FuncVul: An Effective Function Level Vulnerability Detection Model using LLM and Code Chunk

Sajal Halder, Muhammad Ejaz Ahmed, and Seyit Camtepe

Data61, CSIRO, Australia
(sajal.halder,ejaz.ahmed,seyit.camtepe)@data61.csiro.au

Abstract. Software supply chain vulnerabilities arise when attackers exploit weaknesses by injecting vulnerable code into widely used packages or libraries within software repositories. While most existing approaches focus on identifying vulnerable packages or libraries, they often overlook the specific functions responsible for these vulnerabilities. Pinpointing vulnerable functions within packages or libraries is critical, as it can significantly reduce the risks associated with using open-source software. Identifying vulnerable patches is challenging because developers often submit code changes that are unrelated to vulnerability fixes. To address this issue, this paper introduces FuncVul, an innovative code chunk-based model for function-level vulnerability detection in C/C++ and Python, designed to identify multiple vulnerabilities within a function by focusing on smaller, critical code segments. To assess the model's effectiveness, we construct six code and generic code chunk based datasets using two approaches: (1) integrating patch information with large language models to label vulnerable samples and (2) leveraging large language models alone to detect vulnerabilities in function-level code. To design FuncVul vulnerability model, we utilise GraphCodeBERT fine tune model that captures both the syntactic and semantic aspects of code. Experimental results show that FuncVul outperforms existing state-of-the-art models, achieving an average accuracy of 87-92% and an F1 score of 86-92% across all datasets. Furthermore, we have demonstrated that our code-chunk-based FuncVul model improves 53.9% accuracy and 42.0% F1-score than the full function-based vulnerability prediction. The FuncVul code and datasets are publicly available on GitHub [1]

Keywords: Function Code · Vulnerability Detection · Code Chunk · Software Supply Chain · Large Language Model

1 Introduction

With the rapid expansion of technology, cybersecurity has become a growing priority. By October 2024, the National Vulnerability Database (NVD) recorded over 240,000 reported Common Vulnerabilities and Exposures (CVEs) [3, 20]. This number has steadily risen, with an average growth rate of 15-20% per year. Detecting vulnerabilities in

[1] https://github.com/sajalhalder/FuncVul.

In The 30th European Symposium on Research in Computer Security (ESORICS), 22 Sep - 26 Sep, 2025, Toulouse, France.

C/C++ and Python code is a challenging process that demands thorough analysis of the codebase's structure, syntax, and semantics to reveal potential weaknesses exploitable by attackers. Identifying vulnerable functions within the package is crucial because it allows developers to focus their efforts on resolving the specific issue rather than discarding the entire package. It enables organizations to prioritize fixes based on the severity and significance of the affected functions. Moreover, identifying vulnerable source enables the developer to resolve them quickly, minimizing the impact on customers and ensuring service continuity. Identifying vulnerable functions within the vast number of packages released daily is both time-consuming and requires specialized expertise in security. Thus, an automated model capable of effectively identifying vulnerable functions is essential.

Existing research has explored code similarity techniques to detect vulnerable code patterns using machine learning [21], deep learning [2, 29, 14], and graph-based models [26]. Yuan et al. [34] combined serialized features from Gated Recurrent Units (GRUs) and structural features from Abstract Syntax Trees (ASTs) via Gated Graph Recurrent Networks (GGRNs), addressing data scarcity and imbalance with a Random Forest model, achieving superior performance. Vo et al. [25] found that pre-trained deep models for vulnerability type identification (VTI) offered limited improvement over classical TF-IDF baselines and enhanced them by identifying key code tokens. Wang et al. [28] developed ReposVul, a repository-level dataset created using an automated framework with modules for untangling vulnerabilities, dependency extraction, and filtering outdated patches. Other works explored context-aware embeddings [30], RoBERTa models pre-trained on open-source C/C++ code [9], and Word2Vec-LSTM pipelines for Python code [29]. However, these approaches face two key limitations. First, while they can identify whether a function is vulnerable, they cannot determine the exact number of vulnerabilities within the function. Second, pinpointing the precise lines of code that contain vulnerabilities remains a challenge. As a result, security analysts are compelled to manually review functions, significantly increasing the time and effort required for vulnerability analysis. Addressing these limitations is vital for streamlining the vulnerability identification process and improving efficiency.

To address these limitations, we propose a code chunk-based function vulnerability detection model capable of identifying multiple vulnerabilities within a function. Moreover, it highlights smaller code chunks containing vulnerabilities, significantly reducing the time required for experts to address the issues. To sum up, in this paper we aim to answer the following research questions.

RQ1: What modeling strategies can be employed to accurately detect function level vulnerabilities?
RQ2: Does leveraging code chunks enhance model performance compared to analyzing full-function code?
RQ3: Does the FuncVul model leverage generalized code properties for vulnerability detection?
RQ4: How effective is our approach at detecting vulnerabilities in unseen projects?
RQ5: How does the performance of our approach vary with different numbers of source lines in a code chunk?
RQ6. Is our proposed model capable of detecting multiple vulnerabilities within a single function's code?

To evaluate the performance of our proposed FuncVul model, we compared it against several state-of-the-art models: CodeBERT [5], CustomVulBERTa [9], BERT [12] and VUDENC [29] across six datasets. The main contribution of this research work are as follows.

- We propose a novel code chunk-based **Func**tion **Vul**nerability (**FuncVul**) detection model capable of identifying multiple vulnerabilities within a function and, importantly, to identify the specific, smaller code segments responsible for those vulnerabilities.
- We collected and curated four datasets from diverse data sources, such as project source codes from GitHub and vulnerability advisory databases, i.e., OSV. Additionally, we developed novel methods to analyze, process, and curate source code using large language models (LLMs).
- We employ a fine-tuned GraphCodeBERT model for function-level vulnerability prediction, as it effectively captures both syntactic and semantic similarities within the code.
- Our experimental results demonstrate that the proposed FuncVul model outperforms state-of-the-art baselines, achieving an average accuracy of 89.39% and an F1 score of 88.94% across the six datasets.
- Additionally, we show that the FuncVul model is highly generic, capable of handling diverse code chunks and identifying new vulnerable patterns effectively.

The remaining part of the paper is organized as follows. We briefly describe the relevant existing works in Section 2. Then, we discuss the problem statement in Section 3. We introduce our proposed model in Section 4. After that, we present our experiments in Section 5. Finally, we conclude the paper with potential future research directions in Section 6.

2 Existing Works

Software vulnerability detection is a significant challenge for security researchers in both academia and industry. Researcher classified vulnerability research in two directions: vulnerability dataset creation and vulnerability sections. This section reviews related work on function label vulnerability detection.

Hanif et al. [9] presented a deep learning framework named VulBERTa to identify security vulnerabilities in source code. It leverages a RoBERTa model pre-trained with a specialized tokenization pipeline on real-world open-source C/C++ code considering code syntax and semantics. Warschinski et al. [29] proposed VUDENC, a deep learning model for detecting vulnerabilities in Python code, combining Word2Vec embeddings with an LSTM network to classify vulnerable code sequences. Yuan et al.[34] proposed a hybrid approach combining GRU-based serialized features and Gated Graph Recurrent Network (GGRN0 based structural features, using a Random Forest model to improve vulnerability identification under data scarcity and imbalance. Vo et al. [25] showed that deep pre-trained models offer limited gains over a TF-IDF baseline for vulnerability type identification and proposed a lightweight enhancement to identify key tokens for each vulnerability type.

Wang et al. [28] developed ReposVul, a repository-level vulnerability dataset built using an automated framework with modules for untangling fixes, extracting multi-level dependencies, and filtering outdated patches. Wei et al. [30] proposed a supervised framework that utilised pre-trained context-aware embeddings (ELMo) and a Bi-LSTM layer to capture deep contextual representations and learn long-range code dependencies in source code to detect function vulnerability. Li et al. proposed $VulPecker$ [15], an automated tool to identify known vulnerabilities within software source code. It utilises features derived from patches and employs a variety of code-similarity algorithms. Fu et al. [6] designed a Transformer-based line-level vulnerability prediction method named $LineVul$ to detect vulnerabilities in C/C++ codes. Tran et al. [23] implement DetectVul which is a statement-level vulnerability detection approach for Python that uses self-attention to learn patterns directly from raw code, avoiding graph extraction. Li et al.[16] proposed *VulDeePecker* a code gadget-based model that represents programs as vectors by grouping semantically related, though not necessarily consecutive, lines of code.

GNN-based methods have demonstrated state-of-the-art performance in vulnerability detection by leveraging graph representations of source code. Hin et al. [10] introduced a deep learning framework called $LineVD$ for statement-level vulnerability detection in C/C++ codes that combines GNNs and transformers. Notable approaches include Devign [35], which uses graph-level classification with semantic code representations, and GGNN-based methods [27] that capture data, control, and call dependencies with majority voting from traditional classifiers. Li et al. [13] introduced a feature-attentive GCN on program dependency graphs, while VulCNN [32] transformed source code into semantic-preserving images. ReGVD [19] utilised token embeddings from pre-trained models, residual connections, and pooling techniques to enhance graph representations. AMPLE [31] improved performance by refining graph structures and capturing distant node relations. Islam et al. [11] introduced Poacher Flow edges to bridge static and dynamic analyses and manage long-range dependencies for richer vulnerability detection.

Large pre-trained language models like BERT and GPT have become a dominant learning paradigm, achieving notable success in computer vision and NLP by leveraging semantic knowledge from large-scale corpora. This pre-trained and fine-tuned approach has also been extended to code-related tasks with models like RoBERTa [17], CodeBERT [5], and GraphCodeBERT [7], significantly improving applications such as automated program repair [33] and code vulnerability detection. All these works primarily focus on either software package vulnerabilities or function-level vulnerabilities. Although the LineVul [6] model predicts vulnerabilities at the line level, it may still miss certain vulnerabilities due to overlooked semantic patterns. They fail to identify the number of vulnerable lines within the function code or pinpoint the exact location of the vulnerabilities. Therefore, research that can effectively address function-level vulnerabilities is critically needed.

Now a days, LLM are used in different domains, including vulnerability detection. Lu et al. [18] proposed GRACE, a vulnerability detection framework that enhances LLM-based analysis by incorporating code's graph structural information and in-context learning. Akuthota et al. [1] utilised LLM for the purpose of identifying and

monitoring vulnerabilities. Meanwhile, Guo et al. [8] explored the ability of LLMs to detect vulnerabilities in source code by evaluating models outside their typical uses to assess their potential in cybersecurity tasks. However, our proposed model is different than the existing models, where we use LLM to generate datasets and use code based fine tune model to detect vulnerability.

2.1 Differences with Previous Works

Our proposed function-level vulnerability detection model introduces several key advancements over state-of-the-art techniques. First, unlike existing approaches that analyse entire functions or line-based vulnerability detection, our model focuses on code chunk-based vulnerability detection in C/C++ and Python. This approach significantly reduces the time required by experts or developers to address vulnerabilities. Second, our code chunk-based method enables the detection of multiple vulnerabilities within a function, whereas existing models typically provide only a binary assessment of whether a function is vulnerable. Third, our model leverages a large language model that is capable of supporting code chunks from different programming languages, eliminating the need for language-specific preprocessing required by existing methods. Finally, we utilise the pre-trained GraphCodeBERT model to build a function-level vulnerability detection framework that effectively captures both syntactic and semantic features, surpassing traditional approaches that rely solely on code similarity.

3 Preliminaries & Problem Statement

In this section, we first present the key preliminary definitions and then describe the problem statement.

Definition 1 (Function Code Chunk) *: A Function Code Chunk (FC) refers to a contiguous segment of lines extracted from a function's source code, typically centered around a code change or edit. It includes a few lines before and after the change to preserve local context for vulnerability analysis.*

Definition 2 (Generic Code Chunk) *: Generic Code Chunk represents the segments of code where variable names, function names, and other identifiers have been replaced with generic placeholders (e.g., F_1, F_2,...,F_n for functions and V_1, V_2, ..., V_n for variables).*

This generic code chunk transformation standardizes the code, removing specific naming conventions or contextual biases, and ensures a consistent format that focuses on structural and syntactic patterns.

Definition 3 (3-Line Extended-Based Code Chunk) *: 3-Line Extended-Based Code Chunk refers to a segment of code centered on an edited line (or lines), augmented with three preceding and three succeeding lines from the edited lines in the function.*

This design captures the semantic context of code for vulnerability detection. Generally, the code edited lines is fewer than 10 lines. If the edited length is more than 10 lines, we consider edited lines only to make the code chunk. Thus, we can define the code chunk as follows.

$$\text{Code Chunk} = \begin{cases} \{L_i \mid i \in [\min(E) - 3, \max(E) + 3]\} & \text{if } |E| \leq 10 \\ E & \text{if } |E| > 10 \end{cases} \tag{1}$$

where L_i represents the i^{th} line of the function code, $min(E)$ and $max(E)$ refer starting and ending edited lines, respectively. This code chunk approach provides a contextualized view of the code, enabling better understanding and analysis of the detected lines within their surrounding context.

Problem Definition: Given a C/C++ or Python based software patch information based modified function code chunk (fc_i). The main goal of this research work is to develop a vulnerable code detector $\mathcal{V}$ which can identify patch-modified codes as vulnerable or non-vulnerable. It can be defined as follows.

$$\mathcal{V}(fc_i) = \begin{cases} 1, & \text{if } fc_i \text{ is vulnerable,} \\ 0, & \text{non-vulnerable} \end{cases} \tag{2}$$

To solve the problem, we propose 3-line extended based code chunk to detect function label vulnerability using code-based fine-tune models in C/C++ or Python code.

4 Proposed Model

In this paper, we propose an effective function-level vulnerability detection framework that leverages large language models (LLMs) alongside specialized code vulnerability detection techniques. To generate ground truth data, we utilise two distinct types of LLM prompts and employ an additional prompt to transform code chunks into generic code chunks. Subsequently, we fine-tune the prediction models using advanced code vulnerability identification techniques. The next two subsections provide a detailed explanation of the data generation process and the proposed *FuncVul* models.

4.1 Data Generation

Labeling data is critical for training any prediction model, yet identifying vulnerable data often poses significant challenges. In this study, we construct two types of ground truth datasets: code chunks and generic code chunks, derived from function source code and corresponding patch information. Figure 1 illustrates the processes involved in generating these datasets. Detailed descriptions of the ground truth generation for both code chunks and generic code chunks are provided in the following subsections.

In this study, we focus on function code chunks rather than full function code for two key reasons. First, vulnerabilities often exist within just one or two lines of code inside a function, and models trained on entire functions may struggle to pinpoint these specific vulnerable lines that potentially leading to inaccurate predictions. Second, using code chunks reduces the search space and minimizes the number of tokens

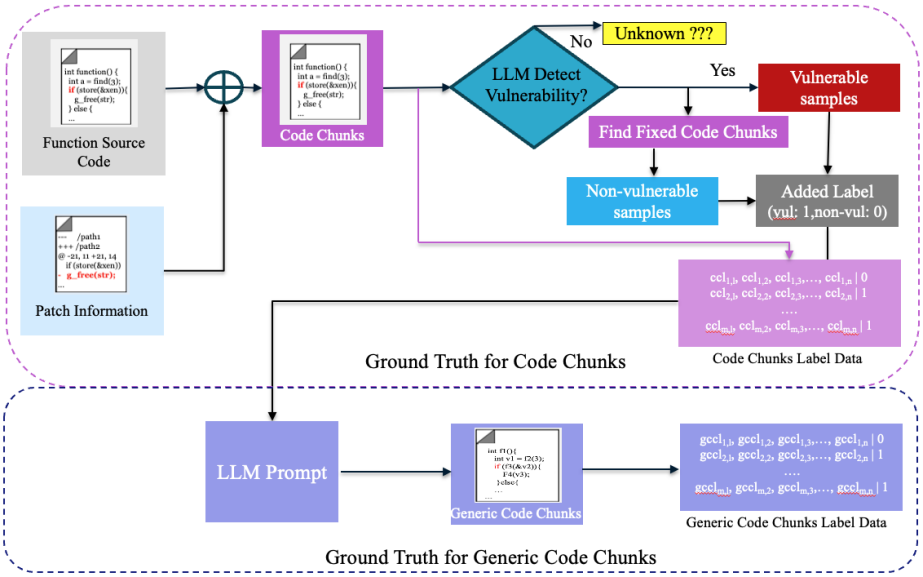


Fig. 1: Code Checks and Generic Code Chunks Label Data Generation.

processed by the tokenizer, enabling the fine-tuned model to more effectively distinguish between vulnerable and non-vulnerable patterns.

In this work, we generate code chunks by leveraging function source code and patch information. Patch information highlights the modifications made to the code, marking added lines with a plus sign (+) and removed lines with a minus sign (-) at the beginning of each line. Additionally, it includes a chunk header that specifies the location and range of the changes, indicating where the modifications begin and the consequences of changes using line numbers.

The algorithm 1 extracts relevant code chunk segments from a function based on patch information. It first parses the patch details to retrieve the chunk header, removed lines, and added lines in line 1. Next, it extracts the starting lines and corresponding modification ranges for both the removed and added lines in line 2. Context parameters are initialized to include three lines before and after the modified region (for three line strategy) in line 3. The algorithm then initialises indices of the removed lines within the function, recording them in a list called modified_index in line 4. If no matches are found, it returns an empty result in line 9. For matched lines, the algorithm determines the bounds of the code chunk using a heuristic for small regions ($\leq$10 lines) to include additional context, or directly uses the minimum and maximum indices for larger regions in lines 11-16. Finally, it extracts and returns the code chunk based on the calculated bounds in line 16 and line 17, respectively.

Generic Code Chunks: Generic code chunk converts code chunks to a generic format. In this work, we have transformed function code into a generic format by renaming functions as F_1, F_2, ..., F_m and variables as v_1, v_2, ..., v_n. The key advantage

Algorithm 1: Find Function Code Chunk (F, P)

```
Data: F: Function source code; P: Patch information.
Result: FC: Extracted function code chunk.
chunk_header, removed_lines, added_lines ← P
removed_start_line, removed_line_range, added_start_line, added_line_range ←
  chunk_header
before_lines, after_lines = 3, 3
modified_index ← {}
for index, line ∈ enumerate(F[removed_start_line : removed_start_line +
  removed_line_range]) do
    if line ∈ removed_lines then
        modified_index.append(removed_start_line + index)
if modified_index == {} then
    return {}
if max(modified_index) - min(modified_index) ≤ 10 then
    start_index ← max(modified_index[0]) - before_lines, 0)
    end_index ←min(max(modified_index[-1]) + after_lines + 1, len(F))
else
    start_index ← min(modified_index)
    end_index ← max(modified_index)
Extract function code chunk FC = F[start_index:end_index]
return FC;
```

lies in mitigating the variations introduced by different developers who often use diverse functions and variable names to achieve the same functionality. The LLM prompt designed to standardize code chunks by converting them into their generic format is presented in Appendix A.1.

In this work, we leverage the Gemini 1.5 Pro [22] LLM model to efficiently transform generic code chunks. A key advantage of utilizing this LLM model is its ability to seamlessly convert code across various programming languages, including C/C++, Java, and Python.

Vulnerable and Non-Vulnerable Samples: Our primary objective is to detect vulnerabilities using function-level code, whether in its original form as code chunks or transformed into generic code chunks. To develop a robust vulnerability detection model, we require a ground truth dataset comprising both vulnerable and non-vulnerable samples. To construct this dataset, we adopt a dual-strategy approach that combines code-based heuristics with predictions from a LLM. This methodology enhances the ability to identify functions with a higher likelihood of containing vulnerabilities, ensuring greater confidence in the dataset's accuracy.

Property 1 ***Code-Based Heuristic (Patch Modification Hypothesis):*** *We hypothesize that functions containing only a single modification within a CVE patch are more likely to contain the vulnerability. This hypothesis stems from the assumption that smaller,*

localized patches often address specific vulnerabilities directly. This property does not guarantee that the code chunks will always be vulnerable, as developers may modify patches to enhance code quality.

Our study consists of clean and localized vulnerability cases from OSV.dev, where our empirical study shows 80.04% (6515 out of 8139) of CVEs have a single Git commit patch. Therefore, we restricted our study to single-patch CVEs, aligning with VFCFinder [4]. Multiple modifications make it unclear which change corresponds to the vulnerability. For dataset reliability, we excluded multifile patch information. Each modified patch contains chunk headers with deleted and added lines between code versions. The before version shows the vulnerable state, while the after version shows the fixed code.

Property 2 ***LLM-Based Heuristic (Vulnerable Line Detection):*** *We utilise a LLM Gemini-1.5 Pro [22] to predict vulnerable lines within code chunks. This model is presented with either (i) the code chunk alone or (ii) the code chunk alongside its corresponding CVE description. This dual input strategy aims to leverage both code structure and vulnerability context for improved ground vulnerability predictions. Appendix A.2 shows the two different prompts that we use in this work to identify vulnerable samples.*

Vulnerable Ground Truth: A code chunk is classified as vulnerable (class label: 1) and included in the ground truth dataset if it satisfies the following criteria:

- **Property 1** must be fulfilled.
- According to **Property 2**, the LLM response for *vul_lines* is not *None*.
- There is at least one overlapping line between the *vul_lines* identified by the LLM and the deleted lines in the patch modification.

If any of the above criteria are not met, the code chunk is labeled as Unknown (see Figure 1).

Non-Vulnerable Ground Truth: After the labeling of vulnerable code chunks, we extract fixed code from the after version using patch modification details and construct non-vulnerable code chunk samples. Additionally, we include random 5 to 10 lines of code from fixed functions in the after version. These samples are classified as non-vulnerable (class label: 0).

Code Chunks and Generic Code Chunks Label Data: Figure 1 illustrates the process of generating labeled data for code chunks and generic code chunks. These chunks are constructed using two types of LLM prompts (detailed in Table 6). Therefore, based on two LLM prompts and code chunks and generic code chunks, we generate four label datasets (Dataset 1, Dataset 2, Dataset 3 and Dataset 4) that shown in dataset section (*c.f.* Section 5.1) in Table 1.

We further created two additional datasets, Dataset 5 and Dataset 6 (*c.f.* Section 5.1) , by providing the full function code to a large language model (LLM) to identify vulnerable lines. If the LLM successfully detects at least one vulnerable line, we apply the N-line code chunking approach to generate positive samples. The same strategy used for generating negative samples in Datasets 1–4 is applied here for consistency.

4.2 Proposed FuncVul Model

Figure 2 provides an overview of the architecture for the proposed FuncVul model, designed to detect function-level vulnerabilities effectively. The process begins with the input data, which consists of either code chunks or generic code chunks. These inputs are preprocessed and split into two subsets: 80% for training and 20% for testing. The training data is then tokenized using the tokenizer from the pre-trained GraphCodeBERT model. This tokenization step transforms the raw code chunks into numerical representations that encode the syntactic and semantic features of the code. Subsequently, the tokenized training data is passed through the pre-trained GraphCodeBERT model, which has been fine-tuned to capture rich features specific to programming languages.

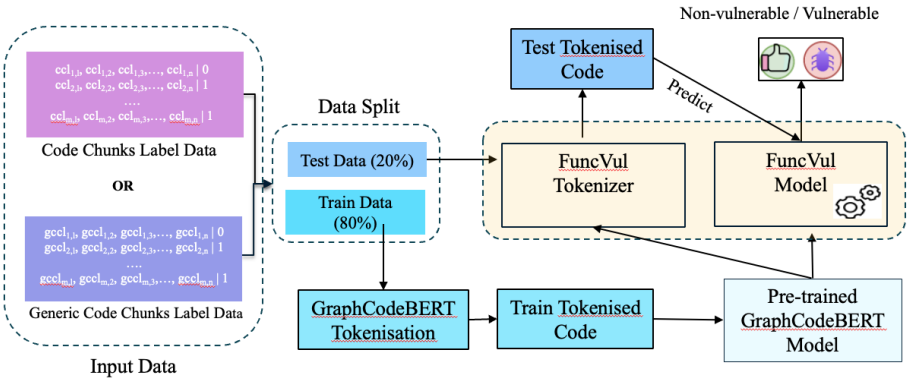


Fig. 2: Proposed code chunk based function vulnerability detection model (FuncVul) architecture.

GraphCodeBERT [7] is a pre-trained model for programming languages that incorporates the semantic structure of code, focusing on data flow rather than abstract syntax trees (AST). Data flow represents variable relationships through a graph, simplifying complexity and enhancing efficiency. The model introduces two structure-aware pre-training tasks: data flow edge prediction to learn code structure representation and variable alignment to bridge source code and data flow representations. Built on a Transformer [24] architecture, GraphCodeBERT extends it with a graph-guided masked attention mechanism, enabling it to effectively integrate code structure and improves code representation learning.

In Figure 2, we illustrate the fine-tuning process of the GraphCodeBERT model using our generated code chunk or generic code chunk data. After fine-tuning, the GraphCodeBERT model builds a new model and tokeniser, which we refer to as the FuncVul model and FuncVul tokeniser, respectively. To evaluate the model's performance on the test data, the test code is first tokenised using the FuncVul tokeniser. The tokenised code is then fed into the FuncVul model, which predicts whether the code chunk is vulnerable or non-vulnerable.

4.3 FuncVul algorithm

Algorithm 2: FuncVul Model (Data)

Data: Data: Code chunks or Generic code chunks data
Result: FuncVul model $\mathcal{M}$ and FuncVul tokeniser $\mathcal{T}$
1 **Training Phase:**
2 Split Data into train_data (80%) and test_data (20%)
3 $\mathcal{M}_{\mathcal{G}}, \mathcal{B}_{\mathcal{G}} \leftarrow$ Load GraphCodeBERT tokeniser and model
4 train_tokenised_code $\leftarrow$ Tokenize train_data using GraphCodeBERT tokenizer $\mathcal{T}_{\mathcal{G}}$(test_data).
5 Set training parameters (e.g., epochs, batch size, logging steps, learning rate).
6 $\mathcal{M}, \mathcal{B} \leftarrow$ Train GraphCodeBERT using train_data $\mathcal{M}_{\mathcal{G}}$(train_tokenised_code) and parameters and generate new model and tokeniser
7 Save the FuncVul model $\mathcal{M}$ and FuncVul tokenizer.
8 **Testing Phase:**
9 test_tokenised_code $\leftarrow$ Tokenize test_data using FuncVul tokenizer $\mathcal{T}$(test_data).
10 predict_label $\leftarrow$ Predict vulnerabilities (vulnerable or non-vulnerable) using FuncVul model $\mathcal{M}$ (test_tokenised_code).
11 Evaluate the model's performance on the test_data label and predict_label.
12 **return** $\mathcal{M}$, $\mathcal{T}$

The algorithm 2 presents the training and testing process for the function-level vulnerability detection model, FuncVul. During the training phase, the data is split into training (80%) and testing (20%) sets (line 2). Next, the GraphCodeBERT tokenizer and model are loaded (line 3), and the training data is tokenized using the GraphCodeBERT tokenizer to create train_tokenised_code (line 4). Training parameters, such as epochs, batch size, and learning rate, are configured (line 5). Using the tokenized training data, the GraphCodeBERT model is fine-tuned to produce the FuncVul model $\mathcal{M}$ and tokenizer $\mathcal{T}$ (line 6). These are saved for future use (line 7).

In the testing phase, the test data is tokenized using the FuncVul tokenizer $\mathcal{T}$ (line 9). The FuncVul model $\mathcal{M}$ then predicts vulnerability labels for the tokenized test data (line 10). Model performance is evaluated by comparing the predicted labels with the ground truth labels from the test data (line 11). Finally, the algorithm returns the FuncVul model and tokenizer (line 12).

5 Experiments

5.1 Experimental Setup

All experiments in this paper were conducted using Python on a MacBook Pro with an Apple M3 processor and 24GB of RAM. For the FuncVul implementation, a batch size of 8 was used with a chunk code embedding vector length of 512. The model was trained for 3 epochs with 50 warmup steps, a weight decay of 0.05, and automatic reloading of the best model at the end.

Datasets: In this research work, we generate six datasets. The dataset generation process is discussed in detail in the corresponding section (*c.f.* Section 4.1). The first four datasets were created by combining patch information (removed lines) with lines detected by the LLM, ensuring that at least one common line is present between the two. In contrast, datasets 5 and 6 were generated solely using code chunks identified by the LLM, without incorporating any removed line information. For identifying vulnerable lines using LLMs, we employ two prompts: one that utilises only the code information and another that incorporates both the code and its description. Table 1 provides details of six datasets, including the code type, and the number of vulnerable and non-vulnerable samples.

Dataset	Prompt	Code Type	Vulnerable Defined By	Vulnerable	Non-vulnerable
1	Code + Description	Code Chunk	LLM + Patch Information	1810 (43.4%)	2357 (56.6%)
2	Code	Code Chunk	LLM + Patch Information	2120 (42.6%)	2851 (57.4%)
3	Code + Description	Generic Code Chunk	LLM + Patch Information	1810 (43.4%)	2357 (56.6%)
4	Code	Generic Code Chunk	LLM + Patch Information	2120 (42.6%)	2851 (57.4%)
5	Code + Description	Code Chunk	LLM	3169 (50%)	3169 (50%)
6	Code	Code Chunk	LLM	6041 (50%)	6041 (50%)

Table 1: Details of various datasets.

Baselines: We compare our proposed FuncVul model with five baselines: **CodeBERT** [5], **CustomVulBERTa** [9], **BERT** [12], **VUDENC** [29] and **LineVul** [6]. These baselines' detailed descriptions are given in Appendix A.3.

Evaluation Metrics: In the prediction models analyses, we applied various evaluation metrics indicating the model performances. Our main goal is to predict code vulnerability. Thus, we evaluate our results using Accuracy, Precision, Recall, F1-score and Matthews Correlation Coefficient (MCC). The details of these evaluation metrics are defined in Appendix A.4.

5.2 Results Analysis

To evaluate the proposed model FuncVul performance, we run a set of experiments to answer our six research questions.

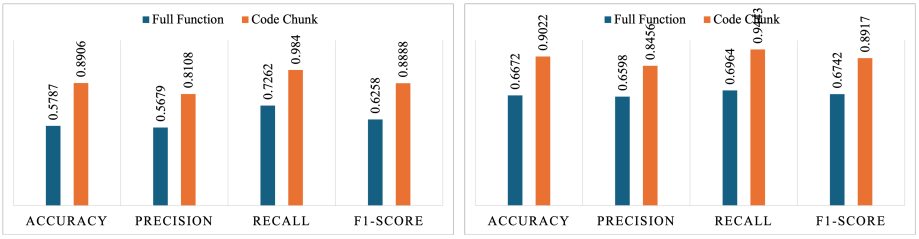
FuncVul Model Performance (RQ1): We compare our proposed FuncVul model against five baseline methods across six benchmark datasets. As shown in Table 2, FuncVul consistently outperforms all other models, ranking first in most evaluation metrics—including F1-Score, Accuracy, Precision, and MCC—across all datasets. Specifically, it achieves the highest score in 25 out of 30 cases and ranks second in two additional cases. The LineVul model also demonstrates strong performance, obtaining

Dataset	Model	Accuracy	Precision	Recall	F1-Score	MCC
1	CodeBERT	0.8707±0.00020	0.7928±0.0216	0.9505±0.0269	0.8641±0.0095 (5)	0.7683±0.0283
	CustomVulBERTa	0.8648±0.0060	0.7974±0.0224	0.9816±0.204	0.8816±0.0091 (3)	0.7898±0.0102
	BERT	0.8812±0.0103	0.8067±0.0168	0.9548±0.0443	0.8739±0.0167 (4)	0.7742±0.0282
	VUDENC	0.8598±0.0111	0.8058±0.0090	0.8560±0.0380	0.8404±0.0225 (6)	0.7166±0.0241
	LineVul	0.8874±0.0074	0.7937±0.0174	0.9802±0.0.0	0.8849±0.0708 (2)	0.7976±0.0118
	FuncVul	**0.8906±0.0042**	**0.8108±0.0136**	**0.9840±0.0206**	**0.8888±0.0055** (1)	**0.9477±0.0025**
2	CodeBERT	0.8950±0.0117	0.8151±0.0315	0.9777±0.0322	0.8882±0.0111 (2)	0.8039±0.0183
	CustomVulBERTa	0.8908±0.0130	0.7975±0.0232	0.9976±0.0053	0.8863±0.0132 (3)	0.8022±0.0200
	BERT	0.8902±0.0119	0.8136±0.0241	0.9650±0.0361	0.8822±0.0126 (5)	0.7919±0.0244
	VUDENC	0.8680±0.0140	0.8463±0.0183	0.8440±0.0301	0.8449±0.0183 (6)	0.7304±0.0295
	LineVul	0.8900±0.0120	0.7951±0.0202	**1.0±0.0**	0.8857±0.0126 (4)	0.8016±0.0193
	FuncVul	**0.9022±0.0157**	**0.8456±0.0212**	0.9443±0.0282	**0.8917±0.0178** (1)	**0.8947±0.0454**
3	CodeBERT	0.8663±0.0176	0.7856±0.0202	0.9512±0.0346	0.8602±0.0205 (4)	0.7545±0.0096
	CustomVulBERTa	0.8675±0.0114	0.7793±0.0210	0.9682±0.0141	0.8634±0.0155 (2)	0.7544±0.0.021
	BERT	0.8054±0.0248	0.7762±0.0143	0.7764±0.0459	0.7758±0.0247 (5)	0.6043±0.0521
	VUDENC	0.7485±0.0199	0.7153±0.0258	0.6981±0.0306	0.7063±0.0244 (6)	0.4867±0.0420
	LineVul	0.8656±0.0121	0.7750±0.0240	**0.9721±0.0121**	0.8622±0.0156 (3)	0.7527±0.0.0192
	FuncVul	**0.8723±0.0114**	**0.7924±0.0245**	0.9544±0.0183	**0.8657±0.0174** (1)	**0.8825±0.0577**
4	CodeBERT	0.8735±0.0141	0.7940±0.0281	0.9526±0.0254	0.8654±0.0140 (2)	0.7602±0.0274
	CustomVulBERTa	0.8684±0.0133	0.7817±0.0206	0.9600±0.0124	0.8616±0.0141 (3)	0.7531±0.0.0239
	BERT	0.80677±0.0158	0.7712±0.0285	0.7784±0.0288	0.7743±0.0198 (5)	0.6058±0.0317
	VUDENC	0.7658±0.0155	0.7238±0.0253	0.7302±0.0401	0.7263±0.0233 (6)	0.5225±0.0321
	LineVul	0.8656±0.0163	0.7759±0.0264	**0.9642± 0.0092**	0.8596±0.0162 (4)	0.7500± 0.0270
	FuncVul	**0.8797±0.0118**	**0.8077±0.0200**	0.9426±0.0182	**0.8698±0.0134** (1)	**0.7982±0.0470**
5	CodeBERT	0.8914±0.0125	0.8914±0.0136	0.9148±0.0329	0.9006±0.0131 (2)	0.8035±0.0253
	CustomVulBERTa	0.8905±0.0147	0.8530±0.0244	0.9446±0.0208	0.8962±0.0130 (3)	0.7860±0.0278
	BERT	0.5897±0.0856	0.7902±0.1950	0.3860±0.3342	0.3997±0.3296 (6)	0.2007±0.01448
	VUDENC	0.8034±0.0135	0.7996±0.0171	0.8097±0.0152	0.8045±0.0145 (5)	0.6064±0.0269
	LineVul	0.8509±0.0199	0.7895±0.0358	**0.9596±0.0306**	0.8655±0.0178 (4)	0.7205±0.0340
	FuncVul	**0.9004±0.0116**	**0.8934±0.0130**	0.9096±0.0154	**0.9013±0.0111** (1)	**0.9556±0.0041**
6	CodeBERT	0.8984±0.0071	0.9007±0.0187	0.9330±0.0207	0.9155±0.0073 (2)	0.8377±0.0137
	CustomVulBERTa	0.8898±0.0170	0.8434±0.0416	**0.9614±0.0269**	0.8975±0.0124 (3)	0.7900±0.0242
	BERT	0.7115±0.0839	0.7069±0.0959	0.7946±0.1116	0.7371±0.0258 (5)	0.4465±0.1162
	VUDENC	0.8290±0.0074	0.8196±0.0161	0.8443±0.0145	0.8316±0.0073 (4)	0.6585±0.0144
	LineVul	0.7951±0.1676	0.6455±0.0.3612	0.7785±0.4355	0.7056±0.3944 (6)	0.7570±0.006
	FuncVul	**0.9184±0.0053**	**0.9056±0.0117**	0.9343±0.0116	**0.9196±0.0057** (1)	**0.9619±0.0031**

Table 2: Comparison of FuncVul and baselines across six datasets, with bold for best scores, underline for second-best, and bracketed numbers indicating F1-score rankings (1 = best, 6 = worst).

the highest Recall in four cases and second-best results in three others. CodeBERT and CustomVulBERTa exhibit competitive results in certain settings, with CodeBERT achieving the second-highest score in 18 cases. CustomVulBERTa achieves seven second-best results and one best-case performance. All experiments are conducted using an 80/20 train-test split, and results are reported as the average of five-fold cross-validation. To determine the best model, we rank F1-scores due to their balanced representation of Precision and Recall. Overall, Table 2 confirms that FuncVul delivers the most consistent and superior performance, effectively answering RQ1.

Code Chunks vs Full Function based Results Analysis (RQ2): Table 3(a) compares the performance of the Full Function and Code Chunk approaches on Dataset 1. The Code Chunk method significantly outperforms the Full Function approach across all metrics. It improves accuracy by 53.9%, precision by 42.8%, recall by 35.5%, and F1-score by 42.0%. We also gets the same kinds of results on Dataset 2 in figure 3(b). In this case our proposed code chunk based model improves accuracy by 35.22%, precision by 28.16%, recall by 35.59% and F1-score by 32.26%. These results highlight that



(a) Dataset 1 (b) Dataset 2

Fig. 3: Comparison between our proposed code chunk based results with full function code based results.

the code chunk approach significantly enhances the model's capability for vulnerability detection than the full-function based approach. These findings effectively address our research question RQ2.

Generic Code Chunks based Results Analysis (RQ3): In this work, we construct datasets based on code chunk and generic code chunk methodologies using the same data. Figure 4(a) presents a comparative analysis between code chunk-based Dataset 1 and generic code chunk-based Dataset 3 results on FuncVul method. The results demonstrate that the code chunk based results consistently outperforms generic code across all evaluation metrics, achieving improvements of 2% in Accuracy, 1.84% in Precision, 2.96% in Recall, and 1.9% in F1-score. Similarly, Figure 4(b) compares Code

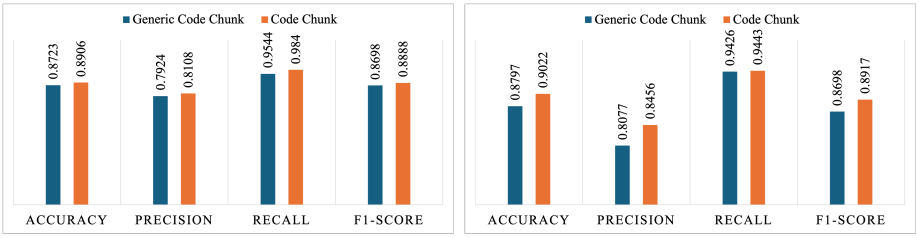


Fig. 4: Comparison between our proposed code chunk based results with generic code chunk based results.

chunk-based Dataset 2 with generic Code Chunk-based Dataset 4, showing improvements of 2.25% in Accuracy, 3.79% in Precision, and 2.19% in F1-score, with a modest 0.17% gain in Recall. These findings underscore the superior effectiveness and adaptability of the Code Chunk approach, validating RQ3.

Effectiveness of Identifying new CVEs and new Project CVEs vulnerabilities (RQ4): Previous results show vulnerability detection outcomes from five-fold cross-validation. However, Figure 1 shows many vulnerable packages remain undetected by LLM Detect, marked as 'Unknown'. To evaluate this unknown data, we created two test datasets. The first contains CVEs absent from training data, while the second includes code from different project IDs than the training data. We trained our proposed model FuncVul using 100% data on Dataset 2. Table 3 shows the two test case data in details.

Case	Type	Vulnerable	Non-Vulnerable
Test Case 1	New CVSs	1245	1753
Test Case 2	New Project ID	179	280

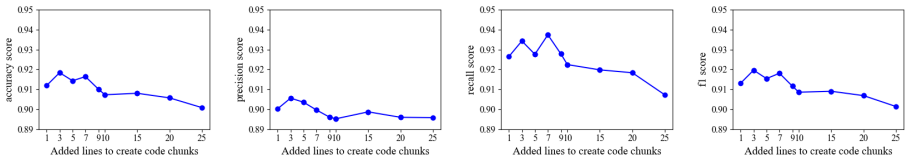
Table 3: New CVEs and new project ID-based test data.

Table 4 shows that the FuncVul model achieves an accuracy of 81.95% in Test Case 1 and 76.69% in Test Case 2. This indicates that the model can correctly identify 81.95% of code chunks that were previously unexplored during dataset 2 construction (labeled as Unknown in Figure 1). The FuncVul model shows strong capability in detecting vulnerabilities, particularly in recall 90.20%. These results support the model's robustness in identifying unknown vulnerabilities, addressing the objectives of RQ4.

Case	Model	Accuracy	Precision	Recall	F1-Score	FP	FN
Test Case 1	FuncVul	0.8195	0.7283	0.9020	0.8059	419	122
Test Case 2	FuncVul	0.7669	0.6552	0.8492	0.7397	80	27

Table 4: New CVEs and new project ID based prediction results for various model on Dataset 2.

Impact of Line Numbers to Create Code Chunks (RQ5): Our primary objective is to generate concise code chunks highlighting vulnerable patterns. We adopted "3-line extended based code chunk", including three lines before and after a detected vulnerable line. To validate this approach, we tested code chunks extended by different line numbers—1, 5, 7, 9, 10, 15, 20, and 25—and assessed their performance with our FuncVul model. Results show the 3-line extended code chunk outperforms other configurations in most metrics. Figure 5 (a) shows highest accuracy for 3-line extended



(a) Accuracy (b) Precision (c) Recall (d) F1 Score

Fig. 5: Impacts of code chunk length on dataset 6.

chunks on Dataset 6, while Figure 5 (b) demonstrates superior precision for this strategy. Although the recall score, shown in Figure 5 (c), is highest for 7-line extended code chunks, the 3-line extended approach ranks second. Figure 5 (d) shows the F1 score, balancing recall and precision, confirming that 3-line chunks provide the best performance. Larger chunks make it difficult to identify vulnerable code effectively, while the 3-line strategy's average chunk length of 6.2 lines provides optimal balance between context and conciseness for detecting vulnerable patterns.

5.3 Detection of Multiple Vulnerabilities Within Function Code (RQ6):

In this work, we use code chunk based data for model built that can split one single function in multiple code chunk. Thus, the proposed model is capable of detecting multiple vulnerabilities within a single function's code. By analyzing smaller, context-rich segments, the model effectively captures various vulnerability patterns, enabling comprehensive detection across different parts of the function.

6 Conclusion

In this paper, we present FuncVul, a novel model for function-level vulnerability detection that leverages function code chunks and the pre-trained GraphCodeBERT model. Unlike existing approaches, FuncVul not only identifies whether a function is vulnerable but also detects the specific number of vulnerable code chunks, significantly reducing the time required by developers or experts to address vulnerabilities. Experimental results demonstrate that FuncVul outperforms baseline models on both code chunk and generic code chunk datasets. Additionally, our analysis reveals that datasets based on three-line code chunks from large language models yield higher accuracy and F1-scores compared to datasets where patch information is derived by removing lines of code. Furthermore, we demonstrate that our dataset can be generalized using large language models, resulting in enhanced model performance.

This study focuses on vulnerability detection strategies, without considering multi-class detection since vulnerabilities have varying risk levels. Future work will extend to multi-class vulnerability detection to address risk variations. Currently, we identify vulnerabilities in C/C++ and Python code, with plans to expand to other programming languages.

References

1. Akuthota, V., Kasula, R., Sumona, S.T., Mohiuddin, M., Reza, M.T., Rahman, M.M.: Vulnerability detection and monitoring using llm. In: 2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE). pp. 309–314. IEEE (2023)
2. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet? IEEE Transactions on Software Engineering **48**(9), 3280–3296 (2021)
3. CVE: Common vulnerabilities and exposures. `https://cve.mitre.org` (2024), accessed: 2024-11-12
4. Dunlap, T., Lin, E., Enck, W., Reaves, B.: Vfcfinder: Pairing security advisories and patches. In: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security. pp. 1128–1142 (2024)
5. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al.: Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)
6. Fu, M., Tantithamthavorn, C.: Linevul: A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 608–620 (2022)
7. Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al.: Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366 (2020)
8. Guo, Y., Patsakis, C., Hu, Q., Tang, Q., Casino, F.: Outside the comfort zone: Analysing llm capabilities in software vulnerability detection. In: European symposium on research in computer security. pp. 271–289. Springer (2024)
9. Hanif, H., Maffeis, S.: Vulberta: Simplified source code pre-training for vulnerability detection. In: 2022 International joint conference on neural networks (IJCNN). pp. 1–8. IEEE (2022)
10. Hin, D., Kan, A., Chen, H., Babar, M.A.: Linevd: Statement-level vulnerability detection using graph neural networks. In: Proceedings of the 19th international conference on mining software repositories. pp. 596–607 (2022)
11. Islam, N.T., Parra, G.D.L.T., Manuel, D., Bou-Harb, E., Najafirad, P.: An unbiased transformer source code learning with semantic vulnerability graph. In: 2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P). pp. 144–159. IEEE (2023)
12. Kenton, J.D.M.W.C., Toutanova, L.K.: Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of naacL-HLT. vol. 1, p. 2. Minneapolis, Minnesota (2019)
13. Li, Y., Wang, S., Nguyen, T.N.: 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. pp. 292–303 (2021)
14. Li, Z., Zou, D., Xu, S., Chen, Z., Zhu, Y., Jin, H.: Vuldeelocator: a deep learning-based fine-grained vulnerability detector. IEEE Transactions on Dependable and Secure Computing **19**(4), 2821–2837 (2021)
15. Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J.: Vulpecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd annual conference on computer security applications. pp. 201–213 (2016)
16. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681 (2018)
17. Liu, Y.: Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692 **364** (2019)

18. Lu, G., Ju, X., Chen, X., Pei, W., Cai, Z.: Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. Journal of Systems and Software **212**, 112031 (2024)
19. Nguyen, V.A., Nguyen, D.Q., Nguyen, V., Le, T., Tran, Q.H., Phung, D.: Regvd: Revisiting graph neural networks for vulnerability detection. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. pp. 178–182 (2022)
20. NVD: National vulnerability database (2024), `https://nvd.nist.gov`, accessed on November 12, 2024
21. Sonnekalb, T.: Machine-learning supported vulnerability detection in source code. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1180–1183 (2019)
22. Team, G., Georgiev, P., Lei, V.I., Burnell, R., Bai, L., Gulati, A., Tanzer, G., Vincent, D., Pan, Z., Wang, S., et al.: Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv preprint arXiv:2403.05530 (2024)
23. Tran, H.C., Tran, A.D., Le, K.H.: Detectvul: A statement-level code vulnerability detection for python. Future Generation Computer Systems **163**, 107504 (2025)
24. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. nips'17. In: Proceedings of the 31st International Conference on Neural Information Processing Systems December. pp. 6000–6010 (2017)
25. Vo, H.D., Nguyen, S.: Can an old fashioned feature extraction and a light-weight model improve vulnerability type identification performance? Information and Software Technology **164**, 107304 (2023)
26. Wang, H., Ye, G., Tang, Z., Tan, S.H., Huang, S., Fang, D., Feng, Y., Bian, L., Wang, Z.: Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Transactions on Information Forensics and Security **16**, 1943–1958 (2020)
27. Wang, X., Chen, K., Kang, T., Ouyang, J.: A dynamic coarse grain discrete element method for gas-solid fluidized beds by considering particle-group crushing and polymerization. Applied Sciences **10**(6), 1943 (2020)
28. Wang, X., Hu, R., Gao, C., Wen, X.C., Chen, Y., Liao, Q.: Reposvul: A repository-level high-quality vulnerability dataset. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings. pp. 472–483 (2024)
29. Wartschinski, L., Noller, Y., Vogel, T., Kehrer, T., Grunske, L.: Vudenc: vulnerability detection with deep learning on a natural codebase for python. Information and Software Technology **144**, 106809 (2022)
30. Wei, H., Lin, G., Li, L., Jia, H.: A context-aware neural embedding for function-level vulnerability detection. Algorithms **14**(11), 335 (2021)
31. Wen, X.C., Chen, Y., Gao, C., Zhang, H., Zhang, J.M., Liao, Q.: Vulnerability detection with graph simplification and enhanced graph representation learning. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp. 2275–2286. IEEE (2023)
32. Wu, Y., Zou, D., Dou, S., Yang, W., Xu, D., Jin, H.: Vulcnn: An image-inspired scalable vulnerability detection system. In: Proceedings of the 44th International Conference on Software Engineering. pp. 2365–2376 (2022)
33. Xia, C.S., Wei, Y., Zhang, L.: Automated program repair in the era of large pre-trained language models. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp. 1482–1494. IEEE (2023)
34. Yuan, X., Lin, G., Mei, H., Tai, Y., Zhang, J.: Software vulnerable functions discovery based on code composite feature. Journal of Information Security and Applications **81**, 103718 (2024)

35. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in neural information processing systems **32** (2019)

A Appendix

A.1 Generic Code Conversion LLM Prompt

Generic code chunks converts code chunks in generic format. Here, we use the following LLM to convert the generic code chunk as described in Table 5. Table 5 also shows the one example of code chunk and converted generic code chunk based on the proposed LLM generic prompt.

Description of Generic Prompt
Here is the function code chunk: $\{code\_chunk\}$ Please convert the code chunk by renaming functions to $F_1, F_2, ..., F_N$ and variables to $v_1, v_2, ..., v_n$. Return the converted code in a variable named *generic_code*.
Example
Code Chunk

goto trunc;
if (length < alen)
goto trunc;
if (!bgp_attr_print(ndo, atype, p, alen))
goto trunc;
p += alen;
len -= alen;

Generic Code Chunk

goto F1;
if (v1 < v2)
goto F1;
if (!F2(v3, v4, v5, v2))
goto F1;
v5 += v2;
v6 -= v2;

Table 5: Prompt and Example for Transforming Code Chunks into Generic Code Chunk

A.2 Vulnerable Samples detection LLM Prompts

In this paper, we construct six datasets using two distinct LLM prompts. Figure 6 provides a detailed overview of these prompts—one utilizing only the code and the other combining a description with the code.

Prompt Type	Input Context
Code Only	Given the following function code: {code}
Code + Description	Given the following function code: {code} And the associated CVE description: {desc}

Task: Extract the following information:

1. Identify the lines of code that contain vulnerabilities. Return these lines in a list of string named as *line_code*. If no vulnerable lines are found, return ['None']. Ensure the list is formatted with items separated by commas and enclosed in square brackets.
2. Determine the line numbers of vulnerable code. Return these line numbers in a list of integer named as *vul_lines*. If no such lines exist, return ['None'].
3. List the affected vulnerability categories. Return these in a list of string named as *vul_category*. If no categories are affected, return ['None'].

Please provide the output in three keys as dictionary format: *line_code*, *vul_lines*, and *vul_category*. Do not need an explanation.

Table 6: LLM prompts for detecting vulnerable samples with different input settings.

A.3 Details of Baselines

We compare our proposed FuncVul model with five baselines that are as follows.

- **CodeBERT** [5]: CodeBERT is a pre-trained model for understanding and generating both natural language and programming code.
- **CustomVulBERTa**: CustomVulBERTa is a fine-tuned version of VulBERTa [9], a RoBERTa-based model pre-trained on real-world C/C++ code, adapted to detect security vulnerabilities and used as a baseline in our experiments.
- **BERT**: BERT [12] is a pretrained deep bidirectional transformer model that uses masked language modeling to capture context from both directions, distinguishing it from unidirectional or shallow concatenation-based approaches.
- **VUDENC**: VUDENC [29] is a deep learning tool designed to detect vulnerabilities in real-world Python code. It uses a word2vec model to generate vector representations of semantically similar code tokens and employs LSTM networks to classify vulnerable code sequences.
- **LineVul**: LineVul [6] is a Transformer-based fine-grained line-level vulnerability prediction model.

A.4 Details of Evaluation Metrics

To evaluate the performance of our proposed model, we employ widely used prediction evaluation metrics, as outlined below.

- **Accuracy:** It measures the overall correctness of a model in predicting both code vulnerabilities and non-vulnerabilities. $Accuracy = \frac{TP+TN}{TP+FP+TN+FN}$
- **Precision:** It measures the proportion of true code vulnerabilities to the total number of code vulnerabilities that have been predicted as vulnerabilities by the model: $Precision = \frac{TP}{TP+FP}$

- **Recall:** It measures the proportion of true vulnerabilities detected by a model to the total number of code vulnerabilities in the dataset: $Recall = \frac{TP}{TP+FN}$
- **F1-Score:** It is the harmonic mean of precision and recall: $F1\text{-}Score = \frac{2*Precision*Recall}{Precision+Recall}$
- **Matthews Correlation Coefficient (MCC):** MCC is a robust metric that reflects balanced performance across all confusion matrix categories and is particularly effective for evaluating models on imbalanced datasets. $MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$