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

ESORICS 2025

Sajal Halder, Muhammad Ejaz Ahmed, Seyit Camtepe
(Data61, CSIRO, Australia)

Motivation

Growing Software Supply Chain Threat

- NVD recorded **240,000+ CVEs** by Oct 2024 (~15–20% annual growth)
- Attackers inject vulnerable code into widely used open-source packages
- Security analysts must **manually review** functions → increased time & effort

Limitations of Existing Approaches

- Can identify **whether** a function is vulnerable, but **not how many** vulnerabilities exist
- Cannot pinpoint **precise lines of code** containing vulnerabilities

Motivation

FuncVul: Code Chunk-Based Solution

- Identifies **multiple vulnerabilities** within a function
- Highlights **smaller code chunks** containing vulnerabilities → reduces expert review time

Why Code Chunks Instead of Full Functions?

- Vulnerabilities often exist in just **1–2 lines** inside a function
- Models trained on entire functions struggle to **pinpoint vulnerable lines**
- Code chunks reduce search space & minimize tokens for the tokenizer

Key Definitions

Def 1: Function Code Chunk (FC)

- Contiguous segment centered on a code change
- Parse **patch info**: chunk header, removed lines (-), added lines (+)
- Match removed lines in function → record **modified indices**
- If edited region ≤ 10 lines → extend **±3 context lines**
- If edited region > 10 lines → use **edited lines only**

Def 2: Generic Code Chunk

- Variables → $\mathbf{V_1, V_2, ..., V_n}$ / Functions → $\mathbf{F_1, F_2, ..., F_m}$
- Mitigates naming variation → focuses on **code structure**
- Converted using **Gemini 1.5 Pro**

```
diff --git a/tools/tiffcrop.c b/tools/tiffcrop.c
index b87a77a8..70a71e17 100644
--- a/tools/tiffcrop.c
+++ b/tools/tiffcrop.c
@@ -3698,7 +3698,7 @@ static int readContigStripsIntoBuffer (TIFF* in, uint8* buf)
                                     (unsigned long) strip, (unsigned long)rows);
                         return 0;
                 }
-                bufp += bytes_read;
+                bufp += stripsize;
         }
         return 1;
```

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

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

Generic Code Chunk

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

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

Problem Definition

- **Goal:** Develop a vulnerable code detector (**V)** for C/C++ and Python
- **Input:** Patch information-based modified function code chunk (**fc_i**)
- **Output:** Binary classification
  - $V(fc_i) = 1 \rightarrow$ Vulnerable
  - $V(fc_i) = 0 \rightarrow$ Non-vulnerable

Data Generation

Step 1. Code Chunk Extraction

- Function source code + Patch information → **Code Chunks**

Step 2. LLM Vulnerability Detection

- **Property 1**: single-modification CVE patches are likely vulnerable
- **Property 2** (LLM): **Gemini 1.5 Pro** predicts vulnerable lines (code-only / code + CVE description)
- **Yes** → Vulnerable (3 conditions: single-patch CVE + LLM detects vuln lines + overlap with deleted lines)
- **No** → labeled as **Unknown** (excluded from training)

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

Task: Extract the following information:

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

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

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

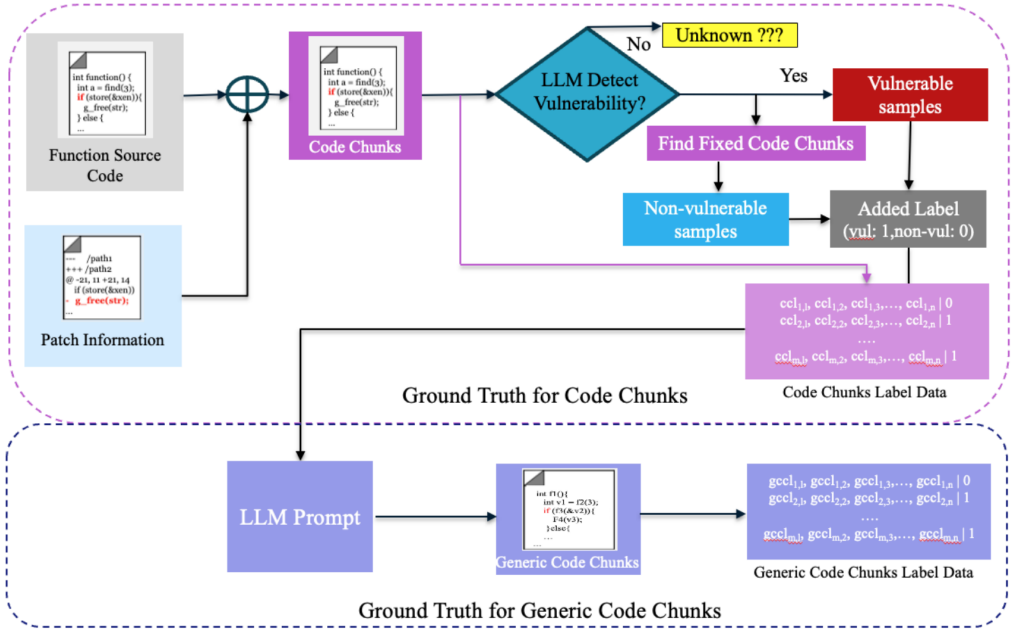


Fig. 1: Code Checks and Generic Code Chunks Label Data Generation.

Key Definitions

Def 1: Function Code Chunk (FC)

- Contiguous segment centered on a code change
- Parse **patch info**: chunk header, removed lines (-), added lines (+)
- Match removed lines in function → record **modified indices**
- If edited region ≤ 10 lines → extend **±3 context lines**
- If edited region > 10 lines → use **edited lines only**

Def 2: Generic Code Chunk

- Variables → $\mathbf{V_1, V_2, ..., V_n}$ / Functions → $\mathbf{F_1, F_2, ..., F_m}$
- Mitigates naming variation → focuses on **code structure**
- Converted using **Gemini 1.5 Pro**

```
diff --git a/tools/tiffcrop.c b/tools/tiffcrop.c
index b87a77a8..70a71e17 100644
--- a/tools/tiffcrop.c
+++ b/tools/tiffcrop.c
@@ -3698,7 +3698,7 @@ static int readContigStripsIntoBuffer (TIFF* in, uint8* buf)
                                        (unsigned long) strip, (unsigned long)rows);
                         return 0;
                 }
-                bufp += bytes_read;
+                bufp += stripsize;
         }
         return 1;
```

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

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

Generic Code Chunk

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

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

Data Generation

Step 3. Non-Vulnerable Sample Construction

- Find **fixed code chunks** from after-version
- Random 5–10 lines from fixed functions

Step 4. Code Chunks Label Data

- Vulnerable (label **1**) + Non-vulnerable (label **0**)

Step 5. Generic Code Chunks Label Data

- Code chunks → **LLM Prompt** → Generic Code Chunks

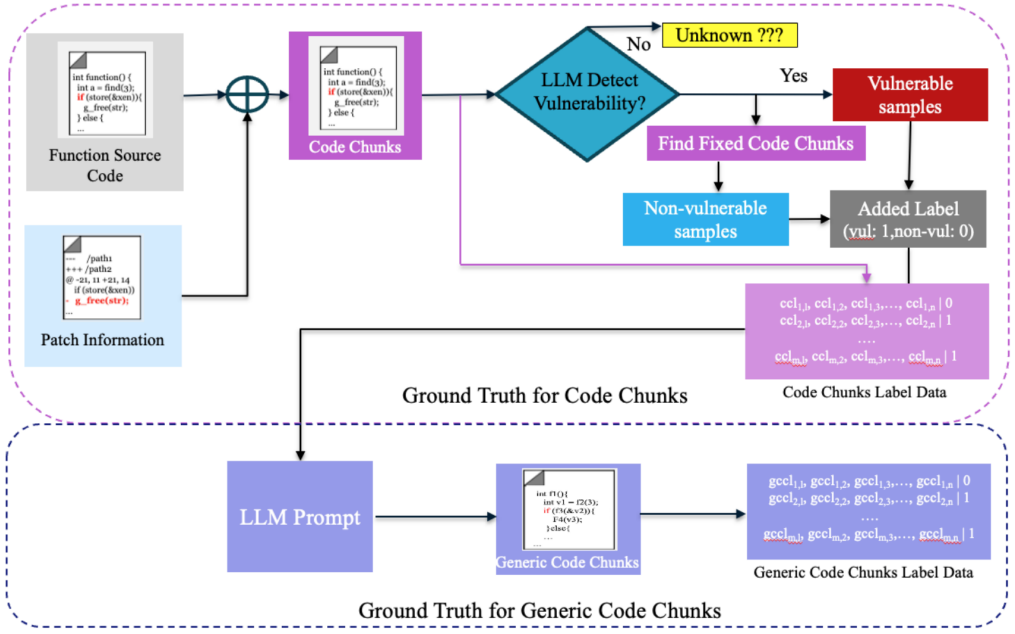


Fig. 1: Code Checks and Generic Code Chunks Label Data Generation.

FuncVul Model Architecture

- Input: code chunks / generic code chunks
- Backbone: **GraphCodeBERT**
- Fine-tuning → **FuncVul Model + Tokenizer**
- Output: binary prediction per code chunk (Vulnerable / Non-vulnerable)

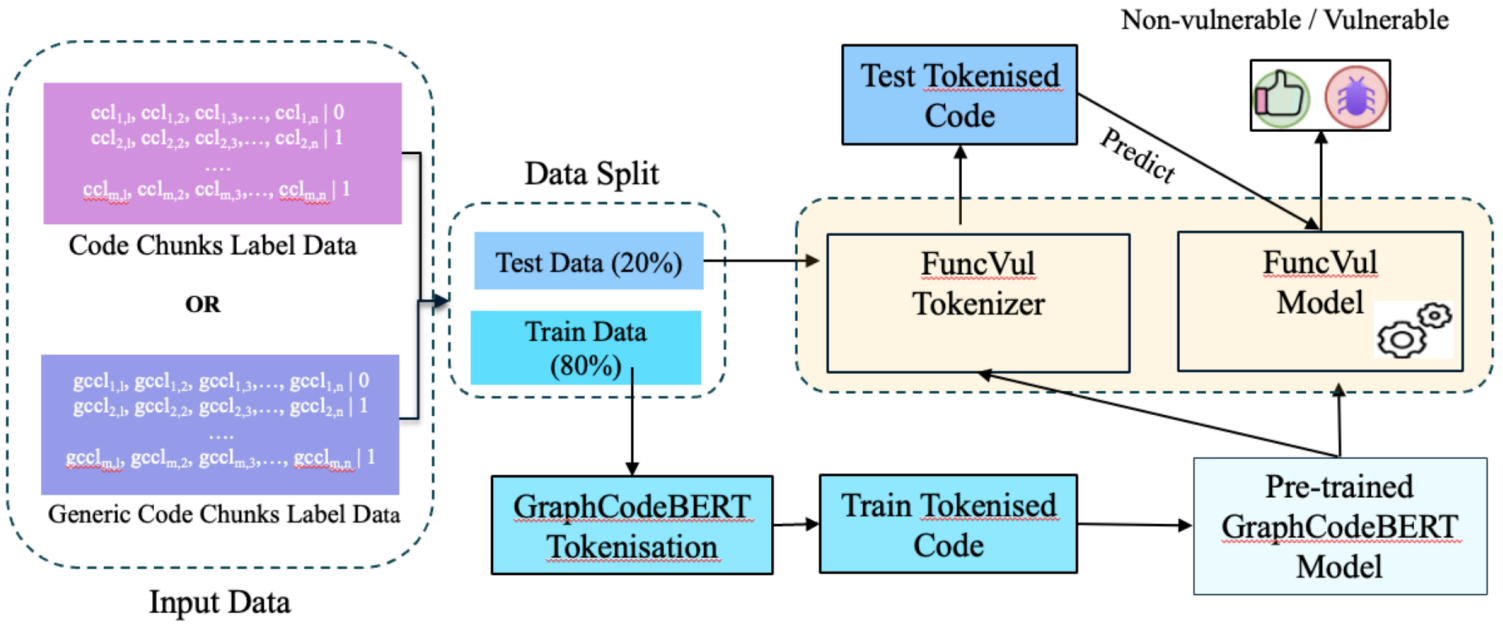


Fig. 2: Proposed code chunk based function vulnerability detection model (FuncVul) architecture.

Experimental Setup

Environment

- MacBook Pro (Apple M3, 24GB RAM) / Embedding **512**

6 Datasets (OSV.dev)

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

Table 1: Details of various datasets.

Baselines

- **CodeBERT**, **CustomVulBERTa**, **BERT**, **VUDENC**, **LineVul**

Metrics

- Accuracy, Precision, Recall, F1-Score, MCC (Matthews Correlation Coefficient)

Research Questions

RQ1: FuncVul Model Performance

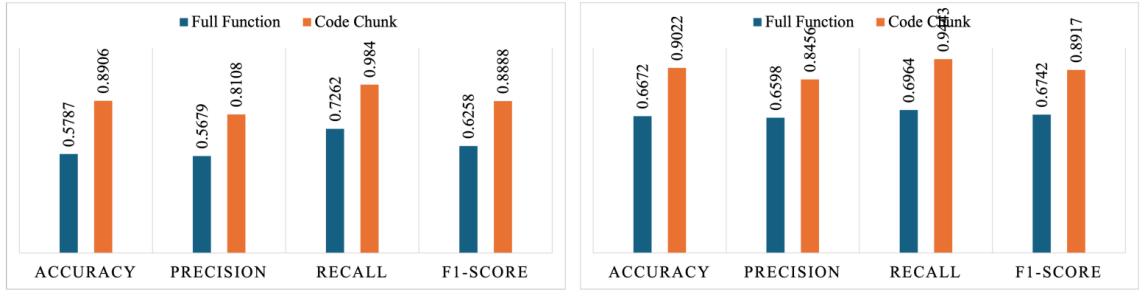
Dataset	Model	Accuracy	Precision	Recall	F1-Score	MCC
1	CodeBERT	0.8707±0.00020	0.7928±0.0216	0.9505±0.0269	0.8641±0.0095 (5)	0.7683±0.0283
	CustomVulBERTa	0.8648±0.0060	0.7974±0.0224	0.9816±0.204	0.8816±0.0091 (3)	0.7898±0.0102
	BERT	0.8812±0.0103	0.8067±0.0168	0.9548±0.0443	0.8739±0.0167 (4)	0.7742±0.0282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

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

Research Questions

RQ2: Code Chunks vs Full Functions

- Smaller, focused segments → **better vulnerability detection**



(a) Dataset 1 (b) Dataset 2

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

RQ3: Code Chunk vs Generic Code Chunk

- Original variable / function naming carries useful vulnerability signals

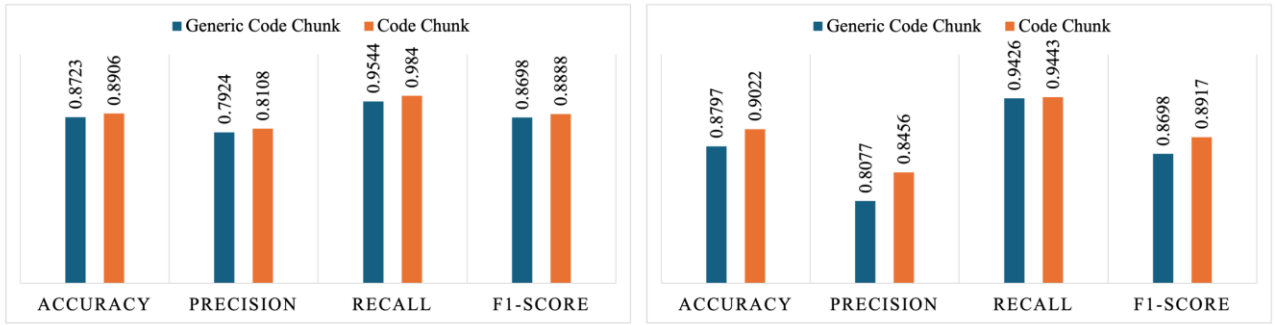


Fig. 4: Comparison between our proposed code chunk based results with generic code chunk based results.

Research Questions

RQ4: Generalization to Unseen CVEs

- **New CVEs (Test Case 1)**: CVEs labeled Unknown during LLM labeling; absent from training data
- **New Project ID (Test Case 2)**: code from entirely different projects than those used in training

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

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

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

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

Data Generation

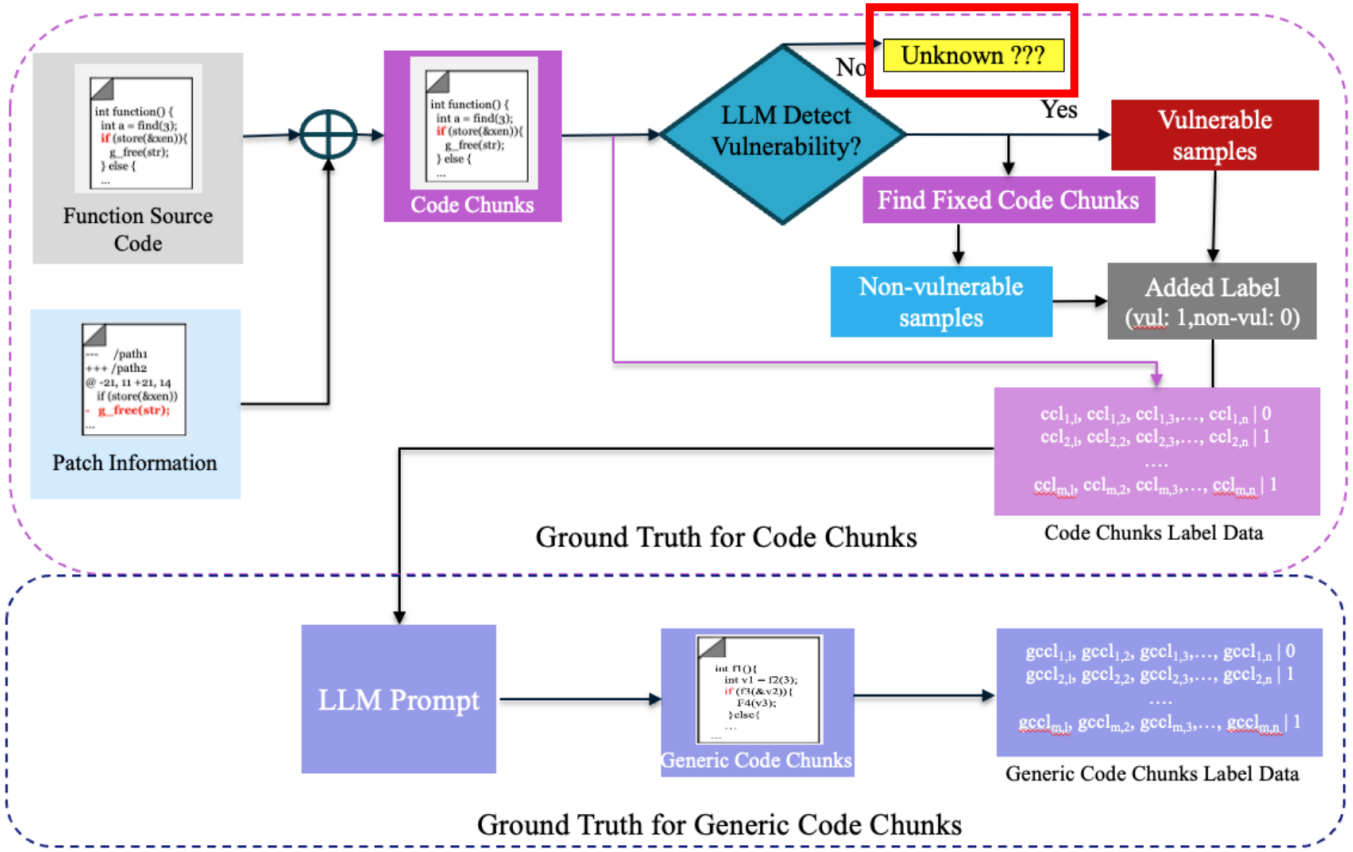


Fig. 1: Code Checks and Generic Code Chunks Label Data Generation.

Summary & Critical Analysis

Strengths

- **Vulnerability labeling**: patch heuristic + LLM cross-validation for more reliable ground truth
- **Code chunk**: +53.9% accuracy over full-function baselines; localizes vulnerable segments

Weaknesses (My thoughts)

- **Unsafe non-vulnerable sampling**: post-patch ≠ fully safe; undiscovered flaws may persist → label noise
- **Patch dependency**: requires diffs to build chunks; cannot detect unknown vulns in unpatched code
- **Single-patch CVEs only**: ~20% multi-commit/multi-file CVEs excluded; complex vulns out of scope
- **No per-language analysis**: C/C++ & Python mixed; vuln patterns differ (buffer overflow vs. injection)

Thank You