Attention Is All You Need for LLM-based Code Vulnerability Localization

Yue Li*, Xiao Li*, Hao Wu*, Yue Zhang[†], Xiuzhen Cheng[‡], Sheng Zhong* and Fengyuan Xu*

*National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

[†]Department of Computer Science, Drexel University, Philadelphia, USA

[‡]School of Computer Science and Technology, Shandong University, Qingdao, China

Abstract—The rapid expansion of software systems and the growing number of reported vulnerabilities have emphasized the importance of accurately identifying vulnerable code segments. Traditional methods for vulnerability localization, such as manual code audits or rule-based tools, are often time-consuming and limited in scope, typically focusing on specific programming languages or types of vulnerabilities. In recent years, the introduction of large language models (LLMs) such as GPT and LLaMA has opened new possibilities for automating vulnerability detection. However, while LLMs show promise in this area, they face challenges, particularly in maintaining accuracy over longer code contexts. This paper introduces LOVA, a novel framework leveraging the self-attention mechanisms inherent in LLMs to enhance vulnerability localization. Our key insight is that self-attention mechanisms assign varying importance to different parts of the input, making it possible to track how much attention the model focuses on specific lines of code. In the context of vulnerability localization, the hypothesis is that vulnerable lines of code will naturally attract higher attention weights because they have a greater influence on the model's output. By systematically tracking changes in attention weights and focusing on specific lines of code, LOVA improves the precision of identifying vulnerable lines across various programming languages. Through rigorous experimentation and evaluation, we demonstrate that LOVA significantly outperforms existing LLM-based approaches, achieving up to a 5.3× improvement in F1-scores. LOVA also demonstrated strong scalability, with up to a 14.6× improvement in smart contract vulnerability localization across languages like C, Python, Java, and Solidity. Its robustness was proven through consistent performance across different LLM architectures.

1. Introduction

Software vulnerabilities are weaknesses or flaws in a software system that can be exploited by attackers to compromise the security, functionality, or data integrity of the system. As of September 2024, over 240,000 vulnerabilities [5] have been publicly reported in the Common Vulnerabilities and Exposures (CVE) database, and these vulnerabilities can have serious consequences for the security and stability of a software system. Accurately localizing vulnerabilities is increasingly vital and has become a key

area of focus [39], [40], [41] in recent years. It allows developers to isolate the specific section or module of code where the flaw exists, enabling targeted and efficient remediation. This precision minimizes the risk of unnecessary changes that can introduce new issues, significantly reducing the time and resources required to fix vulnerabilities. Without precise localization, the process of addressing vulnerabilities can become inefficient, leading to potential system-wide disruptions and further security risks.

Large language models (LLMs), such as GPT [16] and LLaMA [18], are sophisticated AI systems designed to process vast amounts of data to understand and generate text or code with human-like accuracy. These models have become prominent due to their advanced capabilities in tasks such as natural language understanding, translation, summarization, and, more recently, in the domain of software development and security. One emerging use case is vulnerability localization, where LLMs help identify and isolate security flaws within large and complex codebases. Traditionally, vulnerability localization has been a timeconsuming and labor-intensive process, often requiring manual code audits or the use of rule-based tools [38], [37]. These methods not only demand significant expertise but are also prone to human error, potentially leading to overlooked vulnerabilities. Additionally, rule-based tools are often limited in scope, typically targeting specific programming languages or certain types of vulnerabilities. LLMs, however, offer a new level of automation and intelligence in this area. LLMs can analyze extensive codebases and detect patterns that are commonly associated with vulnerabilities. These models are trained on massive datasets that include programming languages, software repositories, and vulnerability databases. By pasting the code into an LLM and explicitly requesting it to localize the vulnerability, the LLM can efficiently pinpoint the location of the vulnerable code.

While LLM has demonstrated potential for vulnerability localization, there are several notable limitations. First, LLMs struggle to maintain reasoning accuracy over longer code contexts [11], [12]. For instance, in our exploratory experiments, we observed a significant drop in the output accuracy as the lines of code (LoC) increased: when the LoC exceeded 300, the accuracy fell below 5%. Although more advanced reasoning techniques such as Self-Consistency [3]—which analyzes the code multiple times using different reasoning paths to ensure consistency—, and rStar [45]-

—which uses a self-play generation-discrimination process with Monte Carlo Tree Search (MCTS), where two similar models mutually verify reasoning paths—can improve accuracy, these methods still fail to effectively address the issue of long context degradation. Moreover, they rely on stochastic decoding strategies, which introduce randomness into the results, making it difficult to accurately identify the exact vulnerable lines. Even methods like majority voting resulted in only marginal improvements.

Our work aims to enhance the accuracy of vulnerability localization by leveraging self-attention mechanisms (selfattention allows the model to assign varying importance to different input elements, which is the core of LLMs). This is because self-attention is intrinsically linked to the model's output and is frequently used to interpret its behavior. We hypothesize that if a LLM has the ability to classify vulnerabilities (e.g., types), it should also inherently possess the capability to pinpoint their exact locations. This is based on the premise that code containing vulnerabilities likely exerts the most substantial impact on the model's attention mechanisms and its subsequent outputs. However, contemporary large models face challenges in accurately localizing vulnerabilities due to their processing of extensive contextual information. This often results in key lines or code segments being abstracted into more generalized, higher-level information, causing a loss of detail and clarity. Therefore, the critical task is to refine the model's focus to concentrate specifically on the lines of code that are vulnerable, enhancing the precision of vulnerability detection and localization.

To address this problem, we propose a strategy whereby, during the construction of prompts, the LLM is deliberately directed to refocus its attention on areas of the code identified as potentially vulnerable. However, given the inherent difficulty in pre-identifying which specific sections of code contain vulnerabilities, it is not feasible to target the LLM's attention exclusively to these areas. Instead, our approach involves directing the LLM's focus across all code lines, both vulnerable and non-vulnerable. This is predicated on the understanding that non-vulnerable code segments have a minimal impact on the model's overall output. allowing for comprehensive scrutiny without compromising the effectiveness of the results. This scenario is analogous to a student who is familiar with all the answers on an exam but commits errors due to the extensive length of the test. By mandating a review of each response, the student has the opportunity to identify and rectify any mistakes, thereby ensuring that the correct answers remain unaffected.

Building on this idea and our exploratory experiments (please refer to §3.2 for validation through a series of experiments), we developed LOVA (*LO*-cating *V*ulnerabilities via Attention), which operates in three key stages: code line highlighting, attention calculation, and vulnerability localization. In the code line highlighting stage, two versions of the input are generated: a baseline version and a set of highlighted versions. In the attention calculation stage, attention maps are produced for both versions, and the differences between them are analyzed to assess the impact of highlighting specific lines. Finally, in the vulnerability local-

ization stage, the attention patterns are examined to identify potential vulnerabilities within individual lines of code.

During the development of LOVA, we overcame several key challenges. *First*, simply highlighting code by adding comments can be ineffective in large code contexts, leading to poor attention enhancement or even hallucinations. To resolve this, we designed a line index-based prompt that avoids adding extraneous content, thus preventing confusion for the model. *Second*, analyzing large attention tensors is impractical due to the massive size of the attention outputs, which can contain millions of elements. To address this, we implemented a novel dimensionality reduction technique to aggregate attention across heads and layers, focusing on critical patterns relevant to vulnerability detection. *Finally*, we employed a language-aware deep learning model to classify the reduced attention matrix, enabling more precise vulnerability identification across different programming languages.

The evaluation of LOVA reveals its superiority in localizing vulnerabilities compared to traditional LLMbased approaches across different programming languages and benchmarks. In terms of effectiveness (RQ1), LOVA achieved a significant improvement in F1-scores on both the Big-Vul and CVEFixes-C datasets, outperforming vanilla output and other sophisticated LLM techniques such as CoT, MoA and rStar by up to $5.3\times$. This high performance stems from the model's ability to balance precision and recall, minimizing false positives while maintaining high accuracy in identifying vulnerabilities. Particularly in longer contexts, LOVA demonstrates resilience where other techniques falter, maintaining high Top-N accuracy across various LOC ranges. In terms of scalability (RQ2), LOVA showcases its generalization capabilities by performing effectively across multiple languages, including C, Python, Java, and Solidity. It achieves up to a 14.6× improvement in contract-level vulnerability localization for smart contracts, underscoring its adaptability to diverse coding environments. The robustness of the system (RQ3) is further demonstrated in its crosscompatibility with various LLM architectures, consistently outperforming baseline methods regardless of the underlying model. Finally, the ablation study (RQ4) highlights the importance of each design component, confirming that the integration of contextual information and attention calculation mechanisms is essential to the method's overall effectiveness in vulnerability localization.

In summary, we make the following contributions.

- We are the first to discover that the self-attention mechanism can be effectively utilized for vulnerability localization. We demonstrate that by tracking changes in attention weights, it is possible to identify specific lines of code that are likely to contain vulnerabilities.
- We design and implement LOVA, a novel framework for vulnerability localization. Key techniques of LOVA include a line index-based prompt design to prevent confusion in large code contexts, dimensionality reduction to simplify massive attention outputs, and the use of attention map differences to assess the impact of focusing on specific lines. Additionally, a languageaware deep learning model is employed to generalize

- the approach across multiple programming languages, making it scalable and effective for diverse codebases.
- We demonstrate that LOVA significantly outperforms traditional LLM-based approaches. It shows improvements in precision, recall, and scalability across multiple programming languages, such as C, Python, Java, and Solidity. Moreover, it adapts well to different code lengths and architectures, ensuring robustness.

2. Background

2.1. LLM Architecture

A large language model (LLM) is an advanced type of artificial intelligence designed to understand and generate human-like text. These models are typically built using deep learning techniques, specifically neural networks with a large number of parameters, trained on vast amounts of text data. LLM is typically based on the transformer architecture [13].

A transformer is a type of deep learning model architecture. The key innovation of the transformer architecture is its use of self-attention mechanisms, which allow the model to weigh the importance of different words in a sentence, regardless of their position (which will be introduced next). The transformer architecture consists of an encoder and a decoder: the encoder processes the input sequence into context-rich representations, which the decoder then uses, along with previously generated tokens, to produce the final output sequence. Since both the transformer and its sub-components (the encoder and decoder) are capable of handling sequence-to-sequence generation tasks, and each component has different functions, the development of current LLMs has branched into three main technical routes: encoder-decoder (such as T5 [17]), encoder-only (such as BERT [15]), and decoder-only (such as GPT [16] and LLaMA [18]). Due to the advantages of higher training efficiency and better inference performance, state-of-the-art models predominantly use the decoder-only architecture. In this paper, we particularly focus on decoder-only LLMs.

2.2. Tokens and Next Token Prediction

Tokens: Tokens serve as the fundamental building blocks that LLMs use to represent and interpret language. Tokenization is a critical step in preparing text for processing by the model. Before handling raw text input, the LLM employs a tokenizer to break the text or code into smaller units known as tokens. These tokens can represent anything from a single character to a whole word, depending on the complexity of the language and the tokenization method. For example, when a LLM processes code, it breaks down the source code into logical tokens, such as keywords, identifiers, operators, and literals. Each component of the code is treated as a token, which the model processes individually. Similarly, during the output process, the LLM generates tokens instead of producing full text all at once. The tokenizer then converts

these tokens back into readable text or code. The use of tokenization enables LLMs to handle different languages and context, regardless of their specific syntax, by converting text or code into tokens that can be systematically processed.

Next Token Prediction: Next token prediction is a core task in training and operating LLM. It involves predicting the next token in a sequence of text based on the preceding tokens. In this process, the model takes an input sequence and generates the most likely next token, continuing this until the desired output length is reached. The training objective of next token prediction can be formalized by the following loss function:

$$L(\theta) = -\sum_{t=1}^{T} \log P(x_t | x_{< t}; \theta)$$

Here, the goal is to minimize the negative log-likelihood to accurately predict the next token based on the given input. To generate an entire sentence, the model starts with a given initial token or prompt. It predicts the next token in the sequence using the learned probabilities, appends this token to the sequence, and then uses the extended sequence as input to predict the subsequent token. This process is repeated iteratively until a complete sentence is formed.

2.3. Self-Attention Mechanism

The core of decoder layers is the self-attention mechanism, which allows a model to calculate the importance of different tokens in a sequence when encoding a specific token. When generating the next token, the LLM first calculates the relevance between the current token and all previous tokens. This relevance, known as the attention weight, is obtained through the self-attention mechanism. The model then uses these attention weights to perform a weighted sum of the values of all previous tokens, resulting in the value for the new token.

In decoder-only LLMs, to maintain the autoregressive property during the generation process—where each token is generated based solely on the previously generated tokens without access to subsequent tokens—masks are applied during the calculation of attention weights. This ensures that the attention weights for the current token to any subsequent tokens are set to zero. Furthermore, to enable parallel computation of self-attention, the multi-head attention mechanism divides each embedding into multiple parts, each processed by separate attention heads. The results from these heads are then concatenated and linearly transformed to produce the final output. Consequently, this approach allows the model to simultaneously attend to different aspects of the input sequence, thereby enhancing its ability to capture diverse features.

Attention captures the relevance between tokens, serving as an inherent explainability feature of the transformer architecture. There has been significant research on leveraging attention for the interpretability of language models. Previous research has shown that the self-attention mechanism

in LLMs can capture both syntactic and semantic information in code. Specifically, this is achieved by analyzing the attention distribution to identify syntactic or semantic relationships between tokens.

3. Motivation, Key Idea, and Threat Model

In this section, we first discuss the motivation driving our work (§3.1), then introduce our key idea, supported by a series of experiments (§3.2). Lastly, we present our threat model (§3.3).

3.1. Motivation

In this section, we discuss the motivations of our work. As shown in Figure 1, we present a vulnerable program where, in line 3, the function calculates the value of len based on the input s and subsequently allocates memory of size len * sizeof(cairo_glyph_t) in line 8. However, the gmalloc function used for memory allocation is unsafe, as it lacks checks for integer overflow. If an overflow occurs when calculating len * sizeof(cairo_glyph_t), it can lead to a buffer overflow, which could potentially be exploited for arbitrary code execution. The goal of vulnerability localization is to take code containing vulnerabilities as input and output the line in the code where the vulnerability resides, which in this example would be line 8.

Figure 1: Vulnerable code example

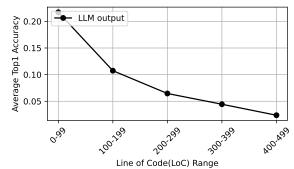
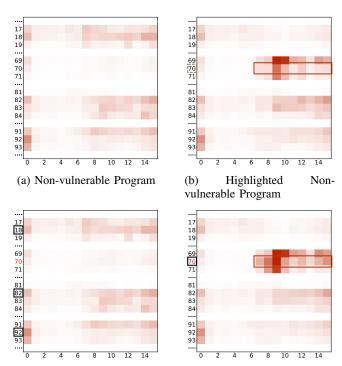


Figure 2: The relationship between Lines of Code (LoC) and Average Top-1 Accuracy



(c) Vulnerable Program (injected on Line 70)

(d) Highlighted Vulnerable Program

Figure 3: Attention maps for four programs, where (a) represents a vulnerability-free program, (c) shows the program after injecting a vulnerability at line 70 of (a). (b) and (d) represent the programs after highlighting line 70 for (a) and (c), respectively. In each figure, each row corresponds to a line number in the code, while the columns indicate the attention from different decoder layers. Darker colors indicate stronger attention, and the black boxes on the line numbers mark the vulnerable lines identified by the LLM.

While it is true that directly querying LLMs to localize code vulnerabilities, or employing more advanced prompt engineering techniques such as Chain of Thought [1], [2] (i.e., we can apply a step-by-step thought process to systematically break down the potential issues in code) can potentially produce the desired outcome, these methods are constrained by several significant limitations.

First, recent studies [11], [12] have demonstrated that the reasoning capabilities of LLMs deteriorate substantially when working with longer contexts. Since vulnerability localization is inherently a reasoning task that often involves analyzing extended code segments, this degradation poses a significant challenge. For example, for the code shown in Figure 1, when we input the complete file containing the vulnerable code into the LLM for vulnerability localization, only two out of ten attempts successfully identified the vulnerability. Furthermore, as shown in Figure 2, as the lines of code (LoC) increase, the output accuracy of the LLM declines significantly. When the LoC exceeds 300, the accuracy drops to below 5%. While existing advanced reasoning methods (e.g., Self-Consistency [3] and rStar [45])

can improve reasoning to some extent, they are not fundamentally designed to address long context degradation. Consequently, there is still a significant decline in accuracy when dealing with long contexts.

Second, the advanced reasoning methods mentioned earlier typically employ stochastic decoding strategies to explore different reasoning paths, thereby enhancing accuracy. However, this approach introduces randomness into the output, leading to variability that can result in unstable localization outcomes. Consequently, it becomes challenging to pinpoint the exact line containing the vulnerability. Even when methods like majority voting or scoring are used to aggregate multiple outputs, the overall improvement in accuracy is often marginal [4], [10].

3.2. Exploratory Experiments and Key Idea

Exploratory Experiments (EE) As discussed in §2, the core of LLMs' internal computation lies in the self-attention mechanism. By calculating attention weights between each pair of tokens, hidden states are derived, which are then used to generate the next token. Consequently, the output of LLMs is intrinsically tied to the attention weights, making self-attention one of the most widely used methods for interpreting LLM behavior. Therefore, we infer that understanding the relationship between attention weights and the LLM's output could potentially help identify specific vulnerability locations (e.g., lines of code containing vulnerabilities may receive higher attention weights).

Specifically, we have formulated four hypotheses to enhance vulnerability detection in code. First, we hypothesize that accurate localization of a vulnerable line of code necessitates the model's focused attention on that specific line. Given the challenges posed by extended contexts, vulnerable lines may be overlooked. To address this, our second hypothesis proposes manually intensifying the model's focus on identified vulnerable lines to improve detection accuracy. Thirdly, we posit that a genuinely vulnerable line will inherently exert a significant influence on the model's output. However, a major challenge is the inability to pre-identify which lines warrant this increased attention due to unknown vulnerability statuses. Intuitively, we suggest directing the model to scrutinize every line of code, irrespective of its vulnerability status. This approach hinges on our fourth hypothesis: intensifying the model's focus across all code lines is unlikely to lead to incorrect evaluations, and focusing on non-vulnerable code will not adversely affect the attention dynamics.

To validate our hypotheses, we conducted four exploratory experiments. To be more specific, we initially prepared a code snippet free of vulnerabilities, then intentionally introduced a vulnerability at a specific line. We tasked the LLM with identifying the vulnerable line of code. Notably, most large Transformer-based models (e.g., BERT, GPT) generate attention weights during training or inference, which indicate the model's focus on different input positions across layers and attention heads. These attention weights can be extracted through APIs, enabling

us to observe how attention shifts across layers and heads when processing different inputs. The experimental setup and results are discussed as follows:

EE-I: We first queried the LLM without highlighting any specific lines. The results are shown in Figure 3 (a) and (c), where Figure 3 (a) illustrates the attention distribution for the code without vulnerability, and Figure 3 (c) shows the attention for code with an injected vulnerability at line 70. Each row in the figures represents different lines of code, while the columns correspond to various decoder layers. We observed that the LLM outputted lines 18, 82, and 92 as containing vulnerabilities, which is a mistaken judgment. In the attention distribution displayed in Figure 3 (c), these lines received more attention than others. In contrast, line 70, where the actual vulnerability exists, did not attract significant attention. This tendency for key content to be overlooked is particularly evident in longer scenarios.

Observation (I). The lines identified as vulnerable by the model are consistently highlighted in the attention distribution. However, insufficient attention may prevent the model from effectively detecting the vulnerability. As a result, the output may be prone to a high false positives.

EE-II: We then explored how to direct the model's attention toward the lines containing vulnerabilities. One straightforward approach is to modify the prompt to guide the model's focus to a specific line. By instructing the model to "pay attention to line 70", we successfully shifted its focus. The attention results, shown in Figure 3 (b) and (d), indicate a significant increase in attention on line 70.

Observation (II). By adjusting the prompt, we can direct the model to focus on a specific line, effectively altering its attention.

EE-III: We analyze the LLM's output (not the attention distribution) after applying highlighting. For the code without a vulnerability, even when highlighting line 70, the LLM did not mistakenly flag it as a vulnerability, suggesting that the highlighting did not cause hallucination issues. Conversely, for the code with a vulnerability, highlighting line 70 enabled the model to successfully identify the code as vulnerable.

Observation (III). By increasing attention on a vulnerable line, we can make it easier for the model to identify it as vulnerable.

EE-IV: We then analyze the LLM's behavior when highlighting a non-vulnerable line, as shown in Figure 3 (b) and (d). The attention distribution for the highlighted line is shown within the red box, we observe that highlighting a vulnerable line leads to a significant increase in attention on that line. In contrast, when a non-vulnerable line is high-

lighted, the corresponding attention shows only a limited increase.

Observation (IV). When a vulnerable line is highlighted, certain attention patterns show significant increases, which do not appear for non-vulnerable lines.

Key Idea: As outlined in Observations (I)-(IV), LLMs tasked with vulnerability localization in extensive contexts frequently miss the actual vulnerable lines (O-I). By strategically modifying the model's prompts, we can enhance the model's focus on these vulnerable lines (O-II), thereby refining its capability to accurately localize vulnerabilities (O-III). Furthermore, directing the model's attention towards non-vulnerable lines does not adversely affect its overall attention allocation (O-IV). Therefore, our idea is that we can systematically iterate over each line of code, individually highlighting them to ensure comprehensive coverage of all potential vulnerabilities. This method is effective because highlighting a vulnerable line results in a marked increase in the model's attention, whereas non-vulnerable lines exhibit minimal changes. By monitoring these attention shifts, we can precisely identify and confirm the presence of vulnerabilities, thus significantly improving the accuracy of vulnerability localization.

3.3. Threat Model

Scope and Assumptions: We now discuss the scope and assumptions of our work. The primary goal is to localize the vulnerabilities within a program, guided by the following requirements. First, there are indeed numerous vulnerability localization tools in the software security domain, but our tool specifically focuses on those powered by LLMs, recognizing the immense potential they offer (e.g.,). Second, this paper focuses on the task of vulnerability localization within a single file. We do not address vulnerabilities that span across multiple files, such as those involving inter-file code dependencies or cross-file data flows. For example, vulnerabilities related to privacy leakage often occur when sensitive data is collected in one file, processed in another, and then improperly exposed or transmitted in yet another file. This is out of our focus.

Problem Formalization: We assume the vulnerable code is represented as $C=[c_1,c_2,\ldots,c_n]$, where c_i denotes the i-th line of code. In the context of vulnerability localization, we only consider inputs that contain vulnerabilities, meaning that the vulnerable code must include vulnerable lines $V=[v_1,v_2,\ldots,v_k]$, where $k\leq n$ and $v_i\in C$. The objective of vulnerability localization is to determine whether each line c_i is vulnerable, ultimately producing an output of the vulnerable lines $O=[o_1,o_2,\ldots,o_m]$, where $m\leq n$ and $o_i\in C$.

4. Design of LOVA

In this section, we now discuss the design details of LOVA. As shown in Figure 4, LOVA consists of three primary stages, i.e., code line highlight, attention calculation, and vulnerability line localization. Specifically, in each of these steps, we first outline the challenges (*C-1, C-2, C-3*) encountered, followed by the corresponding solutions (*S-1, S-2, S-3*).

- Code Line Highlight (§4.1): In this stage, LOVA generates two versions of the input based on the vulnerable program for LLM processing. The first is an unaltered copy with no line highlighted, while the second systematically highlights each line.
- Attention Calculation (§4.2): In this stage, LOVA uses the LLM to generate attention maps for both the base and highlighted versions of the prompt. By comparing these maps, it assesses the impact of highlighting specific lines on the model's attention distribution.
- Vulnerability Line Localization (§4.3): In this stage, LOVA analyzes the distribution of the model's attention to assess whether a specific line contains a vulnerability. By examining how strongly the model focuses on different parts of the code, LOVA can effectively determine which lines are more likely to exhibit security issues.

4.1. Code Line Highlight

In this step, LOVA generates two versions of the input based on the vulnerable program, both of which will be processed by the LLM in subsequent steps. The first version is a direct copy of the vulnerable program with the template prompt inserted, but none of the lines are highlighted (Step ①). The second version highlights every line of the vulnerable program (Step ②). To create the second version, LOVA systematically examines each line of code, sequentially applying highlights. This highlighting is intended to direct the LLM's focus to specific lines. By highlighting a code line, the attention distribution across the prompt is altered (Step ③).

(C-I) - Inappropriate prompt may mislead LLM: At first glance, highlighting a specific line may seem straightforward, with several methods available to achieve this. For instance, comments like //Pay attention to this or markers such as /*FOCUS_BEGIN*/ and /*FOCUS_END*/ can be inserted into the code to draw attention to a specific line [22], [21]. Another approach is to mask unimportant sections of the input, allowing the model to focus exclusively on the critical parts [20].

However, directly highlighting a line without considering domain-specific factors can introduce complications and potentially degrade the performance of LOVA. For example, inserting comments or markers in lengthy contexts may cause these markers to be overlooked due to the sheer amount of surrounding text, which could lead to poor attention enhancement. Imagine highlighting a line within a 200-line code file—comments such as //Pay attention

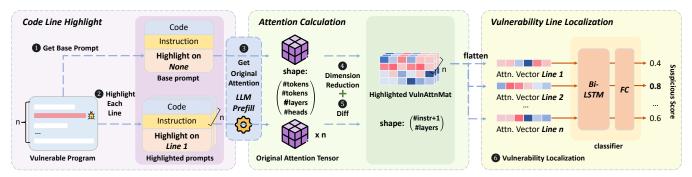


Figure 4: Overview of LOVA. For vulnerable code, the process begins by highlighting each line individually to generate a base prompt and a set of highlighted prompts. These prompts are then processed through the LLM's prefill phase to obtain their respective attention maps. For each highlighted attention map, the difference with the base attention map is computed, and the result is flattened to form an attention vector corresponding to each highlighted line. These attention vectors are then classified by a Bi-LSTM classifier to determine whether each line contains a vulnerability, ultimately yielding a suspicious score for each line.

to this might go unnoticed (as discussed in §3.1), particularly in larger contexts. Also, while specifying areas of focus in the instructions can stabilize attention increases, it may lead to hallucinations in the language model. This can cause the model to mistakenly classify a correct line as vulnerable, especially when the model strongly focuses on that line, thereby failing to reduce the false positive rate effectively. Similarly, masking irrelevant sections can compromise the integrity of the code. For instance, masking entire functions deemed unimportant could prevent a valid comparison of attention differences across highlighted lines, hindering the model's understanding of the overall logic.

(S-I) - Highlight lines using their index. Our approach is straightforward yet effective. Instead of adding extra content that might confuse the LLM, we directly highlight specific lines by specifying their line index. This method accurately enhances attention on the highlighted lines and improves the LLM's success rate in locating them. Additionally, the differences between lines are significant and easily analyzable, providing clear, quantifiable reference points for future analysis.

Next, the key question is how to construct an effective prompt based on (S-I). As depicted in Figure 1, we have implemented the following steps: First, we prepend line indices to the code and instruct the model to "pay attention to" a specific index, thereby enhancing focus on that particular line. These indices are based on absolute positions, rather than relative ones, as LLMs have difficulty with relative positioning [19]. Second, our objective is to direct the model's attention to the highlighted line and prompt it to determine whether that line contains a vulnerability, without introducing any prior assumptions. To this end, we instruct the LLM with the primary objective of identifying potential vulnerabilities in the entire code, while focusing specifically on the correctness of the highlighted line, rather than only judging the correctness of that line. Finally, to

facilitate future analysis, we minimize the inclusion of noncode lines in the prompt. Since highlighting primarily impacts the attention directed toward the highlighted line and the instructions, reducing non-code lines helps streamline subsequent analyses, making it easier to track the model's attention shifts and performance.

```
Code:

8: glyphs = (cairo_glyph_t *) gmalloc (len

→ * sizeof (cairo_glyph_t));

9: glyphCount = 0;

10: }

Pay attention to line {8}. Check whether there

→ are vulnerabilities in it.

vulnerable line: [7]
```

(a) Highlighting the 8th line of code

Figure 5: 1 out of 10 highlighted prompt examples for one code

4.2. Attention Calculation

After generating the base prompt and its corresponding highlighted version for each line, LOVA utilizes the LLM to compute the attention maps for both sets of prompts. By calculating the differences between the attention maps of the highlighted and base prompts, LOVA quantitatively assesses the influence of highlighting individual lines on the model's attention distribution.

(C-2) Impracticality of analyzing large attention tensors: In the original attention output, a tensor with the shape (num_tokens, num_tokens, num_layers, num_heads) is generated, representing the attention scores calculated for every pair of tokens across all layers and heads of the model. This structure captures the intricate relationships between tokens and provides valuable insights into how attention is distributed throughout the network:

- The first num_tokens represents the number of tokens, which is typically the number of the segment of code (or the instructions) after tokenization.
- The second num_tokens represents the relationship of each token in the code (or text) with all other tokens that follow it.
- num_layers refers to the different layers in the model. Each layer models the relationships between tokens more deeply, capturing different features of the code.
- num_heads refers to how many independent attention heads exist in each layer. Each head focuses on different aspects of the code to help the model better understand its structure and semantics.

However, directly utilizing this tensor for assessing the impact of highlighting specific tokens or phrases proves to be highly impractical due to its sheer size. For example, with an input consisting of 1000 tokens, 32 decoder layers, and 32 attention heads, the resulting tensor would contain at least 0.5 billion elements. Analyzing such a large tensor becomes computationally prohibitive, as it not only requires substantial memory but also complicates the extraction of meaningful insights regarding the effects of highlighting.

(S-II) Dimensionality reduction for vulnerability-focused attention analysis. Our approach aims to reduce the dimensionality of the attention output, generating a tensor that not only captures the semantic information relevant to vulnerability localization but also contains fewer elements, thereby simplifying subsequent analysis.

Please note that traditional dimensionality reduction methods are not suitable for this context, as they generally aim to preserve similarities from the high-dimensional space when reducing to lower dimensions. In contrast, our scenario focuses on extracting only the relevant components for comparison, rather than maintaining high-dimensional similarities. Additionally, the original attention outputs differ in size due to varying token counts, meaning there is no inherent similarity to retain in the high-dimensional space. Below, we outline the necessary steps for this stage.

(Step 4) Dimensionality Reduction of Tensor: In this step, our method takes the original tensor with the dimensions (num_tokens, num_tokens, num_layers, num_heads) and systematically reduces it into a more manageable matrix, which we refer to as LayerwiseAttn-Mat. This matrix effectively captures essential attention patterns across layers and heads, streamlining further analysis.

• Attention Head Aggregation (Step ①). We combine the attention from all heads by summing them together. This merges the information captured by the different heads into one attention map with the shape (num_tokens, num_tokens, num_layers). Each attention head focuses on different features and relationships between tokens, so

- combining them helps gather a more complete understanding of the input.
- Multi-Layer Attention Preservation (Step ②). The attention from all layers is retained because each decoder layer possesses a unique capacity to capture different semantic and relational features. For instance, lower layers may concentrate on fundamental patterns such as syntax, while higher layers tend to capture more abstract concepts, like the overall meaning of the code [14]. By preserving attention across all layers, we ensure the model captures both basic and complex information, enriching the representation.
- Last Token Attention for Context Aggregation (Step ③). The last token in the prompt is used as a query to determine how much attention it gives to all the preceding tokens, resulting in a matrix of size (num_tokens, num_layers). This is important because, in decoder-only LLMs, attention is computed sequentially, meaning each token only attends to the tokens that came before it. By using the last token as the query, the model can aggregate all prior semantic information from the entire sequence. This approach helps the model make informed predictions or generate the next token, ensuring that the final token is informed by the full context of the prompt.
- Line-Level Attention Summation (Step ④). The attention values of all tokens in each line of the prompt are added together to represent the overall attention for that line. This process results in a matrix of size (num_lines, num_layers). We refer to this matrix as the LayerwiseAttnMat. The key idea here is that attention is not evenly spread across all tokens—some tokens get more focus than others. By summing the attention for each line, we simplify the data while still capturing the most important information, since many tokens may not show significant differences in attention.

(Step 6) Vulnerability-focused Matrix Generation: Next, LOVA calculates the difference between the LayerwiseAttnMat of the highlighted prompt and the base prompt, resulting in the DiffAttnMat. This matrix represents how the model's attention changes when a specific line is highlighted. The purpose of DiffAttnMat is to quantify the effect of highlighting on the model's attention distributions. The difference in attention impact between vulnerable and non-vulnerable lines is most noticeable in two areas: the instruction section of the prompt (where the model receives guidance on how to process the input) and the highlighted line itself. Since these areas show the strongest influence, LOVA focuses on the rows of the DiffAttnMat corresponding to the instruction section and the highlighted line, allowing it to accurately represent the attention shifts caused by highlighting. The final result is a matrix of size ([num lines of instr] + 1, num_layers), known as the VulnAttnMat. This matrix captures the attention variations in a simplified and structured form, making it easier to analyze how the model

distinguishes between vulnerable and non-vulnerable lines. The VulnAttnMat serves as the final output of this stage.

Algorithm 1 Calculate VulnAttnMat

Require: Highlighted Prompt $P_{\text{highlight}}$, Base Prompt P_{base} , LLM M_{LLM}

Ensure: VulnAttnMat V for the highlighted line

(Step 4) Dimensionality Reduction of Tensor

```
1: function GETLAYERWISEATTNMAT(P, M_{LLM})
           T \leftarrow \text{tokenize}(P, M_{\text{LLM}})
 2:
           n \leftarrow \text{countLines}(P)
 3:
           l, h \leftarrow M_{\text{LLM}}.\text{num\_layers}, M_{\text{LLM}}.\text{num\_heads}
 4:
 5:
           M_{\text{tokens}} \leftarrow \text{getOriginalAttention}(M_{\text{LLM}}, T)
           Step ①:
           M_{\text{tokens}} \leftarrow \text{sum}(M_{\text{tokens}}, \text{axis} = 3)
 6:
           Step 2 and 3:
           M_{\text{last}} \leftarrow M_{\text{tokens}}[-1][:][:]
 7:
           Step 4:
           tokens\_for\_each\_line \leftarrow splitTokensByLines(T)
 8:
           Initialize A as an empty matrix of size (n \times l)
 9:
10:
           for line \in \{1, \ldots, n\} do
                A[line] \leftarrow \sum_{i \in \text{tokens for each line}[line]} M_{\text{last}}[i][:]
11:
           end for
12:
           return A
13:
14: end function
```

(Step 6) Vulnerability-focused Matrix Generation:

```
 \begin{array}{ll} \text{15: instruction\_indices} \leftarrow \text{getInstrIndices}(P_{\text{highlight}}) \\ \text{16: highlighted\_index} \leftarrow \text{getHighlightedIndex}(P_{\text{highlight}}) \\ \text{17: } A_{\text{highlighted}} \leftarrow \text{getLayerwiseAttnMat}(P_{\text{highlight}}, M_{\text{LLM}}) \\ \text{18: } A_{\text{base}} \leftarrow \text{getLayerwiseAttnMat}(P_{\text{base}}, M_{\text{LLM}}) \\ \text{19: } V_{\text{highlighted}} \leftarrow A_{\text{highlighted}}[\text{instrution\_indices} + \\ & \text{highlighted\_index}, :] \\ \text{20: } V_{\text{base}} \leftarrow A_{\text{base}}[\text{instruction\_indices} + \text{highlighted\_index}, :] \\ \text{21: } V \leftarrow V_{\text{highlighted}} - V_{\text{base}} \\ \text{22: } \textbf{return } V \\ \end{array}
```

4.3. Vulnerability Line Localization

In this step, after obtaining the VulnAttnMat for each line of the vulnerable program, LOVA determines whether the corresponding line contains a vulnerability or not through analyzing the attention patterns (**Step 6**).

(C-3) Precise Cross-Language Vulnerability Localization. We found that attention patterns for code or text within a single programming language (e.g., Java, C) often exhibit similarities. These patterns tend to reflect semantic information related to vulnerabilities at specific lines and layers in the code. Therefore, summing the attentions at these positions on the VulnAttnMat can yield a corresponding suspicion score (which is a numerical value assigned to data point that quantifies how "suspicious" or anomalous it is) that highlights which sections of the code are more likely to contain vulnerabilities. However, vulnerabilities differ greatly across languages. For instance, in network protocols written

in C, a buffer overflow might occur during packet processing, whereas in a web application, the vulnerability might involve cross-site scripting in JavaScript. These varying contexts mean that attention patterns that indicate a vulnerability in one language may not be applicable to other languages. Additionally, relying solely on these summed attention values provides a suspicion score, which only ranks how suspicious a line of code is compared to others. While this can prioritize certain areas for review, it does not provide a probability for each line being vulnerable, meaning it lacks precision. For effective vulnerability localization, it's crucial to assess the actual probability of each line containing a vulnerability, rather than just ranking the lines by suspicion.

(S-III) Language-aware deep learning-based vulnerability localization. We chose to employ a deep learning approach to classify the attention matrix and determine whether the lines contain vulnerabilities. This method can leverage complex patterns and relationships within the data to provide a more accurate and probabilistic assessment. Unlike simply ranking lines by suspicion score, a deep learning model can learn to identify specific vulnerability signatures across different languages, improving precision in vulnerability localization.

The key question now is which deep learning method is most suitable for our context. In designing the learning algorithm, it was essential to account for the contextual dependencies within the classification task. Whether an attention matrix is deemed suspicious is influenced by the relationships among the attention matrices across all lines of code.

To address this, LOVA implements a learning approach that integrates sequential information by employing a Bi-LSTM model, which effectively captures these contextual dependencies: First, The identification of anomalies of VulnAttnMat is context-dependent. For example, when inputting three lines of code, we generate three corresponding VulnAttnMat: A, B and C. If A significantly deviates from B and C, it suggests that A may be the most suspicious. To effectively model this, a sequence-to-sequence approach is necessary, with LSTM [34] and Transformer being the most common models for such tasks. Second, given that the features of anomalies in our case are relatively distinct, a straightforward LSTM model is sufficient. However, we opted for a Bi-LSTM [33] to leverage its ability to capture bidirectional dependencies, as standard LSTM only processes information in one direction. This is important because the detection of anomalies is influenced by both the preceding and succeeding contexts, rather than solely relying on prior context.

The training process begins by flattening the VulnAttnMat matrices corresponding to each vulnerable program into vectors, generating num_lines vectors. These vectors are then fed into a Bi-LSTM network, followed by a fully connected layer, to perform binary classification at each time step. The model is optimized by minimizing the Binary Cross Entropy Loss, formulated as:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

where N is the number of lines, y_i is the true label of the i-th line, and \hat{y}_i is the predicted probability of the line containing a vulnerability. During the evaluation phase, the flattened vectors of the VulnAttnMat are input into the model to compute suspicious scores for each line. These scores are subsequently ranked in descending order to inform the vulnerability localization process, with scores greater than 0.5 being considered as vulnerable lines.

5. Evaluation

In this section, we first outline the experimental setup (§5.1), followed by an evaluation of LOVA by addressing the following four research questions (§5.2).

- **RQ1.** How accurately does LOVA localize vulnerabilities compared to existing techniques (Effectiveness)?
- **RQ2.** How effective is LOVA when applied to different languages (Scalability)?
- **RQ3.** How effective is LOVA when applied to different LLMs (Robustness)?
- **RQ4.** How does the design of LOVA impact the results (Abalation Study)?

5.1. Experiment Setup

Dataset. Our dataset, comprises vulnerability benchmarks from C/C++ projects (*Big-Vul*), Solidity smart contracts (*SmartFix*), and multi-language CVE records (*CVEFixes*). This allows for a comprehensive evaluation of LOVA's performance across diverse contexts. Specifically:

- **Big-Vul** [9]: A C/C++ code vulnerability benchmark collected from 384 open-source GitHub projects. *Big-Vul* includes both vulnerable code snippets and their corresponding fixing commits. Vulnerabilities are annotated at function context level, categorizing the types of vulnerabilities and associated commits. *Big-Vul* contains a total of 11,823 vulnerable functions.
- SmartFix [8]: A Solidity code vulnerability benchmark collected from multiple sources. SmartFix covers five common types of smart contract vulnerabilities: Integer Overflow, Reentrancy, Ether Leak, Suicidal, and incorrect use of 'tx.origin'. Each vulnerability is annotated with its corresponding line numbers. SmartFix contains a total of 361 contract files.
- CVEFixes [7]: A multi-language code vulnerability benchmark that is automatically collected and curated from CVE records in the public U.S. National Vulnerability Database (NVD). CVEFixes collects both vulnerabilities and the commits that fix them, including 5,365 CVEs and 50,322 methods. CVEFixes encompasses

multiple languages, such as C, C++, PHP, Python, Java, and JavaScript. From these, we selected the three most commonly used languages, C, Python and Java, as benchmarks.

Big-Vul provides vulnerability data at the function context level, CVEFixes provides vulnerability data at both the function and file context levels, and *SmartFix* provides vulnerability data at the contract context level. To evaluate LOVA's performance across different contexts, we conduct evaluations at the function, file, and contract context levels. Additionally, we filter out contexts that exceed 4,000 tokens to comply with the input constraints of LLMs. In total, seven datasets were created, as shown in Table 1. Among them, the C language datasets Big-Vul and CVEFixes-C each contain 1,000 instances. For Java, there are two datasets: CVEFixes-J-M and CVEFixes-J, corresponding to method-level and file-level contexts, respectively. In Python, the CVEFixes-P-M and CVEFixes-P datasets represent method-level and filelevel contexts, respectively. Additionally, Solidity has one dataset, SmartFix, which focuses on contract-level context.

| Context-level | | Method | Contract | File (CVEFixes) | | | |
|---------------|---------|--------------|--------------|-----------------|-------|-----|-----|
| | Big-Vul | CVEFixes-J-M | CVEFixes-P-M | SmartFix | C | S-J | S-P |
| Count | 1,000 | 500 | 500 | 349 | 1,000 | 500 | 500 |
| Avg LoC | 60 | 23 | 29 | 119 | 230 | 168 | 203 |
| Avg Vuln line | 4 | 3 | 4 | 1 | 5 | 7 | 6 |

TABLE 1: All datasets

Baseline.

We primarily compare our approach with the direct outputs of LLMs. Following previous works [26], [42], we provide the vulnerable code to an instruction-based LLM, which outputs all lines containing vulnerabilities and repeats the process k times. After k iterations, the suspicious score for each line is computed. The suspicious score for line m is calculated using the following formula:

$$score(m) = \frac{1}{k} \sum_{i=1}^{k} \left(\frac{1}{|r_i|} \cdot [m \in r_i] \right)$$

where r_i represents all the vulnerable lines output by the LLM in the i-th iteration. In the case of ties, the lines are ranked according to the order in which they were output. For example, We set k=10, meaning each piece of vulnerable code is processed by the LLM to output ten times.

Metrics: We use the following metrics: Top-N measures how many relevant items (e.g., correct localization) appear within the top N results returned by our system. Following previous works, we use N=1,3,5 to evaluate our approach [42]. Precision measures the proportion of correctly identified positive instances out of all instances that were predicted as positive. Recall, also known as sensitivity or true positive rate, measures the proportion of correctly identified positive instances out of all actual positive instances. F1-Score, which is the harmonic mean of Precision and Recall, provide a balanced metric that considers both false positives and false negatives. The F1-Score provides a single

metric that balances the trade-off between precision and recall. The detailed definitions and formulations are omitted here, as they are extensively covered in other works.

Model: All experiments, except for RQ3 (Robustness), were conducted on the Llama-3.1-8B-Instruct [18] model, which is one of the state-of-the-art LLMs with fewer than 10b parameters.

Environment: The experiments were conducted on a high-performance computing system running Ubuntu 22.04.4 LTS. The system is equipped with an NVIDIA A800 80GB PCIe GPU with CUDA version 12.4 and an AMD EPYC 7763 processor.

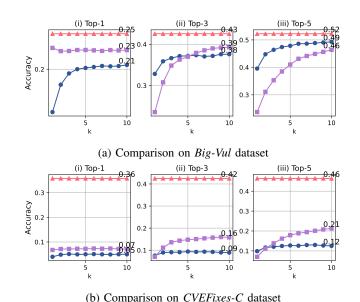
5.2. Experiment Results

Effectiveness (RQ1): The goal of the experiment is to evaluate how accurately LOVA localizes vulnerabilities compared to existing techniques. The evaluation is conducted using four different decoding-based settings of a LLM: (i) vanilla output, (ii) few-shot chain-of-thought (CoT) output [1], (iii) Mixture-of-Agents (MoA) [44] output and (iv) rStar [45] output. In the vanilla output setting, the model directly generates the final result. In the few-shot CoT output, the model employs the few-shot CoT technique, where first reasoning examples are provided to the LLM, and then it first outputs the reasoning process before providing the final answer. This method enhances the model's reasoning capabilities. The MoA output requires the LLM to generate multiple results and then evaluate each output by themselves. Based on these evaluations, the model will provide the final output. Using MoA can help reduce the bias introduced by the randomness of the LLM. rStar combines the reasoning and validation processes by first having the LLM generate multiple human-like reasoning actions to form different reasoning trajectories. Next, the LLM evaluates these trajectories to obtain the most mutually consistent trajectory as the final result. rStar is currently one of the state-of-the-art methods for enhancing the reasoning capabilities of LLMs based on reasoning chains construction. For each dataset, we performed 5-fold cross-validation to obtain results across the entire dataset.

| Method | В | ig-Vul | | CVEFixes-C | | | | |
|--|----------------------------|-----------------------------|-----------------------------|--------------------------|-----------------------------|--------------------------|--|--|
| | Precision Recall F1 | | Precision | Recall F | | | | |
| LOVA | 37.6 | 13.2 | 19.5 | 29.9 | 10.5 | 15.5 | | |
| Vanilla output CoT output MoA output rStar output | 7.8 20.4 8.5 10.8 | 43.4 5.8 38.2 18.5 | 13.1 9.0 13.9 13.7 | 2.4 7.4 2.9 4.1 | 45.1 1.8 32.2 11.0 | 4.6 2.9 5.4 6.0 | | |

TABLE 2: Comparison of Effectiveness Across Methods on C Language Datasets: *Big-Vul* and *CVEFixes-C*. *SmartFix* is excluded in this RQ as it is specific to Solidity language.

As shown in Table 2, LOVA achieved the highest F1-score on both the *Big-Vul* and *CVEFixes-C* datasets compared to the baseline. Specifically, on *Big-Vul*, it improved



Comparison of Tax N Later and OVA and

Figure 6: Comparison of Top-N between LOVA and baselines

by $1.5\times$ and $2.2\times$ over vanilla output and CoT output, respectively. On CVEFixes-C, it achieved gains of $3.4\times$ and $5.3\times$ over vanilla output and CoT output, respectively. While vanilla output attained the highest recall, it had the lowest precision, indicating a higher likelihood of producing false positives. By examining the model's output, it can be seen that vanilla output often marks a large number of suspicious lines, leading to more false positives. In contrast, CoT output, after explaining the vulnerability, more precisely identifies the vulnerable lines, resulting in higher precision but lower recall. LOVA achieved the highest precision, striking a balance between precision and recall compared to the baselines, which led to the highest F1-score.

In addition to F1-score, Top-N accuracy is also a useful metric for evaluating vulnerability localization. We compare the Top-N accuracy of LOVA with both vanilla output and CoT output, and investigate how combining multiple rounds of LLM outputs affects the accuracy. The experimental results are shown in Figure 6. The x-axis represents the accuracy after combining the results of k outputs, and the y-axis represents the Top-N accuracy, where N = 1, 3, 5. LOVA demonstrates a marked improvement over both the vanilla and CoT outputs. In the Top-1, Top-3, and Top-5 categories, the vanilla output fails to generate accurate vulnerability localization results consistently. To further evaluate LOVA 's performance in managing long contexts, we compared its effectiveness against vanilla and CoT outputs across different ranges of lines of code (LoC) using the Big-Vul and CVEFixes-C datasets, as shown in Figure 7. The results indicate a substantial decline in the accuracy of both vanilla and CoT outputs as the LoC increases, with average Top-1 accuracy dropping below 10% when the LoC exceeds 80 lines. In contrast, LOVA consistently maintains high accuracy across varying LoC ranges.

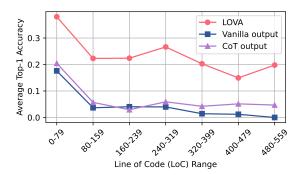


Figure 7: Top-1 Accuracy on different line ranges

Answer (RQ1). LOVA demonstrates exceptional vulnerability localization capabilities compared to the direct outputs of LLMs. In longer contexts, LOVA performs significantly better, achieving up to a $5 \times$ improvement over the baseline on F1-score.

Scalability (RQ2): One key advantage of LLMs over traditional smaller machine learning models lies in their ability to generalize across multiple languages. To evaluate the scalability, we conduct vulnerability localization in three additional languages-Python, Java, and Solidity-beyond C/C++. The results are shown in Table 3. It is clear that LOVA achieved the highest Top-1 accuracy and F1-score across all datasets. On the method-level-context datasets CVEFixes-J-M and CVEFixes-P-M, LOVA improved the F1-score by up to $1.6\times$ and $1.7\times$ compared to the baselines, respectively. In longer contexts, LOVA achieved an impressive 14.6× improvement on the contract-level context dataset SmartFix compared to the vanilla output. The CoT output also achieved a high F1 score of 35.1 on this dataset. This may be due to the SmartFix dataset containing only a few simple types of vulnerabilities, making it easier for CoT to infer vulnerabilities. For the file-level-context datasets CVEFixes-J and CVEFixes-P, it achieved increases of up to $6.7 \times$ and $3.9 \times$, respectively, demonstrating LOVA's remarkable ability to handle long contexts effectively.

Answer (RQ2). LOVA achieves excellent results in vulnerability localization across multiple programming languages, showing up to a 14.6× improvement over the baseline. This highlights LOVA's impressive generalization capabilities.

Robustness (RQ3): The robustness of LOVA across various LLM is a critical factor in assessing its general applicability. To evaluate its effectiveness, we applied LOVA to multiple state-of-the-art LLMs, including Llama-3.1-8B-Instruct [18], Mistral-7B-Instruct-v0.2 [46], and Phi-3.5-mini-instruct [47], and measured its performance across vulnerability localization tasks. The results, summarized in Table 4, show that LOVA consistently outperforms baseline methods regardless of the underlying LLM. Specifically,

when applied to Mistral, LOVA improved the Top-1 accuracy by 1.1× compared to CoT outputs and achieved an F1 score that was 1.5× higher than vanilla outputs on method-level vulnerability localization tasks. On Phi, LOVA exhibited relatively low accuracy across file-level vulnerability datasets. This may be due to Phi having only 3.8b parameters, the fewest among the three models. However, LOVA still achieved a 6× increase in Top-1 accuracy compared to vanilla outputs, demonstrating its adaptability across models with varying parameter sizes. Notably, LOVA 's performance on Llama was particularly impressive in handling longer contexts, where it outperformed CoT outputs by up to $4.9 \times$ in terms of Top-1 accuracy and showed a 3.4× increase in F1 score compared to the vanilla output on the file-level datasets. This demonstrates LOVA 's robust capability to adapt to different LLM architectures and maintain high accuracy in diverse vulnerability localization scenarios.

Answer (RQ3). LOVA can be applied across various LLMs, as the self-attention mechanisms in different models consistently demonstrate semantic connections with the output.

Abalation Study (RQ4): To better assess the contribution of each design element, we conducted an ablation study focusing on three key components of the method: Code Line Highlight, Attention Calculation, and Vulnerability Line Localization. From this, we derived three distinct variants of LOVA: LOVA-C, LOVA-A, and LOVA-V. In LOVA-C, the code line highlighting mechanism was altered by introducing marks at specific positions that require emphasis. LOVA-A utilizes a average pooling technique for dimensionality reduction and feature extraction, compressing the attention outputs into vectors of uniform length. Lastly, LOVA-V replaces the Bi-LSTM in vulnerability localization with a multilayer perceptron (MLP) network, omitting the use of context during the process.

The results of the ablation study are summarized in Table 5. As expected, each variant of LOVA demonstrated varying degrees of effectiveness compared to the full model. LOVA-C, which modifies the code line highlighting approach, showed a moderate reduction in performance, with the Top-1 accuracy dropping by 5% compared to the full LOVA, suggesting that the original highlighting strategy plays a critical role in precise vulnerability localization. In LOVA-V, which replaces contextual information with an MLP classifier, the accuracy dropped by 4%, indicating that contextual information plays a certain role in aiding vulnerability classification. However, LOVA-A, where attention outputs were compressed through average pooling, exhibited the most significant performance decline, with an F1 score reduced by 9%. This underscores the importance of effective dimensionality reduction and feature selection techniques for accurate vulnerability localization, as not all parts of the attention output can capture the semantic meaning relevant to identifying vulnerabilities.

| | | Java | | | | Py | Solidity | | | | |
|------------------------|-----------------|--|--------------------|---|-------------|---------------------------------------|------------------|---|-----------------|--|--|
| | <i>C</i> | VEFixes-J | ces-J CVEFixes-J-M | | CVEFixes-P | | CV | CVEFixes-P-M | | SmartFix | |
| | F1 | Top-1 | F1 | Top-1 | F1 | Top-1 | F1 | Top-1 | F1 | Top-1 | |
| LOVA Vanilla output | 21.3 9.0 | 181 (36.2 %) 54 (10.8%) | 34.6 22.1 | 281 (56.2 %) 173 (34.6%) | 18.1 6.5 | 98 (19.6 %) 66 (13.2%) | 31.0 21.4 | 204 (40.8 %) 177 (35.4%) | 80.0 5.5 | 284 (81.3 %) 90 (25.8%) | |
| CoT output | 3.2 | 59 (11.8%) | 22.1 | 177 (35.4%) | 4.7 | 77 (15.5%) | 17.9 | 150 (30.0%) | 35.1 | 134 (38.3%) | |

TABLE 3: Comparison of F1 and Top-1 Scores across Models for Java, Python, and Solidity

| Datasets | Methods | Ll | ama | Mi | istral | Phi | |
|------------|----------------|------|-------|------|--------|------|-------|
| | | F1 | Top-1 | F1 | Top-1 | F1 | Top-1 |
| | LOVA | 19.5 | 24.5 | 18.0 | 22.9 | 17.0 | 23.9 |
| Big-Vul | Vanilla output | 13.1 | 20.6 | 12.4 | 22.4 | 12.9 | 23.4 |
| | CoT output | 9.0 | 23.0 | 10.5 | 21.2 | 8.2 | 14.7 |
| | LOVA | 15.5 | 35.8 | 14.0 | 32.7 | 11.4 | 29.0 |
| CVEFixes-C | Vanilla output | 4.6 | 5.1 | 4.7 | 6.5 | 4.9 | 4.8 |
| | CoT output | 2.9 | 7.3 | 3.5 | 4.9 | 3.4 | 3.7 |

TABLE 4: Comparison of Methods Across C Datasets: *Big-Vul* and *CVEFixes*. *SmartFix* is excluded in this RQ as it is specific to Solidity language.

| Method | Big-Vul | | | | CVEFixes-C | | | | |
|----------------------------|-----------------------------|---------------------|----------------------|----------------------|-----------------------------|---------------------|---------------------|----------------------|--|
| | P | R | F1 | Top-1 | P | R | F1 | Top-1 | |
| LOVA | 37.6 | 13.2 | 19.5 | 24.5 | 29.9 | 10.5 | 15.5 | 35.8 | |
| LOVA-C LOVA-A LOVA-V | 28.7 40.2 29.0 | 12.7 9.3 13.0 | 17.6 15.0 18.0 | 22.3 21.1 22.7 | 25.2 47.0 22.5 | 10.0 3.4 10.4 | 14.3 6.3 14.2 | 30.2 22.1 32.0 | |

TABLE 5: Accuracy comparison across LOVA variants and C datasets

Answer (RQ4). The ablation study reveals that each component is essential for LOVA 's performance. LOVA-C shows a moderate decline, while LOVA-V overlooks the importance of contextual information in vulnerability detection, resulting in reduced accuracy. LOVA-A performs the worst due to the absence of appropriate dimensionality reduction and feature selection methods.

6. Related Work

Vulnerability Localization. Automated vulnerability localization can significantly reduce developers' efforts in identifying and diagnosing vulnerabilities. Traditional vulnerability localization tools, such as [39], [40], rely on predefined detectors for different vulnerabilities, limiting scalability. Deep learning-based methods[31], [30], [29], [28], [27], [25], [23], [24] focus on learning different program features to determine whether a specific line contains a vulnerability. Program features include Control Flow Graph (CFG), Program Dependency Graph (PDG), Abstract Syntax Tree (AST) and more. However, deep learning methods suffer from poor generalization. On the one hand, these methods are often limited to a specific programming language, and on the other hand, collecting various program features requires the program to be compilable or executable in order to

gather relevant information, imposing additional requirements on the program structure.

With the rise of LLMs, LLM-based vulnerability localization has emerged [6]. The key advantage of LLM-based approaches is their generalizability, allowing them to be applied across different types and domains of vulnerabilities. Current LLM-based methods can be broadly categorized into two types: directly output methods [2], [26] and finetuning-based methods [32], [26]. Our approach innovatively leverages self-attention mechanisms as indicators for vulnerability localization, significantly improving accuracy compared to directly output by LLMs, without the high cost of fine-tuning. Additionally, LOVA mitigates the performance degradation of LLM in long-context scenarios, making it better suited to real-world vulnerability localization needs.

LLM Attention for Code Analysis. Self-attention mechanisms are widely used as interpretability tools for code. For code syntax interpretability, Wan et al. [43] used attention to assess whether pre-trained language models can capture syntactical relationships in source code. They observed that tokens sharing a common parent node in the Abstract Syntax Tree (AST) receive higher attention, suggesting that attention can help explain the model's ability to capture syntactic structures. In terms of code semantic interpretability, Ma et al. [35] explored whether tokens within the same Control Dependency Graph (CDG), Control Flow Graph (CFG), or Data Dependency Graph (DDG) receive significantly increased attention. Their results indicate that many attention heads are indeed capable of capturing semantic information. Additionally, self-attention mechanisms can represent the importance of each token in a prompt relative to the output. Kou [36] examined whether the content that LLMs focus on during coding tasks aligns with human. They used the last token in the prompt and the last token in the output as query tokens, calculating the attention to each token in the prompt as a measure of the LLM's focus during generation. We are the first to apply attention as an indicator for vulnerability localization.

7. Conclusion

Our work demonstrates that leveraging self-attention mechanisms within LLMs offers a powerful new approach to vulnerability localization. By systematically tracking attention weights, we can effectively identify lines of code that are more likely to contain vulnerabilities, even in large and complex codebases. Our experiments show significant improvements in vulnerability localization accuracy, demonstrates

strating the potential of self-attention-based approaches to advance the state of the art in software security. We hope our framework will open new possibilities for automating vulnerability detection, and reduce both the time and resources required for secure software development.

References

- [1] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou et al., "Chain-of-thought prompting elicits reasoning in large language models," Advances in neural information processing systems, vol. 35, pp. 24824–24837, 2022.
- [2] Y. Nong, M. Aldeen, L. Cheng, H. Hu, F. Chen, and H. Cai, "Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities," <u>arXiv preprint arXiv:2402.17230</u>, 2024.
- [3] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," <u>arXiv preprint</u> arXiv:2203.11171, 2022.
- [4] S. Kang, G. An, and S. Yoo, "A quantitative and qualitative evaluation of llm-based explainable fault localization," <u>Proceedings of the ACM</u> on Software Engineering, vol. 1, no. FSE, pp. 1424–1446, 2024.
- [5] MITRE, "Cve common vulnerabilities and exposures," https://cve. mitre.org/, 2024, accessed: 2024-10-08.
- [6] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A survey on large language model (Ilm) security and privacy: The good, the bad, and the ugly," High-Confidence Computing, p. 100211, 2024.
- [7] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," in Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, 2021, pp. 30–39
- [8] S. So and H. Oh, "Smartfix: Fixing vulnerable smart contracts by accelerating generate-and-verify repair using statistical models," in Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 185–197.
- [9] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in <u>Proceedings of the 17th International Conference on Mining Software Repositories</u>, 2020, pp. 508–512.
- [10] M. D. Purba, A. Ghosh, B. J. Radford, and B. Chu, "Software vulnerability detection using large language models," in 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2023, pp. 112–119.
- [11] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," <u>Transactions of the Association for Computational Linguistics</u>, vol. 12, pp. 157–173, 2024.
- [12] Y. Kuratov, A. Bulatov, P. Anokhin, I. Rodkin, D. Sorokin, A. Sorokin, and M. Burtsev, "Babilong: Testing the limits of llms with long context reasoning-in-a-haystack," <u>arXiv preprint</u> arXiv:2406.10149, 2024.
- [13] A. Vaswani, "Attention is all you need," <u>Advances in Neural</u> Information Processing Systems, 2017.
- [14] W. Ma, S. Liu, M. Zhao, X. Xie, W. Wang, Q. Hu, J. Zhang, and Y. Liu, "Unveiling code pre-trained models: Investigating syntax and semantics capacities," <u>ACM Transactions on Software Engineering</u> and Methodology, vol. 33, no. 7, pp. 1–29, 2024.
- [15] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805, 2018.

- [16] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat et al., "Gpt-4 technical report," arXiv preprint arXiv:2303.08774, 2023.
- [17] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," <u>Journal of machine learning</u> research, vol. 21, no. 140, pp. 1–67, 2020.
- [18] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar et al., "Llama: Open and efficient foundation language models," <u>arXiv preprint</u> arXiv:2302.13971, 2023.
- [19] M. Zhang, Z. Meng, and N. Collier, "Attention instruction: Amplifying attention in the middle via prompting," <u>arXiv preprint</u> arXiv:2406.17095, 2024.
- [20] B. Berabi, A. Gronskiy, V. Raychev, G. Sivanrupan, V. Chibotaru, and M. Vechev, "Deepcode ai fix: Fixing security vulnerabilities with large language models," arXiv preprint arXiv:2402.13291, 2024.
- [21] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? an evaluation on quixbugs," in Proceedings of the Third International Workshop on Automated Program Repair, 2022, pp. 69–75.
- [22] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023, pp. 1430–1442.
- [23] Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, and W. Lee, "{VulChecker}: Graph-based vulnerability localization in source code," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 6557–6574.
- [24] N. Visalli, L. Deng, A. Al-Suwaida, Z. Brown, M. Joshi, and B. Wei, "Towards automated security vulnerability and software defect localization," in 2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA). IEEE, 2019, pp. 90–93.
- [25] P. H. N. Rajput, C. Doumanidis, and M. Maniatakos, "{ICSPatch}: Automated vulnerability localization and {Non-Intrusive} hotpatching in industrial control systems using data dependence graphs," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 6861–6876.
- [26] J. Zhang, C. Wang, A. Li, W. Sun, C. Zhang, W. Ma, and Y. Liu, "An empirical study of automated vulnerability localization with large language models," arXiv preprint arXiv:2404.00287, 2024.
- [27] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in <u>Proceedings of the 19th</u> <u>International Conference on Mining Software Repositories</u>, 2022, pp. 608–620.
- [28] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering, 2022, pp. 935–947.
- [29] J. Zhang, S. Liu, X. Wang, T. Li, and Y. Liu, "Learning to locate and describe vulnerabilities," in <u>2023 38th IEEE/ACM International</u> <u>Conference on Automated Software Engineering (ASE)</u>. IEEE, 2023, pp. 332–344.
- [30] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," <u>IEEE Transactions on Dependable and Secure Computing</u>, vol. 19, no. 4, pp. 2821–2837, 2021.
- [31] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 292–303.

- [32] A. Z. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large language models for test-free fault localization," in <u>Proceedings of the 46th IEEE/ACM International Conference on Software Engineering</u>, 2024, pp. 1–12.
- [33] A. Graves, N. Jaitly, and A.-r. Mohamed, "Hybrid speech recognition with deep bidirectional lstm," in 2013 IEEE workshop on automatic speech recognition and understanding. IEEE, 2013, pp. 273–278.
- [34] A. Sherstinsky, "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network," <u>Physica D: Nonlinear Phenomena</u>, vol. 404, p. 132306, 2020.
- [35] W. Ma, S. Liu, M. Zhao, X. Xie, W. Wang, Q. Hu, J. Zhang, and Y. Liu, "Unveiling code pre-trained models: Investigating syntax and semantics capacities," <u>ACM Transactions on Software Engineering</u> and Methodology, p. 3664606, May 2024.
- [36] B. Kou, S. Chen, Z. Wang, L. Ma, and T. Zhang, "Do large language models pay similar attention like human programmers when generating code?" Proceedings of the ACM on Software Engineering, vol. 1, no. FSE, pp. 2261–2284, Jul. 2024.
- [37] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," <u>IEEE Transactions on Software Engineering</u>, vol. 47, no. 2, pp. 332–347, Feb. 2021.
- [38] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," <u>IEEE Transactions on Software Engineering</u>, vol. 42, no. 8, pp. 707–740, Aug. 2016.
- [39] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "On the accuracy of spectrum-based fault localization," in <u>Testing: Academic and Industrial Conference Practice and Research Techniques MUTATION (TAICPART-MUTATION 2007)</u>, Sep. 2007, pp. 89–98.
- [40] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in <u>Verification and Validation</u> 2014 IEEE Seventh International Conference on Software Testing, Mar. 2014, pp. 153–162.
- [41] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in <u>Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis</u>, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 169–180.
- [42] S. Kang, G. An, and S. Yoo, "A quantitative and qualitative evaluation of llm-based explainable fault localization," <u>Proceedings of the ACM on Software Engineering</u>, vol. 1, no. FSE, pp. 1424–1446, Jul. 2024.
- [43] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture?: A structural analysis of pre-trained language models for source code," in <u>Proceedings of the 44th International Conference on Software Engineering</u>. Pittsburgh Pennsylvania: ACM, May 2022, pp. 2377–2388.
- [44] J. Wang, J. Wang, B. Athiwaratkun, C. Zhang, and J. Zou, "Mixture-of-agents enhances large language model capabilities," <u>arXiv preprint</u> arXiv:2406.04692, 2024.
- [45] Z. Qi, M. Ma, J. Xu, L. L. Zhang, F. Yang, and M. Yang, "Mutual reasoning makes smaller llms stronger problem-solvers," <u>arXiv</u> preprint arXiv:2408.06195, 2024.
- [46] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier et al., "Mistral 7b," arXiv preprint arXiv:2310.06825, 2023.
- [47] M. Abdin, S. A. Jacobs, A. A. Awan, J. Aneja, A. Awadallah, H. Awadalla, N. Bach, A. Bahree, A. Bakhtiari, H. Behl et al., "Phi-3 technical report: A highly capable language model locally on your phone," arXiv preprint arXiv:2404.14219, 2024.