

Leveraging Self-Paced Learning for Software Vulnerability Detection

ZERU CHENG*, Nanjing University, China

YANJING YANG*, Nanjing University, China

HE ZHANG, Nanjing University, China

LANXIN YANG[†], Nanjing University, China

JINGHAO HU, Nanjing University, China

JINWEI XU, Nanjing University, China

BOHAN LIU, Nanjing University, China

HAIFENG SHEN, Southern Cross University, Australia

Software vulnerabilities are major risks to software systems. Recently, researchers have proposed many deep learning approaches to detect software vulnerabilities. However, their accuracy is limited in practice. One of the main causes is low-quality training data (*i.e.*, source code). To this end, we propose a new approach: **SPLVD (Self-Paced Learning for Software Vulnerability Detection)**. SPLVD dynamically selects source code for model training based on the stage of training, which simulates the human learning process progressing from easy to hard. SPLVD has a data selector that is specifically designed for the vulnerability detection task, which enables it to prioritize the learning of easy source code. Before each training epoch, SPLVD uses the data selector to recalculate the difficulty of the source code, select new training source code, and update the data selector. When evaluating SPLVD, we first use three benchmark datasets with over 239K source code in which 25K are vulnerable for standard evaluations. Experimental results demonstrate that SPLVD achieves the highest F1 of 89.2%, 68.7%, and 43.5%, respectively, outperforming the state-of-the-art approaches. Then we collect projects from OpenHarmony, a new ecosystem that has not been learned by general LLMs, to evaluate SPLVD further. SPLVD achieves the highest precision of 90.9%, demonstrating its practical effectiveness.

Replication package: <https://figshare.com/s/bef3211194fc18fe375e>.

CCS Concepts: • Security and privacy → Software security engineering.

Additional Key Words and Phrases: Software vulnerability detection, self-paced learning, large language model, CWE

ACM Reference Format:

Zeru Cheng, Yanjing Yang, He Zhang, Lanxin Yang, Jinghao Hu, Jinwei Xu, Bohan Liu, and Haifeng Shen. 2025. Leveraging Self-Paced Learning for Software Vulnerability Detection. 1, 1 (November 2025), 25 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

*Both authors contributed equally.

[†]Corresponding Author: Lanxin Yang.

Authors' Contact Information: **Zeru Cheng**, zeru_cheng@mail.nju.edu.cn, Nanjing University, Nanjing, China; **Yanjing Yang**, yj_yang@mail.nju.edu.cn, Nanjing University, Nanjing, China; **He Zhang**, hezhang@mail.nju.edu.cn, Nanjing University, Nanjing, China; **Lanxin Yang**, lxyang@mail.nju.edu.cn, Nanjing University, Nanjing, China; **Jinghao Hu**, jinghao_hu@mail.nju.edu.cn, Nanjing University, Nanjing, China; **Jinwei Xu**, jinwei_xu@mail.nju.edu.cn, Nanjing University, Nanjing, China; **Bohan Liu**, bohanliu@mail.nju.edu.cn, Nanjing University, Nanjing, China; **Haifeng Shen**, haifeng.shen@southcross.edu.au, Southern Cross University, Gold Coast, Australia.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM XXXX-XXXX/2025/11-ART

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 Introduction

Software vulnerabilities are a security issue in software systems that attackers can exploit to cause harm, including system crashes, privacy data leaks, or financial loss [5, 28, 51]. For instance, the Stuxnet worm exploited multiple vulnerabilities to attack the industrial control systems of Iran's nuclear facilities, causing damage to physical equipment and resulting in significant economic and political impacts [61]. According to statista [49], the number of newly discovered Common Vulnerabilities and Exposures (CVE [43]) has continuously risen, and the public release of these vulnerabilities has led to the formation of a vast vulnerability database [41]. To prevent security issues caused by software vulnerabilities, the vulnerability detection system plays a crucial role in software security [26].

Conducting static code analysis on the submitted code is one of the effective approaches for detecting software vulnerabilities [25, 54]. These approaches can be broadly classified into three categories [27]: (1) Rules-based approaches: These approaches are based on the vulnerability rules predefined by experts, which identify matching vulnerabilities by statically scanning the source code¹ [56]. While they can directly locate specific lines of code, they are limited to known patterns, which limits their effectiveness in identifying new vulnerabilities [2, 18]. (2) Machine/deep learning-based approaches. These approaches are based on the features extracted from the function-level source code [11, 34, 51], and convert them into vectors or graph representations and use different neural networks such as recurrent neural networks (RNNs) for classification models to automatically learn the relationship between vulnerability patterns and labels, eliminating the need for manual rules [5, 32, 33, 72]. However, these approaches have low accuracy rates and can only provide rough predictions at the file or function level [9, 48, 70]. It is difficult to determine the specific location of the vulnerability, which results in the lack of usability of these approaches in production [35, 37, 50]. (3) Fine-tuning pre-trained model-based approaches. Implementing vulnerability detection based on the pre-trained model obtained through training on large-scale source code, such as CodeBERT [15] and UniXcoder [19], has become popular. These approaches adapt them to vulnerability detection tasks through transfer learning [71]. They are capable of capturing code semantics and contextual dependencies effectively, supporting predictions such as line-level detection, which provides significant assistance for practical applications [16, 29, 53, 67]. However, they are reliant on the code patterns learned from pretrained source code, and the quality of the training data influences their performance [8, 20, 21].

Despite the various approaches for vulnerability detection that exist, they place greater emphasis on how to extract the characteristics of the source code and construct the model [3, 47], but neglect the selection of training data. Some research has shown that the vulnerability datasets are generally of low quality [13, 36]. Many vulnerability datasets suffer from issues such as inconsistent labels, duplicate data, and erroneously classified source code [10], which can lead to model bias in fine-grained predictions [24, 31]. The current approaches lack a training approach to filter out high-quality training data. The models are affected by noisy data, which limits their accuracy in practice [4, 52]. Therefore, they need an approach that can dynamically select training data to help the model prioritize the learning of high-quality data.

To this end, we propose a new approach: **SPLVD (Self-Paced Learning for Software Vulnerability Detection)**. The core idea of SPLVD is a data selector based on self-paced learning, which enables the detection model to learn the source code from easy to hard during the training process, thereby reducing the influence of low-quality source code on the model's learning in the early stage. The key of the data selector lies in defining the difficulty of the source code and dynamically selecting the training code according to the training state. Specifically, the difficulty of each source code is

¹Hereafter, source code refers to function-level source code.

calculated based on the model’s confidence in its prediction and the correctness of the prediction. The data selector then uses a difficulty threshold to select the training code, which is automatically updated based on various features according to the training state. Before each training epoch, SPLVD uses the data selector to recalculate the difficulty of each source code. Based on the updated difficulty, the selector then selects new training source code and updates the difficulty threshold accordingly, allowing the training data to be dynamically selected along with the training process of the model.

We evaluate SPLVD on four datasets (BigVul, Devign, ReVeal, and OpenHarmony). SPLVD outperforms all baselines on F1: on BigVul SPLVD achieves an F1 of 89.2%, together with 98.8% accuracy and 96.1% precision; on Devign it obtains accuracy of 74.8% and an F1 of 68.7%; and on the more challenging dataset *i.e.*, ReVeal, it still reaches an F1 of 43.5%. SPLVD achieves the highest F1 in the most common CVE categories. The ablation study indicates that incorporating self-paced learning improves model performance: the F1 on BigVul, Devign, and ReVeal increased by 2.4%, 3.6%, and 2.0%, respectively. Finally, SPLVD achieves a precision of 90.9% on OpenHarmony, demonstrating its effectiveness in real-world applications.

The main contributions of this article are as follows.

- **Approach:** A dynamic self-paced learning approach based on training state feedback.
- **Model:** A vulnerability detection model that is trained using self-paced learning.

The remainder of this article is organized as follows. Section 2 describes the motivating example. Section 3 presents the proposed SPLVD. Section 4 and Section 5 elaborate on experimental designs and results on evaluating SPLVD, respectively. Section 6 discusses implications and limitations. Section 7 reviews related work. Finally, we present threats to validity in Section 8 and conclude this article in Section 9.

2 Motivating Example and Background

This section first presents two examples in the real-world vulnerability dataset to motivate our work, and then explains why self-paced learning is effective for vulnerability detection tasks even when the dataset contains noisy or erroneous source code.

2.1 Motivating Example

We found two types of data in the vulnerability dataset that might be wrongly labeled: (1) unrelated source code, and (2) non-modified source code. Then, we combined relevant studies to emphasize the significance of approaches for automatically filtering out low-quality data.

As illustrated in Figure 1, an example in the BigVul dataset shows a case where unrelated code is wrongly labeled. In this example, only some method names have been modified (*e.g.*, *SetDeviceCredentials* has been changed to *RegisterForDevicePolicy*). Based on the current code context, it is difficult to determine if there are any vulnerabilities. However, in the dataset, it is labeled as vulnerable source code, and the type of the vulnerability is marked as CWE-399 (Improper Management of System Resources), which clearly has no relation to the code modification. Based on our analysis, the reason for this problem lies in the way the dataset is collected, which is by tracking the reference links of CVE. However, due to incorrect reference links or the fact that multiple areas are modified in a single commit, not every area is related to the vulnerability.

Then, we illustrate the potential problems of the dataset from the perspective of the collection approach of the BigVul [14] dataset. The BigVul dataset is obtained by crawling the code modifications related to vulnerabilities in real-world project commits, thereby extracting the code that may contain vulnerabilities. However, we discovered nine data items in the dataset where the code remained unchanged throughout, but was wrongly labeled as vulnerabilities. The example shown in Figure 2 is one of the code labeled as having vulnerabilities, but without any code modifications.

```

1 Index: chrome/browser/policy/browser_policy_connector.cc
2 diff --git a/chrome/browser/policy/browser_policy_connector.cc
3   -> b/chrome/browser/policy/browser_policy_connector.cc
3 index 922a2ca73d3cfa78151c7d1883f0aecc4b17fa7..288698853607cba13d1b8f39eadbe8e9f12aa5c0 100644
4 --- a/chrome/browser/policy/browser_policy_connector.cc
5 +-- b/chrome/browser/policy/browser_policy_connector.cc
6 @@ -98,7 +98,7 @@ ConfigurationPolicyProvider*
7     return recommended_cloud_provider_.get();
8 }
9
10 -void BrowserPolicyConnector::SetDeviceCredentials(
11 +void BrowserPolicyConnector::RegisterForDevicePolicy(
12     const std::string& owner_email,
13     const std::string& token,
14     TokenType token_type) {
15 @@ -146,10 +146,10 @@ std::string BrowserPolicyConnector::GetEnterpriseDomain() {
16     return std::string();
17 }
18
19 -void BrowserPolicyConnector::DeviceStopAutoRetry() {
20 +void BrowserPolicyConnector::ResetDevicePolicy() {
21 #if defined(OS_CHROMEOS)
22     if (device_cloud_policy_subsystem_.get())
23 -     device_cloud_policy_subsystem_->StopAutoRetry();
24 +     device_cloud_policy_subsystem_->Reset();
25 #endif
26 }
```

Fig. 1. The Unrelated Code Is Wrongly Labeled

```

1 // The code of the "func_before" field
2 void AudioHandler::SetMute(bool do_mute) {
3     if (!VerifyMixerConnection())
4         return;
5     DVLOG(1) << "Setting Mute to " << do_mute;
6     mixer_->SetMute(do_mute);
7 }
```

```

1 // The code of the "func_after" field
2 void AudioHandler::SetMute(bool do_mute) {
3     if (!VerifyMixerConnection())
4         return;
5     DVLOG(1) << "Setting Mute to " << do_mute;
6     mixer_->SetMute(do_mute);
7 }
```

Fig. 2. The Non-Modified Code Is Marked as Vulnerable

This function only performed the connection check and forwarded it to the *mixer*. It does not parse the external input or perform any dangerous memory operations. Therefore, this code is more like an error extraction rather than a vulnerability. The reason for this problem might be that the dataset failed to accurately track the commits during the collection process.

In addition to the problems we identify, relevant studies have shown that the datasets in the field of vulnerability detection are generally of poor quality. Croft et al. [10] pointed out that many current vulnerability datasets suffer from issues such as inconsistent labels, duplicate data, and erroneously classified source code. Nie et al. [44] indicated that when noise is introduced into the vulnerability dataset, the effectiveness of various vulnerability detection approaches has significantly declined. These indicate that the quality of the dataset has a considerable impact on the detection performance of the model, especially in the early stage of training. The model is forced to learn from a full dataset that contains a great amount of noise, resulting in slow convergence and potential overfitting to noisy features, which leads to a high error rate in real-world applications. Therefore, in this work, we aim to design an adaptive data selection approach based on self-paced learning. The core motivation is to quantify source code difficulty and dynamically adjust the selection of the training data, allowing the model to select high-quality data autonomously. This approach enables the model to first master core patterns from “clean” data and then gradually introduce more hard source code, thereby reducing noise interference and improving the model’s learning efficiency and stability.

2.2 Technical Preliminaries

Self-paced learning is a machine learning strategy that mimics the human autonomous learning process [1], where students adjust their learning pace and curriculum based on their mastery of knowledge. Its design enables the model (“student”) to autonomously select training samples (“curriculums”) from easy to hard (first learning easy and reliable samples, and then gradually introducing hard and challenging samples), thereby improving model performance and generalization capability [30]. It is highly adaptable to vulnerability detection tasks because the quality of the vulnerability dataset is poor, and self-paced learning can reduce the impact of noise samples [55].

There are two key parameters in self-paced learning: *difficulty* and *age* [58]. The difficulty is determined by using the sample training loss of the current model as the standard for measuring the difficulty of the samples. The age is a parameter used to control the learning pace and determines the proportion of the easiest selected samples at each training epoch. Therefore, leveraging self-paced learning requires designing the measure and update approaches for these two key parameters based on the specific task. The difficulty measurer determines the relative difficulty of samples, providing reference information for sample selection. The age updater updates the age parameter based on the training state and adjusts the proportion of training samples selected.

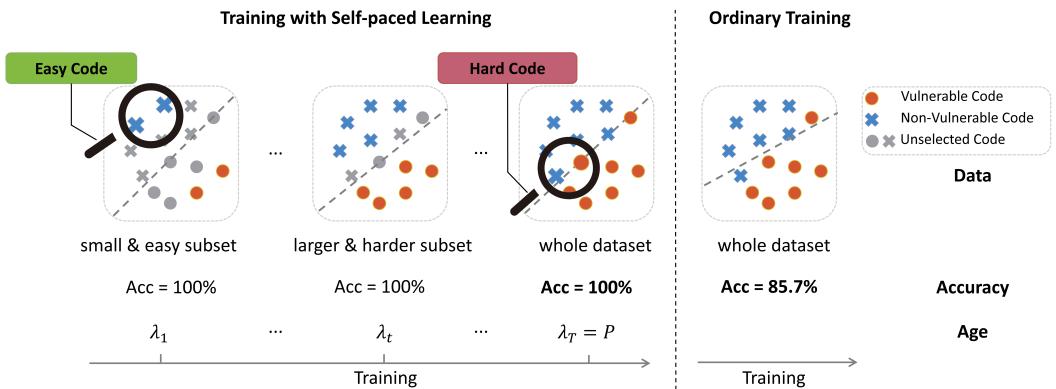


Fig. 3. An Illustration of Self-Paced Learning for Model Training

The workflow of self-paced learning is illustrated in Figure 3. Common training directly learns from the entire training set, which contains a large amount of hard source code and noise. This causes the model to fall into local optima at the early stages, resulting in relatively low accuracy (85.7%). In contrast, self-paced learning mimics the human learning process by starting with a “small and simple subset”, allowing the model to quickly grasp easily distinguishable patterns in the early phase. As training progresses, the model gradually introduces harder subsets, thereby continuously enhancing its capability to handle hard cases. By the time the model is capable of processing the entire dataset, it is still able to maintain high accuracy (100%).

3 Approach: SPLVD

This section first outlines SPLVD, then presents the backbone of the model, the selection of training data, and the self-paced learning training algorithm, respectively.

3.1 Overview

The overview of SPLVD is shown in Figure 4. Built upon a pre-trained code language model, SPLVD incorporates self-paced learning to select training source code. SPLVD first calculates the difficulty

of all source code and initializes the age parameter. Then SPLVD uses the data selector to select source code with lower difficulty as the training data. Unlike traditional approaches that perform fixed-batch learning, self-paced learning operates within the data selector during training. As shown in Figure 4 (b), self-paced learning dynamically chooses the source code whose difficulty is lower than the current age parameter, ensuring that at each stage the model learns from data most suitable for its current learning capability. During each training epoch, the difficulty of the source code is recalculated, and the age parameter is updated. Then, the selected training data is used for model parameter weight updates. This process is repeated continuously until the self-paced learning process is completed. Overall, SPLVD consists of three parts: (1) the model backbone constructed for vulnerability detection, (2) the training data selector based on self-paced learning, and (3) a training algorithm tailored for self-paced learning.

3.2 Model Backbone Construction

The model backbone consists of two components: (1) the pre-trained code language model, and (2) the classifier layer for vulnerability detection.

Pre-trained Code Language Model. We use UniXcoder [19] as the pre-trained code model for feature extraction. UniXcoder integrates code comments and AST, enabling the detection of syntax errors (e.g., null pointers, array out-of-bounds) and logical vulnerabilities by comparing code with

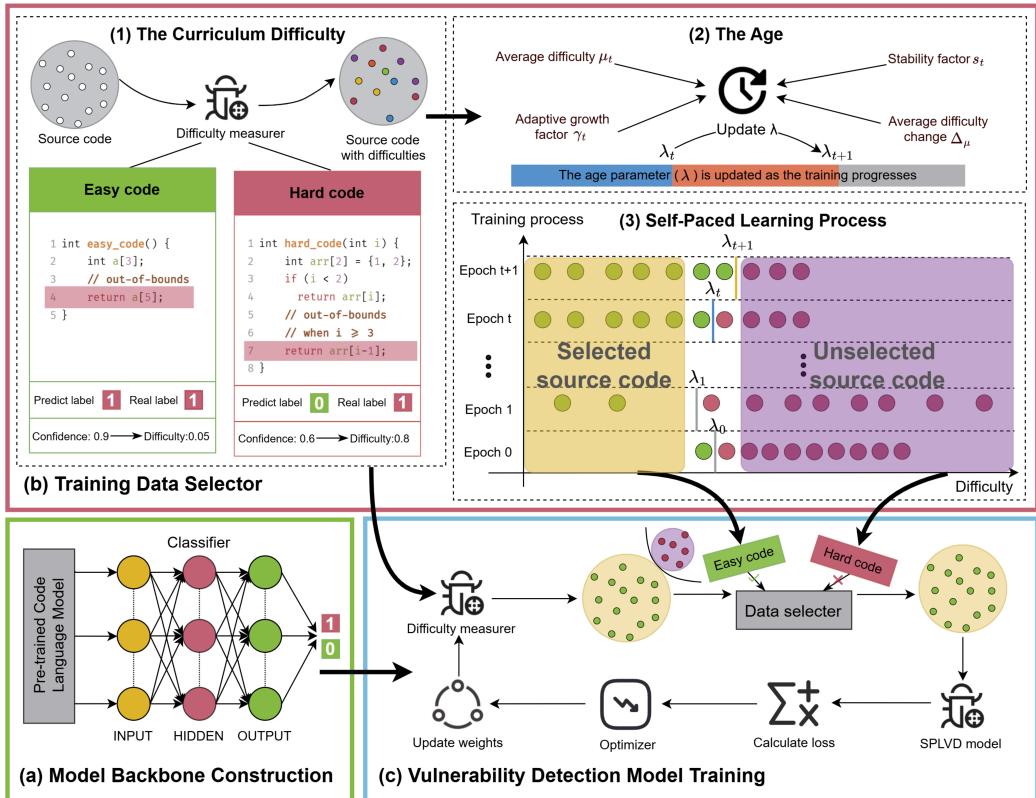


Fig. 4. An Overview of the Proposed SPLVD

comments. Its one-to-one AST mapping preserves structural information better than traditional models, reducing missed detections of hidden source code vulnerabilities.

Classifier. We use RNN as the classifier layer [33] for its compatibility with the context-sensitive nature of vulnerability detection: (1) its gating mechanism and sequence modeling capture long-range code dependencies that static classifiers such as fully connected layers cannot effectively learn; (2) building on sequence-level of UniXcoder syntactic and semantic representations, it further models local and global dynamic dependencies, augmenting the detection of context-sensitive vulnerabilities and improving classification accuracy.

Model Backbone Construction. We build SPLVD using UniXcoder as the base model and LSTM as the classifier. First, SPLVD utilizes the tokenizer of Unixcoder to convert the source code into token ID sequences. Sequences longer than 512 tokens are truncated and shorter ones zero-padded for consistent input length. The token IDs are processed by Transformer encoder of the UniXcoder [57], and the “[CLS]” mark of token token-pooled output provides a global feature vector representing the overall semantics of the source code. The feature vector is then expanded into a sequence and fed into a single-layer RNN, which captures long-range dependencies. Finally, a fully connected layer compresses the RNN output into a logits vector, and a sigmoid function produces the vulnerability probability for binary cross-entropy loss optimization.

3.3 Training Data Selector

To mitigate the impact of noisy or low-quality examples on model training, SPLVD adopts self-paced learning that gradually selects training source code “from easy to hard”, enabling the model to learn more robustly. Self-paced learning process is guided by two key factors: (1) *the curriculum difficulty of training data* and (2) *the age of the vulnerability detection model*. Both the source code difficulty and the age are updated by *the self-paced update algorithm* throughout training, and their relative values determine whether source code is included in each epoch.

The Curriculum Difficulty. For each source code x_i with a true label $y_i \in \{0, 1\}$ (where 1 indicates a vulnerability and 0 indicates no vulnerability), let $p_{i1} \in [0, 1]$ represent the probability that the model predicts x_i as vulnerability source code, and $p_{i0} \in [0, 1]$ represent the probability that the model predicts x_i as non-vulnerability source code (both are outputs from the model’s final sigmoid layer). SPLVD defines the confidence difference $conf_i$ as the difference between the two class prediction probabilities: $conf_i = \text{abs}(p_{i0} - p_{i1})$ where $conf_i \in [-1, 1]$, and the larger the absolute value, the stronger the model’s confidence in the prediction (vulnerability or non-vulnerability). The source code difficulty d_i is calculated based on $conf_i$ and prediction correctness.

$$d_i = \begin{cases} \frac{1-conf_i}{2} & \text{if } \hat{y}_i = y_i \quad (\text{correct prediction}) \\ \frac{1+conf_i}{2} & \text{if } \hat{y}_i \neq y_i \quad (\text{incorrect prediction}) \end{cases} \quad (1)$$

where $\hat{y}_i = \text{argmax}(p_{i1}, p_{i0})$ is the predicted label of models for x_i . This difficulty definition ensures that $d_i \in [0, 1]$, with smaller values indicating lower difficulty. For correctly predicted source code, the difficulty decreases as the prediction confidence $conf_i$ increases (e.g., for source code with $conf_i = 0.9$ and a correct prediction, $d_i = 0.05$, making it easy source code); for misclassified source code, the difficulty increases as confidence grows (e.g., for source code with $conf_i = 0.9$ but a wrong prediction, $d_i = 0.95$, due to the model’s “overconfident error”, making it high-difficulty source code). This design mimics the learning pattern of humans. Source code that is correctly identified with high confidence is considered “easy”. At the same time, those who mislead the model and make it overconfident are hard source code. As the model trains on the source code, its capability to fit the source code improves, and the difficulty of the source code gradually decreases.

The Age. The decision of selection of source code depends on its relationship with the age of the vulnerability detection model (denoted as λ_t in the t -th epoch). The age parameter controls the difficulty threshold for training, dynamically adjusting throughout the training process. The age ensures that the model gradually fits more hard source code as it improves. For initialization, λ_0 is set based on the difficulty distribution of all training source code. Specifically, it is initialized to the difficulty value at a certain quantile of the sorted training source code difficulties, defined by r_{init} . This ensures that the model starts training with the easy source code.

Self-Paced Learning Process. The curriculum difficulty and the age parameter λ are updated by self-paced learning during training the model backbone in SPLVD. At each epoch t , λ is updated as follows.

$$\lambda_{t+1} = \lambda_t + \gamma_t \cdot (1 - \mu_t) \cdot s_t + \Delta_\mu \quad (2)$$

where γ_t is the adaptive growth factor, μ_t is the average difficulty, s_t is the stability factor, Δ_μ is the average difficulty change: $\Delta_\mu = \mu_t - \mu_{t-1}$. The definitions of γ_t and s_t are as follows.

$$\gamma_t = \gamma_0 \cdot (1 + \alpha \cdot (1 - r)) \quad s_t = \frac{1}{1 + k \cdot \sigma_t} \quad (3)$$

where γ_0 is the initial hyperparameter of the growth factor, α is a hyperparameter that controls the growth rate of γ , r is the proportion of the currently selected source code to the total number of source code, σ_t is the standard deviation of source code difficulty, and k is the stability coefficient hyperparameter.

These parameters respectively have the following functions. The adaptive growth factor γ_t controls the update rate of λ . When fewer source code are selected (r is small), γ_t increases, encouraging the inclusion of harder source code. When r is high, the growth slows to prevent early usage to hard source code. The stability factor s_t adjusts the update speed according to the stability of the distribution of difficulties. When σ_t is smaller, s_t becomes larger, thereby enabling λ to grow at a faster rate. The average difficulty change. This allows the update to adapt to changes in overall training difficulty. The inclusion of Δ_μ at the end serves to ensure that the update of λ can adapt to the significant changes in overall training difficulty throughout the entire training process, preventing the changes from causing the self-paced learning to fail.

In addition, to avoid the problem of unstable training causing too much hard source code to be added within one epoch, we limit the number of newly added source code in each epoch. This restriction ensures that the proportion of selected source code does not grow too quickly, helping maintain training stability and reduce overfitting. Specifically, during each training step, we calculate the proportion of the currently selected source code and based on this, determine how much new source code to add in the next step. However, there is an upper limit to the number of additions, and it cannot exceed a pre-defined growth rate r_{max} . In this way, new and more challenging source code will be gradually introduced into the training process, without experiencing significant changes.

Our dynamic age (λ) update algorithm is summarized in Algorithm 1. By integrating the adaptive growth factor γ , stability factor s , and difficulty change Δ_μ , the model progressively includes more source code “easy-to-hard” while maintaining stable training dynamics. The source code selection constraint further reinforces this process by controlling the increase in speed of age.

3.4 Vulnerability Detection Model Training

The main components of training algorithms are: (1) the definition of the loss function, and (2) the selection of optimization approaches.

Loss Function. The optimization objectives of self-paced learning include the model parameters θ and the self-paced weights (data selection variable) of the samples $v = [v_1, \dots, v_N]^\top \in [0, 1]^N$.

Algorithm 1: Age Parameter Update in Self-Paced Learning

Input : Current age parameter λ_t ; current selected ratio r_t ;
 Mean difficulties μ_t, μ_{t-1} ; difficulty set D_t ;
 Growth factor γ_0 ; adjustment factor α ;
 Max increment ratio r_{\max} ; stability coefficient k

Output: Updated age parameter λ_{t+1}

```

1 Function updateLambda( $\lambda_t, r_t, \mu_t, \mu_{t-1}, D_t$ )
2    $\gamma_t \leftarrow \gamma_0 \cdot (1 + \alpha(1 - r_t))$  ;                                 $\triangleright$  Adaptive growth factor
3    $\Delta_\mu \leftarrow \mu_t - \mu_{t-1}$  ;                                          $\triangleright$  Difficulty change
4    $\sigma_t \leftarrow \text{std}(\text{subset near } \lambda_t)$  ;                          $\triangleright$  Local std near threshold
5    $s_t \leftarrow \frac{1}{1+k \cdot \sigma_t}$  ;                                          $\triangleright$  Stability factor
6    $\lambda' \leftarrow \lambda_t + \gamma_t(1 - r_t) \cdot s_t + \Delta_\mu$  ;            $\triangleright$  Proposed update
7    $r' \leftarrow \frac{|\{d_j \in D_t | d_j \leq \lambda'\}|}{|D_t|}$  ;                       $\triangleright$  New selected ratio
8   if  $r' - r_t > r_{\max}$  then
9      $r_{\text{target}} \leftarrow r_t + r_{\max}$  ;
10     $\lambda' \leftarrow \text{quantile}(D_t, r_{\text{target}})$  ;                            $\triangleright$  Re-adjust via quantile
11  end
12   $\lambda_{t+1} \leftarrow \min(1, \max(0, \lambda'))$  ;                                $\triangleright$  Clip to [0, 1]
13  return  $\lambda_{t+1}$ 
14 EndFunction

```

Dividing the training set into M batches, then let the average loss of the j -th batch be $L_j(\theta) = \frac{1}{B_j} \sum_{i \in \text{batch } j} \ell_i(\theta)$ (where ℓ_i is the single-sample cross-entropy). Introduce a batch selection variable $v_j \in \{0, 1\}$ and a age parameter λ [58]. The joint training objective of SPLVD is as follows.

$$\min_{\theta, v \in \{0,1\}^M} \mathcal{E}(\{d\} \mid \theta, \lambda) = \sum_{j=1}^M v_j L_j(\theta) - \lambda \sum_{j=1}^M v_j \quad (4)$$

where $\{d\}$ represents the difficulty list of each batch calculated using the difficulty calculation algorithm in Section 3.3. Notably, v_j depends on the sample difficulty and the current age parameter: $v_j = 1$ if and only if $d_j < \lambda$, otherwise $v_j = 0$. That is, SPLVD selects “easy” batches (with difficulties below the threshold λ) in the early training stage, gradually introduces more hard batches as λ increases, thereby achieving sample scheduling from easy to hard and reducing the impact of sample noise on training.

Optimizer. To solve Equation (4), we use AdamW [38] as the optimization algorithm for self-paced learning, where the model starts by training on easy source code to quickly converge and gradually learn more hard ones. Self-paced learning requires high optimization stability and strong generalization ability. AdamW, an extension of Adam, decouples weight decay from the gradient update, preventing instability caused by the coupling of L2 regularization [22] and adaptive learning rates in traditional Adam. This calculation allows AdamW to maintain rapid convergence while reducing overfitting, thus supporting both training stability and performance in self-paced learning. The main formula of the AdamW algorithm is as follows.

$$\theta_t = \theta_{t-1} - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda_{\text{wd}} \theta_{t-1} \right), \quad \hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (5)$$

where $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$ and $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$, η is the base learning rate, ϵ is a small constant to prevent division by zero, and λ_{wd} is the weight decay coefficient. Using AdamW algorithm can combine adaptive step size (adjusted by \hat{v}_t) with momentum direction (provided by \hat{m}_t) and apply L_2 -type regularization. The optimizer achieves a balance between convergence stability and generalization capability during the easy-to-hard training process required by self-paced learning.

4 Experimental Design

This section presents the designs in our experiment, including research questions, datasets, baselines, evaluation metrics, and hyperparameter settings.

4.1 Research Questions

We propose five research questions (RQs) to guide the experimental evaluation of SPLVD.

RQ1: How effective is SPLVD in terms of overall performance? (Overall Performance)

RQ2: How effective is SPLVD in terms of CVE categories? (Detailed Analysis)

RQ3: How important is self-paced learning of SPLVD? (Ablation Study)

RQ4: How generable is self-paced learning of SPLVD? (Generalization Capability)

RQ5: How effective is SPLVD in real-world application? (Case Study)

4.2 Datasets

We build the experimental datasets through three steps: (1) dataset selection, (2) processing, and (3) splitting. Table 1 shows the basic information of each dataset.

Step 1: Dataset Selection. We select three most commonly used datasets and devise a new one to investigate the usefulness of SPLVD in practice.

- **BigVul** [14] consists of 180K source code from 348 open-source projects. Its key features include: (1) inclusion of both vulnerable and patched code; (2) coverage of 11 types of vulnerabilities in C/C++, e.g., buffer and integer overflows; (3) provision of code change difference analysis. However, researchers have identified incorrect labels in this dataset.
- **Devign** [72] consists of 27K source code from four open-source projects, such as FFmpeg and QEMU, and includes over, labelled through a cross-validation process based on security-related commits. This dataset utilizes Graph Neural Networks (GNNs) to capture program semantics and transforms source code into a Code Property Graph (CPG).
- **ReVeal** [5] consists of 22K source code from two open-source projects. It is constructed through a multi-stage process: it uses the Joern tool to parse code into intermediate representations, employs graph neural networks to model code structures, and incorporates real-world vulnerability data (e.g., CVEs) to enhance source code diversity.
- **OpenHarmony**. To address RQ5, we develop a real-world dataset using OpenHarmony projects. We chose OpenHarmony because its code is relatively new and most of its content has not been utilized by pre-trained models, which can more realistically simulate the production environment. We selected several critical components (e.g., arkcompiler_ets_runtime, arkui_ace_engine, and communication_ipc) in OpenHarmony, which play essential roles in compilation, UI rendering, and inter-process communication. We then queried all reported CWE vulnerabilities from these projects, located the corresponding commit links via their references, and parsed the code change to extract potential vulnerable functions for annotation, thereby building the OpenHarmony dataset.

Table 1. Basic Information of the Experimental Datasets

Dataset	#Non-Vul.	#Vul.	#Projects	Published Year	Ref.
BigVul	177,736	10,900	348	2020	[14]
Devign	14,858	12,460	4	2019	[72]
ReVeal	20,494	2,240	2	2022	[5]
OpenHarmony	574	167	5	2025	[17]

Step 2: Pre-Processing. We conducted data cleaning and processing on the inconsistent labels, duplicate data, and erroneously classified source code present in the dataset, reducing the influence of erroneous code on model training before training.

Step 3: Splitting. Each dataset is divided into training, validation, and test sets in an 8:1:1 ratio. The split is fixed to ensure that the same data partitions are used for all approaches.

4.3 Baselines

We compare SPLVD with eight state-of-the-art approaches that use pre-trained models. Note that VulGPT [29] has two implementations.

- **CodeT5** [59] is a pre-trained model based on an encoder-decoder architecture that captures code characteristics and the correlation between natural language and code. It is fine-tuned for vulnerability detection, combining code comments to help identify logical vulnerabilities.
- **CodeBERT** [15] is a bimodal pre-trained model based on the Transformer architecture, capturing semantic relationships between natural language and code. After fine-tuning, it is effective for vulnerability classification and cross-modal code understanding.
- **UniXcoder** [19] is a unified cross-modal code model that integrates code, comments, and Abstract Syntax Tree (AST) information. It uses multi-task training to enhance robustness and capture dependencies related to vulnerabilities.
- **Starcoder2** [39] is an LLM optimized for long-context processing. It supports multi-objective training, making it highly effective in detecting vulnerability patterns in long code and across multiple programming languages.
- **VulGPT** [29] is a vulnerability detection model based on pre-trained models. It utilizes full-parameter fine-tuning and word embedding extraction to demonstrate efficient transfer learning capabilities for vulnerability classification.
- **EPVD** [67] is a vulnerability detection model based on CodeBERT. It extracts code execution paths from the Control Flow Graph (CFG) and enhances feature fusion through CNN attention.
- **LineVul** [16] is a vulnerability detection model based on Transformer architectures. It utilizes a two-stage process for function-level prediction and line-level localization without the need for external tools.

4.4 Evaluation Metrics

Following prior work [5, 10, 32], we employ five metrics to evaluate each approach: Accuracy (ACC), Precision (P), Recall (R), F1 ($F1$), and Matthews Correlation Coefficient (MCC). TP denotes the number of vulnerable source code that are detected as vulnerable, FP denotes the number of source code that are not vulnerable but are detected as vulnerable, TN denotes the number of source code that are non-vulnerable and are detected as not vulnerable, and FN denotes the number of vulnerable source code that are detected as non-vulnerable.

- **Accuracy** ($Acc = \frac{TP+TN}{TP+TN+FP+FN}$) measures the proportion of correctly classified source code among all source code.
- **Precision** ($P = \frac{TP}{TP+FP}$) measures the proportion of correctly predicted vulnerable source code among all source code predicted as vulnerable.
- **Recall** ($R = \frac{TP}{TP+FN}$) measures the proportion of correctly predicted vulnerable source code among all actual vulnerable source code.
- **F1** ($F1 = \frac{2 \times P \times R}{P+R}$) considers both accuracy and recall rate, reflecting the balance between detecting vulnerabilities and avoiding false positives.

- **Matthews Correlation Coefficient** ($MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$) considers all four categories (TP , TN , FP , and FN), useful for imbalanced vulnerability datasets. MCC values range between -1 and 1, with 1 being the optimal value.

We consider F1 to be the most critical evaluation standard. F1 is the most commonly used metric in vulnerability detection (Kalouptsoglou et al. [27]), making it easier to compare with other research. The reason for choosing MCC is that some datasets we selected are unbalanced, and it is one of the popular metrics used in related work [64].

4.5 Hyperparameter Settings

For baselines such as LineVul, EPVD, and VulGPT, we adopt the best hyperparameters reported in the original paper or the replication packages. For SPLVD, we perform a grid search to seek the optimal hyperparameters. The search space is defined following common practices and related work, including: (1) The proportion of initial selection codes (r_{init}): (0.05, 0.1, 0.15); (2) Stability Coefficient (k): (5, 10, 15); (3) The initial value of the growth factor (γ_0): (0.02, 0.025, 0.03); (4) The growth rate of γ (α): (0.2, 0.3, 0.4); (5) The max value of r growth (r_{max}): (0.05, 0.1, 0.15). For each hyperparameter setting, we train the model and use F1 on the validation set to select the best hyperparameter setting. Training is terminated early if the validation loss does not improve for five consecutive epochs. Table 2 presents the optimal hyperparameter.

Table 2. Hyperparameter Settings for the Four Approaches

LineVul		EPVD		VulGPT		SPLVD	
epochs	10	epochs	8	max_length	512	epochs	50
block_size	512	block_size	400	patience	5	max_length	512
train_batch_size	16	train_batch_size	40	batch_size ¹	64	batch_size	16
eval_batch_size	16	eval_batch_size	64	batch_size ²	8	learning_rate	1e-5
learning_rate	2e-5	learning_rate	2e-5	epochs ¹	100	r_{init}	0.1
max_grad_norm	1.0	max_grad_norm	1.0	epochs ²	10	k	10
		cnn_size	128	learning_rate ¹	1e-3	γ_0	0.025
		filter_size	3	learning_rate ²	2e-5	α	0.3
		d_size	128			r_{max}	0.1

¹ It employs the word_embedding approach.

² It employs the fine-tuning approach.

We implemented SPLVD using PyTorch. The experiments are performed on a machine with 1 NVIDIA GeForce RTX 4090 GPU and 1 vCPU Intel(R) Xeon(R) Gold 6459C.

5 Result and Analysis

This section reports on results and analysis to answer research questions.

5.1 RQ1: Overall Performance

Setup. This experiment is evaluated on the standard test sets of three commonly used vulnerability detection datasets (BigVul, Devign, and ReVeal), comparing eight representative baselines with SPLVD. Five evaluation metrics are adopted: Accuracy (Acc), Precision (P), Recall (R), Matthews Correlation Coefficient (MCC), and F1 ($F1$), with F1 serving as the primary comparison metric. We compare with eight baseline approaches, which can be broadly classified into three categories: using pre-trained models (CodeT5, CodeBERT, and UniXcoder), using Lora [23] to fine-tune large language models (StarCoder2), and the state-of-the-art approaches from related works (VulGPT, EPVD, and LineVul). All approaches are executed under the same data splits and evaluation scripts, with the best metric for each dataset highlighted in bold for comparison (cf. Table 3).

Results. SPLVD achieves the highest F1 across all three datasets, with particularly outstanding performance on BigVul, where it reaches 89.2%, representing an improvement of about 2.9% over the second-best approach VulGPT (fine-tuning) at 86.3%. On BigVul, it achieves the highest Accuracy (98.8%) and Precision (96.1%), while maintaining a relatively high Recall (83.3%), indicating that SPLVD can effectively identify true vulnerabilities with very high precision. On Devign, SPLVD achieves the best Accuracy (74.8%) and F1 (68.7%), which is about 1.7% higher than VulGPT (67.0%), and its overall Accuracy is 13.8% higher, showing a clear advantage despite a relatively moderate Precision (56.1%). On the more challenging ReVeal dataset, SPLVD still achieves the highest F1 (43.5%), about 1.4% higher than VulGPT (42.1%), with the improvement mainly attributed to its higher Recall (51.8%, the best among all approaches). These results suggest that SPLVD demonstrates stable improvements in F1 across different datasets, with particular strengths in enhancing recall and reducing false negatives.

For the *MCC* metric, SPLVD demonstrated the best performance across all datasets. On BigVul, its value reached 0.889, which was 0.03 higher than the second-best VulGPT (0.858). On both Devign and ReVeal, SPLVD achieved the highest *MCC* values of 0.338 and 0.368, respectively. These results indicate that, in addition to the F1 and Recall rate, SPLVD achieves a more balanced classification between vulnerable source code and non-vulnerable source code. Another representative phenomenon is that certain models (e.g., StarCoder2) achieved extremely high Recall (99.8%) on Devign but at the cost of very low Precision, resulting in a very low *MCC* value. This further highlights the necessity of balancing both Precision and Recall in vulnerability detection tasks.

Table 3. The Overall Performance of Nine Approaches

Approach	BigVul					Devign					ReVeal				
	Acc	P	R	MCC	F1	Acc	P	R	MCC	F1	Acc	P	R	MCC	F1
CodeT5	0.980	0.889	0.756	0.810	0.817	0.649	0.632	0.552	0.288	0.590	0.902	0.504	0.286	0.331	0.365
CodeBERT	0.978	0.886	0.709	0.781	0.788	0.618	0.693	0.292	0.232	0.411	0.905	0.569	0.147	0.255	0.234
UniXcoder	0.982	0.907	0.767	0.825	0.831	0.660	0.653	0.543	0.309	0.593	0.904	0.553	0.116	0.222	0.192
StarCoder2	0.953	0.889	0.035	0.171	0.067	0.457	0.456	0.998	-0.012	0.626	0.902	0.625	0.022	0.105	0.043
VulGPT ¹	0.985	0.938	0.792	0.854	0.859	0.610	0.546	0.866	0.292	0.670	0.868	0.359	0.433	0.321	0.392
VulGPT ²	0.985	0.937	0.800	0.858	0.863	0.652	0.624	0.600	0.297	0.611	0.900	0.488	0.371	0.366	0.421
EPVD	0.981	0.871	0.793	0.821	0.830	0.648	0.623	0.581	0.288	0.601	0.886	0.419	0.402	0.347	0.410
LineVul	0.984	0.892	0.828	0.851	0.859	0.644	0.600	0.659	0.289	0.628	0.896	0.467	0.380	0.365	0.419
SPLVD	0.988	0.961	0.833	0.889	0.892	0.748	0.561	0.887	0.338	0.687	0.867	0.374	0.518	0.368	0.435

¹ It employs the word_embedding approach.

² It employs the fine-tuning approach.

Answer to RQ1: SPLVD consistently achieves the best F1 across all datasets, demonstrating its effectiveness in balancing precision and recall for vulnerability detection. The highest *MCC* value of SPLVD indicates a better balance in identifying different types of source code.

5.2 RQ2: Detailed Analysis

Setup. This experiment selects the top ten most frequent vulnerability categories in the BigVul dataset (classified by CWE-ID [42]) as the subjects of analysis (cf. Table 4). For each type, we first pick out the vulnerable source code of this type from the BigVul dataset and then randomly select a certain proportion of non-vulnerable source code from all the non-vulnerable source code in the raw dataset so that the ratio of the selected vulnerable source code to the selected non-vulnerable source code is the same as that in the raw dataset. To ensure the reliability of the comparison, we exclude StarCoder2 since it performs extremely poorly in RQ1 on BigVul and yields an F1 of 0 on several CWE categories. The experiment compares SPLVD with baselines in terms of F1 on each CWE category, to examine the applicability and performance gains of self-paced learning.

Results. SPLVD outperforms baselines on most common CWE categories, with particularly notable advantages in several syntax/pattern-obvious vulnerability types, such as CWE-125 (Out-of-bounds Read), CWE-476 (NULL Pointer Dereference), CWE-190 (Integer Overflow or Wraparound), and CWE-189 (Numeric Errors). This demonstrates that self-paced learning prioritizes “easy-to-detect” source code and gradually introduces harder ones to improve the model’s capability to capture vulnerability patterns with clear code characteristics. Meanwhile, in a few categories (e.g., CWE-200 and CWE-264), baselines slightly outperform SPLVD, suggesting that these vulnerabilities require stronger contextual reasoning or task-specific semantic knowledge, where a pre-trained model and fine-tuning strategy have an advantage.

For buffer overflow/out-of-bounds vulnerabilities (CWE-119/CWE-125), SPLVD achieves a high F1 of 88.2% and 88.3%, respectively, showing an improvement of 1%–3% over most baselines. This indicates that when vulnerabilities exhibit clear local syntactic or data-flow features, self-paced learning can better strengthen the model’s capability to detect such patterns. In contrast, for information disclosure/privacy-related CWE-200 and access control-related CWE-264, SPLVD performs slightly worse than certain VulGPT variants. Upon examining the collected data, we found that a possible reason is that these categories rely on broader contextual information and semantics, where cross-function/cross-file semantics or external knowledge captured during pretraining play an important role. In summary, SPLVD demonstrates superior performance in most common vulnerability categories with distinct code fingerprints, but for vulnerability types that depend on deeper semantics or cross-resource information, further improvements may require incorporating stronger semantic pretraining or context-enhanced approaches.

Table 4. The Performance (F1) of Eight Approaches Regarding CVE Categories

Approach	CWE-ID									
	CWE-119	CWE-20	CWE-399	CWE-125	CWE-200	CWE-264	CWE-476	CWE-190	CWE-189	CWE-362
CodeT5	0.833	0.848	0.790	0.763	0.854	0.875	0.735	0.923	0.851	0.826
CodeBERT	0.808	0.775	0.756	0.714	0.784	0.800	0.640	0.815	0.735	0.708
UniXcoder	0.855	0.847	0.826	0.796	0.874	0.896	0.735	0.925	0.833	0.783
VulGPT ¹	0.863	0.852	0.848	0.828	0.893	0.917	0.824	0.923	0.920	0.898
VulGPT ²	0.874	0.867	0.844	0.863	0.925	0.917	0.824	0.913	0.875	0.875
EPVD	0.842	0.829	0.838	0.757	0.844	0.833	0.800	0.931	0.776	0.868
LineVul	0.857	0.856	0.833	0.862	0.871	0.878	0.755	0.912	0.846	0.846
Our SPLVD	0.882	0.872	0.852	0.883	0.881	0.891	0.830	0.931	0.923	0.902

¹ It employs the word_embedding approach.

² It employs the fine-tuning approach.

✉ **Answer to RQ2:** SPLVD outperforms baselines on most common CWE categories, with particularly significant improvements in categories that exhibit clear syntactic or pattern-specific characteristics.

5.3 RQ3: Ablation Study

Setup. To examine the contribution of self-paced learning to model performance, this ablation study uses the baseline VD (UniXcoder as the pre-trained encoder and LSTM as the classifier) as the reference model, and constructs a variant SPLVD by incorporating self-paced learning into VD. The experiments are conducted on the BigVul, Devign, and ReVeal datasets, reporting the commonly used F1 as well as Top₂₅₀F1, Top₅₀₀F1, and Top₁₀₀₀F1, which are computed by ranking source code based on model confidence and evaluating the top 250, 500, and 1000 predictions, respectively. The reason for choosing Top_NF1 as the metric is that in the actual application of the vulnerability detection model, the Top_N data with higher prediction probabilities are regarded as

potential candidates for vulnerabilities and are submitted to manual inspection. All models were trained and evaluated with the same data splits and evaluation scripts, and Table 5 presents both the values of each metric and the relative improvements (with the values in parentheses denoting the gains over VD).

Results. As shown in Table 4, introducing self-paced learning into SPLVD leads to overall improvements in F1 across all three datasets: on BigVul, F1 increased from 86.6% to 89.1% (+2.5%); on Devign, from 65.1% to 68.6% (+3.5%); and on ReVeal, from 41.5% to 42.9% (+1.4%). For the confidence-ranked Top_N metrics, BigVul and ReVeal demonstrated significant improvements in most Top_N scores (e.g., BigVul’s Top₁₀₀₀F1 rises from 95.9% to 96.8%; ReVeal improves by 3.7%, 4.0%, and 2.4% on Top₂₅₀, Top₅₀₀, and Top₁₀₀₀, respectively), indicating that self-paced learning is effective in enhancing overall and high-confidence code detection effect.

On BigVul, SPLVD improved the overall F1 by 2.5% while increasing Top₁₀₀₀F1 by 0.9%, indicating that even when the high-confidence source code already has a good detection effect, self-paced learning can still further optimize the balance between recall and accuracy on more data. On Devign, although slight decreases are observed in Top₂₅₀ and Top₅₀₀, SPLVD still achieves a 3.5% improvement in overall F1, suggesting that by incorporating more high-confidence (a high level of confidence implies a lower degree of difficulty.) source code, self-paced learning effectively boosts detection capability on the full test set. On ReVeal, SPLVD achieves consistent improvements across all Top_N metrics (e.g., a 4.0% increase in Top₅₀₀F1), demonstrating that self-paced learning brings more pronounced benefits in more challenging data domains. In summary, self-paced learning dynamically selects training source code of varying difficulty, strengthening the model’s learning on easy cases, thereby improving overall F1 in most scenarios.

Table 5. The Effectiveness (F1) of Self-Paced Learning on Three Datasets

Dataset	VD ¹ -SPLVD ²			
	F1	Top ₂₅₀ F1	Top ₅₀₀ F1	Top ₁₀₀₀ F1
BigVul	0.866 - 0.891 (2.5%↑)	1.000 - 1.000 (0)	0.999 - 1.000 (0.1%↑)	0.959 - 0.968 (0.9%↑)
Devign	0.651 - 0.686 (3.5%↑)	0.984 - 0.978 (0.6%↓)	0.910 - 0.900 (1.0%↓)	0.799 - 0.800 (0.1%↑)
ReVeal	0.415 - 0.429 (1.4%↑)	0.534 - 0.571 (3.7%↑)	0.470 - 0.510 (4.0%↑)	0.429 - 0.453 (2.4%↑)

¹ VD means using UniXcoder as the pre-trained model and using LSTM as the classifier.

² SPLVD means training using self-paced learning on the basis of VD.

✍ **Answer to RQ3:** With the introduction of self-paced learning, SPLVD achieved improvements in overall F1 across all three datasets, and it shows an improvement in the identification of high-confidence source code.

5.4 RQ4: Generalization Capability

Setup. To verify whether self-paced learning is generally effective across different pre-trained models, we first select three pre-trained models (UniXcoder, CodeBERT, and CodeT5) based on the evaluation results from related works. We then conduct experiments on them, comparing two training strategies with and without self-paced learning (SPL/NO-SPL). All experiments are conducted on the BigVul, Devign, and ReVeal datasets, reporting four evaluation metrics: Accuracy, Precision, Recall, and F1, with F1 serving as the primary comparison metric. The experimental results are presented in Table 6.

Results. Using self-paced learning consistently improved F1 across all three pre-trained models on all three datasets. For UniXcoder, using self-paced learning raises F1 on BigVul, Devign, ReVeal from 86.6% to 89.1%, 65.1% to 68.6%, and 41.5% to 42.9%, respectively, with gains ranging from 1.4%

to 3.5%. For CodeBERT, self-paced learning improves BigVul and Devign by about 6.6% and 28.9%, respectively, showing that self-paced learning can significantly enhance the capability of certain pre-trained models to identify source code. For CodeT5, self-paced learning delivered substantial gains on BigVul (+16.0%) and Devign (+13.1%).

For CodeBERT, the most improvement appears on Devign, where F1 rises from 28.3 to 57.2 (+28.9%), mainly driven by a dramatic increase in Recall (from 18.7% to 68.6%), indicating that self-paced learning effectively helps the model capture more features. For CodeT5, F1 on BigVul jumped from 21.2% to 37.2% (+16.0%), though accuracy dropped from 89.3% to 83.5%, suggesting that self-paced learning significantly boosted recall of positive source code (R increases from 25.0% to 84.6%), but at the cost of a decrease in accuracy, in exchange for a stronger detection capability.

Table 6. The Effectiveness of Self-Paced Learning Under Different Pre-Trained Models

Pre-trained Model	Approach	BigVul					Devign					ReVeal				
		Acc	P	R	MCC	F1	Acc	P	R	MCC	F1	Acc	P	R	MCC	F1
UniXcoder	SPL ¹	0.988	0.947	0.841	0.887	0.891	0.642	0.572	0.856	0.341	0.686	0.875	0.389	0.478	0.362	0.429
	NO-SPL ²	0.984	0.854	0.878	0.857	0.866	0.652	0.599	0.713	0.313	0.651	0.847	0.333	0.549	0.347	0.415
CodeBERT	SPL	0.892	0.231	0.372	0.238	0.285	0.532	0.491	0.686	0.092	0.572	0.730	0.159	0.406	0.118	0.228
	NO-SPL	0.810	0.144	0.462	0.176	0.219	0.568	0.583	0.187	0.105	0.283	0.406	0.122	0.813	0.109	0.212
CodeT5	SPL	0.835	0.238	0.846	0.393	0.372	0.541	0.498	0.566	0.086	0.530	0.706	0.176	0.540	0.172	0.266
	NO-SPL	0.893	0.185	0.250	0.158	0.212	0.581	0.576	0.306	0.136	0.399	0.654	0.162	0.603	0.162	0.255

¹ SPL means using self-paced learning based on pre-trained models.

² NO-SPL means not using self-paced learning in model training.

↳ **Answer to RQ4:** Self-paced learning enhanced F1 across all pre-trained models and datasets, with particularly notable gains for CodeBERT and CodeT5 on certain datasets. In most cases, the F1 improvement is primarily driven by increased recall.

5.5 RQ5: Case Study

Setup. To validate the practicality of SPLVD in real-world scenarios, we selected the open-source operating system *OpenHarmony* as the target project. The experimental procedure was as follows: first, each trained model scanned the prepared source code and output its predicted vulnerable source code (denoted as *Identified Count*). Next, these candidate vulnerabilities were reviewed by security experts, who confirmed the number of actual vulnerabilities (denoted as *Confirmed Count*), from which precision was calculated. The information we have confirmed with the experts can be found at the replication package [60]. It is worth emphasizing that this process strictly simulates the real industrial vulnerability detection workflow, which refers to automatic identification by the model, followed by expert verification and final confirmation. Among the models, StarCoder2 failed to detect any valid vulnerabilities on this dataset and is therefore reported as N/A.

Results. As indicated in Table 7, the performance of different approaches varies significantly in real-world projects. LineVul identified the largest number of vulnerabilities (25), but its confirmation rate was only 60%, indicating that high detection came with a high false-positive rate. In contrast, SPLVD identified fewer vulnerabilities (11), but 10 of them were confirmed, achieving a precision of 90.9%, the best among all approaches. This result suggests that SPLVD is more suitable as a “high-confidence detector” in practical engineering settings: although its overall number of findings may be lower than some coverage-oriented baselines, its outputs are more reliable, significantly reducing the manual verification workload of security experts. For comparison, UniXcoder achieved relatively high precision (85.7%), but slightly lower than SPLVD, showing that self-paced learning provides additional benefits in reducing false positives and enhancing model stability. Overall, SPLVD demonstrates superior practical value in real-world applications.

Table 7. The Performance (Precision) of Eight Approaches on OpenHarmony

Approach	OpenHarmony		
	#Identified	#Confirmed	Precision
CodeT5	17	13	0.765
CodeBERT	5	3	0.600
UniXcoder	14	12	0.857
StarCoder2	N/A	N/A	N/A
VuLGPT ¹	16	11	0.688
VuLGPT ²	16	13	0.813
EPVD	9	5	0.556
LineVul	25	15	0.600
SPLVD	11	10	0.909

¹ It employs the word_embedding approach.

² It employs the fine-tuning approach.

✉ **Answer to RQ5:** SPLVD achieved highest confirmation rate in the real-world validation. This indicates that it is more suitable as a high-confidence vulnerability detection tool, effectively reducing manual verification costs and enhancing reliability in practical applications.

6 Discussion

This section first presents the implications we gained from the experiments, then explains how to use self-paced learning in practice, and finally discusses some limitations of our approach.

6.1 Implication

The approach presents four key implications: (1) SPLVD effectively measures low-quality source code as hard source code, proving the superiority of our difficulty measurer. (2) Self-paced learning reduces missed detections of vulnerabilities by mitigating overfitting. (3) Model backbone influences absolute performance, with stronger pre-trained encoders yielding higher baselines and weaker ones gaining more from self-paced learning. (4) Prediction threshold setting influences the precision/recall trade-off and should be appropriately selected based on business priorities.

SPLVD measures low-quality source code as hard. To investigate whether SPLVD can effectively separate the easy and hard source code and give priority to learning the easy source code. We extract the difficulty calculation results of the last epoch on three datasets. Since the model learning mainly focuses on the vulnerable source code, we plot the difficulty distribution of the source code on the vulnerable source code as shown in Figure 5. We can find that SPLVD trained on the three datasets can effectively distinguish the easy source code from the hard source code, and the value of the final age parameter enables the model to select more easy source code for learning, thereby reducing the impact of hard source code on the model.

In addition, we find that the source code that is ultimately labeled as hard has a high correlation with the low-quality data. This proves the superiority of our difficulty measurer. Taking the BigVul dataset as an example, all source code that we previously identified as potentially being mislabeled in Section 2 are found within the hard source code. We also select the ten hardest source code for manual analysis, and the result shows that no obvious vulnerabilities are found in these source code. At the same time, we find that among these source code, four of them are simple empty destructors wrongly marked as vulnerabilities. There are some examples, as shown in Figure 6, which have very short contexts, and it is impossible to directly determine whether there are vulnerabilities or not. This once again proves that there is low-quality code in the vulnerability dataset, and our

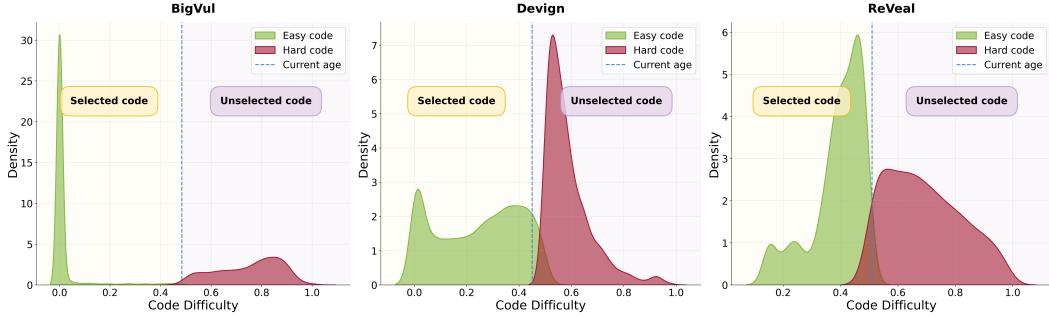


Fig. 5. SPLVD Measures the Distribution of Difficulty for Vulnerable Source Code on Three Datasets

approach is capable of effectively filtering out such source code. The analysis results of the code can be found in the replication package [60].

```

1 // Example 1
2 bool WebMediaPlayerMS::DidPassCORSAccessCheck() const {
3     DCHECK(thread_checker_.CalledOnValidThread());
4     return true;
5 }
```

```

1 // Example 2
2 void Download::start()
3 {
4     NotImplemented();
5 }
```

Fig. 6. Two Examples of Short Contexts Found Among the Ten Hardest Source Code

Self-paced learning reduces missed detections of vulnerabilities. The experimental results of RQ1, RQ3, and RQ4 demonstrate that self-paced learning consistently improves recall and F1 across different datasets and base models, confirming its effectiveness in enhancing vulnerability detection performance. These improvements can be attributed to two main factors: (1) the bespoke learning from easy to hard source code, and (2) dynamically introducing source code based on training age. By gradually learning from easy to hard source code, the model avoids early exposure to hard or noisy data, reducing overfitting and improving generalization, which helps it detect true vulnerabilities more effectively. Meanwhile, by dynamically introducing source code based on training age, self-paced learning gradually selects diverse data into the training process, allowing the model to capture global patterns more comprehensively and preventing gradients from being dominated by a narrow subset of source code. We selected some examples from the BigVul dataset that were correctly labeled. By referring to the code lines that the model’s attention focused on, we explain why the model trained using self-paced learning has better detection performance than the original model. Take Figure 7 as an example; this example is labeled as CWE-20 (Improper Input Validation) in the dataset. The potential vulnerability lies in the fact that the function does not check whether “vcpu” is null before accessing it. We can observe that SPLVD obtained through self-paced learning can precisely focus its attention on the rows that may cause an exception, while the VD that is not trained with self-paced learning fails to do and even focuses on the parentheses. This is the reason why its prediction results are incorrect. Together, these mechanisms explain why self-paced learning is particularly effective in boosting recall, thereby reducing the risk of missed vulnerabilities and improving system security.

Model backbone influences detection performance. According to the results of RQ4, differences in the initial capability of pre-trained encoders affect both the final performance and the gain brought by SPL. With UniXcoder as the baseline, applying SPL already achieved stable and relatively high F1 (BigVul 89.1%, Devign 68.6%, Reveal 43.5%), with steady improvements. In contrast, for base

The code lines SPLVD focuses on	The code lines VD focuses on
<pre> 1 static void vapic_exit(struct kvm_vcpu *vcpu) 2 { 3 struct kvm_lapic *apic = vcpu->arch.apic; 4 int idx; 5 if (!apic !apic->vapic_addr) 6 return; 7 if (idx = srcu_read_lock(&vcpu->kvm->srcu)); 8 kvm_release_page_dirty(apic->vapic_page); 9 mark_page_dirty(vcpu->kvm, apic->vapic_addr >> 10 PAGE_SHIFT); 11 } </pre>	<pre> 1 static void vapic_exit(struct kvm_vcpu *vcpu) 2 { 3 struct kvm_lapic *apic = vcpu->arch.apic; 4 int idx; 5 if (!apic !apic->vapic_addr) 6 return; 7 if (idx = srcu_read_lock(&vcpu->kvm->srcu)); 8 kvm_release_page_dirty(apic->vapic_page); 9 mark_page_dirty(vcpu->kvm, apic->vapic_addr >> 10 PAGE_SHIFT); 11 } </pre>
☒ This mark indicates the code lines that may cause vulnerabilities.	

Fig. 7. An Example Shows Models Trained with and Without Self-Paced Learning Focused on Different Rows

models such as CodeBERT and CodeT5, SPL provided much larger relative gains (e.g., CodeBERT on Devign improved F1 by 28.9%, and CodeT5 on BigVul improved by 16%). The results suggest that when the base model has weaker recall or representation capability, self-paced learning can provide more significant improvements. In engineering practice, it is advisable to prioritize pre-trained models with stable performance and strong semantic representation to achieve higher absolute performance, while recognizing that self-paced learning can serve as an effective enhancement strategy for weaker base models.

Prediction threshold influences detection performance. Our experiments revealed that the choice of prediction threshold τ_e during testing has an impact on results. Figure 8 shows the results in five evaluation metrics of SPLVD under different thresholds on three datasets. It can be observed that, especially on the smaller-scale datasets of Devign and ReVeal, Precision (P) and Recall (R) rates decrease and increase, respectively, as the threshold value increases. The highest F1 corresponds to different threshold values on different datasets. It indicates that the selection of the threshold has an impact on the detection performance of the model. In practical applications, thresholds should be set according to business priorities: higher thresholds when prioritizing precision, and lower thresholds when prioritizing recall. Thresholds can be dynamically adjusted based on the project requirements. For high-risk source code, a “low threshold + manual review” workflow can be adopted to balance efficiency and reliability.

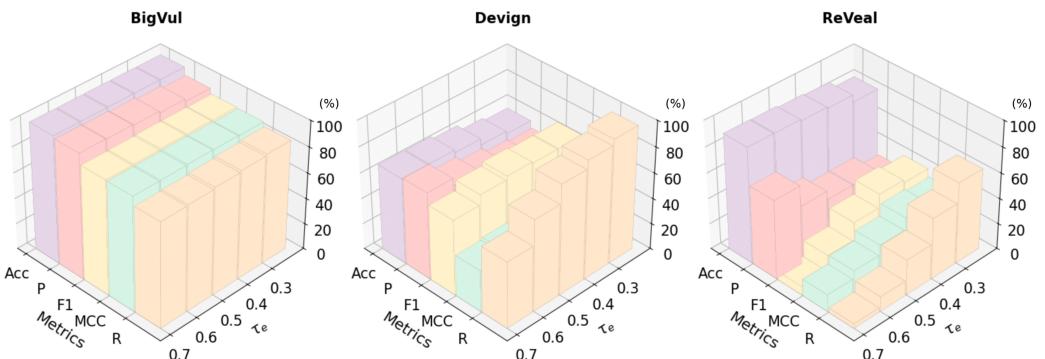


Fig. 8. The Performance of SPLVD Using Different Thresholds

6.2 Usage

The approach can be used to train vulnerability detection models with self-paced learning and then apply the trained model for automated vulnerability detection with expert review.

Usage of Self-Paced Learning in Model Training. For researchers in the field of code vulnerability detection, self-paced learning can be integrated into conventional model training workflows. For example, based on the UniXcoder pre-trained model and an LSTM classifier, source code difficulty is estimated from prediction confidence and correctness, and the selection threshold is dynamically adjusted using an age-parameter function. Only source code meeting the current difficulty standard is used for parameter updates, completing the self-paced learning process. This process progresses automatically with training iterations, eliminating the need for manual intervention in source code selection and thereby significantly reducing dependence on high-quality annotated data.

Usage of SPLVD in Vulnerability Detection. For DevSecOps teams, vulnerability analysts, etc., the trained model can be directly applied to predict vulnerabilities in new source code. For source code predicted with high vulnerability probability, a human review process can be incorporated to leverage expert understanding of vulnerability context, compensating for the model's limitations in hard logical reasoning. This “automated detection + manual verification” paradigm not only takes advantage of self-paced learning but also ensures the reliability of detection results.

6.3 Limitations

Though efforts have been made, SPLVD still has two limitations in terms of *training* and *verification*.

Longer Training Time. Due to the incorporation of additional source code difficulty estimation and dynamic training age updates in self-paced learning, the overall training process is more complex. On large-scale datasets or with deep pre-trained models, the extended training time becomes more pronounced, which limits the applicability of self-paced learning. Therefore, future work could explore optimizing the efficiency of difficulty calculation and source code selection to mitigate this issue while maintaining effectiveness.

Require Manual Verification. Although SPLVD achieved the highest F1 across multiple datasets, the model's precision remains insufficient in certain cases. This suggests that the model may still generate a non-negligible number of false positives. As a result, the current approach cannot fully replace manual inspection and needs to be complemented with human review. Future research should aim to further improve model precision across diverse datasets, driving vulnerability detection systems toward full automation and high reliability.

7 Related Work

This section reviews related work from perspectives of training strategy, data selection, and approaches in vulnerability detection.

7.1 Training Strategy

Using appropriate training strategies plays a significant role in enhancing the model's performance. self-paced learning has been proven to be beneficial in enhancing the performance and robustness of the model [62, 66]. Yuan et al. [65] incorporated self-paced learning with dynamic difficulty measurement and scheduling into hierarchical multi-label classification, achieving superior performance on 20 datasets. Zhao et al. [68] integrated self-paced learning with a symmetric scheduler and gradient-based difficulty measurement into domain generalization, significantly improving cross-domain performance. Chen et al. [6] applied self-paced learning with dynamic thresholding

and staged sampling to few-shot text classification, constructing the Self-paced Contrastive Network, which outperformed state-of-the-art approaches. Chen et al. [7] used self-paced learning with two-dimensional pseudo-label filtering and dynamic thresholding in semi-supervised text classification, constructing the Self-Paced Pairwise model, which outperformed the baseline approach and demonstrated excellent performance in alleviating overfitting and mislabeling problems.

7.2 Data Selection

Many studies have focused on the data quality problems existing in the commonly used vulnerability datasets [13, 36], which can lead to a decline in the performance of the model. For example, Nie et al. [44] investigated the causes, impacts, and denoising effects of label errors on the synthetic and real datasets. The results show that label errors arise during the data collection and annotation stages, and 30% of noise can cause an F1 drop of over 20% for the model. Croft et al. [10] adopted the standard five data quality attributes (accuracy, uniqueness, etc.) to conduct a quality assessment on four vulnerability datasets. They found that all datasets had quality problems. The accuracy rate of vulnerability labels in real-world datasets ranged from 20% to 71%, the duplication rate ranged from 17% to 99%, and these problems led to a 29% to 80% decrease in model precision. Dil et al. [12] noticed that the vulnerability dataset has quality problems. By employing strategies such as LLM combined with generative knowledge prompts, they studied the impact of LLM filtering on data quality and the performance of vulnerability prediction models. After filtering, BigVul increased the accuracy rates of models by 7% to 9%. However, using LLM for data filtering is too costly, and its adaptability to different vulnerability detection models varies.

7.3 Vulnerability Detection

When building approaches for vulnerability detection, researchers have selected different source code features and machine learning models [28, 46, 69]. Li et al. [32] extracted four types of grammatical features from the abstract syntax tree, and integrated data flow and control flow semantics through the program dependency graph. They adopted the bidirectional gated recurrent unit as the model. Fu and Tantithamthavorn [16] used a byte-pair encoding approach to segment sub-words, combined with the pre-trained CodeBERT model to generate word embeddings, and utilized the self-attention mechanism to achieve row-level positioning. Zhang et al. [67] constructed a grammar control flow graph based on the abstract syntax tree and selected three representative execution paths using the greedy algorithm. They chose the pre-trained CodeBERT model to extract features and used a CNN to capture the correlation features. Kalouptsoglou et al. [29] chose Word2Vec, FastText, and Bag-of-Words to extract code features, selected CodeBERT/CodeGPT2 to extract word embeddings or sentence embeddings. Recently, researchers have utilized LLMs for vulnerability detection. Lu et al. [40] used Joern to extract code attributes, employed CodeT5 to extract semantic features, and combined them with highly relevant code examples. Yang et al. [63] extracted the multi-form representations of the source code, the detection probabilities of the deep learning model, and the scan results of the static analysis tool. They then combined the two prompt techniques of Contextual Information Learning and Chain-of-Thought to achieve the adaptive optimization of LLM.

8 Threats to Validity

This section describes threats to validity and the efforts made to mitigate their effects.

Internal Validity. We have made an effort to determine whether the performance gains of SPLVD are attributable to self-paced learning. To eliminate the difference from dataset splits, we predefine and reuse the same train/validation/test partition (ratio 8:1:1) across all experiments. We train and evaluate each baseline using its officially recommended hyperparameters (e.g., learning

rate, batch size) and use grid search to select the hyperparameters in SPLVD to reduce the influence of parameter selection on vulnerability detection. When we explore the improvement effect of self-paced learning, we also conducted experiments using multiple pre-trained models to prevent threats to validity caused by experiments on a single model.

Construct Validity. We control construct validity via careful selection of datasets and evaluation metrics. For dataset selection, we use three widely adopted open-source datasets (BigVul, Devign, and Reveal) based on the datasets chosen by several baseline approaches and further collect data from OpenHarmony to improve industrial representativeness. Some constructed datasets (e.g., NVD [45]) are not selected because the purpose of SPLVD is to improve the usability of vulnerability detection models; therefore, open-source datasets are more capable of testing the performance of SPLVD. For evaluation metrics, we select the five most common metrics in vulnerability detection. In the ablation study, to further compare the improvement in the detection performance of the model on high-confidence source code achieved through self-paced learning, we introduce the evaluation metric Top_NF1 . To maintain the completeness of VulGPT, we select the two best approaches proposed within it as baselines. In addition, in RQ2 we do not compare SPLVD with Starcoder2. This is mainly because its detection results on BigVul are poor, with an F1 of 0 in many CWE categories, and thus lacks comparability.

External Validity. To test the generalization capability of SPLVD, our experiments are conducted on three well-known open-source datasets in vulnerability detection and a new one—OpenHarmony. When collecting OpenHarmony, we selected the key components repositories with a large scale (with 1K+ forks). We conducted experiments using various C/C++ datasets of different sizes (ranging from large to medium/small), with varying levels of class balance (balanced and unbalanced) and different project types. This diverse evaluation demonstrated the stability of SPLVD under different data conditions and proved its strong generalization capability. We have disclosed the data and materials used to develop and evaluate SPLVD to support its continuous improvement.

9 Conclusion

In this article, we propose SPLVD, a new approach to address the issue of low-quality data in software vulnerability detection. SPLVD dynamically quantifies source code difficulty to achieve progressive learning and effectively reduces interference from noisy data. SPLVD combines the pre-trained model UniXcoder with an LSTM classifier and designs a dynamic age parameter function based on training state feedback. Experimental results on three well-known datasets demonstrate that SPLVD outperforms existing approaches. Moreover, we built a real-world dataset using source code from OpenHarmony to evaluate SPLVD further and confirm the detection results with security experts. In summary, our work improves the practicality and reliability of vulnerability detection models through self-paced learning. In the future, we plan to investigate more definitions of source code difficulty and validate SPLVD in programming languages other than C++.

References

- [1] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum Learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, 41–48.
- [2] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. IEEE, 30–39.
- [3] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering*. IEEE, 1456–1468.
- [4] Partha Chakraborty, Krishna Kanth Arumugam, Mahmoud Alfadel, Meiyappan Nagappan, and Shane McIntosh. 2024. Revisiting the performance of deep learning-based vulnerability detection on realistic datasets. *IEEE Transactions on*

- Software Engineering* 50, 8 (2024), 2163–2177.
- [5] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48, 9 (2022), 3280–3296.
 - [6] Junfan Chen, Richong Zhang, Xiaohan Jiang, and Chunming Hu. 2024. SPContrastNet: A self-paced contrastive learning model for few-shot text classification. *ACM Transactions on Information Systems* 42, 5 (2024), 130:1–130:25.
 - [7] Junfan Chen, Richong Zhang, Jiarui Wang, Chunming Hu, and Yongyi Mao. 2024. Self-paced pairwise representation learning for semi-supervised text classification. In *Proceedings of the ACM Web Conference 2024*. ACM, 4352–4361.
 - [8] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology* 30, 3 (2021), 38:1–38:33.
 - [9] Xiao Cheng, Haoyu Wang, Jiayi Hua, Miao Zhang, Guoai Xu, Li Yi, and Yulei Sui. 2019. Static detection of control-flow-related vulnerabilities using graph embedding. In *Proceedings of the 24th International Conference on Engineering of Complex Computer Systems*. IEEE, 41–50.
 - [10] Roland Croft, M. Ali Babar, and M. Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*. IEEE, 121–133.
 - [11] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2018. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering* 47, 1 (2018), 67–85.
 - [12] Charlie Dil, Hui Chen, and Kostadin Damevski. 2025. Towards higher quality software vulnerability data using LLM-based patch filtering. *Journal of Systems and Software* (2025), 112581:1–12.
 - [13] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624* (2024).
 - [14] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th IEEE/ACM International Conference on Mining Software Repositories*. ACM, 508–512.
 - [15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*. ACL, 1536—1547.
 - [16] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based line-level vulnerability prediction. In *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories*. IEEE, 608–620.
 - [17] Gitee. 2025. OpenHarmony. <https://gitee.com/openharmony>.
 - [18] Katerina Goseva-Popstojanova and Andrei Perhinschi. 2015. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology* 68 (2015), 18–33.
 - [19] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-Modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. ACL, 7212–7225.
 - [20] Hazim Hanif and Sergio Maffei. 2022. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International Joint Conference on Neural Networks*. IEEE, 1–8.
 - [21] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories*. ACM, 596–607.
 - [22] Arthur E Hoerl and Robert W Kennard. 1970. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics* 12, 1 (1970), 55–67.
 - [23] Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*. OpenReview.net, 1–20.
 - [24] Ridhi Jain, Nicole Gervasoni, Mthandazo Ndhlovu, and Sanjay Rawat. 2023. A code centric evaluation of C/C++ vulnerability datasets for deep learning based vulnerability detection techniques. In *Proceedings of the 16th Innovations in Software Engineering Conference*. ACM, 6:1–6:10.
 - [25] Julian Jang-Jaccard and Surya Nepal. 2014. A survey of emerging threats in cybersecurity. *J. Comput. System Sci.* 80, 5 (2014), 973–993.
 - [26] Arnold Johnson, Kelley Dempsey, Ron Ross, Sarbari Gupta, and Dennis Bailey. 2011. Guide for security-focused configuration management of information systems. *NIST Special Publication 800* (2011), 1–99.
 - [27] Ilias Kalouptsoglou, Miltiadis Siavvas, Apostolos Ampatzoglou, Dionysios Kehagias, and Alexander Chatzigeorgiou. 2023. Software vulnerability prediction: A systematic mapping study. *Information and Software Technology* 164 (2023), 107303:1–18.

- [28] Ilias Kalouptsoglou, Miltiadis Siavvas, Apostolos Ampatzoglou, Dionysios Kehagias, and Alexander Chatzigeorgiou. 2024. Vulnerability classification on source code using text mining and deep learning techniques. In *Proceedings of the 24th IEEE International Conference on Software Quality, Reliability, and Security Companion*. IEEE, 47–56.
- [29] Ilias Kalouptsoglou, Miltiadis Siavvas, Apostolos Ampatzoglou, Dionysios Kehagias, and Alexander Chatzigeorgiou. 2025. Transfer learning for software vulnerability prediction using Transformer models. *Journal of Systems and Software* 227 (2025), 112448:1–16.
- [30] M Kumar, Benjamin Packer, and Daphne Koller. 2010. Self-paced learning for latent variable models. In *Annual Conference on Neural Information Processing Systems 23*. MIT press, 1189–1197.
- [31] Xiao-Li Li, Bing Liu, and See Kiong Ng. 2010. Negative training data can be harmful to text classification. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*. ACL, 218–228.
- [32] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.
- [33] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).
- [34] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software vulnerability detection using deep neural networks: A survey. *Proc. IEEE* 108, 10 (2020), 1825–1848.
- [35] Guanjun Lin, Wei Xiao, Jun Zhang, and Yang Xiang. 2020. Deep learning-based vulnerable function detection: A benchmark. In *Proceedings of the 21st International Conference on Information and Communications Security*. Springer, 219–232.
- [36] Lili Liu, Zhen Li, Yu Wen, and Penglong Chen. 2022. Investigating the impact of vulnerability datasets on deep learning-based vulnerability detectors. *PeerJ Computer Science* 8 (2022), 1–22.
- [37] Shigang Liu, Guanjun Lin, Qing-Long Han, Sheng Wen, Jun Zhang, and Yang Xiang. 2019. DeepBalance: Deep-learning and fuzzy oversampling for vulnerability detection. *IEEE Transactions on Fuzzy Systems* 28, 7 (2019), 1329–1343.
- [38] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations 2019*. OpenReview.net, 1–18.
- [39] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtiar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173* (2024).
- [40] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software* 212 (2024), 112–236.
- [41] MITRE. 2025. CVE: Common Vulnerabilities and Exposures. <https://www.cve.org/>.
- [42] MITRE. 2025. CWE - Common Weakness Enumeration. <https://cwe.mitre.org>.
- [43] Stephan Neuhaus and Thomas Zimmermann. 2010. Security trend analysis with CVE topic models. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*. IEEE, 111–120.
- [44] Xu Nie, Ningke Li, Kailong Wang, Shangguang Wang, Xiapu Luo, and Haoyu Wang. 2023. Understanding and tackling label errors in deep learning-based vulnerability detection (experience paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 52–63.
- [45] NIST. 2025. NVD - Home. <https://nvd.nist.gov/>.
- [46] Moumita Das Purba, Arpita Ghosh, Benjamin J Radford, and Bill Chu. 2023. Software vulnerability detection using large language models. In *Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 112–119.
- [47] Fangcheng Qiu, Zhongxin Liu, Xing Hu, Xin Xia, Gang Chen, and Xinyu Wang. 2024. Vulnerability detection via multiple-graph-based code representation. *IEEE Transactions on Software Engineering* 50, 8 (2024), 2178–2199.
- [48] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications*. IEEE, 757–762.
- [49] Statista. 2025. Number of common IT security vulnerabilities and exposures 2025. <https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures/>.
- [50] Benjamin Steenhoeck, Hongyang Gao, and Wei Le. 2024. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. IEEE, 1–13.
- [51] Benjamin Steenhoeck, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An empirical study of deep learning models for vulnerability detection. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*. IEEE, 2237–2248.

- [52] Benjamin Steenhoek, Kalpathy Sivaraman, Renata Saldivar Gonzalez, Yevhen Mohylevskyy, Roshanak Zilouchian Moghaddam, and Wei Le. 2024. Closing the Gap: A User Study on the Real-world Usefulness of AI-powered Vulnerability Detection & Repair in the IDE. *arXiv preprint arXiv:2412.14306* (2024).
- [53] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-based language models for software vulnerability detection. In *Proceedings of the 38th ACM Annual Computer Security Applications Conference*. ACM, 481–496.
- [54] Nora Tomas, Jingyue Li, and Huang Huang. 2019. An empirical study on culture, automation, measurement, and sharing of devsecops. In *Proceedings of the 2019 International Conference on Cyber Security and Protection of Digital Services*. IEEE, 1–8.
- [55] Jonathan G Tullis and Aaron S Benjamin. 2011. On the effectiveness of self-paced learning. *Journal of Memory and Language* 64, 2 (2011), 109–118.
- [56] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25 (2020), 1419–1457.
- [57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Annual Conference on Neural Information Processing Systems 30*. MIT press, 5998–6008.
- [58] Xin Wang, Yudong Chen, and Wenwu Zhu. 2021. A survey on curriculum learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 9 (2021), 4555–4576.
- [59] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. ACL, 8696–8708.
- [60] We. 2025. Replication Package. <https://figshare.com/s/bef3211194fc18fe375e>.
- [61] Wikipedia. 2025. Stuxnet. <https://en.wikipedia.org/wiki/Stuxnet>.
- [62] Yanjing Yang, Min Gao, Yuerang Li, Fan Wu, Jia Wang, and Quanwu Zhao. 2021. MSPLD: Shilling attack detection model based on meta self-paced learning. In *2021 International Joint Conference on Neural Networks*. IEEE, 1–8.
- [63] Yanjing Yang, Xin Zhou, Runfeng Mao, Jinwei Xu, Lanxin Yang, Yu Zhang, Haifeng Shen, and He Zhang. 2025. Dlap: A deep learning augmented large language model prompting framework for software vulnerability detection. *Journal of Systems and Software* 219 (2025), 112234:1–15.
- [64] Jingxiu Yao and Martin Shepperd. 2020. Assessing software defect prediction performance: Why using the Matthews correlation coefficient matters. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 120–129.
- [65] Zixuan Yuan, Hao Liu, Haoyi Zhou, Denghui Zhang, Xiao Zhang, Hao Wang, and Hui Xiong. 2024. Self-paced unified representation learning for hierarchical multi-label classification. In *Proceedings of the AAAI Conference on Artificial Intelligence 2024*. AAAI, 16623–16632.
- [66] Chenkang Zhang, Lei Luo, and Bin Gu. 2023. Denoising multi-similarity formulation: A self-paced curriculum-driven approach for robust metric learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. AAAI, 11183–11191.
- [67] Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shaping Li. 2023. Vulnerability detection by learning from syntax-based execution paths of code. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4196–4212.
- [68] Di Zhao, Yun Sing Koh, Gillian Dobbie, Hongsheng Hu, and Philippe Fournier-Viger. 2024. Symmetric self-paced learning for domain generalization. In *Proceedings of the AAAI Conference on Artificial Intelligence 2024*. AAAI, 16961–16969.
- [69] Wei Zheng, Jialiang Gao, Xiaoxue Wu, Fengyu Liu, Yuxing Xun, Guoliang Liu, and Xiang Chen. 2020. The impact factors on the performance of machine learning-based vulnerability detection: A comparative study. *Journal of Systems and Software* 168 (2020), 110659:1–12.
- [70] Weining Zheng, Yuan Jiang, and Xiaohong Su. 2021. Vu1SPG: Vulnerability detection based on slice property graph representation learning. In *Proceedings of 32nd IEEE International Symposium on Software Reliability Engineering*. IEEE, 457–467.
- [71] Xin Zhou, Ting Zhang, and David Lo. 2024. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, 47–51.
- [72] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Annual Conference on Neural Information Processing Systems 32*, Vol. 32. MIT Press, 10197–10207.