CTX-Coder: Cross-Attention Architectures Empower LLMs for Long-Context Vulnerability Detection

Jujie Wang[1], Kangfeng Zheng[1*], Bin Wu[1], Chunhua Wu[1], Yulin Yao[1], Jiaqi Gao[1], Minjiao Yang[1],

[1]Beijing University of Posts and Telecommunications, Beijing, China
{wangjvjie, kfzheng}@bupt.edu.cn

Abstract

Software vulnerabilities have increased sharply, underscoring the growing urgency for effective detection methods. Although large language model (LLM) based methods have shown promise in this task, current state-of-the-art LLM approaches struggle with functions that have long contexts. In this paper, we propose CTX-Coder, a context-enhanced vulnerability detection framework that enables LLMs to selectively focus on relevant contextual functions. To achieve this, we represent the contextual functions as embeddings and integrate them with the target code via cross-attention, thereby enhancing the model's ability to capture contextual information. Furthermore, to equip the model with the ability to recognize these embedding features, we propose a two-stage pretraining pipeline. We also introduce a new dataset, CTX-VUL, which addresses the limitations of existing datasets that either lack contextual information for vulnerable functions or are not publicly available. Extensive experiments demonstrate that CTX-Coder (10B) significantly outperforms baseline models with even larger parameters, such as Qwen2.5-14B and SecGPT. As the input code length increases, CTX-Coder's F1 score drops by only 5.01%, while other models degrade by 25% to 41.5%, showing strong robustness to long-context scenarios and the effectiveness of our design.

Code — https://github.com/wangjvjie/CTX-Coder

Introduction

Software vulnerabilities often arise from flawed security control designs or developer errors during the implementation of the software specification, particularly in large and complex systems where their emergence is almost inevitable (Mirsky et al. 2023; D'Ambros, Bacchelli, and Lanza 2010). In 2024, 28,392 vulnerabilities were publicly reported in the Common Vulnerabilities and Exposures (CVE) database (MITRE 2024), nearly double the 18352 vulnerabilities reported in 2020 (MITRE 2020). This indicates the increasing severity of software security risks in increasingly complex digital environments, highlighting the urgent need for effective vulnerability detection methods. However, finding vulnerabilities in the early stages can significantly reduce the cost and impact of security breaches (Adebiyi, Arreymbi, and Imafidon 2013). Hence, source code vulnerability detection has been a focal point of academic research for decades.

Typically, code is transformed into structural representations such as Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and Data Flow Graphs (DFG) (Zhou et al. 2019; Chakraborty et al. 2022; Wen et al. 2024a), which are then input into deep learning models such as Convolutional Neural Networks (CNN) (Chen et al. 2024; Filus et al. 2021; Hanif and Maffeis 2022), Recurrent Neural Network (RNN) (Guo et al. 2022; Huang et al. 2024), and Graph Neural Network (GNN) (Zhou et al. 2019; Allamanis, Jackson-Flux, and Brockschmidt 2021; Cao et al. 2022; Dinella et al. 2020; Wu et al. 2023). Although these approaches achieve impressive performance, their real-world applicability remains limited. They do not provide detailed information such as the root cause of the vulnerability or an explanation of it (Mirsky et al. 2023). This makes it difficult for developers without security expertise to effectively resolve the issue.

However, large language models (LLMs) can address these challenges thanks to their pretraining on large-scale code corpora. It is capable of understanding vulnerabilities and generating explanations. Despite the strong potential of LLMs in code understanding and generation, they still face the following challenges:

(1) Insufficient understanding of long-context information: In most cases, a vulnerability may involve multiple functions, ranging from the root cause to the manifestation point. For functions that are within three steps of a call chain, the total Lines of Code (LOC) may reach as high as 600 to 900 lines. As shown in Figure 1 the detection accuracy declined markedly (Li et al. 2025). Once the code exceeds 300 lines, the accuracy drops below 5%. These findings indicate that the scale of real-world code contexts far exceeds the effective processing range for current LLM-based methods.

(2) Lack of contextual information in current vulnerability detection datasets: Existing datasets such as QEMU (Zhou et al. 2019), BigVul (Fan et al. 2020), and PrimeVul (Ding et al. 2024) primarily focus on the classification of individual functions, lacking the incorporation of broader contextual or inter-procedural information. Although VulEval (Wen et al. 2024b) introduces context-aware vulnerability detection, the dataset is not publicly available.

*Kangfeng Zheng is the corresponding author.

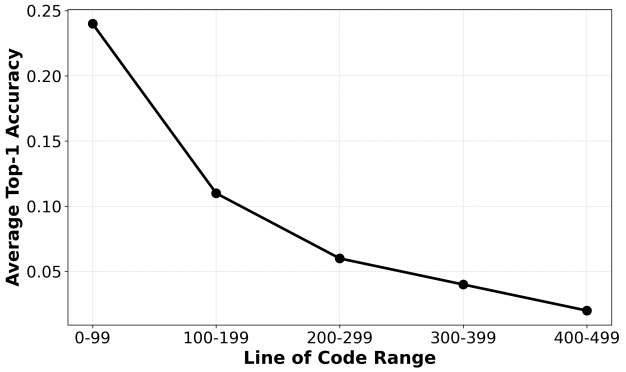

Figure 1: The relationship between Average Top-1 Accuracy and Line of Code Range.

Therefore, there remains an urgent need for an open dataset that explicitly includes contextual information.

To address these challenges, we propose a context-enhanced vulnerability detection framework, CTX-Coder, that enables LLMs to selectively focus on relevant contextual functions (Figure 2). To accomplish this, we embed contextual functions into vectors using Llama 3.1 (Grattafiori et al. 2024) and take the last hidden state as the final representation. These embeddings then interact with the frozen Llama 3.1 model through cross-attention layers. This architecture allows the LLM to make full use of relevant contextual functions and, at the same time, focus adequate attention on the target function under analysis. In order to equip CTX-Coder with the ability to understand and utilize these contextual embeddings, we propose a two-stage pretraining pipeline: unimodal pretraining and multimodal pretraining. In the unimodal stage, individual functions are embedded, and CTX-Coder is trained to reconstruct the original code from the embeddings. In the multimodal stage, multiple related functions are embedded together, and the model is tasked with reconstructing a specific target function based on the given function name. This pretraining pipeline enables CTX-Coder to learn the semantics of function embeddings and selectively focus on relevant contextual functions during the vulnerability detection process.

Moreover, to obtain contextual information for vulnerabilities, we introduce the CTX-Vul dataset. For each vulnerable function, we obtain its GitHub URL and download the corresponding project locally. Then, we trace back to the relevant CVE version and use Doxygen to generate the call and in-call graphs of the target function. As a result, we built a new vulnerability detection dataset with over 500 open-source projects, 2652 CVEs, and 5781 unique vulnerable functions. To support the two-stage pretraining pipeline, we collect 146 open-source C/C++ projects and extract 249,186 call graphs using Doxygen, resulting in 371,682 unique functions. The main contributions are as follows:

- **A context-enhanced framework (CTX-Coder):** We propose a context-enhanced vulnerability detection framework that enables LLMs to selectively focus on relevant contextual functions, thereby enhancing vulnerability detection in long-context scenarios. In addition, a two-stage pretraining pipeline is introduced to improve the model's understanding of function embeddings, which further enhances the model's performance.
- **An extended-context vulnerability dataset (CTX-Vul)** We release CTX-Vul, a new dataset for vulnerability detection that includes rich contextual information across functions. It covers 6 common CWE types and has an average function context length of over 400 lines of code, which is significantly longer than existing datasets such as Devign and Reveal, where context lengths typically range from 32 to 100 lines. Each vulnerable function in CTX-Vul is accompanied by 5 to 10 semantically related functions to provide realistic contextual cues.
- **Experimental Evaluation:** The experimental results demonstrate that our model significantly outperforms baseline methods. Notably, CTX-Coder (10B) achieves an accuracy of 87.06% and 78.10% F1 score, while Qwen 2.5 (14B) is only 81.26% and 68.40%. Furthermore, on the harder CTX-Vul (Medium) subset, Qwen 2.5 suffers a 24% performance drop, while our model drops only 5%, showing stronger robustness to longer contexts.

Related Works

Approaches Prior to LLMs

Early studies used CNN or the RNN model to detect vulnerabilities in source code. They usually focus on the linear representation of the dataflow. However, this prevents the model from learning and utilizing various contexts and semantics (Chen et al. 2024; Filus et al. 2021; Hanif and Maffeis 2022; Guo et al. 2022; Huang et al. 2024; Russell et al. 2018). To overcome these issues, further studies (Zhou et al. 2019; Allamanis, Jackson-Flux, and Brockschmidt 2021; Cao et al. 2022; Dinella et al. 2020; Wu et al. 2023) transform code into structural representations like AST, CFG, and use GNN to capture contexts information.

Although the above methods show extraordinary performance, they still struggle to help developers find vulnerabilities. They provide only a binary classification result for a function without further explanation. Therefore, it is difficult for developers to fix vulnerabilities in large code segments.

LLM Based Methods

LLM can easily generate explanations for vulnerabilities, helping developers identify and fix them more efficiently. Ding et al. (Ding et al. 2024) and Ullah et al. (Ullah et al. 2024) conducted extensive experiments on PrimeVul using various LLMs and found that, despite finetuning and reasoning-enhancement strategies, even the most advanced models perform only marginally better than random guessing in vulnerability detection tasks. Several efforts have been made to enhance LLM's ability to detect vulnerabilities.

Recent research has explored several strategies to enhance LLM vulnerability detection capabilities, such as combining LLM with SAST tools (Li, Dutta, and Naik 2025; Li

Name	Model	Detection Level		Method	Language	Long Context Enhancing	Trainable
		I-F	M-F				
Nong et al. (Nong et al. 2024)	GPT-3.5	✓		COT	C/C++	✗	✗
Li et al. (Li et al. 2025)	Llama 3.1 (8b)	✓		Attention Calculation	Solidity	✓	✗
Appatch (Nong et al. 2025)	GPT-4		✓	RAG	C/C++	✗	✗
IRIS (Li, Dutta, and Naik 2025)	GPT-4		✓	SAST	Java	✗	✗
GRACE (Lu et al. 2024)	GPT-4	✓		Prompt Engineering	Solidity	✗	✗
Lprotector (Sheng et al. 2024)	GPT-4O	✓		RAG	C/C++	✗	✗
RealVul (Cao, Liao, and Shang 2024)	StarCoder2 (7b)	✓		SFT	PHP	✗	✓
Ours	**CTX-Coder (10b)**		✓	SFT	C/C++	✓	✓

Table 1: Comparison of LLM-based detection methods. I-F means Individual Function, M-F means Multiple Function.

et al. 2024), attention calculation (Li et al. 2025), chain-of-thought prompting (COT) (Nong et al. 2024), and retrieval-augmented generation (RAG) (Nong et al. 2025; Liu et al. 2023; Du et al. 2024; Sheng et al. 2024). However, these methods are primarily based on the inherent abilities of the LLM backbone, relying on zero-shot or in-context learning (ICL) rather than model training. As a result, their detection performance drops significantly when using smaller models (e.g., 8B parameters). Notably, Cao et al. (Cao, Liao, and Shang 2024) show that finetuning enables smaller models such as StarCoder-7B to outperform even much larger models like GPT-4O.

Table 1 provides a comparison of recent LLM-based approaches. Unlike prior work that either performs zero-shot inference or uses in-context learning, our method combines multi-function contextual input with supervised finetuning and a long-context-enhanced architecture, enabling superior performance on real-world codebases.

Vulnerability Detection Datasets

Existing vulnerability detection datasets can be generally grouped by their granularity: function-level, file-level and repository-level. Function-level datasets such as Reveal (Chakraborty et al. 2022), Devign (Zhou et al. 2019), and PrimeVul (Ding et al. 2024) construct labeled data by mining vulnerability-fixing commits and extracting affected functions. For file-level datasets, CrossVul (Nikitopoulos et al. 2021) constructs the dataset spanning 40 programming languages and 1,675 projects, while it only provides file-level source code. VulEval (Wen et al. 2024b) provides repository-level inter-function dependencies for contextual analysis, but does not release it publicly.

The Proposed Method

In this section, we detail the architecture of CTX-Coder. As shown in Figure 2, we build a large-scale vulnerability dataset enriched with contextual information, derived from more than 500 open-source C/C++ projects. To enable CTX-Coder to effectively understand and leverage such contextual information, we design a two-stage pretraining pipeline. During inference, the contextual information is integrated with the LLM through cross-attention layers, ultimately allowing the model to predict the corresponding CWE type for each vulnerable function.

Contextual Data Collection

Following the previous work (Wang et al. 2023), the raw data used to build CTX-Vul consists of a vast collection of CVE entries from the Morefix (Akhoundali et al. 2024). Specifically, for each vulnerable function, we identify the parent commit of its earliest fix-related commit. We then use the command "git checkout <commit_hash>" to revert the corresponding project to the vulnerable version.

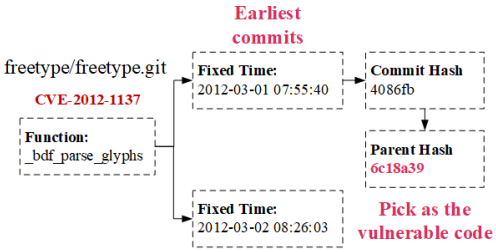

Figure 3: We select the parent hash of the earliest fix commit of a vulnerable function in a CVE as the vulnerable version.

Next, we employ Doxygen to extract function call graphs. Although originally intended for automated documentation, Doxygen is capable of generating call graphs without requiring compilation of the C/C++ codebase, offering a lightweight and convenient solution for our analysis. We enable call graph generation by configuring the Doxyfile, which instructs Doxygen to output ".dot" files that describe the caller-callee relationships between functions. To improve efficiency for large-scale projects (e.g., Linux), where full call graph generation is time-consuming, we restrict the analysis scope. For projects exceeding 100 MB, we generate the call graph using only the directory containing the target function and its two parent directories as the root context, rather than the entire project. It takes a week to generate the data. After analyzing 533 C/C++ projects and performing data filtering, we ultimately extracted 6,310 unique vulnerable functions. For non-vulnerable samples, we randomly selected functions from the remaining functions in the latest versions of the projects.

Dataset Validation: To further ensure data quality, we randomly sampled 300 positive and 300 negative examples for manual verification. Five PhD students in cybersecurity

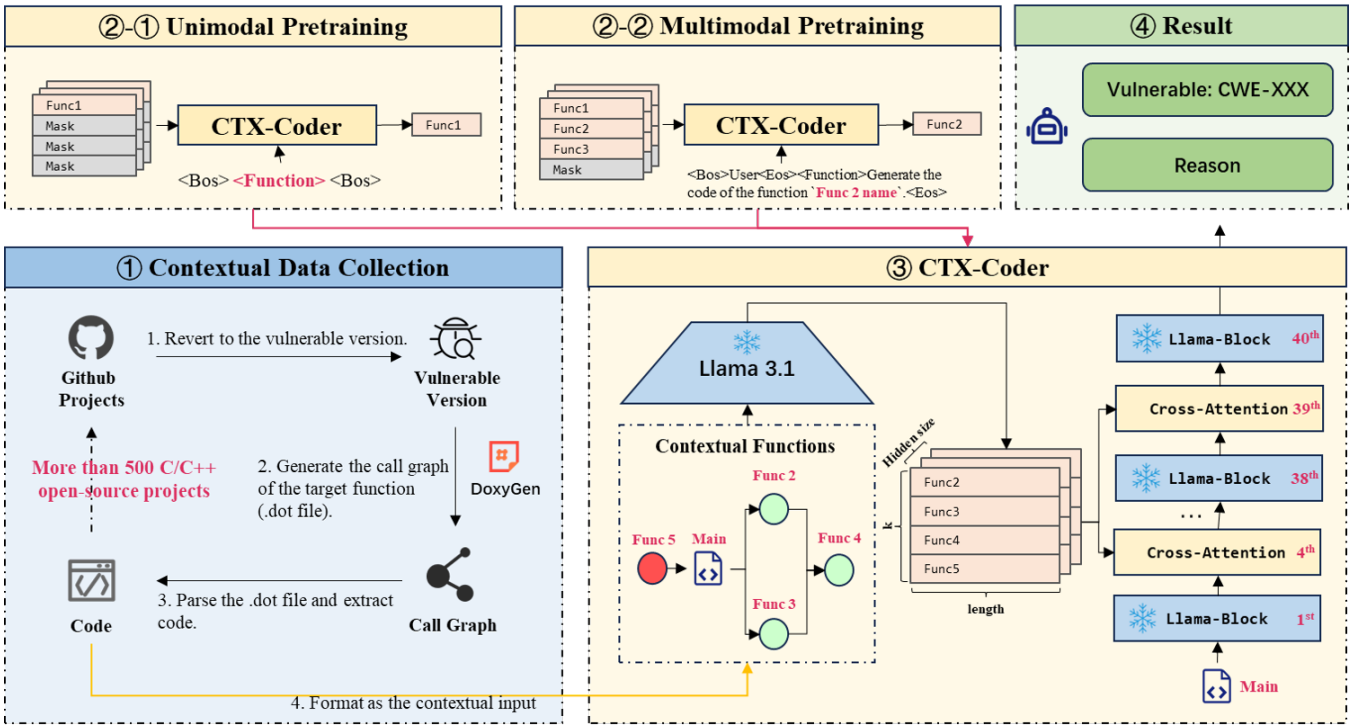

Figure 2: The overview of CTX-Coder. The "Main" function denotes the target function, which serves as the textual input. The Func 2 to Func 5 are the contextual functions of "Main". They are embedded and serve as the input of cross-attention layers.

independently reviewed each sample, and any disputed cases were resolved through group discussion. In total, 548 samples were unanimously considered to be correctly labeled, corresponding to a labeling accuracy of 91.3%. As a result, we obtained 10,065 non-vulnerable samples and 5,781 vulnerable samples. The statistical analysis is shown in Table 2.

CWE ID	Count
CWE-476: NULL Pointer Dereference	690
CWE-787: Out of bounds Write	2600
CWE-125: Out of bounds Read	1016
CWE-190: Integer Overflow or Wraparound	789
CWE-416: Use After Free	628
CWE-415: Double Free	58
Total Vulnerable	5781
Benign	10065

Table 2: The statistic of CTX-Vul.

Table 3 summarizes the distribution of total lines of code (LOC) on CTX-Vul dataset. Most categories have a median LOC between 100 and 200, suggesting that the typical code context length is moderate. Nonetheless, there is considerable variability within each category: the upper quartiles (Q3) often exceed 250 to 400 lines, the Mean LOC are averaged to 250-400, and the maximum LOC can reach several thousand lines, with some cases surpassing 6000. This poses a significant challenge to the model's long-context understanding capabilities.

To facilitate a more structured evaluation of LLM performance across varying context lengths, we partition the test set into three bins based on total LOC: 0–149 (CTX-Vul-easy), 150–299 (CTX-Vul-medium), and 300+ (CTX-Vul-hard). This stratification helps analyze the model's robustness under increasing code complexity and length.

CWE Type	Mean LOC	Median LOC	Q1	Q3	Max
Benign	223.07	114.0	45.00	257.0	4841
CWE-476	243.53	119.0	52.00	267.0	4475
CWE-787	252.26	100.0	32.00	275.5	4475
CWE-190	320.34	207.0	93.25	365.5	2934
CWE-125	404.97	156.0	79.00	430.0	6605
CWE-416	163.77	92.0	34.50	208.0	1309
CWE-415	426.62	305.0	194.00	472.0	1999

Table 3: LOC Summary Statistics on CTX-Vul.

Backbone Network

The backbone of CTX-Coder is built upon Llama 3.1, into which we integrate 8 cross-attention layers by inserting one after every four transformer layers. All the cross-attention layers are trained from scratch, while the parameters of the Llama decoder layers are kept frozen. The model takes two inputs: relative function embeddings H_r and a textual input, typically the target function F_t.

For a target function F_t, and its relative functions $\{F_{ri}\}$ are obtained via a breadth-first search (BFS) traversal, where

i indicates the order in which the functions are discovered. $\{F_{ri}\}$ are then embedded into vector representations $\{h_{ri} \in \mathbb{R}^{l_i \times d}\}, i \in (1, k)$ via Llama 3.1 model (the last hidden state), where l_i denotes the length of the function F_{ri}, d is the hidden size of Llama 3.1. Then we align them into the same length l and stacked as a tensor along the first dimension, which is $H_r = [h_{r1}, h_{r2}, \ldots, h_{rk}] \in \mathbb{R}^{k \times l \times d}$. It is worth noting that if only textual input is given, the cross-attention layers will not be activated, and thus the model can be used as a standard Llama 3.1 model. Because of this architecture, together with the fact that the original Llama layers are kept frozen, we can use CTX-Coder directly for embedding extraction without requiring a separate Llama model.

Unlike common approaches that rely on the `[CLS]` token to represent the embedding of a function, we retain the full token-level representation for each related function. This design enables the model to capture fine-grained contextual semantics distributed across multiple code units. Additionally, we leverage the last hidden state of Llama 3.1 to ensure architectural consistency with the downstream backbone and to take advantage of its pretrained token-level semantics.

Unimodal and Multimodal pretraining Stages

To help CTX-Coder understand the contextual function embeddings H_r, we design a two-stage pretraining pipeline. To support this, we collect 146 open-source C/C++ projects and extract 249,186 call graphs using Doxygen, resulting in 371,682 unique functions.

For unimodal pretraining, we set $k = 1$, such that the contextual input reduces to a single related function. Consequently, $H_r = [h_{r1}] \in \mathbb{R}^{k \times l \times d}$. The model is then tasked with generating the complete code of the target function in an autoregressive manner. The textual input of CTX-Coder is `<Bot><Function><Bot>`, where `<Function>` token is a special token that represents the contextual input.

In multimodal pretraining, all functions within the call graph are retained, and the maximum value of k is set to 10. We use a special template to guide CTX-Coder to complete the specific function. The textual input is as follows. This strategy equips CTX-Coder with a strong contextual understanding of functions. To avoid data leakage, we carefully filtered the pretraining data to ensure that no samples overlap with the CTX-Vul dataset.

Prompt For multimodal pretraining:

<Bot><User><Function>Generate the code of **function name**<Eot><Assistant>...

Experiments

We present empirical results to evaluate the effectiveness of CTX-Coder. We first describe the outstanding performance of long-context vulnerability detection in EXP1. Next, we conduct an ablation study to assess the contributions of each model component (EXP2). Finally, we evaluate CTX-Coder's ability to recognize contextual semantics through code documentation generation (EXP3), and utilize contextual embeddings for cross-file code completion (EXP4).

EXP1: Comparison with Baseline Models under Varying Code Lengths

We split the CTX-Vul dataset into 80% for training and 20% for testing. We compare CTX-Coder with the instructed version of five open-source LLMs: Llama 3.1 (8B), Qwen 2.5 (7B & 14B), StarCoder2 (7B), and SecGPT (14B). For each, we perform full-parameter finetuning on the CTX-Vul training set using identical hyperparameters with a learning rate of 1e-5 for 5 epochs using a batch size of 8 on two NVIDIA A800 GPUs. CTX-Coder only trains the cross-attention layer and the QKV layers, keeping most of the LLM backbone frozen. Among recent LLM-based methods, many rely on closed-source or API-only models (e.g., GPT-4) and are either not suitable for supervised training (Li, Dutta, and Naik 2025; Lu et al. 2024; Li et al. 2025) or not designed for C/C++ (Cao, Liao, and Shang 2024). To represent this category, we include two representative methods: Grace (Lu et al. 2024) and LOVA (Li et al. 2025), both of which operate in a zero-shot or in-context inference setting and do not support task-specific finetuning. A prediction is considered a true positive (TP) only if the predicted CWE-ID is correct; otherwise, it is not counted as a TP. Besides, we also compared with four state-of-the-art methods (Zhou et al. 2019; Chakraborty et al. 2022; Li et al. 2022; Wen et al. 2024a) and trained them until the model is converge. All baseline methods are provided with the contextual information. We make our best effort to reproduce all baselines based on their official open-source implementations or descriptions provided in the original papers.

As shown in Table 4, CTX-Coder consistently outperforms all baseline models, including those with larger parameter sizes such as SecGPT (14B) and Qwen 2.5 (14B), in all LOC settings in both accuracy and F1 score. Specifically, CTX-Coder achieves an F1 score of 0.8440 on the easy setting, which remains as high as 0.5947 even on the hard setting. Besides, LLMs perform much better than the existing models on CTX-Vul. This trend suggests that pretraining on large-scale code corpora provides a strong foundation for vulnerability reasoning. Furthermore, we observe that non-trainable methods such as LOVA and Grace perform significantly worse than trainable methods across all difficulty levels, indicating the importance of finetuning.

We further analyze the decline rate between CTX-Vul-easy (LOC under 150), CTX-Vul-medium (LOC between 150 and 300), and CTX-Vul-hard (LOC over 300). We observe that even the Qwen 2.5 (14B) model exhibits a 24.0% decrease in the F1 score when moving from the easy to the medium setting, while most 7B models suffer an even larger drop—exceeding 40%. However, CTX-Coder shows only a decrease of 5.01% under the same conditions. When comparing the easy and hard settings, most 7B models see their F1 scores drop by more than 50%. In contrast, CTX-Coder's F1 score decreases by only 29.5%, which is comparable to the performance of Qwen 2.5 (14B), despite CTX-Coder having only 10B parameters.

Overall, these results demonstrate that CTX-Coder

Method	Able Training	CTX-Vul (Total)		CTX-Vul (Easy)		CTX-Vul (Medium)		CTX-Vul (Hard)	
		Accuracy	F1 Score	Accuracy	F1 Score	Accuracy	F1 Score	Accuracy	F1 Score
LOVA (Llama 3.1 8B)	✗	0.4544	0.1542	0.5641	0.2384	0.3411	0.1102	0.2787	0.0762
Grace (GPT4)	✗	0.4191	0.2287	0.4138	0.1516	0.5517	0.3158	0.3103	0.2006
Devign	✓	0.4867	0.2892	0.6036	0.3557	0.3564	0.2472 ↓30.5%	0.3086	0.1581 ↓55.6%
SySeVR	✓	0.4678	0.2643	0.5783	0.3157	0.3401	0.1957 ↓37.8%	0.2954	0.1491 ↓52.6%
Reveal	✓	0.5371	0.3231	0.6331	0.3691	0.4274	0.2493 ↓32.4%	0.3542	0.1988 ↓46.1%
MAGNET	✓	0.5829	0.4195	0.6812	0.4980	0.4843	0.3557 ↓28.6%	0.4117	0.2683 ↓46.1%
StarCoder2 (7B)	✓	0.7631	0.5457	0.8109	0.6654	0.7642	0.3889 ↓41.5%	0.6398	0.2776 ↓58.2%
Llama 3.1 (8B)	✓	0.7539	0.5791	0.8203	0.7077	0.7366	0.4196 ↓40.7%	0.6003	0.3583 ↓49.3%
Qwen 2.5 (7B)	✓	0.7652	0.6214	0.8196	0.7495	0.7525	0.4580 ↓38.8%	0.6381	0.3597 ↓52.0%
SecGPT (14B)	✓	0.7987	0.6371	0.8502	0.7543	0.7871	0.5345 ↓29.1%	0.6780	0.4971 ↓34.1%
Qwen 2.5 (14B)	✓	0.8126	0.6840	0.8563	0.7679	0.8009	0.5833 ↓24.0%	0.7119	0.5446 **↓29.0%**
Ours (10B)	✓	**0.8706**	**0.7810**	**0.9033**	**0.8440**	**0.9028**	**0.8017** **↓5.01%**	**0.7572**	**0.5947** ↓29.5%

Table 4: The baseline comparison results on CTX-Vul. All results are averaged over five runs. The values in $\downarrow xx\%$ indicate the relative decrease compared to the corresponding CTX-Vul-easy results. Non-trainable methods perform significantly worse than trainable methods.

not only achieves state-of-the-art performance, but also maintains exceptional robustness to increasing code length—even with fewer parameters than some of the strongest baseline models. This highlights the effectiveness of our approach for long-context vulnerability detection.

EXP2: Ablation Study

We conduct a comprehensive ablation study to quantify the contributions of different architectural components and training strategies in CTX-Coder. Specifically, we examine (1) the impact of cross-attention design and (2) the effect of pretraining strategies.

As shown in Table 5, removing the cross-attention mechanism results in a substantial drop in both accuracy and F1 score, confirming its critical role in integrating long-range contextual information. We further investigate the impact of different cross-attention insertion frequencies. When cross-attention is inserted every 8th block, performance improves compared to the single-in-middle setting (Acc: 0.8124, F1: 0.6891), indicating that more frequent integration is beneficial. Notably, inserting cross-attention every 2nd block yields the best results (Acc: 0.8911, F1: 0.8090), but the marginal improvement over the every-4th-block setting comes at the expense of significantly increased computational cost. Therefore, we adopt the every-4th-block configuration as the default for CTX-Coder to balance performance and efficiency.

Regarding pretraining, training from scratch (*w/o pretraining*) fails to achieve satisfactory results (F1: 0.4269), indicating that naive supervised finetuning is insufficient. pretraining on unimodal data improves performance (F1: 0.7515), but only the full two-stage (unimodal + multimodal) pipeline unlocks the full potential of CTX-Coder (F1: 0.7810). This highlights that both diverse context exposure and pretraining are vital for robust long-context modeling.

Ablated Settings	Changed Value	Acc	F1
Cross-attn	w/o cross-attn	0.7539	0.5791
	Single in middle	0.7713	0.6210
	Every 8th	0.8124	0.6891
	Every 4th	0.8706	0.7810
	Every 2nd	0.8911	0.8090
pretraining Stages	w/o pretraining	0.6313	0.4269
	w/o multimodal	0.8542	0.7515
	unimodal & multimodal	0.8706	0.7810

Table 5: Ablation results for cross-attention architecture (insertion strategy) and pretraining stages.

EXP3: Code Document Generation

To further investigate whether CTX-Coder can truly understand contextual function embedding, we design a dedicated task: code documentation generation. The model is provided with a function snippet and is asked to generate a corresponding docstring.

The benchmark dataset is from CodeBert(Feng et al. 2020) and consists 6 programming languages: Python, Go, Java, JavaScript, and PHP. Due to computational constraints, we select Python as the target language for evaluation. We follow the official settings of Li et al. (Li et al. 2023): docstrings are evaluated using smoothed 4-gram BLEU (Papineni et al. 2002) against the reference docstring from the original function, using only the first lines of the generated and reference docstrings (removing, e.g., descriptions of function arguments and return types that may appear in later lines). Each model generates the documentation string (docstring) for each function using greedy decoding.

First, we perform unimodal pretraining on the Python training set for 1 epoch, resulting in the **CTX-Coder-Python-Single** checkpoint. Building upon this checkpoint, we further fine-tune the model for 1 epoch on the code documentation generation task, using a learning rate of 1e-5 and

a batch size of 8. We compare CTX-Coder with the LLaMA 3.1, Qwen 2.5 (7B), and Qwen 2.5 (14B). The baseline models are also fine-tuned for 1 epoch for fair comparison.

We use the following textual input to guide CTX-Coder in generating documentation, while the baseline models generate code using the default settings. To prevent information leakage from code comments, we remove all comments from the input code before generation.

Prompt For Code Document Generation:

<Bot><User><Function>Generate docs of this function.<Eot><Assistant>...

As shown in Table 6, CTX-Coder achieves a BLEU score of 20.38, outperforming LLaMA 3.1 (8B) and Qwen 2.5 (7B), and closely approaching the performance of the larger Qwen 2.5 (14B). This indicates that CTX-Coder can effectively leverage function embeddings for summarization.

Model	BLEU
Qwen 2.5 (7B)	17.91
Llama 3.1 (8B)	17.23
CTX-Coder (10B)	20.38
Qwen 2.5 (14B)	20.65

Table 6: Performance on the Python portion of the code summarization task, evaluating model's ability to recognize and utilize function embeddings.

EXP4: Cross-File Code Generation

To evaluate the ability of CTX-Coder to utilize contextual information, we use the CrossCodeEval dataset (Ding et al. 2023), which is a multilingual code completion benchmark that requires language models to accurately complete code based on retrieved cross-file contextual information. Following the official guidelines, we use two metrics to evaluate the model.

- **Code Matching (C-M)** directly compares the generated code with the reference code using Exact Match (EM) and Edit Similarity (ES). These metrics assess the overall accuracy of the code completion process while considering elements such as identifiers, keywords, operators, delimiters, and literals.
- **Identifier Matching (I-M)** extracts identifiers from both the model's predictions and the reference code by parsing the code and generating two ordered lists of identifiers. The predicted identifiers are then compared with the reference, and the performance is reported using the Exact Match (EM) and the F1 score.

We split the Python dataset into 80% and 20% for training and testing, and we further trained the CTX-Coder-Python-Single checkpoint and Llama 3.1 for 3 epochs.

Following the official CrossCodeEval format, relevant code snippets from other files are concatenated into a single textual block, which serves as the contextual input to the model. These contextual functions are first embedded using the frozen LLaMA backbone, and the resulting representations are passed into the model via a cross-attention module. Meanwhile, the target function serves as the main textual input. CTX-Coder then combines both inputs to generate the missing code span. This setup enables us to assess the model's ability to understand and leverage contextual information from code embeddings.

The results of the Cross-file Code Completion task are summarized in Table 7. CTX-Coder consistently outperforms LLaMA 3.1 and Qwen 2.5 (7B) across all metrics, demonstrating its ability to effectively capture and utilize contextual functions in code completion task. While Qwen 2.5 (14B) achieves the highest scores, CTX-Coder (10B) remains competitive despite having fewer parameters.

Model	**C-M**		**I-M**	
	EM	**ES**	**EM**	**F1**
Qwen 2.5 (7B)	20.68	60.43	27.15	56.03
Llama 3.1 (8B)	22.73	64.23	29.29	58.91
CTX-Coder (10B)	24.82	71.47	31.3	64.41
Qwen 2.5 (14B)	25.91	74.80	33.5	70.16

Table 7: Evaluation on Cross-file Code Completion task.

Conclusion and Limitations

Conclusion. In this paper, we present CTX-Coder, a context-enhanced framework for code vulnerability detection. To enhance contextual understanding, we encode surrounding functions into embeddings and fuse them with the target function using cross-attention. A two-stage pretraining strategy—unimodal and multimodal—is employed to help the model learn to interpret and utilize these embeddings. As a result, CTX-Coder is capable of effectively modeling long-range inter-function dependencies. CTX-Coder consistently achieves superior performance compared to baseline models with even larger parameter counts, including Qwen2.5-14B and SecGPT. Notably, it maintains high robustness under long-context scenarios, with only a 5.01% drop in F1 score as input length increases significantly lower than the 25%–41.5% degradation observed in other models. These results highlight the effectiveness of our context-aware design. Additional experiments on code documentation and cross-file code completion tasks demonstrate that CTX-Coder can leverage function-level embeddings to enhance downstream code understanding.

Limitations. Despite its strong performance on C/C++ codebases, the generalizability of CTX-Coder to other programming languages such as Java remains unclear. Moreover, the current approach relies on inserting new cross-attention layers into the LLM, which requires substantial pretraining to achieve optimal performance. In future work, we plan to explore the applicability of CTX-Coder to other programming languages, and investigate more efficient or architecture-agnostic integration strategies to further improve detection accuracy and training efficiency.

References

Adebiyi, A.; Arreymbi, J.; and Imafidon, C. 2013. Security Assessment of Software Design using Neural Network. arXiv:1303.2017.

Akhoundali, J.; Nouri, S. R.; Rietveld, K.; and Gadyatskaya, O. 2024. MoreFixes: A Large-Scale Dataset of CVE Fix Commits Mined through Enhanced Repository Discovery. In *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2024, 42–51. New York, NY, USA: Association for Computing Machinery. ISBN 9798400706752.

Allamanis, M.; Jackson-Flux, H.; and Brockschmidt, M. 2021. Self-Supervised Bug Detection and Repair. In Ranzato, M.; Beygelzimer, A.; Dauphin, Y.; Liang, P.; and Vaughan, J. W., eds., *Advances in Neural Information Processing Systems*, volume 34, 27865–27876. Curran Associates, Inc.

Cao, D.; Liao, Y.; and Shang, X. 2024. RealVul: Can We Detect Vulnerabilities in Web Applications with LLM? arXiv:2410.07573.

Cao, S.; Sun, X.; Bo, L.; Wu, R.; Li, B.; and Tao, C. 2022. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, 1456–1468. New York, NY, USA: Association for Computing Machinery. ISBN 9781450392211.

Chakraborty, S.; Krishna, R.; Ding, Y.; and Ray, B. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering*, 48(9): 3280–3296.

Chen, J.; Wang, W.; Liu, B.; Cai, S.; Towey, D.; and Wang, S. 2024. Hybrid semantics-based vulnerability detection incorporating a Temporal Convolutional Network and Self-attention Mechanism. *Information and Software Technology*, 171: 107453.

D'Ambros, M.; Bacchelli, A.; and Lanza, M. 2010. On the Impact of Design Flaws on Software Defects. In *2010 10th International Conference on Quality Software*, 23–31.

Dinella, E.; Dai, H.; Li, Z.; Naik, M.; Song, L.; and Wang, K. 2020. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International conference on learning representations (ICLR)*.

Ding, Y.; Fu, Y.; Ibrahim, O.; Sitawarin, C.; Chen, X.; Alomair, B.; Wagner, D.; Ray, B.; and Chen, Y. 2024. Vulnerability Detection with Code Language Models: How Far Are We? arXiv:2403.18624.

Ding, Y.; Wang, Z.; Ahmad, W.; Ding, H.; Tan, M.; Jain, N.; Ramanathan, M. K.; Nallapati, R.; Bhatia, P.; Roth, D.; and Xiang, B. 2023. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems*, volume 36, 46701–46723. Curran Associates, Inc.

Du, X.; Zheng, G.; Wang, K.; Feng, J.; Deng, W.; Liu, M.; Chen, B.; Peng, X.; Ma, T.; and Lou, Y. 2024. Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG. arXiv:2406.11147.

Fan, J.; Li, Y.; Wang, S.; and Nguyen, T. N. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, 508–512. New York, NY, USA: Association for Computing Machinery. ISBN 9781450375177.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; and Zhou, M. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155.

Filus, K.; Siavvas, M.; Domańska, J.; and Gelenbe, E. 2021. The Random Neural Network as a Bonding Model for Software Vulnerability Prediction. In *Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 102–116. Cham: Springer International Publishing.

Grattafiori, A.; Dubey, A.; Jauhri, A.; and et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783.

Guo, W.; Fang, Y.; Huang, C.; Ou, H.; Lin, C.; and Guo, Y. 2022. HyVulDect: A hybrid semantic vulnerability mining system based on graph neural network. *Computers Security*, 121: 102823.

Hanif, H.; and Maffeis, S. 2022. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, 1–8.

Huang, H.; Guo, L.; Zhao, L.; Wang, H.; Xu, C.; and Jiang, S. 2024. Effective combining source code and opcode for accurate vulnerability detection of smart contracts in edge AI systems. *Applied Soft Computing*, 158: 111556.

Li, R.; Allal, L. B.; Zi, Y.; Muennighoff, N.; Kocetkov, D.; Mou, C.; Marone, M.; Akiki, C.; Li, J.; Chim, J.; Liu, Q.; and et al. 2023. StarCoder: may the source be with you! arXiv:2305.06161.

Li, Y.; Li, X.; Wu, H.; Zhang, Y.; Cheng, X.; Liu, Y.; Xu, F.; and Zhong, S. 2025. If LLMs Would Just Look: Simple Line-by-line Checking Improves Vulnerability Localization. arXiv:2410.15288.

Li, Z.; Dutta, S.; and Naik, M. 2025. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In *The Thirteenth International Conference on Learning Representations*.

Li, Z.; Wang, N.; Zou, D.; Li, Y.; Zhang, R.; Xu, S.; Zhang, C.; and Jin, H. 2024. On the Effectiveness of Function-Level Vulnerability Detectors for Inter-Procedural Vulnerabilities. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24. New York, NY, USA: Association for Computing Machinery. ISBN 9798400702174.

Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; and Chen, Z. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4): 2244–2258.

Liu, Z.; Liao, Q.; Gu, W.; and Gao, C. 2023. Software Vulnerability Detection with GPT and In-Context Learning. In *2023 8th International Conference on Data Science in Cyberspace (DSC)*, 229–236.

Lu, G.; Ju, X.; Chen, X.; Pei, W.; and Cai, Z. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212: 112031.

Mirsky, Y.; Macon, G.; Brown, M.; Yagemann, C.; Pruett, M.; Downing, E.; Mertoguno, S.; and Lee, W. 2023. VulChecker: Graph-based Vulnerability Localization in Source Code. In *32nd USENIX Security Symposium (USENIX Security 23)*, 6557–6574. Anaheim, CA: USENIX Association. ISBN 978-1-939133-37-3.

MITRE. 2020. CVE - Common Vulnerabilities and Exposures. https://cve.mitre.org/. Accessed: 2025-02-04.

MITRE. 2024. CVE - Common Vulnerabilities and Exposures. https://cve.mitre.org/. Accessed: 2025-02-04.

Nikitopoulos, G.; Dritsa, K.; Louridas, P.; and Mitropoulos, D. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, 1565–1569. New York, NY, USA: Association for Computing Machinery. ISBN 9781450385626.

Nong, Y.; Aldeen, M.; Cheng, L.; Hu, H.; Chen, F.; and Cai, H. 2024. Chain-of-Thought Prompting of Large Language Models for Discovering and Fixing Software Vulnerabilities. arXiv:2402.17230.

Nong, Y.; Yang, H.; Cheng, L.; Hu, H.; and Cai, H. 2025. APPATCH: Automated Adaptive Prompting Large Language Models for Real-World Software Vulnerability Patching. arXiv:2408.13597.

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W.-J. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 311–318.

Russell, R.; Kim, L.; Hamilton, L.; Lazovich, T.; Harer, J.; Ozdemir, O.; Ellingwood, P.; and McConley, M. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 757–762.

Sheng, Z.; Wu, F.; Zuo, X.; Li, C.; Qiao, Y.; and Hang, L. 2024. LProtector: An LLM-driven Vulnerability Detection System. arXiv:2411.06493.

Ullah, S.; Han, M.; Pujar, S.; Pearce, H.; Coskun, A.; and Stringhini, G. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *2024 IEEE Symposium on Security and Privacy (SP)*, 862–880.

Wang, C.; Li, Z.; Pena, Y.; Gao, S.; Chen, S.; Wang, S.; Gao, C.; and Lyu, M. R. 2023. REEF: A Framework for Collecting Real-World Vulnerabilities and Fixes. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 1952–1962.

Wen, X.-C.; Gao, C.; Ye, J.; Li, Y.; Tian, Z.; Jia, Y.; and Wang, X. 2024a. Meta-Path Based Attentional Graph Learning Model for Vulnerability Detection. *IEEE Transactions on Software Engineering*, 50(3): 360–375.

Wen, X.-C.; Wang, X.; Chen, Y.; Hu, R.; Lo, D.; and Gao, C. 2024b. VulEval: Towards Repository-Level Evaluation of Software Vulnerability Detection. arXiv:2404.15596.

Wu, B.; Zou, F.; Yi, P.; Wu, Y.; and Zhang, L. 2023. SlicedLocator: Code vulnerability locator based on sliced dependence graph. *Computers Security*, 134: 103469.

Zhou, Y.; Liu, S.; Siow, J.; Du, X.; and Liu, Y. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In Wallach, H.; Larochelle, H.; Beygelzimer, A.; d'Alché-Buc, F.; Fox, E.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.