

SVA-ICL: Improving LLM-based Software Vulnerability Assessment via In-Context Learning and Information Fusion

Chaoyang Gao^a, Xiang Chen^{a,b,*}, Guangbei Zhang^a

^a*School of Artificial Intelligence and Computer Science, Nantong University, Nantong, China*

^b*State Key Lab. for Novel Software Technology, Nanjing University, Nanjing, China*

Abstract

Context: Software vulnerability assessment (SVA) is critical for identifying, evaluating, and prioritizing security weaknesses in software applications.

Objective: Despite the increasing application of large language models (LLMs) in various software engineering tasks, their effectiveness in SVA remains underexplored.

Method: To address this gap, we introduce a novel approach SVA-ICL, which leverages in-context learning (ICL) to enhance LLM performance. Our approach involves the selection of high-quality demonstrations for ICL through information fusion, incorporating both source code and vulnerability descriptions. For source code, we consider semantic, lexical, and syntactic similarities, while for vulnerability descriptions, we focus on textual similarity. Based on the selected demonstrations, we construct context prompts and consider DeepSeek-V2 as the LLM for SVA-ICL.

Results: We evaluate the effectiveness of SVA-ICL using a large-scale dataset comprising 12,071 C/C++ vulnerabilities. Experimental results demonstrate that SVA-ICL outperforms state-of-the-art SVA baselines in terms of Accuracy, F1-score, and MCC measures. Furthermore, ablation studies highlight the significance of component customization in SVA-ICL, such as the number of demonstrations, the demonstration ordering strategy, and the optimal fusion ratio of different modalities.

Conclusion: Our findings suggest that leveraging ICL with information fusion can effectively improve the effectiveness of LLM-based SVA, warranting further research in this direction.

Keywords: Software Vulnerability Assessment; Large Language Model; In-Context Learning; Information Fusion.

1. Introduction

Software vulnerabilities are weaknesses or flaws in software applications that attackers can exploit, leading to unauthorized access, data breaches, financial losses, operational disruptions, reputation damage, and legal consequences. These vulnerabilities can arise from coding errors, design flaws, and configuration mistakes, and their exploitation can significantly impact organizations and users. To address these risks, software vulnerability assessment (SVA) [1, 2] is employed to systematically identify, evaluate, and prioritize security weaknesses within software applications. The primary purpose of SVA is to enhance the security posture of software by proactively detecting and mitigating potential threats, thereby reducing the likelihood of successful attacks, ensuring compliance with regulatory standards, and building trust among users and stakeholders. Recently, researchers primarily used either source code or vulnerability descriptions for SVA. For example, Liu et al. [3] used deep learning methods to classify vulnerability texts and then assessed the severity levels of the vulnerabilities. Le et

al. [4] resorted to vulnerable statements in the code that could be used for software vulnerability assessment.

Large language models (LLMs) have achieved significant success in various software engineering tasks, including code generation [5, 6], vulnerability detection [7, 8], and source code summarization [5, 9]. LLMs excel in software engineering tasks due to their ability to harness extensive training, natural language processing capabilities, pattern recognition skills, contextual awareness, and adaptability. Despite their success in these tasks, to the best of our knowledge, the application of LLMs to the SVA task has not been thoroughly investigated in previous studies. However, applying LLMs to SVA presents the following challenges. First, in-context learning (ICL) has garnered attention for its excellent performance and ease of use without the need for fine-tuning [10]. However, ICL requires the LLM to effectively understand and utilize contextual information from a limited set of demonstrations. Second, selecting high-quality demonstrations as contextual information for prompt template construction is challenging, as it involves identifying examples that are both relevant and representative of the vulnerabilities being assessed. Finally, previous studies usually considered either vulnerability code or vulnerability descriptions [4, 11, 12], whereas considering both modalities may allow for a more accurate selection of similar demonstration examples, necessitating an effective information fusion method.

*Corresponding author

Email addresses: gcyol@outlook.com (Chaoyang Gao), xchen@ntu.edu.cn (Xiang Chen), guangbei0324@gmail.com (Guangbei Zhang)

Moreover, it is necessary to better evaluate the similarity between the source code and vulnerability descriptions.

To alleviate these challenges, we propose SVA-ICL, an approach that leverages the in-context learning capabilities of large language models and fuses information from both source code and vulnerability descriptions. We choose ICL because of its ability to be flexibly applied across different tasks without the need for explicit fine-tuning of the large language models. Additionally, ICL can effectively learn and handle tasks using contextual information, even in few-shot scenarios, providing crucial support for improving the adaptability and accuracy of the SVA task. Specifically, SVA-ICL develops a customized prompt template for SVA. To select high-quality demonstration examples, we employ information fusion, considering the semantic, lexical, and syntactic similarities of the source code, as well as text similarities for vulnerability descriptions. By combining these similarities, we ensure that the selected demonstration examples are highly relevant to the target vulnerabilities, thereby constructing high-quality context prompts.

We evaluate the effectiveness of SVA-ICL using a dataset containing 12,071 entries related to vulnerabilities. In our constructed dataset, we consider the v3 standard of the Common Vulnerability Scoring System (CVSS), which was recently developed for assessing software vulnerability severity more reasonably. Compared to the CVSS V2 standard, the CVSS V3 standard can provide a more comprehensive, flexible, and accurate framework for assessing and prioritizing vulnerabilities.

We use DeepSeek-V2 [13] as the LLM, based on its outstanding performance in handling complex language tasks. It excels not only in processing code information but also shows significant improvements in context window length and processing capability compared to other large language models (such as GPTs). This enables it to better handle long text inputs that include extensive source code and vulnerability descriptions, further enhancing the overall performance of software vulnerability assessment.

Experimental results show that SVA-ICL outperforms baseline approaches, increasing Accuracy, F1-score, and MCC performance measures by at least 7.68 percentage points (pp), 5.34 pp, and 13.59 pp, respectively. Ablation studies demonstrate the effectiveness of our component settings, such as the number of demonstrations, the demonstration ordering strategy, and the ratio of different information modalities. Our findings indicate that the optimal number of demonstration examples is four, with an ascending order of similarity and a source code to vulnerability description ratio of 70% to 30%. These settings help SVA-ICL achieve the best performance. Additionally, multiple random splits of the dataset confirm our approach’s performance improvement consistency when compared with baselines. Finally, we also show the competitiveness of using DeepSeek-V2 as the LLM when compared to other popular LLMs (such as GPT-3.5 Turbo and GPT-4o).

Our findings suggest that leveraging LLMs for SVA can significantly enhance performance compared to state-of-the-art baselines.

These promising results highlight the capabilities of LLMs in improving software security and underscore the need for fur-

ther research in this domain. Therefore, we call for more studies to explore and refine LLM-based approaches for the SVA task and similar vulnerability-related studies.

The novelty and contributions of our study can be summarized as follows:

- **Perspective.** We are the first to utilize Large Language Models (LLMs) for software vulnerability assessment, introducing a novel approach SVA-ICL.
- **Approach.** Our proposed SVA-ICL employs ICL to enhance the LLM’s understanding of the task, utilizing a meticulously designed prompt template. By comprehensively considering both source code and vulnerability descriptions through information fusion, we select high-quality demonstration examples. For source code, we consider semantic, lexical, and syntactic similarities, while for vulnerability descriptions, we focus on text similarity.
- **Dataset.** We constructed a large-scale dataset (i.e., containing 12,071 entries of source code and vulnerability descriptions) and considered the CVSS v3 standard while previous SVA studies only considered the CVSS V2 standard.
- **Practical Evaluation.** We evaluate SVA-ICL on our constructed dataset. The results indicate that SVA-ICL can outperform these baselines. Additionally, ablation experiments confirm the effectiveness of our customization settings in demonstration selection.

Open Science. To facilitate replication and further research, we share our dataset and source code on GitHub (<https://github.com/judeomg/SVA-ICL>).

Paper Organization. Section 2 introduces the research background and challenges of applying LLMs to SVA. Section 3 shows the framework of our proposed approach SVA-ICL and its details. Section 4 describes our experimental setup, including research questions, experimental subjects, baselines, performance measures, and implementation details. Section 5 shows our experimental results and main findings. Section 6 discusses the impact of different large language models, the influence of different data splits, the influence of similarity values on the assessment accuracy, the potential impact of the data leakage issue, and potential threats to our study. Section 7 summarizes related work and emphasizes our novelty. Section 8 summarizes our study and discusses potential future directions.

2. Background

In this section, we first introduce the background of software vulnerability assessment. Then we introduce the large language model and in-context learning. Finally, we analyze the research challenges of applying LLMs to the SVA task.

2.1. Software Vulnerability Assessment

Software vulnerability assessment involves rating the severity of vulnerabilities based on their characteristics, which means calculating a risk score for each vulnerability according to scoring rules and determining its severity based on that score [1, 14]. SVA is critical because it helps organizations prioritize their efforts in addressing security risks. Identifying and accurately assessing the severity of vulnerabilities enables efficient allocation of resources to mitigate the most critical threats, thereby protecting systems and data from potential breaches. Inaccurate assessment could result in overlooking high-risk vulnerabilities or over-prioritizing low-risk ones, leading to inefficient use of security resources.

Assessing vulnerabilities manually is challenging due to the huge volume of vulnerabilities that modern software systems can contain. Manual assessment requires significant expertise and time, which can lead to inconsistencies and errors. Developers might also struggle with maintaining up-to-date knowledge of the ever-evolving landscape of security threats. The complexity and variability of software environments further complicate the manual assessment, making automation a valuable approach.

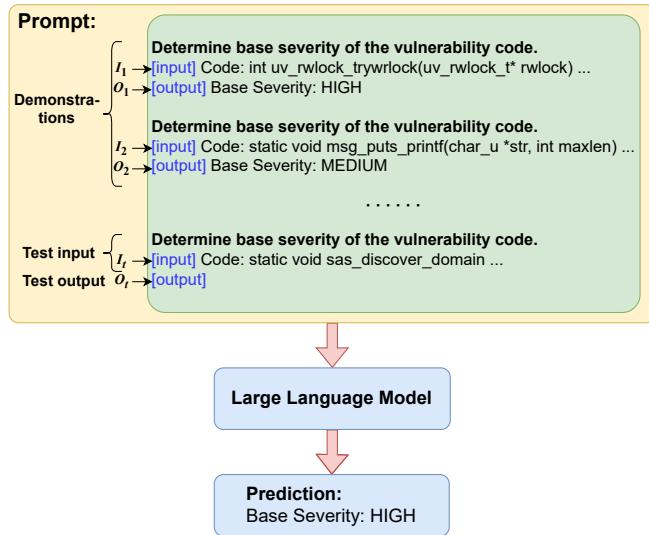


Figure 1: An example of using in-context learning for software vulnerability assessment

The Common Vulnerability Scoring System (CVSS) [15] is a standardized approach for measuring the severity of software vulnerabilities. It assesses vulnerabilities in terms of multiple dimensions (e.g., impact, exploitability) and provides a scoring system that offers organizations and researchers a standard for software vulnerability assessment. CVSS has been updated multiple times, with CVSS v3 being the most widely used version. In previous SVA studies, researchers mainly used CVSS v2 [16, 17] for SVA. However, compared to CVSS v2, CVSS v3 provides a more comprehensive, flexible, and accurate framework for assessing and prioritizing vulnerabilities, such as improved granularity, better exploitability metrics, and enhanced impact metrics. Therefore, we consider CVSS v3 in our con-

structed SVA dataset. In the realm of software vulnerability assessment, reliable and comprehensive data sources are crucial for both research and practical applications. One such valuable resource is the National Vulnerability Database (NVD) [4, 18], a public database maintained by the National Institute of Standards and Technology (NIST). NVD serves as a critical repository, offering detailed information on software vulnerabilities, including descriptions, affected products and versions, patch information, and related reference links. In our research, we primarily use source code and vulnerability description information from NVD.

2.2. Large Language Model and In-Context Learning

Large Language Models (LLMs) are natural language processing models trained on large-scale text data, learning rich language knowledge and contextual relationships [19, 20]. In recent years, LLMs have achieved significant performance in various software engineering tasks, such as code generation [5, 6], code completion, source code summarization [5, 9], source code repair [21], and vulnerability detection [7, 8]. However, to the best of our knowledge, LLMs have not yet been used for software vulnerability assessment.

In-context learning (ICL) is an approach that allows models to learn and handle tasks by providing demonstration examples without explicitly fine-tuning model parameters [10, 22]. The high flexibility, efficiency, and generality of ICL have attracted many researchers and can be extended to specific software engineering tasks. Figure 1 illustrates an example of using in-context learning for software vulnerability assessment. In detail, we can use I_i, O_i, I_i^*, O_i^* , and N to represent the initial input, initial output, reconstructed input, reconstructed output, and the number of demonstration examples, respectively. Here, the reconstructed input represents the initial input after optimization through the prompt template, and the reconstructed output is the corresponding output from the large language model. In this way, we can represent the initial set of N demonstration examples as a collection $\{(I_1, O_1), (I_2, O_2), \dots, (I_N, O_N)\}$. Subsequently, we can fill these initial demonstration examples into a carefully designed prompt template, resulting in reconstructed demonstration examples $\{(I_1^*, O_1^*), (I_2^*, O_2^*), \dots, (I_N^*, O_N^*)\}$. Typically, the number N is fewer than 50 and is far less than the number of training examples required for fine-tuning methods [23, 24]. Therefore, in-context learning is also known as few-shot learning. Specifically, when N is 0, it is referred to as zero-shot learning. We can represent the reconstructed demonstration examples as $D = I_1^* \| O_1^* \| I_2^* \| O_2^* \| \dots \| I_N^* \| O_N^*$ and further concatenate the test example I_t^* with the demonstration examples to obtain the LLM's input prompt $P = D \| I_t^*$, where $\|$ denotes the literal concatenation operation. Finally, the LLM can use the input prompt P to predict the test example's label O_t .

2.3. Research Challenges

However, directly applying ICL to SVA presents several challenges. First, it is necessary to select high-quality demonstrations. Previous studies [4, 16] typically consider either the

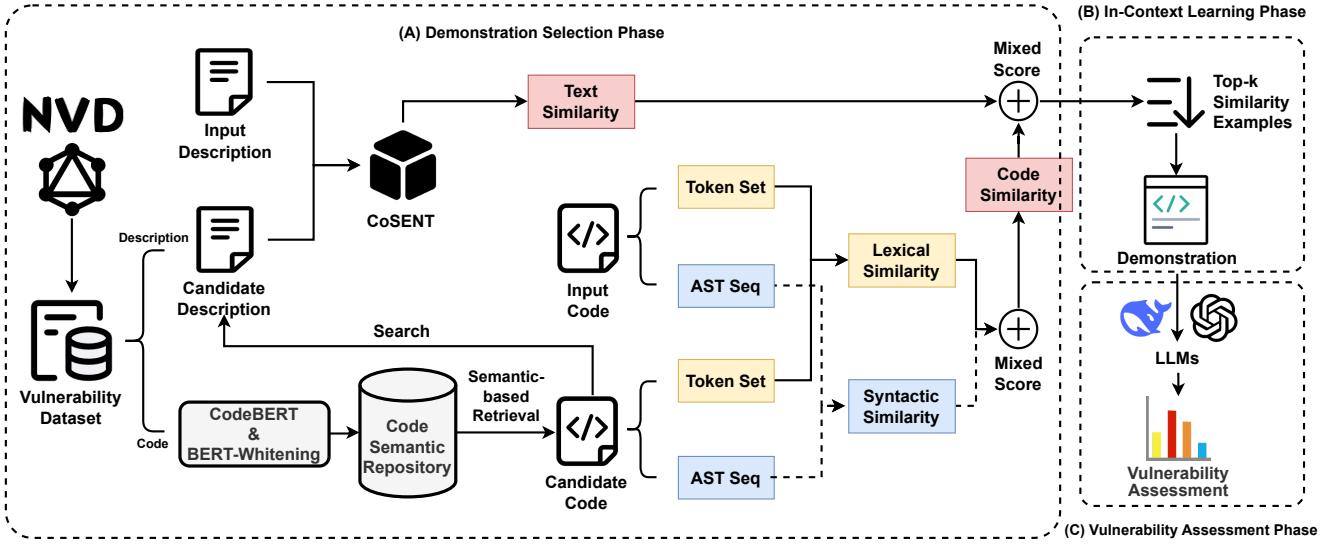


Figure 2: Framework of our proposed approach SVA-ICL

source code or the vulnerability description. However, in practice, considering both types of information simultaneously could be more effective, necessitating the design of efficient information fusion methods. The rationale evaluation of this information fusion setting can be found in Section 5.2. Additionally, for different modalities, it is crucial to develop corresponding similarity evaluation methods for source code and vulnerability description, respectively. Finally, it is important to analyze the impact of the number of demonstration examples and different ordering strategies for ICL on the performance of the LLM-based SVA approach.

3. Approach

Figure 2 shows the framework of our proposed approach SVA-ICL. The framework of SVA-ICL mainly contains three stages. Specifically, in the **demonstration selection phase**, SVA-ICL selects the top- k vulnerabilities as the demonstration examples from the historical repository by using information fusion to simultaneously consider both the source code and the vulnerability description. In the **in-context learning phase**, SVA-ICL inputs the retrieved top- k vulnerabilities with their source code and vulnerability descriptions into our carefully designed prompting templates. Finally, in the **vulnerability assessment phase**, the input consists of the source code and its corresponding vulnerability description of the target vulnerability, and the output is the corresponding vulnerability severity level generated by the LLM. In the remaining part of this section, we show the details of these three stages.

3.1. Demonstration Selection Phase

Recent research [23] on in-context learning for code intelligence tasks (such as source code summarization, bug fixing, and program synthesis) has shown that the performance of ICL depends largely on the quality of demonstration examples.

Therefore, carefully designed strategies for selecting demonstration examples from the historical repository can effectively improve the performance of LLM in specific domain tasks [23]. For the SVA task, to retrieve high-quality demonstration examples, we need to measure the similarity between different vulnerabilities. Some previous SVA studies only consider the vulnerability description, such as the multi-task learning method proposed by Babalau et al. [11]. Other previous SVA studies only consider the source code, such as the deep learning model DeepCVA proposed by Le et al. [17]. However, we conjecture that considering only one type of vulnerability information is not comprehensive enough. Therefore, when calculating similarity, it is necessary to consider both types of vulnerability information by leveraging information fusion. Specifically, given a target vulnerability and the vulnerability from the historical repository, we first calculate the similarity based on the source code and then calculate the similarity based on the vulnerability description. Finally, we fuse the similarities of these two modalities.

Since how to accurately calculate code similarity is still an open problem for semantic-based software analysis, in this study, we consider an information retrieval-based method proposed by our previous study [25] for calculating code similarity, which was successfully used in the LLM-based smart contract code comment generation task [26]. This method simultaneously considers the semantic information, lexical information, and syntactic information of code snippets. Since SVA-ICL further considers vulnerability description for demonstration selection, we design the demonstration selection strategy by modifying our previously proposed information retrieval-based method, which can further consider the text similarity of vulnerability descriptions.

Specifically, our demonstration selection strategy can be divided into three parts: (1) **Semantic-based retrieval part:** We extract semantic information from vulnerable source code using CodeBERT [27] and BERT-whitening [28]. Then, we retrieve

the most similar source code from a historical dataset as candidates. (2) **Syntax and Lexicon-based retrieval part:** The first part only considers the semantic information of vulnerability codes, so in the second part, we compute the lexical and syntactic similarity of these candidate snippets, then combine the similarity of these two parts as the code similarity between the target vulnerability code snippet and the candidate vulnerability code snippet. (3) **Text-based retrieval part:** We retrieve candidate source code from the historical dataset obtained in the first part. In our dataset, each vulnerability code snippet corresponds to a vulnerability description. Therefore, we can find the corresponding vulnerability description based on the candidate source code. Then, we use the CoSENT model implemented based on Text2vec [29] to calculate the text similarity between the target vulnerability description and the candidate vulnerability description. Finally, we combine the code similarity and text similarity to obtain the top- k similar vulnerabilities, determining the final demonstration example. In the rest of this subsection, we provide detailed information for these three parts.

3.1.1. Semantic-based retrieval part

This part involves splitting the vulnerability code snippets in the historical dataset according to the CamelCase naming convention, then inputting the resulting sequences $\{X_i\}_{i=1}^N$ into CodeBERT [26, 30] to obtain semantic vectors $X_i \in \mathbb{R}^D$, where N represents the number of source code in the historical dataset, and D represents the hidden dimension. Subsequently, BERT-whitening is used to improve the isotropy of the representations and reduce the dimension from D to d , resulting in $\{\tilde{X}_i\}_{i=1}^N$. By reducing redundancy, shortening training time, and handling noisy data more effectively, using BERT-whitening can improve the quality of embeddings generated by CodeBERT [31, 32]. Finally, the semantic similarity between the code embeddings \tilde{X}_a and \tilde{X}_b is calculated using the L2 distance, with the following formula.

$$\text{SemSim}(\tilde{X}_a, \tilde{X}_b) = \sum_{i=1}^d (\tilde{X}_a[i] - \tilde{X}_b[i])^2 \quad (1)$$

3.1.2. Syntax and Lexicon-based retrieval part

In the first part, we retrieve the top- n most similar candidate source code from the historical dataset based on semantic similarity. In this part, we further consider the syntactic and lexical information of the vulnerable source code. Hence, we compute the syntactic and lexical similarity between the target vulnerability code and the vulnerability codes in the historical dataset. Specifically, we first employ the tree-sitter library¹ to generate the AST (Abstract Syntax Tree) sequences of the source code. Considering the lengthy nature of our vulnerability codes, we reference the method SimSBT [33] to optimize the representation of the AST sequences. Lastly, we utilize code tokens to calculate lexical similarity and employ the AST sequences to compute syntactic similarity. Finally, combining these two similarity values to derive the final source code similarity. This approach comprehensively considers various aspects of the code,

¹<https://github.com/tree-sitter/tree-sitter>

facilitating the retrieval of source code most similar to the target vulnerability code. Notice that the reason for prioritizing the calculation of semantic similarity before the calculation of syntactic and lexical similarity is its significant improvement in retrieval quality shown in previous studies [25, 34].

For two source code snippets A and B , along with their corresponding AST sequences \tilde{A} and \tilde{B} , we can calculate the syntactic similarity using the following formula.

$$\text{SynSim}(A, B) = \frac{\text{sum}(\text{len}(\tilde{A}), \text{len}(\tilde{B})) - \text{lev}}{\text{sum}(\text{len}(\tilde{A}), \text{len}(\tilde{B}))} \quad (2)$$

where lev represents the Levenshtein distance [35] between \tilde{A} and \tilde{B} .

Concerning lexical similarity, we treat source code as a set of tokens. For two token sets set_A and set_B , we determine lexical similarity using the Jaccard similarity [36].

$$\text{LexSim}(A, B) = \frac{|\text{set}_A \cap \text{set}_B|}{|\text{set}_A \cup \text{set}_B|} \quad (3)$$

After calculating syntactic similarity and lexical similarity, we can fuse these two similarity values by the following formula and use it as the code similarity between the target vulnerability code snippet and the candidate source code.

$$\begin{aligned} \text{CodeSim}(A, B) &= \lambda \times \text{SynSim}(A, B) \\ &\quad + (1 - \lambda) \times \text{LexSim}(A, B) \end{aligned} \quad (4)$$

where λ is a parameter, which is used to adjust the weight between different similarity values. In our previous studies [25, 26], we show the effectiveness of the setting rationality by designing a set of ablation studies when compared with various variants of this information retrieval-based method.

3.1.3. Text-based retrieval part

In the first part, we can retrieve the top- n most similar source code snippets in the historical dataset. For each retrieved vulnerability, we can further consider the corresponding vulnerability description and calculate the text similarity between them. In our study. We utilize the CoSENT model provided by the text2vec-based library similarities². This model can compute the similarity between vulnerability descriptions and is referred to as *TextSim*. We assume that for the vulnerability V_a , the vulnerability description of the source code A is denoted as D_a . For the vulnerability V_b , the vulnerability description of the source code B is denoted as D_b . Then we can compute the similarity of different vulnerabilities using the following formula, which can further select the top- k most similar vulnerabilities in the historical datasets by considering both source code and their respective vulnerability description.

$$\begin{aligned} \text{Sim}(V_a, V_b) &= \phi \times \text{CodeSim}(A, B) \\ &\quad + (1 - \phi) \times \text{TextSim}(D_a, D_b) \end{aligned} \quad (5)$$

where ϕ is a parameter used to adjust the weight between source code similarity and vulnerability description similarity.

²<https://github.com/shibing624/similarities>

3.2. In-context Learning Phase

After retrieving high-quality demonstrations, we move on to designing the prompt template. The design of the prompt template is a crucial part of ICL, as high-quality templates can guide the large language model to output high-quality content, ensuring the accuracy and reliability of the information. Since we are the first to apply ICL to the SVA task, we mainly refer to the prompt templates designed in previous source code summarization studies [23, 26]. Our designed prompt template is shown in Figure 3. Specifically, our prompt template consists of three parts. The first part is the **natural language prompt**, where we succinctly instruct the Large Language Model (LLM) to assess the severity of the target vulnerability based on the provided source code and vulnerability descriptions. To avoid LLM generating irrelevant redundant content, we emphasize instructing LLM to output only the base severity of the target vulnerability without additional explanations. In the **demonstration part**, we combine demonstration examples of source code and their corresponding vulnerability descriptions following a specific structure. First, we inform LLM to evaluate the severity of the vulnerability based on the source code and vulnerability description. Next, we use “Demo i :” as the marker for the beginning of the i -th demonstration example. Then, we mark the start of input with “[Input]:” to indicate the user’s input. Within the input section, we use “Code:” as the prompt for the source code and “Description:” for the vulnerability description. Furthermore, we use “[Output]:” to signify the start of the output section, which indicates the ground truth of this vulnerability. This organizational structure helps LLM better comprehend the input prompts and generate suitable responses [23]. In the **test part**, we follow a similar prompt format as the demonstration part, with the distinction of changing “Demo i :” to “Test i :”. Additionally, after “[Output]”, no content is provided to clarify the specific output LLM needs to generate.

3.3. Vulnerability Assessment Phase

For the prompt template designed in Section 3.2, we can directly fill in the relevant content (such as the demonstration examples and the target vulnerability) into the template to generate the concrete prompt for LLM. Then, we can use the generated concrete prompt as input and call the API interface of the specific LLM. Finally, the API can output the severity level of the target vulnerability.

4. Experimental Setup

In this section, we first introduce our research questions and their design motivation. Then, we provide detailed information about the experimental subject, baselines, performance measures, and implementation details.

4.1. Research Questions

To show the competitiveness of SVA-ICL and the rationale of the component settings in SVA-ICL, we design the following five research questions (RQs) in our empirical study.

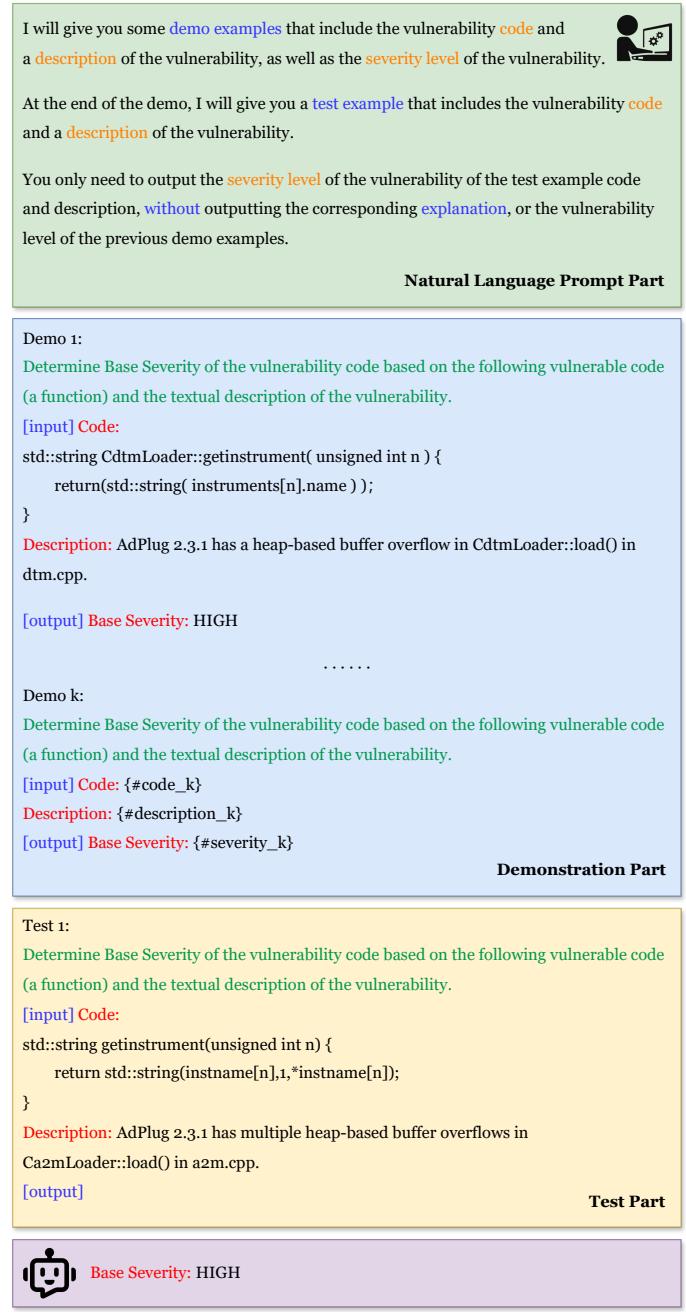


Figure 3: The prompt template designed by our proposed approach SVA-ICL.

RQ1: How does SVA-ICL perform in software vulnerability assessment compared to state-of-the-art baselines?

Motivation. In RQ1, we aim to analyze whether SVA-ICL can outperform state-of-the-art software vulnerability assessment baselines. Since our approach considers both source code and vulnerability descriptions. Therefore, when selecting SVA baselines, we consider both source code-based baselines and vulnerability description-based baselines. For example, for source code-based baselines, we consider baselines, such as Func_{RF} and Func_{LGBM} [4]. For vulnerability description-based baselines, we consider baselines, such as CWM_{NB}, CWM_{SVM}, and CWM_{LR} [16]. In addition, we also consider research base-

lines, such as Cvss-bert [37], SPSV [11], SVA-PT [38], and MTLM [39].

Since we model the SVA problem as a multi-class classification problem, we use widely adopted performance measures, such as Accuracy, Macro F1 score, and Matthews correlation coefficient, to evaluate the performance of SVA-ICL and SVA baselines.

RQ2: Whether considering both source code and vulnerability description with information fusion can improve the performance of SVA-ICL?

Motivation.

Previous software vulnerability assessment studies mainly focused on either source code or vulnerability descriptions [1, 4, 16, 17]. In our study, we conjecture that considering both types of modalities simultaneously can help to select higher-quality demonstrations. To validate this conjecture, we design an ablation study in RQ2 and try to find the optimal fusion ratio of these different modalities for SVA-ICL in our study.

RQ3: How does the number of demonstration examples in the prompt affect the performance of SVA-ICL?

Motivation. There are two reasons for designing RQ3. One reason is that existing work [26] shows that the number of demonstrations affects the effectiveness of ICL. The other reason is the context token limitation in LLM [40, 41]. If the input exceeds the token limitation, it will be truncated. Therefore, we want to investigate the impact of the number of demonstration examples on the performance of SVA-ICL.

RQ4: How do different demonstration ordering strategies affect the performance of SVA-ICL?

Motivation. In RQ3, we mainly focus on analyzing the performance impact of the demonstration number on SVA-ICL. However, previous work [23] indicates that the ordering strategy can also significantly affect performance. Therefore, in RQ 4, we aim to identify the optimal demonstration ordering strategy for SVA-ICL in our study.

RQ5: How does the fusion ratio of syntactic similarity and lexical similarity for source code similarity affect the performance of SVA-ICL?

Motivation. Although the effectiveness of customization of our proposed information retrieval-based method has been validated in the smart contract code comment generation task [25, 26], whether this customization is effective in the SVA task still needs investigation. Therefore, in RQ5, we mainly investigate the performance influence of source code similarity setting (i.e., the fusion ratio of syntactic similarity and lexical similarity) on SVA-ICL.

4.2. Experimental Subject

Our dataset is constructed using the raw data of MegaVul shared by Ni et al. [42]. The MegaVul dataset³ was last updated on April 14, 2024. This dataset was crawled from the Common Vulnerabilities and Exposures (CVE) database [43] and consists of 17,975 vulnerable data entries in C/C++ code.

It covers 176 different types of vulnerabilities from 1,062 open-source projects, spanning from 2006 to 2024. It not only ensures code integrity through advanced tools but also provides various information dimensions (such as function signatures, abstract functions, and code changes). This dataset supports multiple software security tasks, including software vulnerability detection and fixing.

The severity of vulnerabilities is typically determined according to the Common Vulnerability Scoring System (CVSS) standard, a free and open industry standard. In this study, we consider the CVSS v3 standard. Under this standard, the severity of vulnerabilities is divided into four levels⁴ according to the score values: Critical (9.0~10.0), High (7.0~8.9), Medium (4.0~6.9), and Low (0.1~3.9). Compared with CVSS v2 used by previous SVA studies [4, 16, 17], CVSS v3 provides a more refined assessment of vulnerability severity and risk. It includes additional sub-scores and metrics, as well as added temporal and environmental metrics, which more accurately reflect the actual impact of vulnerabilities.

After manual analysis, we found that the MegaVul dataset did not fully capture the severity ratings of all vulnerability scores according to the CVSS v3 standard. Therefore, we re-crawled the severity ratings and vulnerability scores of all vulnerabilities according to the CVSS v3 standard, along with their descriptions. After this step, we obtained 14,235 entries of vulnerability data following the CVSS v3 standard, and 3,740 entries without CVSS v3 information were removed. Then, we utilized the AST sequences of vulnerability code for syntactic similarity calculations. Specifically, we used the tree-sitter library⁵ to extract the AST sequence from the source code. We removed 2,164 entries that could not be successfully processed by the tree-sitter library due to syntax errors in the code or incomplete grammar rules. After this step, we ultimately gathered 12,071 vulnerability entries in our constructed dataset. Though CVSS V4 has introduced new concepts and improvements recently. However, vulnerability data labeled with CVSS V4 scores remain very scarce in practice. The majority of scores in our constructed vulnerability datasets are still based on CVSS V2 and CVSS V3, resulting in a limited sample size for CVSS V4, which might affect the generalizability and reliability of the research findings. Therefore, in our study, we primarily base our analysis on CVSS V3.

Following previous studies [16, 17], we split our constructed dataset into the training set (80%), the validation set (10%), and the test set (10%) by stratified sampling, ensuring that the proportions of the four vulnerability severity levels are consistent across these three sets. The statistical information of our experiment subject is shown in Table 1. In this table, we first summarize the number of entries with different severity levels in the different datasets. For example, there are 1,169 vulnerabilities of critical severity in the training set, 146 in the validation set, and 147 in the test set. Next, we analyze the mean and median token counts in source code and vulnerability descriptions across the different datasets. For instance, the mean number of

³<https://github.com/Icyrockton/MegaVul>

⁴<https://www.first.org/cvss/v3.1/specification-document>

⁵<https://tree-sitter.github.io/tree-sitter/>

tokens in the source code is 616 in the training set, 632 in the validation set, and 577 in the test set.

Table 1
Statistical information of our experimental subject.

Statistic	Train	Validation	Test
Number	9,656	1,207	1,208
Number of Critical severity	1,169	146	147
Number of High severity	4,454	577	577
Number of Medium severity	3,795	474	475
Number of Low severity	238	30	29
Average tokens in codes	616	632	577
Average tokens in descriptions	71	69	72
Median tokens in codes	334	361	331
Median tokens in descriptions	57	57	59

4.3. Baselines

To evaluate the effectiveness of our proposed approach SVA-ICL, we compare it with two baselines based on vulnerability source code (i.e., Func_{RF} and Func_{LGBM} [4]), five baselines based on vulnerability descriptions (i.e., CWM_{NB} , CWM_{SVM} , CWM_{LR} [16], Cvss-bert [37] and SPSV [11]) and two baselines based on both vulnerability source code and descriptions (i.e., SVA-PT [38] and MTLML [39]). In the rest of this subsection, we introduce the characteristics of these baselines as follows.

Le et al. [4] proposed a source code-based vulnerability assessment approach, aimed at conducting evaluations at the function level. Their research considers the contextual information of the vulnerable source code, including lines of code that are directly or indirectly related to the vulnerability, to enhance the performance of the vulnerability assessment model. Depending on the chosen classifier, the approach can be divided into two separate approaches as follows.

- Func_{RF} . This approach uses the Random Forest (RF) as the classifier. RF [44] is an ensemble learning approach that performs classification by building multiple decision trees and uses the concept of ensemble learning to improve overall accuracy and stability. The main hyperparameters of RF are the number of trees, maximum depth, and maximum number of leaves.
- Func_{LGBM} . This approach uses the Light Gradient Boosting Machine (LGBM) as the classifier. LGBM [45] is a fast, scalable, and high-performance gradient-boosting decision tree algorithm that belongs to the Gradient Boosting Framework. Its main advantages lie in efficiency, scalability, and accuracy, enabling it to quickly train high-performance models on large datasets. The main hyperparameters are similar to RF.

In addition to the source code-based approach, Le et al. [16] proposed a systematic approach based on vulnerability descriptions that integrates both character and word features. They

also devised a time-based cross-validation technique for determining the optimal model for each vulnerability characteristic. They created natural language processing (NLP) configurations using different values of n-grams in combination with either tf (term frequency) or tf-idf (term frequency-inverse document frequency) weighting. Similar to the first baseline, this approach can be divided into three approaches depending on the chosen classifier.

- CWM_{NB} . This approach uses the Naïve Bayes (NB) as the classifier. NB [46] is a simple probabilistic classifier based on Bayes' theorem and the assumption of feature conditional independence. NB excels in efficiency with large amounts of data, fast training, easy implementation, and outstanding performance in classification tasks. Following the study of Le et al. [16], no hyperparameters were optimized during the validation process for NB.
- CWM_{SVM} . This approach uses the Support Vector Machine (SVM) as the classifier. SVM [47] is a supervised learning algorithm that distinguishes different classes by finding an optimal hyperplane in the feature space. For multi-class problems, we use the one-vs-rest scheme to convert them into multiple binary classification problems.
- CWM_{LR} . This approach uses Logistic Regression (LR) as the classifier. LR [48] is a statistical learning approach used to solve binary classification problems, and it can be extended to multi-class problems using the one-vs-rest scheme. Note that the optimal hyperparameter values for these three classifiers may vary when using different NLP representations such as tf and tf-idf.

For these baselines based on machine learning methods, we directly used their shared code and performed hyperparameter optimization. The hyperparameter names, candidate values, and optimal values for different baselines are shown in Table 2.

Shahid et al. [37] proposed a SVA approach based on natural language processing. They trained multiple BERT classifiers, one for each CVSS vector metric, to determine the CVSS vector and severity score from vulnerability textual descriptions. A gradient-based input saliency method was used for explainability. For convenience, we refer to this approach as Cvss-bert.

Babalau et al. [11] proposed a sVA approach based on vulnerability text descriptions. They used a Multi-Task Learning architecture with a pre-trained BERT model to compute vector-space representations of words in vulnerability descriptions, aiming to predict the severity score and other metrics of a vulnerability using only its text description available upon discovery. We refer to this approach as SPSV.

Xue et al. [38] applied prompt tuning and continual learning for SVA. They combined confidence-based replay and regularization, used source code and vulnerability descriptions to create hybrid prompts for tuning CodeT5. It outperformed baselines, and ablation studies verified its component effectiveness. We refer to this approach as SVA-PT.

Table 2

The range of values and the optimal value for baseline's hyperparameters.

Baseline	Hyperparameter Name	Candidate Values	Optimal Value
Func _{RF}	No. of estimators	100, 200, 300, 400, 500	200
	Max depth	3, 5, 7, 9, unlimited	unlimited
	Max. no. of leaf nodes	100, 200, 300, unlimited	300
Func _{LGBM}	No. of estimators	100, 200, 300, 400, 500	200
	Max depth	3, 5, 7, 9, unlimited	unlimited
	Max. no. of leaf nodes	100, 200, 300, unlimited	100
CWM _{NB}	None	None	None
CWM _{SVM}	Kernel	linear	linear
	Regularization coefficient	0.01, 0.1, 1, 10, 100	0.1
CWM _{LR}	Regularization coefficient	0.01, 0.1, 1, 10, 100	0.1 for tf 10 for tf-idf

Du et al. [39] proposed a vulnerability severity prediction approach MTLM. Specifically, they preprocessed bimodal data (i.e., vulnerability description and source code), used GraphCodeBert for feature extraction, a Bi-GRU with attention for further feature extraction, and a multi-task learning approach with hard parameter sharing.

4.4. Performance Measures

To assess the performance of our proposed approach SVA-ICL, we utilize commonly used performance measures in previous software vulnerability assessment studies [4, 17], such as Accuracy, F1-score, and MCC (Matthews Correlation Coefficient). The latter two measures are particularly suitable for our constructed dataset, which has a class imbalanced problem [49], as shown in Table 1. The details of these performance measures are as follows:

- **Accuracy.** Accuracy represents the proportion of correctly classified samples among all samples, measuring the overall classification performance of the approach. The range of Accuracy values is from 0 to 1, where 0 indicates all samples are classified incorrectly, and 1 indicates all samples are classified correctly. A higher value signifies better performance.
- **F1-score.** The F1-score is the harmonic mean of precision and recall. To evaluate the performance of an approach across the entire dataset in multi-class tasks, we utilize the macro F1-score [50]. The range of F1-score values is from 0 to 1, with values closer to 1 indicating better performance, demonstrating a good balance between precision and recall.
- **MCC.** MCC is a crucial performance measure for multi-class tasks. MCC is particularly useful for handling class-imbalanced datasets as it considers the predictive performance of each class [49]. Similar to the F1-score, we use the multi-class version of MCC [51]. MCC values range from -1 to 1, with values closer to 1 indicating better performance, meaning the approach's predictions in the classification task are more accurate.

By using these three performance measures, we can comprehensively assess the performance of SVA-ICL and baselines in the software vulnerability assessment task, thereby providing a thorough performance analysis.

4.5. LLM used by SVA-ICL

In contrast to previous LLM-based software engineering studies [26, 52, 53], we did not use GPT-3.5 Turbo as the primary LLM for SVA-ICL. Instead, we employed DeepSeek-Chat (DeepSeek-V2) [13] for several reasons. Firstly, DeepSeek-V2, released in May 2024, is the latest version with capabilities close to GPT-4, surpassing GPT-3.5 Turbo comprehensively [13]. Secondly, due to the extensive ablation studies in our research (such as RQ2 to RQ5), we chose DeepSeek-V2 to reduce experimental costs, as the price of DeepSeek-V2 is only about one-third of GPT-3.5 Turbo. Thirdly, the length of the API input content in LLMs is limited by the context window length. Following previous studies [23], we truncate content exceeding the token limitation. However, previous research also showed that truncation could lead to performance degradation [23, 54]. Given that our dataset's source code and vulnerability descriptions are relatively long (see Table 1), we selected the DeepSeek-V2 LLM with a longer context window to mitigate performance decreases due to truncation. Specifically, the token limitation for the DeepSeek-V2 LLM is 32k tokens, while the GPT-3.5 Turbo LLM is only 16k tokens. Detailed comparison results of using different LLMs for SVA-ICL can be found in Section 6. Regarding the cost of using large language models in this experiment, we acknowledge that different large language model products have varying pricing strategies, although they typically use the same token calculation method. As a reference, we are providing the total token consumption for our experiments. Based on backend records, all experiments in this study consumed approximately 108 million tokens.

4.6. Implementation Details

We constructed prompt templates using four demonstration examples, arranged in ascending order based on their similarity. To measure code similarity, we set the ratio of syntactic to lexical similarity at 60% to 40%. For information fusion, we set the ratio of source code similarity to vulnerability description

similarity at 70% to 30%. The rationale for these experimental settings is detailed in our ablation study analysis.

Regarding the hyperparameters of the LLM, and following previous studies [23, 55, 56], we set the temperature parameter to 0. Both the “frequency_penalty” and “presence_penalty” are also set to 0. Specifically, setting the temperature to 0 allows the model to provide the most probable prediction, resulting in text generation with higher certainty. With a frequency penalty of 0, the model does not penalize repeated occurrences of words. Similarly, with a presence penalty of 0, the model does not emphasize the presence of specific words or phrases.

We conduct all experiments on a server equipped with a GeForce RTX 4090 GPU (24GB graphics memory). The server runs on the Windows 10 operating system.

5. Experimental Results

5.1. RQ1: How does SVA-ICL perform in software vulnerability assessment compared to state-of-the-art baselines?

Approach:

To demonstrate the effectiveness of our proposed approach SVA-ICL for the software vulnerability assessment task, we select Func_{RF} , Func_{LGBM} [4], CWM_{NB} , CWM_{SVM} , CWM_{LR} [16], Cvss-bert [37], SPSV [11], SVA-PT [38] and MTLM [39] as state-of-the-art SVA baselines. For SVA-ICL, we consider the experimental settings shown in Section 4.6. Performance evaluation of these approaches was conducted using the measures introduced in Section 4.4.

Results: The performance comparison results between our proposed approach SVA-ICL and the baselines are presented in Table 3. The optimal results for each performance measure are highlighted in bold, while the second-best results are underscored. The results indicate that our proposed approach SVA-ICL outperforms all baselines in terms of three performance measures, particularly excelling in the MCC measure. Specifically, SVA-ICL achieves 77.07% for Accuracy, 67.89% for F1-score, and 63.01% for MCC. Compared to the other baseline approaches, SVA-ICL improves Accuracy by at least 2.53 pp (percentage point), F1-score by at least 3.04 pp, and MCC by at least 3.49 pp.

Table 3

Comparison results between SVA-ICL and baselines in terms of three performance measures.

Approach	Accuracy (%)	F1-score (%)	MCC (%)
Func_{RF}	62.97	43.69	37.25
Func_{LGBM}	69.39	62.48	48.51
CWM_{NB}	62.44	49.26	37.98
CWM_{SVM}	68.16	58.53	46.74
CWM_{LR}	69.32	62.55	49.42
Cvss-bert	70.31	61.26	50.54
SPSV	72.42	61.76	53.68
SVA-PT	72.55	62.43	56.13
MTLM	<u>74.54</u>	64.85	<u>59.52</u>
SVA-ICL	77.07	67.89	63.01

The cost of calculating the code similarity for the target vulnerability is an important factor in evaluating the effectiveness of SVA-ICL. Although similarity computation requires certain computational resources, the vector representations of vulnerability code snippets can be pre-computed offline, thereby significantly reducing the online computation burden. Specifically, during the preliminary preparation stage, we employ CodeBERT and BERT-whitening to process the vulnerability code snippets in the dataset, generating their semantic vectors and optimizing the vector representations. While this offline computation process does require an initial investment of computational resources, it is a one-time cost. During the subsequent vulnerability assessment, when calculating the similarity between a target vulnerability and historical vulnerabilities, we can directly leverage the pre-computed vector representations, substantially reducing the cost of real-time computation.

Summary for RQ1: The experimental results demonstrate that SVA-ICL outperforms baselines in software vulnerability assessment. Specifically, it shows improvements of 2.53 pp in Accuracy, 3.04 pp in F1-score, and 3.49 pp in MCC compared to the second-best baseline.

5.2. RQ2: Whether considering both source code and vulnerability description with information fusion can improve the performance of SVA-ICL?

Approach: In this RQ, we examine the impact of integrating information from different modalities on the performance of SVA-ICL. We combine code similarity for source code and text similarity for vulnerability descriptions using varying ratios based on Equation 5, which assigns different weights to each modality. We explore 11 different ratios, such as 0%:100% (considering only vulnerability descriptions), 50%:50% (equal weighting of both modalities), and 100%:0% (considering only source code).

Results: Table 4 presents the performance of our proposed approach SVA-ICL under 11 different fusion ratios of code similarity and text similarity. Specifically, when the code similarity and text similarity ratios are set to 70% and 30%, respectively, our proposed approach SVA-ICL achieves the best performance in terms of the MCC measure. This indicates that source code information is more critical for assessing the base severity of vulnerabilities. However, incorporating vulnerability descriptions helps the LLM better understand the severity of the vulnerability and make more accurate assessments. When the text similarity ratio is set to 100%, i.e., without considering source code, our proposed approach SVA-ICL performs poorly across all three performance measures. This suggests that examples selected using only vulnerability descriptions are of low quality, making it nearly impossible for the LLM to correctly assess the base severity of vulnerabilities. Conversely, when the code similarity ratio is set to 100% (i.e., without considering vulnerability descriptions), the performance of SVA-ICL also decreases. This further confirms that integrating both vulnera-

bility descriptions and source code improves the performance of SVA-ICL in the software vulnerability assessment task.

Table 4

Experimental results on the ratio of similarity between source code and vulnerability description in our proposed approach SVA-ICL. Notice that CodeSim represents the ratio of similarity in the source code, and TextSim represents the ratio of similarity in the vulnerability descriptions.

CodeSim	TextSim	Accuracy (%)	F1-score (%)	MCC (%)
100%	0%	74.25	63.56	58.14
90%	10%	76.08	67.66	61.43
80%	20%	75.91	68.34	61.24
70%	30%	77.07	67.90	63.01
60%	40%	75.75	67.40	60.93
50%	50%	76.32	<u>68.16</u>	61.82
40%	60%	75.75	66.80	60.87
30%	70%	<u>76.41</u>	67.83	62.04
20%	80%	75.83	67.14	61.15
10%	90%	75.99	66.98	61.52
0%	100%	46.77	34.72	13.65

In addition to quantitative analysis based on evaluation measures, we also perform qualitative analysis using two cases. The results of these two cases with different ratio settings are shown in Figure 4. In Case 1, when the similarity of source code is not considered, SVA-ICL returns an incorrect assessment result (i.e., the ground-truth value is "High", while the assessment value is "Medium"). Further analysis reveals that this may be due to excessive redundancy in the vulnerability description, making it difficult for the LLM to accurately assess the severity level. However, when the source code is considered, our proposed approach SVA-ICL makes the correct assessment, demonstrating that incorporating both source code and vulnerability description information helps the LLM in software vulnerability assessment. In Case 2, SVA-ICL predicts incorrectly when considering only source code or only vulnerability descriptions, but makes the correct assessment when both bimodal information sources are considered. These two real-world cases further demonstrate that integrating code and description bimodal information and using a reasonable ratio setting helps improve the performance of SVA-ICL.

Summary for RQ2: Compared to considering a single modality, integrating two modalities (i.e., source code and vulnerability descriptions) achieves better performance. Moreover, a ratio setting of 70%:30% achieves the optimal performance of SVA-ICL in our study.

5.3. RQ3: How does the number of demonstration examples in the prompt affect the performance of SVA-ICL?

Approach: Inspired by previous work [26], we investigate the impact of the number of demonstration examples on the performance of our proposed approach SVA-ICL. Specifically, during the demonstration example selection phase, we select the top 10 candidate vulnerabilities most similar to the target vulnerability. We then construct ICL demonstration examples

Case 1:

```
Code: void* _TIFFmalloc(tmsize_t s) {
    return (malloc((size_t)s));
}
```

Description: The _TIFFmalloc function in tif_unix.c in LibTIFF 4.0.3 does not reject a zero size, which allows remote attackers to cause a denial of service (divide-by-zero error and application crash) via a crafted TIFF image that is mishandled by the TIFFWriteScanline function in tif_write.c, as demonstrated by tiffdither.

[output]

SVA-ICL (Code_sim=100%, Text_sim=0%):	High	⊗
SVA-ICL (Code_sim=0%, Text_sim=100%):	Medium	⊗
SVA-ICL (Code_sim=70%, Text_sim=30%):	High	⊗

Ground Truth: High

Case 2:

```
Code: int Read(void* pDestBuffer, int nSize) {
    if ( m_nPos + nSize >= m_nLen ) nSize = m_nLen - m_nPos - 1;
    memcpy( pDestBuffer, (m_sFile + m_nPos), nSize );
    m_nPos += nSize;
    return nSize;
}
```

Description: Onlyoffice Document Server v6.0.0 and below and Core 6.1.0.26 and below were discovered to contain a heap overflow via the component DesktopEditor/fontengine/fontconverter/FontFileBase.h.

[output]

SVA-ICL (Code_sim=100%, Text_sim=0%):	High	⊗
SVA-ICL (Code_sim=0%, Text_sim=100%):	Medium	⊗
SVA-ICL (Code_sim=70%, Text_sim=30%):	Critical	⊗

Ground Truth: Critical

Figure 4: Two cases of the base severity returned by our proposed approach SVA-ICL (which only considers source code, only considers vulnerability description, and considers both with the best ratio setting)

using 0, 1, 4, and 5 of the most similar vulnerabilities. According to the results of RQ2, we set the ratio of code similarity to text similarity at 70% and 30% in RQ3. Additionally, the length of source code and vulnerability descriptions in our dataset is relatively long, which is constrained by the context window length of the LLM. Therefore, we truncate the prompt template as described in Section 4.6. We use the same prompt template for different numbers of demonstration examples as shown in Figure 3 to ensure a fair comparison.

Results: We show the performance of SVA-ICL with different numbers of demonstration examples in Table 5. As shown in the table, when using zero-shot learning (i.e., without demonstration example), SVA-ICL performs the worst in terms of all three performance measures, with an MCC of only 7.14%, which is a 55.87% performance decrease compared to the best experimental setting. This indicates that without considering demonstration examples, it is difficult to utilize the relevant knowledge in the LLM. Comparing the results of using one demonstration example (i.e., with one-shot) versus four demonstration examples, it is evident that the number of demonstra-

tion examples significantly enhances the performance of LLMs. However, when using five demonstration examples to construct the prompt template, the performance decreases compared to using four examples. Based on our manual analysis and the findings of previous studies [54, 57], the performance decrease may be caused by the content truncation of the prompt. This truncation may remove crucial information from the source code and vulnerability descriptions, hindering the LLM’s ability to accurately understand the target vulnerabilities and leading to a performance drop. In this RQ, we find that setting the number of demonstration examples to 4 achieves optimal performance. In a previous study on code generation task [23], Gao et al. also recommend setting the demonstration number to 4 can help to achieve optimal performance. Therefore, the findings of our study are consistent with theirs.

Table 5

The performance impact of different demonstration numbers for our proposed approach SVA-ICL.

Setting	Accuracy (%)	F1-score (%)	MCC (%)
with zero-shot	44.21	27.53	7.14
with one-shot	66.39	54.18	45.73
with 4-shot	77.07	67.89	63.01
with 5-shot	67.47	56.37	47.61

Summary for RQ3: Increasing the number of demonstration examples can improve the performance of our proposed approach, SVA-ICL, to some extent. However, when the number of demonstrations exceeds 4, performance decreases due to content truncation issues. In our study, using four demonstration examples achieves the best performance.

5.4. RQ4: How do different demonstration ordering strategies affect the performance of SVA-ICL?

Approach: Inspired by the research of Gao et al. [23], we analyze the impact of different ordering strategies of demonstration examples on the performance of our proposed approach SVA-ICL. We hypothesize that arranging demonstration examples according to a specific ranking plays an important role in ICL. To test this hypothesis, we consider three different ordering strategies: **Similarity**, **Reverse Similarity**, and **Random**. Specifically, **Similarity** means arranging the demonstration examples in ascending order of their similarity to the test sample, with the examples most similar to the test sample placed closest to the target sample in the prompt template. **Reverse Similarity** is the opposite, arranged in descending order of similarity. **Random** means randomly arranging the demonstration examples.

Results: The results in Table 6 show that arranging the demonstration examples in ascending order of similarity to the test sample (i.e., Similarity strategy) yields the best performance. Specifically, in terms of the MCC performance measure, SVA-ICL outperforms the Random strategy by 5.76 pp and the Re-

verse Similarity strategy by 4.37 pp. This validates our hypothesis. Additionally, we observe that the performance of our approach under the Reverse Similarity strategy is better than under the Random strategy, aligning with Gao et al.’s findings for the source code summarization and program synthesis tasks [23]. This suggests that ordering demonstration examples according to certain rules can improve ICL performance, indicating the potential for exploring more complex demonstration example ordering strategies in the future.

Table 6

Experimental results of different demonstration ordering strategies.

Strategy	Accuracy (%)	F1-score (%)	MCC (%)
Random	73.51	64.69	57.25
Reverse Similarity	<u>74.34</u>	<u>66.38</u>	<u>58.64</u>
Similarity	77.07	67.89	63.01

Summary for RQ4: The ordering strategy of demonstration examples affects the performance of SVA-ICL. Arranging them in ascending order of similarity achieves the best performance in our study.

5.5. RQ5: How does the fusion ratio of syntactic similarity and lexical similarity for source code similarity affect the performance of SVA-ICL?

Approach: In our previous work [25, 26], we analyzed the impact of fusing syntactic and lexical similarity with different ratio settings for source code similarity in the task of smart contract code comment generation. However, for software vulnerability assessment, it is essential to design an ablation study to determine the optimal fusion ratio. To this end, we investigated 11 different ratio settings, while maintaining the other experimental settings as described in Section 4.6.

Results: The experimental results are shown in Table 7. The first row of the table represents considering only syntactic similarity, and the last row represents considering only lexical similarity. The remaining rows show data for simultaneously considering both syntactic and lexical similarity in different proportions. It can be seen that, compared to considering a single kind of similarity, considering both kinds of similarities improves the performance of the approach SVA-ICL. Especially, when the syntactic similarity ratio is set to 40% and the lexical similarity weight is set to 60%, SVA-ICL achieves the best performance. Specifically, under this fusion ratio setting, SVA-ICL improves the performance by at least 1.66 pp in terms of the MCC measure compared to other ratio settings. This indicates that considering both syntactic and lexical similarities provides a more comprehensive measure of source code similarity.

Table 7

Experimental results under different fusion ratios of syntactic similarity and lexical similarity of source code. Notice SynSim represents the syntactic similarity of source code, and LexiSim represents the lexical similarity of source code.

SynSim	LexSim	Accuracy (%)	F1-score (%)	MCC (%)
100%	0%	71.94	63.81	53.90
90%	10%	73.09	65.14	56.17
80%	20%	74.34	66.07	58.35
70%	30%	75.83	66.89	61.03
60%	40%	74.50	65.81	58.66
50%	50%	75.25	67.09	60.13
40%	60%	77.07	67.90	63.01
30%	70%	75.66	66.70	60.68
20%	80%	<u>76.08</u>	<u>67.28</u>	<u>61.35</u>
10%	90%	72.10	63.76	54.74
0%	100%	68.54	59.27	49.78

Summary for RQ5: Considering both syntactic and lexical similarity can provide a comprehensive measure for source code similarity. SVA-ICL achieves the best performance when the ratio of syntactic similarity and lexical similarity is set as 40% and 60% in our study.

6. Discussion

6.1. Comparison with Other Popular LLMs

In Section 4.6, we show the reasons for using DeepSeek-V2 as our LLM. To show the effectiveness of using this LLM for SVA-ICL, we also consider four popular LLMs (i.e., GPT-3.5 Turbo, GPT-4o, GPT-4o mini, and Qwen2.5-Coder-32B-Instruct). These LLMs have been recently used for improving software engineering tasks (such as code summarization [23, 26, 58], bug fixing [59, 60], and program synthesis [61, 62]). In this experiment, we use the same experimental setting shown in Section 4.6 to guarantee a fair comparison.

Table 8

Performance comparison between DeepSeek-V2 and other popular LLMs.

LLM	Accuracy (%)	F1-score (%)	MCC (%)
GPT-3.5 Turbo	61.01	54.38	40.10
GPT-4o mini	67.23	61.42	51.03
GPT-4o	<u>76.74</u>	69.90	63.40
Qwen2.5-Coder-32B-Instruct	64.21	55.93	48.75
DeepSeek-V2	77.07	<u>67.89</u>	<u>63.01</u>

As shown in Table 8, GPT-3.5 Turbo achieves the worst performance among these LLMs, likely due to its context window limitation of 16k tokens, whereas the other LLMs support at least 32k tokens. Additionally, officially released data [13] indicates that DeepSeek-V2 outperforms GPT-3.5 Turbo in areas such as mathematics, reasoning, and programming. GPT-4o mini and Qwen2.5-Coder-32B-Instruct perform better than GPT-3.5 Turbo but are still slightly inferior to DeepSeek-V2. GPT-4o mini benefits from its relatively larger context window compared to GPT-3.5 Turbo, which allows it to handle more

context information. This enables it to capture more details from the source code and vulnerability descriptions. Qwen2.5-Coder-32B-Instruct, on the other hand, has shown strong performance in coding tasks. Its design and pre-training make it better at understanding code-related semantics. However, compared to DeepSeek-V2, GPT-4o mini may lack some specific optimizations for software vulnerability assessment, and Qwen2.5-Coder-32B-Instruct may not be proficient in integrating code and vulnerability description information due to its relatively small parameter size, resulting in a slight decrease in performance. From the table, we find that GPT-4o performs slightly better than DeepSeek-V2. Given that the specific parameters and design details of GPT-4o have not yet been made public, we hypothesize that its advantage may stem from its support for an extended context of 128k tokens and its superior code analysis and language comprehension capabilities. Although GPT-4o performs slightly better than DeepSeek-V2, its cost is significantly higher. For instance, GPT-4o costs five dollars per million tokens, whereas DeepSeek-V2 costs only fourteen cents for the same number of tokens, making GPT-4o approximately 35 times more expensive. Therefore, we primarily consider DeepSeek-V2 as our LLM for SVA-ICL.

In summary, these results not only confirm the competitiveness of using DeepSeek-V2 in SVA-ICL but also show that SVA-ICL can achieve relatively good performance for different LLMs when considering in-context learning and information fusion.

6.2. The Influence of Demonstration Similarity on Assessment Performance

We retrieve the specific similarity values of the selected demonstrations in our empirical study. To better illustrate the relationship between similarity values and assessment performance, we divide the test set based on the average similarity values of the demonstration examples. Note that we set the optimal number of demonstration examples to 4, as determined through our experiments.

Table 9

Influence of demonstration similarity on assessment results of SVA-ICL. Notice: “Avg. Sim. Range” stands for average similarity values range, and “No. Samp.” represents the number of samples.

Avg. Sim. Range	No. Samp.	Accuracy (%)	F1-score (%)	MCC (%)
[0, 0.2)	687	73.37	65.23	61.58
[0.2, 0.5)	401	<u>78.24</u>	<u>68.01</u>	<u>64.53</u>
[0.5, 1]	120	81.05	70.21	68.13

As shown in Table 9, the similarity values of the demonstration examples have an impact on the assessment results. Generally, a higher average similarity value between the demonstration examples and the target vulnerability leads to more accurate assessment and better-performing evaluation metrics. For the target vulnerabilities with an average similarity value between 0.5 and 1, the model achieves an Accuracy of 81.05%, F1-score of 70.21%, and an MCC of 68.13%. In contrast, for the target vulnerabilities with an average similarity value between 0 and 0.2, the Accuracy, F1-score, and MCC decrease

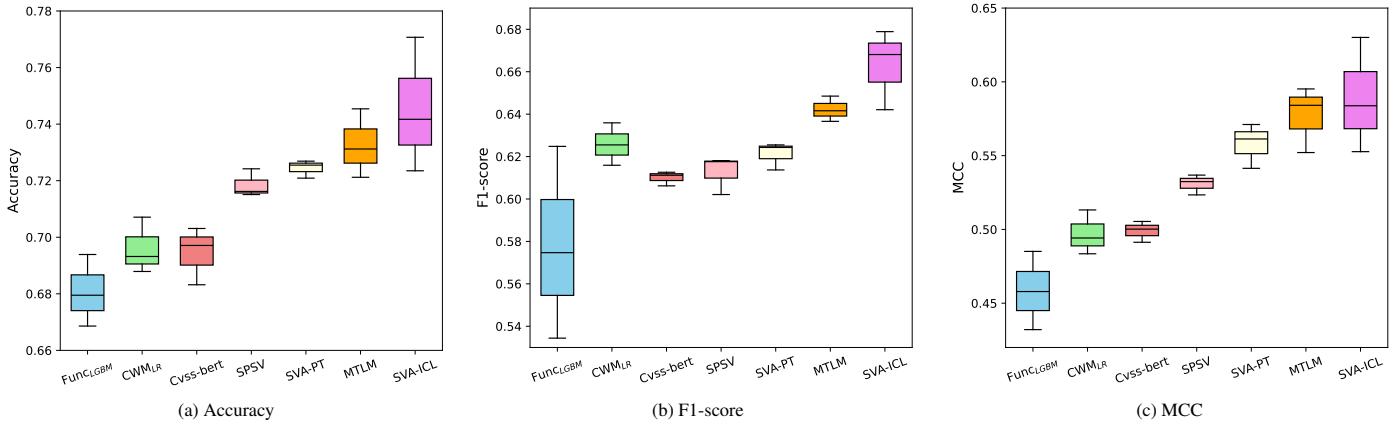


Figure 5: Comparison results between SVA-ICL and baselines in terms of different performance measures for different dataset splits.

to 73.37%, 65.23%, and 61.58%, respectively. This trend indicates that more similar demonstration examples provide the LLM with more relevant and useful information during the in-context learning process. As a result, the model better understands the characteristics of the target vulnerability and makes more accurate assessments of its severity level.

Regarding the threshold for similarity values, although it is challenging to determine a unified threshold for different tasks, based on our experimental results, when the average similarity values of the demonstration examples are larger than 0.5, the model can achieve relatively good performance in software vulnerability assessment. However, it should be noted that the optimal threshold may vary depending on factors such as the scale and diversity of the gathered historical vulnerabilities. Specifically, larger and more diverse vulnerabilities require a higher threshold to ensure the quality of the selected demonstration examples, while smaller and less diverse vulnerabilities need a lower threshold to include sufficient relevant examples. Therefore, we follow previous studies [23, 26] by selecting the top- k most similar demonstrations for in-context learning.

6.3. The Influence of Different Dataset Splits

To address the inherent randomness in dataset splitting, we evaluate the performance of SVA-ICL and baseline approaches with different data splits. We independently split the data using various random seeds and executed these approaches under the same experimental settings described in Section 4.6. The results of these three independent runs are presented in Figure 5 using boxplots. These boxplots illustrate the performance of seven different approaches (Func_{LGBM} , CWM_{LR} , Cvss-bert, SPSV, SVA-PT, MTLM and SVA-ICL) in terms of Accuracy, F1-score, and MCC. Each boxplot shows the highest score (upper edge of the box), lowest score (lower edge of the box), and median score (line inside the box) achieved by these approaches over the three random splits.

From Figure 5, we observe that our proposed approach SVA-ICL consistently achieves the best performance across all three measures (Accuracy, F1-score, MCC). Specifically, SVA-ICL improves Accuracy, F1-score, and MCC by an average of 3.75 pp, 4.81 pp, and 6.88 pp, respectively. This demonstrates that

the competitiveness of our proposed approach SVA-ICL remains robust against different data splits when compared with baseline approaches.

6.4. Comparison with Random Demonstration Selection

To evaluate how the choice of demonstration examples affects model performance, we compare our relevance-based selection strategy with the random selection strategy. For a fair comparison, we set the number of selected demonstration examples to 4 for these two different strategies. As shown in Table 10, the relevance-based selection strategy achieves an accuracy of 77.07%, significantly higher than the 61.75% accuracy achieved by the random selection strategy. For the F1-score, the relevance-based strategy reaches 67.89%, while the random selection strategy only achieves 46.68%. The MCC further supports this finding, with the relevance-based strategy yielding an MCC of 63.01%, compared to 40.92% for the random selection strategy. In summary, the relevance-based selection strategy outperforms the random selection strategy across all three performance measures.

Table 10
Experimental results of different demonstration selection strategies.

Strategy	Accuracy (%)	F1-score (%)	MCC (%)
Random-based	61.75	46.68	40.92
Relevance-based	77.07	67.89	63.01

6.5. The Influence of Potential Data Leakage

To mitigate the potential data leakage issue, we extract test samples collected after July 2023 (i.e., the model’s cutoff day for deepseek v2) and evaluate the model’s performance on this subset. We examine the performance comparison between SVA-ICL and the top-performing baseline, MTLM. As shown in Table 11, SVA-ICL outperforms MTLM across all performance measures, achieving 74.89% accuracy (vs. 73.31%), 65.43% F1-score (vs. 60.79%), and 60.68% MCC (vs. 53.85%). These

results show that our approach still achieves a better performance, demonstrating its effectiveness even when evaluated on data that was not seen during the pre-training phase of the LLM.

Table 11

Comparison results between SVA-ICL and the top-performing baseline in terms of three performance measures.

Approach	Accuracy (%)	F1-score (%)	MCC (%)
MTLM	73.31	60.79	53.85
SVA-ICL	74.89	65.43	60.68

6.6. Insights from Our Empirical Results

Fusing source code and vulnerability description. One of the primary reasons for the superior performance of our proposed approach SVA-ICL is its ability to effectively integrate both source code and vulnerability descriptions through information fusion. Different from the baseline methods, which typically rely on either source code or textual descriptions alone, SVA-ICL combines both modalities. This dual-modality approach enhances the model’s contextual understanding, allowing it to capture more comprehensive patterns that are crucial for accurate software vulnerability assessment. The significant drop in performance when only vulnerability descriptions were considered highlights the limitation of relying solely on textual information. Vulnerability descriptions, while informative, often lack the technical depth needed to fully understand the nature and severity of a vulnerability. Source code, on the other hand, contains critical details that directly influence the severity level, such as the specific operations and logic within the code. Therefore, the reliance on source code is necessary for accurate and precise assessments. While source code provides the foundational technical details, the integration of vulnerability descriptions appears to enhance the model’s ability to interpret and contextualize these details. Descriptions often summarize the implications or nature of the vulnerability in a way that complements the raw data provided by the code. This combination enables the model to not only identify the severity but also to understand the broader context, leading to more accurate predictions.

Using ICL to further improve the performance of LLM-based approach. The results for RQ3 clearly show that the inclusion of demonstration examples substantially improves the performance of SVA-ICL, with the MCC increasing from 7.14% in the zero-shot setting to 63.01% in the four-shot setting. Interestingly, the best performance was achieved with four demonstration examples, beyond which the performance began to decline. This finding aligns with previous research [23], indicating that there is an optimal number of examples that maximizes the model’s ability to generalize. Too few examples (as in the zero-shot or one-shot scenarios) do not provide enough context for the model to effectively leverage its pre-trained knowledge, while too many examples can lead to information overload and potential truncation of critical content. These insights indicate that in designing prompts for tasks involving complex and

lengthy inputs like software vulnerability assessment, it is crucial to carefully consider the number of examples included. The optimal number of examples will depend on the task and the specific context window limitations of the model being used. For future work, this suggests exploring dynamic or adaptive strategies that adjust the number of examples based on the specific input length and complexity. Additionally, the results for RQ4 highlight the importance of carefully considering the order in which demonstration examples are presented in ICL tasks. For tasks like software vulnerability assessment, where context and relevance are crucial, ordering examples by similarity values can significantly enhance the model’s performance. This suggests that in future applications of ICL, particularly for complex tasks, a deliberate approach to example ordering can be a key factor in optimizing outcomes.

Requiring comprehensive code analysis. The syntactic structure of source code provides a framework for code organization and flow control, while the lexical elements contain specific operations and data entities. Together, they offer a more comprehensive understanding of code behavior. Therefore, when these two similarities are effectively combined, the relationship between the target vulnerability code and the candidate vulnerability code can be captured more accurately. This also highlights the need for further exploration and optimization of code representation in future code-related software engineering studies by performing comprehensive code analysis.

6.7. Threats to Validity

In this subsection, we discuss the potential threats to the validity of our study.

Internal Threats. Large language models (LLMs) are trained on extensive open-source datasets, raising concerns about data leakage. This means that LLMs might have learned answers related to the test set, leading them to memorize results rather than predict them. However, our experimental results show that zero-shot learning performance is very poor, suggesting that the likelihood of data leakage is very low.

External Threats. The primary external threat is the version of the LLM API used, as different API versions may exhibit performance variations. We cannot guarantee that all versions of the API will achieve the results presented in our study. To ensure the reproducibility of our results, we recommend using the DeepSeek-V2 version of the API when invoking the LLM. Additionally, for vulnerability types that have a limited presence in the historical data, our proposed approach may not always achieve promising assessment results, but this is a common issue with data-driven approaches. To alleviate this threat, our approach is designed to be continuously updated as new data becomes available. This means that as new vulnerability types emerge, they can be incorporated into the model’s training data, thereby improving its performance over time.

Construct Threats: Construct threats relate to the performance measures used. To mitigate this threat, we selected three commonly used measures in the field of software vulnerability assessment [4, 16, 17]. Additionally, the construction of the prompt template is another construct threat. To address this,

we followed previous research [23] and employed carefully designed prompt templates. Our experimental results demonstrate the effectiveness of these prompt templates.

Conclusion Threats: Our constructed dataset focuses exclusively on C/C++ source code. However, since LLMs are trained on extensive datasets, our proposed approach SVA-ICL can also be applied to software vulnerability assessment for other programming languages. Additionally, we did not select other approaches from related work as baselines because their code is not publicly available, and implementing these approaches would be challenging.

7. Related Work

Software vulnerability assessment (SVA) predicts the severity of vulnerabilities based on vulnerability information (e.g., source code) to identify more critical software vulnerabilities. Developers can prioritize addressing these severe vulnerabilities based on the assessment results and propose corresponding remediation suggestions, thereby improving the overall quality of the software. In recent years, SVA research can be broadly categorized into two types based on the modalities considered: vulnerability description-based SVA and source code-based SVA.

Vulnerability description-based SVA uses textual descriptions of vulnerabilities to predict their severity. Han et al. [63] utilized word embeddings and a shallow convolutional neural network to capture discriminative words and sentence features in vulnerability descriptions for severity prediction. Spanos et al. [50] developed a model combining text analysis and multi-objective classification techniques to estimate vulnerability features and calculate severity scores. Liu et al. [3] proposed deep learning methods for vulnerability text classification, though these methods require significant time for model training. Le et al. [16] combined character and word features in their systematic approach, which necessitated tuning numerous hyperparameters. Babalau et al. [11] used a multi-task learning architecture with a pre-trained BERT model to compute vector-space representations of words in vulnerability descriptions, but this approach requires additional experiments to determine the values of hyperparameters and loss weights. Shahid et al. [37] leveraged recent advances in the field of NLP to predict vulnerability severity scores from textual descriptions via multiple trained BERT classifiers and used a gradient-based method for explainability.

Source code-based SVA involves analyzing the source code to predict potential security vulnerabilities. For instance, Ganesh et al. [64] evaluated the effectiveness of machine learning algorithms in predicting security vulnerabilities by analyzing system source code, but found that their machine learning models exhibited low performance in predictions. Le et al. [4] investigated machine learning models for automatically performing function-level SVA and explored the value of using vulnerable statements as inputs for developing assessment models. Hao et al. [12] proposed a source code-level vulnerability severity assessment approach that does not require vulnerability reports or manual analysis, allowing for quick severity as-

essment by combining function call graphs and vulnerability attribute graphs.

Bimodal data-based SVA uses both vulnerability description and source code to predict severity. Xue et al. [38] proposed a method for SVA, applying prompt tuning and continual learning. It combined confidence-based replay and regularization, used source code and vulnerability descriptions to create hybrid prompts for tuning with CodeT5. Du et al. [39] proposed an SVA method based on bimodal data and multi-task learning. They preprocessed bimodal data (description and source code), used GraphCodeBert for feature extraction, Bi-GRU with attention for further extraction, and a multi-task learning approach with hard parameter sharing.

Different from the previous research, we are the first to introduce large language models into the task of software vulnerability assessment and utilize in-context learning to enhance the performance of vulnerability evaluation. To select high-quality demonstration examples, we apply an information fusion approach that comprehensively considers both source code and vulnerability descriptions. Additionally, we analyze the impact of the number of examples, ordering strategies, and code similarity measurement methods on the performance of SVA-ICL.

8. Conclusion and Future Work

In this study, we propose a novel approach SVA-ICL, which leverages the in-context learning capabilities of large language models for software vulnerability assessment. By performing information fusion for source code and vulnerability description, SVA-ICL can select high-quality demonstration examples from historical corpora. These examples are used to construct customized prompts during the ICL stage, which can aid the LLM DeepSeek-V2 in leveraging relevant knowledge for software vulnerability assessment. Experimental results on our constructed dataset demonstrate that SVA-ICL outperforms SVA baselines in terms of various performance measures. Finally, we conduct a set of ablation studies to show the effectiveness of the customization of SVA-ICL, such as the demonstration number, and the ordering strategy.

Despite the promising performance of our SVA-ICL approach in software vulnerability assessment, there are still several directions worth further exploration. First, we want to design more efficient demonstration example selection strategies to further improve the performance of SVA-ICL. Second, we want to design more high-quality prompts to guide LLM to improve the assessment performance. Lastly, we want to apply SVA-ICL to other programming languages to show the generalization of SVA-ICL.

CRediT authorship contribution statement

Chaoyang Gao: Data curation, Software, Validation, Conceptualization, Methodology, Writing -review & editing. **Xiang Chen:** Conceptualization, Methodology, Writing -review & editing, Supervision.

Guangbei Zhang: Conceptualization, Data curation, Software.

Declaration of competing interest

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

Data availability

We share our data on GitHub (<https://github.com/judeomg/SVA-ICL>).

Acknowledgments

Chaoyang Gao and Xiang Chen have contributed equally to this work and are co-first authors. Xiang Chen is the corresponding author. This research was partially supported by the National Natural Science Foundation of China (Grant No. 61202006), the Open Project of State Key Laboratory for Novel Software Technology at Nanjing University under (Grant No. KFKT2024B21), and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (Grant No. SJCX24_2022).

References

- [1] T. H. Le, H. Chen, M. A. Babar, A survey on data-driven software vulnerability assessment and prioritization, *ACM Computing Surveys* 55 (5) (2022) 1–39.
- [2] N. Dissanayake, A. Jayatilaka, M. Zahedi, M. A. Babar, Software security patch management-a systematic literature review of challenges, approaches, tools and practices, *Information and Software Technology* 144 (2022) 106771.
- [3] K. Liu, Y. Zhou, Q. Wang, X. Zhu, Vulnerability severity prediction with deep neural network, in: 2019 5th international conference on big data and information analytics (BigDIA), IEEE, 2019, pp. 114–119.
- [4] T. H. M. Le, M. A. Babar, On the use of fine-grained vulnerable code statements for software vulnerability assessment models, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 621–633.
- [5] P. Vaithilingam, T. Zhang, E. L. Glassman, Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models, in: Chi conference on human factors in computing systems extended abstracts, 2022, pp. 1–7.
- [6] J. Liu, C. S. Xia, Y. Wang, L. Zhang, Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, *Advances in Neural Information Processing Systems* 36 (2024).
- [7] M. D. Purba, A. Ghosh, B. J. Radford, B. Chu, Software vulnerability detection using large language models, in: 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2023, pp. 112–119.
- [8] X. Zhou, T. Zhang, D. Lo, Large language model for vulnerability detection: Emerging results and future directions, in: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, 2024, pp. 47–51.
- [9] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, X. Liao, Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.
- [10] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, Z. Sui, A survey on in-context learning, *arXiv preprint arXiv:2301.00234* (2022).
- [11] I. Babalau, D. Corlatescu, O. Grigorescu, C. Sandescu, M. Dascalu, Severity prediction of software vulnerabilities based on their text description, in: 2021 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE, 2021, pp. 171–177.
- [12] J. Hao, S. Luo, L. Pan, A novel vulnerability severity assessment method for source code based on a graph neural network, *Information and Software Technology* 161 (2023) 107247.
- [13] DeepSeek-AI, Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model (2024), [arXiv:2405.04434](https://arxiv.org/abs/2405.04434).
- [14] M. Humayun, N. Jhanjhi, M. F. Almuafreh, M. I. Khalil, Security threat and vulnerability assessment and measurement in secure software development, *Comput. Mater. Contin* 71 (2022) 5039–5059.
- [15] 2024. Common Vulnerability Scoring System. <https://www.first.org/cvss/>.
- [16] T. H. M. Le, B. Sabir, M. A. Babar, Automated software vulnerability assessment with concept drift, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, 2019, pp. 371–382.
- [17] T. H. M. Le, D. Hin, R. Croft, M. A. Babar, Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2021, pp. 717–729.
- [18] 2024. National Vulnerability Database. <https://nvd.nist.gov/>.
- [19] E. Kasneci, K. Seßler, S. Küchemann, M. Bannert, D. Dementieva, F. Fischer, U. Gasser, G. Groh, S. Günemann, E. Hüllermeier, et al., Chatgpt for good? on opportunities and challenges of large language models for education, *Learning and individual differences* 103 (2023) 102274.
- [20] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, et al., A survey on evaluation of large language models, *ACM Transactions on Intelligent Systems and Technology* 15 (3) (2024) 1–45.
- [21] H. Tang, K. Hu, J. P. Zhou, S. Zhong, W.-L. Zheng, X. Si, K. Ellis, Code repair with llms gives an exploration-exploitation tradeoff, *arXiv preprint arXiv:2405.17503* (2024).
- [22] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, L. Zettlemoyer, Rethinking the role of demonstrations: What makes in-context learning work?, *arXiv preprint arXiv:2202.12837* (2022).
- [23] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, M. R. Lyu, What makes good in-context demonstrations for code intelligence tasks with llms?, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2023, pp. 761–773.
- [24] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, *arXiv preprint arXiv:2109.00859* (2021).
- [25] G. Yang, K. Liu, X. Chen, Y. Zhou, C. Yu, H. Lin, Ccgir: Information retrieval-based code comment generation method for smart contracts, *Knowledge-Based Systems* 237 (2022) 107858.
- [26] J. Zhao, X. Chen, G. Yang, Y. Shen, Automatic smart contract comment generation via large language models and in-context learning, *Information and Software Technology* 168 (2024) 107405.
- [27] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, *arXiv preprint arXiv:2002.08155* (2020).
- [28] J. Su, J. Cao, W. Liu, Y. Ou, Whitening sentence representations for better semantics and faster retrieval, *arXiv preprint arXiv:2103.15316* (2021).
- [29] X. Ming, Similarities: similarity calculation and semantic search toolkit, <https://github.com/shibing624/similarities> (2022).
- [30] K. Liu, X. Chen, C. Chen, X. Xie, Z. Cui, Automated question title reformulation by mining modification logs from stack overflow, *IEEE Transactions on Software Engineering* (2023).
- [31] W. Yin, L. Shang, Efficient nearest neighbor emotion classification with bert-whitening, in: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, 2022, pp. 4738–4745.
- [32] T. Jiang, J. Jiao, S. Huang, Z. Zhang, D. Wang, F. Zhuang, F. Wei, H. Huang, D. Deng, Q. Zhang, Promptbert: Improving bert sentence embeddings with prompts, *arXiv preprint arXiv:2201.04337* (2022).
- [33] G. Yang, X. Chen, J. Cao, S. Xu, Z. Cui, C. Yu, K. Liu, Comformer: Code comment generation via transformer and fusion method-based hybrid code representation, in: 2021 8th International Conference on Dependable Systems and Their Applications (DSA), IEEE, 2021, pp. 30–41.
- [34] J. K. Siow, S. Liu, X. Xie, G. Meng, Y. Liu, Learning program semantics with code representations: An empirical study, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022, pp. 554–565.
- [35] L. Yujian, L. Bo, A normalized levenshtein distance metric, *IEEE trans-*

- actions on pattern analysis and machine intelligence 29 (6) (2007) 1091–1095.
- [36] S. Bag, S. K. Kumar, M. K. Tiwari, An efficient recommendation generation using relevant jaccard similarity, Information Sciences 483 (2019) 53–64.
- [37] M. R. Shahid, H. Debar, Cvss-bert: Explainable natural language processing to determine the severity of a computer security vulnerability from its description, in: 2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA), IEEE, 2021, pp. 1600–1607.
- [38] J. Xue, X. Chen, J. Wang, Z. Cui, Towards prompt tuning-based software vulnerability assessment with continual learning, Computers & Security 150 (2025) 104184.
- [39] X. Du, S. Zhang, Y. Zhou, H. Du, A vulnerability severity prediction method based on bimodal data and multi-task learning, Journal of Systems and Software 213 (2024) 112039.
- [40] F. Xue, Y. Fu, W. Zhou, Z. Zheng, Y. You, To repeat or not to repeat: Insights from scaling llm under token-crisis, Advances in Neural Information Processing Systems 36 (2024).
- [41] J. He, W. Jiang, G. Hou, W. Fan, R. Zhang, H. Li, Talk too much: Poisoning large language models under token limit, arXiv preprint arXiv:2404.14795 (2024).
- [42] C. Ni, L. Shen, X. Yang, Y. Zhu, S. Wang, Megavul: Ac/c++ vulnerability dataset with comprehensive code representations, in: 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR), IEEE, 2024, pp. 738–742.
- [43] 2024. Common Vulnerabilities and Exposures. <https://cve.mitre.org/>.
- [44] T. K. Ho, Random decision forests, in: Proceedings of 3rd international conference on document analysis and recognition, Vol. 1, IEEE, 1995, pp. 278–282.
- [45] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T.-Y. Liu, Lightgbm: A highly efficient gradient boosting decision tree, Advances in neural information processing systems 30 (2017).
- [46] S. J. Russell, P. Norvig, Artificial intelligence: a modern approach, Pearson, 2016.
- [47] C. Cortes, V. Vapnik, Support-vector networks, Machine learning 20 (1995) 273–297.
- [48] S. H. Walker, D. B. Duncan, Estimation of the probability of an event as a function of several independent variables, Biometrika 54 (1-2) (1967) 167–179.
- [49] A. Luque, A. Carrasco, A. Martín, A. de Las Heras, The impact of class imbalance in classification performance metrics based on the binary confusion matrix, Pattern Recognition 91 (2019) 216–231.
- [50] G. Spanos, L. Angelis, A multi-target approach to estimate software vulnerability characteristics and severity scores, Journal of Systems and Software 146 (2018) 152–166.
- [51] J. Gorodkin, Comparing two k-category assignments by a k-category correlation coefficient, Computational biology and chemistry 28 (5-6) (2004) 367–374.
- [52] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen, et al., A comprehensive capability analysis of gpt-3 and gpt-3.5 series models, arXiv preprint arXiv:2303.10420 (2023).
- [53] Y. Anand, Z. Nussbaum, B. Duderstadt, B. Schmidt, A. Mulyar, Gpt4all: Training an assistant-style chatbot with large scale data distillation from gpt-3.5-turbo, GitHub (2023).
- [54] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, R. Salakhutdinov, Transformer-xl: Attentive language models beyond a fixed-length context, arXiv preprint arXiv:1901.02860 (2019).
- [55] N. Nashid, M. Sintaha, A. Mesbah, Retrieval-based prompt selection for code-related few-shot learning, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 2450–2462.
- [56] Z. Cheng, T. Xie, P. Shi, C. Li, R. Nadkarni, Y. Hu, C. Xiong, D. Raden, M. Ostendorf, L. Zettlemoyer, et al., Binding language models in symbolic languages, arXiv preprint arXiv:2210.02875 (2022).
- [57] A. Bulatov, Y. Kuratov, M. Burtsev, Recurrent memory transformer, Advances in Neural Information Processing Systems 35 (2022) 11079–11091.
- [58] T. Ahmed, K. S. Pai, P. Devanbu, E. Barr, Automatic semantic augmentation of language model prompts (for code summarization), in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.
- [59] R. Ishizue, K. Sakamoto, H. Washizaki, Y. Fukazawa, Improved program repair methods using refactoring with gpt models, in: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, 2024, pp. 569–575.
- [60] T. Do Viet, K. Markov, Using large language models for bug localization and fixing, in: 2023 12th International Conference on Awareness Science and Technology (iCAST), IEEE, 2023, pp. 192–197.
- [61] T. Y. Zhuo, Large language models are state-of-the-art evaluators of code generation, arXiv preprint arXiv:2304.14317 (2023).
- [62] J. Li, G. Li, Y. Zhao, Y. Li, Z. Jin, H. Zhu, H. Liu, K. Liu, L. Wang, Z. Fang, et al., Deeval: Evaluating code generation in practical software projects, arXiv preprint arXiv:2401.06401 (2024).
- [63] Z. Han, X. Li, Z. Xing, H. Liu, Z. Feng, Learning to predict severity of software vulnerability using only vulnerability description, in: 2017 IEEE International conference on software maintenance and evolution (ICSME), IEEE, 2017, pp. 125–136.
- [64] S. Ganesh, T. Ohlsson, F. Palma, Predicting security vulnerabilities using source code metrics, in: 2021 Swedish workshop on data science (SweDS), IEEE, 2021, pp. 1–7.

Chaoyang Gao is currently pursuing the Master degree at the School of Artificial Intelligence and Computer Science, Nantong University. His research interests include software repository mining.

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

<https://xchencs.github.io/index.html>.

Guangbei Zhang is currently pursuing the Bachelor degree

at the School of Artificial Intelligence and Computer Science, Nantong University. Her research interests include automatic vulnerability assessment.