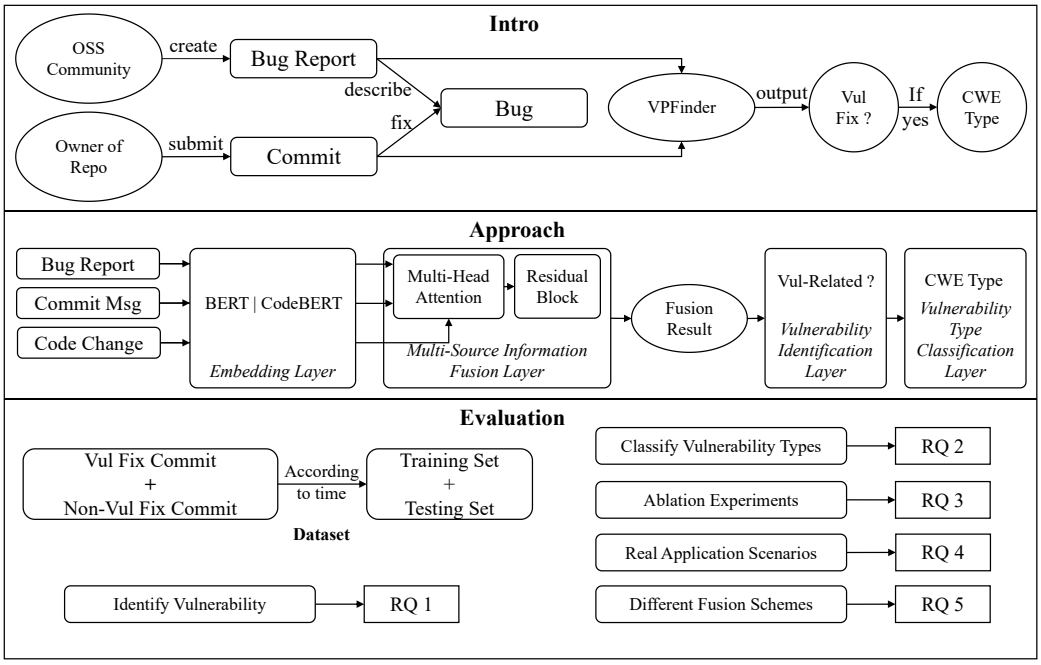
Graphical Abstract

Vulnerability Identification by Harnessing Inter-connected Multi-Source Information

Liyou Chen, Hailong Sun, Xiang Gao, Lin Shi, Yixin Yang, Yi Xu



Highlights

Vulnerability Identification by Harnessing Inter-connected Multi-Source Information

Liyou Chen, Hailong Sun, Xiang Gao, Lin Shi, Yixin Yang, Yi Xu

- We propose VPFinder, a deep learning based method that identifies vulnerabilities by harnessing interconnected multi-source information. By learning and fusing the high-level semantic information from bug reports, commit messages and patches. VPFinder could effectively recognize vulnerabilities and their corresponding types.
- We evaluate VPFinder and the experimental results show that VPFinder achieves an F1 score of 0.941 in vulnerability prediction and 0.610 in vulnerability type prediction, outperforming state-of-the-art approaches.
- We make our dataset, tool, and model publicly accessible at: https://anonymous.4open.science/r/VPFinder-5CE4

Vulnerability Identification by Harnessing Inter-connected Multi-Source Information

Liyou Chen[a], Hailong Sun[a,b,*], Xiang Gao[a,b,*], Lin Shi[a], Yixin Yang[a], Yi Xu[a]

[a]*State Key Laboratory of Complex & Critical Software Environment (CCSE), Beihang University, Beijing, 100191, China*
[b]*Hangzhou Innovation Institute of Beihang University, Hangzhou, 310056, China*

Abstract

The utilization of third-party open-source libraries is widespread in modern software development. Due to the dependency relationships, vulnerabilities within open-source libraries pose significant security threats to downstream software. However, the library vulnerabilities are usually implicitly reported and patched, without explicit notification to dependent software, leaving the downstream software vulnerable to potential attacks. Existing research efforts primarily focus on identifying vulnerability patches according to bug reports, commit messages, or code changes, overlooking the rich semantic connections among various sources of information. In this paper, our main insight is that various sources of information, including the vulnerability descriptions (e.g., bug reports) and its fixing strategies (e.g., commit messages and code changes), are highly interconnected. They express the high-level semantic information about the symptom, root cause and fixing strategies of the bugs. Hence, we propose an approach that involves training an AI model to integrate multiple sources, thus enhancing the effectiveness of vulnerability identification and vulnerability type classification. We introduce VPFinder, a tool that utilizes multi-head attention mechanisms to extract high-level semantic information from diverse sources. Evaluation results demonstrate that VPFinder achieves remarkable 0.941 F1-score in vulnerability identification task and 0.610 F1-score in vulnerability type classification task,

*Corresponding authors.
Email addresses: chenliyou@buaa.edu.cn (Liyou Chen), sunhl@buaa.edu.cn (Hailong Sun), xiang_gao@buaa.edu.cn (Xiang Gao), shilin@buaa.edu.cn (Lin Shi), yixinyang@buaa.edu.cn (Yixin Yang), yi_xu@buaa.edu.cn (Yi Xu)

outperforming state-of-the-art approaches by 5.4%.

Keywords: Open Source Software, Vulnerability, Patch, Deep Learning, Attention Mechanism

1. Introduction

Open source software (OSS) plays a pivotal role in modern computing ecosystem, ranging from infrastructural software systems to application programs across various domains (Duck, 2025). According to Gartner (Gartner, 2023), most modern software systems are assembled rather than developed from scratch, and over 70% of new in-house applications are expected to rely on open-source components by 2025. Despite its benefits, OSS is frequently affected by vulnerabilities, which can lead to malicious attacks, privacy leakage, system failures, service unavailability, etc, thereby posing significant risks to downstream software systems (Zhang et al., 2021).

In practice, when issues are discovered in OSS projects, developers typically submit bug reports to describe, discuss, and track the corresponding problems. These reports may involve functional bugs, feature requests, documentation issues, or security vulnerabilities. Vulnerability-related bug reports are not always explicitly labeled, making it difficult for downstream developers to recognize potential security risks in a timely manner. Many such reports are initially submitted to publicly accessible issue tracking systems (e.g., GitHub Issues) before being officially disclosed (Li et al., 2024b). Once the repository owners are aware of a reported vulnerability, they may submit a commit to fix it or merge a pull request from the community.

Although vulnerabilities may eventually be fixed through commits or merged pull requests, such fixes are often silently patched without explicitly notifying downstream users (Wang et al., 2019; Sun et al., 2023). As a a result, downstream software may remain exposed during the time window between the vulnerability fix and the official release(Zhang et al., 2021). Given the vast number of issue reports and commits in modern OSS ecosystems, it is impractical for developers to manually inspect each commit to determine whether it is related to a security vulnerability, let alone assess the vulnerability's type or urgency. With the identified vulnerability-related bug reports with patches, we can send timely security alerts based on the vulnerability's urgency for downstream software developers and encourage them to take proper actions, such as disabling the vulnerable functionality,

switching to another dependency, fetching the vulnerability patch or updating dependencies. Therefore, it is highly required to design an automated approach that can facilitate timely responses to potential security threats, thereby enhancing downstream software security and reliability.

Existing studies have proposed automated approaches to identify vulnerability-related bug reports or patches by leveraging information from bug reports, commit messages, code changes, or combinations thereof (Nguyen-Truong et al., 2022; Sun et al., 2023; Zhou et al., 2021b,a; Sabetta and Bezzi, 2018; Zhou and Sharma, 2017; Nguyen et al., 2022; Pan et al., 2022; Zhang et al., 2023; Jiang et al., 2024; Zhou et al., 2023; Nguyen et al., 2023; Chen et al., 2025; Cheng et al., 2022). While these approaches have demonstrated promising results, they generally suffer from two limitations. First, methods relying on single or limited information sources may fail to capture the rich semantic context of vulnerabilities. Second, even approaches that utilize multiple sources often treat them independently or combine their predictions at a late stage, which limits their ability to model the intrinsic relationships between vulnerability descriptions and corresponding fixes. Consequently, the accuracy of vulnerability identification and vulnerability type classification remains insufficient, leading to false alarms and reduced trust from downstream developers.

Similar challenges have been observed and successfully addressed in other research domains, such as multi-modal sentiment analysis (Zadeh et al., 2017; Tsai et al., 2019; Wang et al., 2025), where semantic cues are inherently distributed across heterogeneous modalities rather than being confined to a single source. Prior studies demonstrate that explicitly modeling cross-modal semantic interactions—rather than simply concatenating features or aggregating predictions—enables models to capture fine-grained dependencies and significantly improves performance. These findings suggest that understanding complex phenomena often requires jointly reasoning about multiple, interdependent information sources. Inspired by this insight, our main insight is that vulnerability-related information from different sources–such as issue reports, commits messages, and code patches–is inherently interconnected, reflecting complementary perspectives of the same security problem. Jointly modeling both the problem descriptions and their corresponding solutions can provide a more comprehensive understanding of vulnerability characteristics, root causes, and fix strategies. However, existing multi-source approaches largely adopt implicit or shallow fusion strategies, such as feature concatenation or decision-level aggregation, which limits their ability

to explicitly model fine-grained semantic interactions between vulnerability descriptions and corresponding code fixes.

In response to the above challenge, we introduce VPFinder, a deep learning-based approach for identifying vulnerability patches by harnessing interconnected multi-source information. VPFinder consists of four key layers, including *embedding layer*, *multi-source information fusion layer* , *vulnerability identification layer* and *vulnerability type classification layer*. The fusion module leverages transformer encoder (Devlin et al., 2018; Feng et al., 2020) to embed both textual and code data. The vulnerability identification layer forecasts the relevance between the vulnerabilities and multi-source information. If there is a correlation, VPFinder further proceeds to identify its vulnerability categories; Otherwise, the process is terminated. To direct the model's attention toward key vulnerability information and ignore unimportant noises, multi-source information fusion layer applies multi-head attention (Vaswani et al., 2017) to enhance the expression of key concepts in both text and code. In the vulnerability type classification layer, VPFinder will predict the vulnerability type.

To evaluate VPFinder's effectiveness, we measure it on the dataset built on top of *Pan et al.*'s dataset (Pan et al., 2022), PatchDB (Wang et al., 2021) and Java dataset (Nguyen et al., 2023; Zhou et al., 2021a).We compare VPFinder with state-of-the-art baselines: MemVul (Pan et al., 2022), VulFixMiner (Zhou et al., 2021a), VulCurator (Nguyen et al., 2022), SPI (Zhou et al., 2021b), *Sun et al.*'s work (Sun et al., 2023), and TreeVul (Pan et al., 2023). Experimental results demonstrate that the proposed approach achieves an F1-score of 0.941 in the vulnerability identification task, with a 0.061 improvement over the best baseline MemVul (Pan et al., 2022). In terms of vulnerability type classification task, VPFinder achieves an F1-score of 0.610, with a 0.054 improvement over the best baseline TreeVul (Pan et al., 2023). Ablation studies further validate the usefulness of each component of VPFinder's architecture. Experiments on the openEuler dataset demonstrate that certain components of VPFinder also achieves promising performance in real-world scenarios. Comparative experiments with other fusion methods (e.g., voting strategies and embedding vector concatenation) confirm the superior performance of our fusion approach. Experimental results under varying noise ratios demonstrate that VPFinder can effectively filter out irrelevant information. In summary, this work makes the following contributions:

- We propose VPFinder, a deep learning based method that identifies vulnerabilities by harnessing interconnected multi-source information. By learning and fusing the high-level semantic information from bug reports, commit messages and patches. VPFinder could effectively recognize vulnerabilities and their corresponding types.
- We evaluate VPFinder and the experimental results show that VPFinder achieves an F1 score of 0.941 in vulnerability prediction and 0.610 in vulnerability type prediction, outperforming state-of-the-art approaches.
- We make our dataset, tool, and model publicly accessible at: https://anonymous.4open.science/r/VPFinder-5CE4

2. RELATED WORK

Vulnerability Patch Identification.. To automatically identify vulnerability patches, many recent studies learn from different sources, including (1) code changes (Sabetta and Bezzi, 2018; Wang et al., 2019; Zhou et al., 2021a), (2) code commit and code change (Zuo and Rhee, 2024), (3) bug report and commit message (Zhou and Sharma, 2017), (4) bug report and code change (Zhou et al., 2021b). To extract the key feature from these sources, they propose to rely on (1) different machine learning techniques (Zhou and Sharma, 2017; Sabetta and Bezzi, 2018; Wang et al., 2019), such as probability-based K-fold stacking algorithm, (2) word2vec, code2vec or transformers (Zhou et al., 2021a; Sun et al., 2023; Zhou et al., 2021b) such as BERT and CodeBERT. For instance, *Wang et al.* (Wang et al., 2019) first establish a security patch database and then identify 61 code features manually, and utilize machine learning, and code clone detection to identify silent security patches. *Zhou et al.* (Zhou et al., 2021a) introduce VulFixMiner, a Transformer-based method that automatically extracts semantic meaning from commit-level code changes to identify silently patched vulnerabilities. *Nguyen et al.* (Nguyen et al., 2022; Nguyen-Truong et al., 2022) proposed VulCurator, leveraging deep learning on richer sources of information, including commit messages, code changes, and issue reports for vulnerability-fixing commit classification. Although VulCurator has examined and proved the effectiveness of fusing the issue reports and patches for vulnerability identification, VulCurator only trains three classifiers for each input including the issue report, commit message, and patch, and then gets the final result with logistic regression relying on the three classifiers' outputs. While

simply combining different sources is not efficient, compared to VulCurator, VPFinder fuses the inputs before making a classification so VPFinder is capable of uncovering the potential relationships among multi-source information. *Nguyen et al.* (Nguyen et al., 2023) proposed MiDas. MiDas constructs different neural network models for commit-level, file-level, hunk-level, and line-level, and ultimately combines these base models to generate the final prediction. *Wang et al.* (Wang et al., 2023) proposed GraphSPD, a learning-based model that represents patches as graphs with richer semantics and utilizes a patch-tailored graph model for detection. GraphSPD takes the pre-patch and post-patch source code as inputs. By merging two code property graphs for them, the model predict whether the patch is vulnerability-related or not with graph convolution and multi-layer perceptron. *Sun et al.* (Sun et al., 2023) propose a framework utilizing an encoder-decoder model with a binary detector to provide explainable predictions for silent dependency alerts. The model generates key aspects, including vulnerability type, root cause, attack vector, and impact, enhancing credibility and user acceptance. *Zhou et al.* (Zhou et al., 2021b) propose an automated security patch identification system to collect security patches in open-source software. The system also uses Transformer-based models and comprises two neural networks, one focusing on commit messages and the other on code changes. Different from those works that only identify the existence of vulnerabilities, VPFinder also recognizes the vulnerability type. Moreover, VPFinder fuses all the valuable information and tries to learn the latent connection among them to increase the predication accuracy.

Vulnerability Type Identification.. To categorize vulnerability types of security patches, *Pan et al.* (Pan et al., 2023) firstly propose to utilize code changes to classify security patches into categories at the third level of the CWE tree. *Le et al.* (Le et al., 2022) introduce a deep multi-task learning model automating seven commit-level vulnerability assessment tasks based on Common Vulnerability Scoring System metrics. *Zhou et al.* (Zhou et al., 2023) propose an explainable silent vulnerability fix identification framework to learn code changes and demonstrate superior performance in silent fix identification and CWE category classification. In general, different from this kind of work, we not only consider the commit message and code patch but also make use of the detailed information in the bug report. This information can reflect the symptoms and root causes of the vulnerability, which can greatly improve the effect of vulnerability classification.

Classifying vulnerability-related bug report.. Some work provides early warnings through automated identification of risky bug reports. *Peters et al.* (Peters et al., 2017) address the challenge of accurately identifying unlabelled security bug reports in bug tracking systems, and they propose a framework that filters and ranks reports to reduce the impact of security-related keywords. *Shu et al.* (Shu et al., 2021) propose a dual optimizer to optimize models distinguishing between security bug reports and other bug reports. *Pan et al.* (Pan et al., 2022) introduce MemVul to promptly identify bug reports that may lead to information disclosure. Leveraging a memory component to store vulnerability knowledge, MemVul employs an encoder, feedforward neural network, and voting mechanism to identify vulnerability-related bug reports. Once the vulnerability-related reports are recognized, their patches can be easily identified as vulnerability patches. Similar to these works, VPFinder can also provide early warning of vulnerabilities, although the prediction accuracy can be improved with the committed patches as inputs. The advantage of our work is that richer information can be used to improve the performance of vulnerability identification and vulnerability type prediction.

***Positioning of* VPFinder**. Existing methods such as VulCurator, MiDas, and GraphSPD leverage multiple sources of information, including commit messages, code changes, and issue reports, for vulnerability patch identification. However, these approaches either combine sources at a late stage (e.g., VulCurator uses separate classifiers per source and merges outputs with logistic regression), or rely on specialized representations (e.g., GraphSPD constructs patch-level graphs) without explicitly modeling the semantic interactions among heterogeneous sources. The comparison of different methods is shown in Table 1.

In contrast, VPFinder directly fuses multiple sources through a multi-head attention mechanism, enabling the model to capture latent correlations between commit messages, bug reports, and code patches prior to classification. This design allows VPFinder to jointly consider textual descriptions and code changes, uncovering cross-modal dependencies that are ignored by prior work. Furthermore, VPFinder extends the task beyond simple vulnerability detection by also predicting the CWE type, leveraging the fused representation to improve classification accuracy. Ablation studies confirm that both the fusion mechanism and the inclusion of all three sources contribute significantly to the overall performance, highlighting the method's

Table 1: Comparison of different methods.

Method	Input Sources	Fusion Strategy	Output	Notes on Limitations
VulCurator	Issue Report, Commit msg, Code Patch	Late fusion (logistic)	Vulnerability related or not	Ignores cross-modal interactions
MiDas	Commit/Line/ File/Hunk levels code	Multi-level neural networks	Vulnerability related or not	Separate models, merged later
GraphSPD	Pre/post-patch code	Graph convolution	Vulnerability related or not	No textual info fusion
VPFinder	Issue Report, Commit msg, Code Patch	Multi-head attention	Vulnerability related or not + CWE type (If related)	Early fusion, cross-modal correlations captured

novelty and practical effectiveness.

3. Motivating Example

We detail our motivation and overall idea by presenting a motivating example in this section.

FFJPEG[1] is a lightweight JPEG codec implemented purely in C, designed for embedded and resource-constrained systems. Unlike traditional libraries like libjpeg, it eliminates complex dependencies while providing core JPEG compression/decompression functionality, making it ideal for studying JPEG algorithm or optimizing compact applications. A user found a vulnerability and submitted a bug report[2]. The developers committed a patch to solve this vulnerability. This vulnerability corresponds to CWE-125(out-of-bound read, child of CWE-118). The key information of the report and patch is shown in Figure 1. Specifically, the first hunk introduces a null pointer check (*if (!jfif) return ret;*), which can be regarded as a robustness or defensive programming measure. However, the bug report does not describe a null pointer dereference scenario, nor does it indicate that an attacker-controlled input could trigger such a fault. As a result, this modification is not causally

[1]https://github.com/rockcarry/ffjpeg
[2]https://github.com/rockcarry/ffjpeg/issues/23

Commit Message: fix issue #23.

scr/jfif.c

```
@@ -264,6 +264,7 @@ int jfif_save(void *ctxt, char *file)
264 264     int   i, j;
265 265     int   ret  = -1;
266 266
    267     if (!jfif) return ret;
267 268     fp = fopen(file, "wb");
268 269     if (!fp) goto done;
@@ -730,9 +731,9 @@ void* jfif_encode(BMP *pb)
730 731     // init jw & jw, init yuv data buffer
731 732     jw = ALIGN(pb->width , 16);
732 733     jh = ALIGN(pb->height, 16);
733     -   yuv_datbuf[0] = calloc(1, jw * jh / 1 * sizeof(int));
734     -   yuv_datbuf[1] = calloc(1, jw * jh / 4 * sizeof(int));
735     -   yuv_datbuf[2] = calloc(1, jw * jh / 4 * sizeof(int));
    734 +   yuv_datbuf[0] = calloc(1, sizeof(int) * jw * jh / 1);
    735 +   yuv_datbuf[1] = calloc(1, sizeof(int) * jw * jh / 4);
    736 +   yuv_datbuf[2] = calloc(1, sizeof(int) * jw * jh / 4);
736 737     if (!yuv_datbuf[0] || !yuv_datbuf[1] || !yuv_datbuf[2]) {
737 738         goto done;
738 739     }
```

Bug Report: ····· An attacker can exploit this vulnerability by submitting a malicious bmp that exploits this bug which will result in a Denial of Service (DoS).

CWE-125: Out-of-bounds Read.

The product reads data past the end, or before the beginning, of the intended buffer.

Figure 1: An example bug report and commit

linked to the reported security impact. In contrast, the second hunk modifies the memory allocation pattern in calloc by changing the multiplication order involving jw and jh. This change directly mitigates an integer overflow that can lead to an undersized buffer allocation, which in turn causes out-of-bounds reads when processing a malicious BMP input. This behavior precisely matches the exploit scenario described in the bug report (i.e., a malicious image leading to Denial of Service), and thus constitutes the actual vulnerability fix corresponding to CWE-125. Therefore, while we acknowledge that the commit may fix multiple issues, our analysis focuses on identifying which code changes are directly related to the reported vulnerability and its type. This distinction highlights the difficulty faced by existing tools when commits include mixed-purpose fixes, and motivates our approach to leverage bug reports and patches jointly to correctly associate vulnerability semantics with the relevant code changes.

Existing tools, that identify vulnerability patches according to the commit message and changed code, failed to identify the vulnerability in this example. For the first patch, due to the lack of context to determine whether the "jfif"

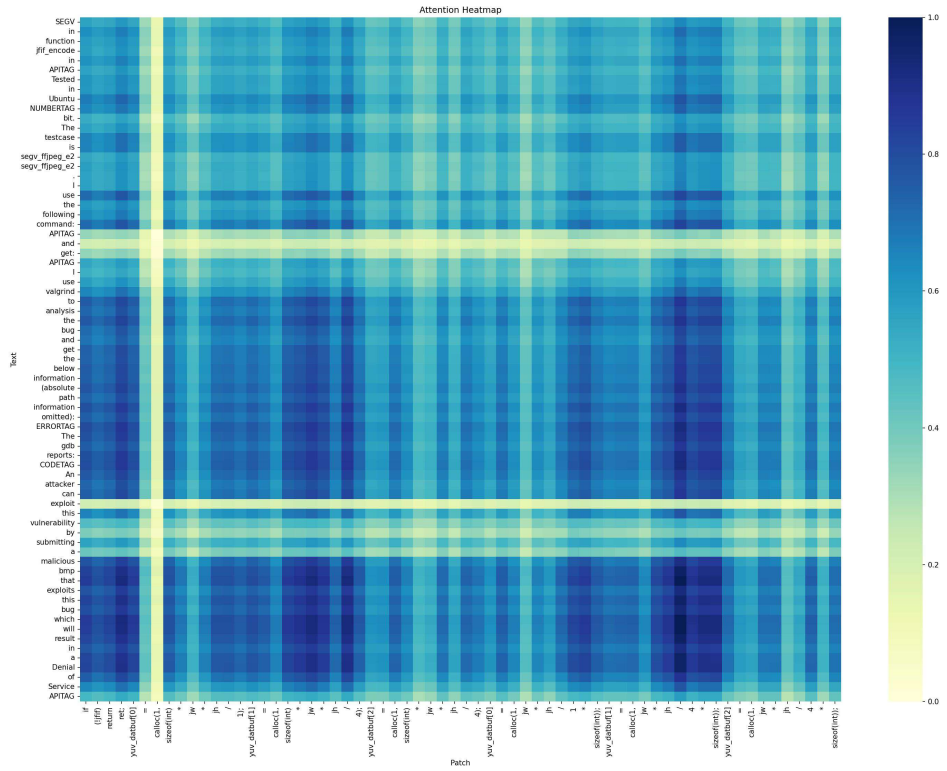


Figure 2: Attention heatmap in VPFinder.

came from external input, there was a 60-70% probability that they would consider it a fix for a regular bug – a case of defensive programming (Nguyen et al., 2022; Zhou et al., 2021a). For the second patch, there was an 85-90% probability they would view it as merely a bug fix (Nguyen et al., 2022; Sun et al., 2023), an adjustment related to code style/readability, since the total allocated memory size remained unchanged. After referring to the bug report information, it was easy to determine that this as a vulnerability. Besides, even if existing tools can recognize this was a vulnerability patch, it was challenging to automatically determine its vulnerability type since the bug report, commit message and patch did not mention anything about out-of-bound read. Recognizing vulnerability type was even more important since it could give developers some hints about the severity of the vulnerability. According to the information given in the bug report, we can observe this vulnerability can result in DoS. DoS attacks are typically associated with resource exhaustion (CWE-400 – Uncontrolled Resource Consumption,

belongs to the same category level of CWE-118) and insufficient input validation (CWE-20 – Improper Input Validation, child of CWE-707). However, the bug report included irrelevant information (Figure 1 just showed partially important content), and such noise might prevent us from correctly matching the bug with the corresponding CWE. We observed existing tools, including VulCurator (Nguyen et al., 2022), *Sun et al.*'s work (Sun et al., 2023), and TreeVul (Pan et al., 2023), failed to recognize the correct vulnerability type for this bug. They would consider it a null pointer dereference (according to the first patch). However, during the execution of "calloc" in the second patch, the calculation method "jw * jh / 4" could lead to a integer overflow, resulting in a smaller buffer allocation than expected. This could later cause out-of-bounds reads when accessing the buffer. Modifying the "sizeof(int)" part might make the calculation safer by preventing the overflow or ensuring proper memory allocation.

VPFinder takes various sources of information into consideration. Specifically, VPFinder relies on multi-head attention mechanisms to highlight the shared semantics between the bug report and the patch. Figure 2 presents the normalized[3] attention heatmap from VPFinder's attention mechanism, where the x-axis represents patch content (comprising both added and deleted code segments) and the y-axis corresponds to bug report content. The visualization demonstrates that VPFinder's attention focuses on two key correlations: (1) vulnerability-related description("malicious bmp that exploits this bug which will result in a Denial of Service") in the bug report, and (2) specific code modifications in the patch that address out-of-bounds read vulnerabilities, particularly the "sizeof(int)" and the "jw * jh / 4" calculation adjustment in the patch (with the vulnerability first appearing at line 734 (deleted) and line 735 (added)). By leveraging the attention mechanism, VPFinder establishes a connection between the vulnerability description in the bug report and the "calloc" calculation pattern in the patch. It then predicts the vulnerability type as CWE-118: Incorrect Access of Indexable Resource ('Range Error') – a parent category of CWE-125 (Out-of-Bounds

[3]During the encoding phase, we employed "padding" to handle insufficient data length. Since we did not prepare attention masks for the multi-head attention mechanism, the padded data inevitably consumed attention weights in the computation process. In the heatmap visualization we presented, we specifically extracted the portion of the weight matrix corresponding to the actual data length (excluding padding positions) before performing normalization.

Read). So with the highlighted information, it can then determine whether this bug is relevant to buffer out-of-bound read, hence recognizing the correct CWE type. In general, the bug report provide insights into the symptoms and root cause of the vulnerability, while the commit message and patch code offer details on the vulnerability's fixing strategy. Integrating various types of information helps VPFinder effectively identify vulnerabilities and their types.

Finally, we would like to clarify that the attention weights in VPFinder are not intended to indicate which specific code hunk constitutes the root-cause vulnerability fix. Instead, they reflect the degree of semantic relevance between elements of the bug report and different parts of the patch. In this example, although the first hunk does not address the root cause of the out-of-bounds read, it introduces a defensive null check that is semantically related to the vulnerability symptoms described in the bug report, such as program crashes and denial-of-service behavior. This explains why the attention mechanism assigns non-negligible weights to the first hunk. Importantly, the actual root cause of the vulnerability—an integer overflow in the buffer size computation leading to out-of-bounds reads—is only addressed in the second hunk. While both hunks are semantically relevant at different levels, the second hunk provides the critical information required for inferring the correct vulnerability type. VPFinder aggregates information across all hunks and modalities, and the final prediction is not determined by attention to a single patch fragment. Consequently, the presence of attention on the first hunk does not indicate misclassification, but rather reflects the model's ability to capture contextual and symptom-level relevance in addition to root-cause signals. This behavior is consistent with real-world development practices, where vulnerability-fixing commits often include auxiliary defensive changes alongside the actual root-cause fix, and we believe it demonstrates the advantage of modeling cross-modal semantic relationships rather than relying on isolated code patterns.

4. APPROACH

To identify vulnerabilities and their type, we propose a deep learning based approach VPFinder. The inputs of VPFinder include two main parts: (1) **bug description**: the problem description from the bug report; (2) **bug fix**: the commit message and the patch. The output is whether the report or patch is related to vulnerability, and vulnerability type (if

applicable). We first present the overall architecture of VPFinder, and then detail each step.

4.1. Overall Architecture

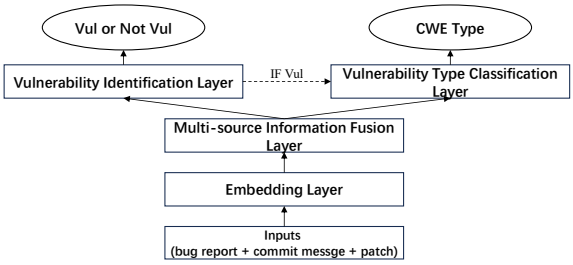


Figure 3: The overall architecture of VPFinder.

The overall structure of VPFinder is illustrated in Figure 3, including four main layers: (1) embedding layer, (2) multi-source information fusion layer, (3) vulnerability identification layer, and (4)vulnerability type classification layer. First, VPFinder takes bug description, bug fix and patch as inputs, which are then fused together to better represent the features of the bug in the embedding layer. Specifically, VPFinder utilizes two encoders to learn the vector representations of the text and code respectively. Second, in the multi-source information fusion layer, VPFinder employs multi-head attention and residual blocks (He et al., 2015) to capture high-level semantic relationships among multi-source information. In the vulnerability identification layer, VPFinder integrates the fusion results with preliminary classification outputs from the embedding layer to predict the relevance between multi-source information and vulnerabilities. If the bug is identified as a vulnerability in vulnerability identification layer, the vulnerability type classification layer will continue classifying the associated CWE types; otherwise, the process concludes.

4.2. Embedding Layer

The fusion of multi-source information dedicates to fusing the high-level semantic bug information from multiple sources. Given N samples in the dataset, as shown in Equations (1)-(4), this module takes the problem description $X_{description_i}$, commit message $X_{message_i}$ and patch X_{patch_i} as inputs, for $0 < i <= N$. Patch vector X_{patch_i} consists of two parts: the added code

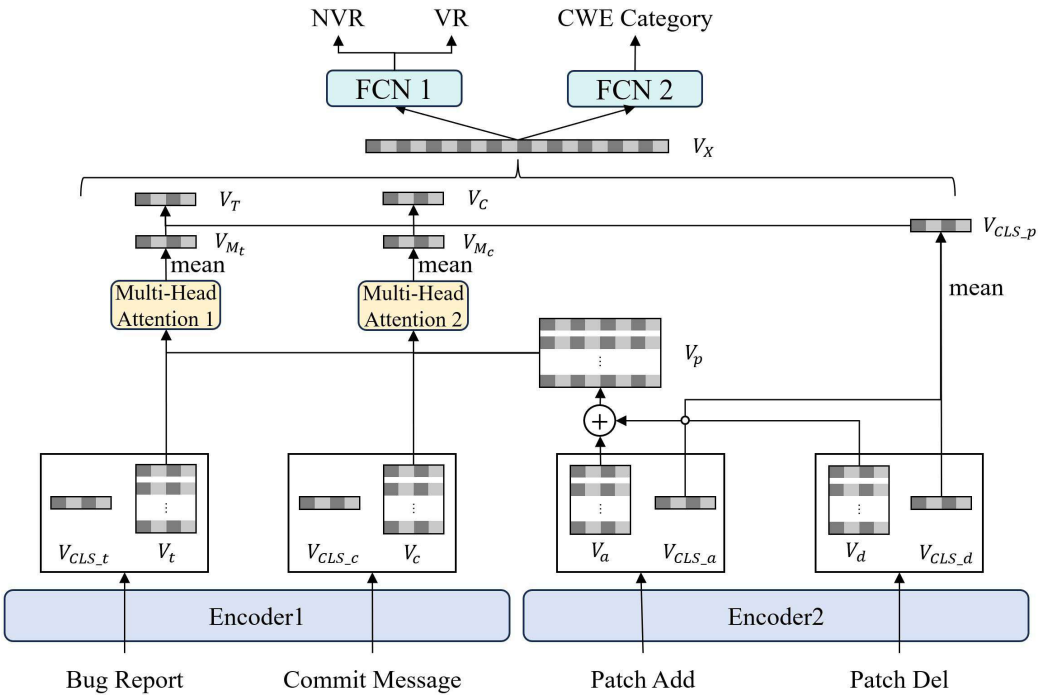


Figure 4: The overall framework of VPFinder.

$X_{patch\ add_i}$ and the deleted code $X_{patch\ del_i}$. We encode the text-based inputs such as problem descriptions and commit messages with `Encoder1`, and encode code-based inputs such as patch with `Encoder2`. `Encoder1` can be any tool like BERT that can process text information and `Encoder2` can be any tool like CodeBert that can process code and annotation information.

$$(V_{t_i}, V_{CLS\_t_i}) = f_{Encoder1}\left(X_{description_i}\right) \tag{1}$$

$$(V_{c_i}, V_{CLS\_c_i}) = f_{Encoder1}\left(X_{commit\ msg_i}\right) \tag{2}$$

$$(V_{a_i}, V_{CLS\_a_i}) = f_{Encoder2}\left(X_{patch\ add_i}\right) \tag{3}$$

$$(V_{d_i}, V_{CLS\_d_i}) = f_{Encoder2}\left(X_{patch\ del_i}\right) \tag{4}$$

The problem descriptions are embedded to yield matrix: $(V_{t_i}, V_{CLS\_t_i})$, where V_{t_i} denotes the hidden states of `Encoder1` at each position, and $V_{CLS\_t_i}$ is a single vector representing the entire text. Similarly, `Encoder1` is also applied to commit messages. Moreover, `Encoder2` is applied to yield patch representation. The [CLS] token of each encoder output has rich semantic

information so it is capable of vulnerability identification by simply fusing. However, for vulnerability type classification, just using tokens is not enough. The rich semantic information needs further fusion to establish potential links between these types of information. We'll discuss how to fuse multi-source information using multi-head attention and residual blocks.

4.3. Multi-Source Information Fusion Layer

As shown in Figure 4, multi-source information plays a crucial role in VPFinder, as it helps to better understand and address the vulnerabilities in each component. VPFinder utilizes multi-source information layer to establish potential connections between the multi-source information. Because bug report and commit message have a certain connection with patches, hidden states will be further used to explore the potential connection between them.

4.3.1. Processing of patches

VPFinder first concatenates the hidden states of the added code and the deleted code, as shown in Equation (5). The hidden states V_{p_i} has semantic information of the patch.

$$V_{p_i} = concat(V_{a_i}, V_{d_i}) \tag{5}$$

Then VPFinder takes the arithmetic mean of the added code and deleted code to obtain the semantic information of the patch, as shown in Equation (6). The obtained average patch semantic representation will be used in the residual blocks (He et al., 2015) to prevent gradient vanishing and gradient explosion.

$$V_{CLS\_p_i} = \frac{1}{2}(V_{CLS\_a_i} + V_{CLS\_d_i}) \tag{6}$$

4.3.2. Multi-Head Attention Layer

The bug report and commit message may include a large part of the irrelevant content to the patch, which may significantly affect the vulnerability identification and classification accuracy. To solve this problem, VPFinder applies attention mechanism (Vaswani et al., 2017) to highlight the patch content by measuring the shared semantic similarities between them. The multi-head attention mechanism is built on the basis of self-attention (Vaswani

et al., 2017), and captures different aspects of information in the input sequence by computing multiple attention heads in parallel. Each head focuses on a different subspace of the input data, which helps the model understand the input information from multiple perspectives and improves the model's ability to express and generalize. The results of all heads are then spliced and then subjected to linear transformation to obtain the final output. Specifically, relying on an attention mechanism, we propose to treat the hidden state of bug report V_{t_i} as *Query*, and the information at each position of the patch sequence as *Key* and *Value*. As shown in Equation (7), Q, K, and V represent *Query*, *Key*, and *Value*, respectively.

$$Attention\left(\mathrm{Q}{=}V_{t_i}, \mathrm{K}{=}V_{p_i}, \mathrm{V}{=}V_{p_i}\right) {=} softmax\left(\frac{\mathrm{Q}\cdot\mathrm{K}^T}{\sqrt{d_k}}\right)\mathrm{V} \tag{7}$$

Query and *Key* are dot multiplied to obtain the similarity between them. The similarity is then scaled using $\sqrt{d_k}$, where d_k is the dimensions of the *Key* vector. Then, the $softmax$ operation is performed to convert the similarity into the weight of $[0, 1]$ interval. A higher similarity between the *Query* and the *Key* results in a higher attention weight. Finally, the weight is multiplied by the *Value* to obtain the weighted results. In this process, the parts with similar semantics between buggy code and problem description will receive more "attention".

VPFinder then trains a multi-head attention (Vaswani et al., 2017), as shown in Equation (8), where W_j^{Q}, W_j^{K}, W_j^{V} and W^{O} are trainable parameters. $j \in (0, J)$ represents the j-th head and J represents the number of heads.

$$\begin{aligned} MultiHead\left(\mathrm{Q},\mathrm{K},\mathrm{V}\right) &= concat\left(head_1, ..., head_J\right) W^{\mathrm{O}} \\ \texttt{where}\ head_j{=}&Attention\left(\mathrm{Q}\cdot W_j^{\mathrm{Q}}, \mathrm{K}\cdot W_j^{\mathrm{K}}, \mathrm{V}\cdot W_j^{\mathrm{V}}\right) \end{aligned} \tag{8}$$

Specifically, multi-head attention divides the input into multiple heads, with each head learning different weights. As shown in Equation (8), in multi-head attention, each attention head has its own set of weight matrices W_j^{Q}, W_j^{K} and W_j^{V} designed to associate different parts of the input sequence. Each head attention score is then calculated using Equation (7) for training stability. Finally, all head outputs are concatenated, forming the output of the multi-head attention. Further matrix multiplication is applied through a linear projection matrix W^{O} to obtain the final output of the multi-head attention. Linear projection allows the model to better capture information

from different positions when dealing with sequential data, fusing both performance and generalization capabilities. To fuse bug report and patch, we calculate multi-head attention $M_{t_i} = MultiHead\,(V_{t_i}, V_{p_i}, V_{p_i})$, which corresponds to `Multi-Head Attention 1` in Figure 4. By calculating attention scores for V_{p_i} based on the V_{t_i}, and then performing a weighted sum on V_{p_i}, the semantics related to vulnerabilities are enhanced. The output is matrix M_t, where each row represents the semantic information of the patch sequence at each position. Average pooling helps in extracting global information from the entire sequence and reducing dimensionality, aiding the model in learning the overall features of the sequence. So an average pooling is performed on the output matrix to obtain a smooth semantic representation of the patch, as shown in Equation (9)-(10), where $M_{t_{i_{jk}}}$ represents the element in the k-th row and j-th column of matrix M_{t_i}, and m represents the number of rows in matrix M_{t_i}.

$$\bar{V}_{M_{t_i}} = \frac{1}{m}\sum_{j=1}^{m} M_{t_{i_{jk}}} \tag{9}$$

$$V_{T_i} = concat(\bar{V}_{M_{t_{i_1}}}, ..., \bar{V}_{M_{t_{i_m}}}) \tag{10}$$

Moreover, VPFinder also applies the same attention approach to fuse the commit message information with the similarity between commit message and patch. Specifically, we calculate the multi-head attention $M_{c_i} = MultiHead\,(V_{c_i}, V_{p_i}, V_{p_i})$, corresponding to `Multi-Head Attention 2` in Figure 4. In the same way, VPFinder also utilizes average pooling to obtain the smooth semantic representation of the patch, as shown in Equation (11)-(12).

$$\bar{V}_{M_{c_i}} = \frac{1}{m}\sum_{j=1}^{m} M_{c_{i_{jk}}} \tag{11}$$

$$V_{C_i} = concat(\bar{V}_{M_{c_{i_1}}}, ..., \bar{V}_{M_{t_{i_m}}}) \tag{12}$$

4.3.3. Residual Block

Residual blocks (He et al., 2015), a cornerstone of modern convolutional neural network (CNN) architectures, have emerged as a pivotal solution to the long-standing issue of training very deep networks. These blocks incorporate shortcut connections that enable the input of a block to be directly added to its output, bypassing the series of convolutions and non-linear transformations within the block. This design principle, known as residual learning,

addresses the degradation problem encountered in traditional deep networks, where the addition of layers often leads to higher training error. By facilitating the flow of gradients during backpropagation and enabling features from earlier layers to propagate deeper into the network, residual blocks enable the construction of networks with unprecedented depths, thereby pushing the boundaries of what is achievable in computer vision tasks such as image classification, object detection, and semantic segmentation. The effectiveness of residual blocks has been amply demonstrated through their widespread adoption in state-of-the-art CNN architectures (Li et al., 2024a), including the seminal ResNet family, which has inspired numerous follow-up works (He et al., 2016; Wang et al., 2017) that further explore and refine the concept of residual learning.

Through the skip connection in residual block, the features of the previous layer can be directly used by the subsequent layers, which helps the network to share and reuse features between different layers and improves the efficiency of the network. This idea can also be used in VPFinder, as shown in Equation (13)-(14). Residual blocks helps reuse features and increase depth and complexity of the network.

$$V_{T_i}^{'} = V_{T_i} + V_{CLS\_p_i} \tag{13}$$

$$V_{C_i}^{'} = V_{C_i} + V_{CLS\_p_i} \tag{14}$$

Finally, VPFinder concatenates the residual blocks' outputs and all [CLS] token vectors, as shown in Equation (15).

$$V_{X_i} = cancat(V_{T_i}^{'}, V_{C_i}^{'}, V_{CLS\_t_i}, V_{CLS\_c_i}, V_{CLS\_p_i}) \tag{15}$$

Up to this point, we have completed the fusion of multi-source information, obtaining the final vector representation V_{X_i}.

4.4. Vulnerability Identification Layer

As shown in Fig 4, to identify whether the bug report with relevant commits are related to vulnerability, VPFinder concatenates each [CLS] token output by BERT or CodeBERT and utilizes a fully-connected neural network FCN_1. FCN_1's output is the probability (P_{vul_i}) representing whether this bug is associated with vulnerability. The above process is shown in Equation (16).

$$P_{vul_i} = FCN_1(V_{X_i}) \tag{16}$$

Then VPFinder generates the prediction result via Equation (17), where $\tilde{z}_i$ denotes the label of the i-th sample predicted by the model and $\tilde{z}_i = 1$ indicates that this sample is predicted to be related to the vulnerability.

$$\tilde{z}_i = \arg\max P_{vul_i} \tag{17}$$

To learn the involved parameters, the cross-entropy loss function is applied, as demonstrated in Equation (18), where z_i denotes the ground truth of i-th sample and $z_i = 1$ indicates that this bug is related to vulnerability.

$$\mathfrak{L}_1 = -\frac{1}{N}\sum_{i=1}^{N}\left(z_i \log \tilde{z}_i + (1 - z_i)\log\left(1 - \tilde{z}_i\right)\right) \tag{18}$$

To learn the involved parameters, the cross-entropy loss function is applied, as demonstrated in Equation (19), where z_i denotes the ground truth of i-th sample and $z_i = 1$ indicates that this bug is related to vulnerability.

$$\mathfrak{L}_1 = -\frac{1}{N}\sum_{i=1}^{N}\left(z_i \log \tilde{z}_i + (1 - z_i)\log\left(1 - \tilde{z}_i\right)\right) \tag{19}$$

If a bug report with relevant commits are predicted as vulnerability-related, VPFinder will further predict its vulnerability type.

4.5. Vulnerability Type Classification Layer

VPFinder utilizes another fully connected neural network FCN_2 to extract features and outputs the probability that the vulnerability corresponds to each categories, as shown in Equation (20).

$$P_{category_i} = FCN_2(V_{X_i}) \tag{20}$$

Then, VPFinder generates the classification results by Equation (21), where $\tilde{r}_i$ denotes the label of the i-th sample predicted by the model and $\tilde{r}_i = k$ indicates that this bug is predicted to be the k-th category of CWE.

$$\tilde{r}_i = \arg\max P_{class_i} \tag{21}$$

Similar to vulnerability identification task, to learn the involved parameters, we utilize the cross-entropy loss function, as depicted in Equation (22), where r_i denotes the ground truth of i-th sample and $r_i = k$ indicates that this bug belongs to k-th category of CWE.

$$\mathfrak{L}_2 = -\frac{1}{N}\sum_{i=1}^{N}\left(r_i \log \tilde{r}_i + (1 - r_i)\log\left(1 - \tilde{r}_i\right)\right) \tag{22}$$

5. EXPERIMENT

In this section, we describe our research questions, dataset, experiment setting and results in detail. Specifically, we evaluate VPFinder to answer the following research questions:

- **RQ1:** How effective is VPFinder in identifying vulnerabilities based on multi-source information?
- **RQ2:** How effective is VPFinder in classifying vulnerability types based on multi-source information?
- **RQ3:** How do architectural variations influence the performance of VPFinder?
- **RQ4:** How effective is VPFinder in real application scenarios on key software package repositories of openEuler (Huawei, 2024) dependencies?
- **RQ5:** Does VPFinder's ability to filter out irrelevant information outperform other methods?

5.1. Dataset

To evaluate VPFinder, we directly reuse the MemVul dataset provided by *Pan et al.* (Pan et al., 2022). MemVul dataset includes 3884 CVE-referred issue reports and nearly $1191k$ not CVE-referred issue reports. For each sample in this dataset, we extract its URL, problem description, and other vulnerability information. Specifically, we utilize GitHub's REST API (GitHub, 2024) to access the issue reports and relevant commits in GitHub repositories. We make an effort to find the commit associated with each issue and record the SHA value of the relevant commit. If we cannot find related commit information in the bug report, we proceed to check the discussion section of the bug report. If the commits are not available, the bug report will not be included in our dataset. Once we identify the relevant commits, we use the REST API to retrieve the patch information for the commit. Further, we extract the modified code (including added and deleted code) based on the commit. If more than one patch file exists, we merge them together. We just retain C, Java, and Python projects, and filter out samples whose fixes do not involve modifying code. After filtering out patches

and limiting the dataset to maintain a roughly 1:2 ratio of vulnerability-fixing samples to non-vulnerability-fixing samples, the final dataset contains 1,090 vulnerability-fixing and 3,385 non-vulnerability-fixing samples. The enhanced MemVul dataset that we used contains all the bug information, including flag (i.e., vulnerability identification label), text (i.e., bug description), message (i.e., commit message), patch, CVE_ID, and CWE_ID. Based on the year disclosed in the CVE ID (vulnerability-related samples) or the year that bug report created at (non-vulnerability-related samples), we categorize all samples into training set (prior to 2020) and testing set (2020).

To observe the performance of VPFinder in more real-world datasets, we also compare VPFinder with existing tools on PatchDB (Wang et al., 2021), and Java dataset (Nguyen et al., 2023; Zhou et al., 2021a). PatchDB is a large-scale security patch dataset that contains around 12K security patches and 24K non-security patches from the real world. We collected the associated issues if available. Java dataset contains vulnerability fixing and non-vulnerability fixing commits collected from the real world. Table 2 presents the statistics of the dataset. The experiments conducted on the PatchDB and Java datasets are performed using the complete datasets, and we adhere to the datasets' original training and testing splits.

Table 2: The number of samples in the dataset.

Dataset	Type	Vul	NVul	Total
MemVul	Training set	961	2041	3002
	Testing set	129	251	380
PatchDB	Training set	9658	18993	28651
	Testing set	2415	4749	7164
Java	Training set	983	31323	32306
	Testing set	300	87856	88156

For vulnerability type classification, we observe that the number of samples belonging to some CWE categories is too low. Hence, we merge the less frequent CWE categories in the dataset into composite categories. When merging minority classes, we follow the tree structure (MITRE, 2025), combining minority classes belonging to the same parent node into the parent class. For example, CWE-693, CWE-435, CWE-697, and CWE-703 belong to the first layer in the CWE tree, so we combine these CWE samples as 'CWE-1'. In addition, CWE-417 is a CWE category (communication channel errors) although it has been discouraged since 2019, so we combine it

into CWE-1, too. CWE-913, CWE-706, CWE-704, CWE-669, CWE-666, and CWE-665 belong to the second layer and they are all child nodes of CWE-664, so we combine them as 'CWE-2'. However, there are still some classes with too few samples, and we merge them into new classes. Additionally, for individual samples with undetermined CWE, we categorize these samples as 'CWE-1000'. Due to limitations in sample quantity, we chose to provide the second-layer CWE category only for 'CWE-664'. This results in eight labels at the first layer and four labels at the second layer. The CWE statistics are detailed in Table 3. The labels are exclusive, indicating that if a sample is predicted as its parent node's type, it is considered as a misprediction.

For we sacrifice some CWE granularity to gain statistical reliability, and the resulting coarse-grained classification remains meaningful and practically relevant. Merging CWE categories enables statistically reliable training and evaluation. Without this step, the performance on rare CWE classes would be dominated by noise, making it difficult to draw meaningful conclusions about model effectiveness. Importantly, while the merging operation results in a coarser-grained classification setting, the task remains meaningful for our study objectives. The merged CWE categories correspond to semantically related vulnerability families, allowing the model to capture higher-level vulnerability patterns rather than overly specific instances. As a result, the obtained results still provide valid insights into the model's ability to distinguish different classes of vulnerabilities at the family level.

5.2. Experiment Setting

5.2.1. Baselines

We compare VPFinder with state-of-the-art work, including MemVul (Pan et al., 2022), VulFixMiner (Zhou et al., 2021a), GraphSPD (Wang et al., 2023), MiDas (Nguyen et al., 2023), VulCurator (Nguyen et al., 2022), SPI (Zhou et al., 2021b), *Sun et al.*'s work (Sun et al., 2023), and Treevul (Pan et al., 2023). Specifically,

- **MemVul:** MemVul utilizes language model and memory component to predict whether a bug report is related to a vulnerability.
- **VulFixMiner:** VulFixMiner is a Transformer-based model for identifying cross-project and cross-language vulnerability fixes according to code changes.

Table 3: Dataset CWE statistics.

	CWE category	Number	Label
First layer	CWE-664	156	1
	CWE-707	90	2
	CWE-710	83	3
	CWE-682	57	4
	CWE-691	58	5
	CWE-1(merged category)	27	6
	CWE-284	33	7
	CWE-1000	26	8
Second layer	CWE-118	398	9
	CWE-404	75	10
	CWE-668	47	11
	CWE-2(merged category)	40	12

- **GraphSPD:** GraphSPD represents patches as graphs with richer semantics and utilizes a patch-tailored graph model for identifying vulnerability patches.

- **VulCurator:** VulCurator trains classifiers for bug reports, commit messages, and patches, and combines the outputs of the three classifiers using logistic regression to identify vulnerabilities. Compared to VPFinder fusing the information at the embedding-level, VulCurator fuses the information at the decision-level.

- **SPI:** SPI learns features from commit messages and patch, and selects high-level features for an information-rich, latent semantic representation. Then SPI effectively integrates the unified representation to build a classifier to determine whether a commit is a security patch or not.

- ***Sun et al.*'s work:** This model utilizes an encoder-decoder framework to identify vulnerability patches according to commit message and code changes.

- **MiDas:** MiDas constructs different base models for each level of code change granularity, corresponding to commit-level, file-level, hunk-level, and line-level, and combines all base models to output the final prediction. As same as VulCurator, MiDas fuses the information at the

decision-level. MiDas also incorporates feature fusion, while its fusion focuses on how to represent contextual information. For instance, it considers whether to treat the contextual information as a whole for internal fusion within the context or to concatenate the added and removed code into a vector for internal fusion. Both of these fusion approaches do not involve the integration of multi-source information.

- **TreeVul:** TreeVul takes as inputs commit message and changed code, and classifies vulnerability patches via a hierarchical and chained architecture.

Among them, MemVul, VulFixMiner, SPI, *Sun et al.*'s work, VulCurator, GraphSPD, and MiDas are used for comparison on the vulnerability identification task. *Sun et al.*'s work and TreeVul are used for comparison on the vulnerability type classification task. In addition, we also extended VulCurator to support vulnerability type classification task to evaluate the performance of decision-level fusion. Table 4 shows the input types used by each model and the types of tasks they can perform.

Table 4: Comparison of models.

Model	Inputs			Tasks	
	Issue Report	Commit Message	Code Patch	vulnerability identification	vulnerability type classification
MemVul	✓			✓	
VulFixMiner			✓	✓	
VulCurator	✓	✓	✓	✓	✓
SPI			✓	✓	
Sun et al.'s work		✓	✓	✓	✓
GraphSPD			✓	✓	
MiDas			✓	✓	
TreeVul		✓	✓		✓
VPFinder	✓	✓	✓	✓	✓

During the experiments, BERT (bert-base uncased, 2023) and codeBERT (codebert-base-finetuned-detect-insecure code, 2023) are selected to embed text and code, respectively. The number of parameters of these two models are relatively small compared with large language models, such as Llama2 (Touvron et al., 2023) and CodeLlama (Roziere et al., 2023), and they show great

performance in extracting semantic information of text or code. For fair comparison, we ensure uniformity in BERT or CodeBERT models employed across all baselines. Additionally, models sharing the same architecture are configured with identical hyperparameters. Due to Code2Vec's limitation to Java (Alon et al., 2019), we employed CodeBERT as the encoder for code-type data in SPI model implementation.

5.2.2. Evaluation Metrics

To evaluate the vulnerability identification ability, we divide all samples into two classes. To evaluate the vulnerability type classification ability, we filter out non-vulnerability samples, keeping only samples with 'CWE_ID' ranging from 1 to 12. To evaluate VPFinder's effectiveness, we use commonly employed metrics including precision, recall, F1-score, and AUC (Pan et al., 2022; Peters et al., 2017; Shu et al., 2021; Nguyen et al., 2023). For precision, recall, and F1-score, we report weighted results to account for class imbalance. Specifically, the contribution of each class is weighted by its proportion in the ground-truth data, which helps avoid the evaluation being dominated by majority classes.

AUC is also reported because it provides a threshold-independent evaluation of classification capability and is particularly suitable for imbalanced datasets. Since the three datasets corresponding to different benchmark settings in prior work, we follow the evaluation protocol commonly used for each dataset when comparing with existing methods. Specifically, on MemVul and PatchDB, we report weighted precision, recall, and F1-score, because these can be consistently obtained under our experimental setting. On the Java dataset, we report AUC for all compared methods, because the most relevant prior benchmark studies on this dataset primarily report AUC (Zhou et al., 2021a; Nguyen et al., 2023), and one of the compared baselines (MiDas) is only available from the literature in this form. Using AUC consistently on the Java benchmark therefore avoids mixing directly reproduced metrics with partially reported literature results and ensures a fairer comparison on that dataset.

We use a weighted calculation method when calculating these three indicators to deal with imbalanced data sets. It calculates these indicators by assigning different weights to samples of different categories. Specifically, when calculating the precision, recall, and F1 score, the weight of each category is proportional to its proportion in the actual samples. This means that the prediction results of the minority class, which has a smaller number

of samples, will be given higher priority, thus avoiding the majority class, which has a larger number of samples, dominating the overall performance evaluation. This is especially useful for evaluating models trained on imbalanced datasets. AUC provides a comprehensive evaluation of the model's classification capability across various thresholds, making it also particularly suitable for imbalanced datasets.

5.2.3. Implementation

We implement VPFinder according to the method described in Section 4. We utilize pre-trained BERT (bert-base uncased, 2023) and CodeBERT (codebert-base-finetuned-detect-insecure code, 2023) from the Huggingface Transformer library. The embedding dimension is set as 768 (consistent with the common encoding dimension of BERT) for all input fields, and the padding or truncation length is set to 512. The number of heads in the multi-head attention is set to 8. In the vulnerability identification, `FCN 1` consists of a fully connected network with layer sizes of 768 * 5, 768 * 3, 768, 256, 64, and 2. In the vulnerability type identification, `FCN 2` has layer sizes of 768 * 5, 768 * 3, 768, 256, and 12. All networks use ReLU as the activation function (Glorot and Bengio, 2010), and a dropout (Hinton et al., 2012) of 0.3 for each layer to prevent overfitting. Cross-entropy loss function (Krizhevsky et al., 2012; Devlin et al., 2018) is used to calculate the loss and AdamW (Loshchilov and Hutter, 2017) as the optimizer with a learning rate of $5e^{-5}$. We made the model parameters publicly available to facilitate the reproduction of our experimental results.

To ensure a fair and unbiased comparison, we strictly followed the original dataset splits (i.e., training, validation, and test sets) provided or described in the corresponding baseline studies. No re-splitting or random repartitioning of the datasets was performed in our experiments.

For baseline methods with publicly available implementations, we re-ran the models using the same evaluation protocols as reported in their original papers and replication packages. For the Java dataset, we re-implemented and evaluated VulFixMiner and VulCurator under the original train/test split of the benchmark. For MiDas, although we made efforts to follow the experimental description in the original paper, its implementation could not be reliably reproduced in our environment. Therefore, we directly report the AUC value published in the original study under the same Java benchmark setting (Nguyen et al., 2023). Reporting additional metrics only for re-implemented methods would mix directly computed results with incom-

pletely reported literature results, which would make the comparison less fair. Accordingly, in Table 6, the results of VulFixMiner, VulCurator, and VPFinder are obtained from our own experiments, while the MiDas result is taken from the literature.

We acknowledge that some baseline results are obtained from the literature; however, this practice is consistent with prior comparative studies in this domain and does not advantage our approach, as VPFinder is evaluated under the same dataset constraints. Moreover, for each dataset, VPFinder was trained from scratch using only the corresponding training set and evaluated exclusively on the designated test set. Model parameters were not shared, reused, or transferred across datasets. This ensures that each dataset represents an independent experimental setting and that no information leakage occurs between training and testing phases.

The experiments are conducted on a server equipped with NVIDIA A100-SXM GPUs and Intel Xeon Gold 5218 CPUs, running the Ubuntu operating system.

5.3. Experiment Results

5.3.1. RQ1: Effectiveness of VPFinder *in identifying vulnerability*

Table 5: (Weighted) Performance in vulnerability identification task.

Dataset	Model	Precision	Recall	F1-score
MemVul	MemVul	0.892	0.878	0.880
	VulFixMiner	0.814	0.800	0.806
	VulCurator	0.860	0.863	0.861
	SPI	0.838	0.840	0.839
	Sun et al.'s work	0.827	0.829	0.828
	VPFinder	**0.943**	**0.943**	**0.941**
PatchDB	GraphSPD	0.706	0.352	0.470
	VPFinder	**0.785**	**0.699**	**0.618**

The comparison between VPFinder and baseline methods for the vulnerability identification task is presented in Table 5 and Table 6, with the best results highlighted in bold. On the MemVul dataset, we compare VPFinder with MemVul, VulFixMiner, VulCurator, SPI, and *Sun et al.*'s work. Among these methods, MemVul demonstrates strong performance, which differs from the results reported in its original paper. One possible reason is that after filtering and constructing the dataset used in our experiments, the distribution

Table 6: (Weighted) Performance on the Java vulnerability identificaiton benchmark.

Dataset	Model	Precision	Recall	F1-score	AUC
Java	VulFixMiner	0.854	0.800	0.826	0.83
	VulCurator	0.901	0.902	0.902	0.88
	MiDas	-	-	-	0.85
	VPFinder	**0.998**	**0.998**	**0.998**	**0.95**

of CWE categories becomes more concentrated, which may strengthen the effect of the CWE anchors used in MemVul. The results also indicate that VulFixMiner achieves the lowest performance among the compared methods on this dataset. This suggests that relying solely on patch information has limitations for vulnerability identification. VulCurator, which combines information from bug reports, commit messages, and patches at the decision level, performs better than methods that only use code changes, indicating that textual information provides useful complementary signals. Although the approach of *Sun et al.*'s work uses both commit messages and code changes, its performance is slightly lower than SPI, suggesting that simple attention mechanisms may not fully capture the higher-level semantic relationships required for vulnerability identification. Overall, VPFinder achieves the best F1-score on MemVul dataset, indicating that embedding-level fusion of bug reports, commit messages, and code patches is effective for vulnerability identification.

For PatchDB and the Java dataset, we follow the benchmark settings used in the corresponding baseline studies. Specifically, we compare VPFinder with GraphSPD on PatchDB and with VulFixMiner, VulCurator, and MiDas on the Java dataset. On the PatchDB dataset, GraphSPD shows relatively high precision but limited recall, suggesting that relying only on source-code-level patch representations may not capture sufficient contextual information for vulnerability identification. In comparison, VPFinder achieves higher recall and F1-score, indicating that incorporating multiple sources of information can improve the robustness of vulnerability detection.

On the Java dataset, VulFixMiner and VulCurator were re-implemented and evaluated under the original benchmark split, achieving AUC values of 0.83 and 0.88, respectively. For MiDas, we report the AUC value (0.85) published in the original study (Nguyen et al., 2023), because its implementation could not be reliably reproduced in our experimental environment. Under the same Java benchmark setting, VPFinder achieves an AUC of 0.95, which

is higher than the compared approaches.

Overall, the results across the three datasets show that VPFinder consistently achieves the strongest or most competitive performance under the corresponding benchmark settings. These results suggest that jointly modeling bug reports, commit messages, and code patches helps capture complementary vulnerability-related signals and improves vulnerability identification performance.

Answering RQ1: VPFinder achieves the best or most competitive performance across the vulnerability identification benchmarks, demonstrating the effectiveness of fusing multi-source information.

5.3.2. RQ2: Effectiveness of VPFinder *in classifying vulnerability types*

Table 7: (Weighted) Performance in vulnerability type classification task.

Approach	Precision	Recall	F1-score
Sun et al.'s work	0.364	0.326	0.326
TreeVul-2	0.475	0.434	0.433
TreeVul-1	0.579	0.550	0.556
VulCurator	0.549	0.597	0.555
VPFinder	**0.611**	**0.620**	**0.610**

In terms of vulnerability type classification task, we compare VPFinder on MemVul dataset with *Sun et al.*'s work, TreeVul and VulCurator, since other tools do not support type classification yet.

Since Treevul is capable of producing outputs at different CWE layers, we compared its performance on both the first and second layers. (i.e., TreeVul-1 represents the TreeVul only predicts the results of the first layer and TreeVul-2 represents the TreeVul predicts the results of the first two layers.) The comparison results are presented in Table 7. *Sun et al.*'s work, which learns features from bug fixes, shows a certain level of vulnerability type classification capability. In comparison, VPFinder achieves better results in all metrics. When compared to TreeVul, TreeVul-1 and VPFinder produce a similar F1-score. However, the performance of TreeVul-2 becomes poor compared to VPFinder. VPFinder pays attention to the latent connections among bug report, commit message and patch, trying to model and capture the important contents on vulnerability. The "collaboration" among inter-

connected multi-source information can also help VPFinder better understand the boundaries between each vulnerability category. However, TreeVul tries to learn from code changes, paying attention to the difference between the original code and the modified code. TreeVul ignores connections between code changes and commit message, so it can't learn the classification boundary better. Moreover, the advantage of VPFinder over TreeVul is that it does not need to train multiple classifiers for each layer, that is, it does not need to train a classifier for the child nodes of each parent node, as TreeVul does by training classifiers for each layer. The results of VulCurator and TreeVul-1 are similar, indicating that a sample fusion of multi-source information does not significantly outperform the use of single-source information. Comparing VulCurator and VPFinder, VPFinder outperforms VulCurator across all metrics, demonstrating that embedding-level fusion outperforms decision-level fusion. In summary, VPFinder nearly surpasses all models in performance, suggesting that fusing multi-source information enhances the effectiveness of classification.

Answering RQ2: VPFinder outperforms all baselines, demonstrating the effectiveness of multi-source information fusion approach.

5.3.3. RQ3: How do architectural variations influence the performance of VPFinder*?*

To better understand the factors that govern the effectiveness of VPFinder, we investigate how different architectural design choices impact its overall performance. Specially, this research question examines the extent to which variations in model architecture, input configurations, hyperparameters , and fusion strategies affect VPFinder's detection capability. By systematically analyzing these architectural variations, we aim to identify the most influential components and provide empirical guidance for designing more robust and effective VPFinder models. To answer this research question, we analyze the impact of architectural variations from the following aspects:

- Model Architecture Design and Input Configurations
- Hyperparameter Settings
- Fusion Strategies

The ablation study was conducted on the MemVul dataset. This choice was intentional for two reasons. First, MemVul is the only dataset among

the three that provides fine-grained CWE annotations, which are required for evaluating the impact of different components on vulnerability type classification. Second, MemVul fully supports all three input modalities used by VPFinder, namely bug reports, commit messages, and code patches. In contrast, the other datasets do not provide complete support for these inputs, making them unsuitable for a comprehensive ablation analysis.

Model Architecture Design and Input Configurations. We conduct the comparison of model's each artifact in vulnerability type classification task on the MemVul dataset.

- **VPFinder-i** takes problem description as the only input.
- **VPFinder-c** takes commit message as the only input.
- **VPFinder-p** takes patch as the only input.
- **VPFinder-CLS** takes problem description, commit message and patch as inputs, utilizing their [CLS] tokens directly for vulnerability type classification.
- **VPFinder-att1** takes problem description and patch as inputs, utilizing `Multi-Head Attention 1` and their [CLS] tokens for vulnerability type classification.
- **VPFinder-att2** takes commit message and patch as inputs, utilizing `Multi-Head Attention 2` and their [CLS] toekns for vulnerability type classification.
- **VPFinder-att12** shares the same steps with VPFinder, but the model directly use the attention outputs for vulnerability type classification.

The results of the ablation study experiments on the vulnerability type classification task are presented in Table 8, with the best results highlighted in bold. The first three models each use a single input, and the best of them achieves a F1-score of 0.549, indicating that all three kinds of information are helpful for vulnerability classification. Among these three models, VPFinder-i achieves the best performance, indicating that the information of bug report is very helpful to distinguish the vulnerability categories. VPFinder-CLS which takes all information as inputs performs worse than

Table 8: (Weighted) Ablation study results on vulnerability type classification task.

Approach	Precision	Recall	F1-score
VPFinder-i	0.586	0.543	0.549
VPFinder-c	0.410	0.403	0.396
VPFinder-p	0.504	0.535	0.512
VPFinder-CLS	0.536	0.558	0.525
VPFinder-att1	0.530	0.574	0.535
VPFinder-att2	**0.647**	0.581	0.582
VPFinder-att12	0.507	0.566	0.518
VPFinder	0.611	**0.620**	**0.610**

VPFinder-i, indicating that directly generating predictions without fusing multi-source information can degrade the model's performance. Comparing VPFinder-att1, VPFinder-att2 and VPFinder-att12, we observe that the use of attention mechanism can mine the potential relationship between bug report (or commit message) and patch to a certain extent. In addition, it can be observed that simply employing fusion does not always guarantee better results. The relationship between commit message and patch is closer than that between bug report and patch. Comparing VPFinder-c, VPFinder-p, VPFinder-CLS, and VPFinder-att2, Comparing VPFinder-CLS, VPFinder-att12 and VPFinder, there are certain limitations whether using [CLS] tokens directly or using the attention mechanism, although they all show good classification abilities. While VPFinder gets the best results because VPFinder takes advantages of both [CLS] tokens and attention mechanism, capturing the latent connection among the inputs.

Overall, the study provides valuable insights into the impact of different model designs in vulnerability classification performance.

Answering RQ3: Various sources and different fusion strategies affect performance to a large extent, proving necessity of each VPFinder's component.

Hyperparameter Settings. In deep learning, the selection of the number of heads is an important hyperparameter. The purpose of this RQ is to analyze how the number of heads affects the model's expressive and generalization abilities by comparing the model performance under different numbers

of heads, and to find the most suitable configuration of the number of heads to achieve optimal performance. The experimental results when the number of heads is set to 1, 2, 4, 8, 16, and 32 in table 9. From the experimental results, it can be observed that a single-head attention mechanism may not fully capture the complex patterns in the data. When the number of heads is increased to two, the results show significant improvement, indicating that the multi-head attention mechanism is better able to capture different features in the data. However, excessively increasing the number of heads may lead to attention dispersion and an inability to effectively focus on key information, which may cause the model to encounter difficulties in extracting key features, thereby affecting the results.

Table 9: (Weighted) Results on different number of heads setting.

$number of heads$	Precision	Recall	F1-score
1	0.601	0.558	0.540
2	0.644	0.574	0.591
4	0.491	0.473	0.452
8	0.611	0.620	0.610
16	0.563	0.581	0.559
32	0.627	0.605	0.585

Answering RQ3: The number of heads impact VPFinder's effectiveness and excessively increasing the number of heads may lead to performance degradation.

Fusion Strategies. To validate whether our fusion method is superior, we compare the following two fusion approaches with VPFinder: decision-level voting and embedding vector concatenation.

1. **Decision-Level voting**

   This includes three voting strategies:

   - **Hard Voting:** The final decision is based on the majority vote.
   - **Average Voting:** Probabilities are summed and averaged before making the decision.
   - **Weighted Voting:** A learnable neural network is added to assign weights to voting results before the final decision.

In the implementation of decision-level voting, we modify VPFinder as follows:

- After encoding with BERT or CodeBERT, different sources of information (issue, commit message, and patch) are fed into their respective classifiers (here, fully connected networks).
- Each classifier outputs voting results (that is, probabilities of being a positive or negative sample).
- **Hard Voting** makes decisions based on vote counts.
- **Average Voting** aggregates probabilities by summation and averaging before deciding.
- **Weighted Voting** introduces a neural network layer that can be trained to weight the voting results before the final decision.

2. **Embedding vector concatenation**

   After encoding, the [CLS] tokens are directly concatenated and fed into a fully connected network to produce the final output.

This structured comparison ensures a rigorous evaluation of our fusion method against established alternatives.

Table 10: (Weighted) Results on different fusion methods.

Task	Method	Precision	Recall	F1-score
Vulnerability Identification	VPFinder-hard	0.797	0.793	0.795
	VPFinder-avg	0.938	0.939	0.938
	VPFinder-weighted	0.886	0.882	0.884
	VPFinder-concat	0.919	0.918	0.918
	VPFinder	**0.943**	**0.943**	**0.941**
Vulnerability Type Classification	VPFinder-hard	0.508	0.454	0.449
	VPFinder-avg	**0.625**	0.605	0.605
	VPFinder-weighted	0.604	0.605	0.586
	VPFinder-concat	0.536	0.558	0.525
	VPFinder	0.611	**0.620**	**0.610**

The results on different fusion methods are presented in Table 10, with the best results highlighted in bold. VPFinder-hard, VPFinder-avg, and VPFinder-weighted are models employing the three voting strategies—hard voting, average voting, and weighted voting, respectively. VPFinder-concat

refers to the model using embedding vector concatenation, which is the same as the VPFinder-CLS mentioned in RQ3. VPFinder-hard performs the worst, probably because hard voting cannot handle class imbalance and disregards the model's confidence scores for each class. VPFinder-weighted outperforms hard voting but is still inferior to VPFinder-avg, suggesting that simple averaging may be more robust than assigning specific weights to certain classes, indicating that the contribution of each class to the classification task is not stable. VPFinder-concat achieves moderate performance, implying that embedding vector concatenation may fail to fully capture the complex relationships between multi-source information. VPFinder (with attention mechanism) delivers the best results, demonstrating that the attention mechanism can effectively identify key features and enhance model performance.

Answering RQ3: Attention mechanism outperforms both decision-level voting and embedding vector concatenation. It effectively addresses class imbalance, captures complex multi-source relationships, and identifies key features, leading to the best model performance.

5.3.4. RQ4: Effectiveness if VPFinder *in real application scenarios on key software package repositories of openEuler (Huawei, 2024) dependencies*

OpenEuler is an open-source operating system spearheaded by Huawei and supported by the global open-source community. It supports multiple hardware architectures and is optimized for cloud-native, big data, AI, and HPC applications. OpenEuler boasts performance, stability, and security, with regular updates and security patches. Hosted by the OpenAtom Foundation, it fosters collaboration with upstream and downstream partners, promoting a vibrant ecosystem. Its frequent innovation and LTS releases cater to diverse user needs, positioning openEuler as a key player in the open-source OS landscape.

To assess the applicability of VPFinder in real-world industrial settings, we conduct an additional experiment on the openEuler ecosystem. This experiment is not designed as a cross-dataset generalization or transfer-learning evaluation. Instead, it aims to evaluate whether VPFinder can still effectively identify vulnerability-fixing commits under substantially more challenging and realistic conditions than those of public benchmark datasets.

Unlike prior datasets, the openEuler dataset lacks issue reports and detailed vulnerability descriptions. As a result, only commit messages and code patches are available as input features. Moreover, vulnerability labels in this ecosystem may be incomplete or delayed, and the dataset exhibits extreme class imbalance, with the ratio of vulnerability-fixing commits to non-vulnerability-fixing commits below 1:100 in both training and testing sets.

Huawei provided a list of software packages within openEuler that had experienced a relatively high number of vulnerabilities in the past two years. Based on this list, we crawled all commits for these packages from January 2022 to April 2024. In addition, Huawei supplied a list of CVEs associated with these software packages. Using the CVE information, we matched corresponding vulnerability-fixing commits via public vulnerability databases, including NVD, Red Hat Bugzilla, and Debian security advisories. In total, we collected 340 CVE-associated fixing commits across 85 software packages. Commits whose SHA values matched known CVE fixes were labeled as positive samples, while all remaining commits were labeled as negative.

The dataset was split chronologically to reflect realistic deployment scenarios. Commits from 2022 and earlier were used for training, while more recent commits were reserved for testing. The training set contains 195 positive and 58,449 negative samples, and the test set contains 145 positive and 82,019 negative samples. For this experiment, VPFinder was trained from scratch using only the openEuler training set, and no model parameters were shared or transferred from experiments on other datasets. Since issue reports are unavailable, we adopt VPFinder-att2, which utilizes only commit messages and code patches as input.

Table 11: Results on openEuler dataset.

Approach	Precision	Recall	F1-score
Sun et al.'s work	0.077	0.276	0.120
VPFinder-att2	0.256	0.276	0.265

Table 12: Results on openEuler dataset after manual inspection.

Approach	Precision	Recall	F1-score
Sun et al.'s work	0.088	0.293	0.135
VPFinder-att2	0.276	0.274	0.275

The experimental results are reported in Table 11. Although the absolute performance metrics are relatively low, this behavior is expected given the scarcity of labeled vulnerability samples, label incompleteness, and extreme class imbalance inherent to the openEuler dataset, which similarly affect the other compared method. Under these adverse yet realistic conditions, VPFinder-att2 achieves a substantial improvement over the baseline approach, with an increase of 0.145 in F1-score, indicating that explicitly modeling the semantic interaction between commit messages and code patches remains beneficial even in the absence of issue reports.

To further analyze the model's predictions, we conducted a manual inspection of the test results with the assistance of security experts from Huawei and the results are shown in Table 12. Manual analysis of the testing set corrected 12 samples originally labeled as negative to positive, increasing the number of true vulnerability-fixing commits from 145 to 157. Among the 156 commits predicted by VPFinder-att2 as vulnerability-fixing, 3 false positives were verified as true positives with corresponding CVEs: 2 fixing buffer overflows and 1 fixing a memory out-of-bounds access. The remaining false positives included 22 functional optimizations, 6 memory out-of-bounds fixes without CVEs, 4 memory leak fixes, and other changes such as bug fixes, testing, business process optimizations, and exception handling.

Answering RQ4: The results indicate that VPFinder consistently outperforms the baseline method in the real-world openEuler scenario, despite the absolute performance metrics being relatively low. This outcome is expected given the challenging characteristics of the openEuler dataset, including the absence of issue reports, extreme class imbalance, and potentially incomplete or noisy vulnerability labels. Under such realistic and adverse conditions, the superior performance of VPFinder demonstrates its robustness and practical effectiveness in identifying vulnerability-fixing commits in industrial OSS ecosystems.

5.3.5. RQ5: Does VPFinder*'s ability to filter out irrelevant information outperform other methods?*

To systematically evaluate whether VPFinder can more effectively filter out irrelevant information than existing methods, we conduct a controlled noise injection experiment on the MemVul testing set. The underlying hypothesis is that a model with stronger noise-filtering capability should exhibit

Table 13: Text and code block noise.

No.	Text context	Code content
1	the	`x = 0\nfor i in range(10):\n  x += i`
2	and	`def dummy_func():\n  return None`
3	of	`print('Hello World')`
4	a	`# Here's a note\npass`
5	to	`temp_list = [1, 2, 3]\nfor item in temp_list:\n  print(item)`
6	-	`try:\n  pass\nexcept:\n  pass`
7	-	`import os\nos.getcwd()`

greater robustness, i.e., less performance degradation as irrelevant information increases.

Specifically, we inject noise into the test data at five ratios (5%, 10%, 15%, 20%, and 25%). As summarized in Table 13, noise is introduced in two forms. For textual content, five high-frequency stop words ("the", "and", "of", "a", "to") are randomly inserted according to the specified noise ratio. For code content, irrelevant code blocks are randomly inserted at corresponding proportions. These perturbations simulate realistic irrelevant infomation that does not contribute to vulnerability identification.

Table 14 reports the vulnerability identification performance of different models under varying noise ratios. An Interesting observation is that increasing noise does not necessarily lead to monotonic performance degradation. In some cases, slight performance improvements are observed under low or moderate noise ratios. This phenomenon has also been reported in prior work (Bishop, 1995; Li, 2024; Yu et al., 2025) and can be contributed to a regularization-like effect, where injected noise discourages models from over-relying on spurious local patterns. In fact, training or evaluating models under noisy conditions may reduce over-reliance on spurious local patterns and improve robustness to input perturbations (Bishop, 1995). Moreover, when irrelevant tokens or code blocks are injected, models utilizing attention mechanisms are forced to rely less on spurious local patterns and more on task-relevant, global correlations between commit messages, bug reports, and code patches. As a result, the model may correct for overfitting to small idiosyncratic cues in the original data, leading to modest F1 improvements. We also confirmed that this effect is reproducible across both textual and code noise, indicating it is not an artifact of a specific noise type. While artificially injected noise differs from naturally occurring noise in real-world repositories, the attention-based design of VPFinder explicitly assigns higher importance to relevant elements and suppresses irrelevant components, suggesting

Table 14: (Weighted) Results of different noise ratios on vulnerability identification task.

Model	Noise Ratio	Precision	Recall	F1-score
VulFixMiner	0%	0.814	0.800	0.806
	5%	0.819	0.804	0.810
	10%	0.815	0.800	0.806
	15%	0.807	0.791	0.797
	20%	0.821	0.802	0.809
	25%	0.812	0.797	0.803
VulCurator	0%	0.860	0.863	0.861
	5%	0.854	0.856	0.855
	10%	0.862	0.863	0.863
	15%	0.859	0.854	0.856
	20%	0.848	0.840	0.843
	25%	0.852	0.846	0.848
Sun et al.'s work	0%	0.827	0.829	0.828
	5%	0.827	0.831	0.828
	10%	0.825	0.831	0.828
	15%	0.822	0.829	0.824
	20%	0.819	0.827	0.822
	25%	0.821	0.829	0.824
VPFinder	0%	0.943	0.943	0.941
	5%	0.943	0.943	0.941
	10%	0.943	0.943	0.941
	15%	0.947	0.947	0.945
	20%	0.943	0.943	0.941
	25%	0.948	0.949	0.947

that the model is likely to retain robustness under realistic noisy conditions.

Among the compared methods, VulFixMiner exhibits the most stable behavior, with only minor performance fluctuations across noise ratios, although its absolute performance remains relatively low. VulCurator achieves higher performance under low-noise increases, resulting in the largest performance gap between its best and worst cases, which indicates limited robustness.

Robustness to irrelevant input variations has been closely linked to a model's ability to focus on task-relevant features rather than superficial signals (Goodfellow et al., 2015). Attention-based architectures are particularly effective in this regard, as they explicitly assign different importance weights to input elements and suppress noisy or uninformative components (Vaswani et al., 2017; Lin et al., 2017). So the approach proposed by *Sun et al.*, which incorporates attention mechanisms, demonstrates moderate performance and relatively stable degradation trends. Moreover, VPFinder consistently maintains high performance across all noise ratios, showing minimal degradation even under heavy noise injection. This stability suggests that VPFinder is more effective at suppressing irrelevant information and focusing on vulnerability-related features. The results provide strong evidence that VPFinder 's attention-based design offers superior robustness to noise compared to existing methods, thereby positively answering **RQ5**.

Answering RQ5: The experimental results demonstrate that VPFinder consistently maintains high performance under increasing noise ratios, exhibiting negligible performance degradation compared to baseline methods. While other models show varying degrees of sensitivity to injected irrelevant information, VPFinder remains stable across all noise settings. These results indicate that VPFinder is more effective at filtering out irrelevant information and is significantly more robust than existing approaches.

6. Discussion

6.1. Application scenarios

VPFinder requires various input data to make decisions. If all sources of information required by VPFinder are available (that is, the optimal scenario), VPFinder can make the best decisions. However, such a situation

is not commonly encountered in practice and may also present challenges related to attack windows. In cases where certain information is unavailable, such as when only commit data is available, VPFinder can also support it. VPFinder is designed as a general framework that accommodates a variety of data sources. It can make decisions based on commit messages and patches or solely on issue reports. Relevant experiments (Section 5.3.3 and Section 5.3.4) have shown that the integration of these two types of information is beneficial for vulnerability identification and vulnerability type classification.

Therefore, VPFinder applies to the following scenarios (especially for developers of downstream software):

1) To identify whether a commit is trying to fix a vulnerability (or vulnerabilities). If the result is 'yes' VPFinder will further output the possible CWE type. Developers of downstream software dependent on the repository can promptly analyze the patch and subsequently improve their code.

2) To identify whether a bug report is describing a security issue. If the result is 'yes' VPFinder will further output the possible CWE type. As the repository maintainer, one can prioritize the bug report and promptly address the vulnerability. Meanwhile, developers of downstream software can take proactive defensive measures in advance.

3) To identify whether a resolved bug report is related to vulnerabilities, this scenario provides all the inputs required by VPFinder and yields more accurate prediction results. It can help validate the judgments made in the previous two scenarios. If VPFinder identifies the content as vulnerability-related, downstream software developers who previously overlooked the issue should reconsider their code.

6.2. Generalizability to industrial and less-controlled environments

While VPFinder achieves strong performance on curated public datasets, the openEuler case study demonstrates a noticeable performance drop under real-world conditions characterized by extreme class imbalance, incomplete vulnerability labels, and heterogeneous project histories. These factors reflect the inherent difficulty of vulnerability identification in industrial OSS ecosystems rather than a flaw in the proposed approach.

In practice, repositories often lack complete issue reports, contain sparse or noisy commit messages, and include fixes that have not yet been assigned official CVE identifiers. VPFinder is designed to be flexible with respect to input availability: when some sources are missing, the model can operate using the remaining modalities, such as commit messages and code patches. Although the absence of certain inputs leads to lower absolute performance, the model consistently maintains a relative advantage over baseline methods, indicating that modeling interactions between available sources remains beneficial.

These results suggest that VPFinder is particularly suitable as an assistive or prioritization tool in realistic pipelines, helping practitioners narrow down suspicious commits for further manual inspection, even when complete information is unavailable. In this way, the proposed approach degrades gracefully under partially observed conditions while still providing practical value.

6.3. The attack window between issue report and fix commit

The premise for VPFinder to make better predictions is that the fix commit corresponding to the issue report is available. However, the period of waiting for the fix commit is also a vulnerable time for potential attacks. *Pan et al.* (Pan et al., 2022) conducted an investigation and pointed out that the majority (98.7%) of issue reports from CVE-referred issue reports are created before the vulnerability disclosure date (with a median time of 13 days between the creation of the issue report and its NVD disclosure). This can lead to the leakage of sensitive vulnerability information (39.9% of issue reports contain attack steps). Besides, *Pan et al.* (Pan et al., 2024) conducted an empirical study and found that the window of opportunity for attackers (i.e., the risk of software vulnerability information leakage) can start at the very beginning of the remediation (i.e., issue report reporting the software vulnerability), and lasts over 30 days for over half of the software vulnerabilities. On one hand, although the average delay of upstream patches is 30 days, *Jiang et al.* (Jiang et al., 2020) point out that in the Linux kernel, downstream kernel vendors often fail to promptly adopt patches released in the mainstream version, with delays ranging from several months to several years. On the other hand, while the average delay of upstream patches is 30 days, vulnerabilities may persist in downstream codebases for about five years without being fixed after the patches are released: the audit service team of Black Duck investigated 1067 commercial codebases from 17 industries and

conducted security assessments on 936 of them. In their released "2024 Open Source Security and Risk Analysis" (Duck, 2025), they noted that 8 out of the top 10 security vulnerabilities appeared in the jQuery JavaScript library, and more than one-third of the codebases contained the two cross-site scripting (XSS) vulnerabilities, CVE-2020-11023 and CVE-2020-11022. However, patches for these vulnerabilities were released as early as April 2020 but are still widely present in commercial codebases. What's more, in a study (2018) conducted by the Ponemon Institute on behalf of ServiceNow, half of the organizations reported experiencing one or more data breaches in the past two years, and 34% stated that they were aware of vulnerabilities in their systems before being attacked. The study surveyed nearly 3,000 IT professionals worldwide regarding their patching practices (Reading, 2023). Overall, downstream practitioners show little strong intention to fix vulnerabilities during the attack window from the time of issue report publication to patch release, and they may not take effective actions even when they are aware of the risk of being attacked.

During the attack window, VPFinder provides practitioners with a range of options to understand potential risks in advance. When an issue report is initially published, tools such as VPFinder or MemVul can be used to make a preliminary assessment based on the description in the issue report and alert repository maintainers or downstream software developers about the potential presence of a vulnerability (if the assessment is positive). This provides them with time to take defensive measures. Once the fix commit becomes available, VPFinder can perform further analysis to assist downstream software developers in identifying security patches (because they need to identify vulnerability-fixing commits among a large number of commits.). Finally, after repository maintainers associate the fix commit with the issue report (if this step is taken), VPFinder can further confirm the relevance of the issue report and its resolution to security vulnerabilities.

In addition, recommendations have been proposed to timely sense patches committed to the codebase in a timely sense (Pan et al., 2024).

6.4. Reasons for misclassification of vulnerability type classification task

To investigate the reasons for misclassification, we analyzed the samples that were incorrectly predicted by VPFinder. One type of misclassification occurs at the first layer of the CWE category, where VPFinder incorrectly identifies the parent CWE categories. This is the most frequent type of error,

accounting for 90% of all misclassifications. Such misclassification can be attributed to the limited number of samples in certain CWE categories, which prevents the model from learning more discriminative features. Additionally, the presence of descriptions in bug reports that resemble other CWE categories may also lead to misjudgments by VPFinder. Another type of misclassification occurs at the second layer, where VPFinder correctly identifies the sample as belonging to the CWE-664 subclass but misclassifies the specific subclass. This type of misclassification is primarily caused by the unclear boundaries between each CWE category (due to the complexity and diversity of software vulnerabilities, there may be overlaps or ambiguous boundaries between certain categories), which is further influenced by the descriptions in bug reports.

7. Threats to Validity

7.1. Internal validity

The primary threats stem from the handling of a limited length of data. The extraction of buggy code in the dataset is straightforward, involving the entire fragments of functions and classes where the patch code is located, including deletions and additions. However, the handling capacity of BERT or CodeBERT for text or code length is limited, and excessively long text or code fragments may lead to the loss of important text or code segments. Fortunately, most of the patches do not exceed the length limit. Another concern arises from the dataset itself. A small number of CVEs do not have associated CWE categories, and we addressed this by manually assigning CWE numbers to them with the help of ChatGPT. Two of the authors double-checked the labeling results in case of any bias. There are some commits in the openEuler dataset that have not been marked as CVE patches, although we manually analyzed the results of the model predictions. However, there may be labeling errors in the large training set, and the cost of manual analysis is high, so we do not correct the labels. Therefore, this part of the dataset is also subject to internal threats.

Another potential threat arises from the merging of CWE categories. Due to the limited number of samples in some CWE classes, we merged several low-frequency CWE categories into higher-level composite categories following the hierarchical structure defined by MITRE. Although this strategy improves the statistical reliability of training and evaluation, it also reduces the granularity of the vulnerability type classification task. As a consequence,

the reported classification performance may be higher than what would be observed under a fine-grained CWE taxonomy where categories are more numerous and often sparsely annotated. Furthermore, different studies may adopt different CWE taxonomies (e.g., fine-grained CWE IDs, parent-level CWE categories, or custom merged classes), which may affect the direct comparability of results across approaches. However, our merging strategy groups semantically related CWE categories that share similar root causes and remediation patterns. Therefore, while the task becomes more coarse-grained, the resulting classification still reflects meaningful vulnerability families rather than arbitrary groupings.

7.2. External validity

The threat to external validity pertains to the generalizability of VPFinder. Our dataset is constructed based on bug reports from open-source GitHub repositories. However, the collected security vulnerabilities and their patches may be relatively scarce compared to the real world, limiting their representativeness. Moreover, we specifically focus on commits involving .c, .cc, .java, and .py files, which may not be representative enough. Additionally, the model relies on various pieces of information, with the prerequisite that commits must be associated with bug reports to obtain information from bug descriptions. This places constraints on the model inputs. However, ablation studies and RQ5 suggest that sacrificing some performance for fewer model inputs is feasible, especially for vulnerability identification.

8. Conclusion and Future Work

In this paper, we introduced VPFinder, a deep learning-based model that fuses inter-connected multi-source information from issue reports and fix commits including commit messages and patches. Relying on the multi-source information fusion layer, VPFinder is capable of mining high-level semantic information from inter-connected multi-source data and establishing potential relationships among them. The experiments demonstrate that VPFinder outperforms all state-of-the-art (SOTA) methods, validating the effectiveness of the embedding-level fusion approach. Due to the limited number of training samples and the influence of the description of issue reports, VPFinder struggles to learn the boundaries between certain CWE categories (or the distinguishing features that separate them from other categories). In the future, we plan to further study how to better encode and

utilize the information of the CWE Information to assist type classification tasks.

References

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. 3, 40:1–40:29. URL: http://doi.acm.org/10.1145/3290353, doi:10.1145/3290353.

Bishop, C.M., 1995. Training with noise is equivalent to tikhonov regularization. Neural Computation 7, 108–116. doi:10.1162/neco.1995.7.1.108.

Chen, X., Hu, X., Huang, Y., Jiang, H., Ji, W., Jiang, Y., Jiang, Y., Liu, B., Liu, H., Li, X., et al., 2025. Deep learning-based software engineering: progress, challenges, and opportunities. Science China Information Sciences 68, 1–88.

Cheng, T., Zhao, K., Sun, S., Mateen, M., Wen, J., 2022. Effort-aware cross-project just-in-time defect prediction framework for mobile apps. Frontiers of Computer Science 16, 166207.

codebert-base-finetuned-detect-insecure code, 2023. Hugging face. https://huggingface.co/mrm8488/codebert-base-finetuned-detect-insecure-code.

Devlin, J., Chang, M., Lee, K., Toutanova, K., 2018. BERT: pre-training of deep bidirectional transformers for language understanding. CoRR abs/1810.04805. URL: http://arxiv.org/abs/1810.04805, arXiv:1810.04805.

Duck, B., 2025. 2024 open source security risk analysis. URL: https://www.blackduck.com/zh-cn/resources/analyst-reports/open-source-security-risk-analysis.html. accessed: 2025-03-10.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv:2002.08155.

Gartner, 2023. Hype Cycle for Open-Source Software, 2023. https://www.gartner.com/en/documents/4552899. [Online; accessed 19-Nov-2023].

GitHub, 2024. Github rest api documentation. URL: https://docs.github.com/en/rest. accessed: 2024-11-29.

Glorot, X., Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks, in: Proceedings of the thirteenth international conference on artificial intelligence and statistics, JMLR Workshop and Conference Proceedings. pp. 249–256.

Goodfellow, I., Shlens, J., Szegedy, C., 2015. Explaining and harnessing adversarial examples, in: International Conference on Learning Representations. URL: http://arxiv.org/abs/1412.6572.

He, K., Zhang, X., Ren, S., Sun, J., 2015. Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385 .

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition, in: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770–778.

Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.R., 2012. Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580 .

Huawei, 2024. openeuler. www.openeuler.org.

Jiang, M., Jiang, J., Wu, T., Ma, Z., Luo, X., Zhou, Y., 2024. Understanding vulnerability inducing commits of the linux kernel. ACM Trans. Softw. Eng. Methodol. 33. URL: https://doi.org/10.1145/3672452, doi:10.1145/3672452.

Jiang, Z., Zhang, Y., Xu, J., Wen, Q., Wang, Z., Zhang, X., Xing, X., Yang, M., Yang, Z., 2020. Pdiff: Semantic-based patch presence testing for downstream kernels, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 1149–1163.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems 25.

Le, T.H.M., Hin, D., Croft, R., Babar, M.A., 2022. Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning, in:

Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, IEEE Press. p. 717–729. URL: https://doi.org/10.1109/ASE51524.2021.9678622, doi:10.1109/ASE51524.2021.9678622.

Li, J., Nie, Q., Fu, W., Lin, Y., Tao, G., Liu, Y., Wang, C., 2024a. Lors: Low-rank residual structure for parameter-efficient network stacking, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 15866–15876.

Li, X., 2024. Positive-incentive noise. IEEE Transactions on Neural Networks and Learning Systems 35, 8708–8714. doi:10.1109/TNNLS.2022.3224577.

Li, Z., Pan, M., Pei, Y., Zhang, T., Wang, L., Li, X., 2024b. Empirically revisiting and enhancing automatic classification of bug and non-bug issues. Frontiers of Computer Science 18, 185207.

Lin, Z., Feng, M., dos Santos, C.N., Yu, M., Xiang, B., Zhou, B., Bengio, Y., 2017. A structured self-attentive sentence embedding, in: International Conference on Learning Representations (ICLR). URL: https://openreview.net/forum?id=HkpbnH9lx, arXiv:1703.03130.

Loshchilov, I., Hutter, F., 2017. Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101 .

MITRE, 2025. Common weakness enumeration (cwe) – a community-developed list of software and hardware weaknesses. https://cwe.mitre.org/index.html. Accessed: 2026-01-21.

Nguyen, T.G., Le-Cong, T., Kang, H.J., Le, X.B.D., Lo, D., 2022. Vulcurator: a vulnerability-fixing commit detector, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1726–1730.

Nguyen, T.G., Le-Cong, T., Kang, H.J., Widyasari, R., Yang, C., Zhao, Z., Xu, B., Zhou, J., Xia, X., Hassan, A.E., et al., 2023. Multi-granularity detector for vulnerability fixes. IEEE Transactions on Software Engineering 49, 4035–4057.

Nguyen-Truong, G., Kang, H.J., Lo, D., Sharma, A., Santosa, A.E., Sharma, A., Ang, M.Y., 2022. Hermes: Using commit-issue linking to detect

vulnerability-fixing commits, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE. pp. 51–62.

Pan, S., Bao, L., Xia, X., Lo, D., Li, S., 2023. Fine-grained commit-level vulnerability type prediction by cwe tree structure, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 957–969. doi:10.1109/ICSE48619.2023.00088.

Pan, S., Bao, L., Zhou, J., Hu, X., Xia, X., Li, S., 2024. Unveil the mystery of critical software vulnerabilities, in: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, pp. 138–149.

Pan, S., Zhou, J., Cogo, F.R., Xia, X., Bao, L., Hu, X., Li, S., Hassan, A.E., 2022. Automated unearthing of dangerous issue reports, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 834–846.

Peters, F., Tun, T.T., Yu, Y., Nuseibeh, B., 2017. Text filtering and ranking for security bug report prediction. IEEE Transactions on Software Engineering 45, 615–631.

Reading, D., 2023. Unpatched vulnerabilities: The source of most data breaches. URL: https://www.darkreading.com/vulnerabilities-threats/unpatched-vulnerabilities-the-source-of-most-data-breaches. accessed: 2025-03-10.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al., 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 .

Sabetta, A., Bezzi, M., 2018. A practical approach to the automatic classification of security-relevant commits, in: 2018 IEEE International conference on software maintenance and evolution (ICSME), IEEE. pp. 579–582.

Shu, R., Xia, T., Chen, J., Williams, L., Menzies, T., 2021. How to better distinguish security bug reports (using dual hyperparameter optimization). Empirical Software Engineering 26, 1–37.

Sun, J., Xing, Z., Lu, Q., Xu, X., Zhu, L., Hoang, T., Zhao, D., 2023. Silent vulnerable dependency alert prediction with vulnerability key aspect explanation. 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) , 970–982.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al., 2023. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 .

Tsai, Y.H.H., Bai, S., Liang, P.P., Kolter, J.Z., Morency, L.P., Salakhutdinov, R., 2019. Multimodal transformer for unaligned multimodal language sequences, in: Proceedings of the conference. Association for computational linguistics. Meeting, p. 6558.

bert-base uncased, 2023. Hugging face. https://huggingface.co/bert-base-uncased.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I., 2017. Attention is all you need, in: Proceedings of the 31st International Conference on Neural Information Processing Systems, Curran Associates Inc., Red Hook, NY, USA. p. 6000–6010.

Wang, F., Jiang, M., Qian, C., Yang, S., Li, C., Zhang, H., Wang, X., Tang, X., 2017. Residual attention network for image classification, in: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 3156–3164.

Wang, H., Tuerhong, G., Wushouer, M., Guo, X., 2025. Multi-modal sentiment analysis based on multi-level modal information interaction. Procedia Computer Science 266, 41–51.

Wang, S., Wang, X., Sun, K., Jajodia, S., Wang, H., Li, Q., 2023. Graphspd: Graph-based security patch detection with enriched code semantics, in: 2023 IEEE Symposium on Security and Privacy (SP), IEEE Computer Society, Los Alamitos, CA, USA. pp. 2409–2426. URL: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00035, doi:10.1109/SP46215.2023.00035.

Wang, X., Sun, K., Batcheller, A., Jajodia, S., 2019. Detecting "0-day" vulnerability: An empirical study of secret security patch in oss, in: 2019

49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 485–492. doi:10.1109/DSN.2019.00056.

Wang, X., Wang, S., Feng, P., Sun, K., Jajodia, S., 2021. Patchdb: A large-scale security patch dataset, in: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 149–160. doi:10.1109/DSN48987.2021.00030.

Yu, X., Huang, Z., Chen, M., Zhang, L., Liu, T., Zhu, D., 2025. Noisynn: Exploring the impact of information entropy change in learning systems. URL: https://arxiv.org/abs/2309.10625, arXiv:2309.10625.

Zadeh, A., Chen, M., Poria, S., Cambria, E., Morency, L.P., 2017. Tensor fusion network for multimodal sentiment analysis, in: Palmer, M., Hwa, R., Riedel, S. (Eds.), Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Copenhagen, Denmark. pp. 1103–1114. URL: https://aclanthology.org/D17-1115/, doi:10.18653/v1/D17-1115.

Zhang, T., Han, D., Vinayakarao, V., Irsan, I.C., Xu, B., Thung, F., Lo, D., Jiang, L., 2023. Duplicate bug report detection: How far are we? ACM Trans. Softw. Eng. Methodol. 32. URL: https://doi.org/10.1145/3576042, doi:10.1145/3576042.

Zhang, Z., Zhang, H., Qian, Z., Lau, B., 2021. An investigation of the android kernel patch ecosystem, in: 30th USENIX Security Symposium (USENIX Security 21), pp. 3649–3666.

Zhou, J., Pacheco, M., Chen, J., Hu, X., Xia, X., Lo, D., Hassan, A.E., 2023. Colefunda: Explainable silent vulnerability fix identification, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE. pp. 2565–2577.

Zhou, J., Pacheco, M., Wan, Z., Xia, X., Lo, D., Wang, Y., Hassan, A.E., 2021a. Finding a needle in a haystack: Automated mining of silent vulnerability fixes, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE. pp. 705–716.

Zhou, Y., Sharma, A., 2017. Automated identification of security issues from commit messages and bug reports, in: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp. 914–919.

Zhou, Y., Siow, J.K., Wang, C., Liu, S., Liu, Y., 2021b. Spi: Automated identification of security patches via commits. ACM Trans. Softw. Eng. Methodol. 31. URL: https://doi.org/10.1145/3468854, doi:10.1145/3468854.

Zuo, F., Rhee, J., 2024. Vulnerability discovery based on source code patch commit mining: a systematic literature review. International Journal of Information Security 23, 1–14. doi:10.1007/s10207-023-00795-8.