# LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models

Ahmed Lekssays[1*], Hamza Mouhcine[1*], Khang Tran[2], Ting Yu[3], Issa Khalil[1]
*[1]Qatar Computing Research Institute, [2]New Jersey Institute of Technology,*
*[3]Mohamed bin Zayed University of Artificial Intelligence*
{alekssays, hmouhcine, ikhalil}@hbku.edu.qa,
kt36@njit.edu, ting.yu@mbzuai.ac.ae
** Joint first authors with equal contribution*

## Abstract

Software vulnerabilities present a persistent security challenge, with over 25,000 new vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) database in 2024 alone. While deep learning based approaches show promise for vulnerability detection, recent studies reveal critical limitations in terms of accuracy and robustness: accuracy drops by up to 45% on rigorously verified datasets, and performance degrades significantly under simple code modifications. This paper presents LLMxCPG, a novel framework integrating Code Property Graphs (CPG) with Large Language Models (LLM) for robust vulnerability detection. Our CPG-based slice construction technique reduces code size by 67.84 to 90.93% while preserving vulnerability-relevant context. Our approach's ability to provide a more concise and accurate representation of code snippets enables the analysis of larger code segments, including entire projects. This concise representation is a key factor behind the improved detection capabilities of our method, as it can now identify vulnerabilities that span multiple functions. Empirical evaluation demonstrates LLMxCPG's effectiveness across verified datasets, achieving 15-40% improvements in F1-score over state-of-the-art baselines. Moreover, LLMxCPG maintains high performance across function-level and multi-function codebases while exhibiting robust detection efficacy under various syntactic code modifications.

## 1 Introduction

Software vulnerabilities continue to pose significant security risks, with the Common Vulnerabilities and Exposures (CVE) database reporting over 25,000 new vulnerabilities in 2024 alone [30]. Detecting these vulnerabilities early in the development life cycle is crucial for preventing security breaches and maintaining software integrity. However, despite extensive research, identifying vulnerabilities in complex codebases remains a challenging problem.

Recent approaches leveraging deep learning models have shown promise in vulnerability detection [2, 4, 24, 29]. However, these approaches face several critical limitations that hinder their practical application. First, they typically focus on function-level analysis, overlooking crucial inter-procedural dependencies and broader program context [5, 18, 19, 21, 27, 28, 43]. Second, recent comprehensive evaluations have exposed weaknesses in their reliability. In a thorough assessment, Ding et al. [5] introduced PrimeVul, a rigorously verified dataset where vulnerability labels were validated through multiple rounds of expert review and dynamic analysis. When state-of-the-art models were evaluated on this dataset, they exhibited dramatic performance degradation, with accuracy dropping by up to 45% compared to their reported results on traditional datasets. This significant performance gap suggests that these models may be learning superficial patterns rather than meaningful vulnerability indicators. This hypothesis is further supported by Risse et al. [26], who demonstrated significant performance drops when evaluating these models on datasets with simple modifications such as changes to function or parameter names. Finally, many approaches [14, 29] are constrained by small embedding models with limited context windows, restricting their ability to analyze large code segments effectively.

To address these fundamental challenges, we present LLMxCPG, a novel approach that combines Code Property Graphs (CPG) with Large Language Models (LLM) for robust vulnerability detection. Our approach systematically addresses the limitations of existing methods through its technical architecture. First, to overcome the context limitation and enable effective analysis of large codebases, LLMxCPG introduces a sophisticated CPG-based analysis that converts input code into precise vulnerability-focused code slices. This slice construction process works in three phases: i) extracting potential vulnerable execution paths using the Static Application Security Testing (SAST) tool Joern and its CPGQL query language, ii) analyzing execution paths through CPG traversal to identify code elements that interact with the execution paths, and iii) constructing focused code snippets containing only essential components related to potential vulnerabilities by applying backward slicing. Second, to ensure robust feature learning beyond superficial patterns, we leverage these focused code slices to fine-tune a large language model specifically for vulnerability detection, enabling it to learn from and identify vulnerability patterns in concise and relevant code contexts.

Our empirical analysis demonstrates how LLMxCPG successfully addresses the limitations of existing approaches. The slice construction approach achieves significant code reduction ratios, ranging from 67.84% for function-level datasets to 90.93% for multi-function codebases. By focusing the model's attention on these concise, vulnerability-relevant code segments rather than entire codebases, LLMxCPG enables more effective learning of vulnerability characteristics. This targeted learning translates directly into robust performance: unlike existing approaches that show significant degradation on high-quality datasets, LLMxCPG maintains consistent performance across both traditional datasets and rigorously verified ones. The effectiveness of our focused learning approach is reflected in substantial improvements over state-of-the-art baselines, with increases of 15%-40% in F1-score and 9-27% in Accuracy on function-level real-world vulnerability detection tasks. Moreover, while current approaches struggle with complex codebases, LLMxCPG demonstrates strong generalization capabilities on both function-level and multi-function codebases, performing effectively in scenarios where existing approaches fail to exceed random-guess performance. The system's robustness is further evidenced by its consistent detection efficacy under various syntactic modifications while preserving semantic equivalence, directly addressing the brittleness observed in current approaches.

**Contributions.** The contributions of this paper are summarized as follows:

- *Integration of LLMs with Program Analysis.* We present a novel framework that effectively combines traditional program analysis techniques (CPG) with modern large language models, creating a hybrid approach that leverages the strengths of both methodologies for improved vulnerability detection.

- *Vulnerability-Focused Slice Construction.* We introduce a sophisticated code slicing technique that leverages Code Property Graphs (CPG) to extract vulnerability-relevant code segments that capture the essential elements of potential vulnerabilities while eliminating irrelevant code.

- *Generalizability to Complex Codebases.* We demonstrate the generalizability of LLMxCPG to both unseen function-level datasets and real-world open-source software.

- *Robustness against Code Transformations.* We evaluate LLMxCPG under different code transformations and show that it maintains its detection efficacy.

- *Open-Source Code and Datasets.* All source code and datasets used in this study are open-source, supporting reproducibility, transparency, and further research in the domain of software security.

**Outline.** The remainder of this paper is organized as follows: Section 2 provides background on vulnerability detection and code property graphs. Section 3 details our LLMxCPG approach. Section 4 presents our experimental setup, evaluation methodology, and results. Section 5 discusses our results and their implications. Section 6 reviews related work, and Section 7 concludes with future research directions.

## 2 Background

### 2.1 Vulnerability Detection

Software vulnerability detection remains a critical research area in computer security, with approaches broadly categorized into static and dynamic analysis techniques. On the one hand, dynamic analysis techniques observe program behavior during execution through methods such as fuzzing, and dynamic taint tracking [23]. While these approaches provide more precise vulnerability detection by analyzing actual program execution paths, they face challenges in achieving comprehensive code coverage and handling the exponential growth of execution paths. Modern approaches increasingly leverage large language models and hybrid techniques that combine static and dynamic analysis, demonstrating improved detection capabilities while managing computational overhead [5].

Static analysis methods, on the other hand, examine program code without execution, utilizing techniques such as pattern matching, data flow analysis, and abstract interpretation to identify potential security flaws [32]. These approaches offer comprehensive coverage but often generate false positives due to the inability to verify runtime behavior. Recent advancements in machine learning-based vulnerability detection have shown promise in improving detection accuracy, though challenges remain in handling complex codebases and reducing false positives [4]. In this work, we focus on developing novel static code analysis tools powered by LLMs to enhance vulnerability detection performance on real-world code snippets.

### 2.2 Code Property Graphs

Code Property Graphs (CPGs) represent a unified approach to program analysis by merging multiple code representations into a single graph structure [39]. This representation combines abstract syntax trees (ASTs), control flow graphs (CFGs), and program dependence graphs (PDGs) into a joint data structure, enabling comprehensive analysis of code properties. The CPG preserves the syntactic structure from ASTs, control flow information from CFGs, and both control and data dependencies from PDGs, allowing complex patterns to be expressed through graph traversals. Analysts can express patterns through graph traversals using custom graph query languages. These traversals can be efficiently executed using graph databases, making it practical to analyze large-scale software projects.

Due to this advantage, CPGs have emerged as a particularly effective tool for vulnerability detection. By combining syntactic, control flow, and data dependency information in a unified representation, CPGs enable the precise formulation of vulnerability patterns through graph traversals. This comprehensive view allows security analysts to express complex vulnerability patterns that would be difficult to capture using traditional static analysis approaches. For instance, CPGs can effectively model patterns for buffer overflows by simultaneously analyzing allocation operations in the abstract syntax tree, validating control flow paths for proper bound checking, and tracking data dependencies to identify attacker-controlled input [23]. In this work, we employ Joern [39], an open-source static analysis tool that generates CPGs and supports queries over CPGs using the CPGQL language.

## 3 Methodology

### 3.1 System Overview

We propose LLMxCPG, a novel approach that capitalizes on the structural representation provided by CPGs while leveraging the sophisticated pattern recognition and generative abilities of Large Language Models (LLMs) for code vulnerability detection as depicted in Figure 1. Our approach comprises two specialized models, each fine-tuned for distinct tasks in our two-phase process. The first model, LLMxCPG-Q, focuses on the slice construction phase by generating CPGQL queries that identify potentially vulnerable execution paths within the code. These queries enable the extraction of focused, security-critical code segments. The second model, LLMxCPG-D, handles the classification phase by analyzing these extracted code slices to determine their vulnerability status, categorizing them as either Vulnerable or Safe. This dual-model architecture combines the precision of graph-based program analysis with the advanced reasoning capabilities of state-of-the-art language models, enabling more accurate and interpretable vulnerability detection compared to traditional approaches. We note that the two models LLMxCPG-Q and LLMxCPG-D are finetuned from Qwen2.5-Coder-32B Instruct and QwQ-32B-Preview, respectively (see Section 4.2 for more details). We note that, in what follows, when we refer to LLMxCPG, we are referencing the entire process, including both slice construction and vulnerability detection.

### 3.2 Slice Construction

Current vulnerability detection methods typically analyze code in its raw, unprocessed form, rather than first distilling it down to smaller, more focused code snippets [29]. This approach is problematic because vulnerable code often contains only a small fraction of lines that are actually related to the vulnerability. As a result, detection models face two key challenges: i) including codes irrelevant to vulnerabilities increases token usage, ii) struggling to discern truly relevant vulnerability patterns, often leading to models relying on spurious features [26].

An intuitive approach to deal with the spurious feature issue is to use program slicing [36]. Program slicing is a method to reduce a program into a smaller representation using data flow and control flow analysis. The outcome of this operation is called a slice, which is an independent program that faithfully represents the original program within the domain of the specified subset of behavior. More specifically, in our case, slices can be utilized to capture vulnerability behavior, minimize noise, and focus on the relevant vulnerability features.

To generate a slice, a criterion point must be selected first. In vulnerability detection, the criterion point is a line of code that contains an insecure variable or an insecure function call. A variable or function call is considered insecure when it can potentially lead to security vulnerabilities if not properly handled. This includes user-controlled input that flows into security-sensitive operations without proper validation or sanitization, functions known to be dangerous if misused (like `strcpy` in C which can cause buffer overflows), or variables that store sensitive data (like passwords or encryption keys) without appropriate protection. These insecure elements are typically identified through a combination of pattern matching against known vulnerable function signatures, taint analysis to track the flow of untrusted data, and control flow analysis to understand how variables and function parameters are used throughout the program. Once a criterion point is identified, the program slice is constructed by analyzing both its dependencies and its impacts: backward slicing captures all code elements that may influence the criterion point's behavior, while forward slicing identifies all code elements that the criterion point may affect. For both operations, the captured code elements include:

- **Data dependencies:** All variables, expressions, and statements that directly or indirectly affect the value of variables used in the criterion point, captured by tracing back the data flow.
- **Control dependencies:** Structures such as `if`, `for`, and `while` statements that determine whether the criterion point is executed, captured by tracing back the control flow.

**Challenges in Program Slicing.** Program slicing is often not straightforward, as it presents various challenges depend-

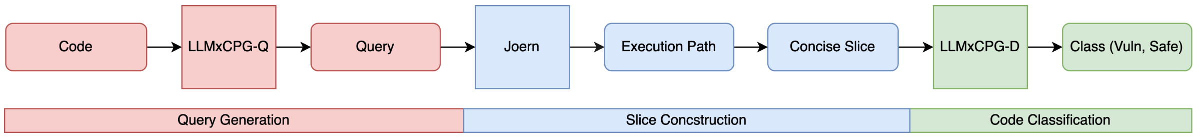
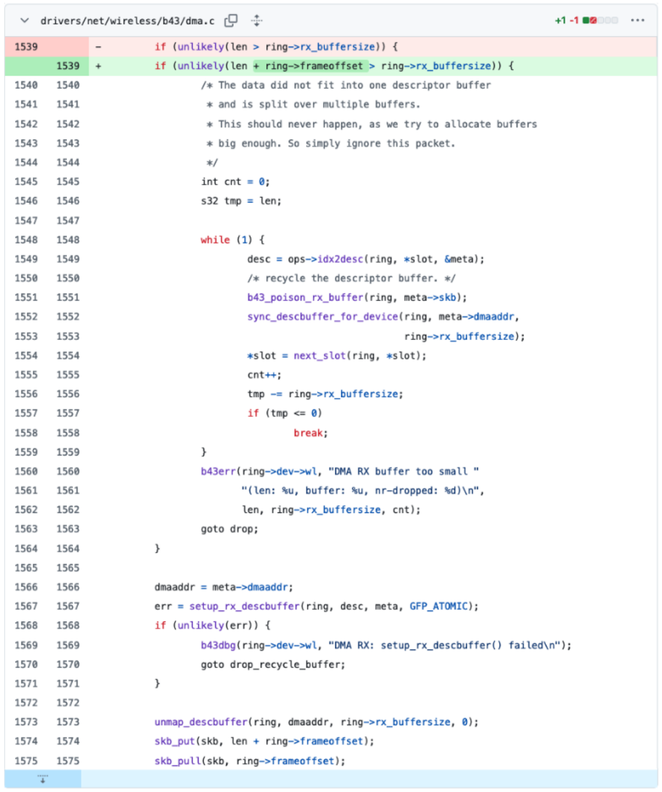
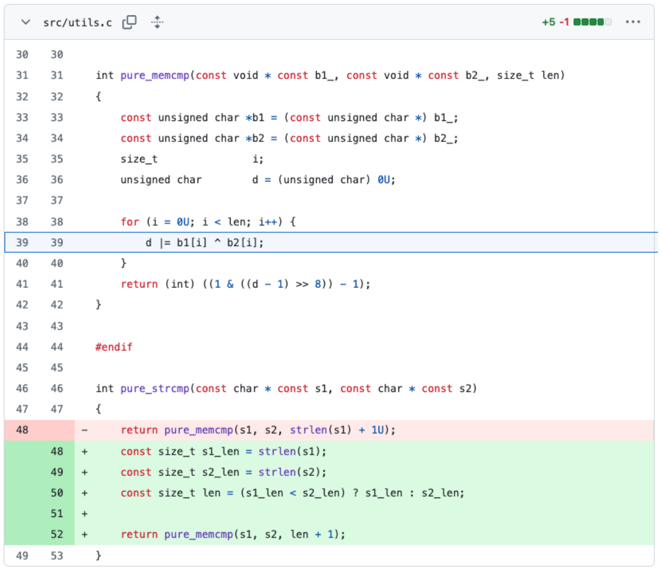


Figure 1: System Overview



```
drivers/net/wireless/b43/dma.c                                    +1 -1
1539       -    if (unlikely(len > ring->rx_buffersize)) {
     1539  +    if (unlikely(len + ring->frameoffset > ring->rx_buffersize)) {
1540 1540               /* The data did not fit into one descriptor buffer
1541 1541                * and is split over multiple buffers.
1542 1542                * This should never happen, as we try to allocate buffers
1543 1543                * big enough. So simply ignore this packet.
1544 1544                */
1545 1545               int cnt = 0;
1546 1546               s32 tmp = len;
1547 1547
1548 1548               while (1) {
1549 1549                       desc = ops->idx2desc(ring, *slot, &meta);
1550 1550                       /* recycle the descriptor buffer. */
1551 1551                       b43_poison_rx_buffer(ring, meta->skb);
1552 1552                       sync_descbuffer_for_device(ring, meta->dmaaddr,
1553 1553                                                  ring->rx_buffersize);
1554 1554                       *slot = next_slot(ring, *slot);
1555 1555                       cnt++;
1556 1556                       tmp -= ring->rx_buffersize;
1557 1557                       if (tmp <= 0)
1558 1558                               break;
1559 1559               }
1560 1560               b43err(ring->dev->wl, "DMA RX buffer too small "
1561 1561                      "(len: %u, buffer: %u, nr-dropped: %d)\n",
1562 1562                      len, ring->rx_buffersize, cnt);
1563 1563               goto drop;
1564 1564       }
1565 1565
1566 1566       dmaaddr = meta->dmaaddr;
1567 1567       err = setup_rx_descbuffer(ring, desc, meta, GFP_ATOMIC);
1568 1568       if (unlikely(err)) {
1569 1569               b43dbg(ring->dev->wl, "DMA RX: setup_rx_descbuffer() failed\n");
1570 1570               goto drop_recycle_buffer;
1571 1571       }
1572 1572
1573 1573       unmap_descbuffer(ring, dmaaddr, ring->rx_buffersize, 0);
1574 1574       skb_put(skb, len + ring->frameoffset);
1575 1575       skb_pull(skb, ring->frameoffset);
```

Figure 2: An example of a buffer overflow vulnerability from CVE-2011-3359.



```
src/utils.c                                                       +5 -1

    int pure_memcmp(const void * const b1_, const void * const b2_, size_t len)
    {
        const unsigned char *b1 = (const unsigned char *) b1_;
        const unsigned char *b2 = (const unsigned char *) b2_;
        size_t                i;
        unsigned char         d = (unsigned char) 0U;

        for (i = 0U; i < len; i++) {
            d |= b1[i] ^ b2[i];
        }
        return (int) ((1 & ((d - 1) >> 8)) - 1);
    }

    #endif

    int pure_strcmp(const char * const s1, const char * const s2)
    {
-       return pure_memcmp(s1, s2, strlen(s1) + 1U);
+       const size_t s1_len = strlen(s1);
+       const size_t s2_len = strlen(s2);
+       const size_t len = (s1_len < s2_len) ? s1_len : s2_len;
+
+       return pure_memcmp(s1, s2, len + 1);
    }
```

Figure 3: An example of a buffer overflow vulnerability from CVE-2020-9365.

ing on the specific vulnerability being addressed. First, selecting appropriate criterion points is complex: while approaches like UltraVCS [37] use predefined sets of sensitive function calls in C, developers often create custom wrappers around these functions, making them harder to detect. Other methods, such as Snopy [1] and MVP [38], attempt to identify criterion points by analyzing differences between vulnerable and patched code versions, but this approach can be unreliable as patches frequently include unrelated refactoring changes. Second, even when criterion points are correctly identified, the resulting slices often contain unnecessary code. For example, in the buffer overflow vulnerability shown in Figure 2 (CVE-2011-3359), selecting the `if` condition at line 1539 as the criterion point would include the entire `if` block in the slice, despite many of these lines being irrelevant to the vulnerability.

To address these challenges, we propose a novel approach that shifts focus from individual criterion points to execution paths. Instead of relying on predefined sensitive functions or code differences, we leverage CPGs to identify potentially vulnerable execution paths. This graph-based approach allows us to capture the essential flow of data and control while minimizing the inclusion of irrelevant code.

**Slice Construction.** Our slice construction approach works in three main steps. First, we use the SAST tool Joern and its CPGQL query language to identify potential vulnerability root causes in the code, focusing on execution paths rather than analyzing the entire codebases that span multiple functions or files. Second, we analyze each execution path by traversing the CPG to identify variables that interact with that path. Third, we build our final slice by gathering all code elements that influence both the execution path and its interacting variables. The result is a focused code snippet that contains only the essential components: the execution path itself, the variables that interact with it, and any code that affects either of these elements.

### 3.2.1 Taint Path Extraction

For taint path extraction, we employ CPGQL, a specialized query language designed for analyzing code property graphs

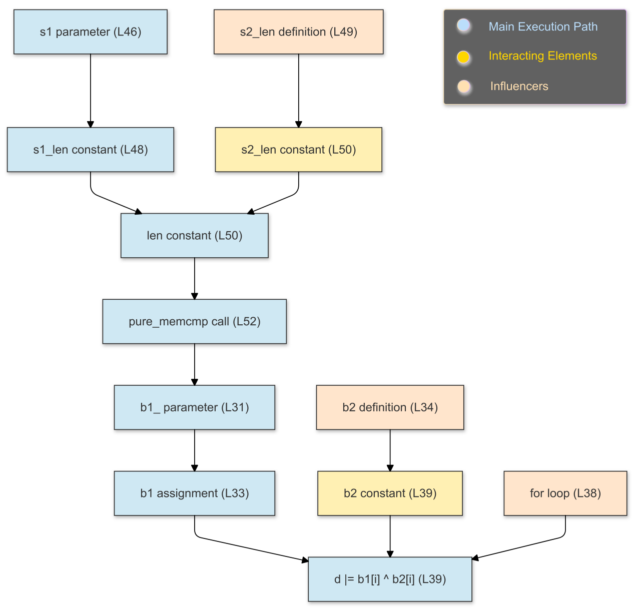


Figure 4: Slice extraction on CVE-2011-3359.

```
val source = cpg.identifier.name("len")
val sink = cpg.call.name("skb_put").where(_.
    argument.order(2).codeExact("len + ring->
    frameoffset"))
val execution_paths = sink.reachableByFlows(
    source)
```

Listing 1: Example CPGQL queries to identify sources, sinks, and execution paths.

in Joern. CPGQL facilitates the navigation and analysis of code property graphs via queries targeting specific code patterns. For instance, the following query identifies all function calls within a method named `processData`:

```
cpg.method.name("processData").call.name
```

A critical step in LLMxCPG is the design of CPGQL queries to extract execution paths relevant to the target vulnerabilities. For example, in Listing 1, the first query identifies the source of the vulnerability (the identifier `len`), while the second locates the sink (a call to `skb_put`), which might fail without proper buffer size checks. The final query identifies all execution paths between the source and sink.

**Fine Tuning for Query Generation.** The goal of this step is to to have a model that can generate valid CPGQL queries that target a specific vulnerability pattern depending only on the provided code snippet with no additional information such as CWE type, or vulnerability location. While powerful language models like DeepSeek, ChatGPT, and Qwen excel at general code generation, they initially struggle with generating effective CPGQL queries since CPGQL is a low-resource language. However, since CPGQL is based on Scala, we leverage this similarity to fine-tune the Qwen2.5-Coder-32B-Instruct model (currently the best code model available) to generate CPGQL queries. To create our training data, we utilize DeepSeek-v3 to generate an initial set of queries. We generate and test queries on a Joern server, Feedback is provided to the model whenever queries contain syntax errors or fail to identify vulnerable paths, enabling iterative refinement and improvement. Henceforth, we will refer to the query generation fine-tuned model as LLMxCPG-Q.

### 3.2.2 Interacters: Finding Variables that Interact with the Execution Path

As shown in Figure 4, each extracted path represents a specific flow of data and control through the code. For example, in the CVE-2020-9365 vulnerability (Figure 3), LLMxCPG identifies a key execution path (shown in Blue Boxes). However, this execution path alone does not provide a complete vulnerability assessment, as it lacks the definition of the `s2` constant. Without this contextual information, determining the security implications of this execution path is not feasible. To build a comprehensive understanding, we must identify all code elements that interact with this path. We achieve this by navigating the CPG using the query language to find all interacting code elements. Listing 2 shows two queries. The first query, generated by LLMxCPG-Q, captures potential vulnerable execution paths. The second query identifies all identifiers that interact with the captured execution path. An identifier is considered an interacter if its line number matches the line number of at least one code element in the execution path.

```
val execution_path_nodes = <query to extract the
    execution path generated by LLMxCPG-Q>
cpg.identifier.filter(id => execution_path_nodes.
    lineNumber.toSet.intersect(id.lineNumber.l.
    toSet).size.equals(1))
```

Listing 2: CPGQL query to identify nodes that interact with the extracted execution path.

### 3.2.3 Backward Slicing for Focused Code Snippet Construction

In the final step, we use backward slicing to build a complete code snippet that includes both the execution path and its interacting elements. As illustrated in Figure 4, this process captures all essential dependencies, such as the `s2_len` definition, the `b2` string initialization, and the relationship between the `for loop` and the `len` variable. This comprehensive slice provides all the context needed to understand the potential vulnerability. We note that we automate the process of applying a backward slice using a CPGQL query. Listing 3 shows the query used to perform backward slicing. The `reachableByFlows` API identifies all code elements that influence either the execution path or its interacters. Internally, Joern utilizes the Program Dependency Graph (PDG) to construct the backward slice.

```
... queries to extract execution path and the
    interacters.
execution_path_and_interacters.reachableByFlows(
    cpg.all)
```

Listing 3: CPGQL query to apply backward slicing.

## 3.3 Vulnerability Detection

The slice construction process reduces code samples that span multiple functions and multiple files to concise snippets of code that better represent the characteristics of the vulnerabilities. For example, the 85 lines of code function in Figure 2, can be compressed to an 18-line code function by using CPGQL queries as shown in Listing 4. This path extraction approach significantly enhances vulnerability detection by isolating the security-relevant code patterns specific to each CWE type while eliminating non-essential code. By focusing on the critical data and control flow paths that characterize potential vulnerabilities (such as source-to-sink paths for taint-style vulnerabilities or validation-check patterns for input handling flaws), this method minimizes noise that would otherwise obscure vulnerability signatures. We show the overall workflow to classify code as *Vulnerable* or *Safe* in Figure 6.

**Fine Tuning for Classification.** As previously noted, we fine-tuned the QwQ-32B-Preview model for the classification task. To construct the fine-tuning dataset for classifying code slices as either vulnerable or safe, we extracted code snippets from both vulnerable and safe samples in the training dataset using LLMxCPG-Q given the groundtruth labels in our training datasets. To this end, the fine-tuned classifier model will be referred to as LLMxCPG-D.

Note that this approach is equally effective for compressing safe code samples. For instance, in the previously mentioned example in Figure 2, the patched version of the code involves only a single change at line 1539, where the if condition was modified to ensure sufficient buffer size. Applying the same CPGQL query-based approach to this patched version reduces noise and highlights the specific security-relevant modification. The ability to precisely identify and isolate security-critical changes between vulnerable and patched versions makes LLMxCPG particularly valuable for understanding vulnerability fixes and generating high-quality training data for vulnerability detection models. In fact, since high-quality datasets in the vulnerability detection domain are scarce, we plan to use our approach in future work to compile an open-source, large, and high-quality dataset for training vulnerability detection models.

```c
static void dma_rx(struct b43_dmaring *ring,
    int *slot)
{
    u16 len;
    len = le16_to_cpu(rxhdr->frame_len);
    if (unlikely(len > ring->rx_buffersize)) {
        s32 tmp = len;
        while (1) {
            tmp -= ring->rx_buffersize;
            if (tmp <= 0)
                break;
        }
        goto drop;
    }

    skb_put(skb, len + ring->frameoffset);
    drop:
    return;
}
```

Listing 4: CPGQL captured path represented as a code snippet.

## 4 Evaluation

This section presents the employed datasets and shows details of our implementation. In addition, we provide a detailed analysis of LLMxCPG performance on both function-level and project-level datasets. Moreover, we analyze its ability to generalize to unseen datasets and its robustness against code transformations as defined in [26].

## 4.1 Datasets

**Training Datasets.** We used two datasets for training, FormAI-v2 [33] and PrimeVul [5]. The FormAI-v2 dataset includes 331,000 compilable C programs generated using various LLMs including Google's GEMINI-pro, OpenAI's GPT-4, TII's 180 billion-parameter Falcon, CodeLLama2, and other compact models. These programs are generated using a dynamic zero-shot prompting technique and comprise programs with varying levels of complexity. Each program is labeled for code vulnerabilities using a formal verification method based on the Efficient SMT-based Bounded Model Checker (ESBMC) [10]. FormAI minimizes false negatives by ensuring comprehensive formal verification of the code within a defined timeframe. We note that FormAI C programs are not directly mapped to CWEs, but rather correspond to errors returned by ESMBC. To address this, we manually created a mapping of error messages (provided by ESBMC) to CWEs. On the other hand, the PrimeVul dataset was created to address the shortcomings of existing vulnerability datasets, such as poor data quality, low label accuracy, and high duplication rates. PrimeVul employs novel data labeling techniques,

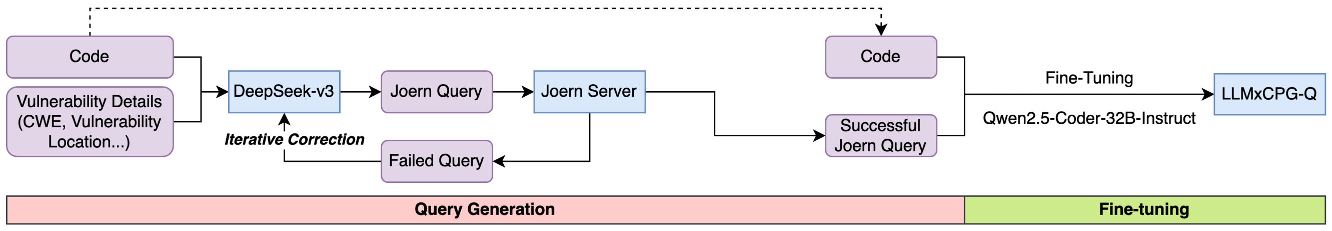


Figure 5: Query Generation Workflow

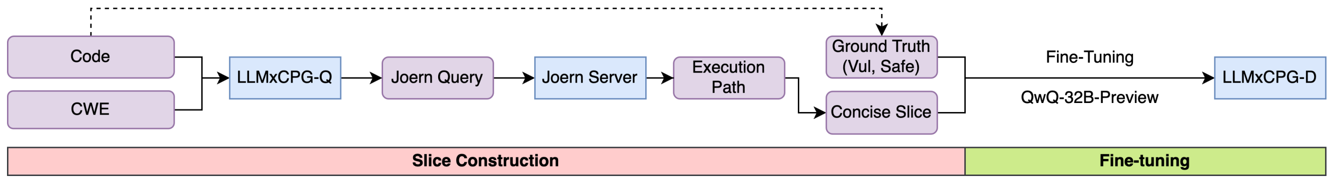


Figure 6: Code Classification Workflow

achieving a label accuracy comparable to human-verified benchmarks, while significantly expanding the dataset. The dataset implements rigorous data de-duplication and chronological data splitting to avoid data leakage issues. The dataset contains 228,800 safe functions and 6,968 vulnerable functions covering 140 CWEs, making it a diverse and accurate resource for vulnerability detection research. The dataset is split chronologically based on commit dates, with 80% for training, 10% for validation, and 10% for testing.

**Generalizability.** To evaluate generalizability, we utilize two distinct vulnerability datasets: SVEN [15] and ReposVul [35]. SVEN is a manually curated dataset of approximately 1,600 C/C++ and Python programs derived from real-world GitHub security fixes, with a rigorous verification process to ensure high data quality and relevance to security hardening. ReposVul complements this with a broader repository-level perspective, encompassing 6,134 CVE entries across 236 CWE types from 1,491 projects in four programming languages. Together, these datasets provide a comprehensive foundation for assessing vulnerability detection capabilities across different real-world scenarios.

**Studied CWEs.** Our analysis focuses on a specific subset of memory-related CWEs that are amenable to static analysis via Code Property Graphs (CPGs): CWE-119 (Buffer Overflow), CWE-190 (Integer Overflow), CWE-415 (Double Free), and CWE-416 (Use After Free). Additionally, we include CWE-120 (Buffer Copy without Checking the Size of Input), CWE-121 (Stack-based Buffer Overflow), CWE-122 (Heap-based Buffer Overflow), CWE-125 (Out-of-bounds Read), and CWE-787 (Out-of-bounds Write), which are specialized variants of CWE-119. This selection criterion balances the prevalence of real-world vulnerabilities with the technical constraints of CPG-based detection. We specifically excluded vulnerabilities that rely on dynamic program behavior, such as race conditions, as these cannot be reliably modeled through static CPG analysis due to their runtime-dependent nature.

**CWE Distribution.** The CWE distribution varies across our datasets, as illustrated in Table 1. FormAI-v2's training set contains 5,893 vulnerable samples across four CWE types (CWE-119, CWE-190, CWE-415, and CWE-416), with relatively balanced distribution ranging from 1,395 to 1,500 samples per CWE, alongside 4,431 safe samples. It is worth noting that the FormAI-v2 dataset was validated using ESBMC with default memory violation assertions and does not provide paired vulnerable–patched code samples, resulting in non-indication of safe samples per CWE. In other words, the safe samples were formally verified to be free from memory violations under the default ESBMC configuration. To this end, we exclude samples with unknown or timed-out verification results, as these do not conclusively demonstrate the presence or absence of a vulnerability. Regarding CWE mapping for the safe samples, this does not apply to FormAI-v2 because, unlike real-world cases where vulnerable and patched versions of the same code are available, the code snippets in FormAI-v2 are generated independently. PrimeVul's training data spans eight CWE types, with CWE-119 being the most prevalent (518 samples) and some CWEs having limited representation (e.g., CWE-121 and CWE-122 with only 1-2 samples). For our test datasets, we ensure balanced representation where possible. SVEN's test set contains four CWE types (CWE-125, CWE-190, CWE-416, and CWE-787) with paired vulnerable and safe samples ranging from 37 to 122 pairs per CWE. Similarly, we select a balanced subset from ReposVul, including matched safe and vulnerable samples across seven CWE types.

## 4.2 Implementation

Our implementation has three main components: slice construction, fine-tuning, and inference.

Table 1: CWE Distribution Across Datasets

| **CWE** | **Training Datasets** | | | **Test Datasets** | | | | |
|---|---|---|---|---|---|---|---|---|
| | FormAI | PrimeVul | Total | SVEN | ReposVul | FormAI | PrimeVul | Total |
| CWE-119 | 1,395/NA | 518/518 | 1,913/518 | – | 19/19 | 51/NA | – | 70/19 |
| CWE-120 | – | 35/35 | 35/35 | – | 4/3 | – | –/2 | 4/5 |
| CWE-121 | – | 1/1 | 1/1 | – | 1/1 | – | – | 1/1 |
| CWE-122 | – | 2/2 | 2/2 | – | 2/– | – | – | 2/– |
| CWE-125 | – | 391/391 | 391/391 | 122/122 | 13/19 | – | 9/6 | 144/147 |
| CWE-190 | 1,500/NA | 138/138 | 1,638/138 | 37/37 | 4/3 | 41/NA | 11/12 | 93/52 |
| CWE-415 | 1,499/NA | 49/49 | 1,548/49 | – | 4/3 | 50/NA | 8/15 | 62/18 |
| CWE-416 | 1,499/NA | 176/176 | 1,675/176 | 56/56 | 13/12 | 43/NA | 12/5 | 124/73 |
| CWE-787 | – | – | – | 44/44 | – | – | – | 44/44 |
| Total Vulnerable | 5,893 | 1,310 | 7,203 | 259 | 60 | 185 | 40 | 544 |
| Total Safe | 4,431 | 1,310 | 5,741 | 259 | 60 | 198 | 40 | 557 |

Note: For the PrimeVul, SVEN, and ReposVul datasets, numbers are shown as vulnerable/safe pairs. '–' indicates absence of the CWE type. For FormAI dataset, the samples are not paired, thus only the total safe samples are stated.

**Slice Construction.** In this step, we train the CPGQL query generation model using a training dataset of valid Joern queries. To construct these training queries, we leverage DeepSeek v3 to generate potential candidates, which are then validated using Joern, as illustrated in Figure 5. Failed queries are returned to DeepSeek along with the error message generated by Joern for up to two additional attempts. If the query remains invalid after three attempts, it is discarded. We deploy a cluster of Joern servers using Docker containers to enable parallel processing of multiple repositories.

**Finetuning.** After extensive evaluation of state-of-the-art language models including Phi-4, Qwen2.5-Coder-32B-Instruct, and Codestral 22B, we selected QwQ-32B-Preview as our base model for fine-tuning LLMxCPG-D and Qwen2.5-Coder-32B-Instruct for finetuning LLMxCPG-Q (see Appendix B for more details). For both models, the fine-tuning process employs Low-Rank Adaptation (LoRA) [16] using the LLaMA-Factory[1] framework. We configure the LoRA parameters with rank 8 and alpha 4, while setting the learning rate to $10^{-4}$. The fine-tuning is performed on an NVIDIA A100-80GB GPU running Ubuntu 22.04, allowing us to effectively adapt the pre-trained model to the specific task of vulnerability detection.

**Inference Pipeline.** For query inference, we employ the inference library vLLM[2] with a CPGQL generation prompt shown in Appendix E (Figure 8). For classification inference, we employ Unsloth[3], a library for efficient finetuning and inference. We implement an optimized pipeline that reduces the LLMxCPG-D's language modeling head to focus exclusively on binary classification between vulnerable and safe code samples. Specifically, we extract the weight vectors corresponding to the *"Yes"* (i.e., Vulnerable) and *"No"* (i.e., Safe) tokens from the original `lm_head`, constructing a reduced classification head. During inference, LLMxCPG-D generates logits for these two tokens, which are then passed through a softmax function to obtain prediction probabilities. This approach significantly reduces the computational overhead compared to full token generation while maintaining classification accuracy. A threshold $\gamma$ is applied to these probabilities to determine the final classification.

[1] `https://github.com/hiyouga/LLaMA-Factory`
[2] `https://github.com/vllm-project/vllm`
[3] `https://docs.unsloth.ai`

The classification pipeline can be formally described as follows: Let $w_v$ and $w_b$ represent the weight vectors corresponding to *Vulnerable* and *Safe* tokens respectively, extracted from the original language model head $W_{\text{lm}}$. The classification logits $\mathbf{l}$ for an input code sample $x$ are computed as:

$$\mathbf{l} = [l_v, l_b] = f_\theta(x) \cdot [w_v, w_b]^T$$

where $l_v$ is the logit corresponding to *Vulnerable*, $l_b$ is the logit corresponding to *Safe*, and $f_\theta(x)$ represents the model's output. These logits are transformed into probabilities through the softmax function:

$$p(y \mid x) = \text{softmax}(\mathbf{l}) = \left[ \frac{e^{l_v}}{e^{l_v} + e^{l_b}}, \frac{e^{l_b}}{e^{l_v} + e^{l_b}} \right]$$

The final classification is determined by comparing the vulnerability probability against a dataset-specific threshold $\gamma$:

$$\hat{y} = \begin{cases} 1 & \text{if } p(y = \text{vulnerable} \mid x) > \gamma \\ 0 & \text{otherwise} \end{cases}$$

The threshold $\gamma$ can be predefined for the system as a hyper-parameter. Furthermore, it can be selected by security analysts based on their small validation set of known vulnerabilities, which is practical and facilitates system adaptation without extensive manual labeling. Our empirical analysis, shown in Figure 7 revealed that threshold calibration is essential for optimal performance across different datasets. We determine distinct threshold values for each dataset in our study through validation splits. Notably, we observe higher threshold values for datasets used in training unlike unseen datasets. This pattern suggests the model expresses higher confidence when evaluating code patterns similar to its training distribution. For users applying our model to new datasets, we recommend calibrating the threshold using a small random sample (as few as 20 data points) of labeled examples from their target domain. We use the following thresholds after sampling 20 data points from validation splits of the employed datasets: PrimeVul $\gamma = 0.594$, FormAI: $\gamma = 0.547$, SVEN: $\gamma = 0.334$, and ReposVul: $\gamma = 0.193$. We show the effect of vaying thresholds on LLMxCPG-D's accuracy in Appendix D.

## 4.3 Performance Analysis

### 4.3.1 Query Generation

As shown in Table 2, LLMxCPG-Q effectively learns the syntax of CPGQL, whereas the base models, Qwen2.5-Coder-32B-Instruct and DeepSeek-v3, encounter difficulties in generating valid CPGQL queries. To further investigate the quality differences among the studied models, we randomly selected 50 samples where DeepSeek generated invalid queries to analyze what LLMxCPG-Q had successfully learned. We classify the invalid queries into the following categories:

Table 2: Number of valid queries per model out of 1278 test samples.

| **Model** | Number of valid queries |
|---|---|
| DeepSeek-v3 | 132 |
| Qwen2.5-Coder-32B-Instruct | 19 |
| LLMxCPG-Q | 1278 |

- **Incorrect usage of CPGQL APIs:** DeepSeek often misuses the `.code` API to filter nodes by their name. For instance, to retrieve all `Call` nodes with the name `"print"`, DeepSeek generates the query `cpg.call.code("print")`. However, this query returns empty results because the `code` property matches the entire statement, including the arguments of the function. The correct query for this scenario is `cpg.call.name("print")`, which matches only the function's name.

- **Missing or incorrect API usage:** DeepSeek mixes node types with operations allowed on each type. For example, it generated the query `val inputSources = cpg.call.code("scanf").argument.filter(_ .typeFullName.matches("float")).toList`, where `typeFullName` is incorrectly applied to `argument` nodes.

- **Incorrect handling of regex-supporting APIs:** Some CPGQL filtering APIs support regular expressions, but DeepSeek fails to distinguish between these APIs. For example, it may use `cpg.call.code('a + b')` to search for an addition operation, which fails due to regex interpretation. The correct query for this case is `cpg.call.codeExact('a + b')`, which performs an exact string match.

While our evaluation demonstrates that LLMxCPG-Q successfully learns the intricacies of CPGQL syntax—including proper API usage and regex handling—syntactic correctness alone is insufficient for real-world applications. A truly effective system must also maintain semantic precision, ensuring the generated queries accurately isolate vulnerability patterns within the code.

To assess this semantic dimension, we conducted a rigorous human-auditing process involving three security experts who manually evaluated 50 queries generated by LLMxCPG-Q. These queries were sampled from the PrimeVul and SVEN test datasets and included 25 true positive/negative and 25 false positive/negative cases, where the ground truth labels (true/false positive/negative) were determined by the final classification decisions of LLMxCPG-D on the constructed slices (e.g., our evaluation considers a case as positive if LLMxCPG-D correctly labels the constructed slice, even when the generated query targets a different CWE than the actual vulnerability's CWE type). Each expert assessed whether the queries are semantically aligned with the intended vulnerability patterns and successfully captured execution paths that isolate the vulnerabilities with minimal noise. In 76% of the true positive/negative samples, the queries matched the vulnerability pattern semantically and yielded meaningful paths. The experts achieved a Fleiss' Kappa [8] score of 0.6429 across the 25 true positive/negative samples, indicating substantial inter-rater agreement according to the guidelines provided by Landis and Koch [17]. Discrepancies were resolved through discussion. To better understand the limitations, we analyzed the 25 false positive/negative samples and categorized them as follows:

- 28% of the queries were semantically correct and accurately identified the vulnerability pattern, despite being labeled incorrectly.

- 40% of the queries targeted a different CWE, failing to capture the correct vulnerability.

- 32% correctly identified the CWE but missed critical contextual elements necessary for a complete vulnerability match.

Although LLMxCPG-D misclassifies certain cases even when a vulnerability is present in the constructed code slice (28% of false positives/negatives and 24% of true positives/negatives), most of these errors occur when the query model fails to detect the target vulnerability pattern.

In Appendix C, we evaluate the performance of LLMxCPG-D on full code snippets (omitting the slicing step). We then run Joern-scan[4], a tool that executes predefined queries targeting various vulnerability patterns, on a subset of the test set to gauge the impact of the slice construction and query generation steps.

**Code Reduction.** Our empirical analysis reveals significant code reduction ratios achieved through our slice construction approach across multiple datasets. In the synthetic FormAI dataset, we observe an average reduction of 78.70% in code size on the test set. Similar efficiency is demonstrated in real-world scenarios, with code reductions of 67.84% and 70.22% in the function-level datasets PrimeVul and SVEN respectively, and a substantial 90.93% reduction in the project-level dataset ReposVul. In the subsequent section, we conduct a comprehensive analysis of LLMxCPG's performance on these reduced code slices to examine whether the system effectively capitalizes on these significant reductions while maintaining vulnerability detection capabilities.

### 4.3.2 Function-level Vulnerability Detection

Using the FormAI and PrimeVul datasets, we evaluate LLMxCPG's performance in detecting vulnerabilities across diverse

[4]https://docs.joern.io/scan/

function-level code snippets. These datasets provide a comprehensive testbed, capturing various vulnerability types and coding practices.

Table 3: Average Performance of LLMxCPG on PrimeVul & FormAI datasets

| Dataset | Accuracy | Precision | Recall | F1-score |
|---|---|---|---|---|
| FormAI | 0.8146 | 0.8097 | 0.8054 | 0.8075 |
| PrimeVul | 0.7250 | 1.0 | 0.45 | 0.6206 |

Table 3 illustrates the results of LLMxCPG. In general, LLMxCPG achieves high performance on vulnerability detection in function-level code snippets. Specifically, LLMxCPG reaches up to 0.8146 Accuracy and 0.8075 F1-score on the FormAI dataset. Similar results are observed on the PrimeVul dataset, which indicates the effectiveness of LLMxCPG in vulnerability detection at the function level.

Table 4: A Breakdown of the Performance of LLMxCPG by CWEs on PrimeVul & FormAI datasets

| **CWE** | **Accuracy** | **Precision** | **Recall** | **F1-Score** |
|---|---|---|---|---|
| CWE-119 | 0.941 | 1.000 | 0.941 | 0.970 |
| CWE-415 | 0.757 | 0.891 | 0.817 | 0.852 |
| CWE-416 | 0.778 | 1.000 | 0.736 | 0.848 |
| CWE-190 | 0.672 | 0.844 | 0.745 | 0.792 |

Table 4 illustrates the performance of LLMxCPG with respect to different CWEs. LLMxCPG exhibited strong performance on several memory-related CWEs. Specifically, for CWE-119, the model achieved significantly high accuracy and F1-scores (0.941 and 0.97, respectively). Similar results are also observed in other CWEs.

## 4.4 Generalizability

### 4.4.1 Function-level Vulnerability Detection

This section assesses the generalizability of LLMxCPG for detecting vulnerabilities in function-level code snippets. To provide a comprehensive evaluation, we benchmark its performance on real-world code snippets against a range of state-of-the-art baselines, including VulSim [29], ReGVD [25], and both VulBERTA-CNN and VulBERTA-MLP [14]. VulSim combines the structural and semantic information similarity of a function-level code snippet with the snippets in the training datasets to classify whether it is vulnerable or safe. Similarly, VulBERTA models incorporate the CodeBERT transformers model to extract the semantics of the snippets to make classifications. In addition, ReGVD leverages the graph structural information of the code snippets to classify whether it is vulnerable or not. To ensure a fair comparison, we employ SVEN dataset as it was not part of the training data of LLMxCPG and the considered baselines.

Table 5: Function-level vulnerability detection average performance on SVEN dataset, which includes CWE-125, CWE-190, CWE-416, CWE-476.

| | Accuracy | Precision | Recall | F1-score |
|---|---|---|---|---|
| VulSim [29] | 0.33 | 0.31 | 0.31 | 0.31 |
| VulBERTA-CNN [14] | 0.5 | 0.51 | 0.38 | 0.44 |
| VulBERTA-MLP [14] | 0.5 | 0.5 | 0.37 | 0.43 |
| ReGVD [25] | 0.51 | 0.53 | 0.46 | 0.55 |
| LLMxCPG | **0.6020** | **0.5590** | **0.9534** | **0.7048** |

Table 5 presents the vulnerability detection performance of LLMxCPG on the SVEN dataset, offering insights into its effectiveness on previously unseen code snippets. The results indicate that LLMxCPG significantly outperforms state-of-the-art baselines in detecting vulnerabilities. In particular, LLMxCPG achieves a remarkable 20% improvement in accuracy over the competing approaches. This substantial margin highlights its ability to generalize across different code snippets and quickly uncover vulnerabilities. Such performance underscores the robustness of LLMxCPG in scenarios where training data does not directly overlap with the test set, further solidifying its value in practical, real-world applications. The superior performance of LLMxCPG can be attributed to its key design features. One of the critical factors is its integration of CPGs, which allows it to understand the semantics and structure of the code. Additionally, LLMxCPG leverages specialized LLMs, enabling it to effectively capture vulnerability signatures and adapt to unseen scenarios.

### 4.4.2 Project-level Vulnerability Detection

Project-level vulnerability detection presents fundamentally different challenges compared to function-level analysis in terms of code complexity. To comprehensively evaluate the model's performance, we consider five complexity metrics: Lines of Code (LOC), Cyclomatic Complexity (CC), Number of Functions, Number of Branches, and Nesting Depth. These metrics collectively capture different dimensions of code complexity, from pure size (LOC) to structural intricacy (CC, Nesting) and modularity (Functions, Branches). We formally define these metrics in Appendix A.

We evaluate LLMxCPG on a sampled dataset from ReposVul [35], despite it not being trained on project-level real-world data. The sampled dataset from ReposVul [35] comprises 120 balanced samples from 53 real-world projects with complex code snippets spanning multiple files and multiple functions as evidenced by its code metrics statistics shown in Table 6. On average, each file contains 659 lines of code (LOC), 102.91 branches, 9.28 functions, and a nesting depth of 4.62 levels. These metrics highlight the increased complexity of project-wide vulnerability detection compared to function-level analysis. For instance, while the average number of functions per file is 9.28, a single file may contain up to 59 functions interacting across 335 branch points, substantially exceeding the complexity seen in isolated function analysis. Additionally, the high average nesting depth (4.62 levels, with a maximum of 11) underscores the challenge of identifying vulnerabilities that may emerge from deeply nested and interdependent code structures. Such complexity demonstrates the inadequacy of state-of-the-art models limited to function-level detection. These models fail to capture the dependencies and interactions that span multiple functions and deeply nested branches within a project. The ReposVul dataset thus provides a challenging benchmark for advancing project-wide vulnerability detection models capable of addressing these challenges.

Table 6: Code Metrics Statistics of the Sampled ReposVul Dataset

| Metric | Mean | Min | Max |
|---|---|---|---|
| LOC | 659.06 | 54.00 | 1951.00 |
| CC | 101.71 | 3.00 | 436.00 |
| Functions | 9.28 | 0.00 | 59.00 |
| Branches | 102.91 | 0.00 | 335.00 |
| Nesting | 4.62 | 2.00 | 11.00 |

**Performance Analysis on ReposVul dataset.** Despite these challenges, LLMxCPG achieves promising results with an average Accuracy of 0.634 and an F1-score of 0.610 on ReposVul. Detailed analysis across complexity metrics reveals interesting patterns. Performance is not affected by the LOC as LLMxCPG achieves 0.83 accuracy on samples with extremely high LOC (1623 mean), demonstrating robustness to code size variations. It shows strong performance on samples with high cyclomatic complexity (0.75 Accuracy for CC 115-150), indicating effective handling of complex control flows. In addition, detection capability is maintained even as the number of functions increases (0.71 Accuracy for 13-14 functions), suggesting successful modeling of inter-function dependencies. Notably, the model maintains consistent performance even in the highest complexity bins across multiple metrics, with no significant degradation on complex samples. For instance, it achieves 0.75 accuracy on samples with high CC (196 mean). This stability across complexity metrics demonstrates the model's ability to handle realistic project-level codebases without being overwhelmed by increased complexity.

**Performance Analysis on Post-Knowledge-Cutoff CVEs.** To evaluate our model's generalization capabilities on emerging vulnerabilities, we compiled a balanced dataset comprising 60 samples from CVEs published in 2025. To construct this dataset, we crawled the NVD for CVEs published between January 1, 2025, and May 12, 2025, filtering for those associated with the following CWEs: CWE-119, CWE-120, CWE-121, CWE-122, CWE-125, CWE-190, CWE-415, CWE-416, and CWE-787. We focused exclusively on CVEs with public references linking to GitHub or GitLab commits. Where applicable, we also resolved repositories through known official mirrors. For instance, while the Linux kernel's primary codebase is hosted at git.kernel.org, it has an official mirror at github.com/torvalds/linux. In total, we crawled 1,583 CVEs. Of these, 1,194 did not include a Git commit in their public references. Among the remaining 389 CVEs, only 121 had valid CWE tags (i.e., not labeled as NVD-CWE-noinfo). Out of those, only 81 belonged to one of the CWEs listed in Table 1, namely CWE-190, CWE-416, and CWE-125. We included CWE-125 despite our suboptimal performance on it due to limited training samples. Among the 81 relevant CVEs, only 57 had commit files that fit within the input context of our model (32k tokens, with potential extension to 128k tokens given additional resources, see Sec. 5 for more details). Each of these 57 CVEs is paired with its corresponding patch, resulting in a total of 114 samples, balanced across the *Vulnerable* and *Safe* labels in our final dataset.

On this dataset, LLMxCPG-D achieved an F1-score of 0.617 and Accuracy of 0.600, comparable to its performance on the ReposVul dataset. This demonstrates the model's robust generalization to novel vulnerability patterns that emerged after its knowledge cutoff, suggesting effective learning of fundamental vulnerability characteristics rather than mere memorization of known CVE instances. Table 7 shows our performance on ReposVul and Post-Knowledge-Cutoff datasets.

Table 7: Average Performance of LLMxCPG on ReposVul & 2025 Post-Knowledge-Cutoff (PKCO-25) datasets

| Dataset | Accuracy | Precision | Recall | F1-score |
|---|---|---|---|---|
| ReposVul | 0.634 | 0.542 | 0.700 | 0.610 |
| PKCO-25 | 0.600 | 0.592 | 0.644 | 0.617 |

**Limitations.** During our analysis, we notice a performance degradation on few samples with very high nesting depth (>7 levels), dropping to 0.33 accuracy. While the model maintains reasonable performance across function count increases, the accuracy variance (0.33-0.84) suggests room for improvement in modeling some extremely complex inter-function relationships. These results represent a significant step toward practical vulnerability detection at the project level, though they also highlight specific areas where current approaches can be enhanced. LLMxCPG's ability to maintain consistent performance across most complexity metrics, despite not being trained on project-level samples, demonstrates its potential for real-world deployments.

## 4.5 Misclassification Analysis

This section presents a comprehensive error analysis of the LLMxCPG's performance across multiple datasets, including

FormAI, PrimeVul, ReposVul, and SVEN. In other words, we analyze errors made by LLMxCPG under the optimal threshold of each dataset to identify error sources, explain performance variations, and provide insights for future improvements. Table 8 shows the performance of LLMxCPG by CWE.

Table 8: Performance Metrics of LLMxCPG by CWE Category

| | Accuracy | Precision | Recall | F1 Score |
|---|---|---|---|---|
| CWE-119 | 0.684 | 1.000 | 0.608 | 0.756 |
| CWE-120 | 0.333 | 1.000 | 0.111 | 0.200 |
| CWE-125 | 0.500 | 0.579 | 0.375 | 0.455 |
| CWE-190 | 0.582 | 0.681 | 0.688 | 0.684 |
| CWE-415 | 0.691 | 0.891 | 0.721 | 0.797 |
| CWE-416 | 0.618 | 0.762 | 0.557 | 0.643 |

**Performance on Trained Datasets.** LLMxCPG exhibit strong performance on several memory-related Common Weakness Enumeration (CWE) categories within the FormAI and PrimeVul datasets used for training. Specifically, the model achieves high accuracy and F1-scores for CWE-119 (Buffer Overflow), CWE-190 (Integer Overflow), CWE-415 (Double Free), and CWE-416 (Use After Free). This success can be attributed to the balanced representation of these CWEs in the training data and the model's ability to capture distinct code patterns associated with these vulnerability types.

**Challenges with Underrepresented CWEs.** Despite the model's strong performance on some memory-related CWEs, it faces challenges in detecting CWE-120 (Classic Buffer Overflow) and CWE-125 (Out-of-bounds Read). The low performance on these CWEs can be attributed to the limited number of examples, as shown in Table 1. Their underrepresentation in the training dataset likely contributed to the model's reduced effectiveness. Additionally, these vulnerabilities often involve subtle code variations and complex memory access patterns, which are inherently more difficult for the model to capture—especially under conditions of limited training data.

**Generalization Performance on Unseen Datasets.** A critical aspect of evaluating the LLMxCPG's effectiveness is its ability to generalize to unseen datasets. Despite the challenges posed by the ReposVul and SVEN datasets, which were not used during training, our model demonstrates promising generalization capabilities. The model achieved accuracy scores exceeding 0.6 on both datasets, indicating its ability to detect vulnerabilities in diverse, real-world codebases. While the model demonstrates promising generalization performance on unseen datasets, there is still room for improvement. We aim to bridge this gap by compiling a new high-quality dataset by synthetically augmenting real-world datasets spanning multiple files and functions and verifying the output with bounded model checking.

## 4.6 Robustness to Code Augmentation

Robustness in vulnerability detection refers to a model's ability to maintain consistent performance when the input code undergoes semantically-preserving transformations. Robustness against code transformations indicates the model's ability to capture fundamental vulnerability patterns rather than superficial code characteristics. Robustness is also essential for security applications, as adversaries might attempt to evade detection by applying simple code transformations while maintaining the vulnerable behavior.

In this section, we study the ability of LLMxCPG to handle different types of noise and variations. Risse et al. [26] define different data transformations to evaluate the performance of state-of-the-art vulnerability detection models and inspect their reliance on unrelated features. Risse's results indicate that state-of-the-art models overfit unrelated features.

We select four code transformation algorithms, one from each category defined in Risse's work [26]. The used transformations are described in Table 9.

Table 9: The semantic preserving transformations that we use in our experiments.

| Identifier | Type | Description |
|---|---|---|
| T1 | Identifier Renaming | Rename all function parameters to a random token. |
| T2 | Statement Insertion | Insert unexecuted code. |
| T3 | Statement Reordering | Move the code of the function into a separate function. |
| T4 | Statement Removal | Remove all comments. |

**Experimental Setup.** To evaluate the robustness of LLMxCPG, we conduct experiments using three diverse datasets: FormAI, PrimeVul, and SVEN. For each code slice in these datasets, we apply four previously mentioned code transformations. These transformations are designed to reflect common code modifications that preserve program semantics but may challenge model performance. Finally, we evaluate LLMxCPG's performance on both the original and transformed code slices.

Table 10: Comprehensive Performance Results Across Datasets and Transformations

| Dataset | Transformation | Accuracy | Precision | Recall | F1-Score |
|---|---|---|---|---|---|
| FormAI | Normal and T4 | 0.8146 | 0.8097 | 0.8054 | 0.8075 |
| | T1 | 0.8146 | 0.8098 | 0.8054 | 0.8076 |
| | T2 | 0.8068 | 0.8000 | 0.8000 | 0.8000 |
| | T3 | 0.8355 | 0.8506 | 0.8000 | 0.8245 |
| PrimeVul | Normal and T4 | 0.7250 | 1.0000 | 0.4500 | 0.6206 |
| | T1 | 0.7375 | 1.0000 | 0.4750 | 0.6441 |
| | T2 | 0.6650 | 1.000 | 0.3250 | 0.4906 |
| | T3 | 0.6750 | 0.6750 | 0.6750 | 0.6750 |
| SVEN | Normal and T4 | 0.6020 | 0.5590 | 0.9534 | 0.7048 |
| | T1 | 0.5551 | 0.5488 | 0.6977 | 0.6143 |
| | T2 | 0.6220 | 0.6279 | 0.6279 | 0.6279 |
| | T3 | 0.6220 | 0.5864 | 0.8682 | 0.7000 |

Table 10 shows the results of this experiment, which reveal several interesting patterns:

1. **Complete robustness against comments removal**: LLMxCPG is unaffected by the comment removal transformation (T4) because our slice construction approach inherently ignores comments, focusing solely on execution paths. As a result, the constructed slices are free of comments by design.

2. **Dataset-Dependent Robustness**: The model shows varying levels of robustness across different datasets. Notably, it demonstrates the highest robustness on the FormAI dataset, where performance improves slightly under transformations (F1-score increase of 2.10%).

3. **Transformation Impact**: Among the transformations, T3 (function extraction) generally had the most significant impact on the model's performance, particularly affecting recall. This suggests that the slice-based approach is somewhat sensitive to changes in function boundaries.

4. **Precision-Recall Trade-off**: For PrimeVul, we observe an interesting trade-off where transformations lead to decreased precision but improved recall, indicating that the model becomes more conservative in its vulnerability predictions under code transformations.

The robust performance of LLMxCPG, particularly on the FormAI dataset, can be attributed to two key factors. First, our CPG-based slice construction inherently focuses on semantic relationships rather than syntactic features, making it naturally resistant to surface-level code changes. Second, by constructing slices that capture essential vulnerability-related interactions, we effectively filter out irrelevant code modifications.

These findings highlight the importance of semantic-aware vulnerability detection approaches and suggest that future improvements should focus on maintaining robust performance across diverse vulnerability patterns while preserving the ability to capture essential semantic relationships in the code.

## 5 Discussion

**Limitations in Vulnerability Type Coverage.** Current code property graph (CPG) approaches, while effective for many vulnerability types, face inherent limitations in modeling certain classes of security flaws. As demonstrated by Yamaguchi et al. [40], vulnerabilities like race conditions and design errors remain challenging to express using graph traversals since they often depend on runtime properties or require deeper understanding of the system's intended design. This limitation stems from the static nature of CPG analysis, which cannot capture dynamic program behaviors or complex architectural decisions.

**Dataset Quality and Availability.** A significant challenge in vulnerability detection research is the scarcity of high-quality datasets. As evidenced in the PrimeVul study, existing benchmarks suffer from poor data quality, low label accuracy, and high duplication rates. For instance, their analysis revealed that only 38%-64% of functions labeled as vulnerable in popular datasets actually contained security flaws. The ReposVul paper further highlights this issue by demonstrating how vulnerability-fixing commits often include unrelated code changes, leading to noisy labels when using automated collection methods. This data quality problem fundamentally limits the effectiveness of machine learning approaches for vulnerability detection.

**Project-level Vulnerability Detection.** The promising performance of LLMxCPG on project-level vulnerability detection, despite being primarily trained on function-level data, suggests significant potential for advancing automated security analysis of complex software systems. While achieving 0.634 accuracy on ReposVul demonstrates meaningful progress, the pipeline's performance variance across different complexity metrics indicates opportunities for further enhancement. A critical path forward lies in the development of high-quality, project-level vulnerability datasets that capture the intricate dependencies and interactions present in real-world codebases. The creation of these curated datasets, combined with LLMxCPG's capability to handle complex code structures through its CPG-guided approach, could significantly advance the state-of-the-art in project-wide vulnerability detection. This represents a promising direction for future research, potentially enabling more comprehensive and reliable security analysis of large-scale software projects.

**Limitations of Binary Classification.** While LLMxCPG does output a binary classification (vulnerable/safe), we designed our approach to mitigate reasoning opacity through several mechanisms. The CPG-based slice construction inherently preserves reasoning pathways by capturing execution flows and data dependencies that contribute to vulnerability presence. For example Figure 4 provides visibility into how specific code elements interact to create vulnerable conditions. In order to explore the reasoning effect on this task, we fine-tuned LLMxCPG-D with reasoning traces extracted from DeepSeek-v3, but the results did not improve as expected as the accuracy was similar to binary classification. This was likely due to the quality and quantity of available reasoning traces. Recent distillation work [13] (e.g., from DeepSeek-R1 to Qwen-2.5) demonstrates that effective reasoning transfer requires substantial data volumes (i.e., 800k datapoints), which exceeded our available data for this specific task. Enhancing vulnerability reasoning through high-quality traces remains a future work.

**Context Length Limitation of the Base LLM for Query Generation.** While the base model for LLMxCPG-Q (Qwen/Qwen2.5-Coder-32B-Instruct) supports a 128K token context, we fine-tuned it with a 32K token limit due to constraints in our available computing power. It is worth noting that while versions of Qwen (and other models) with even larger context windows (e.g., 1M tokens) are available, fine-tuning these necessitates substantial computational resources, which were beyond our current capacity. Consequently, the query generation process with LLMxCPG-Q is most effective for codebases or commit files that fit within this 32K fine-tuned context length (and theoretically 1M given the necessary resources). Importantly, this context length limitation was specific to the query model (LLMxCPG-Q) and did not pose a problem for our detection model (LLMxCPG-D, based on Qwen/QwQ-32B-Preview). The Code Property Graph (CPG) based slicing was indeed effective in reducing the code to a manageable size for the detection model (even when limting the context to 8K in LLMxCPG-D for faster generation), ensuring its input context was not exceeded. This addresses the core function of CPG in our framework – to precisely slice and reduce code to fit the detection model's context effectively.

## 6 Related Work

**Deep Learning Approaches.** The evolution of deep learning in vulnerability detection progresses through several architectural paradigms. Initial approaches centered on sequential modeling, with VulDeePecker [22] establishing the viability of LSTM networks for processing code gadgets. SySeVR [20] advanced this foundation by introducing systematic feature extraction based on semantic relationships. A pivotal investigation by Chakraborty et al. [2] revealed a critical limitation: while deep learning models demonstrated promising results, their decision-making often relied on superficial code patterns rather than fundamental vulnerability characteristics. Architectural innovations emerged to address these limitations. LineVul [9] introduced transformer-based architectures for fine-grained vulnerability detection at the line level, while Steenhoek et al. [31] incorporated dataflow analysis principles into deep learning frameworks to enhance detection efficiency. Graph-based representations have proven particularly effective, with several notable implementations. Vul-LMGNN [23] achieved superior results by integrating pre-trained code language models with code property graphs through a specialized gated Graph Neural Network architecture. FUNDED [34] enhanced the reliability of graph-based approaches through automated data acquisition and probabilistic learning mechanisms. ReGVD [25] further refined graph neural network architectures, demonstrating substantial performance improvements through targeted architectural modifications.

**LLMs for Vulnerability Detection.** Large Language Models (LLMs) have emerged as a transformative approach to vulnerability detection, though recent research has revealed important nuances in their application. PrimeVul [5] provided critical insights by demonstrating that conventional benchmarks substantially overestimate model performance, necessitating more rigorous evaluation methodologies. VulBERTa [14] established that domain-specific pre-training protocols significantly enhance detection capabilities, while VulSim [29] introduced an innovative multi-dimensional embedding approach that simultaneously captures semantic, contextual, and syntactic code properties. Recent advances have focused on specialized fine-tuning strategies and architectural integration. VulLLM [6] developed a multi-task instruction fine-tuning framework that demonstrably improves model generalization across diverse vulnerability types. MSIVD [41] advanced this direction through carefully designed instruction sets and decomposed task structures, achieving enhanced detection accuracy. Hybrid architectures have shown particular promise, with VDDA [3] successfully combining deep learning with attention mechanisms, and CPVD [42] enabling cross-project vulnerability detection through graph attention networks. Foundational models including UnixCoder [11], CodeBERT [7], and GraphCodeBERT [12] have demonstrated the value of incorporating structural code information during the pre-training phase, establishing essential building blocks for future advances in the field.

LLMxCPG, differs from previous approaches by uniquely leveraging LLMs to generate valid CPGQL queries for traversing code property graphs. Unlike traditional deep learning approaches that directly learn from code representations, or pure LLM approaches that may miss structural information, LLMxCPG uses LLMs to guide the graph traversal process itself. While VulLLM and MSIVD demonstrate the potential of fine-tuned LLMs, and Vul-LMGNN shows promise in combining LMs with graph neural networks, our approach maintains interpretability through explicit query generation. This novel integration preserves the benefits of graph-based program analysis while utilizing LLMs' pattern recognition capabilities in a more controlled and explainable manner.

## 7 Conclusion

In this paper, we have presented LLMxCPG, a novel vulnerability detection approach that effectively addresses fundamental limitations in current deep learning-based methods. By combining Code Property Graphs with Large Language Models, our framework achieves superior performance across multiple evaluation dimensions. The empirical results demonstrate substantial improvements over existing approaches, with F1-score increases of up to 40% and consistent performance on rigorously verified datasets. LLMxCPG's ability to maintain robust detection capabilities under code transformations while generalizing effectively to complex, multi-function codebases represents a significant advancement in automated vulnerability detection. These results establish LLMxCPG as a promising foundation for future research in software security analysis.

## Acknowledgments

We thank the anonymous reviewers for their valuable feedback and their help to improve the quality of this manuscript.

## Ethics Considerations

Our vulnerability detection research adheres to responsible disclosure protocols and established security research guidelines. We carefully balance the benefits of identifying security weaknesses against potential risks. All discovered vulnerabilities are reported through appropriate channels, allowing sufficient time for patches before public disclosure. We maintain strict confidentiality throughout the research process and ensure our methods do not compromise system integrity or user privacy.

## Open Science

Our source code, fine-tuned models, and testing datasets is available publicly to the community to foster research in this field at: https://github.com/qcri/llmxcpg and https://zenodo.org/records/15614095.

## References

[1] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, David Lo, Lili Bo, Bin Li, Xiaolei Liu, Xingwei Lin, and Wei Liu. Snopy: Bridging sample denoising with causal graph learning for effective vulnerability detection. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 606–618, New York, NY, USA, 2024. Association for Computing Machinery.

[2] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet. In *IEEE Transactions on Software Engineering*, 2021.

[3] Jia Chang, Zheng Ma, Bin Cao, and Erping Zhu. Vdda: An effective software vulnerability detection model based on deep learning and attention mechanism. In *26th International Conference on Computer Supported Cooperative Work in Design*, pages 474–479, 2023.

[4] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Research in Attacks, Intrusions and Defenses*, pages 654–668, 2023.

[5] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*, 2024.

[6] Xiang Du, Mingji Wen, Jiacheng Zhu, Zicheng Xie, Bingfeng Ji, Han Liu, Xiaofei Shi, and Hai Jin. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. *arXiv preprint arXiv:2406.03718*, 2024.

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[8] Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[9] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022.

[10] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. Esbmc 5.0: an industrial-strength c model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 888–891, New York, NY, USA, 2018. Association for Computing Machinery.

[11] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[12] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow, 2021.

[13] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[14] Hazim Hanif and Sergio Maffeis. Vulberta: Simplified source code pre-training for vulnerability detection. *arXiv preprint arXiv:2205.12424*, 2022.

[15] Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on*

*Computer and Communications Security*, pages 1865–1879, 2023.

[16] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[17] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[18] Yi Li, Shaohua Wang, and Tien N Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 292–303, 2021.

[19] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.

[20] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.

[21] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[22] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[23] Ruitong Liu, Yanbin Wang, Haitao Xu, Bin Liu, Jianguo Sun, Zhenhao Guo, and Wenrui Ma. Source code vulnerability detection: Combining code language models and code property graphs. *arXiv preprint arXiv:2404.14719*, 2024.

[24] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[25] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Tien Le, Quang Huu Tran, and Dinh Phung. Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering*, pages 178–182, 2022.

[26] Niklas Risse and Marcel Böhme. Uncovering the limits of machine learning for automatic vulnerability detection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4247–4264, Philadelphia, PA, August 2024. USENIX Association.

[27] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.

[28] Danielle M Seid. Reveal. *Transgender Studies Quarterly*, 1(1-2):176–177, 2014.

[29] Samiha Shimmi, Ashiqur Rahman, Mohan Gadde, Hamed Okhravi, and Mona Rahimi. Vulsim: Leveraging similarity of multi-dimensional neighbor embeddings for vulnerability detection. In *33rd USENIX Security Symposium*, 2024.

[30] Statista Research Department. Number of Common Vulnerabilities and Exposures (CVE) reported worldwide from 1999 to 2024, 2024. Statistical analysis of global vulnerability trends documented in the CVE database.

[31] Benjamin Steenhoek, Hongyu Gao, and Wei Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection. *arXiv preprint arXiv:2312.16771*, 2023.

[32] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. An empirical study of deep learning models for vulnerability detection. In *Proceedings of the 45th International Conference on Software Engineering*, pages 2237–2248, 2023.

[33] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C. Cordeiro. How secure is ai-generated code: a large-scale comparison of large language models. *Empirical Software Engineering*, 30(47), 2025.

[34] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 2020.

[35] Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. Reposvul: A repository-level high-quality vulnerability dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 472–483, 2024.

[36] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[37] Tongshuai Wu, Liwei Chen, Gewangzi Du, Dan Meng, and Gang Shi. Ultravcs: Ultra-fine-grained variable-based code slicing for automated vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 19:3986–4000, 2024.

[38] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. MVP: Detecting vulnerabilities using Patch-Enhanced vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1165–1182. USENIX Association, August 2020.

[39] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[40] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[41] Andrew Z. Yang, Huanhuan Tian, Hongru Ye, Ruben Martins, and Claire Le Goues. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv preprint arXiv:2406.05892*, 2024.

[42] Chenyu Zhang, Baojiang Liu, Yu Xin, and Liang Yao. Cpvd: Cross project vulnerability detection based on graph attention network and domain adaptation. *IEEE Transactions on Software Engineering*, 2023.

[43] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

## A Code Metrics

This section defines foundational software metrics utilized in quantitative code analysis, with particular emphasis on structural and cognitive complexity assessment.

**Lines of Code (LOC).** A fundamental volumetric metric quantifying program size through source code line enumeration. LOC encompasses physical lines containing executable statements, declarations, and definitions, while excluding blank lines and comments. This metric serves as a primary indicator of implementation scale and maintenance burden.

**Cyclomatic Complexity (CC).** A graph-theoretic metric measuring program flow complexity through control flow analysis. CC quantifies the number of linearly independent paths through program source code, calculated as:

$$CC = E - N + 2P \tag{1}$$

where:

- $E$ represents the number of edges in the control flow graph
- $N$ represents the number of nodes
- $P$ represents the number of connected components

For a given function $f$, the complexity can be alternatively expressed as:

$$CC(f) = 1 + \sum_{d \in D} p(d) \tag{2}$$

where $D$ is the set of decision points and $p(d)$ represents predicates at each decision point.

**Number of Functions.** A modularity metric quantifying discrete functional units within the codebase. This metric encompasses all function declarations and definitions, including methods, procedures, and subroutines, providing insight into code compartmentalization and potential maintenance complexity.

**Number of Branches.** A control flow metric enumerating decision points within the code. This encompasses conditional statements (if-else constructs), switch cases, and loop conditions. The total branch count $B$ for a program $P$ can be expressed as:

$$B(P) = \sum_{i=1}^{n} b_i \tag{3}$$

where $b_i$ represents individual branching constructs.

**Nesting Depth.** A structural complexity metric measuring the maximum level of control structure embedding within the codebase. For a given code block $c$, the nesting depth $ND$ is defined as:

$$ND(c) = \max_{s \in S} d(s) \tag{4}$$

where $S$ represents the set of all statements in the code block and $d(s)$ represents the nesting level of statement $s$.

## B Comparison with Other Base Models

In order to choose the final model that we used in final detection (i.e., LLMxCPG-D), we fine-tuned Phi-4 (14B), Qwen2.5-Coder (32B), Codestral (22B), and QwQ-Preview (32B). Table 11 shows the comparison among the models on PrimeVul dataset.

Table 11: Comparison of fine-tuned models on the PrimeVul dataset.

| Model | Accuracy | F1-score |
|---|---|---|
| Phi-4 (14B) | 0.5912 | 0.5356 |
| Codestral (22B) | 0.6233 | 0.5521 |
| Qwen2.5-Coder (32B) | 0.6823 | 0.6001 |
| QwQ-Preview (32B) | 0.7250 | 0.6206 |

## C Impact of Slice Construction and Query Generation Model

To assess the impact of code slicing on detection performance, we evaluate LLMxCPG-D on the test datasets using the full code, bypassing the CPG slicing step. The model's performance consistently declines across all datasets when slicing is omitted: FormAI achieves an accuracy of 0.6762 (down from 0.8146 with slicing), PrimeVul drops to 0.4875 (from 0.7250), and SVEN falls to 0.5078 (compared to 0.6020).

To further demonstrate the flexibility of the query-based model, we employ Joern-scan[5], a tool that executes predefined queries targeting various vulnerability patterns, to analyze the 50 samples previously selected for the semantic correctness experiment (see Section 4.3.1). Notably, Joern-scan fails to detect any of the vulnerable samples in this subset, as it relies on a fixed set of sensitive function calls in C. In real-world scenarios, however, developers often implement custom wrappers around these functions, making them more difficult to detect using static query-based approaches.

## D Choosing a Threshold

The process of choosing a threshold for the model starts by select few labeled datapoints (e.g., 20 was used for our case) from the validation splits of the target datasets. Then, we generate predictions with LLMxCPG-D and maximize the accuracy by performing a complete search over the interval of threshold values $[0, 1]$. Figure 7 shows the effect of varying threshold on the accuracy of LLMxCPG-D on different datasets.

[5]https://docs.joern.io/scan/

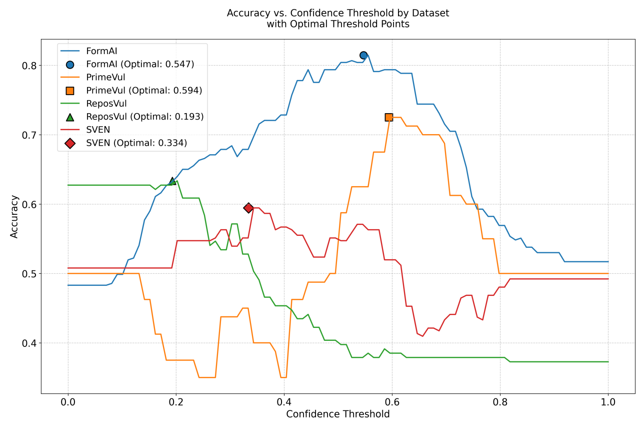


Figure 7: LLMxCPG-D's Accuracy vs Confidence Threshold for Different Datasets.

## E Prompt Templates

In this section, we present the employed prompts for query generation and vulnerability detection.

**Instruction:**
*Your task is to design Precise Joern CPGQL Queries for Vulnerability Analysis.*

**Objective:**
Develop targeted CPGQL Joern queries to:

- Identify taint flows based on your analysis.
- Capture potential vulnerability paths.

**Constraints:**

- Queries must be executable in Joern/CPGQL
- Use Scala language features for query construction
- Last query must use reachableByFlows to identify vulnerable paths

**Output Requirements:**
Provide a JSON object with one field "queries": Sequence of CPGQL queries to detect vulnerability

**Expected JSON Output Format:**
{
"queries": ["Query1" , "Query2", ..., "Final Reachable Flows Query"]
}

**Example Output:**
Example in Figure 9

**Input:** <Code>

Figure 8: Prompt to generate CPGQL queries.

```
{
    "queries": [
        "val freeCallsWithIdentifier = cpg.method.name("(.*_)?free")
          .filter(_.parameter.size == 1)
          .callIn
          .where(_.argument(1).isIdentifier)
          .l",
        "freeCallsWithIdentifier.flatMap(f => {
            val freedIdentifierCode = f.argument(1).code
            val postDom             = f.postDominatedBy.toSetImmutable
            val assignedPostDom = postDom.isIdentifier
              .where(_.inAssignment)
              .codeExact(freedIdentifierCode)
              .flatMap(id => id ++ id.postDominatedBy)
            postDom
              .removedAll(assignedPostDom)
              .isIdentifier
              .codeExact(freedIdentifierCode)
              .reachableByFlows(f.argument(1))
          }).l"
    ]
}
```

Figure 9: Example CPGQL Queries.

**Instruction:**
*You are a security code vulnerability analyzer. Your task is to carefully analyze the provided code snippet. Note that the provided code snippet might not be complete, but it has all the important context.*

Your output must be EXACTLY ONE WORD:

- If you detect any potential security vulnerability in the specified code segment, return: VULNERABLE
- If the code segment appears to be secure and free from obvious vulnerabilities, return: BENIGN

**IMPORTANT GUIDELINES:**
Consider common vulnerability types such as:

- Buffer overflows
- Improper input validation
- Integer Overflow
- Memory corruption potential
- Double free
- Use after free

Your response must be either 'VULNERABLE' or 'SAFE' - no additional explanation

**Output format:**
One word: VULNERABLE or SAFE

**Input:** <Code>

Figure 10: Prompt to classify code slices.