CE 815 – Secure Software Systems

Modern Vulnerability Detection Methods (LLM4Vuln)

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.

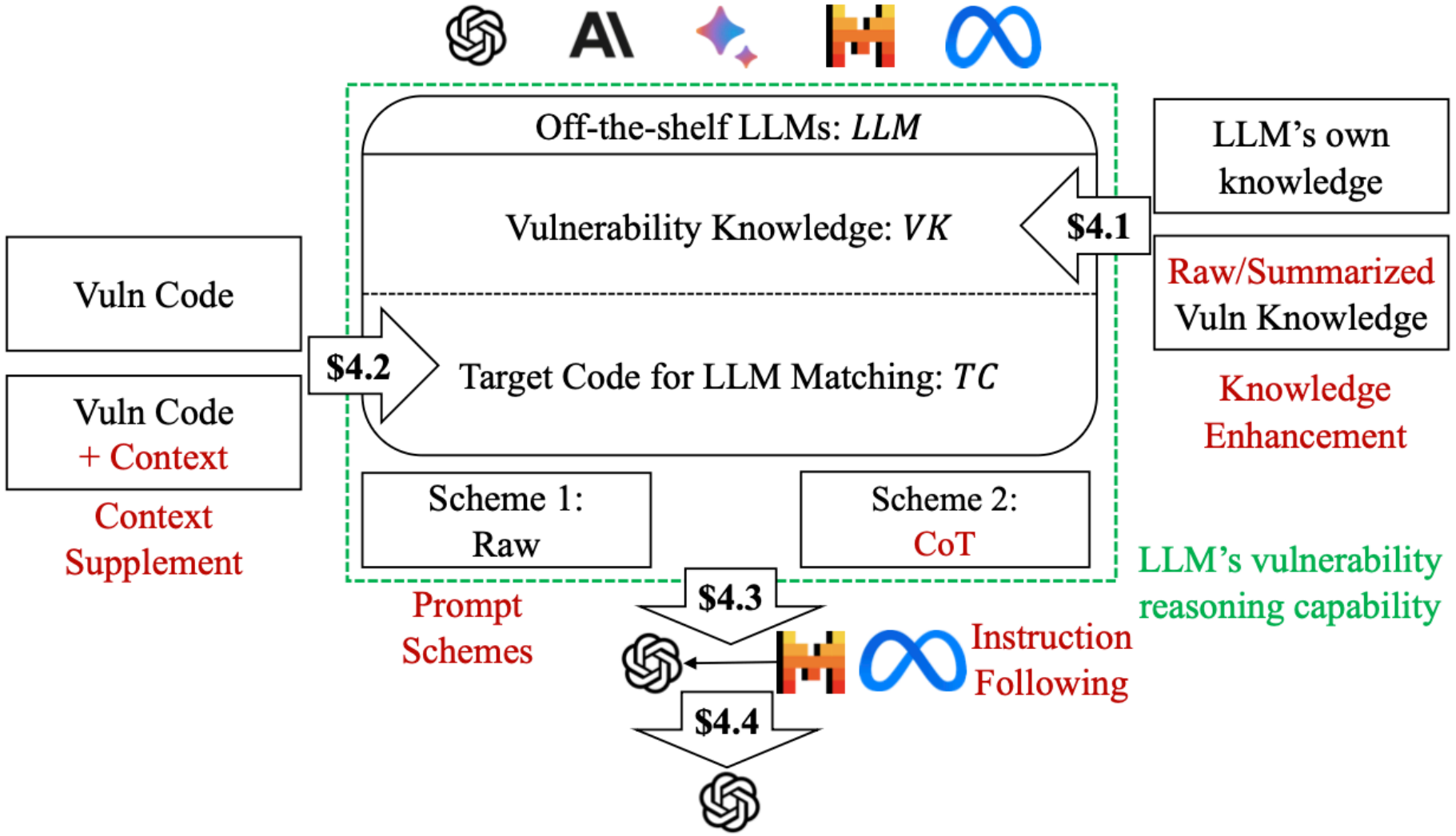
LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning, Y. Sun and D. Wu and Y. Xue and H. Liu and W. Ma and L. Zhang and Y. Liu and Y. Li, arXiv, 2025.

Introduction

- Large Language Models (LLMs) have “significantly transformed our approach to complex challenges” in computer security.
- With extensive pre-trainin” and strong instruction-following, LLMs excel at understanding the semantics of human + programming languages.
- This has led to LLM-based vulnerability detection, offering superior intelligence and flexibility vs. traditional program analysis and neural detectors.

What is missing with LLMs?

- Existing setups use target code (TC) + prompt schemes, but often overlook extra context (e.g., functions/variables) obtainable via tools.
- LLM pre-training cutoff dates make latest vulnerability knowledge hard to adopt; incorporating vulnerability knowledge (VK) is essential.
- Instruction-following differences (open-source vs OpenAI/RLHF) can affect reasoning outcomes in automatic evaluation.
- In this work LLMs' vulnerability reasoning capability is de-coupled from their other capabilities.
    - LLMs' vulnerability reasoning is assessed when combined with the enhancement of other capabilities.

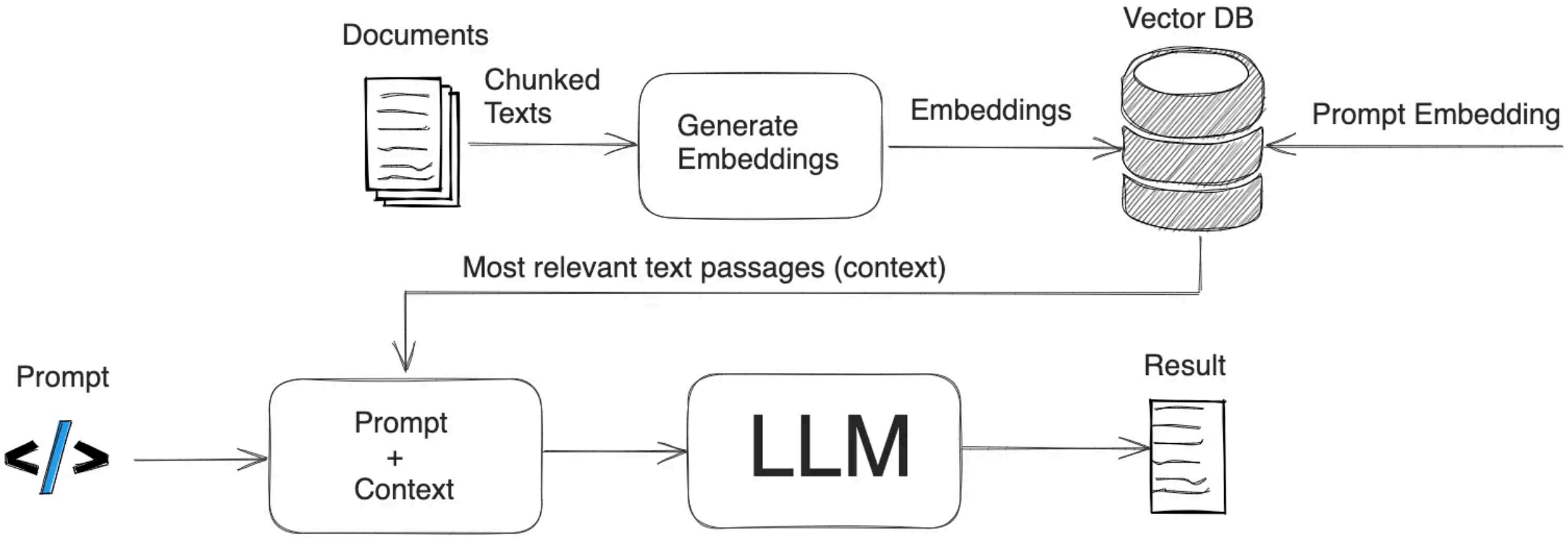


(Annotated) Vulnerability Results

LLM4Vuln

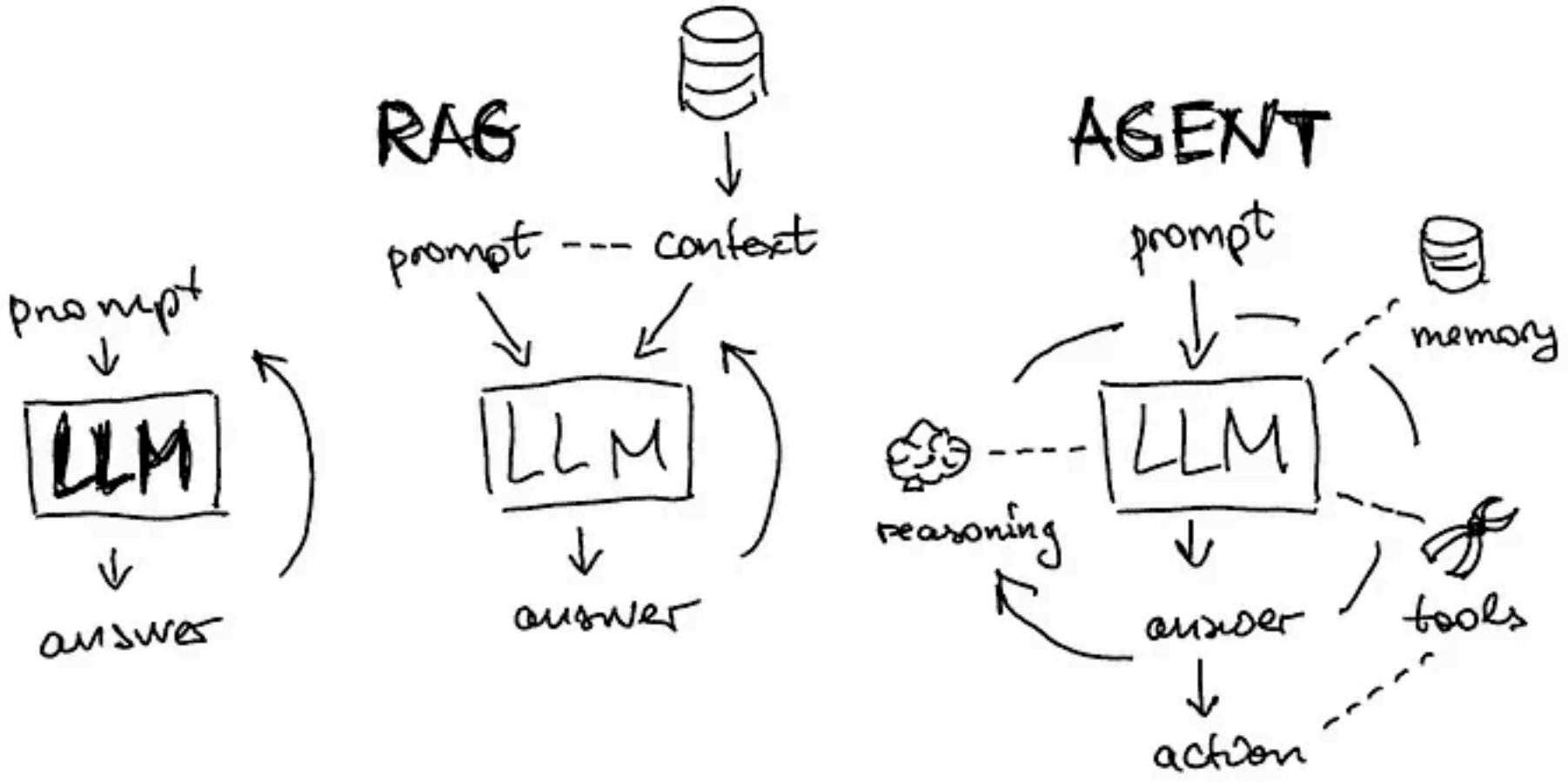
- LLM4Vuln is a unified evaluation framework for decoupling and enhancing LLMs' vulnerability reasoning.
- Considers tool invocation for extra Target Code info (e.g., function calling).
- Enhances Vulnerability Knowledge via a searchable vector DB (RAG-like).
- Incorporates prompt engineering.

RAG: Retrieval-Augmented Generation



[https://learnmycourse.medium.com/retrieval-augmented-generation-rag-process-using-an-llm-339430ff0a05]

LLM vs. RAG vs. Agent



[https://medium.com/data-science/intro-to-llm-agents-with-langchain-when-rag-is-not-enough-7d8c08145834]

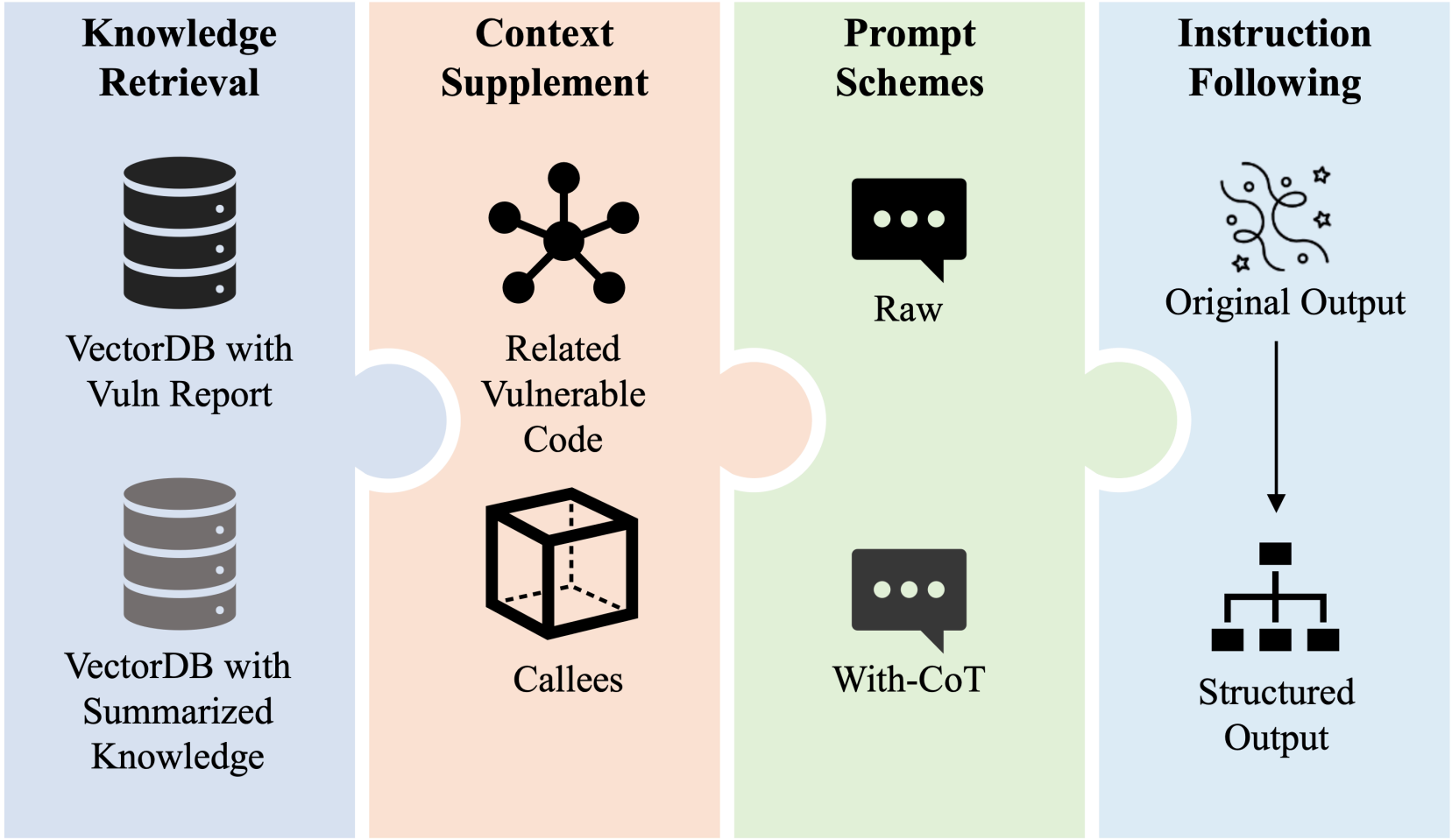
Analysis

- Let L be an LLM and T be target code (TC); the task is to decide whether T contains a vulnerability and identify its type and cause.
- L's performance does not solely depend on its pretrained reasoning capability, but on externally enhanceable factors:
  - Vulnerability Knowledge (K): Supplemental knowledge retrieved externally via similarity search from vulnerability knowledge bases.
  - Contextual Information (C): Additional code context around T, such as caller or related functions
  - Prompt Scheme (P): zero-shot, CoT, or role-playing.
  - Instruction-following Capability (I): ability to adhere to the output format for reliable automatic evaluation.

Analysis (Con't)

- Formally, the output R of the detection task can be viewed as a function:
  - $R = f_1(T, K, C, P, I)$.
- These components often entangle, making it hard to isolate L's inherent reasoning.
- Goal: decouple L's vulnerability reasoning from K, C, P, I to ask how much comes from the model vs external aids.

LLM4Vuln's four pluggable components



Knowledge Retrieval

- LLMs typically have a knowledge cutoff date for pre-training.
  - As a result, LLMs do not have up-to-date vulnerability knowledge
  - particularly crucial for detecting dynamically evolving logic vulnerabilities, such as those found in smart contracts.
- LLM4Vuln proposes two types of knowledge retrieval methods
  - Vulnerability reports + code embeddings.
  - GPT 4.1 summaries + functionality embeddings.

Vulnerability reports + code embeddings

- Collect original vulnerability reports along with the corresponding vulnerable code.
- Calculate their embeddings and create a vector database containing both the embeddings of the code and the associated vulnerability report.
- When a target code segment TC is provided, it can be used to directly search the vector database for the most similar code segments.
- After the retrieval process, only use the text of the vulnerability report, excluding the code, as raw vulnerability knowledge for subsequent analysis.

GPT 4.1 summaries + functionality embeddings

- Employ GPT-4.1 to summarize the vulnerability reports, which includes the functionality of the vulnerable code and the root cause of the vulnerability
- Calculate the embedding of this functionality part and create a vector database that contains only the functionality embeddings.
- Given a target code segment TC, use GPT-4.1 to first summarize its functionality and then use this extracted functionality to retrieve similar functionalities in the vector database.
- With the matched functionality, directly retrieve the corresponding vulnerability knowledge as summarized knowledge for further analysis.

Prompt for Summarizing the Function

Given the following vulnerability description, following the task:

1. Describe the functionality implemented in the given code. This should be answered under the section ”Functionality:” and written in the imperative mood, e.g., ”Calculate the price of a token.” Your response should be concise and limited to one paragraph and within 40-50 words.

2. Remember, do not contain any variable or function or expression name in the Functionality Result, focus on the functionality or business logic itself.

Prompt for Root Causes from Vulnerability Reports

Please provide a comprehensive and clear abstract that identifies the fundamental mechanics behind a specific vulnerability, ensuring that this knowledge can be applied universally to detect similar vulnerabilities across different scenarios. Your abstract should:

1. Avoid mentioning any moderation tools or systems.

2. Exclude specific code references, such as function or variable names, while providing a general yet precise technical description.

3. Use the format: KeyConcept:xxxx, placing the foundational explanation of the vulnerability inside the brackets.

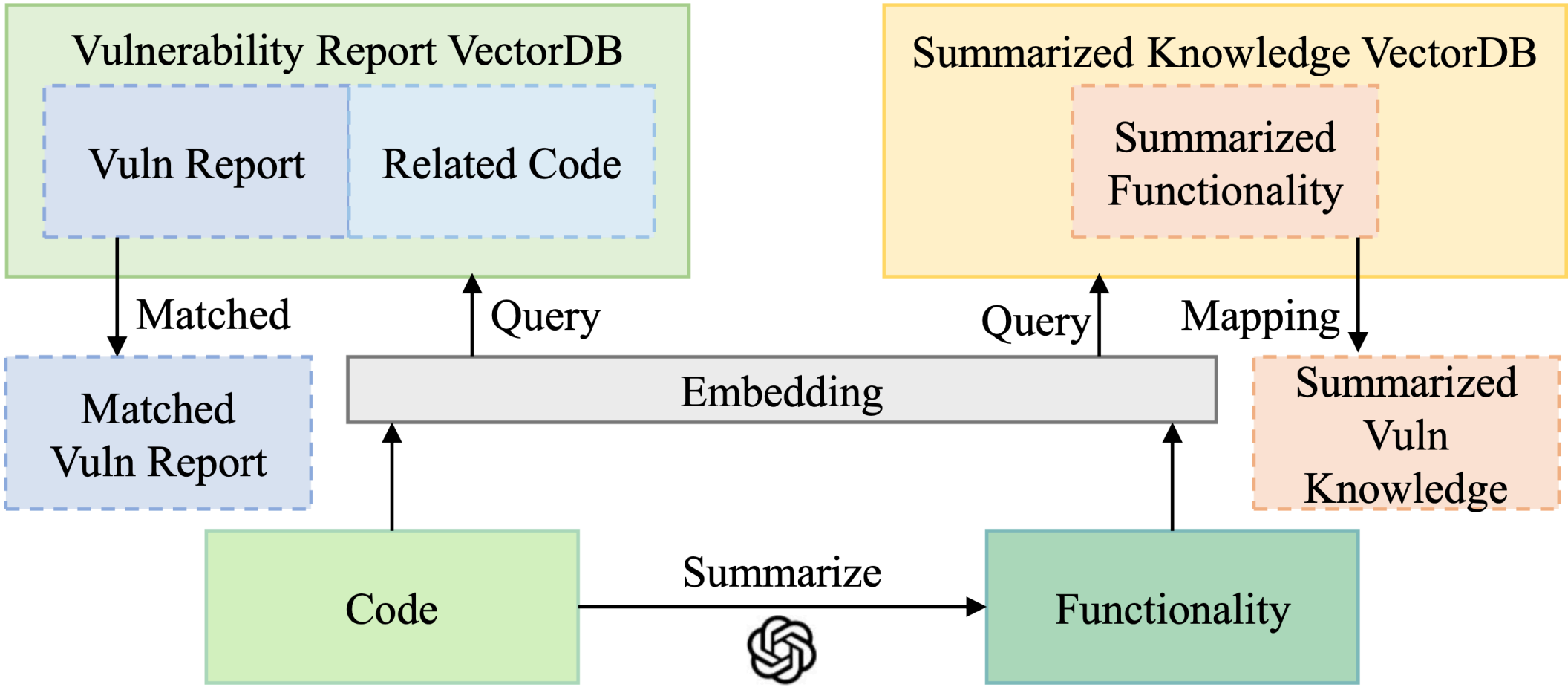
4. Guarantee that one can understand and identify the vulnerability using only the information from the VulnerableCode and this KeyConcept.

5. Strive for clarity and precision in your description, rather than brevity.

6. Break down the vulnerability to its core elements, ensuring all terms are explained and there are no ambiguities.

By following these guidelines, ensure that your abstract remains general and applicable to various contexts, without relying on specific code samples or detailed case-specific information.

Two types of vulnerability knowledge retrieval



Context Supplement

- Sometimes, the trigger of a vulnerability may be hidden in multiple functions or may require additional context to be detected.
- Even if the triggering logic is entirely within the given code segment, context may help a large language model better understand the function semantics.
- We provide two types of context supplements:
  - Extract all the functions that are mentioned in the vulnerability reports, such as those from Code4Rena and issues from GitHub.
  - Extract all the calling relation for all the samples, which can be used to provide a better understanding of the code.

Prompt Schemes

- Three types of knowledge usage:
  - i) LLM's own knowledge, ii) Raw knowledge, and iii) Summarized knowledge.
- Scheme 1 - Raw, simply ask LLMs to generate results without any specific instructions.
  - LLMs can use the APIs mentioned in earlier to retrieve related code segments (does not work for open-source models)
- Scheme 2 - CoT, request LLMs to follow chain-of-thought instructions before generating the result.
  - The LLMs should first summarize the functionality implemented by the given code segment, then analyze for any errors that could lead to vulnerabilities, and finally determine the vulnerability status.

Prompt Combination: Knowledge

Prefix 1 - LLM's own knowledge:

As a large language model, you have been trained with extensive knowledge of vulnerabilities. Based on this past knowledge, please evaluate whether the given smart contract code is vulnerable.

Prefix 2 - Raw knowledge:

Now I provide you with a vulnerability report as follows: {report}. Based on this given vulnerability report, pls evaluate whether the given code is vulnerable.

Prefix 3 - Summarized knowledge:

Now I provide you with a vulnerability knowledge that {knowl}. Based on this given vulnerability knowledge, evaluate whether the given code is vulnerable.

Prompt Combination: Scheme

Scheme 1 - Raw:

Note that if you need more information, please call the corresponding functions.

Scheme 2 - CoT:

Note that during your reasoning, you should review the given code step by step and finally determine whether it is vulnerable. For example, you can first summarize the functionality of the given code, then analyze whether there is any error that causes the vulnerability. Lastly, provide me with the result.

Instruction Following

- Since all outputs from LLMs are in natural language, they are unstructured and need summarization and annotation to derive the final evaluation results.
- Use GPT-4.1 to automatically annotate the outputs of LLMs.
- LLM4Vuln generates structured results based on the answers provided by different prompt schemes and LLMs, in two parts:
  - Whether the LLM considers the code to be vulnerable.
  - The rationale for its vulnerability or lack thereof.

Prompt for Instruction Following and Auto-Annotation

Generate Type and Description

I will give you some text generated by another LLM. But the format may be wrong. You must call the report API to report the result.

Compare Types between Output and Ground Truth

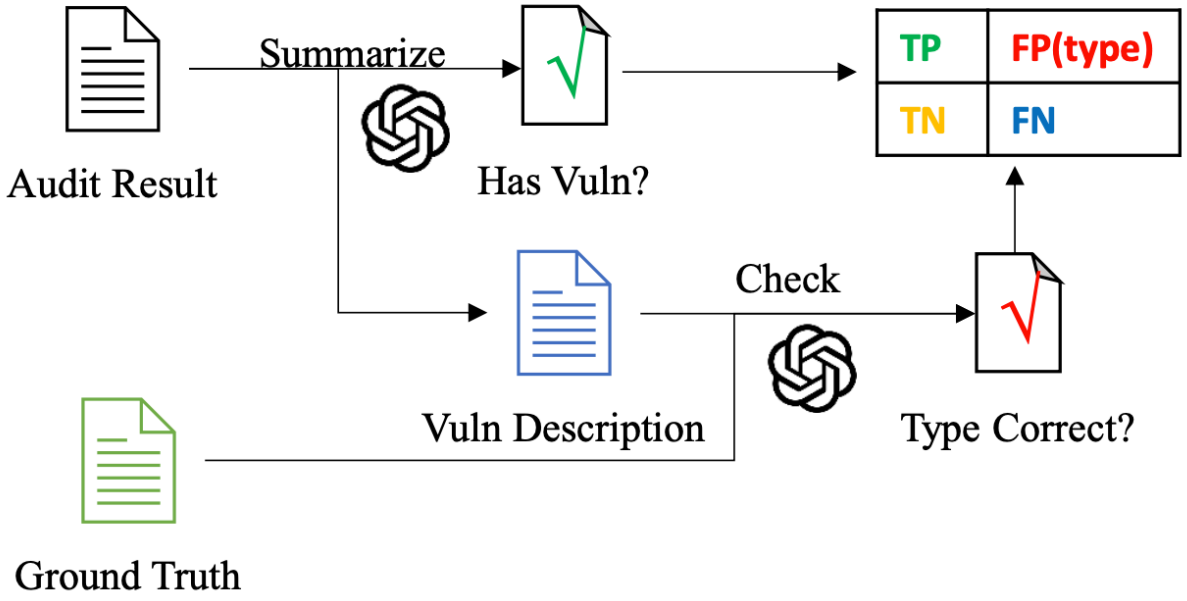
You are a senior code auditor. Now I will give you a ground truth of vulnerability, and a description written by an auditor. You need to help me identify whether the description given by the auditor contains a vulnerability in the ground truth. Please report the result using the function call.

Ground truth: {Ground Truth}

Description: {Output}

LLM-based Result Annotation and Analysis

- Annotation accuracy check (GPT-4.1)
  - Randomly sampled 100 cases from each programming language and manually checked the results.
- Vulnerable vs not, 100% accuracy.
- Vulnerability type matches ground truth, 81% (Solidity), 98% (Java), 97% (C/C++).



UniVul

- Previous datasets typically lack (i) explicit vulnerability knowledge retrievable for in-context learning, and (ii) surrounding code context to supplement understanding.
- Evaluates vulnerability reasoning across traditional programming languages (C/C++, Java) and smart contract languages (Solidity).
- Java/C/C++ vulnerabilities are tied to unsafe memory operations, insecure input handling, and unsafe deserialization.
- Solidity smart contracts are more susceptible to business logic vulnerabilities, which are significantly underrepresented in many LLM pre-training corpora due to the emerging and domain-specific nature of decentralized applications.

Data Used

- Solidity: 1,013 high-risk vulnerabilities from Code4rena GitHub issues (Jan 2021–Jul 2023) as the knowledge set.
  - Testing set has 51 vulnerable + 51 non-vulnerable segments from projects audited after Jul 2023, with non-vulnerable segments sampled for similar length/complexity.
- Java/C/C++: extract 77 CWE categories for Java and 86 for C/C++.
  - CWEs are high-level, hence prompt GPT-4.1 to generate 10 representative code examples per CWE, then generate audit-style reports → 770 knowledge items (Java) and 860 (C/C++).
  - 46 Java and 50 C/C++ vulnerabilities from CVE and BigVul, each paired with a patched/non-vulnerable version.

Prompt for Data Augmentation

Vulnerable Code Generator

[%CWE INFO%]

Base on the given CWE information, please help me generate 10 different vulnerable code snippets in [%LANGUAGE%] language. Each code snippet should be different from each other, and trying to cover as many as business logic as possible. You do not need to generate the description of the vulnerabilities, only the code is needed. For each codesnippet, please include it in a code block, which starts with ”“‘‘” and ends with ”“‘‘”.

Vulnerability Report Generation

[%CODE%]

The above code has [%CWE TYPE%] vulnerability. Please help me generate a vulnerability report for it. The report should include the following sections: 1) Vulnerability Description, 2) Vulnerable Code, 3) Root Cause, 4) Impact, 5) Mitigation. Each section should be clearly labeled and contain relevant information. The report should be concise and easy to understand.

Knowledge Retrieval setup

- Use FAISS to build the vector DB
  - Set top-K to retrieve the top-3 most relevant pieces of knowledge per query.
  - FAISS embeds the query, computes dot products with all vectors, and returns the top-K highest-dot-product vectors.
- 8,232 test cases per model = (147 vulnerable + 147 non-vulnerable cases) x (7 knowledge retrievals) x (2 prompt schemes) x (2 context variations).

Minimizing Data Leakage

- Some test cases may appear earlier than model cutoff dates (example: QwQ-32B trained on data before Nov 2024).
- Minimize the risk of data leakage from pre-training corpora by rewriting all testing data to prevent lexical overlap while preserving semantics.
- Use GPT-4.1 to rename function names, variable names, and comments while strictly maintaining the original program semantics.
  - Underlying code statements remain unchanged to preserve vulnerability behavior.

Statistics of the UNIVUL Benchmark

Dataset	Samples	Projects	Period
Solidity Knowledge set	1,013	251	Jan 2021 - Jul 2023
Solidity Testing set	51 + 51	11	Aug 2023 - Jan 2024
Java Knowledge Set	770	N/A	N/A
Java Testing Set	46 + 46	N/A	Jan 2013 - Dec 2022
C/C++ Knowledge Set	860	N/A	N/A
C/C++ Testing Set	50 + 50	N/A	Jan 2013 - Dec 2022

Evaluation

- Use default model configurations from providers.
- Set temperature to 0 for all models (except o4-mini, which does not support temperature settings).
  - Aim for the most deterministic results for reproducibility.

TABLE 3. SOLIDITY VULNERABILITY ANALYSIS RESULTS OF TP ■, TN ■, FP ■, FN ■, FP_T ■ UNDER DIFFERENT COMBINATIONS OF KNOWLEDGE (NK/OK/SK) & CONTEXT (C/N) FOR DIFFERENT LLMS.

M	Setup	Metrics	M	Setup	Metrics
GPT-4.1	NkC		o4-mini	NkC	
	NkN			NkN	
	OkC			OkC	
	OkN			OkN	
	SkC			SkC	
	SkN			SkN	
Llama-3-8B	NkC		DeepSeek-R1	NkC	
	NkN			NkN	
	OkC			OkC	
	OkN			OkN	
	SkC			SkC	
	SkN			SkN	
Phi-3-mini-128k	NkC		QwQ-32B	NkC	
	NkN			NkN	
	OkC			OkC	
	OkN			OkN	
	SkC			SkC	
	SkN			SkN	

TABLE 4. JAVA VULNERABILITY ANALYSIS RESULTS OF TP ■, TN ■, FP ■, FN ■, FP_T ■ UNDER DIFFERENT COMBINATIONS OF KNOWLEDGE (NK/OK/SK) AND CONTEXT (C/N) FOR DIFFERENT LLMS.

M	Setup	Metrics	M	Setup	Metrics
GPT-4.1	NkC		o4-mini	NkC	
	NkN			NkN	
	OkC			OkC	
	OkN			OkN	
	SkC			SkC	
	SkN			SkN	
Llama-3-8B	NkC		DeepSeek-R1	NkC	
	NkN			NkN	
	OkC			OkC	
	OkN			OkN	
	SkC			SkC	
	SkN			SkN	
Phi-3-mini-128k	NkC		QwQ-32B	NkC	
	NkN			NkN	
	OkC			OkC	
	OkN			OkN	
	SkC			SkC	
	SkN			SkN	

TABLE 5. C/C++ VULNERABILITY ANALYSIS RESULTS OF TP ■, TN ■, FP ■, FN ■, FP_T ■ UNDER DIFFERENT COMBINATIONS OF KNOWLEDGE (NK/OK/SK) & CONTEXT (C/N) FOR DIFFERENT LLMS.

M	Setup	Metrics	M	Setup	Metrics
GPT-4.1	NkC		o4-mini	NkC	
	NkN			NkN	
	OkC			OkC	
	OkN			OkN	
	SkC			SkC	
	SkN			SkN	
Llama-3-8B	NkC		DeepSeek-R1	NkC	
	NkN			NkN	
	OkC			OkC	
	OkN			OkN	
	SkC			SkC	
	SkN			SkN	
Phi-3-mini-128k	NkC		QwQ-32B	NkC	
	NkN			NkN	
	OkC			OkC	
	OkN			OkN	
	SkC			SkC	
	SkN			SkN	

RQ1: Effects of Knowledge Enhancement

- For traditional foundation models, knowledge enhancement substantially improves Solidity performance (business-logic-heavy vulnerabilities); F1 nearly doubles on average.
- For Java and C/C++, gains are limited or negative, likely due to stronger pre-training on CWE-like patterns.
- Deep reasoning models show less benefit from external knowledge, suggesting more robust internal modeling of vulnerability semantics.

RQ2: Effects of Context Supplementation

- Supplying local program context yields inconsistent improvements.
- Traditional foundation models tend to reach higher peak F1 when context is included.
- Deep reasoning models often perform better without context.
- Context can help, but can also distract if not well-aligned with the vulnerability.

RQ3: Effects of Different Prompt Schemes

- CoT prompting improves precision and reduces false positives for both model types.
- CoT's impact on recall varies by task.
- Deep reasoning models are more stable under CoT.
- Traditional foundation models show larger variance under CoT.
- CoT is particularly helpful for traditional models to bridge reasoning gaps in multi-step analysis.

RQ4: Testing for Zero-Day Vulnerabilities

- Deployed a Solidity-specific version of LLM4Vuln to an industry partner (a Web3 security company).
  - Because knowledge enhancement shows the most significant benefits for logic-based vuln.
- Used LLM4Vuln to audit four bug bounty smart contract projects: Apebond, Glyph AMM, StakeStone, and Hajime.
- Submitted 29 issues across the four projects; 14 were confirmed; Total bounty received: $3,576.

Conclusion

- Introduced LLM4Vuln, a unified and modular evaluation framework designed to decouple and enhance LLMs' inherent vulnerability reasoning.
- Support fair, extensible, and reproducible evaluations, described as the first benchmark providing retrievable knowledge and context-supplementable code across Solidity, Java, and C/C++.
- Applied LLM4Vuln to 294 vulnerable and non-vulnerable cases across 3,528 scenarios.
- Conducted a comprehensive study of how enhancement strategies—knowledge enhancement, context supplementation, and prompt schemes—impact LLM performance.

Acknowledgments

- [LLM4Vuln] LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning, Y. Sun and D. Wu and Y. Xue and H. Liu and W. Ma and L. Zhang and Y. Liu and Y. Li, arXiv, 2025.