CE 815 – Secure Software Systems

Modern Vulnerability Detection Methods (LLMxCPG)

Mohammad Haddadian/Mehdi Kharrazi
Department of Computer Engineering
Sharif University of Technology

Acknowledgments: Some of the slides are fully or partially obtained from other sources. A reference is noted on the bottom of each slide, when the content is fully obtained from another source. Otherwise a full list of references is provided on the last slide.

LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models, Lekssays A., Mouhcine H., Tran K., Yu T., Khalil I., Usenix Security 2025.

Why Do We Need Better Vulnerability Detection?

- **Scale**: 25,000+ new vulnerabilities reported in CVE database (2024 alone)
- Current Limitations of Deep Learning Approaches:
  - Accuracy drops up to 45% on rigorously verified datasets
  - Performance degrades significantly under simple code modifications
  - Focus only on function-level analysis, missing inter-procedural dependencies
  - Learn superficial patterns rather than meaningful vulnerability indicators

Trend to 2025

- Move from **raw-code classifiers** toward **structured program representations + controlled evaluation + robustness**, with LLMs increasingly used as **tools-orchestrators** rather than pure predictors.

Research Questions

- How can we combine traditional program analysis with LLMs to improve vulnerability detection?
- Can we reduce code size while preserving vulnerability-relevant context to enable analysis of larger code segments?
- **How do we ensure robust detection** across different datasets and under code transformations?
- **Can the approach generalize** from function-level to project-level codebases?

So what?

- Integration of LLMs + program analysis (CPG)
- Vulnerability-focused slicing using CPG queries
- **Generalization** to unseen datasets / complex codebases
- **Robustness** under code transformations

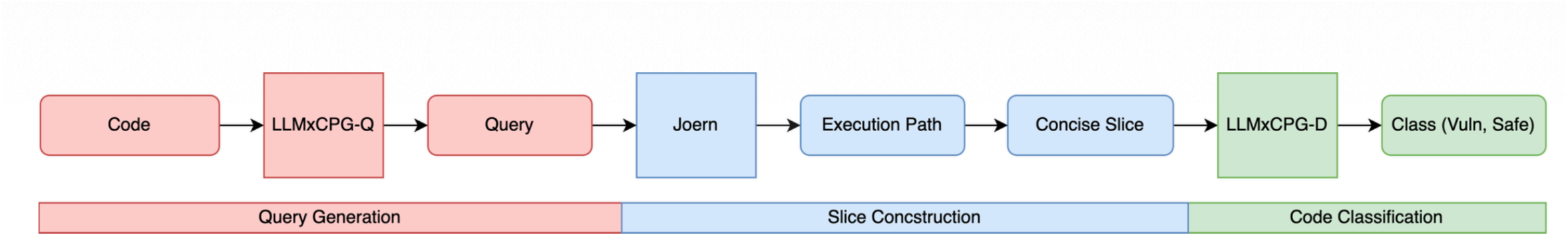
Background - Code Property Graphs (CPG)

- A unified graph representation that merges:
  - Abstract Syntax Trees (ASTs) → syntactic structure
  - Control Flow Graphs (CFGs) → execution flow
  - Program Dependence Graphs (PDGs) → data/control dependencies
- Key Advantage:
  - Express complex vulnerability patterns through graph traversals
  - Tool used: **Joern** with CPGQL query language

Two-Phase Architecture

- Phase 1: Query Generation + Slice Construction (LLMxCPG-Q)
  - Fine-tuned from Qwen2.5-Coder-32B-Instruct
  - Generates CPGQL queries to identify vulnerable execution paths
  - Extracts focused, security-critical code segments
- Phase 2: Vulnerability Detection (LLMxCPG-D)
  - Fine-tuned from QwQ-32B-Preview
  - Classifies code slices as VULNERABLE or SAFE
  - Binary classification on reduced code

System Overview



Slice Construction Problem

- Challenge with Traditional Approaches:
  - Vulnerable code often contains only a **small fraction** of lines related to the vulnerability
  - Including irrelevant code:
    - Increases token usage
    - Models struggle to identify true vulnerability patterns
    - Models rely on spurious features (e.g., variable names)
- Solution: Program Slicing
  - Reduce program to smaller representation
  - Focus on **execution paths** rather than individual criterion points
  - Capture vulnerability behavior with minimal noise

Criterion-point slicing fails

- Selecting criterion points is non-trivial:
  - Predefined sensitive calls (e.g., C library) are incomplete
  - Developers use wrappers → static “dangerous call lists” miss cases
- Patch-diff-based criterion selection is noisy:
  - Patches often include refactoring unrelated to the vulnerability
  - Even with correct criterion points, slices can include substantial irrelevant code

Slice Construction - Three-Step Process

- **Step 1: Taint Path Extraction**
- CPGQL supports graph navigation for precise patterns (method/call/identifier queries)
- In LLMxCPG, queries aim to **extract execution paths relevant to target vulnerabilities**
- Typical taint-style pattern:
  - Define source (untrusted / key variable)
  - Define sink (security-sensitive operation)
  - Compute **reachableByFlows(source)** from sink

Slice Construction - Query example

- **cpg.call — start from all Call nodes**
  - In Joern's CPG, a function invocation like `skb_put(skb, len + ring->frameoffset)` is represented as a **Call** node.
  - cpg.call returns a traversal over **all call sites** in the codebase.

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

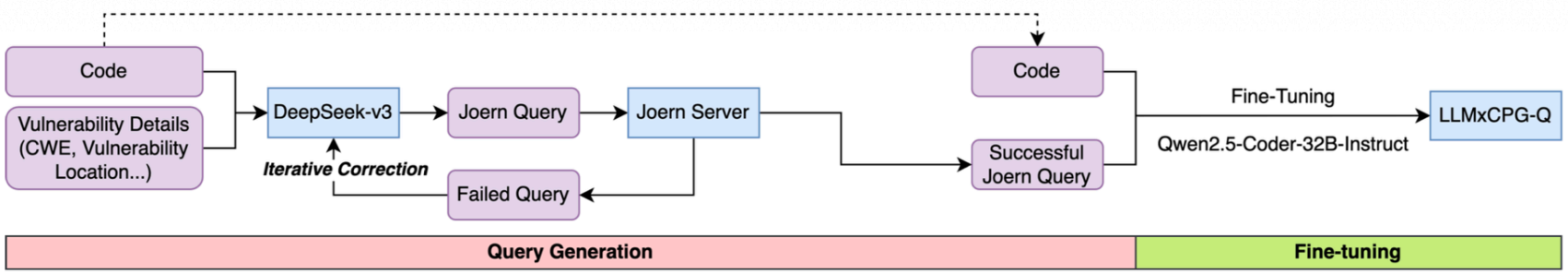
Slice Construction - Query example

- **.name("skb_put") — keep only calls whose callee name is skb_put**
- Filters the Call nodes to those where the callee name equals “skb_put”.
- **Why** `.name(...)` **and not** `.code(...)`
- Common failure mode where models use .code("print") expecting to match the callee name, but .code typically corresponds to **the whole statement**, including arguments

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

Query generation model (LLMxCPG-Q): why it exists

- Challenge: CPGQL is “low-resource”; general LLMs struggle to produce effective Joern queries
- Approach: fine-tune a code LLM (Qwen2.5-Coder-32B-Instruct) to generate valid CPGQL queries
- Output: LLMxCPG-Q produces queries **from code alone** (no CWE label, no vuln location required)



Slice Construction (Example)

85-lines CVE-2011-3359

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);
```

18-lines sliced

```
1  static void dma_rx(struct b43_dmaring *ring,
       int *slot)
2  {
3      u16 len;
4      len = le16_to_cpu(rxhdr->frame_len);
5      if (unlikely(len > ring->rx_buffersize)) {
6          s32 tmp = len;
7          while (1) {
8              tmp -= ring->rx_buffersize;
9              if (tmp <= 0)
10                 break;
11         }
12         goto drop;
13     }
14
15     skb_put(skb, len + ring->frameoffset);
16     drop:
17     return;
18 }
```

Slice Construction - Three-Step Process

- **Step 2: Identify Interacters**
  - Execution path alone may lack critical context
  - Traverse CPG to find variables that interact with execution paths
- Define **interacter**:
  - An **identifier** is an "interacter" if it appears on the same line as an execution path element
  - Goal: expand from path to minimal necessary context

```
46  46    int pure_strcmp(const char * const s1, const char * const s2)
47  47    {
48        -     return pure_memcmp(s1, s2, strlen(s1) + 1U);
      48  +     const size_t s1_len = strlen(s1);
      49  +     const size_t s2_len = strlen(s2);
      50  +     const size_t len = (s1_len < s2_len) ? s1_len : s2_len;
      51  +
      52  +     return pure_memcmp(s1, s2, len + 1);
49  53    }
```

Slice Construction - Three-Step Process

- **Step 3: Backward Slicing**
  - Final slice should include:
    - the execution path
    - interacters
    - all dependencies influencing either of those
  - Mechanism: automated backward slice using Joern query
  - Internals: Joern leverages the Program Dependency Graph (PDG) for backward slice construction

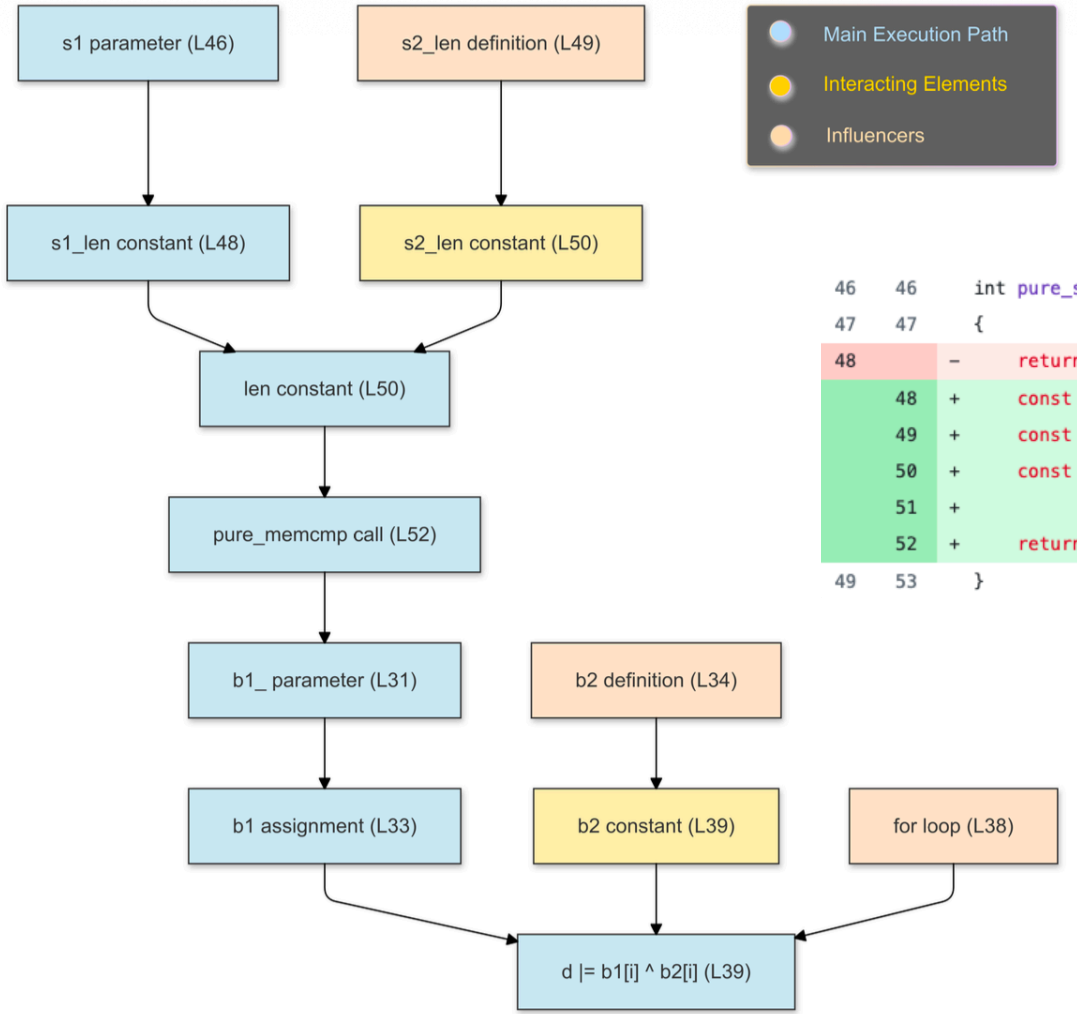
Slice Construction - Three-Step Process

- **Final Query:**

```
execution_path_and_interacters.reachableByFlows(cpg.all)
```

- Reads “collect all nodes that can flow into my target set”
- This computes a backward slice over program dependencies

Slice Construction (Example)




```
46  46      int pure_strcmp(const char * const s1, const char * const s2)
47  47      {
48          -       return pure_memcmp(s1, s2, strlen(s1) + 1U);
        48  +       const size_t s1_len = strlen(s1);
        49  +       const size_t s2_len = strlen(s2);
        50  +       const size_t len = (s1_len < s2_len) ? s1_len : s2_len;
        51  +
        52  +       return pure_memcmp(s1, s2, len + 1);
49  53      }
```

Prompt to generate CPGQL queries

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>

Response of 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"
    ]
}
```

Why metadata can be absent at inference

- The key is: **metadata is used to produce labels (good Joern queries), not necessarily as a required input feature at deployment.**
  - **Training supervision**: they end up with pairs like
    - (code snippet → valid CPGQL query)
    - even if the query was originally discovered using extra metadata during query synthesis/verification.
  - **Inference behavior**: the deployed LLMxCPG-Q is trained to map **from code to query**, so it can learn to infer what graph patterns are likely relevant from code structure itself—without needing vulnerability location metadata at runtime.

Why this slicing approach improves learning

- Models learn vulnerability signatures better when:
  - Irrelevant statements removed (less confounding)
  - Path-centric representation highlights patterns per CWE (e.g., source→sink, missing check)
- Slice construction makes it easier to:
  - Compare vulnerable vs patched versions (security-critical delta stands out)
  - Build cleaner training data for fine-tuning

Design choices & trade-offs

- **Execution-path focus**: reduces noise, but depends on quality of extracted paths
- **Interacters via line-number overla**p:
  - Pragmatic and fast
  - May miss semantic dependencies not on same line (or include unrelated vars on same line)
- **Backward slice scope**:
  - Can still balloon if dependencies are broad
  - PDG precision affects slice precision

What patterns the slice preserves

- The method focuses on **critical data + control-flow paths** that characterize potential vulnerabilities.
- Examples of preserved patterns include:
  - **Source-to-sink paths** for “taint-style” vulnerabilities
  - **Validation-check patterns** for input-handling flaws
- By removing irrelevant code, the slice reduces “noise” that could obscure vulnerability signatures.

Training Strategy

- **LLMxCPG-Q (Query Generation):**
  - Training data generation:
    - Use DeepSeek-v3 to generate initial queries
    - Test on Joern server
    - Provide feedback on syntax errors
    - Iterative refinement (up to 3 attempts)

Training Strategy

- **LLMxCPG-D (Classification):**
  - Fine-tune on extracted code slices from training data
  - Binary classification: VULNERABLE vs SAFE
  - Uses focused code → enables effective vulnerability learning
  - How training data is created:
    - Extract code snippets (slices) from **both vulnerable and safe samples** in the training dataset
    - Use **LLMxCPG-Q** to extract slices, leveraging **ground-truth labels** available in training datasets
  - The fine-tuned classifier is referred to as **LLMxCPG-D.**

LLMxCPG Workflow

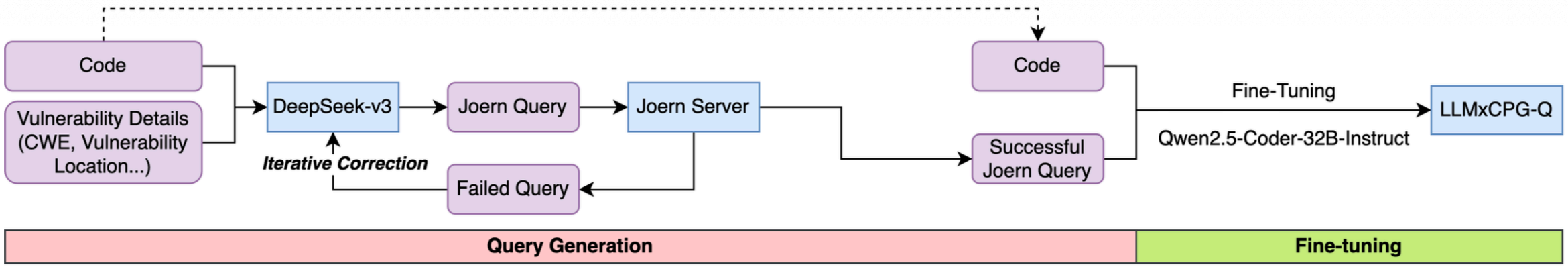


Figure 5: Query Generation Workflow

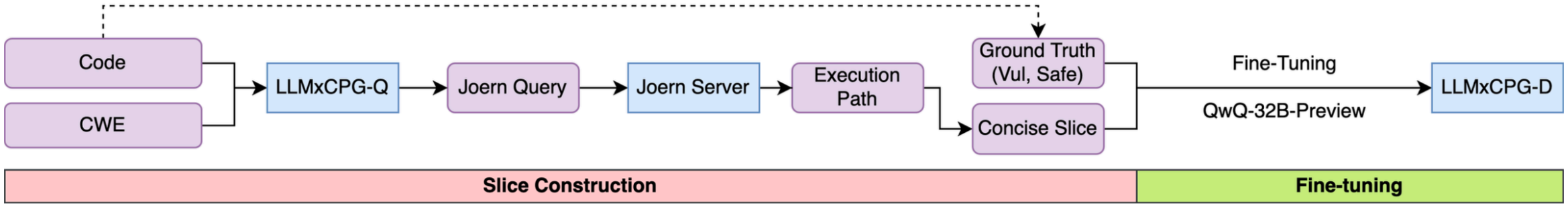


Figure 6: Code Classification Workflow

Prompt to classify code slices

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>

Evaluation

- Evaluation covers:
  - **Datasets** used (training + unseen/generalization) and why these CWEs
  - **Implementation & protocol** (slice construction, fine-tuning, inference)
  - **Performance analysis**: query validity, function-level detection, project-level generalization, misclassification analysis, robustness to transformations
- Key research questions:
  - Can the system reliably generate **valid CPGQL** queries?
  - Does slicing preserve enough signal to improve detection?
  - Does it **generalize** to unseen datasets / project-level code?
  - Is it robust to semantically-preserving code transformations?

Training Datasets

- **FormAI-v2: 331,000 compilable C programs** generated by multiple LLMs (e.g., Gemini-pro, GPT-4, Falcon, CodeLlama2, etc.) using a “dynamic zero-shot prompting” approach.
  - Labeled by **formal verification** using ESBMC bounded model checking.
  - Samples are **not paired** (no vuln-vs-patch pairs) and safe samples aren’t CWE-tagged in the same way because snippets are generated independently.
- **PrimeVul**: built to fix common dataset issues
  - Includes **228,800 safe functions** and **6,968 vulnerable functions** across **140 CWEs** (broad coverage),
  - Here select a subset of memory CWEs for experiments.

Generalization / unseen datasets

- **SVEN**: manually curated **~1,600 C/C++ and Python programs** derived from real-world GitHub security fixes, with “rigorous verification” for data quality/relevance.
- **ReposVul**: repository-level dataset: **6,134 CVE entries, 236 CWE types, 1,491 projects, 4 languages**—used to test project-level realism.

CWE scope

- **Studied CWEs**: memory-related CWEs amenable to static analysis via CPGs:
  - CWE-119/190/415/416 + variants CWE-120/121/122/125/787; exclude vulnerabilities needing runtime behavior (e.g., race conditions).
  - PrimeVul / SVEN / ReposVul: counts are given as **vulnerable/safe pairs** (paired).
  - FormAI: not paired; safe samples are totals only.
- **Balancing in test sets**: ensure balanced representation where possible for SVEN and a balanced subset for ReposVul.

CWE scope

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.

Implementation and experiment protocol

- **Fine-tuning setup**
  - LLMxCPG-Q base: Qwen2.5-Coder-32B-Instruct; LLMxCPG-D base: QwQ-32B-Preview.
  - LoRA via LLaMA-Factory; rank=8, alpha=4, lr=1e-4; NVIDIA A100-80GB (Ubuntu 22.04).
- **Inference tooling**
  - Query inference uses **vLLM** and a prompt template in Appendix E.
  - Classification inference uses **Unsloth**.

Query generation results

- **Validity metric (hard constraint)**
- **Error taxonomy (why base models fail)**
  - **API misuse**: .code("print") filters by the entire call statement (incl. args) → returns empty; correct is .name("print").
  - **Node-type misuse**: applying properties like typeFullName to the wrong node type (arguments).
  - **Regex pitfalls**: .code('a + b') interpreted as regex; must use .codeExact('a + b').

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

Query semantic correctness

- Valid syntax is not enough; teach semantic precision.
  - 3 security experts audited 50 generated queries (PrimeVul + SVEN): 25 true pos/neg and 25 false pos/neg; judged whether queries align with intended vulnerability patterns and isolate vulnerabilities with minimal noise.
- Results:
  - **76%** of true pos/neg: semantically matched vulnerability pattern and yielded meaningful paths.
  - For false cases, breakdown:
    - 28% semantically correct but misclassified
    - 40% targeted different CWE
    - 32% right CWE but missing critical context

Slice efficiency + ablation

- Average reduction on test set:
  - FormAI: **78.70%**
  - PrimeVul: **67.84%**
  - SVEN: **70.22%**
  - ReposVul (project-level): **90.93%**
- Accuracy drops when slicing is removed:
  - FormAI: 0.6762 (vs 0.8146 with slicing)
  - PrimeVul: 0.4875 (vs 0.7250)
  - SVEN: 0.5078 (vs 0.6020)
  - No result about ReposVul! Maybe because of context-window limit
- Slicing is not just for token budget; it changes the **signal-to-noise ratio** in the classifier's input.

Function-level detection performance

- FormAI: Acc **0.8146**, F1 **0.8075**
- PrimeVul: Acc **0.7250**, Precision **1.0**, Recall **0.45**, F1 **0.6206**

- Why does PrimeVul show perfect precision but low recall?
- Why is FormAI higher?

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

Project-level & “post-knowledge-cutoff”

- How PKCO-[20]25 is built
- Start with CVEs with public GitHub/GitLab commit references; crawl **1,583 CVEs**.
- Filter cascade:
  - many lack Git commits; many lack valid CWE tags; then restrict to their studied CWEs; then restrict to commit files that fit input context size (32k tokens; possibly extend to 128k).
- Final PKCO dataset: **57 CVEs**, each paired with its patch → **114 samples** balanced vulnerable/safe.

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

Misclassification analysis by CWE

- Better on well-represented CWEs (balanced in training)
- Worse on underrepresented CWEs; Table 1 shows low sample counts (e.g., PrimeVul has extremely limited CWE-121/122 in their used split).

	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

Robustness to code transformations

- Robustness = consistent performance under semantics-preserving transformations; prevents evasion via superficial edits.

- Transformations tested (one from each category in Risse et al.)
  - T1: rename all function parameters randomly
  - T2: insert unexecuted code
  - T3: move code into a separate function
  - T4: remove all comments

Robustness Results

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

Acknowledgments

- [LLMxCPG] LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models, Lekssays A., Mouhcine H., Tran K., Yu T., Khalil I., Usenix Security 2025.