The Journal of Systems and Software 230 (2025) 112544

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Learning never stops: Improving software vulnerability type identification via incremental learning☆

Jiacheng Xue [a], Xiang Chen [a,*], Zhanqi Cui [b], Yong Liu [c]

[a] School of Artificial Intelligence and Computer Science, Nantong University, Nantong, China
[b] Computer School, Beijing Information Science and Technology University, Beijing, China
[c] College of Information Science and Technology, Beijing University of Chemical Technology, Beijing, China

ARTICLE INFO

Keywords:
Software vulnerability type identification
Incremental learning
Prompt tuning
Vulnerability description
Vulnerability code

ABSTRACT

As new vulnerabilities are continuously discovered, software vulnerability type identification (SVTI) data is dynamic. Moreover, SVTI data often exhibits a long-tailed distribution, where some vulnerability types (i.e., head classes) have numerous samples, while rare ones (i.e., tail classes) have very few. These issues present challenges for SVTI, such as catastrophic forgetting when learning new data and poor performance for rare vulnerability types. To address these challenges, we propose an approach *VulTypeIL*. Specifically, for incremental learning, we employ a hybrid replay strategy and a regularization strategy with EWC to alleviate the catastrophic forgetting issue. We also integrate focal loss and label smooth cross-entropy loss to tackle the long-tailed distribution issue. For model construction, we customize the verbalizer and hybrid prompt by fusing the Vulnerability code and description. Then, we perform prompt tuning on the pre-trained model CodeT5. To evaluate the effectiveness of *VulTypeIL*, we construct a large-scale SVTI dataset containing 6,269 vulnerabilities from 992 real-world projects. Our experimental results demonstrate that *VulTypeIL* outperforms state-of-the-art baselines (such as VulExplainer and LIVABLE) with a significant improvement. The ablation studies further confirm the effectiveness of key component settings (such as the incremental learning setting and long-tailed learning setting) in our approach.

1. Introduction

In today's rapidly evolving software landscape, vulnerabilities present significant risks, leading to data breaches, financial losses, and compromised user trust (Mallick and Nath, 2024; Jimmy, 2024; Shiri Harzevili et al., 2024). The National Vulnerability Database (NVD)[1] serves as a critical repository, documenting vulnerabilities across a wide range of software systems. As of 2024, the NVD has reported a substantial increase in vulnerabilities, emphasizing the persistent nature of this issue in modern software development. This concerning trend underscores the urgent need for robust security practices and proactive measures to mitigate the risks associated with vulnerabilities throughout the software development and maintenance lifecycle (Cheng et al., 2021; Fu et al., 2024; Cao et al., 2023; Lu et al., 2024; Wang et al., 2025b; Cai et al., 2024).

Vulnerabilities are often categorized using Common Weakness Enumeration (CWE)[2] identifiers, which enable consistent communication and a shared understanding of security issues. Recently, a typical example is the Log4j vulnerability (CVE-2021-44228),[3] which exploited a flaw in the widely used logging library, leading to widespread attacks across numerous applications. As a result, by automating the identification of vulnerability types, security experts can prioritize remediation efforts for higher-risk vulnerabilities, ensuring faster and more targeted responses to critical threats. This proactive approach is crucial for addressing the increasing complexity and frequency of vulnerabilities in modern software systems.

In our study, we refer to this task as Software Vulnerability Type Identification (SVTI). Due to the large number of vulnerability types, SVTI can be modeled as a multi-class classification problem. In Early studies, researchers mainly constructed SVTI models based on vulnerability descriptions. For example, Na et al. (2017) apply a Naïve Bayes classifier to classify CVE entries using keywords extracted from the NVD description field. Shuai et al. (2013) integrate Latent Dirichlet

☆ Editor: Yan Cai.

* Corresponding author.
E-mail address: xchencs@ntu.edu.cn (X. Chen).

[1] https://nvd.nist.gov.
[2] https://cwe.mitre.org.
[3] https://nvd.nist.gov/vuln/detail/cve-2021-44228.

https://doi.org/10.1016/j.jss.2025.112544
Received 3 March 2025; Received in revised form 19 May 2025; Accepted 18 June 2025
Available online 5 July 2025

Allocation (LDA) for topic modeling with a hierarchical SVM classifier, leveraging both topic distribution and vulnerability-specific keywords. In recent years, researchers have begun constructing SVTI models based on vulnerability code. For example, Fu et al. (2023) propose VulExplainer, a Transformer-based approach that divides CWE-IDs into subgroups and distills knowledge from TextCNN models into a Transformer. Wen et al. (2024) present LIVABLE, which utilizes GNN-based learning and adaptive re-weighting techniques to enhance the performance of SVTI. However, these previous studies still have the following limitations.

Limitation Analysis. The first limitation is that existing approaches construct SVTI models with static datasets (Wang et al., 2024). However, new vulnerabilities constantly emerge, making it difficult for SVTI models developed with these approaches to keep pace with the evolving landscape of software vulnerabilities. We refer to this issue as the dynamic characteristics of SVTI data (Wang et al., 2024). To make SVTI methods more practical, we should consider incremental learning when constructing models and propose effective approaches to mitigate the catastrophic forgetting issue (McCloskey and Cohen, 1989). **The second limitation** is that the SVTI data suffers from a long-tailed distribution issue, where some vulnerability types (head classes) have numerous samples, while rare ones (tail classes) have very few. Due to the insufficient number of vulnerability samples, the model struggles to achieve satisfactory identification results for tail vulnerabilities, and the impact is even more severe, especially when some of these tail vulnerabilities are highly critical. Therefore, when constructing the SVTI model, we need to propose effective approaches to ensure that tail vulnerabilities also receive adequate attention. **The third limitation** is that existing research primarily relies on the training from scratch paradigm (Wen et al., 2024) or the fine-tuning paradigm (Fu et al., 2023), but these paradigms have several drawbacks, including high computational costs and the risk of overfitting, particularly when the amount of labeled data is limited. To overcome these challenges, we aim to construct the model using the prompt tuning paradigm (Liu et al., 2023). This paradigm requires the design of customized prompts and verbalizers that are tailored to the SVTI task, allowing for the full utilization of the knowledge embedded within large language models. **The final limitation** is that existing work typically considers only either vulnerability code (Fu et al., 2023; Wen et al., 2024) or description (Na et al., 2017; Aota et al., 2020; Shuai et al., 2013). However, in practice, when analyzing vulnerability information, both modalities should be taken into account. Therefore, based on the prompt tuning paradigm, it is necessary to propose effective information fusion methods for constructing the prompt when performing prompt tuning.

Approach. To address the aforementioned limitations, we propose an approach named *VulTypeIL*, aimed at improving the performance of software vulnerability type identification. First, to mitigate catastrophic forgetting in incremental learning, we employ Elastic Weight Consolidation (EWC) to regularize key model parameters, which allows the SVTI model to preserve essential knowledge from previous tasks while adapting to new tasks. Additionally, we propose a hybrid replay strategy that combines tail samples and high-uncertainty samples from previous tasks to both reinforce prior knowledge and facilitate incremental learning. Second, to address the long-tail distribution issue, we combine Focal Loss with Label Smooth Cross-Entropy (LSCE) Loss. The former is designed to focus more on hard-to-classify examples, particularly those from the tail of the distribution, by down-weighting the loss associated with well-classified examples. The latter helps regularize the model by softening the target labels, thereby reducing the model's overconfidence and preventing it from becoming overly biased towards certain classes. Finally, when constructing the SVTI model based on prompt tuning, we use a hybrid prompt to combine the vulnerability code and description. For the verbalizers, we adopt a one-to-three verbalizer configuration. Specifically, we retrieve the specific vulnerability type name corresponding to the CWE-IDs from the NVD database and then identify two additional synonyms (i.e., similar names) for each type.

RQ1: Can *VulTypeIL* improve the performance of software vulnerability type identification?

Results. Our proposed approach *VulTypeIL* demonstrates better performance over eight state-of-the-art baselines (such as VulExplainer (Fu et al., 2023) and LIVABLE (Wen et al., 2024)) for SVTI in the incremental learning scenario across all performance metrics.

RQ2: Can our customized regularization and replay strategies in *VulTypeIL* achieve better performance in the incremental learning scenario for software vulnerability type identification?

Results. The ablation study reveals that our incremental learning strategy of combining the hybrid replay strategy and the regularization with EWC achieves better performance than the incremental learning strategies that only use regularization strategies or replay strategies.

RQ3: Can our customized loss function in *VulTypeIL* improve the performance of software vulnerability type identification by alleviating the long-tail distribution issue?

Results. The ablation study reveals that our customized loss function of combining Focal Loss and LSCE Loss performs better than other loss functions.

RQ4: Can fusing vulnerability code and descriptions in *VulTypeIL* improve the performance of software vulnerability type identification?

Results. By fusing vulnerability code and descriptions, our approach outperforms the control approaches, which rely solely on vulnerability code or description.

RQ5: Can performing prompt tuning on the pre-trained model CodeT5 achieve the best performance of software vulnerability type identification for *VulTypeIL*?

Results. Our approach *VulTypeIL* using CodeT5 achieves an MCC of 70%, which is 7.69%–536.36% higher than using other pre-trained models (i.e., T5 Raffel et al., 2020, CodeBERT Feng et al., 2020, GraphCodeBERT Guo et al., 2020 and UnixCoder Guo et al., 2022).

Based on the above empirical results, our proposed approach enables SVTI models to continuously learn from new data while retaining previously acquired knowledge, effectively mitigating the catastrophic forgetting issue. Additionally, our approach addresses the issue of long-tail distribution, ensuring improved performance across both head and tail classes. Due to the dynamic and long-tail characteristics of the data, which are unique to the SVTI task, we call on researchers to propose more effective solutions from these perspectives.

To the best of our knowledge, the contributions of our study can be summarized as follows.

- **Scenario.** In the context of SVTI, new vulnerabilities and vulnerability types are continuously emerging, making it essential for models to not only learn new patterns but also retain knowledge of previously identified vulnerabilities. Thus, we identify this incremental learning scenario for SVTI. In our experiments, we split the SVTI dataset into different tasks according to the commit time of the vulnerabilities to simulate this scenario.
- **Approach.** We introduce *VulTypeIL*, a prompt-tuning-based approach for SVTI that incorporates incremental learning to mitigate catastrophic forgetting, integrating the hybrid replay strategy and the regulation strategy with EWC. Then we employ Focal Loss and LSCE Loss to tackle the long-tailed issue. Finally, we leverage both types of vulnerability information (i.e., vulnerability code and descriptions) to construct hybrid prompts to enhance SVTI performance.
- **Dataset.** We construct a large-scale SVTI dataset using real-world vulnerabilities from the CVE database. This dataset contains 6269 vulnerabilities from 992 real-world projects, with a focus on C/C++ vulnerabilities.

- **Experiments.** We conduct comprehensive experiments to evaluate the effectiveness of *VulTypeIL*. The results demonstrate that *VulTypeIL* consistently outperforms four state-of-the-art SVTI baselines, particularly in terms of MCC. Furthermore, ablation studies validate the design rationale of customized component settings within *VulTypeIL*.

Open Science. To support the open science community, we share data, models, scripts, and detailed experimental results in our GitHub repository(https://github.com/xjcway123/VulTypeIL)

2. Background and research motivation

In this section, we first show the background of software vulnerability type identification and incremental learning. Then we analyze the research motivation of our study.

2.1. Software vulnerability type identification

Software vulnerabilities are common in software projects and are a major cause of security flaws and cyberattacks (Liu et al., 2024). As a result, identifying and fixing these vulnerabilities is both critical and challenging. Since SVTI is a key first step in the repair process, identifying vulnerabilities by their specific types enables developers to concentrate on more targeted and effective remediation efforts.

The Common Weakness Enumeration (CWE) is a comprehensive, community-driven repository of software and hardware vulnerabilities. It provides a hierarchical framework for categorizing weaknesses, where each CWE-ID represents a specific vulnerability type, and abstract CWE categories group related weaknesses into broader classifications. This structure offers standardized terminology for discussing vulnerabilities, enabling security analysts to more effectively identify and understand flaws in software systems.

Recently, line-level software vulnerability detection methods (Fu and Tantithamthavorn, 2022; Hin et al., 2022) have been introduced to identify specific lines of code likely to be vulnerable. These methods significantly reduce the manual effort required to analyze large codebases. However, they often lack interpretability, as they do not map detected vulnerabilities to specific CWE-IDs. To address these limitations, software vulnerability type identification has emerged as a complementary approach. Unlike line-level detection, SVTI classifies vulnerable code into specific CWE-IDs, providing clear and actionable explanations for identified vulnerabilities (Wang et al., 2020). This capability is crucial in cybersecurity, where the accurate identification and classification of vulnerabilities are essential for preventing exploitation.

2.2. Incremental learning

To address the dynamic nature of the real world, intelligent systems must continually acquire, update, accumulate, and utilize knowledge throughout their lifespan (Wang et al., 2024). This capability, known as incremental learning, serves as a foundation for the adaptive development of artificial intelligence systems. However, incremental learning is often hindered by catastrophic forgetting (McCloskey and Cohen, 1989; Li et al., 2019), where learning a new task can lead to significant performance degradation on previously learned tasks. Artificial neural networks excel at solving classification problems for specific, rigid tasks by acquiring knowledge through separate training phases, resulting in a static body of knowledge (De Lange et al., 2021). However, when attempting to expand this knowledge without optimizing for the original task, learning subsequent tasks often leads to catastrophic forgetting, where the performance of earlier tasks deteriorates as the model adapts to new tasks.

According to a recent survey (De Lange et al., 2021), existing research on incremental learning can generally be categorized into three main approaches: replay strategies (Bagus and Gepperth, 2021), regularization strategies (Yin et al., 2020), and parameter isolation strategies (Yin et al., 2020). However, parameter isolation methods incur significant computational and storage overhead, which escalates considerably as the number of tasks increases. As a result, we exclude parameter isolation methods from the scope of our study.

Replay strategies. Replay-based strategies (Bagus and Gepperth, 2021) address catastrophic forgetting by storing a limited set of exemplars from previous datasets, which are periodically used to retrain the model. These strategies have demonstrated promising results in NLP tasks (Han et al., 2020). For example, the confidence-based replay method (Xue et al., 2025) selects exemplars based on the confidence of the model. However, a key challenge is selecting high-quality exemplars, as relying on random selection may not effectively capture the most informative examples necessary for model retention.

Regularization strategies. In contrast, regularization-based strategies (Yin et al., 2020) aim to preserve knowledge from previous datasets by constraining the model's parameters to prevent drastic changes. These regularization strategies include Elastic Weight Consolidation (EWC) (Kirkpatrick et al., 2017), Synaptic Intelligence (SI) (Zenke et al., 2017), and Riemannian Walk (RWalk) (Chaudhry et al., 2018). Compared to replay-based methods, regularization-based strategies are generally more computationally efficient and can serve as a complementary strategy to enhance model stability. This is especially valuable, as the constraints in replay-based strategies may be insufficient due to the limited size of exemplars available for replay.

2.3. Research motivation

In this subsection, we present the research motivation by analyzing the limitations of previous studies.

Limitation 1: Lack of Adaptability to the Dynamic Characteristics of Software Vulnerabilities. Existing approaches (Fu et al., 2023; Wen et al., 2024) rely on static datasets to construct SVTI models. However, the landscape of software security is dynamic, with new vulnerabilities being reported regularly. As a result, models built using static datasets cannot adapt to this evolving nature of vulnerabilities, and we refer to this issue as the dynamic characteristics of SVTI data. A natural solution is to use incremental learning (Wang et al., 2024), which can adapt to new vulnerabilities over time without needing to retrain from scratch. However, incremental learning introduces a significant challenge known as the catastrophic forgetting issue (Nguyen et al., 2019). This occurs when learning new knowledge causes the model to forget previously acquired knowledge. As introduced in Section 2.2, two strategies are employed to address this: the replay strategy and the regularization strategy. The former involves storing a subset of previous exemplars and reintroducing them during training to ensure that the model retains knowledge of past vulnerabilities. The latter adds penalties to the model's weights to prevent drastic changes, thereby helping to preserve important knowledge acquired from previous data. In this study, we aim to improve and combine these strategies based on the characteristics of the SVTI task, ensuring that the model not only adapts to new vulnerabilities but also retains the ability to accurately classify previously encountered vulnerabilities.

Limitation 2: Lack of Effectively Handling Long-tailed Distribution Issue. In the collected SVTI data, a long-tailed distribution issue exists (Zhou et al., 2023). Specifically, some vulnerability types have numerous samples, which we refer to as head classes, while others have fewer samples, which we refer to as tail classes. Due to the limited number of samples in tail classes, the model's identifications for these vulnerabilities are often unsatisfactory. This is particularly concerning when the tail classes are high-risk. For example, CWE-415, shown in Fig. 1, is a vulnerability sample from the tail class, yet its CVSS score of 9.8 indicates a critical severity. If such vulnerabilities are not accurately identified, it can result in severe consequences. Therefore, in the incremental learning scenario, it is essential to consider the long-tailed distribution issue when constructing SVTI models and to design

CWE-IDs: CWE-415

Source Code:

```
void DFcleanup()
{
  struct nlist *VAR_0, *VAR_1;
  for (VAR_2=0; VAR_2 <VAR_3; VAR_2++)
  {
    for (VAR_0 = VAR_4[VAR_2]; VAR_0; VAR_0=VAR_1)
    {
      VAR_1=VAR_0- >next;
      free(VAR_0- >name);
      free(VAR_0);
    }
    VAR_4[VAR_2] = 0;
  }
}
```

Vulnerability Description:
An issue was discovered in the sys-info crate before 0.8.0 for Rust. sys_info::disk_info calls can trigger a double free.

CVSS scores: 9.8 CRITICAL

Fig. 1. A vulnerability example in the tail classes poses a critical risk, as indicated by its CVSS score.

an effective loss function to improve the identification accuracy for tail classes.

Limitation 3: Lack of Efficiency when Training from Scratch Paradigm or the Fine-tuning Paradigm. Existing research primarily relies on the training-from-scratch paradigm (Wen et al., 2024) or the fine-tuning paradigm (Fu and Tantithamthavorn, 2022). However, these paradigms have several drawbacks, including high computational costs and the risk of overfitting, particularly when the amount of labeled data is limited. To overcome these issues, we attempt to build the SVTI model using prompt tuning, which offers several advantages, such as lower computational costs and reduced memory requirements compared to fine-tuning. Prompt tuning also allows for faster adaptation to new tasks with minimal changes to the pre-trained model. For the SVTI problem, when performing prompt tuning, we also need to design new prompts and verbalizers specifically tailored to the characteristics of this task.

Limitation 4: Lack of Effectively Fusing the Vulnerability Code and Description. Previous studies in constructing SVTI models have primarily focused on a single input source. For example, some researchers have considered vulnerability code (Wen et al., 2024; Fu et al., 2023; Wang et al., 2020), while others have focused on vulnerability descriptions (Na et al., 2017; Aota et al., 2020; Shuai et al., 2013). These two sources provide different perspectives on the same vulnerability. Therefore, it is important to analyze whether these two input sources have a certain complementarity in the context of SVTI model construction. If such complementarity exists, we need to design effective information fusion methods to further enhance model performance, particularly in the incremental learning scenario.

3. Our proposed approach *VulTypeIL*

The general framework of our proposed approach *VulTypeIL* is shown in Fig. 2. In this figure, *VulTypeIL* includes four main modules. In particular, **Data Preprocessing Module** preprocesses vulnerability code and descriptions to alleviate the input limitations and split the SVTI dataset chronologically to simulate the Incremental learning scenario. **Hybrid Replay Module** aims to select the tail samples and the high-uncertainty samples for replaying. **Regularization Module** incorporates a regulation strategy with EWC and introduces a novel loss function (i.e., Focal Loss combined with LSCE Loss) to address overfitting and alleviate the challenges posed by long-tail distribution. Note that these two modules form the core part of incremental learning. **Prompt Tuning Module** focuses on performing prompt tuning on the pre-trained model CodeT5 with the customized prompt and verbalizer for SVTI. In the rest of this section, we show the details of each module.

3.1. Data preprocessing module

We preprocess vulnerability code, specifically, we apply code simplification, removing elements such as blank lines, comments, and leading whitespace. This preprocessing ensures that the input length is effectively reduced, while still preserving the original semantic information as much as possible.

Specifically, the original SVTI dataset is split into N tasks, with each subset further divided into training, validation, and test sets in an 80%:10%:10% ratio. The datasets of these N tasks are represented as $D_{1:N} = \{\{T_1, V_1, S_1\}, \{T_2, V_2, S_2\}, \ldots, \{T_N, V_N, S_N\}\}$, where T_i, V_i, and S_i denote the training set, the validation set, and the test set for the ith task, respectively. The model is trained on T_i and validated on V_i. After completing training on the ith dataset, the model is evaluated on the test sets of all the current and previous tasks, $\{S_1, S_2, \ldots, S_i\}$, to assess its ability to retain knowledge from earlier tasks while adapting to the new task. This setup provides a robust evaluation framework for the SVTI model's capacity to balance knowledge retention and adaptability over time.

3.2. Hybrid replay module

To prevent the catastrophic forgetting of prior knowledge and address the long-tail distribution issue, we design a hybrid replay strategy to retrain the SVTI model. Our hybrid replay strategy contains two key components. The first component involves replaying samples in tail classes, and the second involves replaying high-uncertainty samples. By ensuring that samples in the tail vulnerabilities and uncertain samples play a central role in training, this strategy effectively mitigates the long-tail distribution problem while improving the SVTI model's performance across all vulnerability types.

Algorithm 1 Hybrid Replay Strategy

Input: Dataset from previous tasks D; Threshold for tail classes τ; The size of D N; Replay numbers of the high-uncertainty samples ρ;
Output: Replay samples R.

1: $R \leftarrow \emptyset$
2: $D_{uncertain} \leftarrow \emptyset$
3: Compute class frequencies $Count(c)$ for each class c in D
4: Identify tail classes TC: $TC = \{c \mid Count(c)/|D| < \tau\}$
5: $R_{tail} \leftarrow$selected samples belong to TC
6: $R \leftarrow R \cup R_{tail}$
7: **for** $i \leftarrow 1$ **to** N **do**
8: Compute the mean vector μ and covariance matrix Σ of feature vectors of D_i
9: Compute Mahalanobis distance d_M
10: $D_{uncertain} \leftarrow D_{uncertain} \cup$ {Sample with calculated Mahalanobis distance}
11: **end for**
12: Sort $D_{uncertain}$ by d_M in descending order
13: $R_{uncertain} \leftarrow$ Select top ρ samples from $D_{uncertain}$
14: $R \leftarrow R \cup R_{uncertain}$
15: **return** R

The details of the hybrid replay strategy are shown in Algorithm 1, which contains two main components. In the first component (Lines 2–5), this strategy focuses on the tail classes. For the SVTI dataset, the majority of samples belong to a few dominant vulnerability types, while tail classes, with fewer samples, are often neglected during training. By prioritizing the replay of tail classes, we ensure that the model

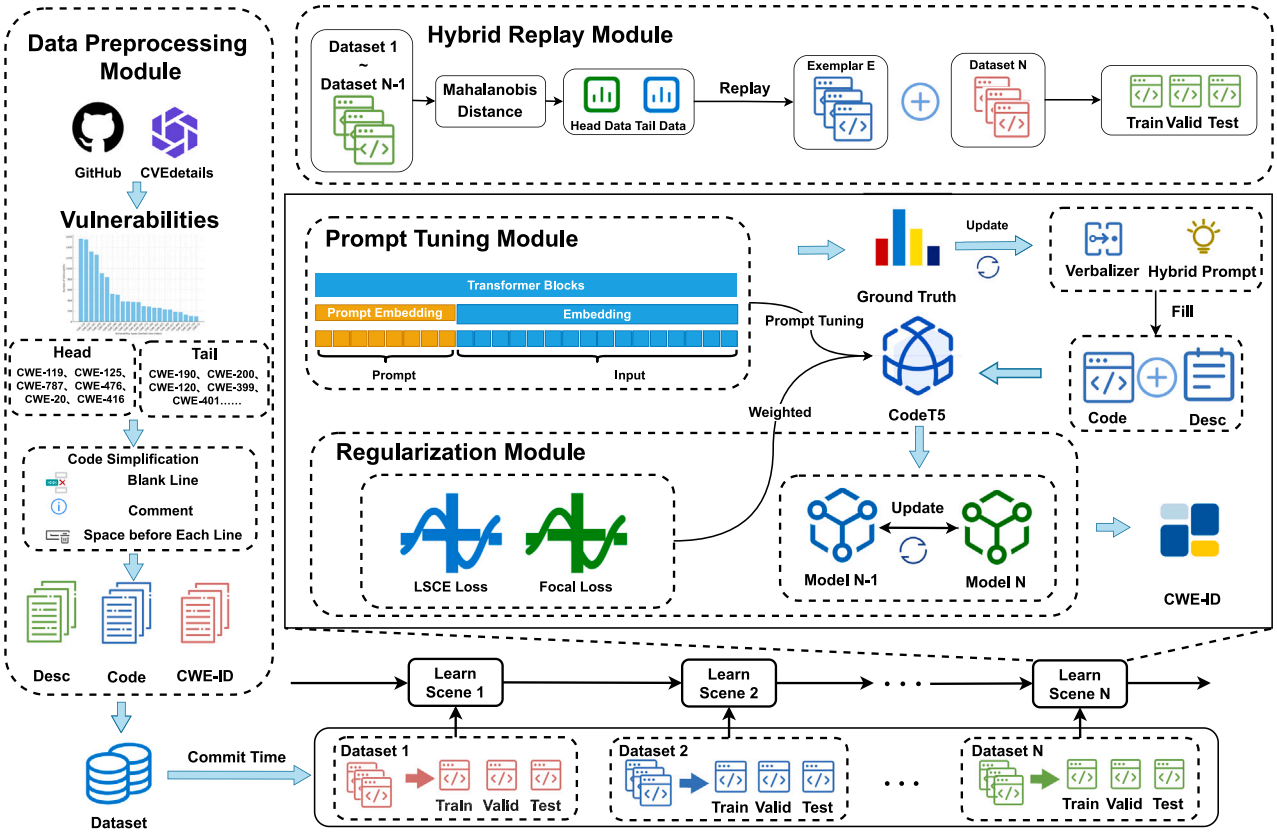

Fig. 2. Framework of our proposed approach *VulTypeIL*.

continuously focuses on these tail classes during training, reducing the likelihood of forgetting them over time. To identify tail classes, we set a threshold τ, such that classes representing less than 5% of the total samples in the dataset, and classify them as tail classes. Specifically, a tail class TC can be defined as follows:

$$TC = \{class\ c \mid \frac{Count(c)}{TS} < \tau\} \tag{1}$$

where τ is set as 5% in our study, TS represents total samples.

The second component of the hybrid replay strategy (Lines 6–13) focuses on high-uncertainty samples, which are the samples that the model finds most challenging to predict. These samples are typically located near decision boundaries, where predictions are less confident. By prioritizing the replay of high-uncertainty samples, the SVTI model strengthens its understanding of ambiguous regions within the feature space.

To identify high-uncertainty samples, we use the Mahalanobis distance (De Maesschalck et al., 2000) to measure uncertainty. This metric quantifies the divergence of each sample's feature vector from the mean of all feature vectors, effectively highlighting the most uncertain and challenging samples for replay. Specifically, in the first step, we extract feature representations, i.e., obtaining the feature vector x_i for each sample. In our implementation, we extract x_i from the penultimate layer of the SVTI model, which corresponds to the final hidden representation before the classification layer. This layer is chosen because it encodes rich semantic information that effectively captures the model's internal understanding of the input, making it suitable for measuring uncertainty and performing exemplar selection.

In the second step, we calculate the mean feature vector μ and the covariance matrix Σ for all sample features. In the third step, for each sample, we compute its Mahalanobis distance $d_{M(x_i)}$ as follows:

$$d_M(x_i) = \sqrt{(x_i - \mu)^T \Sigma^{-1} (x_i - \mu)} \tag{2}$$

where x_i denotes the feature vector of the sample i, μ denotes the mean feature vector, and Σ^{-1} denotes the inverse of the covariance matrix.

Finally, we select replay samples by combining tail-class samples and high-uncertainty samples. In our study, we select the top ρ samples from the sample set for which the Mahalanobis distance has been calculated. This ensures that the SVTI model retains knowledge of tail-class samples while boosting its confidence by learning from high-uncertainty samples.

3.3. Regularization module

As previously analyzed (Wen et al., 2024), software vulnerability type identification data often exhibits a long-tail distribution, where tail classes consist of many vulnerability types but each type has very few samples. This imbalance makes it difficult for models to accurately learn vulnerability information from these tail vulnerability types. To address this issue, we develop a two-stage training framework that incorporates a tailored loss function and regularization strategy. In the first training stage, our approach combines Focal Loss and LSCE Loss. The details of this stage are shown as follows.

Focal Loss introduces a scaling factor to standard CE Loss, enabling the model to focus more on hard-to-classify samples, which are often tail samples. It achieves this by applying a focusing parameter γ, which increases the weight of misclassified samples, and a balancing factor α to address the class imbalance. This loss function is defined as:

$$\mathcal{L}_{FL} = -\sum_{i=1}^{n} (1 - \hat{y}_i)^{\gamma} y_i \log(\hat{y}_i) \tag{3}$$

where y_i and $\hat{y}_i$ denote the label distribution in the ground truth and the predicted output, respectively. γ is the modulating factor for focusing on misclassified samples.

LSCE Loss, on the other hand, softens the target labels, reducing overconfidence in predictions and improving generalization, especially for rare classes. This loss function is defined as:

$$\mathcal{L}_{LSCE} = -\sum_{i=1}^{n} \log\left(\hat{y}_i\right)\left((1-\epsilon)y_i + \epsilon\delta_i\right) \tag{4}$$

where ϵ denotes a smoothing parameter, and δ_i denotes the uniform distribution to smooth the ground-truth distribution y_i.

By combining these two loss functions, we propose a composite loss that leads to better handling of tail classes and enhanced performance on long-tail distributed data.

$$\mathcal{L} = \omega \times \mathcal{L}_{FL} + (1-\omega) \times \mathcal{L}_{LSCE} \tag{5}$$

where the parameter ω is used to balance the contributions of Focal Loss and LSCE Loss.

In addition, due to the dynamic characteristics of SVTI data, models tend to forget previously learned knowledge over time during incremental learning. This can further impact their ability to recognize rare vulnerabilities with limited samples. To mitigate this issue, we incorporate the regularization strategy EWC (Kirkpatrick et al., 2017) in the second training stage. EWC introduces a regularization term that prevents model parameters from deviating too far from those considered important for previous tasks, thereby retaining previously learned knowledge. Traditional EWC requires storing the Fisher matrix and parameters for each task, which increases computational complexity as the number of tasks grows.

To address this, we improve the Fisher information matrix calculation by using cumulative and weighted updates, reducing storage and computational costs while enhancing scalability. Specifically, first, we calculate the Fisher matrix, let F_t represent the Fisher matrix for the current task t, and $\tilde{F}$ the accumulated Fisher matrix. After each task, $\tilde{F}$ is updated as follows:

$$\tilde{F} = \beta \times \tilde{F} + (1-\beta) \times F_t \tag{6}$$

where β denotes a decay factor that assigns a higher weight to recent tasks while retaining knowledge from earlier ones.

Using the accumulated Fisher matrix, the regularization term for EWC is defined as:

$$\mathcal{L}_{EWC} = \frac{\lambda}{2}\sum_{i}\tilde{F}_i(\theta_i - \theta_i^*)^2 \tag{7}$$

where λ is the penalty coefficient, $\tilde{F}_i$ is the ith element of the accumulated Fisher matrix, θ_i is the current parameter, and θ_i^* is the parameter value learned from previous tasks.

3.4. Prompt tuning module

Instead of using the training-from-scratch or fine-tuning paradigms, we aim to construct the SVTI model using the prompt tuning paradigm. Prompt tuning (Liu et al., 2023) utilizes prompt templates to transform the original input into a format that is more suitable for the model to understand and predict, guiding the model's predictions. However, designing effective prompt templates tailored to specific downstream tasks remains challenging (Wang et al., 2022; Yang et al., 2024; Wang et al., 2025a). Based on the characteristics of prompt tokens, prompt templates can be broadly classified into three types: hard prompts, soft prompts, and hybrid prompts. Specifically, hard prompts are manually crafted based on natural language prompts, while soft prompts are templates constructed using trainable vectors, which are initialized with vector representations derived from the corresponding hard prompt. In contrast to the above two types, hybrid prompts combine hard and soft prompts, allowing for more flexible expression. For SVTI, we use the hybrid prompt template and define it as:

$$\begin{aligned} f_{hybrid} = &The\ code\ snippet: [X] \\ &The\ vulnerability\ description: [Y]\ [SOFT]\ [Z] \end{aligned} \tag{8}$$

Here, we use "code snippet" and "vulnerability description" as hard prompts, providing explicit guidance for the model to distinguish between vulnerability code and description. In contrast, the instruction "Identify the vulnerability type:" is represented as a soft prompt, denoted by [SOFT], as it allows for alternative phrasings such as "differentiate the vulnerability type" or "the vulnerability type of this vulnerability is". By combining hard and soft prompts, the hybrid prompt template ensures that the model can effectively differentiate between the two input modalities (i.e., code and description) while maintaining flexibility in how it processes the task instruction. This design allows the model to leverage the structured nature of the code and the contextual information provided by the description, leading to more accurate vulnerability type identification. The effectiveness of using the hybrid prompt is supported by our ablation study, as shown in Section 6.1.

The verbalizer (Hu et al., 2021) maps predictions at the `[MASK]` position by linking specific words from the vocabulary to predefined classification labels. `[MASK]` represents that the model needs to predict the label for the vulnerability type based on the provided code snippet and vulnerability description. Each label selects a set of words to best represent its semantic meaning. The word with the highest probability at the `[MASK]` position is then mapped to the corresponding target class label through the verbalizer, allowing the model to make its final prediction based on the highest probability word associated with each vulnerability type. One-to-one verbalizer may cause the model to be too dependent on these specific descriptions, thus limiting its generalization ability.

To solve this issue, we define a one-to-many verbalizer where each CWE-ID corresponds to a specific software vulnerability type, derived from the CVE database. This mapping is set in the verbalizer to capture subtle semantic differences among different vulnerability types, enabling the model to more accurately classify and distinguish between similar vulnerabilities. We retrieve the specific vulnerability type corresponding to the CWE-IDs from the NVD database, and then identify related synonyms for each type based on ChatGPT. For example, when handling CWE-125, which represents an 'Out-of-bounds Read', the instruction provided to ChatGPT would be as follows:

> "Based on the core concepts of vulnerabilities, please generate two alternative expressions for CWE-125 'Out-of-bounds Read'".

In response, ChatGPT generates synonymous expressions such as "Memory Access Violation" and "Read Beyond Boundaries". These alternative expressions are used to construct a mapping that helps the model capture subtle semantic differences between different vulnerability types. The verbalizer we define for this task is shown as follows:

$$Verbalizer = \begin{cases} \text{CWE-125:["Out-of-bounds Read",} \\ \text{"Memory Access Violation",} \\ \text{"Read Beyond Boundaries"]} \\ \\ \text{CWE-787:["Out-of-bounds Write",} \\ \text{"Buffer Overflow",} \\ \text{"Memory Corruption"]} \\ \\ \text{CWE-476:["NULL Pointer Dereference",} \\ \text{"Access to Null Pointer",} \\ \text{"Dereferencing Null"]} \\ \ldots \\ \text{CWE-189:["Numeric Error",} \\ \text{"Numerical Miscalculation",} \\ \text{"Mathematical Error"]} \end{cases} \tag{9}$$

Finally, we leverage the hybrid prompt template to distinguish between the vulnerability code and descriptions. Then, we perform prompt tuning on the pre-trained model CodeT5 (Wang et al., 2021) to train both the SVTI model and the prompt template.

4. Experimental setup

4.1. Research questions

To evaluate the effectiveness of our proposed approach *VulTypeIL* and the rationality of the customized component settings, we design the following five research questions (RQs).

RQ1: Can *VulTypeIL* improve the performance of software vulnerability type identification?

Motivation. In RQ1, we aim to investigate whether resorting to incremental learning allows SVTI models to adapt to the dynamic characteristics of vulnerability data over time, effectively retaining previous knowledge while incorporating new knowledge. By further addressing the challenges posed by long-tailed distribution, we aim to evaluate whether *VulTypeIL* can outperform baselines in terms of performance in incremental learning scenarios, particularly for tail classes.

RQ2: Can our customized regularization and replay strategies in *VulTypeIL* achieve better performance in the incremental learning scenario for software vulnerability type identification?

Motivation. In RQ2, we aim to evaluate the effectiveness of the component settings in continual learning through ablation studies. Specifically, we assess the performance of the hybrid replay method by comparing it to different regularization methods and replay methods, respectively.

RQ3: Can our customized loss function in *VulTypeIL* improve the performance of software vulnerability type identification by alleviating the long-tail distribution issue?

Motivation. In RQ3, we aim to investigate whether the combination of loss functions can improve the accuracy of vulnerability identification, particularly for underrepresented tail classes, compared to other individual loss functions.

RQ4: Can fusing vulnerability code and descriptions in *VulTypeIL* improve the performance of software vulnerability type identification?

Motivation. In RQ4, we aim to evaluate whether fusing vulnerability code and descriptions improves the performance of VulTypeIL. Additionally, we will conduct experiments using vulnerability code alone and vulnerability descriptions alone to identify which input type most effectively improves the performance of VulTypeIL.

RQ5: Can performing prompt tuning on the pre-trained model CodeT5 achieve the best performance of software vulnerability type identification for *VulTypeIL*?

Motivation. In RQ5, we aim to assess whether leveraging the pre-trained model CodeT5 (Wang et al., 2021) enhances the performance of VulTypeIL compared to other pre-trained models, including T5 (Raffel et al., 2020), CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), and UnixCoder (Guo et al., 2022).

4.2. Experimental subjects

In previous research (Wen et al., 2024; Fu et al., 2023), the dataset commonly used for studying SVTI tasks is Big-Vul (Fan et al., 2020). However, Big-Vul contains only 3754 vulnerabilities, and it was collected in 2020, which means it does not include the latest vulnerabilities. To address this limitation, we aim to construct a new SVTI dataset that includes a larger number of vulnerabilities, encompasses a broader range of projects, and features more up-to-date vulnerabilities.

To build the SVTI dataset, we begin by retrieving all publicly available CVE entries from the National Vulnerability Database (NVD) in JSON format, which include CVE descriptions, CVSS scores, CWE identifiers, and reference links. Unlike previous studies that directly used specific repositories from Big-Vul, we manually curate a list of 28 widely used Git-based code hosting platforms referenced by these CVEs, such as GitHub, GitLab, and Bitbucket, ensuring broader project coverage and more current codebases.

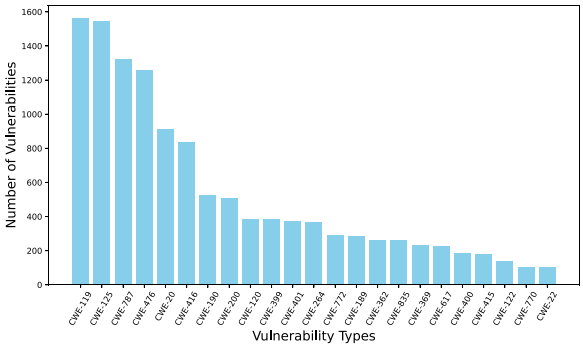

Fig. 3. Statistics of CWE-IDs distribution in our constructed SVTI dataset.

Since most CVE references do not directly point to commits, we adopt a multi-step construction process: First, we analyze issue tracker events, comments, and pull request discussions linked to CVE references to extract candidate commit URLs potentially fixing the vulnerabilities. Second, we cross-check commit messages and metadata with CVE and CWE identifiers through keyword matching and regex patterns to increase mapping confidence. Third, for commits referencing multiple CVEs or affecting multiple functions, we conduct spot manual validations to confirm relevance. Such commits are included with all associated CWE labels, and we apply further heuristics to disambiguate function-level vulnerability labels when possible.

For each identified vulnerability-fixing commit, we download the modified files and their parent versions, focusing exclusively on C and C++ source files and headers to reduce noise. We then employ the Tree-sitter parser,[4] enhanced with additional rules to handle macro definitions commonly found in open-source projects, to extract individual functions before and after the commit. To avoid parsing interference, macros are temporarily removed before parsing. The pre-commit version of each modified function is labeled as vulnerable, while the post-commit version is labeled as non-vulnerable. This function-level labeling assumes that vulnerabilities are localized within the modified functions, an assumption validated via spot checks. When commits modify multiple functions, each modified function inherits the commit's CWE labels. We apply heuristics to exclude unrelated modifications, such as formatting-only changes, by analyzing semantic diff sizes.

To improve data quality, we apply a series of filtering steps: deduplication to remove duplicate functions across commits and projects; anomaly filtering to discard functions with extreme sizes or parsing failures; test-related filtering to exclude test code identified by filenames and comments; and label consistency checks to prevent contradictory labels on the same function using hashing and source location tracking. Additionally, CWE categories with insufficient samples are excluded to avoid bias and support effective training and evaluation.

To maintain statistical robustness, we exclude CWE-ID categories with insufficient samples that could hinder effective training or lead to biased evaluation. The final dataset comprises 6269 vulnerabilities from 992 open-source projects, covering 23 unique vulnerability types. As shown in Fig. 3, CWE-119, CWE-125, CWE-787, CWE-476, CWE-20, and CWE-416 are categorized as head classes, each constituting more than 5% of the total samples, while the rest fall into tail classes.

4.3. Baselines

To demonstrate the effectiveness of our proposed approach *VulTypeIL*, we select the following eight state-of-the-art baselines. VulExplainer (Fu and Tantithamthavorn, 2022) and LIVABLE (Wen et al.,

[4] https://tree-sitter.github.io/tree-sitter/.

2024) are recently proposed classic approaches for SVTI. Devign (Zhou et al., 2019) and ReGVD (Nguyen et al., 2022) are two classical approaches designed for vulnerability detection. In our study, we adapted these two baselines from binary classification to multi-class classification for the SVTI task. LFME (Xiang et al., 2020) and BAGS (Li et al., 2020) both address the long-tailed distribution issue in their original studies. To apply these two baselines to SVTI, we use the pre-trained model CodeT5 to generate code representation for each vulnerability code. For all selected baselines, we conduct experiments using our constructed SVTI dataset, which is divided into five tasks in chronological order, and train the model on each task respectively for comparison in the incremental learning scenario. In the rest of this subsection, we briefly introduce the characteristics of these eight baselines.

- **VulExplainer.** Fu et al. (2023) introduced VulExplainer, a Transformer-based hierarchical distillation approach for SVTI. VulExplainer frames the task as vulnerability classification and addresses the challenge of highly imbalanced software vulnerability types.
- **LIVABLE.** Wen et al. (2024) proposed LIVABLE, a method for long-tailed vulnerability classification. The vulnerability representation learning module enhances representations through differentiated propagation and a sequence-to-sequence model, while the adaptive re-weighting module adjusts learning weights based on training epochs and sample distribution to address the long-tailed distribution issue.
- **Devign.** Zhou et al. (2019) proposed a GNN-based approach Devign, that utilizes a graph embedding layer to generate multiple representations (e.g., abstract syntax trees, control flow graphs, and data flow graphs) by leveraging composite code semantics.
- **ReGVD.** Nguyen et al. (2022) proposed ReGVD, applying graph convolutional network (GCN) layers with pooling to generate a graph embedding, which is used to predict the final targets.
- **LFME.** Xiang et al. (2020) proposed the Learn from Multiple Experts (LFME) model, dividing the imbalanced label distribution into several groups, each with a more balanced distribution, and training an expert model for each group.
- **BAGS.** Li et al. (2020) proposed a balanced training strategy (BAGS), where an imbalanced dataset is grouped into more balanced subsets. A shared CNN model extracts features, while multiple classification heads, each tailored to a specific group, are trained.

4.4. Performance metrics

In this subsection, we introduce the evaluation metrics used to evaluate the performance of the SVTI task, including accuracy, precision, recall, F1 score, and MCC. These metrics rely on the classification outcomes: True Positive (TP) indicates the number of samples where the model correctly identifies the positive class, accurately assigning the correct severity level. True Negative (TN) denotes cases where the model accurately identifies the negative class, correctly recognizing samples that do not belong to a specific severity level. False Negative (FN) refers to samples where the model fails to classify a positive case, incorrectly excluding samples that should belong to a severity level. Conversely, False Positive (FP) represents cases where the model misclassifies a negative class as positive, mistakenly assigning samples to a specific severity level.

Then we show the details of these evaluation metrics. Here we define $Accuracy_i$, $Precision_i$, $Recall_i$, $F1\text{-}score_i$, MCC_i as the performance on the test set of the ith task, given the SVTI model trained on the data of the last task.

Accuracy: Accuracy is the proportion of correctly classified samples to all samples.

$$Accuracy_i = \frac{TP + TN}{TP + FP + TN + FN} \tag{10}$$

Precision: Precision is the proportion of relevant samples among those retrieved.

$$Precision_i = \frac{TP}{TP + FP} \tag{11}$$

Recall: Recall is the proportion of relevant samples retrieved.

$$Recall_i = \frac{TP}{TP + FN} \tag{12}$$

F1 score: The F1 score is the harmonic mean of Precision and Recall, representing a balance between these two metrics. The macro average F1 score is the average of the F1 scores across categories and is defined as follows.

$$F1\text{-}score_i = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{13}$$

MCC: MCC considers TP, TN, FP, and FN and is effective for datasets with the class imbalance problem. For the multi-class classification problem, we calculate the macro average of each class's MCC as follows.

$$MCC_i = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{14}$$

Finally, we present the process of computing the metric values in the incremental learning scenario. Assume that the SVTI data is divided into N tasks in chronological order. We then train the SVTI model on the last task using our proposed approach. Afterward, we compute the metric values on the test sets of all tasks based on this model and return the average value. For example, the MCC metric is computed as follows:

$$MCC = \frac{1}{N} \sum_{j=1}^{N} MCC_i \tag{15}$$

where MCC_i represents the MCC value on the test set of the jth task. Notice that the above metrics are all designed for multi-classification tasks. The calculation for other performance metrics follows a similar procedure. In the following result analysis for RQs, we denote Accuracy as ACC, Precision as P, Recall as R, and F1 score as F1 for convenience.

4.5. Experimental settings

To simulate the scenario of incremental learning, we split our constructed SVTI dataset into five tasks according to the commit time of the vulnerabilities. The detailed procedure can be found in Section 3.1.

We implement our proposed approach by using the PyTorch and Transformers libraries, and baselines by using the PyTorch library. In the prompt tuning module, we utilized the OpenPrompt library by following the tutorial provided by Ding et al. (2021). Specifically, we employed the $ManualTemplate$, $SoftTemplate$, and $MixedTemplate$ APIs to implement hard, soft, and hybrid prompt templates, respectively. Then these templates were used with $PromptForGeneration$ API for prompt tuning. Detailed implementation details are available in our GitHub repository.

In the incremental learning part, for the hybrid replay strategy, we set the replay size to 20% of the training set. The replayed rare samples and uncertainty samples each account for half, thus ρ (which determines the replayed high-uncertainty samples) is 10% of the training set. For the regularization strategy with EWC, we set the EWC weight λ to 0.4. Finally, we balance the weights of the Focal Loss and LSCE Loss by setting ω to 0.5. The rationale behind the setting of these hyperparameter values is discussed in Section 6.4. We utilized the official repository of CodeT5 for our study. To ensure stable convergence during fine-tuning, we employed the Adam optimizer with an initial learning rate of 5e−5. The batch size was set to 16, striking a balance between computational efficiency and gradient stability.

We run our proposed approach and baselines on a computer with a 3.50 GHz CPU and a GeForce RTX 4090 GPU with 24 GB of graphics memory. The running operating system is Windows 10.

Table 1
Performance comparison of different approaches of software vulnerability type identification.

SVTI approach	Acc	P	R	F1	MCC
Devign	0.16	0.07	0.07	0.09	0.08
ReGVD	0.13	0.08	0.05	0.09	0.08
LFME	0.26	0.26	0.20	0.21	0.19
BAGS	0.16	0.16	0.13	0.11	0.10
VulExplainer-CodeBERT	0.21	0.17	0.17	0.16	0.21
VulExplainer-GraphCodeBERT	0.24	0.22	0.18	0.19	0.16
VulExplainer-CodeGPT	0.20	0.19	0.17	0.17	0.13
LIVABLE	0.20	0.17	0.20	0.15	0.14
VulTypeIL	**0.73**	**0.59**	**0.63**	**0.58**	**0.70**

5. Result analysis

5.1. RQ1: Comparison with baselines

Approach. To evaluate the effectiveness of our proposed approach *VulTypeIL*, we compare it with eight state-of-the-art baselines (i.e., Devign Zhou et al., 2019, ReGVD Nguyen et al., 2022, LFME Xiang et al., 2020, BAGS Li et al., 2020, VulExplainer Fu et al., 2023, and LIVABLE Wen et al., 2024). In this RQ, according to the selected pre-trained model, VulExplainer has three variants: VulExplainer-CodeBERT, VulExplainer-GraphCodeBERT, and VulExplainer-CodeGPT. These baselines represent different types of approaches to vulnerability type identification, ensuring a comprehensive comparison. To simulate a realistic incremental learning scenario, we partition our gathered SVTI dataset into five tasks in chronological order, each representing a distinct data subset. This setup allows us to evaluate how the baselines and *VulTypeIL* adapt to new tasks in a sequential manner. We conduct training and testing for each split dataset, computing performance metrics (such as F1 score and MCC) across all approaches and tasks.

Results. The detailed comparison results of our proposed approach *VulTypeIL* with the baselines are shown in Table 1. In this table, we find that *VulTypeIL* outperforms eight state-of-the-art baselines across all evaluation metrics. In particular, *VulTypeIL* achieves an accuracy of 73%, showing an improvement of 180.77% to 461.54% compared to the baselines. For precision, *VulTypeIL* reaches 59%, demonstrating an increase of 126.92% to 742.86% over the baselines. Similarly, *VulTypeIL* achieves a recall score of 63%, representing an improvement of 215.00% to 1160.00%. The F1 score of *VulTypeIL* is 58%, reflecting an enhancement of 176.19% to 544.44% compared to the baselines. Finally, *VulTypeIL* achieves an MCC of 70%, outperforming the baselines with an improvement of 233.33% to 775.00%. These results demonstrate the effectiveness of *VulTypeIL* in the incremental learning setting for vulnerability type identification.

The lower performance of the baselines can be attributed to the following reasons. First, without continuous learning techniques, these models struggle to adapt to evolving data, limiting their ability to generalize to emerging vulnerability patterns. Second, they fail to integrate up-to-date domain knowledge and evolving vulnerability trends, diminishing their overall effectiveness. Lastly, static models are more susceptible to overfitting or underfitting, especially when trained on limited or outdated data, which adversely affects their predictive accuracy.

To assess whether the performance differences between *VulTypeIL* and the baselines are statistically significant, we perform the Wilcoxon signed-rank test (Wilcoxon, 1992) with a 95% confidence level. The resulting p-values, calculated across five different performance metrics, are all below 0.05. These findings indicate that our proposed approach achieves significantly better results than the baselines.

Last, we evaluated the performance of *VulTypeIL* and eight state-of-the-art baselines specifically on rare CWE types. As shown in Table 2, *VulTypeIL* achieves an accuracy of 71%, showing an improvement of 273.68% to 3450.00% compared to the baselines. For precision, *VulTypeIL* reaches 55%, demonstrating an increase of 323.08% to 685.71% over the baselines. Similarly, *VulTypeIL* achieves a recall score of 61%, representing an improvement of 510.00% to 6000.00%. The F1 score of *VulTypeIL* is 51%, reflecting an enhancement of 466.67% to 2450.00% compared to the baselines. Finally, *VulTypeIL* achieves an MCC of 68%, outperforming the baselines with an improvement of 580.00% to 3300.00%.

Table 2
Performance comparison of different approaches on rare CWE types.

SVTI approach	Acc	P	R	F1	MCC
Devign	0.02	0.07	0.01	0.02	0.02
ReGVD	0.04	0.08	0.03	0.04	0.03
LFME	0.05	0.10	0.04	0.05	0.05
BAGS	0.11	0.13	0.07	0.08	0.08
VulExplainer-CodeBERT	0.06	0.09	0.04	0.05	0.04
VulExplainer-GraphCodeBERT	0.07	0.10	0.05	0.06	0.06
VulExplainer-CodeGPT	0.12	0.10	0.07	0.06	0.08
LIVABLE	0.19	0.09	0.10	0.09	0.10
VulTypeIL	**0.71**	**0.55**	**0.61**	**0.51**	**0.68**

Table 3
Performance comparison of different regularization strategies.

Strategy	Acc	Prec	Rec	F1	MCC
VulTypeIL with SI	0.64	0.55	0.61	0.54	0.62
VulTypeIL with RWalk	0.65	0.53	0.62	0.53	0.62
VulTypeIL with EWC	**0.73**	**0.59**	**0.63**	**0.58**	**0.70**

Answer to RQ1

Our proposed approach *VulTypeIL* demonstrates better performance in vulnerability type identification compared to the baselines across all metrics. Notably, it achieves improvements of 233.33% to 775.00% in MCC. These results highlight the effectiveness of *VulTypeIL* in addressing the dynamic characteristics and the long-tailed distribution issues.

5.2. RQ2: Ablation study on incremental learning settings

Approach. To evaluate the effectiveness of our incremental learning settings in *VulTypeIL*, we consider the following two ablation studies. Specifically, we first investigate the performance impact of three regularization strategies: RWalk (Chaudhry et al., 2018), SI (Zenke et al., 2017), and EWC (Kirkpatrick et al., 2017). We secondly investigate the performance impact of three replay strategies: random replay (Random), confidence-based replay (Confidence), and hybrid replay (Hybrid).

Results. The comparison results of using different regularization strategies are presented in Table 3. Based on the comparison results, EWC shows significant improvements across all evaluation metrics compared to both SI and RWalk. Specifically, compared to SI, the F1-score is increased by 7.41%, and the MCC is improved by 12.90%. In comparison to RWalk, the F1-score is increased by 9.43%, while the MCC shows an increase of 12.90%.

EWC's significant improvements across various metrics can be attributed to its unique regularization mechanism, which alleviates catastrophic forgetting and preserves knowledge from previous tasks. By introducing an additional penalty term, EWC constrains updates to important parameters, preventing negative interference from new tasks. The method EWC enables a better stability-plasticity trade-off, allowing it to avoid catastrophic forgetting. In contrast, methods like SI and RWalk do not effectively constrain parameter updates, making them more prone to forgetting previously learned tasks. While these methods perform reasonably well in some metrics, their stability and generalization are weaker, especially in long-term learning.

The comparison results of using different replay strategies are shown in Table 4. Based on the comparison results, the hybrid replay strategy demonstrates significant improvements across all evaluation metrics compared to random replay and confidence-based replay.

Table 4
Performance comparison of using different replay strategies.

Replay strategy	Acc	P	R	F1	MCC
VulTypeIL with random	0.68	0.58	0.62	0.55	0.66
VulTypeIL with confidence	0.71	0.58	0.62	0.56	0.68
VulTypeIL with hybrid	**0.73**	**0.59**	**0.63**	**0.58**	**0.70**

Specifically, the F1-score increases by 5.45% and MCC improves by 6.06% compared to random replay. When compared to confidence-based replay, the F1-score improves by 3.57%, while MCC increases by 2.94%.

Random replay introduces diversity by selecting a wide range of samples, which helps stabilize the training process. However, it may overlook more informative or challenging examples that are difficult for the model to identify. In contrast, confidence-based replay targets uncertain samples where the model has low confidence, thereby improving the model's ability to generalize in these challenging areas. However, both random and confidence-based replay strategies fail to address the long-tailed distribution issue in the SVTI data. Hybrid replay strategy, on the other hand, effectively compensates for the limitations of random and confidence-based replay strategies by accounting for both samples in the tail classes and uncertain samples.

Answer to RQ2

Our ablation study results demonstrate the effectiveness of the hybrid replay strategy combined with the regularization strategy using EWC in *VulTypeIL*, which outperforms other incremental learning strategies for SVTI.

5.3. RQ3: Ablation study on loss function setting

Approach. In the real world, SVTI data often exhibit a long-tailed distribution issue (Zhang et al., 2023), where a small number of vulnerability types contain the majority of the samples, while the remaining vulnerability types are associated with very few samples. To address the challenges posed by this distribution, our proposed approach VulTypeIL combines Focal Loss and LSCE Loss by leveraging the strengths of both. Specifically, Focal Loss (Ross and Dollár, 2017) helps the model focus more on hard-to-classify samples, particularly those from minority classes, by down-weighting well-classified examples, typically from the majority class. Meanwhile, LSCE Loss (Szegedy et al., 2016) reduces the model's overconfidence in the majority class by smoothing the label distribution, encouraging the model to be less confident about the head classes. This combination can enhance the model's ability to learn from rare classes. To evaluate the effectiveness of our approach *VulTypeIL*, which combines Focal Loss and LSCE Loss (FL+LSCE), we conduct ablation studies by considering different loss function settings in this RQ. These loss functions include Label Aware Smooth Loss (LASL), Focal Loss (FL), LSCE Loss (LSCE), and the combination of Focal Loss and Label Aware Smooth Loss (FL+LASL). Here, Label Aware Smooth Loss (Zhong et al., 2021) captures the predicted probability distributions over different classes.

Results. The ablation study results on different loss function settings are shown in Table 5. Based on the comparison results, we find that the combination of Focal Loss and LSCE Loss, as utilized in *VulTypeIL*, achieves the best performance across all metrics when compared to other loss functions. Specifically, compared to other loss function settings, the F1 score is improved by 1.75% to 9.43%, and MCC is increased by 1.45% to 4.48%.

Label Aware Smooth Loss reduces overfitting by smoothing labels but lacks the focus on uncertain samples, making it less effective in alleviating the long-tailed distribution issue. Focal Loss, on the other hand, excels at focusing on tail classes but has limited label smoothing capabilities, which may cause the model to overlook class distribution differences. LSCE Loss reduces overfitting but lacks Focal Loss's focus on uncertain samples, hindering learning tail class effectively. Combining Focal Loss with Label Aware Smooth Loss balances focus and smoothing, but still lacks the additional smoothing effect of LSCE Loss, limiting its effectiveness for tail classes. In contrast, the combination of Focal Loss and LSCE Loss achieves a synergistic effect: Focal Loss targets uncertain samples, while LSCE Loss reduces overfitting to the head class, resulting in better performance for the SVTI data with the long-tailed distribution issue.

Table 5
Performance comparison of *VulTypeIL* using different loss functions to mitigate the long-tailed distribution issue.

Loss setting	Acc	P	R	F1	MCC
VulTypeIL with LSCE	0.72	0.58	0.60	0.55	0.69
VulTypeIL with LASL	0.71	**0.59**	**0.63**	0.57	0.68
VulTypeIL with FL	0.71	0.56	0.60	0.53	0.68
VulTypeIL with FL+LASL	0.70	**0.59**	0.61	0.55	0.67
VulTypeIL with FL+LSCE	**0.73**	**0.59**	**0.63**	**0.58**	**0.70**

Table 6
Performance comparison of *VulTypeIL* using different input configurations of vulnerability information.

Input configuration	Acc	P	R	F1	MCC
VulTypeIL with code	0.13	0.03	0.08	0.03	0.05
VulTypeIL with desc	0.67	0.52	0.56	0.50	0.65
VulTypeIL with both	**0.73**	**0.59**	**0.63**	**0.58**	**0.70**

Answer to RQ3

Our proposed approach *VulTypeIL*, which combines Focal Loss and LSCE Loss, demonstrates better performance compared to other loss functions in alleviating the long-tailed distribution issue.

5.4. RQ4: Ablation study on information fusion

Approach. To show the competitiveness of fusing source code and vulnerability descriptions on the performance of *VulTypeIL*, we conduct a comprehensive ablation study. In this study, we investigate three distinct input configurations: source code only, vulnerability descriptions only, and a fusion of both source code and descriptions. Each configuration was evaluated under the same experimental settings to ensure a fair comparison.

Results. The ablation study results can be found in Table 6. Based on the comparison results, we find that by fusing both code and description information, *VulTypeIL* can outperform models that are trained by using only code or description across all evaluation metrics. Specifically, in terms of accuracy, *VulTypeIL* achieves a 461.54% improvement over the code-only configuration and an 8.96% improvement over the description-only configuration. For precision, *VulTypeIL* outperforms the code-only configuration by 1866.67% and the description-only configuration by 13.46%. In recall, *VulTypeIL* achieves a 687.50% increase over the code-only configuration and a 12.50% increase over the description-only configuration. The F1 score for *VulTypeIL* is 1833.33% higher than when using code alone and 16.00% higher than when using description alone. Regarding MCC, *VulTypeIL* improves by 1300.00% over the code-only configuration and 7.69% over the description-only configuration. These results demonstrate that fusing both code and description information leads to substantial improvements in all performance metrics, highlighting the robustness and potential of *VulTypeIL* for software vulnerability type identification in the incremental learning scenario.

In the context of prompt-based learning for *VulTypeIL*, vulnerability descriptions offer critical context and insights that may not be easily extracted from the source code alone. Natural language descriptions often explicitly highlight the nature and potential impact of vulnerabilities, enabling the model to better understand and assess their

Table 7
Comparison of the performance among different pre-trained models.

Pre-trained model	Acc	P	R	F1	MCC
VulTypeIL with T5	0.19	0.12	0.12	0.09	0.11
VulTypeIL with CodeBERT	0.63	0.54	0.61	0.53	0.61
VulTypeIL with GraphCodeBERT	0.65	0.56	0.62	0.54	0.63
VulTypeIL with UnixCoder	0.67	0.54	0.62	0.55	0.65
VulTypeIL with CodeT5	**0.73**	**0.59**	**0.63**	**0.58**	**0.70**

severity. By integrating both source code and vulnerability descriptions, *VulTypeIL* takes advantage of the complementary strengths of both modalities: source code provides syntactic details, while vulnerability descriptions offer semantic context. This combination leads to a more holistic understanding of vulnerabilities, enhancing the model's ability to predict their severity and improving overall assessment accuracy.

Summary for RQ4

By fusing source code and vulnerability descriptions, *VulTypeIL* enhances its ability to understand and predict vulnerability types. The source code provides crucial syntactic details, while the descriptions offer important semantic context, together enabling more accurate identification of software vulnerability types.

5.5. RQ5: Ablation study on pre-trained models

Approach. To evaluate the impact of different pre-trained models on the performance of *VulTypeIL*, we further consider four pre-trained code models: T5 (Raffel et al., 2020), CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), UnixCoder (Guo et al., 2022) as the foundational models for our approach. Specifically, T5 (Raffel et al., 2020) treats all NLP tasks as a unified "text-to-text" problem, where both the input and output are represented in text form. Whether it is classification, generation, question answering, or translation, all tasks are reformulated as text generation tasks. CodBERT (Feng et al., 2020) uses a bidirectional Transformer architecture and extends BERT's Masked Language Modeling task to handle the unique structure of programming languages. GraphCodeBERT (Guo et al., 2020) is a pre-trained model optimized for program understanding and generation tasks, combining the strengths of source code's syntactic structure and graph neural networks (GNNs). By incorporating structural information such as Abstract Syntax Trees (AST) and Control Flow Graphs (CFG), it captures the syntax and control flow features of code more effectively. UnixCoder (Guo et al., 2022) is a pre-trained model specifically designed for code understanding and generation tasks, focusing on command-line code in Unix operating system environments. By pre-training on a large-scale dataset of Unix commands and scripts, the model effectively captures the syntax, semantics, and execution logic of commands.

Results. The comparison results of using different pre-trained models can be found in Table 7. These comparison results demonstrate that *VulTypeIL*, using CodeT5, can outperform the other pre-trained models across most evaluation metrics. Specifically, compared to UnixCoder (i.e., the second-best model), *VulTypeIL* achieves an improvement of 8.96% in accuracy, 9.26% in precision, 1.61% in recall, 5.45% in F1 score, and 7.69% in MCC. These results validate the effectiveness of using CodeT5 as the backbone for *VulTypeIL*, emphasizing its ability to comprehensively capture the semantic information of vulnerability code and description more effectively than other pre-trained models.

The differences in model performance may stem from their unique designs and pre-training objectives. Specifically, T5, optimized for general NLP tasks with text-to-text transformations, cannot capture code's syntactic and semantic nuances, making it less effective for code-related tasks. CodeBERT, pre-trained on source code with masked language modeling, captures syntax and semantics but lacks structural insights such as Abstract Syntax Trees. GraphCodeBERT addresses this by integrating graph-based representations like ASTs and Control Flow Graphs, enhancing its understanding of code structure and context. UnixCoder uses a UniLM-style design to support multiple tasks with input attention masks, relying on a single encoder, which hampers its performance on the SVTI task. In contrast, CodeT5's specialized design for handling both code and description semantics gives it a clear advantage for the SVTI task.

Table 8
Performance comparison of *VulTypeIL* using the prompt-tuning paradigm or the fine-tuning paradigm.

Paradigm	Acc	P	R	F1	MCC
VulTypeIL with fine-tuning	0.31	0.24	0.25	0.21	0.25
VulTypeIL with prompt tuning	**0.73**	**0.59**	**0.63**	**0.58**	**0.70**

Summary for RQ5

VulTypeIL utilizing CodeT5 outperforms the use of other pre-trained models, with performance improvements of 7.69% in MCC and 5.45% in F1 score compared to the next-best model. These results highlight CodeT5's effectiveness in capturing semantic information from vulnerability code and description, making it the most suitable choice for software vulnerability type identification.

6. Discussions

In this section, we first demonstrate the effectiveness of the prompt-tuning paradigm by comparing it with the fine-tuning paradigm. Second, we analyze the performance impact of different verbalizer configurations. Third, we investigate the influence of key hyperparameters' value setting. Fourth, we demonstrate the generalization ability of our proposed approach on the public dataset Big-Vul. Finally, we discuss the potential threats to the validity of our empirical study.

6.1. Comparison with fine-tuning paradigm

In this subsection, we aim to demonstrate the effectiveness of prompt tuning for *VulTypeIL*. To achieve this, we compare our approach, which leverages the prompt tuning paradigm, with our approach based on the fine-tuning paradigm. For fine-tuning, we use the same pre-trained model CodeT5 (Wang et al., 2021). In this control approach, we freeze the bias terms and LayerNorm weights of the model, while updating the other parameters by following previous studies (Hu et al., 2022; Pfeiffer et al., 2020; Zaken et al., 2021), which can help reduce computational overhead and mitigate the risk of overfitting. The Adafactor optimizer (Shazeer and Stern, 2018) is used to optimize the trainable parameters.

The comparison results are shown in Table 8. The results show that the prompt-tuning paradigm can outperform the fine-tuning paradigm across all evaluation metrics. Specifically, accuracy increases by 135.48%, precision by 145.83%, recall by 152.00%, and the F1-score achieves a remarkable improvement of 176.19%. The largest gain is observed in MCC, which is increased by 180.00%. Moreover, in terms of model training time, the prompt-tuning paradigm can reduce the training time cost by approximately 50% compared to the fine-tuning paradigm. These improvements highlight the effectiveness of the prompt-tuning paradigm in fully utilizing the inherent knowledge of the pre-trained model CodeT5 for *VulTypeIL*.

6.2. Comparison with different prompt templates

In this subsection, we aim to discuss why we use a hybrid prompt in our proposed approach. To achieve this goal, we consider three types of prompt templates: hard prompts, soft prompts, and hybrid prompts, each offering varying levels of flexibility and expressiveness. Specifically, hard prompts use fixed natural language patterns to

Table 9
Performance comparison of *VulTypeIL* using different prompt templates.

Prompt template	Acc	P	R	F1	MCC
VulTypeIL with hard prompt	0.69	0.56	0.59	0.53	0.66
VulTypeIL with soft prompt	0.69	0.58	0.61	0.54	0.67
VulTypeIL with hybrid prompt	**0.73**	**0.59**	**0.63**	**0.58**	**0.70**

ensure interpretability and alignment with domain-specific contexts, specifying placeholders for inputs and predictions in a clear, structured manner. Specifically, the hard prompt template for SVTI in this experiment can be designed as:

$$\begin{aligned} f_{hard} = &\, The\ vulnerability\ code : [X] \\ &\, The\ vulnerability\ description : [Y] \\ &\, Identify\ the\ vulnerability\ type : [Z] \end{aligned} \tag{16}$$

where the input slot $[X]$ is used for the vulnerability code, and the input slot $[Y]$ is reserved for the vulnerability description. The pre-trained model then predicts the probability distribution over label words at position $[Z]$. The label word with the highest probability is selected as the intermediate generated answer by the model.

In contrast, soft prompts replace static text, such as "The vulnerability code is:" and "The vulnerability description is:", with trainable embeddings. This dynamic design allows the model to flexibly adapt to training data. For SVTI, the soft prompt template can be designed as:

$$f_{soft} = [SOFT]\,[X]\,[SOFT]\,[Y]\,[SOFT]\,[Z] \tag{17}$$

In our study, the [SOFT] tokens are assigned specific vector representations derived from the natural language phrases in the hard prompt templates. For example, the first [SOFT] token is initialized as "The vulnerability code:", the second [SOFT] token as "The vulnerability description:", and the third [SOFT] token as "Identify the vulnerability type:".

The comparison results in Table 9 demonstrate that *VulTypeIL* with the hybrid prompt consistently outperforms both the hard prompt and soft prompt across all metrics. Compared to the hard prompt, the hybrid prompt shows a relative improvement of 5.80% in accuracy, 5.36% in precision, 6.78% in recall, 9.43% in F1 score, and 6.06% in MCC. Similarly, compared to the soft prompt, the hybrid prompt achieves a relative improvement of 5.80% in accuracy, 1.72% in precision, 3.28% in recall, 7.41% in F1 score, and 4.48% in MCC, while recall shows a slight increase. These results highlight the effectiveness of the hybrid prompt in combining the strengths of hard and soft prompts, leading to superior performance in software vulnerability type identification.

6.3. Comparison with different verbalizer settings

Each vulnerability type ID typically has a corresponding vulnerability type name. For example, the vulnerability type ID "CWE-125" corresponds to the name "Out-of-bounds Read". In our study, we find that when setting the verbalizer, considering multiple names during the prediction of IDs can effectively improve the performance of *VulTypeIL*. In our experiments, we observe that considering the additional two names most similar to the corresponding name for each ID helps our proposed approach achieve the best performance. We refer to this as the "one-to-three verbalizer configuration".

To validate the effectiveness of this setting, we additionally explore two other configurations. The first is the "one-to-one verbalizer configuration", where each ID is associated with only one name. The second is the "one-to-five verbalizer configuration", where each ID considers the additional four names most similar to the corresponding name. Note that we use ChatGPT-4 to determine the names most similar to the specified name.

Table 10 presents the performance of different verbalizer configurations on *VulTypeIL*. Increasing the number of candidate vulnerability type names from 1 to 3 leads to significant performance improvements across all metrics: a 23.73% increase in accuracy, 13.46% in precision, 10.53% in recall, 13.73% in F1 score, and 25.00% in MCC. However, further increasing the number of candidate names to 5 slightly reduces accuracy, precision, F1 score, and MCC, likely due to the introduction of irrelevant names. Notice, we only present a subset of the verbalizers in the table, and details of different Verbalizer configurations are available on our project GitHub.

6.4. Analysis on the hyperparameter influence

In this subsection, we analyze the influence of key hyperparameters' value setting on *VulTypeIL*. Specifically, we examine the hyperparameter ω, which determines the weight used to combine Focal Loss and LSCE Loss, the hyperparameter λ, which controls the weight of EWC, and the hyperparameter rs, which determines the replay size in the hybrid replay strategy.

First, we analyzed the impact of the hyperparameter ω on performance. Specifically, we considered the following values for ω: 0.1, 0.3, 0.5, 0.7, and 0.9. Table 11 presents the performance when considering these different values. As ω varies from 0.1 to 0.9, the best performance is achieved when $\omega = 0.5$, where all metric values reach their optimal levels. This suggests that a balanced integration of Focal Loss and LSCE Loss effectively leverages the complementary strengths of both loss functions.

Second, we analyze the impact of the hyperparameter λ, which controls the weight of EWC, on performance. The values of λ we considered are 0.1, 0.2, 0.3, 0.4, and 0.5. As shown in Table 12, *VulTypeIL* achieves the best performance when $\lambda = 0.4$, striking an optimal balance between regularization and flexibility.

Third, we examine the impact of the hyperparameter rs in the hybrid replay strategy, which is set as 5%, 10%, 20%, and 25% of the total number of data samples for each task. Table 13 shows how different values of rs affect performance. A smaller value (such as $rs = 5\%$) results in lower performance, indicating that using a limited amount of historical data cannot effectively alleviate the catastrophic forgetting issue. When $rs = 10\%$, there is a slight performance improvement, with accuracy increasing to 0.72 and MCC increasing to 0.69. The best performance is achieved when $rs = 20\%$. However, further increasing rs to 25% leads to a decline in performance, suggesting that excessive replay of historical data may introduce noise. Therefore, in our work, we set the value of rs to 20%.

6.5. Generalization of our approach

To further assess the generalization of our proposed approach, we conducted additional experiments on the public dataset Big-Vul (Fan et al., 2020).

The detailed comparison results of our proposed approach *VulTypeIL* against the baselines on the public dataset Big-Vul are presented in Table 14. As shown in this table, *VulTypeIL* consistently and substantially outperforms all eight state-of-the-art baselines across all evaluation metrics. Specifically, *VulTypeIL* achieves an accuracy of 72%, representing a relative improvement ranging from 140.00% to 380.00% over the baselines. For precision, *VulTypeIL* reaches 57%, yielding a relative gain of 200.00% to 850.00%. In terms of recall, it achieves 61%, which corresponds to a relative improvement of 205.00% to 1120.00%. The F1 score of *VulTypeIL* is 55%, reflecting an increase of 223.53% to 1000.00%. Lastly, *VulTypeIL* attains an MCC of 69%, showing a significant relative enhancement of 245.00% to 1625.00% compared to the baselines. These results further demonstrate the effectiveness of *VulTypeIL* in the context of software vulnerability type identification under the incremental learning setting. Finally, after performing the Wilcoxon signed-rank test (Wilcoxon, 1992) with a 95% confidence level, we find that the resulting p-values, calculated across five different performance metrics, are all below 0.05. This indicates that our proposed approach achieves significantly better results than the baselines.

Table 10
Performance of *VulTypeIL* with different verbalizer settings.

Verbalizer setting	Acc	P	R	F1	MCC
"CWE-125": ["Out-of-bounds Read"] "CWE-787": ["Out-of-bounds Write"] "CWE-476": ["NULL Pointer Dereference"] ..... "CWE-189": ["Out-of-bounds Read"]	0.59	0.52	0.57	0.51	0.56
"CWE-125": ["Out-of-bounds Read", "Memory Access Violation", "Read Beyond Boundaries"] "CWE-787": ["Out-of-bounds Write", "Buffer Overflow", "Memory Corruption"] "CWE-476": ["NULL Pointer Dereference", "Access to Null Pointer", "Dereferencing Null"] ..... "CWE-189": ["Numeric Error", "Numerical Miscalculation", "Mathematical Error"]	**0.73**	**0.59**	**0.63**	**0.58**	**0.70**
"CWE-125": ["Out-of-bounds Read", "Memory Access Violation", "Read Beyond Boundaries", "Buffer Access Error", "Memory Corruption"] "CWE-787": ["Out-of-bounds Write", "Buffer Overflow", "Memory Corruption", "Write Beyond Boundaries", "Uncontrolled Memory Access"] "CWE-476": ["NULL Pointer Dereference", "Access to Null Pointer", "Dereferencing Null", "Null Pointer Access", "Uninitialized Pointer"] ..... "CWE-189": ["Numeric Error", "Numerical Miscalculation", "Mathematical Error" "Precision Loss", "Floating Point Error"]	0.68	0.57	0.61	0.55	0.66

Table 11
Comparison of the performance among different values of ω, which determines the weight used to combine Focal Loss and LSCE Loss.

Value of ω	Acc	P	R	F1	MCC
$\omega = 0.1$	**0.73**	**0.59**	0.61	0.57	**0.70**
$\omega = 0.3$	0.70	**0.59**	0.62	0.56	0.68
$\omega = 0.7$	0.71	0.57	0.62	0.54	0.68
$\omega = 0.9$	0.71	0.57	0.60	0.54	0.68
VulTypeIL ($\omega = 0.5$)	**0.73**	**0.59**	**0.63**	**0.58**	**0.70**

Table 12
Comparison of the performance among different values of λ, which controls the weight of EWC.

Value of λ	Acc	P	R	F1	MCC
$\lambda = 0.1$	0.71	0.58	**0.63**	0.56	0.68
$\lambda = 0.2$	0.72	**0.59**	0.61	0.56	**0.70**
$\lambda = 0.3$	0.71	0.58	0.62	0.56	0.68
$\lambda = 0.5$	0.72	0.56	0.59	0.53	0.69
VulTypeIL ($\lambda = 0.4$)	**0.73**	**0.59**	**0.63**	**0.58**	**0.70**

Table 13
Comparison of the performance among different values of rs, which determines the replay size in the hybrid replay strategy.

Value of rs	Acc	P	R	F1	MCC
$rs = 5\%$	0.70	0.57	0.61	0.55	0.68
$rs = 10\%$	0.72	0.56	0.61	0.54	0.69
$rs = 25\%$	0.69	**0.59**	**0.63**	0.56	0.67
VulTypeIL ($rs = 20\%$)	**0.73**	**0.59**	**0.63**	**0.58**	**0.70**

Table 14
Performance comparison of different approaches of software vulnerability type identification on the public dataset Big-Vul.

SVTI approach	Acc	P	R	F1	MCC
Devign	0.16	0.12	0.13	0.12	0.13
ReGVD	0.17	0.17	0.16	0.16	0.14
LFME	0.30	0.07	0.07	0.06	0.05
BAGS	0.19	0.09	0.07	0.07	0.06
VulExplainer-CodeBERT	0.15	0.06	0.05	0.05	0.05
VulExplainer-GraphCodeBERT	0.17	0.07	0.06	0.06	0.05
VulExplainer-CodeGPT	0.19	0.07	0.05	0.05	0.04
LIVABLE	0.28	0.19	0.20	0.17	0.20
VulTypeIL	**0.72**	**0.57**	**0.61**	**0.55**	**0.69**

6.6. Threats to validity

Threats to internal validity. The first internal threat is related to the setting of hyperparameters. Given the substantial computational cost associated with hyperparameter tuning, we primarily relied on suggestions from previous studies and our empirical results when determining the settings. Based on our experimental results, we find that the current hyperparameter settings can still outperform the baselines. In future work, we plan to explore more optimal hyperparameter values further to enhance the performance of *VulTypeIL*. The second internal threat pertains to the implementation of our proposed approach *VulTypeIL* and the baselines. To ensure the correctness of the implementations, we followed the descriptions in the corresponding studies and performed code inspections and software testing. The third internal threat relates to the experimental setup of the baselines. To ensure a fair comparison, we run these baselines in the incremental learning scenario. The fourth internal threat is that vulnerability descriptions may contain type-description-like comments, which could potentially introduce bias. However, after examining the dataset we used, we found such type-description comments in only two vulnerability descriptions, and neither of these vulnerabilities was included in the test set. Therefore, we think this threat can be ignored in our study.

Threats to external validity. An external threat concerns the representativeness of the experimental subjects. To address this, we constructed a large-scale SVTI dataset by considering up-to-date vulnerabilities, with a particular focus on C/C++ vulnerabilities. Compared to the previously more commonly used BigVul dataset, our dataset includes more vulnerabilities from a wider range of projects.

Threats to construct validity. A construct threat relates to the selection of performance metrics. To address this concern, we carefully chose a comprehensive set of metrics. Our evaluation includes widely used metrics for SVTI, such as accuracy, precision, recall, F1 score, and MCC. By adopting these metrics, we ensure a thorough comparison with state-of-the-art baselines.

7. Related work

Software vulnerability type identification (SVTI) is a critical task focused on automatically identifying vulnerability types (e.g., CWE-IDs). By identifying specific vulnerability types, developers can promptly recognize high-risk vulnerabilities and allocate resources to fix them as quickly as possible. Relying solely on manual identification by experts is time-consuming, labor-intensive, and prone to errors. For instance, among the 28,902 vulnerabilities added to the NVD database in 2023, 4113 remain uncategorized. Therefore, there is a need to design automated SVTI methods. Early research primarily focused on vulnerability

description-based approaches, while in recent years, researchers have shifted their attention to source code-based approaches.

Vulnerability description-based approaches. These approaches leverage textual information from vulnerability descriptions to automate the identification of vulnerability types. For example, Shuai et al. (2013) address the challenges of high-dimensional and sparse text data by incorporating Latent Dirichlet Allocation (LDA) for topic modeling. They combine this with a hierarchical SVM classifier to enhance accuracy, utilizing both topic distributions and vulnerability-specific keywords. Na et al. (2017) apply a Naïve Bayes classifier to classify CVE entries using keywords extracted from the NVD description field, focusing on prominent CWE types such as CWE-119 and CWE-79. This method includes preprocessing steps like stop-word removal and phrase extraction, achieving high accuracy for the most common vulnerabilities, but it struggles as the number of classes increases.

Source code-based approaches. In recent years, source code-based vulnerability identification has emerged as a key focus in security research. This approach offers distinct advantages by directly analyzing the code to capture structural and semantic details, such as variable names, function calls, and contextual information, all of which are essential for identifying complex vulnerability types. For example, Fu et al. (2023) introduce VulExplainer, a hierarchical distillation method designed to address data imbalance. VulExplainer organizes CWE-IDs into sub-groups based on shared characteristics (e.g., 'base', 'category'), achieving more balanced distributions within each group. Knowledge from TextCNN teacher models trained on these sub-groups is distilled into a Transformer-based student model, improving classification without major architectural changes. Wen et al. (2024) propose the LIVABLE method to address the challenges of long-tail distributions in vulnerability classification. It consists of two modules: a representation learning module using Graph Neural Networks (GNNs) with refined propagation to capture code relationships while mitigating over-smoothing, and an adaptive re-weighting module that dynamically adjusts class weights to improve classification accuracy for both head and tail classes.

Novelty of Our Study. Different from previous SVTI studies, we are the first to investigate the incremental learning scenario for SVTI. To address the challenges posed by the dynamic nature of SVTI data, we employ a regularization strategy with EWC and a hybrid replay strategy to mitigate the issue of catastrophic forgetting. Additionally, we tackle the long-tailed distribution problem in SVTI data. *VulTypeIL* incorporates long-tailed learning techniques, such as combining Focal Loss and LSCE Loss, to emphasize underrepresented tail classes, thereby enhancing classification performance for rare vulnerability types. Finally, we integrate prompt tuning into SVTI by designing customized prompt templates and verbalizers tailored for this task. Our empirical results demonstrate that these component designs contribute positive impacts to improving the performance of *VulTypeIL* in the incremental learning scenario.

8. Conclusion

Since new vulnerabilities and vulnerability types are continuously emerging, we are the first to identify an incremental learning scenario for the software vulnerability type identification problem. In this scenario, we recognize that SVTI data exhibits dynamic characteristics and a long-tailed distribution issue. To address these challenges, we propose the approach *VulTypeIL*. Specifically, our approach mitigates the catastrophic forgetting issue in incremental learning by introducing a hybrid replay strategy and a regularization strategy with EWC. It then tackles the long-tailed distribution issue by integrating Focal Loss and LSCE Loss. Finally, we employ prompt tuning to train the SVTI model. During prompt tuning, we design a hybrid prompt that fuses vulnerability code and description to comprehensively learn vulnerability information. We adopt a one-to-three verbalizer configuration, where we retrieve the specific vulnerability type name corresponding to the CWE-IDs from the NVD database and identify two additional similar names for each type.

Experimental results show that *VulTypeIL* can outperform state-of-the-art SVTI baselines, achieving F1 score improvements ranging from 176.19% to 544.44% and MCC enhancements from 233.33% to 775.00%. Furthermore, our ablation study validates the effectiveness of key components, including the hybrid replay strategy, the regularization strategy with EWC, the loss function setting, vulnerability information fusion, the usage of CodeT5, and hybrid prompts.

In the future, we aim to extend our proposed approach to support vulnerability identification for other programming languages. We also want to continue to gather more up-to-date vulnerabilities to further update our trained SVTI model. Finally, we want to incorporate advanced incremental learning techniques to further improve the performance of *VulTypeIL*.

CRediT authorship contribution statement

Jiacheng Xue: Conceptualization. **Xiang Chen:** Writing – review & editing, Conceptualization. **Zhanqi Cui:** Writing – review & editing. **Yong Liu:** Writing – review & editing, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Jiacheng Xue and Xiang Chen contributed equally to this work and are co-first authors. Xiang Chen is the corresponding author. This research was partially supported by the National Natural Science Foundation of China (Grant no. 61202006), the Open Foundation of Beijing Institute of Control Engineering (Grant No. LHCESET202307), and the Open Project of State Key Laboratory for Novel Software Technology at Nanjing University under (Grant No. KFKT2024B21) and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (Grant No. SJCX25_2005).

Data availability

Data will be made available on request.

References

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

Bagus, Benedikt, Gepperth, Alexander, 2021. An investigation of replay-based approaches for continual learning. In: 2021 International Joint Conference on Neural Networks. IJCNN, IEEE, pp. 1–9.

Cai, Zhilong, Cai, Yongwei, Chen, Xiang, Lu, Guilong, Pei, Wenlong, Zhao, Junjie, 2024. CSVD-TF: Cross-project software vulnerability detection with TrAdaBoost by fusing expert metrics and semantic metrics. J. Syst. Softw. 213, 112038.

Cao, Sicong, Sun, Xiaobing, Bo, Lili, Wu, Rongxin, Li, Bin, Wu, Xiaoxue, Tao, Chuanqi, Zhang, Tao, Liu, Wei, 2023. Learning to detect memory-related vulnerabilities. ACM Trans. Softw. Eng. Methodol. 33 (2), 1–35.

Chaudhry, Arslan, Dokania, Puneet K, Ajanthan, Thalaiyasingam, Torr, Philip HS, 2018. Riemannian walk for incremental learning: Understanding forgetting and intransigence. In: Proceedings of the European Conference on Computer Vision. ECCV, pp. 532–547.

Cheng, Xiao, Wang, Haoyu, Hua, Jiayi, Xu, Guoai, Sui, Yulei, 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Trans. Softw. Eng. Methodol. (TOSEM) 30 (3), 1–33.

De Lange, Matthias, Aljundi, Rahaf, Masana, Marc, Parisot, Sarah, Jia, Xu, Leonardis, Aleš, Slabaugh, Gregory, Tuytelaars, Tinne, 2021. A continual learning survey: Defying forgetting in classification tasks. IEEE Trans. Pattern Anal. Mach. Intell. 44 (7), 3366–3385.

De Maesschalck, Roy, Jouan-Rimbaud, Delphine, Massart, Désiré L, 2000. The mahalanobis distance. Chemometr. Intell. Lab. Syst. 50 (1), 1–18.

Ding, Ning, Hu, Shengding, Zhao, Weilin, Chen, Yulin, Liu, Zhiyuan, Zheng, Hai-Tao, Sun, Maosong, 2021. Openprompt: An open-source framework for prompt-learning. arXiv preprint arXiv:2111.01998.

Fan, Jiahao, Li, Yi, Wang, Shaohua, Nguyen, Tien N., 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. pp. 508–512.

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

Fu, Michael, Nguyen, Van, Tantithamthavorn, Chakkrit Kla, Le, Trung, Phung, Dinh, 2023. VulExplainer: A transformer-based hierarchical distillation for explaining vulnerability types. IEEE Trans. Softw. Eng. (01), 1–17.

Fu, Michael, Nguyen, Van, Tantithamthavorn, Chakkrit, Phung, Dinh, Le, Trung, 2024. Vision transformer inspired automated vulnerability repair. ACM Trans. Softw. Eng. Methodol. 33 (3), 1–29.

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

Guo, Daya, Lu, Shuai, Duan, Nan, Wang, Yanlin, Zhou, Ming, Yin, Jian, 2022. Unixcoder: Unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850.

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

Han, Xu, Dai, Yi, Gao, Tianyu, Lin, Yankai, Liu, Zhiyuan, Li, Peng, Sun, Maosong, Zhou, Jie, 2020. Continual relation learning via episodic memory activation and reconsolidation. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 6429–6440.

Hin, David, Kan, Andrey, Chen, Huaming, Babar, M. Ali, 2022. LineVD: statement-level vulnerability detection using graph neural networks. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 596–607.

Hu, Shengding, Ding, Ning, Wang, Huadong, Liu, Zhiyuan, Wang, Jingang, Li, Juanzi, Wu, Wei, Sun, Maosong, 2021. Knowledgeable prompt-tuning: Incorporating knowledge into prompt verbalizer for text classification. arXiv preprint arXiv:2108.02035.

Hu, Edward J, Shen, Yelong, Wallis, Phillip, Allen-Zhu, Zeyuan, Li, Yuanzhi, Wang, Shean, Wang, Lu, Chen, Weizhu, et al., 2022. Lora: Low-rank adaptation of large language models. ICLR 1 (2), 3.

Jimmy, FNU, 2024. Cyber security vulnerabilities and remediation through cloud security tools. J. Artif. Intell. Gen. Sci. (JAIGS) ISSN: 3006-4023 2 (1), 129–171.

Kirkpatrick, James, Pascanu, Razvan, Rabinowitz, Neil, Veness, Joel, Desjardins, Guillaume, Rusu, Andrei A, Milan, Kieran, Quan, John, Ramalho, Tiago, Grabska-Barwinska, Agnieszka, et al., 2017. Overcoming catastrophic forgetting in neural networks. Proc. Natl. Acad. Sci. 114 (13), 3521–3526.

Li, Yu, Wang, Tao, Kang, Bingyi, Tang, Sheng, Wang, Chunfeng, Li, Jintao, Feng, Jiashi, 2020. Overcoming classifier imbalance for long-tail object detection with balanced group softmax. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 10991–11000.

Li, Xilai, Zhou, Yingbo, Wu, Tianfu, Socher, Richard, Xiong, Caiming, 2019. Learn to grow: A continual structure learning framework for overcoming catastrophic forgetting. In: International Conference on Machine Learning. PMLR, pp. 3925–3934.

Liu, Chongyang, Chen, Xiang, Li, Xiangwei, Xue, Yinxing, 2024. Making vulnerability prediction more practical: Prediction, categorization, and localization. Inf. Softw. Technol. 171, 107458.

Liu, Pengfei, Yuan, Weizhe, Fu, Jinlan, Jiang, Zhengbao, Hayashi, Hiroaki, Neubig, Graham, 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Comput. Surv. 55 (9), 1–35.

Lu, Guilong, Ju, Xiaolin, Chen, Xiang, Pei, Wenlong, Cai, Zhilong, 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. J. Syst. Softw. 212, 112031.

Mallick, Md Abu Imran, Nath, Rishab, 2024. Navigating the cyber security landscape: A comprehensive review of cyber-attacks, emerging trends, and recent developments. World Sci. News 190 (1), 1–69.

McCloskey, Michael, Cohen, Neal J., 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In: Psychology of Learning and Motivation. Vol. 24, Elsevier, pp. 109–165.

Na, Sarang, Kim, Taeeun, Kim, Hwankuk, 2017. A study on the classification of common vulnerabilities and exposures using naïve bayes. In: Advances on Broad-Band Wireless Computing, Communication and Applications: Proceedings of the 11th International Conference on Broad-Band Wireless Computing, Communication and Applications (BWCCA–2016) November 5–7, 2016, Korea. Springer, pp. 657–662.

Nguyen, Cuong V, Achille, Alessandro, Lam, Michael, Hassner, Tal, Mahadevan, Vijay, Soatto, Stefano, 2019. Toward understanding catastrophic forgetting in continual learning. arXiv preprint arXiv:1908.01091.

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

Pfeiffer, Jonas, Rücklé, Andreas, Poth, Clifton, Kamath, Aishwarya, Vulić, Ivan, Ruder, Sebastian, Cho, Kyunghyun, Gurevych, Iryna, 2020. Adapterhub: A framework for adapting transformers. arXiv preprint arXiv:2007.07779.

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

Ross, T-YL.P.G., Dollár, GKHP, 2017. Focal loss for dense object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 2980–2988.

Shazeer, Noam, Stern, Mitchell, 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In: International Conference on Machine Learning. PMLR, pp. 4596–4604.

Shiri Harzevili, Nima, Boaye Belle, Alvine, Wang, Junjie, Wang, Song, Jiang, Zhen Ming, Nagappan, Nachiappan, 2024. A systematic literature review on automated software vulnerability detection using machine learning. ACM Comput. Surv. 57 (3), 1–36.

Shuai, Bo, Li, Haifeng, Li, Mengjun, Zhang, Quan, Tang, Chaojing, 2013. Automatic classification for vulnerability based on machine learning. In: 2013 IEEE International Conference on Information and Automation. ICIA, IEEE, pp. 312–318.

Szegedy, Christian, Vanhoucke, Vincent, Ioffe, Sergey, Shlens, Jon, Wojna, Zbigniew, 2016. Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 2818–2826.

Wang, Jiyu, Chen, Xiang, Pei, Wenlong, Yang, Shaoyu, 2025a. Improving prompt tuning-based software vulnerability assessment by fusing source code and vulnerability description. Autom. Softw. Eng. 32 (2), 1–29.

Wang, Liping, Lu, Guilong, Chen, Xiang, Dai, Xiaofeng, Qiu, Jianlin, 2025b. SIFT: enhance the performance of vulnerability detection by incorporating structural knowledge and multi-task learning. Autom. Softw. Eng. 32 (2), 38.

Wang, Yue, Wang, Weishi, Joty, Shafiq, Hoi, Steven C.H., 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859.

Wang, Xinda, Wang, Shu, Sun, Kun, Batcheller, Archer, Jajodia, Sushil, 2020. A machine learning approach to classify security patches into vulnerability types. In: 2020 IEEE Conference on Communications and Network Security. CNS, IEEE, pp. 1–9.

Wang, Chaozheng, Yang, Yuanhang, Gao, Cuiyun, Peng, Yun, Zhang, Hongyu, Lyu, Michael R, 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 382–394.

Wang, Liyuan, Zhang, Xingxing, Su, Hang, Zhu, Jun, 2024. A comprehensive survey of continual learning: theory, method and application. IEEE Trans. Pattern Anal. Mach. Intell..

Wen, Xin-Cheng, Gao, Cuiyun, Luo, Feng, Wang, Haoyu, Li, Ge, Liao, Qing, 2024. LIVABLE: exploring long-tailed classification of software vulnerability types. IEEE Trans. Softw. Eng..

Wilcoxon, Frank, 1992. Individual comparisons by ranking methods. In: Breakthroughs in Statistics: Methodology and Distribution. Springer, pp. 196–202.

Xiang, Liuyu, Ding, Guiguang, Han, Jungong, 2020. Learning from multiple experts: Self-paced knowledge distillation for long-tailed classification. In: Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part V 16. Springer, pp. 247–263.

Xue, Jiacheng, Chen, Xiang, Wang, Jiyu, Cui, Zhanqi, 2025. Towards prompt tuning-based software vulnerability assessment with continual learning. Comput. Secur. 150, 104184.

Yang, Shaoyu, Chen, Xiang, Liu, Ke, Yang, Guang, Yu, Chi, 2024. Automatic bi-modal question title generation for stack overflow with prompt learning. Empir. Softw. Eng. 29 (3), 63.

Yin, Dong, Farajtabar, Mehrdad, Li, Ang, Levine, Nir, Mott, Alex, 2020. Optimization and generalization of regularization-based continual learning: a loss approximation viewpoint. arXiv preprint arXiv:2006.10974.

Zaken, Elad Ben, Ravfogel, Shauli, Goldberg, Yoav, 2021. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. arXiv preprint arXiv:2106.10199.

Zenke, Friedemann, Poole, Ben, Ganguli, Surya, 2017. Continual learning through synaptic intelligence. In: International Conference on Machine Learning. PMLR, pp. 3987–3995.

Zhang, Yifan, Kang, Bingyi, Hooi, Bryan, Yan, Shuicheng, Feng, Jiashi, 2023. Deep long-tailed learning: A survey. IEEE Trans. Pattern Anal. Mach. Intell. 45 (9), 10795–10816.

Zhong, Zhisheng, Cui, Jiequan, Liu, Shu, Jia, Jiaya, 2021. Improving calibration for long-tailed recognition. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 16489–16498.

Zhou, Xin, Kim, Kisub, Xu, Bowen, Liu, Jiakun, Han, DongGyun, Lo, David, 2023. The devil is in the tails: How long-tailed code distributions impact large language models. arXiv preprint arXiv:2309.03567.

Zhou, Yaqin, Liu, Shangqing, Siow, Jingkai, Du, Xiaoning, Liu, Yang, 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv. Neural Inf. Process. Syst. 32.

Jiacheng Xue is currently pursuing the Master degree at the School of Artificial Intelligence and Computer Science, Nantong University. Her research interests include software vulnerability analysis.

Xiang Chen received the B.Sc. degree in information management and systems from Xi'an Jiaotong University, China in 2002. Then he received his M.Sc., and Ph.D. degrees in computer software and theory from Nanjing University, China in 2008 and 2011 respectively. He is currently an Associate Professor at the School of Artificial Intelligence and Computer Science, Nantong University. He has authored or co-authored more than 160 papers in refereed journals or conferences, such as IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, Empirical Software Engineering, Information and Software Technology, Journal of Systems and Software, Software Testing, Verification and Reliability, Journal of Software: Evolution and Process, International Conference on Software Engineering (ICSE), International Conference on the Foundations of Software Engineering (FSE), International Symposium on Software Testing and Analysis (ISSTA), International Conference Automated Software Engineering (ASE), International Conference on Software Maintenance and Evolution (ICSME), International Conference on Program Comprehension (ICPC), and International Conference on Software Analysis, Evolution and Reengineering (SANER). His research interests include software engineering, in particular large language models for software engineering, software testing and maintenance, software repository mining, and empirical software engineering. He received two ACM SIGSOFT distinguished paper awards in ICSE 2021 and ICPC 2023. He is the editorial board member of Information and Software Technology. More information can be found at: https://xchencs.github.io/index.html.

Zhanqi Cui received the B.E. degree in software engineering and the Ph.D. degree in computer software and theory in 2005 and 2011, respectively, from Nanjing University, Nanjing. He was a visiting Ph.D. student at the University of Virginia, Virginia, from Sep. 2009 to Sep. 2010. He is a Professor at Beijing Information Science and Technology University, Beijing. His research interests include intelligent software engineering and trustworthy artificial intelligence. More information can be found at: https://zqcui.github.io.

Yong Liu received the B.Sc. and M.Sc. degrees in computer science and technology from Beijing University of Chemical Technology, China, in 2008 and 2011, respectively. Then, he received the Ph.D. degree in control science and engineering from Beijing University of Chemical Technology in 2018. He is currently with the College of Information Science and Technology at Beijing University of Chemical Technology as a professor. His research interests are mainly in software engineering, particularly in software debugging and software testing, such as source code analysis, mutation testing, and fault localization. In these areas, he has published more than 50 papers in refereed journals and conferences, such as the Journal of Systems and Software, IEEE Transactions on Reliability, Software Testing Verification and Reliability, Information Sciences, ICSME, ASE, ISSRE, QRS, SATE, and COMPSAC. He is a member of CCF in China, IEEE, and ACM. More information can be found at: https://liuyong0076.github.io/.