c_j for class CWE_i, and $\hat{t}_{ji}$ is the predicted probability for code c_j for class CWE_i.

Similar to expert training, we construct a router by fine-tuning a pre-trained model (e.g., CodeBERT) by adding a multi-class classification head. We select the pre-trained model because it has demonstrated proficiency in capturing the patterns from source code [24, 46, 50], which is essential for our router to assign input code to the appropriate experts based on their CWE patterns. We initialize the router model with the pre-trained weights and then optimize the parameters θ_g by minimizing the overall loss function.

3.2.3 Combiner. The combiner aggregates the output of the selected experts using a weighted summing mechanism:

$$P(vul|c) = \sum_{i=1}^{K} Softmax(g(CWE_i|c, \theta_g))\ P(vul|CWE_i, c, \theta_e^i) \tag{6}$$

K is the number of experts that are selected to handle the code c. In this study, we choose the top K experts most suitable for handling c based on the probabilities $g_i(c, \theta_g)$ produced by the router. Instead of using all experts with different weights, using only the top K experts reduces the inference cost and reduces the impact/bias caused by irrelevant experts. Note that we use the probability of each expert produced by the router output as the weight to control the contribution from each selected expert.

The $Softmax()$ function ensures that the probabilities for the selected top K experts sum up to 1. For example, setting K to 2 and the probabilities for the top 2 experts are 0.45 and 0.15. After Softmax normalization, their probabilities are 0.57 and 0.43, respectively.

We investigate the impact of different K values in MoEVD in Section 5.4. Our experiments suggest by selecting $K = 2$, MoEVD provides the best trade-off between performance and inference time agreeing with existing practices on the selection of K [39].

3.3 Inference Phase

During the inference phase, the router processes both vulnerable and non-vulnerable input code. The router first assigns the input code to one or more experts based on the CWE classification result. Then the selected experts further predict whether the input code belongs to a specific vulnerability or not. The aggregation mechanism combines the outputs from the experts to make a final prediction on whether the code is vulnerable or non-vulnerable. By leveraging the specialized knowledge of each expert and the ability of the router to classify CWE types, MoEVD is able to outperform existing tools in vulnerability detection.

Note that our router is trained exclusively on vulnerable code, it may lack specific knowledge about non-vulnerable code. One question raised here is which expert will be assigned for the non-vulnerable code. According to the design of our router, non-vulnerable code will be assigned to the experts that handle vulnerable code with the most similar characteristics to that of the input non-vulnerable code, so that it can be further identified by the corresponding experts. However, even though the non-vulnerable code is assigned to other experts, we do not need to worry about

this since we train all our experts including the data of the entire same set of non-vulnerable code. In theory, all experts should have consistent capability in identifying non-vulnerable code.

4 Experiment Design

In this section, we outline our research questions (RQs), describe the datasets used, define the evaluation metrics, present our analysis approach for each RQ, and provide implementation details.

4.1 Research Questions

We evaluate MoEVD across various dimensions to address the following research questions:

- *RQ1: How effective is MoEVD compared with SOTA approaches in vulnerability detection?*
- *RQ2: How effective is the MoE framework within MoEVD?*
- *RQ3: How effective is MoEVD in detecting each CWE type and handling long-tailed distributions compared to SOTA approaches?*
- *RQ4: What is the impact of varying the number of experts on the performance of MoEVD?*

In **RQ1**, we aim to assess the overall performance of MoEVD compared to current SOTA approaches. **RQ2** focuses on evaluating the effectiveness of the MoE framework in routing inputs to the most appropriate experts. **RQ3** examines MoEVD's effectiveness in detecting various CWE types and its ability to handle long-tailed distributions compared to SOTA approaches. **RQ4** investigates the impact of selecting different numbers of experts on the performance of MoEVD.

4.2 DLVD Baselines

In our study, we select two representative groups of DLVD baselines, graph-based models and transformer-based models:

- **Graph-based models:** We first select three of the most commonly used models, SySeVR [25], Devign [50] and Reveal [5]. SySeVR leverages the abstract syntax tree (AST) graph generated from code. The rest two models leverage graph neural network (GNN) [37] to learn code feature representations and have shown promising results. We also employ LIVABLE [42], which is so far the state-of-the-art approach that combines graph-based model with text-based model.
- **Transformer-based models:** We select two base models CodeBERT [13] and UniXcoder [17], which have been shown to be effective in vulnerability detection [13, 16, 17]. Additionally, we select the recent SOTA approach CausalVul [35], which introduced calculus-based causal learning. For a fair comparison, we implement CausalVul on these two models, resulting in $\text{CausalVul}_{CodeBERT}$ and $\text{CausalVul}_{UniXcoder}$.

4.3 Experiment Dataset

We select our experiment vulnerability detection dataset based on two criteria: 1) Diversity. They must be collected from a wide range of code repositories to maintain various types of vulnerability. 2) Annotations. They must contain CWE type or CVE-id annotation for training MoEVD. Therefore, we select the BigVul dataset [11] as our experiment dataset, which is widely used in prior studies [16, 24, 42, 46]. The BigVul dataset contains CVEs from 2002 to 2019, extracted from over 300 different open-source C/C++ projects, encompassing 88 different CWE vulnerability types. It contains 10,547 vulnerable functions and 168,752 non-vulnerable functions. Following the same experiment setting of previous studies [35, 46], we use the same 80%/10%/10% split as train/val/test data for the BigVul dataset. We also use the same exact dataset partition used in CausalVul for a fair comparison. After aggregating CWE types as described in Section 3.1, we obtain 12 CWE types.

Table 1. Overview of the studied dataset.

Dataset	Granularity	#Project	#Vuln	#Non-Vuln	#CWE
BigVul [11]	function	348	10,547	168,752	88

4.4 Evaluation Metrics

We use F1-score, precision, and recall as our evaluation metrics, which are widely used in previous studies [5, 24, 35, 46, 50]. The F1-score provides a balance between precision and recall, offering a single metric that considers both false positives and false negatives. Precision measures the accuracy of positive predictions, while recall measures the ability to identify all relevant instances. Unlike previous research, we do not use accuracy as an evaluation metric due to the highly imbalanced nature of vulnerability detection datasets, which can result in misleading accuracy values dominated by the majority class [19].

4.5 Approach for Research Questions

4.5.1 Approach of RQ1. To demonstrate the effectiveness of MoEVD, we compare it with graph-based models (Devign [50], Reveal [5], LIVABLE [42]) and transformer-based models (CodeBert [13], UniXcoder [17], and CausalVul [35]). Depending on different base transformer models (CodeBERT and UniXcoder), there are two variants of CausalVul, namely CausalVul$_{CodeBERT}$ and CausalVul$_{UniXcoder}$. Similarly, we also implement MoEVD using two different base transformer models, resulting in two variants, namely MoEVD$_{CodeBERT}$ and MoEVD$_{UniXcoder}$.

4.5.2 Approach of RQ2. In RQ2, we evaluate the effectiveness of the MoE framework of MoEVD. To do that, we keep every trained expert untouched and compare MoEVD with two different variants: ensemble and random as described below.

- **Ensemble** Ensemble techniques train a machine learning (ML) model to fuse multiple models. We compare our MoE framework with ensemble techniques, replacing the MoE framework with ensemble techniques while keeping all other components unchanged. Specifically, we use the prediction probabilities of the positive class (vulnerable) from each expert model as input features for training an ensemble model. We select Random Forest (RF)[4], XGBoost[6], CatBoost [34], and LightGBM [22] as baselines due to their widespread use and competitive performance in classification tasks [3, 40, 48]. For simplicity, we call those variants with different ensemble methods as Ensemble$_{RF}$, Ensemble$_{XGBoost}$, Ensemble$_{CatBoost}$, Ensemble$_{LightGBM}$.
- **Random Router:** To test the effectiveness of our router, we also compare MoEVD with a variant in which the router selects experts *randomly*. In this variant, we replace our router with a random expert selector and keep all other components unchanged. We define this variant as MoEVD $_{random}$.

We evaluate the performance of the above two settings and compare them with MoEVD. In this RQ, we choose the best-performing implementation of MoEVD, MoEVD$_{CodeBERT}$ to compare with.

4.5.3 Approach of RQ3. To answer RQ3, we compare the performance between the selected best-performing baseline (CausalVul$_{CodeBERT}$) and MoEVD$_{CodeBERT}$. We first analyze the performance (recall) of selected models in detecting vulnerabilities across each CWE type. BigVul dataset has 88 CWE types. Note that the model performance may not be stable enough on some CWE types which have an extremely small amount of vulnerabilities. In order to reduce the potential performance bias, we group the CWE types that have fewer than 10 vulnerable codes together as CWE-N$\leq$10.

Note that for RQ3, CWE types are not aggregated by the CWE hierarchy; therefore, they are not the same as the CWE types used for training the expert models.

To evaluate the effectiveness of MoEVD in the long-tailed scenario, we first construct the long-tailed dataset. We split the BigVul dataset into two groups (the head group and the tail group) by following the settings from previous research [49]. More specifically, we categorize vulnerabilities into two groups: the head group, comprising the most frequent CWEs that constitute at least 50% of BigVul, and the tail group, consisting of the remaining vulnerabilities. In the BigVul dataset, 4 CWE types (CWE-119, CWE-noinfo, CWE-20, and CWE-399) belong to the head group (56.2% of vulnerabilities in total), while the other 84 CWE types belong to the tail group (43.8% of vulnerabilities in total). Finally, we compare the performance of $\text{CausalVul}_{CodeBERT}$ and $\text{MoEVD}_{CodeBERT}$ between the two groups, respectively.

4.5.4 Approach of RQ4. In MoE, the number of experts K is an important parameter. In this RQ, we aim to investigate the impact of the value K on MoEVD's performance. Our goal is to find the value of K that achieves the optimal balance between performance and computational cost.

4.6 Implementation Details

We downloaded the official pre-trained models and tokenizers for all evaluated transformer models from HuggingFace [43]. We implemented and conducted all experiments using PyTorch [32] and Autogluon [9]. To train the expert and router models, we used a batch size of 32, a learning rate of 1e-5, the AdamW [28] as the optimizer, and trained the model for 10 epochs. For Focal Loss, we use the default setting [14], i.e., setting the focusing parameter γ to 1 and the balancing factor α_t as the inverse of its percentage of number of samples for each CWE type. To optimize computational efficiency, we used the bfloat16 (brain floating point) mixed precision. All experiments were conducted on a Linux server equipped with four Nvidia RTX 3090 GPUs, 24 CPU cores, and 128GB of memory. Each expert's training time is approximately 1 hour, and the router model's training time is approximately 10 minutes. During inference, when we set the number of experts to $K = 2$, MoEVD can scan more than 100 code snippets per second. For training the ensemble model in4.5.2, We used a state-of-the-art AutoML tool Autogluon [10] for training the ensemble model. Autogluon trains ML models with hyperparameter tuning, bagging, stacking and other techniques to optimize the performance, we select the four highest-performing ensemble models as our baselines.

We reproduce all DLVD baselines by using the replication packages provided in their works [5, 13, 17, 35, 42, 50]. We also adopt the same hyperparameter settings in their works.

5 Results

5.1 RQ1: MoEVD vs. DLVD Baselines

MoEVD outperforms the best baseline by 12.8%. Table 2 presents the performance comparison between our approach and studied baselines. We observe that MoEVD outperforms all SOTA baselines, including graph-based and transformer-based models in all evaluated metrics (F1-score, precision, and recall). Specifically, $\text{MoEVD}_{CodeBERT}$ achieves the best F1-score of 0.44, improving baselines from 12.8% to 69.2%. When looking at recall, $\text{MoEVD}_{CodeBERT}$ also achieves the best performance (0.47), while in terms of precision, $\text{MoEVD}_{UniXcoder}$ achieves the best performance (0.47).

MoEVD is able to deliver the best performance for all base models. For both CodeBERT and UniXcoder variants of MoEVD, MoEVD outperforms the base model and the corresponding CausalVul variant. $\text{MoEVD}_{CodeBERT}$, which is the best-performing variant based on CodeBERT, achieves the highest F1-score of 0.44, achieving a 15.8% improvement over CodeBERT with an F1-score of 0.38. MoEVD also consistently outperforms other approaches in terms of recall, which

indicates MoEVD is effective in recognizing more vulnerabilities, highlighting the effectiveness of MoEVD over existing one-for-all DLVD approaches.

Table 2. The performance of different VD approaches on BigVul dataset in terms of studied metrics.

Model Type	Approaches	F1-score	Precision	Recall
Graph-based	SySeVR	0.19	0.31	0.14
	Devign	0.27	0.30	0.25
	Reveal	0.26	0.23	0.30
	LIVABLE	0.39	0.40	0.37
Transformer-based	CodeBERT	0.38	0.45	0.33
	CausalVul$_{CodeBERT}$	0.39	0.43	0.36
	MoEVD$_{CodeBERT}$	**0.44**	0.42	**0.46**
	UniXCoder	0.38	0.46	0.32
	CausalVul$_{UniXcoder}$	0.39	0.35	0.45
	MoEVD$_{UniXcoder}$	0.43	**0.47**	0.40

MoEVD significantly outperforms all SOTA baselines. MoEVD's implementation with the CodeBERT model (MoEVD$_{CodeBERT}$) achieves the best F1-score of 0.44 with 12.8% improvement over the best SOTA baseline. Additionally, MoEVD's implementation with the UniXcoder model (MoEVD$_{UniXcoder}$) achieves the highest precision of 0.47 and a notable F1-score of 0.43.

5.2 RQ2: Effectiveness of MoE

MoEVD outperforms variants with ensemble methods by at least 12.7% F1-score. Table 3 presents the comparison between MoEVD and its variants with ensemble methods. MoEVD significantly outperforms those variants in terms of F1-score and recall. Specifically, compared to the best-performing variant, Ensemble$_{XGBoost}$, MoEVD achieves a 12.7% improvement in F1-score. These results suggest that the MoE framework is superior to other ensemble methods. A potential reason is that combining the results of all experts introduces additional noise, leading to less accurate final results.

To understand the impact of the router, we compare the performance of MoEVD and MoEVD $_{random}$. From Table 3, we observe that the performance of MoEVD $_{random}$ is extremely low (a F1-score of 0.08). Such a result indicates that selecting appropriate experts is crucial for our approach.

Table 3. The performance of variants with different ensemble methods and routers.

Variant	F1-score	Precision	Recall
Ensemble$_{LightBGM}$	0.38	**0.54**	0.29
Ensemble$_{CatBoost}$	0.39	0.50	0.31
Ensemble$_{XGBoost}$	0.39	0.51	0.31
Ensemble$_{RF}$	0.38	0.49	0.30
MoEVD $_{random}$	0.08	0.07	0.09
MoEVD	**0.44**	0.42	**0.46**

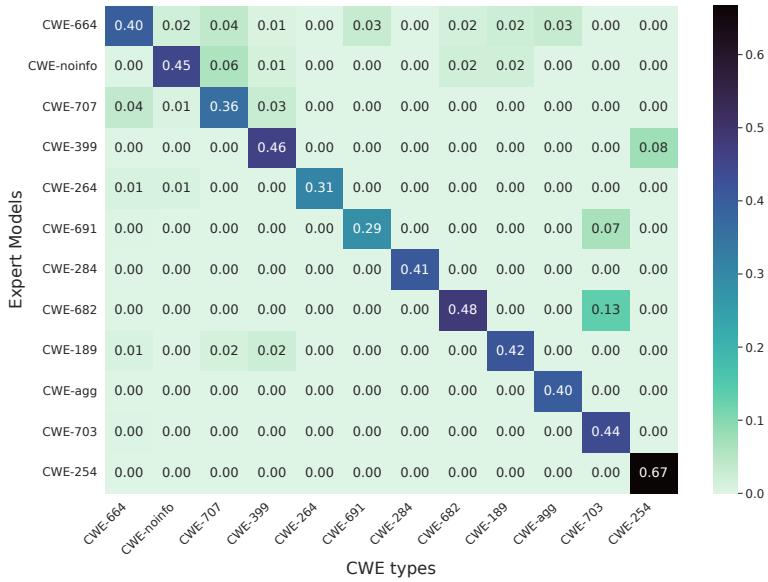

Fig. 5. The performance matrix of each expert model on each CWE type in terms of F1-score. Darker colors indicate better performance.

Table 4. The performance of MoEVD when the router can select experts correctly (Correctly selected) and wrongly (Wrongly selected).

Expert Selection	**% vuln code**	**F1-score**	**Precision**	**Recall**
Correctly Selected	63.8%	0.44	0.36	0.56
Wrongly Selected	36.2%	0.18	0.13	0.28

We further measure the router's accuracy in selecting the correct expert for vulnerable input code. If the router can select the correct expert within the top K assigned experts, we consider it correct, otherwise, we consider it incorrect. The ground truth of an expert is determined by its CWE types after aggregation. If a vulnerable code of a CWE type A is assigned to an expert whose training data consists of CWE A, we consider the vulnerable code to be assigned correctly. As shown in Table 4, in 63.8% of the cases, the router can select the correct experts. In other words, there is still room for improvement (see more details in Section 6.3).

The router is an essential component of the MoE framework, correctly assigning the vulnerability to the correct CWE expert is important. To further understand how the router's performance affect the overall VD performance of MoEVD, we compare the performance between the cases where experts are correctly selected by the router and experts are wrongly selected. As the results shown in Table 4, if the experts can be selected correctly, the performance is significantly better (by 133.7%) than those who failed.

The subpar performance when the expert is wrongly selected is expected due to the design of MoE. Each expert is trained on a specific CWE type and cannot identify the pattern of other CWE types. This can be observed in Figure 5, where we show the performance of each expert across different CWE types. As expected, the experts only perform well on their own specific CWE type.

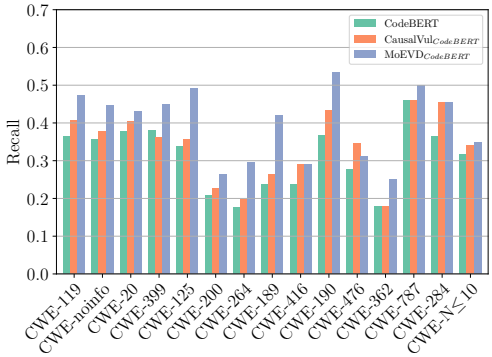

Fig. 6. Recall of CodeBERT, MoEVD$_{CodeBERT}$, and CausalVul$_{CodeBERT}$ on different CWE types. The CWE types are displayed in descending order of frequency, from left to right.

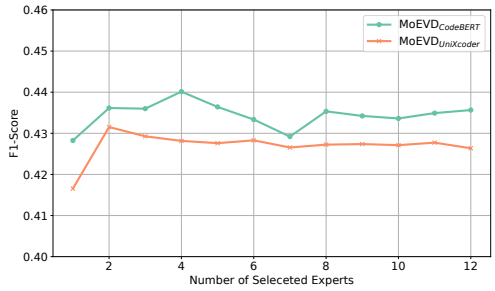

Fig. 7. F1-score of MoEVD$_{CodeBERT}$ and MoEVD$_{UniXcoder}$ with different number of selected experts (K).

Table 5. Performance achieved by CodeBERT, MoEVD$_{CodeBERT}$, and CausalVul$_{CodeBERT}$ on Head and Tail groups.

	Head			**Tail**		
	F1-score	**Precision**	**Recall**	**F1-score**	**Precision**	**Recall**
CodeBERT	0.34	0.30	0.38	0.27	0.24	0.32
CausalVul$_{CodeBERT}$	0.34	0.30	0.39	0.26	0.23	0.31
MoEVD$_{CodeBERT}$	**0.37**	0.29	**0.50**	**0.29**	0.23	**0.42**

Mixture-of-Experts (MoE) design outperforms all ensemble models. The router plays a crucial role in MoEVD design, and significantly impacts overall MoEVD performance.

5.3 RQ3: Effectiveness of MoEVD across CWE Types

MoEVD outperforms baselines in most CWE types in terms of all studied metrics, typically recall and F1-score. Figure 6 presents the recall achieved by CodeBERT, CausalVul$_{CodeBERT}$, and MoEVD$_{CodeBERT}$ on all CWE types. We observe that in most of the CWE types (14 out of 15), MoEVD$_{CodeBERT}$ outperforms CodeBERT and CausalVul$_{CodeBERT}$ with a range of improvement from 9% (CWE-787) to 77.78% (CWE-189) in terms of recall. In terms of F1-score, we observe a similar trend that in 11 out of 15 CWE types, MoEVD$_{CodeBERT}$ outperforms CausalVul$_{CodeBERT}$, and 9 out of 15 in terms of precision. Due to the page limit, we put the complete results of F1-score and precision in our replication package [1].

MoEVD$_{CodeBERT}$ outperforms SOTA baselines CausalVul$_{CodeBERT}$ and CodeBERT in both the head and the tail groups in terms of F1-score and recall, and achieves similar precision compared to SOTA baselines. Table 5 presents the results of studied approaches on the head and the tail groups. We observe an improvement of MoEVD$_{CodeBERT}$ over the SOTA baselines in both the head and the tail groups. In terms of F1-score, MoEVD$_{CodeBERT}$ improves CodeBERT by 9.8% and CausalVul$_{CodeBERT}$ by 7.3% in the head group. In the tail group, MoEVD$_{CodeBERT}$ improves CodeBERT by 7.3% and CausalVul$_{CodeBERT}$ by 11.4%. When comparing recall, MoEVD$_{CodeBERT}$

achieves a significant improvement of 31.2%, compared with baselines while maintaining a similar precision. The results indicate that, unlike baselines, when improving the binary classification performance, MoEVD does not sacrifice the performance in long-tailed CWE types. On the contrary, benefiting from the MoE design that enables the model to learn various vulnerability patterns individually, MoEVD improves the performance on the tail group.

MoEVD$_{CodeBERT}$ outperforms SOTA baselines on most of the CWE types in terms of F1-score and recall. For instance, MoEVD$_{CodeBERT}$ improves the recall of SOTA baseline by a range from 9% to 77.8%. MoEVD$_{CodeBERT}$ also mitigates long-tailed issue by improving F1-score of SOTA baselines on the tail group at least by 7.3%.

5.4 RQ4: Impact of the Number of Selected Experts

More experts do not yield significant performance improvement. Figure 7 presents the F1-score of MoEVD$_{CodeBERT}$ and MoEVD$_{UniXcoder}$ with a different number of selected experts (K). We observe that no significant advantages are achieved by increasing the number of experts. The worst performance for both models occurs at $K = 1$. MoEVD$_{CodeBERT}$ achieve its optimal performance when $K = 4$, and MoEVD$_{UniXcoder}$ achieves its optimal performance at $K = 2$. When K is greater than 2, the increase of the number of selected experts does not yield significant benefits for both models. To balance the overhead and the performance, we set K to 2 in this study.

One possible reason for the lack of significant improvement with more experts is that each expert is only effective in its own CWE type and cannot identify the vulnerability for other types as indicated in Figure 5. Assigning additional experts will reduce the contribution of the most effective experts therefore adding more noise to the combiner.

We do not observe significant advantages in increasing the number of experts. Employing a single expert yields sub-optimal results. Through our experiments, we find $K = 2$ achieves an optimal balance between inference time and performance.

6 Discussion

In this section, we discuss our special design in terms of expert training, the overhead, and the potential future research direction.

6.1 Expert Training Task

Different from 1) classifying vulnerability types for a given vulnerability and 2) identifying whether a given input is a vulnerability or not, our experts learn vulnerability knowledge by identifying whether the given input belongs to a specific type of vulnerability or not. As described in Section 3, when training an expert for a specific CWE type, we only consider the vulnerable code of that type as positive, and all other code (including other CWE type vulnerable code and non-vulnerable code) as negative. By doing so, an expert is more focusing on learning the pattern of a specific type of vulnerability, improving the performance on distinguishing the target type of vulnerability.

To validate our approach, we also trained a variant of MoEVD$_{CodeBERT}$ using only non-vulnerable data as the negative label, excluding code from other CWE types. The F1-score dropped from 0.44 to 0.41. This result indicates that using data from other CWE categories as negative helps the experts learn to distinguish between different types of vulnerabilities more effectively.

6.2 Overhead of MoE Compared With One-for-All Model

The training cost of MoEVD depends on the number of experts to train. For example, in the BigVul dataset, we have 12 CWE types and consequently we have 12 experts to train. Assume the total

tunable parameters of a one-for-all model with the same base model (e.g. CodeBERT) is N. Then, MoEVD have a total of $12N + R$ tunable parameters, where R is the tunable parameters of the router network. During inference, the computation cost depends on the number of experts K as discussed in RQ4. Assume the inference cost of a one-for-all model is x, and we set $K = 2$, then the total computation cost will be $2x + r$ where r is the inference cost of the router. In our case, we use the same base model for experts and the router, and we set K to 2. The inference cost is $3x$. The time cost under our experiment setting can be found in section4.6.

6.3 Potential Future Direction

Our research is the first study in vulnerability detection that employs the MoE framework. The design of MoEVD is straightforward, utilizing the vanilla MoE framework proposed by Jacobs et al. [21], which already demonstrates a significant improvement over the one-for-all design. In RQ2 (table 4), we investigate the performance of the router, and we notice that only 63.8% of the vulnerable code was correctly directed to the appropriate experts. In the ideal setting, if all the vulnerable code can be correctly assigned to the appropriate experts, the F1-score for MoEVD would be improved from 0.44 to 0.51. Thus, there remains a large margin to improve with the MoE framework and we believe that leveraging MoE in vulnerability detection is key to making DLVD methods more practical in real-world scenarios. One possible future direction is to improve the accuracy of the router to select the appropriate experts. Another possible direction is to leverage a more advanced MoE framework [12, 23, 39]. We encourage future research to investigate those two directions.

6.4 Threats to Validity

6.4.1 Internal Validity. MoEVD 's performance may be influenced by the distribution of CWE types within the training data, particularly for unseen CWEs. To mitigate this potential bias, we trained MoEVD on the BigVul dataset, which is one of the most comprehensive datasets, containing over 10,000 vulnerability instances across 88 distinct CWE types. For vulnerabilities belonging to unseen CWE types, MoEVD can be easily extended by training and adding specific experts for new vulnerabilities and re-training the router.

6.4.2 External Validity. Threats to external validity relate to the generalizability of our findings. In this study, we evaluate MoEVD using CodeBERT and UniXCoder as base models due to their widespread use in code-related tasks. While our findings might not be generalized to other models. However, As a framework, MoEVD is designed to be flexible and can integrate any model as an expert, future research is encouraged to explore the performance of MoEVD with a broader range of models.

7 Related Work

7.1 Machine Learning-Based Vulnerability Detection

DLVD approaches could be categorized into two families based on the way of feature extraction: token-based or graph-based approaches. In the token-based approach [16, 25, 26, 35], code is considered as a sequence of tokens and is represented as a vector via text embedding techniques (e.g., GloVe [33] and CodeBERT [13]). For instance, LineVul [16] leverages CodeBERT [13] to embed the whole sequence of tokens in a function for vulnerability detection. Instead of considering the whole code, approaches such as VulDeePecker [26] and SySeVR [25], extract slices from points of interest in code (e.g., API calls, array indexing, pointer usage, etc.) and use them for vulnerability detection since they assume that different lines of code are not equivalently important for vulnerability detection. Another family of approaches [5, 20, 24, 50] consider code as graphs and incorporate

the information in different syntactic and semantic dependencies using graph neural network (GNN) [44] when generating the representation vectors. Different types of syntactic/semantic graphs, (e.g., abstract syntax tree (AST), program dependency graph (PDG), code property graph (CPG)) can be used. For example, Devign [50] and Reveal [5] leverage code property graph (CPG) [45] to build their graph-based vulnerability detection model. IVDetect [24] and LineVD [20] consider the vulnerable statements and capture their surrounding contexts via a program dependency graph. Both existing graph-based and text-based models follow the one-for-all design, in which only one final model is trained to handle all types of vulnerabilities. To address the limitation of previous existing approaches, we propose MoEVD to leverage the MoE framework to assign appropriate experts for handling the vulnerabilities of specific CWE types.

7.2 Mixture-of-Experts (MoE) in Software Engineering Tasks

The Mixture-of-Experts (MoE) is relatively new in the software engineering domain and we only found it was leveraged in defect prediction [31, 38]. Omer et al. presented a Mixture-of-Experts (MoE)-based approach named ME-SFP [31] to perform software defect prediction. ME-SFP uses the experts trained using f decision trees and multilayer perceptions and a Gaussian mixture model as a gating function to select appropriate experts. Aditya and Santosh [38] explored two variations of the MoE method, a mixture of implicit experts (MIoE) and a mixture of explicit experts (MEoE) for the defect prediction task. The MIoE method randomly partitions the input data into a number of sub-spaces using an employed error function. The local experts become specialized in each sub-space. These specialized experts are further used for the final prediction. The MEoE method explicitly partitioned the input data into a number of sub-spaces using a clustering method before the training process. Each expert is assigned one of these sub-spaces. The local experts become specialized in these sub-spaces, which are subsequently used for the final prediction. In this study, we follow the MEoE method, in which we partition the input space based on CWE types, which is more suitable for the nature of the task.

8 Conclusion

In this work, we present MoEVD, the first approach in vulnerability detection using the Mixture-of-Experts (MoE) framework. MoEVD mitigates the limitations of traditional "one-for-all" design by leveraging specialized experts for different CWE types. MoEVD achieved the highest F1-score of 0.44 on the BigVul vulnerability detection dataset, surpassing the best existing approach by 12.8%. In handling long-tailed distributions, MoEVD maintains strong performance across both common and rare CWE types. Benefiting from its ability to learn various vulnerability patterns individually, MoEVD improved the performance on the tail group by 31.2% in terms of recall while maintaining similar precision. As the first application of the MoE framework in vulnerability detection, our work suggests the possibility of leveraging specialized models and indicates potential for more advanced MoE architectures to further enhance detection capabilities.

Data Availability

We make all the datasets, results and the code used in this study openly available in our replication package [1].

References

[1] 2024. *MoEVD replication pakage*. Zenodo. doi:10.5281/zenodo.11661787

[2] 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)*. IEEE, 1–7.

[3] Candice Bentéjac, Anna Csörgő, and Gonzalo Martínez-Muñoz. 2021. A comparative analysis of gradient boosting algorithms. *Artificial Intelligence Review* 54 (2021), 1937–1967.
[4] Gérard Biau and Erwan Scornet. 2016. A random forest guided tour. *Test* 25 (2016), 197–227.
[5] 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 (2021), 3280–3296.
[6] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.
[7] Steve Christey, J Kenderdine, J Mazella, and B Miles. 2013. Common weakness enumeration. *Mitre Corporation* (2013).
[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 17th 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (HLT-NAACL)*. 4171–4186.
[9] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505* (2020).
[10] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data. *arXiv preprint arXiv:2003.06505* (2020).
[11] 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*. 508–512.
[12] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.
[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of EMNLP*.
[14] focal loss. 2024. Focal Loss Documentation. https://focal-loss.readthedocs.io/en/latest/generated/focal_loss.binary_focal_loss.html. Accessed: 2024-06-08.
[15] 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* (2023).
[16] 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*. 608–620.
[17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).
[18] 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* (2020).
[19] Haibo He and Edwardo A Garcia. 2009. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering* 21, 9 (2009), 1263–1284.
[20] 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*. 596–607.
[21] RA Jacobs, MI Jordan, SJ Nowlan, and GE Hinton. 1991. [a]Adaptive Mixtures of Local Experts, [o] Neural Computation, vol. 3. (1991).
[22] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).
[23] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).
[24] 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*. 292–303.
[25] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021).
[26] 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. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.
[27] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*. 2980–2988.
[28] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).
[29] MITRE. 2024. CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/

[30] 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*. 178–182.

[31] Aman Omer, Santosh Singh Rathore, and Sandeep Kumar. 2023. ME-SFP: A Mixture-of-Experts-Based Approach for Software Fault Prediction. *IEEE Transactions on Reliability* (2023).

[32] 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. *Advances in neural information processing systems* 32 (2019).

[33] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[34] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. CatBoost: unbiased boosting with categorical features. *Advances in neural information processing systems* 31 (2018).

[35] Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 2024. Towards causal deep learning for vulnerability detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–11.

[36] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering* 40, 10 (2014), 993–1006.

[37] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.

[38] Aditya Shankar Mishra and Santosh Singh Rathore. 2023. Implicit and explicit mixture of experts models for software defect prediction. *Software Quality Journal* 31, 4 (2023), 1331–1368.

[39] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[40] Robert Szczepanek. 2022. Daily streamflow forecasting in mountainous catchment using XGBoost, LightGBM and CatBoost. *Hydrology* 9, 12 (2022), 226.

[41] Shengye Wan, Joshua Saxe, Craig Gomes, Sahana Chennabasappa, Avilash Rath, Kun Sun, and Xinda Wang. 2024. Bridging the Gap: A Study of AI-based Vulnerability Management between Industry and Academia. *arXiv preprint arXiv:2405.02435* (2024).

[42] 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* (2024).

[43] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[44] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[45] 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*. IEEE, 590–604.

[46] Xu Yang, Shaowei Wang, Yi Li, and Shaohua Wang. 2023. Does data sampling improve deep learning-based vulnerability detection? Yeas! and Nays!. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2287–2298.

[47] Seniha Esen Yuksel, Joseph N Wilson, and Paul D Gader. 2012. Twenty years of mixture of experts. *IEEE transactions on neural networks and learning systems* 23, 8 (2012), 1177–1193.

[48] Yakun Zhang, Wensheng Dou, Jiaxin Zhu, Liang Xu, Zhiyong Zhou, Jun Wei, Dan Ye, and Bo Yang. 2020. Learning to detect table clones in spreadsheets. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 528–540.

[49] Xin Zhou, Kisub Kim, Bowen Xu, Jiakun Liu, DongGyun 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)*. IEEE, 40–52.

[50] 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 (2019).

Received 2024-09-11; accepted 2025-01-14