CLNX: Bridging Code and Natural Language for C/C++ Vulnerability-Contributing Commits Identification

Zeqing Qin[1,2], **Yiwei Wu**[1], **Lansheng Han**[*1,2,3]

[1]School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China
[2]Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security
[3]Wuhan JinYinHu Laboratory
zeqing@hust.edu.cn, cnwyw77777@gmail.com, hanlansheng@hust.edu.cn

Abstract

Large Language Models (LLMs) have shown great promise in vulnerability identification. As C/C++ comprise half of the open-source Software (OSS) vulnerabilities over the past decade and updates in OSS mainly occur through commits, enhancing LLMs' ability to identify C/C++ Vulnerability-Contributing Commits (VCCs) is essential. However, current studies primarily focus on further pre-training LLMs on massive code datasets, which is resource-intensive and poses efficiency challenges. In this paper, we enhance the ability of BERT-based LLMs to identify C/C++ VCCs in a lightweight manner. We propose CodeLinguaNexus (CLNX) as a bridge facilitating communication between C/C++ programs and LLMs. Based on commits, CLNX efficiently converts the source code into a more natural representation while preserving key details. Specifically, CLNX first applies Structure-level Naturalization to decompose complex programs, followed by Token-level Naturalization to interpret complex symbols. We evaluate CLNX on public datasets of 25,872 C/C++ functions with their commits. The results demonstrate that CLNX substantially improves the ability of LLMs to detect C/C++ VCCs. Moreover, CLNX-equipped CodeBERT achieves new state-of-the-art performance and identifies 38 OSS vulnerabilities in the real world.

1 Introduction

In recent years, with the rapid growth of open-source software (OSS) applications, the number of OSS vulnerabilities has been increasing significantly. According to the data from the 2023 OSSRA report (Synopsys 2023), in the 1,703 codebases analyzed by the Black Duck audit team, 84% of the codebases contained at least one known open-source vulnerability, and 48% contained high-risk vulnerabilities. Moreover, 52.13% of reported vulnerabilities in OSS are implemented in C/C++ (Wang et al. 2023) over the past decade. As patch commit is the primary way to update code in OSS, identifying Vulnerability-Contributing Commits (VCCs) can prevent new vulnerabilities from being introduced into OSS to a large extent (Meneely et al. 2013).

Large Language Models (LLMs), particularly those based on BERT (Devlin et al. 2018) architecture, have demonstrated their potential to identify vulnerabilities by effectively learning code dependencies and contextual nuances (Xu et al. 2022). The models' effectiveness arises from their bidirectional encoder architecture, which captures contextual semantics around code segments. However, since these models are trained initially on natural language, their code comprehension capabilities can still be considerably enhanced. Current research primarily focuses on further pre-training LLMs on extensive code datasets. (Xu et al. 2022). For example, CodeBERT (Feng et al. 2020) has been pre-trained on six programming languages: Python, Java, JavaScript, PHP, Ruby, and Go. Nevertheless, it exhibits suboptimal performance in C/C++ due to the absence of specific pre-training for these languages. More importantly, the improvements remain marginal even after extensive further pre-training. For instance, CodeBERT-cpp (Zhou et al. 2023), an extension of CodeBERT further pre-trained on C++, shows only marginal improvements (a rise of 2.03% in accuracy) in identifying C/C++ vulnerability while consuming significant computation resources. It indicates that further pre-training is inefficient and occasionally ineffective (Liu et al. 2023a).

Specifically, we address the major challenge in our paper.

- How to enhance the effectiveness of LLMs in identifying C/C++ VCCs while ensuring a lightweight implementation?

To address this challenge, we introduce CodeLinguaNexus (CLNX), a middleware that transforms original C/C++ code into a format more amenable to LLMs. To do so, we first perform the Structure-level Naturalization. Specifically, we linearize the structures of the C/C++ source code with commit and shorten their length. Then, we perform Token-level Naturalization, interpreting complex C/C++ symbols into their natural language equivalents.

We implement CLNX and evaluate it on a dataset of 25, 872 C/C++ functions with corresponding commits, including 10, 894 VCCs. The result indicates that CLNX significantly improves LLMs' ability to identify C/C++ VCCs. Moreover, equipped with CLNX, BERT undergoes an increase of 14.48% in precision, surpassing other models that have been further pre-trained on code. Finally, CLNX-CodeBERT achieves state-of-the-art performance. Lastly, CLNX-CodeBERT finds 38 real-world OSS vulnerabilities by identifying vulnerability-contributing commits.

*Corresponding author

Listing 1: A Vulnerability-Contributing Commit

```
1  @@ -212,7 +212,7 @@
2  diff --git a/server/util_mutex.c b/
       server/util_mutex.c
3  - a/server/util_mutex.c
4  + b/server/util_mutex.c
5  @@ -120 +120 @@ AP_DECLARE(apr_status_t)
6  -  *mutexfile = ap_server_root_relative(
       pool, file);
7  +  *mutexfile = ap_runtime_dir_relative(
       pool, file);
8  @@ -307 +307 @@ static const char
9  -  return ap_server_root_relative(p,
10 +  return ap_runtime_dir_relative(p,
11 @@ -555 +555 @@ AP_CORE_DECLARE(void)
12 -  dir = ap_server_root_relative(p,
       mxcfg->dir);
13 +  dir = ap_runtime_dir_relative(p,
       mxcfg->dir);
```

In summary, our contributions in this paper are:

- We propose CLNX, a pioneering framework for improving LLMs' performance in C/C++ VCCs identification in an effective and efficient way.
- We successfully implement a prototype of CLNX and conduct extensive experiments to evaluate its effectiveness.
- We equip CodeBERT with CLNX to achieve the new state-of-the-art performance and demonstrate CLNX-CodeBERT's ability to identify vulnerabilities in the real world.

2 Preliminaries

Vulnerability-Contributing Commits

In OSS development, patch commits record the differences between two versions of the source code (Zuo and Rhee 2024). They can be categorized into two types: vulnerable patch commits and non-vulnerable patch commits. Vulnerable patch commits refer to those that will introduce new vulnerabilities into the original code, which are also called Vulnerability-Contributing Commits (VCCs) (Meneely et al. 2013). In this research, a patch commit is considered "vulnerable" if it introduces vulnerabilities that belong to any of the Common Weakness Enumeration (CWE), regardless of its triggering conditions (Wang et al. 2023). Listing 1 shows a vulnerable patch commit with code revisions marked by plus and minus signs (+/−) on the left side. This commit is a configuration item change aimed at improving the path settings for mutexes in the Apache HTTP Server. However, it introduces a vulnerability related to permission bypass. Vulnerable patch commits highlight critical information about vulnerabilities. When identifying VCCs at the functional level, both the patch commit and the source code of the revised function are analyzed.

Pre-training and Fine-tuning

Pre-training in this paper refers to the training phase of LLMs conducted on large-scale unlabeled datasets. LLMs can generally be divided into two categories: BERT-based and GPT-based. Since GPT-based LLMs are composed of a decoder structure and are more suitable for generative tasks (Hadi et al. 2023), we primarily focus on the performance of BERT-based models in vulnerability identification, a code classification task (Xu et al. 2022). BERT-based LLMs are pre-trained on tens of millions of text data using techniques like Masked Language Modeling (MLM) and Next Sentence Prediction (NSP) (Xu et al. 2022). During the pre-training phase, these models capture helpful information from the data and store it in their weights. These pre-trained models are then fine-tuned on labeled data for specific downstream tasks like text classification or question answering. While pre-training requires substantial computational resources, fine-tuning is comparatively more resource-efficient (Radiya-Dixit and Wang 2020).

3 Methodology

This section presents an overview of our approach and details each component, including Structure-level Naturalization and Token-level Naturalization.

Overview

The overview of CLNX is shown in Figure 1, with CLNX's internal structure displayed on its left side. CLNX is performed at the function level. In the process of handling input source code and patch commit, CLNX initially performs Structural-level Naturalization. The first step employs CLNX's code analyzer to transform the source code into a graph of linear execution paths. Next, patch information is integrated to identify the critical path. Then, CLNX proceeds with Token-level Naturalization, which maps the identified critical path to the corresponding source code and converts key symbols into their natural language equivalents. Finally, CLNX outputs the fully naturalized source code. The system workflow for deploying CLNX to enhance LLMs' performance in VCCs identification is shown on the right side of Figure 1. For a given set of programs with their corresponding patch commits, the programs are naturalized by CLNX and then provided to LLMs for fine-tuning. When an unknown program with its patch commit is analyzed, CLNX transforms the program into naturalized form and then forwards the results to the fine-tuned LLMs for vulnerability identification. The remainder of Section 3 provides details on each component of CLNX.

Code Analyzer

C/C++ programs' complex structures and excessive length challenge LLMs in understanding them. In response to these challenges, CLNX's Structure-level Naturalization is designed with two primary goals: The first goal is to linearize complex program structures. The second goal is to reduce the overall program length. In particular, the code analyzer extracts linear execution paths within a program.

The concept of "basic blocks," borrowed from LLVM (Racordon 2021), underpins the design of CLNX's code analyzer. A "basic block" is a sequence of instructions that executes sequentially, characterized by a single entry and a

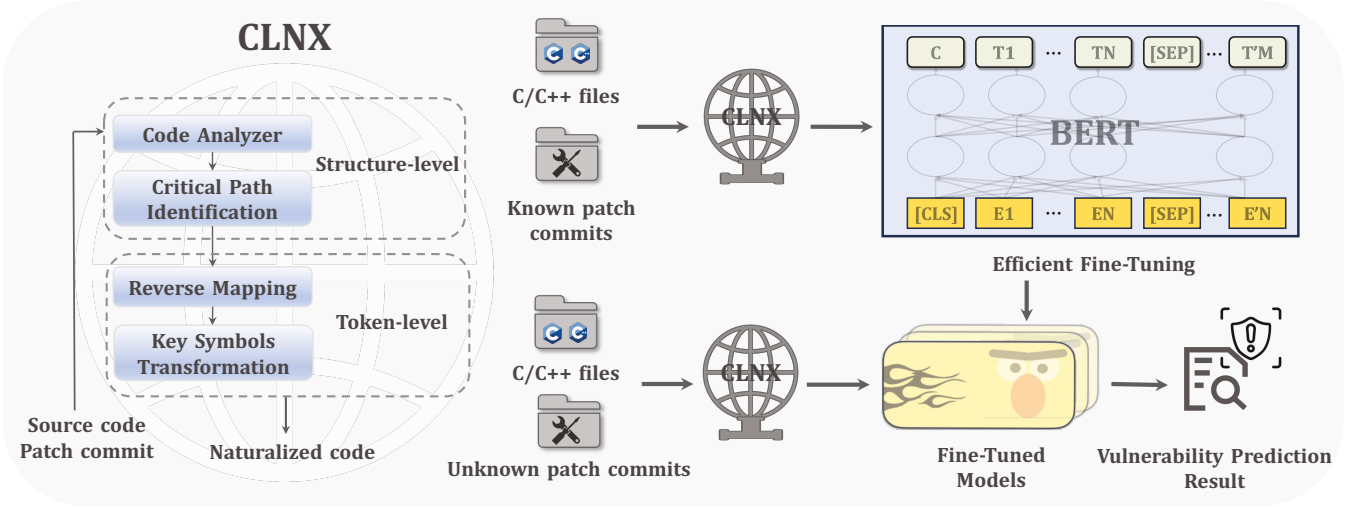

Figure 1: The Overview of CLNX

single exit point, devoid of any internal branching. The code analyzer transforms programs into basic blocks and generates a graph $G = (V, E)$, where each vertex in V corresponds to a basic block, and each edge in E represents the control flow between blocks. The graph G's entrance point v_{entry} corresponds to the program's entry basic block, and its exit point v_{exit} corresponds to the program's final basic block. As a result, any path traversed from v_{entry} to v_{exit} within G delineates a linear execution path of the program. In particular, when there is a loop structure, for simplicity, we directly convert the control flow to single executions and label the corresponding nodes as loop structures. It should be noted that CLNX only uses Abstract Syntax Tree (AST) and Control Flow Graph (CFG) for code embedding. The Program Dependence Graph (PDG), integrating both control dependency graph (CDG) and data dependency graph (DDG), is commonly used to abstractly represent source code (Wang et al. 2023). However, we believe that complex structures risk subjectively introducing excessive irrelevant information, thereby complicating the accurate semantic representation of the code. We compare our method with complex graph-based approaches (embedding AST/CFG/DDG/CDG) in RQ2 to demonstrate CLNX's effectiveness.

The code analyzer deploys Joern to generate AST. The whole process is illustrated in Step 1 of Figure 2. In contrast to LLVM, CLNX's code analyzer does not impose requirements on the actual compilability of the program. This attribute is particularly significant for identifying function-level vulnerabilities, especially in scenarios where the absence of relevant header files precludes successful compilation.

Critical Path Identification

After obtaining the graph G, composed of basic blocks, the focus of CLNX shifts to identifying a critical execution path within the graph that encompasses the maximum amount of vulnerability-related basic blocks. This process can be divided into two primary steps, as illustrated in Step 2 of Figure 2; Firstly, determining the basic blocks that are directly related to a patch commit. Secondly, the critical path within G is selected, which offers the most extensive coverage of these identified basic blocks.

Commit-related Basic Blocks Identification Based on the idea of taint analysis (Boxler and Walcott 2018), CLNX identifies the code removed in the corresponding patch commits of a program as contamination points. CLNX also considers an extended range, which includes three lines (Wang et al. 2023) before and after the lines corresponding to the removed code, as the affected tainted area. This area is represented as $S = [l_s, l_e]$, where l_s and l_e are the start and the end line numbers of the tainted area, respectively. A basic block BB_i, covering the line number range $[b_{is}, b_{ie}]$, is regarded as commit-related, denoted $BB_{tainted_i}$, if its range intersects with the tainted area, i.e., $\{BB_{tainted_i} \mid [l_s, l_e] \cap [b_{is}, b_{ie}] \neq \emptyset\}$.

Critical Executing Path Selection Further, CLNX is going to select one critical executing path in graph G. CLNX designates the basic block corresponding to the program's entry point as the source (BB_{source}) and the basic block corresponding to the program's exit point as the sink (BB_{sink}). Based on this, the critical linear execution path P in the graph structure, which originates from BB_{source} and terminates at BB_{sink}, aims to maximize the coverage of vulnerability-related basic blocks $BB_{tainted}$. CLNX designs its critical_path_selecting algorithm based on dynamic programming to circumvent the issue of path explosion. critical_path_selecting algorithm selects the critical execution path in graph G by satisfying three primary criteria: first, the path covers as many $BB_{tainted}$ as possible; second, the path minimizes length; and third, if two paths have the same length and contain the same number of $BB_{tainted}$, then select the one with the highest information entropy value. The Pseudo code of the critical_path_selecting algorithm is

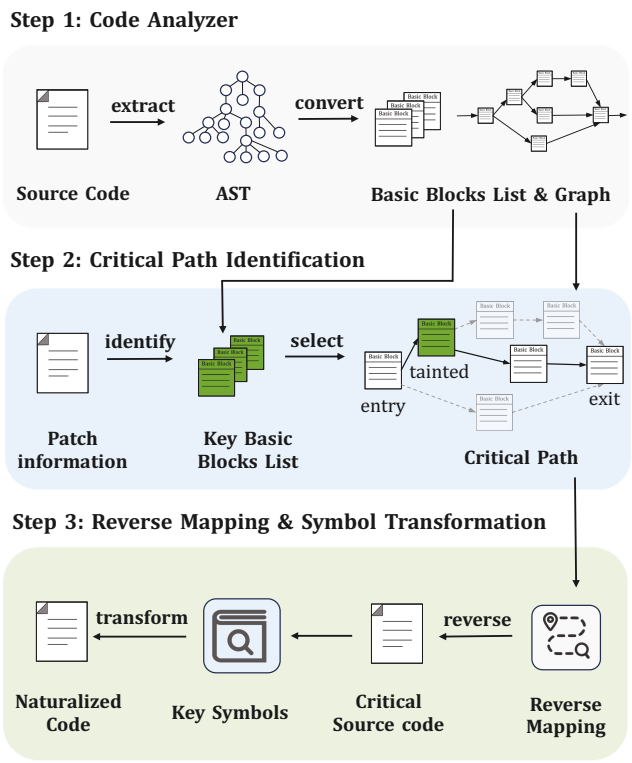

Figure 2: The Workflow of CLNX

shown in the extended version as Algorithm 1 (Qin, Wu, and Han 2024).

Reverse Mapping

As a compiled programming language, C/C++ has more low-level symbols compared to natural languages. In response, CLNX's Token-level Naturalization is designed to translate complex symbols into their natural linguistic equivalents. Initially, CLNX undertakes the task of reverse mapping the critical path, composed of basic blocks, back to the source code. This step involves reconstructing the source code information by tracing the sequence of basic blocks within the critical path. Owing to the grand architecture of CLNX basic blocks, the implementation of reverse mapping is straightforward and efficient.

Key Symbols Transformation

Given the source code, CLNX deploys appropriate transformations based on the symbols' types and rewrites the symbols to their natural language equivalents. This part corresponds to Step 3 of Figure 2. First, we extract all keywords and symbols following the lexical rules in the C++ reference documentation, then provide natural language equivalents based on these descriptions. To further refine the output, we apply the Longest Common Subsequence (LCS) algorithm to remove duplicates, enhancing the distinctions between different equivalents. For instance, the symbols "<<" and ">>" are defined in the C++ reference as "Bitwise left shift" and "Bitwise right shift." Our approach simplifies these to "left shift" and "right shift," enabling the language model to recognize these symbols accurately while distinguishing their meanings. Table 1 provides some illustrative examples.

Type	Example	Natural Language Equivalents
Operator	*p	dereference p
	&var	address of var
	a->b	access b via pointer a
	a.*b	access a's member via b
	a ^b	a XOR b
	∼ a	Bitwise NOT a
	a << b	a left shift by b
	a >> b	a right shift by b
Memory	alignas(16)	alignment as 16 bytes
	alignof(type)	get type alignment
	decltype(x) y	get x's type to y
	nullptr	null pointer
Concurrency Keywords	co_await task()	wait for task
	co_yield value	produce value temporarily
Preprocessor Directive	#include<h>	include header file <h>
	#undef M	undefine the macro M

Table 1: Examples of Key Symbols Transformations

4 Experimental Setup

Our evaluation is designed to answer the following research questions:

- RQ1: How does CLNX enhance LLMs for the C/C++ VCCs identification task?
- RQ2: How does the performance of CLNX-equipped LLMs compare to other vulnerability identification-related methods?
- RQ3: How does CLNX-equipped LLM perform in identifying real-world OSS vulnerabilities that are contributed through commits?

Evaluation Task

The evaluation task of our paper is Vulnerable-Contributing Commits (VCCs) identification, where the input is the source code and the corresponding patch commit, and the output is a label denoting whether the commit will introduce vulnerabilities into the original code or not.

Datasets

To evaluate our research questions using real-world data, we construct experimental datasets based on the publicly available Devign dataset (Zhou et al. 2019). Our dataset contains 25,872 pairs of vulnerable and non-vulnerable functions, along with their associated commit IDs, from two major open-source C/C++ projects: FFmpeg and Qemu. The dataset is randomly split into training, validation, and test sets in an 8:1:1 ratio.

Evaluation Metrics

In our experiments, different metrics are used to evaluate downstream tasks. We follow the metrics that CodeXGLUE

(Lu et al. 2021) used for evaluation, and the details are listed below:

- Prec: Precision measures the proportion of correct positive identifications made by the model compared to the total predicted positives.
- Acc: Accuracy defines the ratio of correct predictions (i.e., the exact match) in the test set.
- Recall: This metric concentrates on the model's ability to correctly identify all genuine positive instances. It calculates the proportion of true positives accurately detected by the model out of the total positives.
- F1: This metric is the harmonic mean of precision and recall, balancing these two metrics. It is advantageous when class distribution is imbalanced.

Baselines

We evaluate both BERT-based and GPT-based LLMs for this study, with a primary focus on CLNX's effectiveness in improving BERT-based LLMs since their proficiency in comprehensively understanding vulnerability through their bidirectional encoder structure. For comparison, we include vulnerable patch commit identification methods, deep learning vulnerability identification methods, and a traditional vulnerability identification tool. The details of the baselines are listed in Table 2 of the extended version (Qin, Wu, and Han 2024). Notably, as GPT-based LLMs with parameter quantities scaling up to Billion, BERT-based LLMs have a relatively lower parameter scale. To maintain fairness in the assessment, we have chosen the 7b edition of CodeLlama (Roziere et al. 2023) for comparison.

Experimental Settings

In our evaluation tasks, we utilize the established configuration parameters for LLMs following the standardized settings provided by CodeXGLUE (Lu et al. 2021). All the compared methods are reimplemented to adhere to the default specifications outlined in their foundational papers. Our implementation of CLNX utilizes Joern v2.0.120 and Scala v3.3.1. All operations of CLNX, including the code analyzer, critical path selection, and key symbol transformation, are executed on an Intel Xeon(R) Gold 6326 CPU @ 2.90GHz. We perform LLMs fine-tuning on a dedicated machine with an NVIDIA A100 GPU featuring 64GB of memory. The fine-tuning parameters and the process are in accordance with the defect-detection subject of CodeXGLUE (Lu et al. 2021), where the block size is 400, the train batch size is 32, the eval batch size is 64, and the learning rate is 2e-5. Additionally, the dataset is split into training, testing, and validation sets in an 8:1:1 ratio.

5 Experimental Result

RQ1: Effectiveness

We conduct extensive experiments and an ablation study to assess the effectiveness of CLNX's two sequential naturalization phases in enhancing LLMs' ability to identify C/C++ VCCs. It should be noted that RoBERTa, ContraBERT, CodeBERT, and CodeBERT-cpp have undergone further pre-training with programming data. The results, including precision, accuracy, recall, and F1 score, are presented in Table 2. "CLNX_s-" denotes models equipped only with CLNX's Structure-level Naturalization, while "CLNX-" signifies models that completed both naturalization phases.

Technique	Prec	Acc	Recall	F1
GPT-3.5 Turbo	16.78%	31.88%	11.05%	34.84%
GPT-4.0	37.08%	42.68%	33.16%	42.05%
CodeLlama-7b	50.39%	49.16%	50.23%	49.69%
BERT	58.60%	59.85%	48.94%	54.66%
CLNX_s-BERT	70.33%↑	62.53%↑	46.52%	55.99%↑
CLNX-BERT	73.08%↑	63.19%↑	49.91%↑	59.98%↑
DistilBERT	63.94%	61.47%	46.56%	53.88%
RoBERTa	65.85%	61.21%	47.64%	55.28%
ContraBERT	64.78%	63.89%	48.92%	55.74%
CodeBERT-cpp	62.97%	64.21%	48.37%	54.71%
CodeBERT	66.89%	62.18%	45.16%	53.91%
CLNX_s-CodeBERT	71.66%↑	63.97% ↑	43.47%	53.95% ↑
CLNX-CodeBERT	**75.16%**↑	**65.47%**↑	**51.83%**↑	**60.64%** ↑

Table 2: Results of LLMs in C/C++ VCCs identification

From Table 2, we can see that GPT-based models like CodeLlama-7b-Instruct do not perform well on this task, so we mainly focus on BERT-based LLMs. There are significant improvements in C/C++ VCCs identification for BERT and CodeBERT after the sequential deployment of CLNX's two-phase naturalization. Specifically, BERT's precision improved by 14.48%, and CodeBERT's by 8.27%, with CLNX-CodeBERT outperforming all LLMs across all metrics, highlighting CLNX's impact. Although BERT's initial precision (58.60%) is relatively low compared to CodeBERT (66.89%), BERT with only CLNX's Structure-level Naturalization achieves a precision result of 70.33%. It surpasses all the models that have been further pre-trained on program data, including CodeBERT and RoBERTa. These results directly validate that CLNX yields a better effect than pre-training strategies. We attribute this improvement to CLNX's effectiveness in simplifying complex structures and emphasizing critical vulnerability information. However, we notice that accuracy and precision values are improved for both BERT and CodeBERT after CLNX's Structure-level Naturalization; the recall values decreased by 2.42% and 1.69%, respectively. These results suggest that the models miss some vulnerabilities. We believe this phenomenon is caused by CLNX's mission to reduce the source code length. In CLNX's Structure-level Naturalization stage, it excessively prioritizes program length reduction when dealing with multiple paths with consistent coverage of critical nodes, which may result in the loss of certain vulnerability-related information. Yet, the complete CLNX process eventually led to the highest recall rates for both models, indicating the Token-level Naturalization phase's effectiveness in enhancing the understanding of retained information.

Answer to RQ1: Both the Structure-level and Token-level Naturalization phases play crucial roles in CLNX's effectiveness. CLNX enhances LLMs' performance in C/C++

Technique	Prec	Acc	Recall	F1
Cppcheck	37.02%	50.65%	17.13%	23.43%
GraphSPD	64.57%	62.65%	40.75%	50.12%
VulFixMiner	50.35%	53.61%	11.72%	19%
Russell et al.	53.02%	57.93%	39.67%	45.38%
VulDeePecker	48.42%	53.55%	26.40%	34.17%
SySeVR	48.52%	52.67%	64.67%	55.44%
REVEAL	56.95%	62.43%	**67.80%**	59.76%
Devign	53.62%	58.62%	61.44%	57.26%
CLNX-CodeBERT	**75.16%**	**65.47%**	51.83%	**60.64%**

Table 3: Results of comparative analysis

VCCs identification significantly.

RQ2: Comparision

To further evaluate the performance of CLNX-equipped LLM, we compare it with popular deep learning vulnerability identification methods, traditional tools, and vulnerable commit identification methods. We use CodeBERT as the base model for this comparison. The results are presented in Table 3.

As shown in Table 3, CLNX-CodeBERT significantly outperforms all the compared methods in precision (improve 10.59%), accuracy, and F1 score, achieving new state-of-the-art performance for this task. Notably, CLNX-CodeBERT excels over three graph-based methods (GraphSPD, Devign, REVEAL) that use complex code embedding methods. This success can be attributed to two factors: The first factor is that BERT-based LLMs can perform comprehensive code analysis by considering surrounding elements like variables and functions. The second factor is that CLNX's simple code embedding method enables LLMs to emphasize key semantic information and operate more efficiently, addressing the redundancy issue commonly seen in graph-based models.

However, while CLNX-CodeBERT achieves the highest recall score among LLMs (Table 2), it does not achieve the top recall score when compared to other methods in Table 3. This discrepancy likely arises from the pre-training objectives of LLMs, which emphasize precision-focused tasks over exhaustive recall coverage, as discussed in (Le Bronnec et al. 2024). Nevertheless, as previously noted, CLNX-CodeBERT excels across all other metrics. In particular, the F1 score provides a balanced and comprehensive measure of vulnerability detection performance, validating the effectiveness of CLNX.

Answer to RQ2: With the help of CLNX, LLM achieves new state-of-the-art performance in C/C++ VCCs identification. The simple and lightweight code embedding approach of CLNX enables the LLM to capture key semantic information effectively.

RQ3: Real World Vulnerabilities

To evaluate the performance of CLNX-equipped LLM in detecting real-world vulnerabilities, we deploy the finetuned CLNX-CodeBERT to scan the repositories of 35 C/C++ open-source projects. Notably, VCCs are relatively rare in open-source software. CLNX-CodeBERT reports 106 vulnerable commits, 38 of which are verified as valid. Finally, we categorize these vulnerabilities by CWE, and the results are shown in Table 5 of the extended version (Qin, Wu, and Han 2024). The vulnerabilities cover types of Improper Permission Assignment for Critical Resource (CWE-264), Cryptographic Issues (CWE-310), Information Disclosure (CWE-200), Null Pointer Dereference (CWE-476), Out-of-Bounds Read (CWE-125), Resource Management Errors (CWE-399), Buffer Error (CWE-119), Race Condition (CWE-362), Improper Input Validation (CWE-20), Use After Free (CWE-416), Numeric Errors (CWE-189), and Double Free (CWE-415).

The results indicate that CLNX-CodeBERT can identify vulnerabilities of real-world C/C++ open-source projects introduced by commits. Furthermore, we observe that the model is proficient at identifying specific types of vulnerabilities, which can be ascribed to the CLNX's capability to distill critical information from vulnerability functions, thereby aiding CodeBERT in learning the specific patterns of these vulnerabilities. For instance, the model detected six Null Pointer Dereference (CWE-476) vulnerabilities and nine Buffer Error (CWE-119) vulnerabilities, which become more apparent without extraneous information. We attribute this to CLNX's effectiveness in refining key information from vulnerability functions, thus reducing the interference of irrelevant information on LLMs. However, the model only detects one Cryptographic Issue (CWE-310). This result is because vulnerabilities of such type often involve complex processing logic and do not have relatively uniform patterns.

Answer to RQ3: CLNX-CodeBERT effectively finds real-world vulnerabilities in open-source C/C++ repositories, demonstrating CLNX's potential to help LLMs report 0-day C/C++ vulnerabilities in OSS.

6 Discussion

This section discusses the implications, limitations, and potential threats to the validity of our work.

Implications

We propose a novel, cost-effective framework that enhances the effectiveness of LLMs in identifying C/C++ VCCs. The findings in our research are expected to inspire researchers to improve LLMs' ability to identify VCCs across more programming languages. CLNX offers guidelines for improving LLMs' performance in VCCs identification of specific programming languages in a lightweight manner, moving beyond the traditional reliance on extensive pre-training, which requires substantial computational resources.

Limitations

The experimental results demonstrate that CLNX significantly enhances the performance of LLMs in VCCs identification. The advancement is mainly due to CLNX's effective two-stage naturalization, making the code more compatible with LLMs. However, challenges arise from a decline in the Recall score, mainly due to its Structure-level Naturalization, which might inadvertently omit important code

information. When confronted with multiple paths having equivalent coverage of tainted basic blocks, CLNX's critical_path_selecting algorithm prioritizes the shortest path for length minimization at the risk of overlooking important details. A more effective approach could involve considering data flow more substantially in the critical path selection process. However, it involves dynamic program analysis. We will explore it in our future work.

Threats to Validity

Internal Validity: Our analysis identifies two potential threats to internal validity. Firstly, the uniform standard requirement of CLNX's code analyzer necessitates standardizing source code format before its use. Secondly, CLNX calculates path length by counting the number of basic blocks, assuming each block adds uniformly to the total length. To maintain algorithmic integrity in our critical key path selection algorithm, all edges of the input graph structure must be of equal length (by default, set to one).
External Validity: Regarding external validity, the performance of the original GPT-based LLMs is significantly lower than that of BERT-based models, so we mainly focus on how CLNX improves BERT-based LLMs' performance in C/C++ VCCs identification.

7 Related Work

Large Language Models: In recent years, there has been a notable emergence of LLMs, which are increasingly recognized as promising solutions for the field of vulnerability identification (Thapa et al. 2022) (Ding et al. 2022) (Feng et al. 2020) (Hanif and Maffeis 2022). BERT (Devlin et al. 2018) is a deep bidirectional encoder based on the transformer architecture, pre-trained by Google on a vast corpus comprising millions of text passages and billions of words. BERT-based LLMs are usually pre-trained on two tasks: Masked Language Model (MLM) and Next Sentence Prediction (NSP), thus equipping them with robust semantic understanding and endowing them with substantial knowledge, making it suitable for fine-tuning on specific tasks with limited data, such as vulnerability identification (Xu et al. 2022). Successful applications include BERT's superior detection accuracy on the SARD database, outperforming traditional machine learning models like LSTM and BiLSTM. Similarly, further pre-training of CodeBERT (Feng et al. 2020) and its variants, such as DistilBERT (Sanh et al. 2019), RoBERTa (Liu et al. 2019), ContraBERT (Liu et al. 2023b), and CodeBERT-cpp (Zhou et al. 2023), enhances LLM performance on programming languages.
Deep Learning Vulnerability Identification: These methods train various deep learning models with existing datasets (Steenhoek et al. 2023) (Russell et al. 2018) (Okun et al. 2013) (Black 2018) (Booth, Rike, and Witte 2013). Subsequently, these models are deployed to identify undetected vulnerabilities. They generally fall into two primary categories: token-based methods (Li et al. 2018) (Russell et al. 2018) (Li et al. 2021b) and graph-based (Zhou et al. 2019) (Chakraborty et al. 2021). Token-based approaches process the source code as sequences of tokens, leveraging models such as RNN (Li et al. 2021a) (Li et al. 2021b) (Li et al. 2018) (Zhang et al. 2019), CNN (Russell et al. 2018), and MLP (Coimbra et al. 2021) for training purposes. Some strategies utilize code slices to distill pivotal information. Conversely, graph-based methods seek to encapsulate the source code's multifaceted information into graphs, then analyze using various GNN (Cheng et al. 2021) (Cao et al. 2022). For example, the Code Property Graph (CPG) leverages information from abstract syntax trees, control flow graphs, and program dependency graphs to model the combined semantic and syntactic information of a program.
Patch Commit Identification: In OSS, code commits serve as the core building block units of a version control system in software development (Zuo and Rhee 2024). The patch commit (i.e., code changes + description of changes), or patch for short, is a general concept involving modifications that are specifically focused on code updates, such as introducing new features. However, this process may introduce new vulnerabilities into the original code. To address this, a significant amount of work has focused on patch commit analysis targeting vulnerability discovery (Zuo and Rhee 2024) (Meneely et al. 2013). In the early stage, handcrafted features-based methods are proposed. For example, VCCFinder (Perl et al. 2015) utilized an SVM model to automatically identify commits that might introduce vulnerabilities. Wang et al. (Wang et al. 2019) studied code diffs exclusively, employing 61 features, including 22 from previous work (Tian, Lawall, and Lo 2012), to form an input vector for their machine learning model. In recent years, advancements in neural networks, particularly in natural language processing (NLP) and applied graph theory, have revolutionized this field. E-SPI (Wu et al. 2022), for instance, analyzes both code diffs and commit messages by first extracting a contextual abstract syntax tree (AST) from code changes, then encoding it into paths using a BiLSTM. Commit messages are converted into graphs and processed with a graph neural network (GNN). However, the quality of commit messages can limit the usefulness of such analyses. In VulFixMiner (Zhou et al. 2021), the authors only consider code change information. To extract semantics from the code changes, they adopt CodeBERT. It is noteworthy that VulFixMiner only investigates Python and Java projects (Zuo and Rhee 2024). Most recently, a detection system called GraphSPD is proposed (Wang et al. 2023), which proposes a novel graph structure called PatchCPG to represent patches.

8 Conclusion

In this research, we propose CLNX, a middleware designed to make C/C++ programs compatible with LLMs, thereby improving their ability to identify C/C++ VCCs. CLNX operates with minimal resource overhead, offering efficiency advantages over pre-training methods. Extensive experiments confirm that CLNX-equipped LLMs demonstrate robust improvements in C/C++ VCCs identification, achieving new state-of-the-art performance. We anticipate that CLNX will allow developers to effectively improve the performance of LLMs in identifying VCCs of specific programming languages without additional pre-training.

Acknowledgments

This work is supported by the National Natural Science Foundation of China Under Grant Nos (62172176, 62127808, 62072200), and the National Key Research and Development Program of China Under Grant Nos (2022YFB3103400).

References

Black, P. E. 2018. A software assurance reference dataset: Thousands of programs with known bugs. *Journal of Research of the National Institute of Standards and Technology*, 123(1).

Booth, H.; Rike, D.; and Witte, G. 2013. The National Vulnerability Database (NVD): Overview. National Institute of Standards and Technology.

Boxler, D.; and Walcott, K. R. 2018. Static taint analysis tools to detect information flows. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, 46–52.

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*, 1456–1468.

Chakraborty, S.; Krishna, R.; Ding, Y.; and Ray, B. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*.

Cheng, X.; Wang, H.; Hua, J.; Xu, G.; and Sui, Y. 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3): 1–33.

Coimbra, D.; Reis, S.; Abreu, R.; Păsăreanu, C.; and Erdogmus, H. 2021. On using distributed representations of source code for the detection of C security vulnerabilities. arXiv:2106.01367.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805.

Ding, Y.; Suneja, S.; Zheng, Y.; Laredo, J.; Morari, A.; Kaiser, G.; and Ray, B. 2022. VELVET: a novel ensemble learning approach to automatically locate vulnerable statements. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 959–970. IEEE.

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

Hadi, M. U.; Qureshi, R.; Shah, A.; Irfan, M.; Zafar, A.; Shaikh, M. B.; Akhtar, N.; Wu, J.; Mirjalili, S.; et al. 2023. A survey on large language models: Applications, challenges, limitations, and practical usage. *Authorea Preprints*.

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. IEEE.

Le Bronnec, F.; Vérine, A.; Negrevergne, B.; Chevaleyre, Y.; and Allauzen, A. 2024. Exploring Precision and Recall to assess the quality and diversity of LLMs. In *62nd Annual Meeting of the Association for Computational Linguistics*.

Li, Z.; Zou, D.; Xu, S.; Chen, Z.; Zhu, Y.; and Jin, H. 2021a. VulDeeLocator: A deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*, 19(4): 2821–2837.

Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; and Chen, Z. 2021b. SysEVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4): 2244–2258.

Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; and Zhong, Y. 2018. VulDeepecker: A deep learning-based system for vulnerability detection. arXiv:1801.01681.

Liu, P.; Yuan, W.; Fu, J.; Jiang, Z.; Hayashi, H.; and Neubig, G. 2023a. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9): 1–35.

Liu, S.; Wu, B.; Xie, X.; Meng, G.; and Liu, Y. 2023b. Contrabert: Enhancing code pre-trained models via contrastive learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2476–2487. IEEE.

Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; and Stoyanov, V. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692.

Lu, S.; Guo, D.; Ren, S.; Huang, J.; Svyatkovskiy, A.; Blanco, A.; Clement, C.; Drain, D.; Jiang, D.; Tang, D.; et al. 2021. CodexGlue: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664.

Meneely, A.; Srinivasan, H.; Musa, A.; Tejeda, A. R.; Mokary, M.; and Spates, B. 2013. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 65–74. IEEE.

Okun, V.; Delaitre, A.; Black, P. E.; et al. 2013. Report on the static analysis tool exposition (SATE) IV. *NIST Special Publication*, 500(297).

Perl, H.; Dechand, S.; Smith, M.; Arp, D.; Yamaguchi, F.; Rieck, K.; Fahl, S.; and Acar, Y. 2015. VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 426–437.

Qin, Z.; Wu, Y.; and Han, L. 2024. CLNX: Bridging Code and Natural Language for C/C++ Vulnerability-Contributing Commits Identification. arXiv:2409.07407.

Racordon, D. 2021. From ASTs to Machine Code with LLVM. In *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming*, 68–76.

Radiya-Dixit, E.; and Wang, X. 2020. How fine can fine-tuning be? learning efficient language models. In *International Conference on Artificial Intelligence and Statistics*, 2435–2443. PMLR.

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

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. IEEE.

Sanh, V.; Debut, L.; Chaumond, J.; and Wolf, T. 2019. DistilBERT, A Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter. arXiv:1910.01108.

Steenhoek, B.; Rahman, M. M.; Jiles, R.; and Le, W. 2023. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2237–2248. IEEE.

Synopsys. 2023. 2023 Open Source Security and Risk Analysis Report. https://www.blackduck.com/content/dam/black-duck/en-us/reports/rep-ossra-2023.pdf. Accessed: 2023-12-15.

Thapa, C.; Jang, S. I.; Ahmed, M. E.; Camtepe, S.; Pieprzyk, J.; and Nepal, S. 2022. Transformer-based language models for software vulnerability detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*, 481–496.

Tian, Y.; Lawall, J.; and Lo, D. 2012. Identifying Linux bug fixing patches. In *2012 34th International Conference on Software Engineering (ICSE)*, 386–396. IEEE.

Wang, S.; Wang, X.; Sun, K.; Jajodia, S.; Wang, H.; and Li, Q. 2023. GraphSPD: Graph-based security patch detection with enriched code semantics. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2409–2426. IEEE.

Wang, X.; Sun, K.; Batcheller, A.; and Jajodia, S. 2019. Detecting "0-day" vulnerabilities: An empirical study of secret security patches in open-source software. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 485–492. IEEE.

Wu, B.; Liu, S.; Feng, R.; Xie, X.; Siow, J.; and Lin, S.-W. 2022. Enhancing security patch identification by capturing structures in commits. *IEEE Transactions on Dependable and Secure Computing*.

Xu, F. F.; Alon, U.; Neubig, G.; and Hellendoorn, V. J. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 1–10.

Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; and Liu, X. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 783–794. IEEE.

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

Zhou, S.; Alon, U.; Agarwal, S.; and Neubig, G. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. arXiv:2302.05527.

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. *Advances in neural information processing systems*, 32.

Zuo, F.; and Rhee, J. 2024. Vulnerability discovery based on source code patch commit mining: a systematic literature review. *International Journal of Information Security*, 1–14.