

Universiteit Leiden

Master Computer Science

VulnReason-RAG:
Investigating the Effects of LLMs' Internal Reasoning and RAG on Vulnerability Detection

Name:	Jayme Hebinck
Student ID:	s2736136
Date:	04/07/2026
Specialisation:	Advanced Computing and Systems
1st supervisor:	Dr. ir. E. Makri MSc
2nd supervisor:	Dr. O. Gadyatskaya
TNO supervisor:	J. J. Kazemier MSc

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

This thesis examines the effect of LLMs' internal reasoning and Retrieval-Augmented Generation (RAG) on vulnerability detection. The exact effects of internal reasoning and RAG are not yet well established. We introduce the tool `VulnReason-RAG`, with which we inspect these two techniques separately and combined. We evaluate their effects on vulnerability detection with the developed tool using different evaluation perspectives: binary classification and CWE classification, tested on code snippets in an experimental dataset. Statistical tests using McNemar's test are performed to determine the reliability of our results. The results show that internal reasoning has a model-dependent, statistically significant effect on binary classification; more vulnerable code snippets are correctly predicted, although this decreases the number of correctly predicted safe code snippets. The largest impact can be seen with Gemini 2.5 Flash-Lite, for which the recall for vulnerable code snippets increases by 89.43% when internal reasoning is enabled. No statistically significant effects are observed for CWE classification performance. Additionally, our results indicate that RAG has inconsistent effects on vulnerability detection and classification using GPT-OSS 120B. Nevertheless, we observe promising but non-significant, reasoning-dependent patterns. These patterns suggest that the retrieval of knowledge, CVE data, CWE data, and the inclusion of an LLM re-ranker may have potential benefits for vulnerability detection.

Contents

1 Introduction **6**
- 1.1 Research Questions . . . 7
- 1.2 Contributions . . . 8
- 1.3 Thesis Overview . . . 8
- 1.4 Acknowledgments . . . 9

2 Background **10**
- 2.1 Vulnerabilities . . . 10
  - 2.1.1 CVEs . . . 10
  - 2.1.2 CWEs . . . 11
  - 2.1.3 Vulnerability Databases . . . 12
- 2.2 Large Language Models . . . 13
  - 2.2.1 LLM Architecture . . . 13
  - 2.2.2 Fine-tuning . . . 14
  - 2.2.3 Prompt Engineering . . . 14
  - 2.2.4 Chain-of-Thought Prompting . . . 14
  - 2.2.5 Reasoning Models . . . 15
- 2.3 Retrieval-Augmented Generation . . . 16
  - 2.3.1 RAG Framework . . . 16
  - 2.3.2 Retrieval Algorithms . . . 16
  - 2.3.3 LLM Re-ranker . . . 17
  - 2.3.4 Zero-shot and Few-shot . . . 18
- 2.4 Vulnerability Detection . . . 18
  - 2.4.1 LLMs for Vulnerability Detection . . . 19
  - 2.4.2 Finding the correct vulnerability . . . 19

3 Related Work **22**
- 3.1 Traditional Vulnerability Detection . . . 22
  - 3.1.1 Static Analysis . . . 22
  - 3.1.2 Fuzzing . . . 22
  - 3.1.3 Machine Learning- and Deep Learning-based Vulnerability Detection . . . 23
- 3.2 LLM-based Vulnerability Detection . . . 23

4 Methodology **27**
- 4.1 System Diagram . . . 27
- 4.2 Prompt Construction . . . 29
- 4.3 Vulnerability Data Source . . . 29
  - 4.3.1 Data Statistics . . . 30
- 4.4 Vulnerability Knowledge Base Pipeline . . . 32
  - 4.4.1 Data Filtering . . . 32
  - 4.4.2 Vulnerability Knowledge Base Construction . . . 33
  - 4.4.3 Vulnerability Knowledge Base Validation . . . 34
  - 4.4.4 Vulnerability Knowledge Base Statistics . . . 34
- 4.5 Vulnerability Detector Pipeline . . . 36
  - 4.5.1 Code Description Generation . . . 36
  - 4.5.2 Vulnerability Knowledge Base Retrieval . . . 36

4.5.3 Final Vulnerability Classification . . . 37

5 Experimental Evaluation **39**
5.1 Experimental Setup . . . 39
5.1.1 Model Selection . . . 39
5.1.2 Experimental Dataset . . . 40
5.1.3 Parameter Configuration . . . 40
5.1.4 Evaluation Metrics . . . 41
5.1.5 Statistical Significance Tests . . . 44
5.1.6 Hardware & Runtime Environment . . . 45
5.2 Experiment 1: Reasoning Influence . . . 47
5.2.1 Setup . . . 47
5.2.2 Results . . . 47
5.2.3 Statistical Significance . . . 51
5.2.4 Costs . . . 52
5.3 Experiment 2: Vulnerability Data Influence . . . 53
5.3.1 Setup . . . 53
5.3.2 Results . . . 53
5.3.3 Statistical Significance . . . 57
5.3.4 Costs . . . 59
5.4 Experiment 3: LLM Re-ranker Influence . . . 60
5.4.1 Setup . . . 60
5.4.2 Results . . . 60
5.4.3 Statistical Significance . . . 64
5.4.4 Costs . . . 65
5.5 Experiment 4: VKB Data Availability Influence . . . 67
5.5.1 Setup . . . 67
5.5.2 Results . . . 67
5.5.3 Statistical Significance . . . 71
5.5.4 Costs . . . 72

6 Discussion **74**
6.1 Discussion of Results . . . 74
6.1.1 Reasoning Influence . . . 74
6.1.2 RAG Influence . . . 75
6.2 Discussion of RQ . . . 77
6.3 Limitations and Threats to Validity . . . 78

7 Conclusion **81**

References **83**

Appendices **93**

A Prompts **93**
A.1 Vulnerability Knowledge Generation Prompt . . . 93
A.2 LLM-as-a-Judge Prompt . . . 94
A.3 Textual Representation Input Code Snippet Prompt . . . 95

A.4 LLM Re-ranker Prompt 96
A.5 Vulnerability Detection Prompt 97

B Experiment Evaluation Failure Analysis **98**
B.1 Failure Analysis Experiment 1 98
B.2 Failure Analysis Experiment 2 98
B.3 Failure Analysis Experiment 3 100
B.4 Failure Analysis Experiment 4 100

1 Introduction

Vulnerability detection forms an important topic in cybersecurity. Every year, thousands of vulnerabilities are identified and exploited [77]. This results in significant outages, data leaks, and other hacks. Although organizations and governments keep investing in cybersecurity, and new security solutions are constantly being proposed, ensuring complete safety of digital systems is complicated; hundreds of vulnerabilities can be successfully identified and patched, but one missed vulnerability can already lead to a complete system being compromised. Therefore, sophisticated techniques, such as fuzzing [10], are continuously developed to improve vulnerability detection. However, several limitations are present with existing methods, such as the complex process of generating valid inputs to detect and exploit vulnerabilities for fuzzing [10].

In recent years, a new vulnerability detection method has been emerging: vulnerability detection using Large Language Models (LLMs) [49]. This method is rapidly improving in performance because of the increased code comprehension and growing context windows of LLMs. Patterns within code can be more accurately captured, and this has a positive influence on the detection of weak or vulnerable code. Additionally, it helps mitigate some of the limitations of the more established vulnerability detection techniques. For example, the detection of vulnerabilities is no longer dependent on the generation of a valid input, as is the case with fuzzing. Therefore, recent vulnerability detection research applies LLMs to more effectively and efficiently detect vulnerabilities. Current research, as shown in a systematic review [49] from July 2025, covers a variety of subtopics of vulnerability detection using LLMs. These encompass the various specific tasks that come with vulnerability detection, the input representation used in prompts, the system architecture of detection systems, and the adaptation techniques that are used to improve the detection rates. Two of these adaptation techniques are internal reasoning and Retrieval-Augmented Generation (RAG). Internal reasoning focuses on the 'thinking' process that an LLM goes through during the generation of an answer. It is the process that takes place between the original prompt of the user (excluding the prompt) and the final output (excluding the final output) [50]. This allows for reflection on, verification of, and, when needed, correction of previously generated tokens. RAG refers to the process of retrieving relevant data from an external database or knowledge base, which is incorporated into the input prompt of an LLM for the generation of more accurate content [36]. To improve the relevance of retrieved information, RAG systems can include an additional re-ranking step using, for example, an LLM. Consequently, the LLM re-ranks retrieved entries based on semantic relevance.

Several studies show that RAG by itself has a positive influence on vulnerability detection. One such study is Vul-RAG, introduced by Du et al. [30], in which the authors show that using knowledge-level RAG for vulnerability detection using high-level vulnerability knowledge is a promising direction. However, several limitations exist in Vul-RAG and other similar approaches, such as the study by Liu et al. [63], which investigates the effect of RAG with retrieval using term-matching algorithms. These studies do not analyze and compare the effects of RAG, including an LLM re-ranker step, in models for which internal reasoning can be disabled and enabled. Consequently, the influence of these two adaptation techniques, internal reasoning and RAG, in a full vulnerability detection pipeline is not yet well established. Additionally, experiments are often performed with the risk of LLM contamination [20]: it is uncertain whether the LLM has been trained on the data it is experimented on, which can significantly impact performance.

To address these gaps and limitations, we propose a new vulnerability detection tool: `VulnReason-RAG`. Using this tool, two open problems are examined and addressed: (1) we analyze the effects of internal reasoning and knowledge-level RAG, including an LLM re-ranker step, on vulnerability detection, and (2) to facilitate a reliable analysis, we construct a vulnerability knowledge base (VKB) based on real-world vulnerabilities. These are collected and filtered from the MoreFixes database [3], which is, to the best of our knowledge, the largest vulnerability database with fix commits that can collect new vulnerabilities with associated fixes. We also construct an experimental dataset to evaluate `VulnReason-RAG` on vulnerabilities identified exclusively after the knowledge cut-off dates of the used LLMs to prevent LLM contamination. Additionally, we equip `VulnReason-RAG` with a pipeline to create and update a VKB based on the MoreFixes database to enable future research and practical application in the industry.

1.1 Research Questions

Our main research question (**RQ**) is: "How do internal reasoning and RAG affect vulnerability detection and classification performance of LLMs?"

To answer our main research question, we investigate four sub-research questions:

- **SRQ1**: "How does internal reasoning in LLMs, in combination with a RAG pipeline, influence vulnerability detection and classification?"
- **SRQ2**: "What elements from the vulnerability knowledge base lead to the highest quality vulnerability detection and classification?"
- **SRQ3**: "How does an LLM re-ranker affect the retrieval process of RAG for vulnerability detection and classification?"
- **SRQ4**: "How does vulnerability knowledge base data availability for the RAG pipeline influence vulnerability detection and classification?"

For every SRQ, we perform a separate experiment. We perform these experiments using incremental ablation studies. An ablation study is a controlled experiment in which components of a system are systematically removed or modified. The concept has commonly been used in evaluating artificial neural networks (ANNs), where components such as trainable weights are removed to determine their contribution to the overall system performance [70]. In this thesis, we perform incremental ablation studies, meaning that instead of removing or modifying a component from a complete system, we add it to a baseline system. This enables analysis of the contribution of each component to the system performance.

To answer **SRQ1**, we determine the influence of internal reasoning in combination with a full RAG pipeline using three different LLMs. All three different LLMs support disabling and enabling internal reasoning, allowing us to observe and analyze the effects of internal reasoning on the LLMs.

To answer **SRQ2**, we investigate the additional retrieval of data types from the VKB and determine which elements lead to the best-performing RAG pipeline.

To answer **SRQ3**, we compare the vulnerability detection rates with only the initial retrieval stage to using both an initial retrieval stage and an LLM re-ranker. Based on the resulting differences, we assess how the LLM re-ranker influences the RAG pipeline.

To answer **SRQ4**, we analyze the effect of incrementing the VKB with new vulnerability data. Using three different temporal snapshots of the VKB, we limit the amount of new data the vulnerability detection pipeline has access to. We compare the differences in performance and investigate the overall effect.

1.2 Contributions

In this research, we make four contributions:

- We show the effects of LLMs' internal reasoning and RAG, combined and separate, on vulnerability detection.
- We determine which elements from the MoreFixes database are more suitable for inclusion in a VKB.
- We investigate the effect of an LLM re-ranker on the RAG retrieval process.
- We design a novel vulnerability detection tool, `VulnReason-RAG`, with an updatable VKB.

These contributions can impact the research area of vulnerability detection by providing researchers with an overview of the effects of internal reasoning and RAG, and how an effective configuration with these adaptation techniques could take form. Additionally, we provide researchers with an updatable VKB that can retrieve newly discovered vulnerabilities. Consequently, the limitation that static databases do not contain enough recent vulnerabilities to adequately determine whether a developed tool performs well in a realistic environment is mitigated.

All artifacts (complete VKB, experimental dataset, gathered outputs, generated results, and source code for `VulnReason-RAG`) are publicly available [44].

1.3 Thesis Overview

The current chapter is the introduction of this thesis. Chapter 2 gives a background of all the topics that are discussed. This includes vulnerabilities, LLMs, RAG, and vulnerability detection. Afterwards, Chapter 3 states the related work that has been done in this area. Chapter 4 discusses the architecture and the implementation of `VulnReason-RAG`. In Chapter 5, we describe the experiments we perform and analyze the results. After our experiments, in Chapter 6, we discuss and answer our research questions, and state the limitations and threats to the validity of our research. Finally, in Chapter 7, we conclude the thesis and discuss potential future research.

This master's thesis is supervised by Dr. ir. E. Makri MSc and Dr. O. Gadyatskaya and is part of the Leiden Institute of Advanced Computer Science (LIACS). In addition, this thesis is supervised by J. J. Kazemier MSc at De Nederlandse Organisatie voor toegepast-natuurwetenschappelijk onderzoek (TNO), which also provides funding and research support.

1.4 Acknowledgments

First and foremost, I would like to thank my supervisors from Leiden University and TNO: Dr. ir. Eleftheria Makri MSc, Dr. Olga Gadyatskaya, and Jan Kazemier MSc. Eleftheria, your guidance, support, and advice have helped me tremendously throughout this thesis. Every meeting gave me new hope and energy. Olga, your knowledge and advice have been very valuable to me. No matter the situation, you always have a solution in mind. Jan, your expertise from a more practical perspective has taught me not just to look at my work as a theoretical framework, but also as a practical foundation for future opportunities. To my fellow interns and colleagues at TNO, especially at the CST department, I thank you for all the days we have spent together. I cannot recall how many coffee breaks I have shared with you, but my new caffeine addiction has been more than worth it. In addition, I would like to thank my parents, my brother, and my partner. You have been there for me through thick and thin. Without your support, I would not have been able to do this. You mean the world to me. Finally, a big thank you to my friends in Leiden, the Netherlands, and all over the world. Especially to Rajeck and Teun. I have been able to learn so much with and from you these past years. I will always value those times.

2 Background

In this chapter, we present essential background information and definitions relevant to this thesis. We cover vulnerabilities, how they are represented, and where they are stored. We examine LLMs, their underlying principles, and how they operate. Finally, we inspect the implementation of RAG in LLMs and the application of LLMs to vulnerability detection.

2.1 Vulnerabilities

In this section, we explain what vulnerabilities are, how they are represented, and how they are stored once identified. We first examine the CVE Program, in which vulnerabilities are represented as CVE Records. We then explain CWEs, which describe the common weaknesses that cause vulnerabilities. Finally, we inspect how vulnerabilities are stored in databases.

2.1.1 CVEs

CVE stands for Common Vulnerabilities and Exposures. CVEs form a catalog of disclosed cybersecurity vulnerabilities. Every CVE Record represents one vulnerability [112], which may be present in or across multiple files and programming languages.

Each CVE Record includes at least the following:

- CVE ID: the letters 'CVE', appended by the year in which the vulnerability is found, appended by four or more digits. E.g. 'CVE-2026-1234';
- Brief description of the vulnerability;
- Affected products and versions;
- References to vulnerability reports and advisories.

Optional data, such as severity scores, credits, additional (programming) languages, affected products, community contributions, etc., may be added to provide further context [23].

The severity score of a CVE is represented using the Common Vulnerability Scoring System (CVSS) [22]. This produces a numerical score based on the severity of the vulnerability. It covers several metrics, including the ease of exploiting the vulnerability and the impact on the confidentiality, integrity, and availability of the affected component if the vulnerability is exploited. The CVSS metrics are also represented as a vector, which describes the metric values that are used to determine the final score. Additionally, it is mapped to a qualitative representation (i.e., none, low, medium, high, critical).

All CVE Records found through the CVE Program are maintained in the CVE List [21]. This catalog is maintained by The MITRE Corporation, funded by the U.S. Department of Homeland Security (DHS) and the Cybersecurity and Infrastructure Security Agency (CISA), and serves as the main source for newly discovered vulnerabilities. In addition, several other databases provide enriched information about these vulnerabilities. One such enrichment database is the NVD database [76], which incorporates all newly published CVEs and enriches them by adding Reference Tags, more severity scores (e.g., scores based on different CVSS versions),

and additional CWE and CPE information. CPE stands for Common Platform Enumeration and refers to a structured naming scheme for software, systems, and packages [24], making it clear what exact platform is affected by the vulnerability. Another relevant database is the GHSA [40], which covers the CVEs and GitHub-originated security advisories that affect open-source projects on GitHub. This database includes a subset of multiple CVE databases, relevant for open source software projects.

2.1.2 CWEs

CWE stands for Common Weakness Enumeration. CWEs form a community-developed list of common weakness types. They define a type of vulnerabilities that have the same underlying weakness. Multiple CVEs or unidentified vulnerabilities can be mapped to these weakness types. Consequently, CWEs provide a standardized method to describe the type of weakness that results in a vulnerability [26].

A CWE is assigned a unique number as an ID. It is then followed by a descriptive name for the weakness. CWEs frequently provide the following information:

- Name: including intended behaviour, mistake, affected resource, affected technology (if relevant);
- Summary: one or two sentences that describe the weakness;
- Extended Description: one or two paragraphs describing how the weakness is a problem;
- Modes of Introduction: how and when the weakness may be introduced;
- Potential Mitigations: how to eliminate/reduce the frequency or impact of the weakness;
- Applicable Platforms: affected programming languages, operating systems, architectures, technologies, etc.;
- Demonstrative Examples: illustrate the weakness through code, text, and/or diagrams;
- Observed Examples: publicly reported vulnerabilities in real-world products that exhibit the weakness, such as CVEs;
- Relationships: other CWEs related to the weakness;
- References: citations for academic papers, white papers, etc., that describe the weakness.

Often, a CVE is assigned a CWE upon being found, which classifies the weakness that lies at the core of the vulnerability. However, at times, a precise CWE cannot be found, or the analysts do not have the time to identify the right CWE. The label `NVD-CWE-noinfo` or `NVD-CWE-Other` is then assigned to the CVE.

2.1.3 Vulnerability Databases

To use previously found vulnerabilities in the above processes, they are stored in vulnerability databases. These databases typically contain entries that represent either specific CVEs or specific patch files. Entries typically contain supplementary information, such as additional CVE information, CWE information, vulnerable code, patched code, and code differences. Especially the inclusion of vulnerable code and patched code makes these databases useful for vulnerability analysis.

Additional enhancements increase the usability of vulnerability datasets. Knowledge in the form of, e.g., a vulnerability report, including an analysis of vulnerable and patched code, can be used to more effectively retrieve relevant vulnerabilities in a RAG pipeline [35]. When a vulnerability database includes additional generated knowledge, we refer to it as a vulnerability knowledge base (VKB).

There exist various vulnerability databases and knowledge bases. These can be classified as static databases and dynamic databases.

Static Databases. Static databases [9, 11, 57, 91, 121, 32, 120, 16, 8, 75, 97, 19, 43, 105, 28, 30, 55] are databases that consist of manually curated or synthetic entries. The data cannot be updated or appended with new data. Therefore, they face the risk of being outdated due to the increasing number of vulnerabilities found every year. This can reduce the quality of tools using these databases and knowledge bases in the long-term. For example, Fan et al. introduce Big-Vul [32], which contains vulnerabilities found between 2002 and 2019, but not from after 2019, leading to a gap in vulnerabilities between 2019 and the current date. In this period, new vulnerabilities with new or different weaknesses may be found, which could be needed for the understanding of more recent vulnerabilities.

Dynamic Databases. Dynamic databases mitigate the risk of outdated static vulnerability databases and knowledge bases. Tools are provided to update and append them with new data. For example, ReposVul, introduced by Wang et al. [108], is an automatic framework that crawls open-source vulnerabilities from Mend [69], which includes CVEs from the NVD, aggregated from vulnerabilities reported in security advisories and open-source projects' issue trackers. ReposVul stores the programming language, CWE type, CVSS scores, and fixes, and covers 6134 CVEs. MegaVul [74] collects vulnerabilities from the NVD and contains a total of 8254 CVEs. Information about the vulnerabilities is given over 6 dimensions, including the vulnerable and patched code. CVEfixes, introduced by Bhandari et al. [8], automatically fetches all available CVE Records. Additionally, it scrapes enriched CVEs from the NVD, covering the vulnerable code, fixes, specific fix commits, and repository information of the CVEs. CVEFixes consists of 5365 CVEs with fixes. MoreFixes, introduced by Akhoundali et al. [2], follows the structure of CVEfixes, but includes, besides the NVD, additional vulnerabilities from the GitHub Security Advisory Database (GHSA) [40], leading to a 397% increase in CVEs with fixes compared to CVEfixes, covering 26617 CVEs with fixes. To the best of our knowledge, this is the largest vulnerability database with fix commits currently available.

2.2 Large Language Models

In this section, we discuss LLMs, how they function, and several optimization techniques.

2.2.1 LLM Architecture

LLMs are AI systems that can process and generate text, and can give coherent answers to instructions in an input prompt [73]. They are pre-trained on terabytes (TBs) of data, both scraped from the public internet and from private data. This enables LLMs to find patterns and correlations within language. This learned knowledge allows LLMs to give accurate answers on a wide range of tasks, without having to be fine-tuned for these tasks, provided that detailed descriptions of the tasks are given as input. Additionally, a core factor that influences the recognition of patterns and correlations is the number of parameters that are used to pre-train the LLMs. This can be up to hundreds of billions to trillions of parameters; these parameters represent weights, numerical values determining the importance of inputs, and biases, constant values added to values from previous layers in the network [6].

The architecture of LLMs uses the decoder-only Transformer model [61, 87]. It generates and decodes a tokenized answer based on a prompt into human-readable text. These models can take into account a window of sequential tokens, while preserving the context between these sequential tokens. Transformer models use attention, referring to the ability to prioritize more relevant parts of input data, while paying less attention to less relevant data. The Transformer models can determine relevant tokens themselves, using self-attention. Contextual representations of the meaning of tokens are built by paying attention to surrounding tokens, with which it can be determined what tokens are relevant and which are not relevant [7].

The total number of tokens that can be taken into account is called the context window. Within the context window, the context between different tokens can be memorized, and the attention function can be executed. The context of the tokens outside of the context window is not taken into account, leading to a significant decrease in performance. The context window, and thus the input length, is limited due to the maximum length of training data that an LLM is trained on [29]. Longer context windows require significantly more time and computation.

The self-attention mechanism is used to learn the context of language, and the parameters of LLMs are trained on the output of this self-attention process. It encapsulates the context between different tokens, and thus different words and sentences. When an LLM is given an input, it predicts each next token based on the parameters on which the LLM was trained, generating an output. The coherence of the output depends on the number of parameters the LLM architecture has. When the LLM architecture has more parameters, it can capture the context of language more precisely, leading to better functioning, but also a longer and computationally more expensive training process and usage. This leads to an important trade-off when pre-training an LLM, and results in several LLM vendors pre-training different versions of their LLMs with different numbers of parameters to provide access to more users. For example, the model Qwen3-VL by Qwen [86] has versions with 235, 30, 8, 4, and 2 billion parameters, where the version with 235 billion parameters is the most computationally expensive, but best-functioning, and the version with 2 billion parameters is computationally cheaper but at the cost of functionality.

2.2.2 Fine-tuning

The pre-training of a model is necessary to learn patterns and correlations within natural language to infer contextual relationships among terms and phrases. The LLM is trained on a variety of data. Therefore, LLMs can adapt to several tasks without issues, making the pre-trained model effective for these tasks. However, for certain domain-specific tasks, the model may benefit from fine-tuning. This is done by further training the LLM on task-specific data, adjusting the parameters to generate an improved output for this specific task [81]. This process is smaller and faster than pre-training, since it utilizes the pre-trained model as a base. There are several ways to perform fine-tuning, such as unsupervised fine-tuning and supervised fine-tuning [81].

Unsupervised fine-tuning uses data that is not labeled. Like during pre-training, the LLM is exposed to a large quantity of data to refine the current patterns and correlations it recognizes. This can be useful for new domains, such as in the legal field. Supervised fine-tuning uses data that is labeled, which is often significantly less available since it requires human verification for correctness. Supervised fine-tuning can be used for more precise tasks, such as classification tasks, which require more learning of context between a term or phrase and a given class; something the pre-trained model may not have been trained on before.

2.2.3 Prompt Engineering

LLMs are instructed via prompts. These prompts contain a set of instructions, making the LLM more effective at performing the task at hand. When a prompt is shallow and unclear, it may lead to an inadequate answer. Prompts can be improved by changing wording and adding additional expectations. We refer to this as prompt engineering. The goal of prompt engineering is to systematically enhance the output of an LLM without modifying the core model parameters, using methods such as Chain-of-Thought prompting (see Section 2.2.4) and few-shot prompting (see Section 2.3.4) [93].

Additionally, LLMs are highly sensitive to the construction of prompts [4]. To improve the structure of prompts, White et al. [113] provide a framework for prompt structures that can be adapted in several domains. The framework consists of 16 prompt patterns, spread over 6 pattern categories. Table 1 shows an overview of the prompt patterns with a brief description.

2.2.4 Chain-of-Thought Prompting

To further enhance prompts, CoT instructions can be added [109]. CoT refers to requesting the LLM, through the prompt, to think in steps. It allows models to split a problem into subproblems, which can be solved independently. The answers to these subproblems can then be used to reflect on other answers and construct the final answer. Additionally, CoT can be used to provide the LLM with the possibility to fully reflect on its final answer, giving it the chance to debug and potentially correct the answer, if it realizes it reached an incorrect conclusion.

Pattern Category	Prompt Pattern	Description
Input Semantics	Meta Language Creation	Creating a custom language for the LLM to understand.
Output Customization	Output Automater	Create scripts that can automate tasks the LLM output suggests.
	Persona	Gives LLM a persona or role.
	Visualization Generator	Producing visualizations in other tools based on textual outputs.
	Recipe	Obtain a sequence of steps to realize the stated result.
	Template	Specify a template for the output.
Error Identification	Fact Check List	Requires LLM to generate a list of facts the output depends on.
	Reflection	Requires LLM to introspect on its output.
Prompt Improvement	Question Refinement	LLM always suggests a better version of the user's question.
	Alternative Approaches	Requires LLM to suggest alternative ways to accomplish the task.
	Cognitive Verifier	Instructs LLM to suggest subquestions for the user to answer.
	Refusal Breaker	Requires LLM to reword the user's question when it refuses to answer.
Interaction	Flipped Interaction	Requires LLM to ask questions instead of generating output.
	Game Play	Requires LLM to generate output in the form of a game.
	Infinite Generation	Requires LLM to generate output indefinitely.
Context Control	Context Manager	Allows user to specify context for output of LLM.

Table 1: Prompt Pattern Framework by White et al. [113].

2.2.5 Reasoning Models

Because of the effectiveness of CoT prompting, more comprehensive approaches to giving LLMs the ability to reason are being researched. One such approach is the use of reasoning models to enable internal reasoning [116]. Reasoning models introduce the concept of 'thought' as intermediate, internal steps in tokens before generating the final output. This incorporation of intermediate reasoning steps allows the LLM to split the problem into multiple steps and allows for comprehensive reflective reasoning steps.

To achieve these reasoning capabilities, reasoning models are fine-tuned specifically on CoT prompts [116]. Supervised fine-tuning is one of the primary methods to achieve improved reasoning capabilities, particularly in domain-specific areas. Fine-tuning a reasoning model on domain-specific reasoning traces leads to better reflection on intermediate answers and debugging if the LLM reaches an incorrect conclusion.

Another method that shows significant improvement in reasoning capabilities in reasoning models utilizes reinforcement learning [118]. It shows greater generalization than supervised fine-tuning, especially on advanced reasoning behaviors such as reflection and verification. This happens because supervised fine-tuning guides the reasoning model to memorize the supervised labels, whereas reinforcement learning rewards the reasoning behavior of the model. The approach uses policies to reward the LLM for advanced reasoning behavior, based on both unsupervised and supervised data. The DeepSeek R1 reasoning model is a recent example that uses reinforcement learning to improve reasoning and is currently one of the best-performing reasoning models.

These methods for enhancing the reasoning capabilities of LLMs can distinguish reasoning models from non-reasoning models, allowing for potentially improved behavior in several fields in which answer reflection and verification are of importance. Some models have a toggle to disable or enable internal reasoning behavior. Consequently, the effects of this reasoning behavior can be isolated.

2.3 Retrieval-Augmented Generation

In this section, we discuss RAG. We examine different retrieval algorithms and re-ranking mechanisms, and investigate the difference between zero-shot and few-shot.

2.3.1 RAG Framework

RAG refers to the retrieval of information from a non-parametric memory to a parametric-memory generation model, such as an LLM [54]. This non-parametric memory, which we refer to as a knowledge base from this point onward, usually contains a vast amount of data that can be utilized for a specific task. Using a retrieval algorithm and potentially a re-ranker, based on a given prompt, the most relevant entries in the knowledge base are retrieved. These relevant entries are then added to the prompt of the LLM, which takes this information into the context and can use it to answer the prompt. Thus, even though an LLM may not be fine-tuned for this specific task, it can still answer the prompt more accurately while generating less factually incorrect content [36]. Therefore, RAG is especially effective for domain-specific and knowledge-intensive tasks. More benefits of RAG over fine-tuning an LLM include the immediate updates that can be made to the knowledge base. On the contrary, with a fine-tuned LLM, this new information is not immediately available, as the LLM needs to be retrained for this; a process that not only takes time, but is also computationally expensive. This makes RAG suitable for tasks in which data is often added and changed [36].

2.3.2 Retrieval Algorithms

Several retrieval algorithms can be used in the RAG pipeline, all having different benefits and disadvantages. An algorithm that is often used in the RAG pipeline is BM25, introduced by Robertson et al. [90], due to its speed and robustness. It incorporates important aspects that could make a document relevant, given a set of documents, namely the term frequency and inverse document frequency. In the following paragraphs, we discuss important components on which BM25 is built, and how BM25 mitigates the shortcomings of these individual components.

Term frequency (TF). The term frequency refers to the number of occurrences a given term t appears in a document d [89]. The higher the term occurrence, the more relevant it is. However, this could also mean the term is a stopword, such as 'and', 'or', etc. The formula for TF is:

$$tf(t,d) = f(t,d) \tag{1}$$

In which $f(t,d)$ represents the frequency of term t in document d.

Inverse document frequency (IDF). The inverse document frequency refers to the number of documents in the collection in which a given term t is found [89]. The lower the number of documents, the more relevant this term is, because it indicates that this term is only used in specific contexts that may match the context of the term in question. The formula for IDF is:

$$idf(t) = \log\frac{N}{n_i} \tag{2}$$

In which t represents a given term, N represents the number of documents in the collection, and n_i represents the number of documents in which the given term is present.

TF.IDF. To combine the strengths of both term frequency and inverse document frequency, the two have been combined into TF.IDF [88]. For a given term, the TF.IDF score is high if the term is present in as few documents as possible, and in these few documents, the term occurs as much as possible. Therefore, the issue with high term frequency of stopwords is mitigated, and only relevant terms get a higher score. The formula for TF.IDF is:

$$tf.idf(t, d, N) = tf(t, d) \times idf(t) = f(t, d) \times \log \frac{N}{n_i} \tag{3}$$

In which t represents a term, d represents a document, N represents the collection of documents, and n_i represents the number of documents that term t is present in. However, TF.IDF has some shortcomings, such as that it does not take into account document length; it does not matter if a term is present in a long document or in a short document, both documents get the same relevance score.

BM25. BM25 [65] is an improved version of TF.IDF. It incorporates mitigations against shortcomings of TF.IDF. It introduces two new parameters: b and k_1. Parameter b determines the scaling by document length, where $b = 1$ corresponds to fully scaling by document length, and $b = 0$ corresponds to no document length normalization. Parameter k_1 tunes the document term frequency scaling: $k_1 = 0$ corresponds to a binary model: the term is either present in the document or not, and the relevance score does not increase if the term occurs more than once. A larger value for k_1 corresponds to the raw term frequency of the given term: the more occurrences of a term in the document, the higher the score.

$$RSV_{BM25} = \sum_{t \in q} \log \frac{N}{df_t} \times \frac{(k_1 + 1) tf_{td}}{k_1((1 - b) + b \times (\frac{L_d}{L_{ave}})) + tf_{td}} \tag{4}$$

Here, RSV stands for Retrieval Status Value, meaning a value that indicates the relevance of a document for the whole query. q is a given query and t a term in this query. N is the total number of documents in the collection, and df_t is the number of documents in which term t is found. k_1 and b are the above-described parameters, tf_{td} the frequency of term t in document d, L_d the length of document d and L_{ave} the average document length for the whole collection. The RSV is computed by performing BM25 on all query terms for a query - document pair. The document resulting in the highest RSV score is considered the most relevant document.

2.3.3 LLM Re-ranker

BM25, described in Section 2.3.2, is an effective and efficient retrieval algorithm. However, it does not account for a number of important aspects of information retrieval:

- Semantic meaning: BM25 is a term matching algorithm, meaning it matches the presence of exact terms. Nevertheless, there are situations in which multiple terms can be used to describe the same situation. Consider, for example, the documents 'Bike lane' and 'Bicycle lane', and the query 'Bike'. In this scenario, 'Bike lane' would obtain a higher retrieval score than 'Bicycle lane', since exact term matching is used, even though both terms refer to the same object.

- Sentence order: BM25 does not take into account the order of a sentence; documents are treated as a bag of words [85], meaning the retrieval score is determined on the words independently from each other, and not based on the context between the words.

To mitigate these limitations, an LLM re-ranker can be applied after the initial retrieval stage. As described in Section 2.2.1, LLMs take into account the context of a prompt, since they are based on Transformer models. This enables LLMs to perform self-attention.

Since this re-ranking process takes more time and is computationally more expensive, it is often only performed on the initial top-l knowledge entries retrieved by an initial retriever, such as BM25, and retrieves a final top-k knowledge entries using the re-ranker. This approach balances functionality and speed, which results in a retrieval process that takes both exact term matching and the context of the terms into account for a given query and document.

Often, listwise re-ranking is used to perform the re-ranking process. This takes as input into an LLM prompt the top-l knowledge entries, and requests the LLM to sort the entries based on relevance to the input. This returns the list of re-ranked entries, out of which the top-k is taken. An additional benefit is that the context between different entries is also captured, which allows for a more thorough comparison between the entries themselves, besides scoring the individual entries.

Alternatively, pointwise or pairwise re-ranking can be performed. Pointwise re-ranking determines a relevance score for every initially retrieved entry, in separate prompts to prevent unwanted influence from other entries. Pairwise re-ranking determines the entry that has a higher relevance between every pair of knowledge entries, and uses this information to construct the final re-ranking. However, both of these methods require significantly more LLM requests than are needed for listwise re-ranking. They are thus seen as less efficient and more expensive.

2.3.4 Zero-shot and Few-shot

In some cases, an LLM may not be sufficiently trained for a specific task. Instead of fine-tuning the LLM on the specific task, which can be costly, the LLM can be given a small number of examples in the prompt that describe how it should determine and construct its answer. This is referred to as few-shot in-context learning [12], commonly known as few-shot prompting. When no examples are provided, the LLM relies solely on its own pre-trained knowledge. This is referred to as zero-shot prompting. Few-shot prompts can be created with the use of RAG: through the RAG pipeline, relevant entries are retrieved, and these can then be added to a prompt to form examples. With few-shot prompting, the prompt increases in tokens, but may return improved results.

2.4 Vulnerability Detection

In this section, we examine how different techniques can be used to detect vulnerabilities. First, we discuss how LLMs are applied to vulnerability detection. Afterwards, we explain two forms of vulnerability detection: binary detection and the classification of a vulnerability.

2.4.1 LLMs for Vulnerability Detection

In addition to large text datasets, recent LLMs are also trained on extensive code bases. Therefore, they learn the patterns and correlations between code and can infer contextual relationships between parts of code as well as between parts of text. This additional pattern recognition within code improves the LLM in tasks such as code generation and determining the functional correctness of code.

The overall improvement in code comprehension can be seen in several benchmarks, such as HumanEval [17]. This benchmark consists of 164 original programming problems. The lowest-scoring model, Grok-1.5 by xAI [115], released in March 2024, scores 0.741 out of 1. However, the highest-scoring LLM, Kimi K2 0905 by Moonshot AI [104, 72], released in September 2025, has a score of 0.945 out of 1, which demonstrates significant improvements over 1.5 years.

Based on LLMs' performance on the two tasks mentioned above, they are increasingly capable of generating and analyzing functional code. This improved code comprehension shows promise for vulnerability detection, as also demonstrated in prior work such as Vul-RAG [30]. By analyzing the behavior of code and determining contextual relationships between, e.g., several system calls, an LLM may be able to recognize unsafe code patterns. Such patterns could subsequently be connected to specific CWE labels.

In addition, vulnerability detection and classification performance can be further enhanced using the LLM optimization techniques discussed in Section 2.2.

2.4.2 Finding the correct vulnerability

Vulnerability detection tools take as input a code snippet and determine whether it is vulnerable or not. However, this process can differ between different tools. A key distinction is whether the tool only performs binary detection or whether it can also classify the specific vulnerability that is found.

Binary labeling. Tools that perform binary labeling take as input a code snippet, analyze it, and then return to the user a label of whether this code is vulnerable. It only returns this, and no further information is given. However, the lack of returning a specific vulnerability can lead to several issues. One issue is that the user of the tool does not know how to patch the vulnerable code. Another potentially more critical issue is that it is not verifiable whether the tool correctly identified a vulnerability. There are scenarios in which the tool incorrectly targets code as vulnerable, even though it may be safe, or misidentifies the type of vulnerability [123]. Consequently, this can create a false sense of security or clarity. See Figure 1 for a simplified example prompt that can be used for binary labeling.

Finding the correct vulnerability. The above-described issues can be mitigated by the inclusion of an explanation or indicator of what kind of vulnerability we are dealing with. An example is the inclusion of the CWE ID that can be associated with the findings of the tool. This makes it possible for the user to both verify the vulnerability and shape an initial path the user can take to patch the vulnerability. Additionally, the weakness recognition of the LLM

can be verified by asking for a brief explanation of the found vulnerability. This can be further enhanced by the use of RAG, which can provide the LLM with relevant vulnerable code with potentially similar weaknesses from an external knowledge base. See Figure 2 for a simplified example of an LLM prompt to verify whether the LLM finds the correct vulnerability, and Figure 3 for a simplified example of an LLM prompt to verify whether the LLM finds the correct vulnerability with the additional presence of relevant vulnerable code snippets.

Binary Labeling

You are a vulnerability detector. Your goal is to determine whether the following code snippet is vulnerable or not.

```
[Code Snippet]
```

Please use the following output structure for your output:

```
[Vulnerable]: yes/no
```

Figure 1: Simplified example of a prompt for vulnerability detection using binary labeling.

Correct Vulnerability

You are a vulnerability detector. Your goal is to determine whether the following code snippet is vulnerable or not, and what weakness causes this vulnerability.

```
[Code Snippet]
```

Please use the following output structure for your output:

```
[Vulnerable]: yes/no
[Weakness]: CWE ID(s)
[Brief explanation]
```

Figure 2: Simplified example of a prompt for vulnerability detection using CWE classification.

Correct Vulnerability with RAG

You are a vulnerability detector. Your goal is to determine whether the following code snippet is vulnerable or not, and what weakness causes this vulnerability.

```
[Code Snippet]
```

You may use the following relevant examples in your judging process:

```
Relevant Entry 1
[Code Snippet]
[Vulnerable]: yes/no
[Weakness]: CWE-xxx

...

Relevant Entry n
[Code Snippet]
[Vulnerable]: yes/no
[Weakness]: CWE-xxx
```

Please use the following output structure for your output:

```
[Vulnerable]: yes/no
[Weakness]: CWE ID(s)
[Brief explanation]
```

Figure 3: Simplified example of a prompt for vulnerability detection using CWE classification with the usage of RAG.

3 Related Work

In this chapter, we review related work in the fields of traditional and LLM-based vulnerability detection. By traditional, we mean pre-LLM-based vulnerability detection methods.

3.1 Traditional Vulnerability Detection

Several traditional methods to perform vulnerability detection exist. We discuss static analysis techniques, fuzzing, and machine learning- and deep learning-based vulnerability detection. We also discuss what limitations they face.

3.1.1 Static Analysis

Static analysis for vulnerability detection focuses on the investigation of vulnerabilities in the source code [60]. We also refer to security-focused static analysis as Static Application Security Testing (SAST) [27]. There are various analysis techniques, which can be split into two methods: syntactic static analysis and semantic static analysis [60]. Syntactic static analysis focuses on the presence of syntactic patterns in the source code, such as dangerous C functions like `strcpy`. Semantic static analysis focuses on control- and/or data-flow information, often lifting the code into a more abstract representation such as an abstract syntax tree. This allows for the investigation of more complex vulnerabilities. These two methods have been applied in several tools. For example, Flawfinder [34] implements a syntactic analysis technique. It compares source code against a local database of vulnerable code patterns. The database primarily consists of patterns that have been susceptible to vulnerabilities in the past. However, a limitation that tools such as Flawfinder come with is high false positive rates, meaning that not every hit should be automatically treated as an actual vulnerability. Another example is Infer [46], which implements a semantic analysis approach, utilizing formal verification techniques to detect vulnerabilities primarily in pointer structures. However, a limitation that comes with semantic static analysis techniques is undecidability, which covers the problem that out of all executable code paths, we cannot know (i.e., it is undecidable) which code paths are actually executed during runtime for all possible inputs. Consequently, there are relevant program properties that cannot be determined perfectly, since it is often dependent on the input, which code paths are executed. Therefore, this program behavior needs to be approximated. The syntactic and semantic techniques are also combined, for example, in Cppcheck [25]. It uses unsound flow-sensitive analysis and path-sensitive analysis based on abstract interpretation. The main goal of Cppcheck is to detect only real errors in the code, therefore generating as few false positives as possible. However, this increases the risk of false negatives. Overall, static analysis methods face a trade-off between a high false positive rate and a high false negative rate, which are both caused by the limitation that the techniques cannot observe the execution of the program.

3.1.2 Fuzzing

A widely used vulnerability detection technique that mitigates some of the limitations that come with static analysis methods is fuzzing. It is a dynamic vulnerability detection technique that aims to detect exploitable vulnerabilities during runtime. It was first introduced by Miller et al. as a random character generator [71]. The tool, fuzz, generates a stream of random characters and uses these streams as input in the target program. Using the generated inputs, the target

program is run. During execution, the generated inputs may enter previously unentered parts of the code and potentially trigger the detection or even exploitation of a vulnerability. Since the introduction of fuzzing, it has developed into different fields. Dunbar et al. developed KLEE [13], a symbolic execution tool that automatically generates tests that achieve high coverage on the targeted code. Jiang et al. developed ContractFuzzer [48], a fuzzing tool used to detect vulnerabilities in smart contracts, with a high true positive rate. Wang et al. developed a Checksum-Aware directed fuzzing tool, TaintScope [107], which mitigates a limitation in several earlier fuzzing tools, in which vulnerable programs reject inputs early because of a checksum mechanism that checks the integrity of inputs. This is prevented in TaintScope, demonstrating a more thorough vulnerability analysis of such programs. However, fuzzing does face certain challenges, as described by Böhme et al. [10], such as automating the fuzzing process in systems that interact with their environment and efficiently generating valid inputs.

3.1.3 Machine Learning- and Deep Learning-based Vulnerability Detection

The limitations of automated vulnerability detection and high false positive rates are partially mitigated by the introduction of artificial intelligence-based vulnerability detection. Initially, this was done using machine learning. Machine learning refers to the ability to automate the process of building an analytical model using problem-specific training data to solve associated tasks [47]. In the case of vulnerability detection, a model using machine learning uses vulnerability data to learn how to automatically detect and classify vulnerabilities. Algorithms used in this process are often shallow, which means they consist of only one or a few processing stages, or layers in the case of artificial neural networks (ANNs). A study by Perl et al. [83] shows that the use of a support vector machine (SVM), a shallow machine learning algorithm, as a primary vulnerability classifier significantly reduces the false positive rate compared to static analysis methods by over 99%. Following the advancements in machine learning, a study by Wu et al. [114] shows that deep learning, which is machine learning using multi-layered deep neural networks [47], outperforms shallow machine learning methods. They show that convolutional neural networks (CNNs), long short-term memory (LSTM), and a combination of the two all outperform a traditional shallow multi-layered perceptron (MLP). CNNs are further researched by Russell et al. [92], in addition to several other deep learning methods, such as Bidirectional Long Short-Term Memory (BLSTM) by Li et al. [58], and graph neural networks (GNNs) by Zhou et al. [121]. However, these studies report the same overarching problems: the learning of irrelevant features, data duplication, and data imbalance, as is stated by Chakraborty et al. [16]. This shows that machine learning and deep learning by themselves may not be the optimal solution to automatic vulnerability detection. These limitations motivate the exploration of different approaches that mitigate these overarching problems, such as LLM-based approaches.

3.2 LLM-based Vulnerability Detection

LLMs can be applied in several methods to partially alleviate the above-mentioned limitations of traditional vulnerability detection methods. In this thesis, we focus on direct vulnerability detection using LLMs.

Earlier studies, such as the study by Purba et al. [84], demonstrate that 'raw' LLMs, with no (prompt) optimizations for vulnerability detection, perform poorly. A reason for this is the lack of proper context, such as code examples, as is shown by Yue Li et al. [56]. However, LLMs have a limited number of tokens that can be used to provide context. Lekssays et al. [53] use code property graphs to reduce the code from code snippets to reduce the number of tokens used, while not giving up the quality of the detection.

Another method to improve the detection rates is the usage of CoT. Wei et al. [109] have shown that CoT improves the performance of LLMs in various tasks by asking the LLM to generate a series of intermediate reasoning steps. Applying CoT to vulnerability detection shows promising results. Several studies, such as the study done by Bae et al. [5], report that implementing CoT in the input prompt improves the performance of vulnerability detection by giving clues to the LLMs to think step by step. Applying CoT by giving examples in the prompt, done by Chen et al. [18], and by Ceka et al. [15], among others, also shows an improvement in detection rates. By instructing the LLM to reflect on its thought process, it is given the opportunity to reflect on previous decisions and improve the detection quality. However, besides the improvements shown using these different techniques, Zhu et al. [123] show that the performance of LLMs for vulnerability detection is limited by the lack of understanding of security specifications. With the inclusion of general and domain-specific security specifications, vulnerability detection using LLMs can be improved even further. Although relevant work shows that CoT improves detection rates, the influence of internal reasoning is generally not yet well established.

Retrieval-Augmented Generation, another adaptation technique to improve vulnerability detection, can be applied in several ways, following a systematic review done on vulnerability detection using LLMs [49]. One way is to retrieve similar code snippets or code dependencies along with known vulnerability information [59, 63, 64, 95, 110, 117, 119, 122]. Lu et al. [64] show that retrieving similar code snippets results in significant improvements over other (at the time) state-of-the-art baselines. Another method in which RAG is applied to LLMs is by retrieving abstract code functionality descriptions [30, 94, 96, 101]. Du et al. [30] introduce Vul-RAG, which generates functional semantics, vulnerability causes, and fixing solutions for vulnerabilities, and uses these representations as entries that can be retrieved. Shimmi et al. [96] generate a description of the functionality of vulnerabilities using LLMs and use SBERT, an encoder-only Transformer model, to semantically identify vulnerabilities using cosine similarity. Sun et al. [101] introduce LLM4Vuln, which converts collected vulnerable code and vulnerability reports to a vulnerability vector database, from which it retrieves the most relevant entry. This differs from our approach, where we generate vulnerability reports and store them together with the respective code snippets in a VKB for text matching rather than embedding matching. Additionally, based on the vector database, Sun et al. [101] test the effectiveness of the addition of RAG and the influence of reasoning models. However, they state that LLM4Vuln is dependent on the data that the model is pre-trained on. Since this is a black box, it is unclear if the LLM effectively captures the patterns of underlying vulnerabilities, or if the LLM was trained on these vulnerabilities prior and recognized the vulnerabilities themselves rather than the underlying patterns. Their claim that LLMs possess sufficient built-in knowledge for vulnerability detection may not be reliable if LLM contamination has an influence on their experiments. In contrast, we perform experiments strictly with vulnerabilities found after the knowledge cut-off date of the LLM to prevent LLM contamination. Finally,

RAG can also be applied in combination with CWE descriptions [14, 51, 52, 66, 82]. Cao et al. [14] found that using CWE descriptions as external knowledge enables CWE classification of vulnerabilities.

These three different methods of applying RAG to LLMs all show improvements. Therefore, a combination of the three is applied in this thesis; we use a VKB that stores abstract code functionality in the form of a vulnerability report, code snippets, and CWE descriptions. Outside of the three applications of RAG stated in the systematic review [49], we also store CVE data in our VKB. Retrieving entries from the VKB is done by matching the abstract code functionality and vulnerability information of the vulnerabilities in the VKB with the code description of the input code snippet, as we focus on retrieval based on overlap in textual representations of (potential) weaknesses in the code snippets from the VKB and the input code snippet.

The foundation of this approach is similar to Vul-RAG [30], although several important distinctions exist, which are also not covered by the systematic review [49]. First, Vul-RAG constructs a large vulnerability report over 3 dimensions for its VKB entries. When a VKB entry is retrieved, this generated knowledge over the different dimensions is retrieved, together with code differences. In `VulnReason-RAG`, we focus on a smaller vulnerability report, and the retrieval of CVE and CWE data, in addition to this generated knowledge and code differences. Consequently, we can quantify the influence of the addition of CVE and CWE data in the vulnerability detection process, which is not addressed in Vul-RAG. Additionally, the re-ranking method in the retrieval process differs. Vul-RAG uses the Reciprocal Rank Fusion (RRF) method to re-rank initially retrieved entries, whereas `VulnReason-RAG` uses a listwise LLM re-ranker. Therefore, we can assess if an LLM re-ranker can improve retrieval quality by adding semantic matching. Moreover, there are several differences in the evaluation process, such as different evaluation metrics, a different experimental dataset, and the inclusion of statistical testing. This reduces the risk of LLM contamination and enables a more reliable evaluation in `VulnReason-RAG`. Finally, besides the influence of RAG, `VulnReason-RAG` focuses on the effects of internal reasoning as well, a factor that is not examined in Vul-RAG.

A comprehensive comparison between the different studies utilizing internal reasoning and RAG described above and our study can be found in Table 2. Here, the use of reasoning models, different retrieval and RAG strategies, and the updatability of the used VKB between papers are compared. Exact Term Matching refers to retrieving relevant entries based on the overlap of exact terms. LLM Re-ranking refers to the process of (listwise) re-ranking an initial ranking by an LLM. Under RAG, we mark Code Snippets, Abstract Code Functionality, and CWE Descriptions if they are retrieved from an external knowledge base. Lastly, by Updatable VKB, we refer to the possibility of updating the VKB at a later stage through a dynamic pipeline.

	Reasoning	**Retrieval**		**RAG**			**Updatability**
	Reasoning Model	Exact Term Matching	LLM Re-ranking	Code Snippets	Abstract Code Functionality	CWE Info	Updatable VKB
SAVANT [59]	✓	✗	✗	✓	✗	✗	✗
VUL-GPT [63]	✗	✓	✗	✓	✗	✗	✗
GRACE [64]	✗	✗	✗	✓	✗	✗	✗
LProtector [95]	✓	✗	✗	✓	✗	✓	✗
VulEval [109]	✗	✗	✗	✓	✗	✗	✗
DLAP [117]	✗	✗	✗	✓	✗	✗	✗
Benchmarking for SVD [119]	✗	✗	✗	✓	✗	✗	✗
SSRFSeek [122]	✓	✗	✗	✓	✗	✗	✗
Vul-RAG [30]	✓	✓	✗	✗	✓	✗	✗
AutoPatch [94]	✓	✗	✗	✓	✓	✗	✓
LLM + Additional Information [96]	✗	✗	✗	✗	✓	✗	✗
LLM4Vuln [101]	✓	✗	✗	✓	✓	✓	✗
LLM-CloudSec [14]	✗	✗	✗	✓	✗	✓	✗
LSAST [51]	✗	✗	✗	✓	✓	✓	✗
LLM Android Vuln. Benchmark [52]	✗	✗	✗	✓	✗	✓	✗
LLbezpeky [66]	✗	✗	✗	✓	✓	✓	✗
LLM Static Vulnerability Analysis [82]	✗	✗	✗	✓	✓	✓	✗
`VulnReason-RAG`	✓	✓	✓	✓	✓	✓	✓

Table 2: Comparison between different studies and the models and techniques they use. `VulnReason-RAG` is the tool proposed by us.

4 Methodology

In this chapter, we explain the methodology of VulnReason-RAG. We cover the complete pipeline, including all separate components and data sources. We describe what design choices we make and what LLMs are integrated.

4.1 System Diagram

The complete system diagram of VulnReason-RAG is shown in Figure 4. The pipeline consists of three components: the MoreFixes database (introduced by Akhoundali et al. [2]), the VKB, and the Vulnerability Detection Tool. The pipeline contains multiple LLM-based components. The used LLMs are discussed in the respective components, together with the specific prompt construction according to the prompt pattern framework explanation in Section 2.2.3. Preliminary information about the structure of the prompts is presented in Section 4.2.

The MoreFixes component collects enriched vulnerability data from the NVD and the GHSA and stores it in the MoreFixes database. Additional information, such as project locations, CWEs, and file changes, is mapped to the collected vulnerabilities. The MoreFixes database is the primary vulnerability data source for VulnReason-RAG, and is further covered in Section 4.3.

The VKB Generation constructs the VKB based on the MoreFixes database. First, vulnerabilities from the MoreFixes database are filtered by a number of constraints. The filtered entries are converted into VKB entries; suitable data is extracted from the MoreFixes entry and aggregated into the VulnReason-RAG data format. Subsequently, additional knowledge about these vulnerabilities, such as the functionality of the code and a brief vulnerability description, is generated. To prioritize high-quality vulnerabilities in the VKB, all knowledge generated is validated. Validated knowledge, together with the aggregated data, is stored in the VKB. Disapproved knowledge is discarded along with the aggregated data from the MoreFixes database; this vulnerability is fully skipped. The construction of the VKB is further explained in Section 4.4.

The Vulnerability Detection Tool component uses the VKB in its detection process. First, the tool takes as input independent entries. From every input entry, the code snippet is extracted. For every code snippet, a code description is generated, including a description of the functionality of the input code snippet and a list of potential weaknesses. This code description is used in the two-stage retrieval process. First, BM25 retrieves the top-l relevant VKB entries from the VKB based on exact term overlap between the code description and stored generated knowledge. Second, the top-l relevant VKB entries are re-ranked by an LLM re-ranker into the top-k VKB entries. Finally, based on the input code snippet, code description, and the retrieved VKB entries, the vulnerability detector performs vulnerability classification. In this classification process, internal reasoning can be disabled or enabled. The tool outputs a binary vulnerability status (i.e., yes or no), and, when relevant, the predicted CWE label(s). This is stored in an external file. The vulnerability detector is further covered in Section 4.5.

In the system diagram, references to the respective SRQs are present. These are the components that are used in the experiments to answer the SRQs.

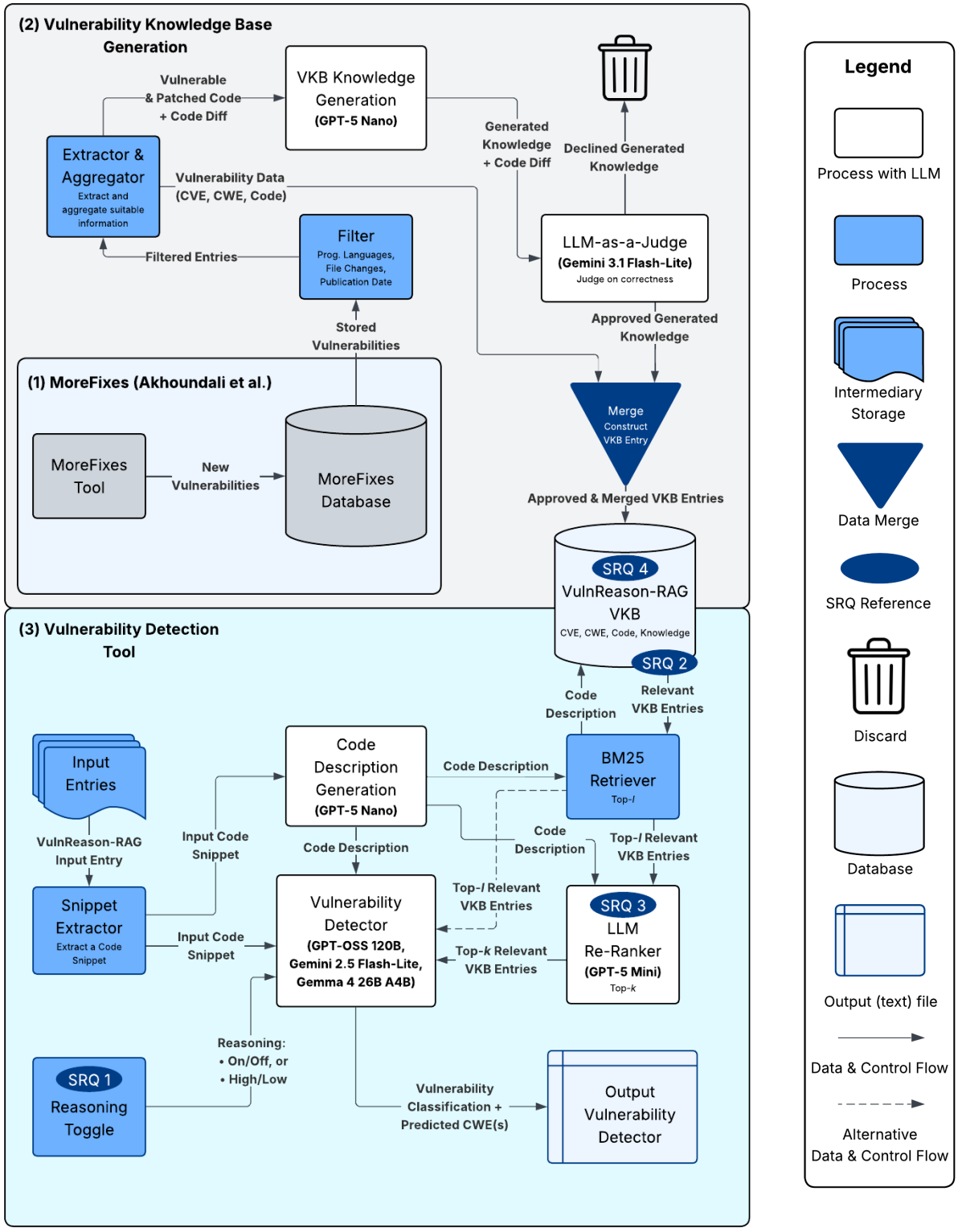


Figure 4: System Diagram of `VulnReason-RAG` with corresponding legend. The system consists of three main components: MoreFixes (introduced by Akhoundali et al. [2]), the Vulnerability Knowledge Base Generation, and the Vulnerability Detection Tool.

4.2 Prompt Construction

All prompts used for the LLM-based processes in the pipeline are constructed following the prompt pattern framework, introduced by White et al. [113]. This framework is discussed in Section 2.2.3. Out of the 16 prompt patterns present, 3 patterns are consistently used across all LLM-based components. These are the **Persona pattern**, the **Template pattern**, and the **Reflection pattern**. Table 3 presents all prompt patterns. For the prompt patterns not suitable for our components, a brief explanation is provided with a justification. The prompt patterns that are selected are further explained in the respective descriptions of the LLM-based components.

Pattern Category	Prompt Pattern	Used	Explanation
Input Semantics	Meta Language Creation	✗	We do not require a custom language.
Output Customization	Output Automater	✗	We do not require task automation.
	Persona	✓	[**Component Specific**]
	Visualization Generator	✗	Output is not used for visualizations.
	Recipe	✗	We only require the final answer with brief reasoning.
	Template	✓	[**Component Specific**]
Error Identification	Fact Check List	✗	We do not require a list of facts.
	Reflection	✓	[**Component Specific**]
Prompt Improvement	Question Refinement	✗	We do not require improvements to the question.
	Alternative Approaches	✗	To ensure consistency, we do not use alternative approaches.
	Cognitive Verifier	✗	We are not interested in subquestions, but rather in analysis.
	Refusal Breaker	✗	We perform experiments multiple times to prevent tainting of results by occasional refusals.
Interaction	Flipped Interaction	✗	We are only interested in the analysis of the LLM.
	Game Play	✗	We are only interested in the analysis of the LLM.
	Infinite Generation	✗	We have a limited budget for tokens.
Context Control	Context Manager	✗	For all LLM calls, there should be a consistent context.

Table 3: Selection of prompt patterns for all prompts in our pipeline. Prompt Pattern Framework by White et al. [113].

Certain limitations potentially affect the effectiveness of the constructed prompts. See Section 6.3 for limitations on prompt engineering and optimization.

4.3 Vulnerability Data Source

The primary data source used in this research is the MoreFixes database, introduced by Akhoundali et al. [2]. MoreFixes aggregates vulnerability data from the NVD [76] and the GHSA [40], and provides a tool to collect new data and update the database. It consists of nine tables, which are covered below. Figure 5 presents the entity relationship diagram.

- **cve**: Contains information about the stored CVE, such as publication and modification dates, CVSS, and additional scores. Each entry represents a distinct CVE.
- **cwe**: Information about the stored CWE, such as a description and URL. Each entry represents a distinct CWE.
- **cwe_classification**: Defines the mappings between CVEs and assigned CWEs. A CVE may be mapped to multiple CWEs.
- **cve_project**: Defines the mappings between CVEs and project repositories.
- **fixes**: Connects CVEs to project repositories and code changes. Each entry represents a distinct fix that patches a CVE.
- **commits**: Information about stored commits, such as authors, commit messages, and the number of lines changed. Each entry represents a commit.

- **repository**: Repositories with additional metadata for projects with known CVEs. Each entry represents a project repository.
- **file_change**: Files in which fix changes occur to patch one or multiple vulnerabilities. Each entry represents a file in which changes have occurred.
- **method_change**: Method changes that occur in the files when a vulnerability is patched. Each entry represents a method in which changes have occurred.

Due to the data being separated into various tables, the tables can be independently updated. This allows for dynamic mappings for added or modified CVEs. It also allows for a new CVE to be linked to already existing data, such as a CVE found in a repository that already exists in the MoreFixes database.

An essential feature of the database is the inclusion of a 'fixes score', which is located in the fixes table. This score represents the likelihood that a located fix is the relevant fix for the associated CVE. To only consider CVE fixes that have a high likelihood of being the relevant fix for associated CVEs and reduce noise, Akhoundali et al. have set the base threshold of this score to a minimum of 65. We adopt this threshold in our research.

4.3.1 Data Statistics

Akhoundali et al. provide database dumps of MoreFixes. The most recent dump (covering data up until 26-09-2024) [3] has been restored. It has been updated using the provided tool and contains information up until 04-04-2026. Table 4 provides a comparison between the statistics of the database on 26-09-2024 and 04-04-2026. Included entries have a fixes score of at least 65. The number of CVEs with associated fixes, projects, and commits collected after the last dump shows the increasing number of vulnerabilities found in recent years.

	26-09-2024	**04-04-2026**
CVEs	29203	36010
Projects	7238	9184
Commits	35276	43316

Table 4: Statistics MoreFixes dump (26-09-2024) and updated (04-04-2026).

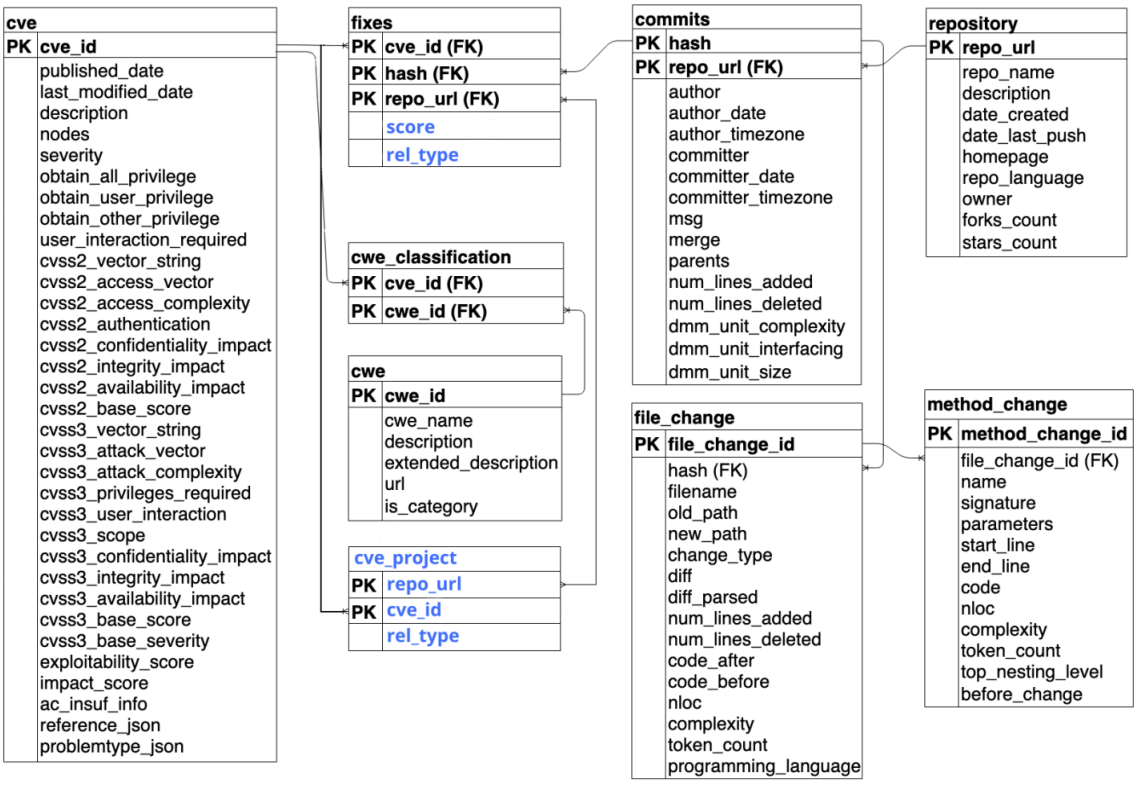


Figure 5: Entity relationship diagram of the MoreFixes database [2].

Algorithm 1: VKB Construction

Input: MoreFixes Database *MD*
Output: Vulnerability Knowledge Base *VKB*

```
1  VKB = ∅
2  filteredEntries ← filter(MD)
3  foreach MoreFixesEntry in filteredEntries do
4      vulnData ← aggregateVulnData(MoreFixesEntry)
5      codeBefore, codeAfter, diff ← extractCode(MoreFixesEntry)
6      knowledge ← generateKnowledge(codeBefore, codeAfter, diff)
7      knowledgeValid ← validateKnowledge(knowledge, diff)
8      if knowledgeValid then
9          VKBEntry ← constructEntry(vulnData, knowledge)
10         VKB.add(VKBEntry) # Add entry to the VKB
11 return VKB
```

4.4 Vulnerability Knowledge Base Pipeline

This section covers the pipeline to create the VKB using data from the MoreFixes database. The pipeline consists of three stages: a data filter, the construction of VKB entries, and the quality validation of these entries. Algorithm 1 shows a high-level overview of this pipeline.

4.4.1 Data Filtering

We filter data from the MoreFixes database that is suitable for our research. This stage corresponds to lines 2-3 in Algorithm 1. In the scope of this research, we extract the following three types of data from the database:

- CVE: CVE ID, published date
- CWE: CWE ID, CWE name, CWE description
- File changes: hash, filename, code before, code after, code difference, programming language

Vulnerabilities are filtered on three separate specifications, covering structural, domain, and temporal constraints.

Structural constraint: Vulnerabilities may only have exactly one file change as a patch. This ensures that the complete vulnerability and patch are present within one VKB entry.

Domain constraint: Vulnerabilities are filtered by programming language. We filter on the programming languages C and C++, as to this covering the largest portion of vulnerabilities that currently exist.

Temporal constraint: Vulnerabilities are sorted by publication date. Vulnerabilities that are published before the knowledge cut-off dates of the LLMs are included in the VKB. Suitable vulnerabilities published after the publication date are used as experimental data, about which more is explained in Section 5.1.2.

The information about CVE data, CWE data, and the code is used for two purposes. First, knowledge is generated using the code, which forms the core of our VKB. Second, it allows us to use this additional information in the vulnerability detection and classification prompt. Consequently, the effect of adding the separately described components in the final vulnerability detection prompt can be measured, enabling us to answer **SRQ2**. Additionally, the temporal constraint enables us to measure the effect of having access to CVEs from different publication dates, which is necessary to answer **SRQ4**.

4.4.2 Vulnerability Knowledge Base Construction

In the second stage of the pipeline, the filtered entries from the MoreFixes database are converted to entries of the VKB. This stage corresponds to lines 4-5 in Algorithm 1. It consists of two steps:

Aggregation of extracted data: CVE data, CWE data, and code differences are extracted from the MoreFixes database and prepared to be turned into a VKB entry. Dynamic mappings from CVEs to CWEs are converted to mappings from a CVE to a list of the associated CWEs.

Generating knowledge: To obtain additional textual information about the vulnerability and the patch to use for retrieval purposes in the vulnerability detection pipeline, knowledge about the vulnerability is generated. This is done in the form of a brief vulnerability report, based on the extracted vulnerable code, patched code, and code difference from the vulnerability. This includes the functionality of the code, the vulnerability introduced in the vulnerable code together with its impact, the patch, and two security rules. The first security rule aims to provide a file-specific rule that covers the security issues within, e.g., specific code libraries. If the input code snippet uses these code libraries as well, the corresponding VKB entries are more likely to be retrieved based on this knowledge, since the security rule would match with the code description generated for this input code snippet. The second security rule aims to give a more abstract and general security rule. These security rules are added to allow the retrieval of vulnerabilities with similar weaknesses, even when the exact implementation of the code differs.

LLM Implementation. The generation of the vulnerability report is done using an LLM. The specific LLM used for this task is GPT-5 Nano [41], chosen for its cost-efficient and fast text generation.

The prompt is constructed according to the prompt pattern framework, as discussed in Section 4.2. The **Persona pattern** is integrated into the prompt by giving the LLM the role of vulnerability knowledge generator. The **Reflection pattern** is integrated to encourage the LLM to reflect on the received instructions and determine if the generated output conforms to all given requirements. The **Template pattern** is integrated to describe the exact expected output. By following this template, the LLM generates every output in a consistent structure. The full prompt used for generating knowledge is present in Appendix A.1.

4.4.3 Vulnerability Knowledge Base Validation

To prioritize high-quality knowledge, we apply a validation strategy to the generated knowledge. We deploy a judge that determines if the generated knowledge is an adequate representation of the associated code snippet.

The judge receives both the generated knowledge and the code difference between the vulnerable code and the patched code; this difference captures what has changed to patch the vulnerability. It is then assessed whether the generated knowledge and code difference match, i.e., if the knowledge accurately reflects the code difference. When the judge determines the generated knowledge of a code snippet to be matching, the aggregated vulnerability data and generated knowledge are merged into one VKB entry and are added to the VKB. In the case that the judge determines they do not match, both the extracted data and the generated knowledge are discarded. The validation process corresponds to lines 6-9 of Algorithm 1. The final construction of the VKB entry happens in the validation process to prevent unnecessary VKB entries from being generated that end up discarded afterwards. Figure 6 presents the VKB table schema.

LLM Implementation. The judge used to validate entries for the VKB is an LLM. Therefore, from this point onward, we refer to the judge as an LLM-as-a-Judge. The specific LLM used for this task is Gemini 3.1 Flash Lite (Preview) [38], chosen for its cost-efficient and fast text generation, and for its independence from GPT-5 Nano (used to generate the knowledge). Consequently, the validation process is cross-model independent of the knowledge generation, preventing self-bias and family-bias in this process [100].

The prompt is constructed according to the prompt pattern framework, discussed in more detail in Section 4.2. The **Persona pattern** is integrated into the prompt by giving the LLM the role of LLM-as-a-Judge. The **Reflection pattern** is integrated to encourage the judge to reflect on its decisions and, if needed, revise the decision. The **Template pattern** is integrated by requesting the LLM-as-a-Judge to generate a JSON object as output, containing both a binary decision (i.e., 'yes' or 'no') and a brief explanation that can be used for manual analysis. See Appendix A.2 for the full prompt.

4.4.4 Vulnerability Knowledge Base Statistics

After applying the filtering constraints, a total of 2011 vulnerabilities are present in the MoreFixes database that are suitable for our VKB. After the initial pass over all suitable vulnerabilities, a total of 1845 VKB entries were successfully generated and approved by the judge. A second pass is performed on the initially discarded vulnerabilities, due to an API outage during the first pass. This second pass resulted in an additional 97 VKB entries being successfully generated and approved. Consequently, the VKB contains a total of 1942 VKB entries. This is 96.6% of the suitable vulnerabilities from the MoreFixes database, showing that the pipeline is able to generate and validate knowledge for the vast majority of vulnerabilities. The CVEs cover a total of 137 distinct CWEs, with the three most common CWEs being CWE-119 (buffer overflow, 202 occurrences), CWE-125 (out-of-bounds read, 195 occurrences), and CWE-787 (out-of-bounds write, 193 occurrences). This indicates that the greatest number of vulnerabilities in the VKB originate from memory safety issues. The top 10 assigned CWE labels are presented in Figure 7. The generated VKB is publicly available [44].



Figure 6: Schema of the VKB.

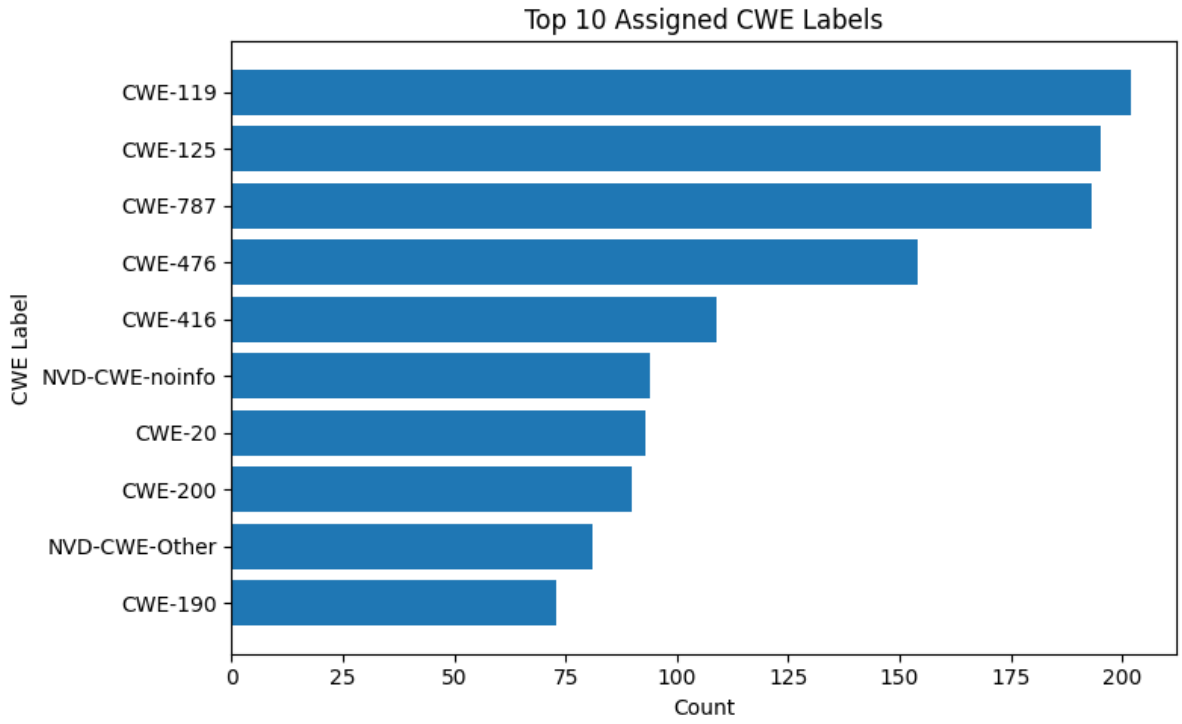


Figure 7: Top 10 Assigned CWE Labels in the VKB. The most common CWE is CWE-119 (buffer overflow).

Algorithm 2: Vulnerability Detection Pipeline

Input: Input Entries *IE*, Selected LLM *LLM*, Reasoning Mode *RM*, Vulnerability Knowledge Base *VKB*, BM25 Retrieval Size l, LLM Re-rank Size k
Output: Vulnerability Detector Output *VDOutput*

1 **foreach** *entry in IE* **do**
2 snippet ← `extractCodeSnippet(`*entry*`)`
3 codeDescription ← `generateDescription(`*snippet*`)`
4 top-l-Retrieved ← `BM25Retrieval(`*codeDescription, VKB, l*`)`
5 top-k-Reranked ← `LLMReranker(`*codeDescription, top-l-Retrieved, k*`)`
6 VC, CWE ← `detectVulns(`*snippet, LLM, RM, codeDescription, top-k-Retrieved*`)`
7 VDOutput.`write(`*VC, CWE*`)`
8 **return** VDOutput

4.5 Vulnerability Detector Pipeline

This section covers the pipeline to detect vulnerabilities in an input code snippet. The pipeline consists of three stages: a code description generator, a two-stage retrieval process, and the final vulnerability classification. Algorithm 2 shows a high-level overview of this pipeline.

4.5.1 Code Description Generation

The input of the vulnerability detection tool is a JSON object, which contains a code snippet. This code snippet is extracted, which corresponds to line 2 in Algorithm 2. Based on this, we generate a description containing a textual representation of the functionality of the code. This is complemented by a list of potential weaknesses, i.e., a list of unsafe coding practices present in the snippet that may lead to the introduction of vulnerabilities. This description serves two purposes. First, it is used in the retrieval process to retrieve relevant VKB entries (see Section 4.5.2). Second, it is used in the vulnerability classification (see Section 4.5.3). The code description generation corresponds to line 3 of Algorithm 2.

LLM Implementation. We use an LLM to generate the description, including a textual representation and a list of potential weaknesses of the input code snippet. The LLM used here is GPT-5 Nano [42], for the same reasons as in Section 4.4.2.

The prompt is constructed according to the prompt pattern framework, as discussed in Section 4.2. The **Persona pattern** is integrated into the prompt by giving the LLM the role of code analyzer. The **Reflection pattern** is integrated to encourage the LLM to reflect on the description it has generated and check if it accurately describes the input code snippet. The **Template pattern** is integrated by requesting the LLM to generate the output in a certain structure to encourage consistent outputs. See Appendix A.3 for the full prompt.

4.5.2 Vulnerability Knowledge Base Retrieval

The retrieval process consists of two parts. These two parts complement each other by both providing distinct benefits, keeping the retrieval process efficient and effective. The retrieval is done based on the previously generated code description of the input code snippet, which is compared to the generated knowledge stored in the VKB. The initial retrieval of the top-l most

relevant entries is done using exact-term matching, where relevant entries from the VKB are retrieved based on the presence of relevant terminology and code functionality. Subsequently, semantic matching is performed to retrieve the top-k most relevant entries from the top-l based on semantic overlap.

The retrieval of the top-k VKB entries forms a core part of the RAG pipeline. They are provided to the vulnerability detector and classifier as additional context. This augments the final classification to be additionally based on the retrieved VKB entries, alongside the input code snippet and code description.

Stage 1: BM25. As initial retriever, we use BM25, explained in Section 2.3.2. It is a fast exact-term matching algorithm. The initially retrieved entries from the VKB are the entries with generated knowledge that have the most textual overlap with the generated description of the input code snippet. Consequently, this process is fast, but it only captures exact-term matching. We retrieve the top-l VKB entries with BM25. Stage 1 corresponds to line 4 in Algorithm 2.

Stage 2: LLM-based Re-ranking. The initial top-l retrieved entries are re-ranked by an LLM to cover the context of the vulnerability and how it correlates to the current input code snippet. This process requires an LLM response, which increases ranking duration. Meanwhile, instead of basing relevance on term overlap, it bases relevance on contextual overlap, i.e., which generated knowledge is most semantically aligned with the input code description. This is discussed in more detail in Section 2.3.3. From the final re-ranking, the top-k vulnerabilities are retrieved and sent to the vulnerability detector. Since we have a limited context for the vulnerability detector, k represents a small number of entries from the VKB. This component is especially of value for **SRQ3**, which covers the effectiveness of the LLM re-ranker. Stage 2 corresponds to line 5 in Algorithm 2.

The LLM used in this re-ranking process is GPT-5 Mini, chosen for its capability to capture more complex patterns and relationships between text and code, while still generating at high speeds. To keep this process cost-efficient and fast, we perform listwise re-ranking in contrast to pointwise or pairwise re-ranking. The differences between these are explained in Section 2.3.3

The prompt is constructed according to the prompt pattern framework, discussed in Section 4.2. The **Persona pattern** is integrated into the prompt by giving the LLM the task of a listwise re-ranker. The **Reflection pattern** is integrated into the prompt by requesting the LLM to reflect on the generated ranking to encourage accurate retrieval. The **Template pattern** is integrated into the prompt by asking the LLM to generate a list of knowledge IDs separated by commas. This encourages consistent outputs and enables direct post-processing of the retrieved VKB entries. See Appendix A.4 for the full prompt.

4.5.3 Final Vulnerability Classification

To perform the final vulnerability classification, this component receives the complete input code snippet, the generated code description, and the top-k relevant retrieved entries from the VKB. Based on this information, it determines whether the given input code snippet is

vulnerable or not, and, if so, which CWE labels are predicted. Consequently, all extracted and generated data in the complete pipeline contribute to the final classification outcome. The inclusion of the retrieved top-k relevant VKB entries augments the generation of this classification, thus completing the RAG pipeline. The final vulnerability classification and post-processing of the results correspond to lines 6-7 of Algorithm 2.

LLM Implementation. The final vulnerability classification is performed by an LLM. The variety of LLMs that are used in this component is as discussed in Section 5.1.1. For the integrated LLM, a toggle can be set to disable or enable internal reasoning, as discussed in Section 2.2.5. This reasoning toggle is used to answer **SRQ1**.

The prompt is constructed according to the prompt pattern framework, discussed in Section 4.2. The **Persona pattern** is integrated into the prompt by giving the LLM the role of vulnerability detector. The **Template pattern** is integrated into the prompt by requesting the LLM to structure the output as a JSON object. This encourages consistency across outputs and allows for direct processing of the vulnerability classification. The **Reflection pattern** is integrated into the prompt by requesting the LLM to reflect on the generated classification and, if needed, revise the classification. See Appendix A.5 for the full prompt.

5 Experimental Evaluation

In this chapter, we cover the evaluation of `VulnReason-RAG`. We examine the experimental setups and inspect the results of each experiment. Statistical testing under McNemar's test is performed.

5.1 Experimental Setup

In this section, we examine the choices we make in our experimental setup. We justify our choices behind the models we select, the experiment data, the parameter configuration, the budget constraints, and the hardware that apply to our experiments.

5.1.1 Model Selection

For the final vulnerability detection and classification in our experiments, we select the models based on a series of requirements, which stem from our operational and other constraints. They ensure high-quality results that do not suffer from limitations such as LLM contamination. The following constraints are applied:

- **Knowledge cut-off date**: To prevent LLM contamination, we choose models that have a knowledge cut-off date prior to February 2025. Vulnerabilities published from February 2025 onward are used as experiment data;
- **Context window limitations**: To allow for large code snippets, we select models with a context window of at least 128000 tokens;
- **Pricing**: Due to budget limitations, we select models with a maximum cost of $0.10 per 1M input tokens, and $0.40 per 1M output tokens;
- **Reasoning toggle**: To analyze the effects of internal reasoning, we select models that have an internal reasoning toggle (i.e., internal reasoning can be turned off/on or low/high, while using the same model architecture). The specific parameters used in `VulnReason-RAG` to disable and enable internal reasoning are discussed in Section 5.1.3;
- **Closed source and open source**: To analyze the differences between open source and closed source models, we select at least one model that is closed source and one model that is open source;
- **Generalization across vendors**: To ensure that our results remain consistent across models of different vendors, we select models from at least two separate vendors.

Due to the above constraints, we are limited to choosing from a set of cost-efficient LLMs. This is an influential factor in the results of all experiments. Consequently, the quality of the results is not based on the absolute values of the evaluation metrics, but rather on the relative difference in the results between the methodological configurations evaluated in our experiments. The selected models for the final vulnerability classification are GPT-OSS 120B by OpenAI [78], Gemini 2.5 Flash-Lite by Google [37], and Gemma 4 26B A4B by Google [39].

5.1.2 Experimental Dataset

To obtain ground truth experimental data, we apply the same constraints as applied on the VKB (see Section 4.4.1). To capture the full vulnerability within the context window of the LLM, we filter for vulnerabilities with only one file change. To limit the scope of considered weaknesses, we solely select vulnerabilities in the programming languages C and C++. These two programming languages have the largest number of vulnerabilities. Other programming languages may be affected differently by `VulnReason-RAG`. Lastly, to prevent LLM contamination, we filter on vulnerabilities published after the latest knowledge cut-off date, from February 2025 onward. This ensures that the LLM does not recognize specific vulnerable code snippets or CWE label classifications.

Applying these filters results in 128 CVEs. To create a balanced dataset, we randomly select 64 of these CVEs and extract the patched code. In the scope of this thesis, we assume the vulnerability in the patched code to be free of the initially located vulnerabilities, and to be free of other vulnerabilities. However, in reality, this may not always be the case. More information about this threat to validity can be found in Section 6.3. We extract the vulnerable code from the remaining 64 code snippets. Consequently, our experimental data consists of 64 safe code snippets and 64 vulnerable code snippets.

5.1.3 Parameter Configuration

Several hyperparameters, which we also refer to as configurations, are tested in our experiments. However, to ensure consistency across all our results, the hyperparameters not tested in the experiments remain consistent. When not specifically given a value, the following values apply to these `VulnReason-RAG` hyperparameters:

- Number of retrieved VKB entries using BM25 (l): 50;
- Number of re-ranked VKB entries using LLM re-ranker (k): 5;
- VKB Access: access to all VKB entries;
- VKB Retrieved: Knowledge, CVE data, CWE data, Code differences;
- Prompts: identical to how they are constructed in Chapter 4.

The number of re-ranked VKB entries using an LLM re-ranker (k) is especially important due to its direct effect on the RAG process. Multiple studies [103, 106] show that for few-shot prompting, a small number of examples in the prompt (e.g., 5-20 examples) can lead to improvements in LLM behavior. In our research, the VKB entries serve as examples. However, increasing the number of examples in a few-shot prompt can lead to diminishing returns: there may only be small improvements in overall performance, while increasing token consumption. This leads to higher costs and a larger context. Therefore, to provide relevant VKB entries while limiting excessive prompt size, we select $k = 5$. We keep this value for k consistent throughout our experiments. In addition, we retrieve a larger number of entries using BM25 (l), namely $l = 50$, to provide the LLM re-ranker with a sufficient variety of VKB entries to re-rank and select a high-quality set of VKB entries to include in the prompt.

Additionally, for all LLMs in the pipeline, we use the default parameter values provided via the OpenRouter API [80]. The parameter used to disable and enable reasoning is the `reasoning` parameter. For this parameter, two model-dependent subparameters are used. The `enabled` parameter can be set to False or True for Gemini 2.5 Flash-Lite and Gemma 4 26B A4B, and the `effort` parameter can be set to Low, Medium, or High for GPT-OSS 120B. All experiments run for this research are repeated three times to ensure consistency of results. Results shown are averaged over these three repetitions.

5.1.4 Evaluation Metrics

To evaluate the performance of `VulnReason-RAG`, we use a variety of evaluation metrics. The core evaluation metrics are the True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). Based on these four metrics, we compute the Precision (P) score, Recall (R) score, and F1 score. We apply these scores to evaluate the results from three different evaluation perspectives: binary classification, per-label CWE classification, and per-sample CWE classification. The TPs, FPs, TNs, FNs, and the formulae for precision, recall, and F1 are defined per evaluation perspective.

$$Precision = \frac{TP}{TP + FP} \tag{5}$$

$$Recall = \frac{TP}{TP + FN} \tag{6}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{7}$$

Binary Classification. We measure whether the vulnerability detector classifies code snippets correctly as vulnerable or not. We observe which code snippets the vulnerability detector classifies as vulnerable and which as not vulnerable, and compare this to the ground truth labels. The following holds for our evaluation metrics:

- **TP**: a code snippet is correctly predicted to be vulnerable;
- **FP**: a code snippet is incorrectly predicted to be vulnerable;
- **TN**: a code snippet is correctly predicted to be safe;
- **FN**: a code snippet is incorrectly predicted to be safe.

We use Equation 5 to calculate the precision, Equation 6 to calculate the recall, and Equation 7 to calculate the F1 scores.

Per-Label CWE Classification We identify, for each predicted CWE label per vulnerable code snippet, whether it matches a CWE label that is assigned. We inspect exact matching (does the predicted CWE label exactly match one of the assigned CWE labels).

The following holds for our evaluation metrics:

- **TP**: a CWE label is correctly predicted;

- **FP**: a CWE label is incorrectly predicted;
- **TN**: a CWE label is correctly not predicted;
- **FN**: a CWE label is incorrectly not predicted.

We use Equation 5 to calculate the precision, Equation 6 to calculate the recall, and Equation 7 to calculate the F1 scores. We do not cover the TNs in our final results. Per CWE label, the number of TNs is equal to all CWE labels that are not assigned and not predicted, which grows large and would otherwise skew our evaluation metrics.

Since CWE labels represent classes of vulnerabilities, some of which are overlapping or closely related, the vulnerability detector may have difficulties predicting the exact CWE labels. Therefore, we compare the correctness of the exact predicted CWE labels with a relaxed matching framework. This relaxed framework allows for a predicted CWE label to match a set of relevant CWEs. This determines if the vulnerability detector sees a correct correlation between the vulnerability and the predicted CWE label, covering the broader performance of the vulnerability detector. We use three different relaxation methods.

- **Family**: if the assigned CWE label is a direct child or parent of the predicted CWE label, the prediction is counted as correct (as determined in the CWE hierarchical tree). This shows whether the vulnerability detector predicts a CWE label that may be more granular or broader than the assigned CWE label;
- **Causal**: if the assigned CWE label is a direct predecessor or successor of the predicted CWE label, the prediction is counted as correct (as determined in the CWE hierarchical tree). This shows whether the vulnerability detector predicts a CWE label that may be a cause or result of the assigned CWE label;
- **Categorical**: if the assigned CWE label is equal to one of the CWE labels within the same category as the predicted CWE label, the prediction is counted as correct (according to the official CWE categories). This shows whether the vulnerability detector predicts a CWE label closely related to the assigned CWE label.

However, this leads to a new challenge: a predicted CWE label can now match multiple assigned CWE labels. This can lead to inflated TPs. Moreover, multiple assigned CWE labels can now match one predicted CWE label, also contributing to inflated TPs. To prevent this from influencing our results, we apply a 1-1 mapping strategy, i.e., every predicted CWE label may at most satisfy one assigned CWE label (and vice versa). To obtain the optimal mapping of predicted CWE labels to assigned CWE labels, we apply the bipartite graph maximum matching problem. From the predicted CWE labels and assigned CWE labels, we construct a bipartite graph. We apply the maximum matching method as implemented in NetworkX [67] to determine an optimal mapping that satisfies the most CWE labels. Consequently, the determined results are not influenced by inflated values of TPs.

Per-Sample CWE Classification. We identify per code snippet whether one or more of the predicted CWE labels correspond to the assigned CWE labels. This is a broader metric than the per-label CWE Classification that indicates whether the model's predictions are in the right region, while ignoring irrelevant labels additionally predicted. We inspect exact

matching (predicted CWE labels equal the assigned CWE labels) and partial matching (at least one predicted CWE label corresponds to at least one assigned CWE label). Per-sample CWE classification combines the two above classification tasks: (1) whether a code snippet is correctly seen as vulnerable, and (2) whether the CWE labels are correctly predicted. To separate the effects from both classification tasks and focus on the correctness of the CWE classification, we split up FPs into two separate buckets, as is done by Sun et al. [101]. We adopt their evaluation method. The following holds for our evaluation metrics:

- **TP**: the code snippet is correctly vulnerable, and the predicted (subset of) CWE labels are correctly predicted for this code snippet;
- **FP**: the code snippet is incorrectly predicted as vulnerable;
- **FPt**: the code snippet is correctly predicted as vulnerable, but there is no (partial) CWE label overlap (type);
- **TN**: the code snippet is correctly predicted as safe;
- **FN**: the code snippet is incorrectly predicted as safe.

We also adopt and redefine the precision, recall, and F1 scores for per-sample CWE classification.

$$Precision = \frac{TP}{TP + FP + FP_t} \tag{8}$$

$$Recall = \frac{TP}{TP + FN + FP_t} \tag{9}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{10}$$

We use Equation 8 to calculate the precision, Equation 9 to calculate the recall, and Equation 10 to calculate the F1 scores.

We use an adapted version of the three relaxation methods as discussed in the per-label CWE classification evaluation perspective. Instead of evaluating every CWE label individually, we investigate whether there is at least one predicted CWE label per sample that matches at least one assigned CWE label according to the three relaxation methods.

For both per-sample and per-label CWE classification, we consider a predicted CWE label semantically aligned with the assigned CWE label when the predicted and assigned CWE labels share a parent/child relationship, causal relationship, or belong to the same category. Although the exact CWE label is not predicted, a CWE label that is hierarchically or semantically related to it is. Therefore, we additionally evaluate the CWE classification when all three relaxation methods are enabled: a predicted CWE label is counted as correct if it matches a CWE label allowed by at least one of the relaxation methods. We refer to this as the combined relaxation method.

For every experiment, we perform the evaluation from the broadest to the finest perspective. We start with binary classification, after which we observe the per-sample CWE classification. Finally, we observe the finest-grained per-label CWE classification.

5.1.5 Statistical Significance Tests

To determine the reliability of our results, we apply McNemar's test to the results of the binary classification and per-sample CWE classification evaluation perspectives. McNemar's test is used to test whether the difference in disagreement between two configurations is significant or due to random chance on paired, binary data [68]. The test has one degree of freedom. It is performed on paired predictions from two configurations for each input entry. The test uses a 2x2 contingency table, as shown in Table 5. In this table, configuration A and configuration B refer to two separate configurations that are tested on the same data. For each data point, the outputs of configuration A and configuration B are compared to the given ground truth. One point is added in the table (i.e., to a, b, c or d) that corresponds to either configuration being correct or incorrect in correspondence with the ground truth. This test is suitable, since we use incremental ablation studies. By adding a component, we can determine if this addition results in a statistically significant difference in performance between the two configurations.

		Config B	
		Correct	Incorrect
Config A	Correct	a	b
	Incorrect	c	d

Table 5: 2x2 Contingency Table used for McNemar's test to determine statistical significance of results.

$$\chi^2 = \frac{(|b-c|-1)^2}{b+c} \tag{11}$$

$$F(x; p, n) = \sum_{i=0}^{x} \binom{n}{i} (p)^i (1-p)^{(n-i)} \tag{12}$$

We focus on the difference in disagreements between the two configurations (i.e., b: configuration A is correct, but configuration B is incorrect, and c: configuration A is incorrect, but configuration B is correct; both configurations have a different output). We determine whether this difference is statistically significant or likely due to random chance. This is done using two separate methods:

1. If $b+c \geq 25$, the Chi-square approximation is used. In our tests, Edwards' continuity correction [31] is applied for a more accurate approximation. The Chi-square approximation with continuity correction is given in Equation 11;

2. If $b+c < 25$, the Chi-square approximation is deemed to be less reliable due to too few samples [102], risking an increase in Type I errors [99]. Thus, the cumulative binomial distribution function is used, given in Equation 12. It is used when there are two mutually exclusive outcomes (i.e., success ('correct') or failure ('incorrect')). With this exact version of McNemar's test, we obtain the probability of observing x successes in n samples, with the probability of success denoted by p [1]. In this equation, $p = 0.5$, since the chance of having a success or failure is assumed to be equal for both configurations as part of the null hypothesis of McNemar's test. Additionally, the value of the total number of samples is $n = b + c$. We use $x = min(b, c)$. The resulting value is the one-sided P-value, and is multiplied by 2 to get the two-sided P-value [33].

The cumulative binomial distribution function is not used when more than 25 samples are available, since the Chi-square approximation is highly accurate in this case [102]. This makes the exact version of the test unnecessary.

The null hypothesis (H_0) of McNemar's test is that the disagreement between both configurations is balanced, suggesting that $b = c$ [68]. Subsequently, the chance that either configuration is better than the other is 0.5 (i.e., the same for either configuration), meaning $p = 0.5$ is used in the exact cumulative binomial distribution. The null hypothesis can be rejected if the resulting P-value of the McNemar's test is lower than the set threshold value α; there is a difference between the two configurations, and it is unlikely to be due to random chance. We set this threshold as $\alpha = 0.05$. In the case that H_0 is rejected, the results support the alternative hypothesis (H_A). This states that there is a difference between the two configurations that does not rely on random chance. Nevertheless, the hypothesis is not proved with certainty, since rejecting H_0 only indicates that it is unlikely due to random chance.

As discussed in Section 5.1.2, we have two classes in our experimental data: vulnerable code snippets and safe code snippets. To isolate the effects of the configuration differences on both classes, we analyze them separately. This means that for each class, we perform a separate McNemar's test. We apply the Holm-Bonferroni correction [45, 111] to adjust the significance threshold and reduce the chance of statistical errors. We order the P-value per class from low to high, and use adjusted significance thresholds for each P-value. This adjusted threshold is $\frac{\alpha}{\#\text{ of classes left}}$. If the P-value is lower than the adjusted threshold, this class is determined to be statistically significant, and we continue to the next lowest P-value and repeat this process. If the P-value is higher, we do not reject the current test's H_0, and stop rejecting all subsequent hypotheses. Since we have two classes, the lowest P-value must be below $\frac{\alpha}{2} = 0.025$, and the highest P-value below $\frac{\alpha}{1} = 0.05$.

We perform the statistical tests on the binary classification and per-sample CWE classification perspectives (matches within the combined relaxed framework). In addition, if there is disagreement between the two configurations that we compare, we indicate using arrows whether configuration A performs better, or configuration B performs better. A left arrow ($\leftarrow$) refers to configuration A, and a right arrow ($\rightarrow$) to configuration B. We do not perform the test for per-label CWE classification, since this is a multi-label classification problem and thus does not work with paired, binary data.

All results of the performed statistical tests, including contingency tables and P-values, are publicly available [44].

5.1.6 Hardware & Runtime Environment

All experiments are run on a private virtual machine. This virtual machine has the following configurations:

- **CPU**: AMD EPYC 7413 24-Core Processor
- **RAM**: 64GB
- **OS**: Ubuntu 24.04.4 LTS (GNU/Linux 6.8.0-110-generic x86_64)

- **Storage**: 500GB SSD

As an LLM provider, we use the OpenRouter API [79]. OpenRouter is a provider for a variety of LLMs from several vendors, including large vendors such as OpenAI and Google. The platform supports several parameters in the API, enabling, e.g., the possibility to disable or enable internal reasoning for a suitable LLM.

We use the tokenizers from the respective models to ensure the input and output tokens do not exceed the context windows.

`VulnReason-RAG` and all experiments are developed and run in Python, and run using Python 3.12.3. The primary choice for Python is because of its broad coverage of APIs such as the OpenRouter API and the APIs for the tokenizers. All used Python packages are available [44].

5.2 Experiment 1: Reasoning Influence

In this experiment, we examine the effect of internal reasoning on vulnerability detection. First, we cover the setup we are using for this experiment. Afterwards, we analyze and interpret the results, and test for statistical significance. This experiment aims to uncover an answer to **SRQ1**.

5.2.1 Setup

The majority of the setup is as discussed in Section 5.1. However, one of the hyperparameters in `VulnReason-RAG`, discussed in Section 5.1.3, is of importance in this experiment: the reasoning toggle. For the selected models GPT-OSS 120B, Gemini 2.5 Flash-Lite, and Gemma 4 26B A4B, we run our pipeline with internal reasoning disabled and enabled. Using the results, we examine the influence of internal reasoning to answer **SRQ1**.

The full system is used in this experiment, including the complete RAG pipeline with BM25 and LLM re-ranking. There is a total of 6 configurations: 3 models, all run with 2 configurations: with internal reasoning disabled and enabled. To isolate the effect of internal reasoning, the other components of the pipeline are consistent for both configurations with internal reasoning disabled and enabled, per model. This includes all non-deterministic LLM outputs in previous stages of the pipeline. All configurations are repeated three times for consistency purposes. All results are averaged over these three repetitions.

5.2.2 Results

In this section, we discuss the results of the first experiment. We analyze the results averaged over three repetitions. Across all models, per iteration for each configuration, the total number of code snippets resulting in failures ranged between 4 and 18. These failures are excluded from our results. A detailed analysis of the failures is provided in Appendix B.1. We interpret the results per evaluation perspective.

Binary classification. First, the results presented in Table 6 show the performance of GPT-OSS 120B, Gemini 2.5 Flash-Lite, and Gemma 4 26B A4B using the full configured pipeline with internal reasoning disabled and enabled on binary classification. We see that for all models with reasoning disabled, precision lies between 0.540 and 0.582; a relatively small difference. A larger difference is visible for the recall, where GPT-OSS 120B has a recall of

Binary Classification	Reasoning Disabled			Reasoning Enabled		
	P	R	F1	P	R	F1
GPT-OSS 120B	**0.582**	0.426	0.489	**0.536**	0.642	0.584
Gemini 2.5 Flash-Lite	0.557	0.473	0.511	0.519	**0.896**	**0.657**
Gemma 4 26B A4B	0.540	**0.700**	**0.610**	0.498	0.687	0.577

Table 6: Results of binary classification for models GPT-OSS 120B, Gemini 2.5 Flash-Lite, and Gemma 4 26B A4B with reasoning disabled compared to reasoning enabled. Effects of internal reasoning indicate a small decrease in precision and substantial increases in recall for GPT-OSS 120B and Gemini 2.5 Flash-Lite. For Gemma 4 26B A4B, internal reasoning leads to a slight decrease in recall. Results are averaged over three repetitions.

Per-Sample CWE Classification	Reasoning Disabled			Reasoning Enabled		
	P	R	F1	P	R	F1
GPT-OSS 120B						
Exact	0.072	0.053	0.060	0.065	0.079	0.071
Partial	0.100	0.074	0.084	0.119	0.142	0.129
Relaxed - Family	0.229	0.168	0.193	0.259	0.311	0.282
Relaxed - Causal	0.206	0.152	0.174	0.256	0.305	0.278
Relaxed - Categorical	0.194	0.142	0.163	0.219	0.263	0.239
Relaxed - Combined	**0.384**	**0.284**	**0.324**	**0.401**	**0.479**	**0.436**
Gemini 2.5 Flash-Lite						
Exact	0.018	0.016	0.017	0.038	0.065	0.048
Partial	0.093	0.079	0.085	0.085	0.146	0.107
Relaxed - Family	0.235	0.199	0.216	0.204	0.353	0.259
Relaxed - Causal	0.205	0.174	0.188	0.205	0.353	0.259
Relaxed - Categorical	0.204	0.173	0.187	0.188	0.325	0.239
Relaxed - Combined	**0.452**	**0.384**	**0.415**	**0.393**	**0.679**	**0.498**
Gemma 4 26B A4B						
Exact	0.016	0.021	0.018	0.056	0.076	0.065
Partial	0.094	0.121	0.106	0.146	0.200	0.169
Relaxed - Family	0.243	0.316	0.275	0.241	0.330	0.279
Relaxed - Causal	0.253	0.327	0.285	0.258	0.356	0.299
Relaxed - Categorical	0.192	0.248	0.216	0.228	0.313	0.264
Relaxed - Combined	**0.411**	**0.532**	**0.463**	**0.391**	**0.539**	**0.453**

Table 7: Results per-sample CWE classification for models GPT-OSS 120B, Gemini 2.5 Flash-Lite, and Gemma 4 26B A4B with reasoning disabled compared to reasoning enabled. Effects of reasoning indicate a slightly fluctuating precision but an increase in recall across exact, partial, and relaxed matching. This results in higher F1 scores with reasoning turned on for GPT-OSS 120B and Gemini 2.5 Flash-Lite. The F1 scores of Gemma 4 26B A4B slightly decrease. Results are averaged over three repetitions.

0.426 and Gemma 4 26B A4B has a recall of 0.700. This means that even without internal reasoning, Gemma 4 26B A4B can correctly classify $\approx 70\%$ of vulnerable code snippets. When reasoning is enabled, precision decreases slightly; for all models, it lies within 0.498 and 0.536. However, a substantial improvement can be seen for the recall with reasoning turned on, with Gemini 2.5 Flash-Lite reaching a recall of 0.896 (a relative improvement of $+89.43\%$ in comparison to having reasoning disabled). A similar improvement is observed for GPT-OSS 120B as well, leading to a higher F1 score. However, we observe that the recall decreases slightly for Gemma 4 26B A4B when reasoning is enabled, demonstrating that internal reasoning may affect models differently.

Per-sample CWE classification. In Table 7, the per-sample CWE classification results are presented. These results are visualized in Figure 8 (absolute scale) and Figure 9 (relative scale). With reasoning disabled, all three models have difficulty predicting the exact set of CWE labels that are assigned as ground truth. This indicates that predicting the exact set of assigned CWE labels is too fine-grained for the cost-efficient LLMs used in this experiment.

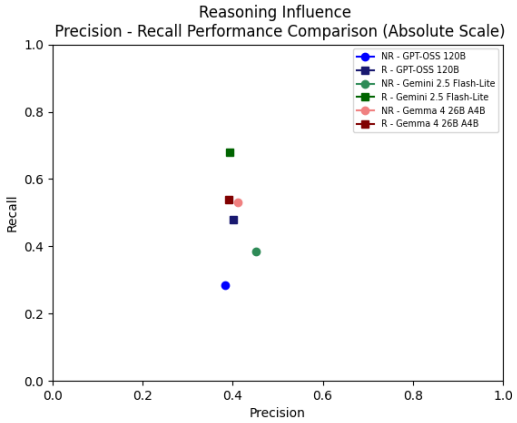


Figure 8: Per-sample CWE classification absolute precision-recall performance comparison. Blue markers represent GPT-OSS 120B, green markers Gemini 2.5 Flash-Lite, red markers Gemma 4 26B A4B. Round markers represent internal reasoning disabled, square markers internal reasoning enabled. Represents the Relaxed - Combined precision and recall scores.

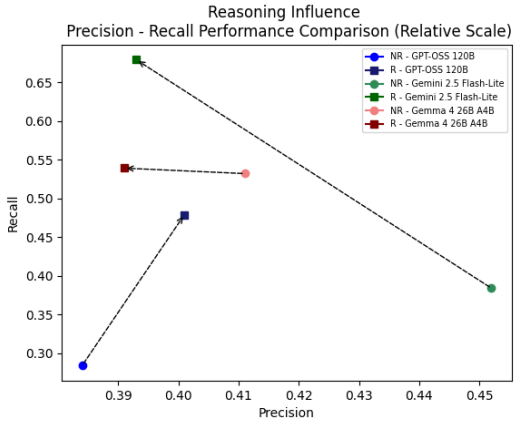


Figure 9: Per-sample CWE classification relative precision-recall performance comparison. Blue markers represent GPT-OSS 120B, green markers Gemini 2.5 Flash-Lite, red markers Gemma 4 26B A4B. Round markers represent internal reasoning disabled, square markers internal reasoning enabled. Represents the Relaxed - Combined precision and recall scores.

However, in more cases, part of the predicted CWE labels match with part of the assigned CWE labels, showing that at times, part of the predicted CWE labels are correct. Viewing the results under our relaxed framework, we observe that both precision and recall rise when allowing semantically aligned CWE labels to be counted as correct; especially allowing direct child and parent CWE labels increases the F1 score. Comparing this to the model configuration with reasoning enabled, we see that the precision fluctuates slightly, and the recall rises substantially for GPT-OSS 120B and Gemini 2.5 Flash-Lite. The recall score of Gemma 4 26B A4B increases slightly, indicating that even though the binary classification performance decreases with reasoning enabled, the semantic reasoning behind the correct CWE classification increases. Comparing the combined relaxed framework methods between the models, with reasoning enabled, we see that Gemini 2.5 Flash-Lite scores the best overall, with a precision score of 0.393 and a recall score of 0.679. This means that 67.9% of all vulnerable samples have a predicted CWE label that is semantically similar to a ground truth CWE label.

Per-label CWE classification. Table 8 presents the results of per-label CWE classification. We observe that with both reasoning disabled and enabled, the models predict a substantial number of semantically irrelevant CWE labels. Small increases across almost all metrics can be observed, with only a small portion of exact matches between predicted CWE labels and assigned ground truth CWE labels. When allowing semantically similar CWE labels, we observe small increases in recall, but not as substantial as with the other two evaluation perspectives. However, internal reasoning does show a relative improvement over reasoning disabled. All F1 scores across all models increase when reasoning is enabled. Gemma 4 26B A4B shows the best overall performance. Of all CWE labels it predicts, 15.4% of labels are semantically similar to the assigned ground truth CWE labels. 26.1% of assigned ground truth CWE labels are covered by semantically similar CWE label predictions.

Per-Label CWE Classification	Reasoning Disabled			Reasoning Enabled		
	P	R	F1	P	R	F1
GPT-OSS 120B						
Exact	0.072	0.055	0.064	0.076	0.107	0.089
Relaxed - Family	0.107	0.079	0.090	0.104	0.154	0.128
Relaxed - Causal	0.106	0.079	0.089	0.110	0.154	0.128
Relaxed - Categorical	0.119	0.083	0.097	0.107	0.150	0.125
Relaxed - Combined	**0.173**	**0.126**	**0.144**	**0.163**	**0.229**	**0.191**
Gemini 2.5 Flash-Lite						
Exact	0.046	0.059	0.051	0.054	0.110	0.072
Relaxed - Family	0.088	0.114	0.099	0.080	0.163	0.107
Relaxed - Causal	0.070	0.091	0.079	0.080	0.164	0.108
Relaxed - Categorical	0.067	0.087	0.076	0.082	0.168	0.110
Relaxed - Combined	**0.130**	**0.170**	**0.147**	**0.132**	**0.270**	**0.177**
Gemma 4 26B A4B						
Exact	0.043	0.092	0.059	0.090	0.153	0.113
Relaxed - Family	0.073	0.155	0.099	0.117	0.197	0.147
Relaxed - Causal	0.075	0.159	0.102	0.109	0.184	0.137
Relaxed - Categorical	0.060	0.127	0.082	0.114	0.195	0.144
Relaxed - Combined	**0.112**	**0.238**	**0.153**	**0.154**	**0.261**	**0.194**

Table 8: Results per-label CWE classification for models GPT-OSS 120B, Gemini 2.5 Flash-Lite, and Gemma 4 26B A4B with reasoning disabled compared to reasoning enabled. Effects of reasoning result in slight fluctuations in precision and moderate increases in recall. This results in higher F1 scores with reasoning enabled for all models. Results are averaged over three repetitions.

Overall, our results indicate that enabling internal reasoning does not influence precision strongly in one direction: fluctuations in both directions are observed. However, both on binary classification and CWE classification, substantial improvements in recall are seen for GPT-OSS 120B and Gemini 2.5 Flash-Lite. This indicates that internal reasoning results in more correct vulnerability predictions without causing substantially more false positives, demonstrating that reasoning improves vulnerability detection under fitting semantic reasoning for some models. Additionally, although Gemma 4 26B A4B with reasoning enabled leads to poorer binary classification performance, increases in per-sample CWE classification and per-label CWE classification indicate that the semantic reasoning of predictions more often aligns with the assigned CWE labels.

GPT-OSS 120B shows clear improvements with reasoning enabled across all evaluation metrics. The model is not negatively influenced by reasoning across any evaluation perspective, which is the case with Gemma 4 26B A4B. However, since GPT-OSS 120B still reaches almost the same per-label CWE classification performance, we determine that this model shows the most stable improvements out of the tested models. These stable improvements are an important foundation to test other parts of our pipeline, such as the RAG process. Therefore, we use GPT-OSS 120B as the main model for the remainder of our experiments.

5.2.3 Statistical Significance

To determine the reliability of our results, we perform McNemar's test on the results of the binary classification and per-sample CWE classification. The test is explained in Section 5.1.5.

We observe in Table 9 that the differences between reasoning disabled and enabled for all iterations for both the vulnerable and safe classes for both GPT-OSS 120B and Gemini 2.5 Flash-Lite are significant. The binary classification results of the vulnerable class are positively influenced when reasoning is enabled. The opposite is observed for the safe class, which is negatively influenced when reasoning is enabled. This presents a statistically significant trade-off: enabling reasoning increases the number of correct vulnerable predictions, while it decreases the number of correct safe predictions. This supports the increase in recall for both models in the binary classification evaluation perspective: more of the vulnerable code snippets are correctly predicted when reasoning is enabled. However, we cannot exclude that all changes in Gemma 4 26B A4B could be a result of random chance, as none of the calculated P-values are statistically significant. This also fits the binary classification results for the model, which presents only a slight negative fluctuation of the recall when reasoning is enabled.

	GPT-OSS 120B		**Gemini 2.5 Flash-Lite**		**Gemma 4 26B A4B**	
Class	Vulnerable	Safe	Vulnerable	Safe	Vulnerable	Safe
Reasoning Disabled ↔ Reasoning Enabled						
Iteration 1	$\rightarrow \mathbf{5.19 \times 10^{-4}}$	$\leftarrow \mathbf{1.15 \times 10^{-3}}$	$\rightarrow \mathbf{3.30 \times 10^{-4}}$	$\leftarrow \mathbf{6.01 \times 10^{-6}}$	$\leftarrow 2.23 \times 10^{-1}$	$\leftarrow 4.53 \times 10^{-1}$
Iteration 2	$\rightarrow \mathbf{1.69 \times 10^{-2}}$	$\leftarrow \mathbf{2.66 \times 10^{-2}}$	$\rightarrow \mathbf{1.08 \times 10^{-5}}$	$\leftarrow \mathbf{6.58 \times 10^{-7}}$	$\rightarrow 3.91 \times 10^{-2}$	$\rightarrow 3.86 \times 10^{-2}$
Iteration 3	$\rightarrow \mathbf{2.10 \times 10^{-5}}$	$\leftarrow \mathbf{1.31 \times 10^{-3}}$	$\rightarrow \mathbf{3.35 \times 10^{-7}}$	$\leftarrow \mathbf{1.81 \times 10^{-5}}$	$\rightarrow 2.23 \times 10^{-1}$	$\leftarrow 6.54 \times 10^{-2}$

Table 9: P-values for the statistical tests performed on binary classification to determine statistically significant differences between reasoning disabled and reasoning enabled in experiment 1. The significance threshold is set to $\alpha = 0.05$. The P-values are given per class, per iteration, and statistical significance is determined after performing Holm-Bonferroni correction. Bold values are statistically significant. Results show that the difference for both classes for all iterations of GPT-OSS 120B and Gemini 2.5 Flash-Lite between reasoning disabled and enabled is statistically significant. The left arrow ($\leftarrow$) indicates that the first configuration performs better, while the right arrow ($\rightarrow$) indicates that the second configuration performs better.

	GPT-OSS 120B	**Gemini 2.5 Flash-Lite**	**Gemma 4 26B A4B**
Class	Vulnerable	Vulnerable	Vulnerable
Reasoning Disabled ↔ Reasoning Enabled			
Iteration 1	$\rightarrow 8.15 \times 10^{-1}$	$\rightarrow 1.0$	$\leftarrow 4.42 \times 10^{-1}$
Iteration 2	$\rightarrow 1.0$	$\rightarrow 5.74 \times 10^{-2}$	$\rightarrow 7.27 \times 10^{-1}$
Iteration 3	$\rightarrow 6.25 \times 10^{-1}$	$\rightarrow \mathbf{3.42 \times 10^{-3}}$	$\rightarrow 2.89 \times 10^{-1}$

Table 10: P-values for the statistical tests performed on per-sample CWE classification to determine statistically significant differences between reasoning disabled and reasoning enabled in experiment 1. The significance threshold is set to $\alpha = 0.05$. The P-values are given per class and per iteration. Bold values are statistically significant. Results show that no models are experiencing consistent statistically significant results. The left arrow ($\leftarrow$) indicates that the first configuration performs better, while the right arrow ($\rightarrow$) indicates that the second configuration performs better.

To explore the effects of reasoning on the semantic alignment of vulnerable decisions, we perform and analyse the statistical test on the per-sample CWE classification perspective. The results are presented in Table 10, and show that for GPT-OSS 120B and Gemma 4 26B A4B, the effects of enabling reasoning are not statistically significant for the prediction of semantically aligned CWE labels. For Gemini 2.5 Flash-Lite, we observe a statistically significant effect in iteration 3, but not in the other two iterations. This suggests that the reasoning based on which the vulnerability detector classifies a code snippet as vulnerable does not consistently improve by enabling internal reasoning. Therefore, the observed differences, most notably the slight increases in recall, in the per-sample CWE classification perspective, may be a result of random chance.

5.2.4 Costs

Table 11 presents the total costs per component spent for experiment 1. The code description and LLM re-ranking remain consistent across all vulnerability detection configurations. The vulnerability detection represents the total cost of all configurations that are tested. Table 12shows the vulnerability detection costs broken down into the separate configurations. We observe that enabling reasoning increases the cost of the final vulnerability classification by 61.5%. This shows that the improvements that come with using reasoning also come at a higher cost.

Overall, Gemini 2.5 Flash-Lite is the most expensive model, whereas GPT-OSS 120B has the lowest cost. The improved performance of GPT-OSS 120B at a lower cost than Gemini 2.5 Flash-Lite is an additional reason, besides its stable performance, why we perform the rest of our experiments with GPT-OSS 120B.

Component Costs	Costs
Code Description	$1.11
LLM Re-ranking	$2.54
Vulnerability Detection	$7.48
Total	**$11.13**

Table 11: Total cost of every LLM-based component in the vulnerability detection pipeline for experiment 1.

Vulnerability Detection Costs	Reasoning Disabled	Reasoning Enabled
GPT-OSS 120B	$0.57	$0.83
Gemini 2.5 Flash-Lite	$1.18	$2.01
Gemma 4 26B A4B	$1.11	$1.78
Total	**$2.86**	**$4.62**

Table 12: Total cost per configuration for the vulnerability classification for experiment 1.

5.3 Experiment 2: Vulnerability Data Influence

In this experiment, we examine the effects of the data that is retrieved from the VKB using RAG. We focus on the retrieval of knowledge and CVE data, and perform an incremental ablation study to determine the effects of the addition of CWE data and code differences. The experiments are performed using GPT-OSS 120B. First, we cover the setup we are using for this experiment. Afterwards, we analyze and interpret the results. This experiment aims to find an answer to **SRQ2**.

5.3.1 Setup

The majority of this setup is as discussed in Section 5.1. The important configuration in this experiment is the RAG implementation. For GPT-OSS 120B, we test the effect of the retrieval of different data from the VKB in the final vulnerability classification. To determine the effects of retrieving knowledge, CVE data, CWE data, and code differences, we perform an incremental ablation study. We compare these with a no RAG baseline, and use the results to answer **SRQ2**.

There is a total of 8 configurations: 1 model run with 4 different RAG modes, with 2 reasoning modes. To isolate the effects of RAG, the other components of the pipeline are consistent for all configurations, including all non-deterministic LLM outputs in previous stages (consistent per iteration) of the pipeline. All configurations are repeated three times for consistency purposes. All results are averaged over these three iterations.

5.3.2 Results

In this section, we discuss the results of the second experiment. We analyze the results averaged over three repetitions. Across all configurations, per iteration, the total number of code snippets resulting in failures ranged between 6 and 10. These failures are excluded from our results. A detailed analysis of the failures is provided in Appendix B.2. We interpret the results per evaluation perspective.

Binary Classification	Reasoning Disabled			Reasoning Enabled		
	P	R	F1	P	R	F1
No RAG	0.563	0.431	0.488	0.534	**0.758**	0.626
RAG Knowledge + CVE	**0.621**	0.448	0.519	**0.573**	0.736	**0.644**
RAG Knowledge + CVE + CWE	0.606	**0.492**	**0.543**	0.572	0.715	0.635
RAG Knowledge + CVE + CWE + Code	0.572	0.445	0.500	0.533	0.630	0.577

Table 13: Results of binary classification for model GPT-OSS 120B with reasoning disabled compared to reasoning enabled, measuring the effect of VKB data retrieval. The results indicate that when reasoning is disabled, increasing the amount of VKB data that is retrieved increases the precision and recall slightly. When reasoning is enabled, we observe that the precision and recall also increase slightly. However, for both reasoning disabled and enabled holds that the recall is higher when no RAG is used, and that retrieving code differences leads to a decrease in precision and recall. Results are averaged over three repetitions.

Binary classification. The results of binary classification are presented in Table 13. First, when internal reasoning is disabled for GPT-OSS 120B, the usage of RAG leads to a relative increase in both precision and recall for all configurations in comparison with the no RAG baseline. Initially, when only retrieving the generated knowledge and CVE data from the VKB, precision increases (from 0.563 to 0.621; a relative increase of 10.3%). This precision slightly decreases when CWE data and code differences are additionally retrieved. This suggests that when the model is presented with knowledge and CVE data, more correct vulnerable predictions are made, but also that CWE data and code differences may influence the model's predictions too much, leading to a decrease in correct vulnerable predictions. A similar effect is observed with the recall, which grows when knowledge and CVE data are retrieved. However, when CWE data is additionally retrieved, the recall grows further. When code differences are added, this additional increase in recall from adding CWE data is diminished. This indicates that knowledge, CVE data, and CWE data lead to an increase in vulnerable code snippets being correctly classified as vulnerable. However, the addition of code differences may confuse the model, since it could expect the exact code differences in the code snippet as the code differences that are retrieved. Therefore, the configuration that leads to the highest F1 score is retrieving knowledge, CVE data, and CWE data (F1 score of 0.543). The results when internal reasoning is enabled show that precision also increases when retrieving knowledge and CVE data (from 0.534 to 0.573; a relative increase of 7.3%). This suggests that providing the model with knowledge and CVE data positively supports the model in making vulnerable predictions. The addition of CWE data does not have a consistent effect on the precision, but the addition of code differences lowers the precision. Recall decreases when the RAG pipeline is enabled, and keeps decreasing for each additional data type that is retrieved. This suggests that the internal reasoning becomes biased towards the retrieved data when it is provided. The increase in precision when knowledge and CVE data are retrieved is larger than the decrease in recall in comparison with the no RAG baseline, leading to the highest F1 score (F1 score of 0.644).

Per-sample CWE classification. Table 14 presents the per-sample CWE classification results. The results are visualized in Figure 10 (absolute scale) and Figure 11 (relative scale). When internal reasoning is disabled, we observe increases in precision and recall when RAG is enabled compared to when RAG is disabled. The highest precision is present when knowledge, CVE data, and CWE data are retrieved (from 0.389 to 0.414; a relative improvement of 4.0%, compared to the no RAG baseline). This same configuration leads to the highest recall (from 0.298 to 0.335; a relative improvement of 12.4% compared to the no RAG baseline), and thus to the highest F1 score. This indicates that the addition of knowledge, CVE data, and CWE data supports the model in predicting more CWE labels that are semantically aligned with the ground truth, and supports the model in reducing the number of false CWE labels predicted. When internal reasoning is enabled, we observe a trend that follows binary classification: the precision grows with the addition of knowledge and CVE data (from 0.404 to 0.441; a relative increase of 9.2% compared to the baseline of no RAG), remains stable with the addition of CWE data, and decreases when code differences are added. Additionally, the recall shows the same trend of decreases when more data is retrieved. This leads to the retrieval of knowledge and CVE data having the highest F1 score, even though neither the precision nor the recall is the absolute highest. This indicates that RAG has a mixed influence when internal reasoning is enabled; the number of correct CWE label predictions increases, but the number of correctly predicted CWE labels decreases. This shows that the retrieved data has a different effect when internal reasoning is disabled and enabled.

Per-Sample CWE Classification	Reasoning Disabled			Reasoning Enabled		
	P	R	F1	P	R	F1
No RAG						
Exact	0.042	0.032	0.036	0.019	0.027	0.022
Partial	0.145	0.112	0.126	0.126	0.179	0.148
Relaxed - Family	0.221	0.170	0.192	0.252	0.358	0.296
Relaxed - Causal	0.243	0.186	0.211	0.233	0.332	0.274
Relaxed - Categorical	0.222	0.170	0.193	0.237	0.337	0.278
Relaxed - Combined	**0.389**	**0.298**	**0.337**	**0.404**	**0.574**	**0.474**
RAG Knowledge + CVE						
Exact	0.052	0.037	0.043	0.037	0.048	0.042
Partial	0.154	0.109	0.128	0.140	0.180	0.158
Relaxed - Family	0.269	0.193	0.224	0.288	0.370	0.324
Relaxed - Causal	0.228	0.162	0.189	0.269	0.345	0.302
Relaxed - Categorical	0.242	0.172	0.200	0.268	0.345	0.301
Relaxed - Combined	**0.410**	**0.292**	**0.340**	**0.441**	**0.566**	**0.496**
RAG Knowledge + CVE + CWE						
Exact	0.053	0.042	0.0047	0.047	0.059	0.052
Partial	0.117	0.094	0.105	0.119	0.151	0.133
Relaxed - Family	0.271	0.220	0.243	0.257	0.323	0.286
Relaxed - Causal	0.265	0.215	0.237	0.256	0.323	0.285
Relaxed - Categorical	0.233	0.189	0.208	0.231	0.291	0.257
Relaxed - Combined	**0.414**	**0.335**	**0.370**	**0.446**	**0.559**	**0.495**
RAG Knowledge + CVE + CWE + Code						
Exact	0.061	0.048	0.053	0.036	0.042	0.039
Partial	0.114	0.090	0.100	0.112	0.132	0.121
Relaxed - Family	0.252	0.196	0.220	0.242	0.286	0.262
Relaxed - Causal	0.240	0.190	0.212	0.233	0.275	0.252
Relaxed - Categorical	0.209	0.164	0.184	0.230	0.270	0.248
Relaxed - Combined	**0.398**	**0.312**	**0.350**	**0.408**	**0.482**	**0.442**

Table 14: Results per-sample CWE classification for GPT-OSS 120B with reasoning disabled compared to reasoning enabled, measuring the effect of VKB data retrieval. The results indicate that retrieving knowledge, CVE data, and CWE data results in the highest precision and recall scores when internal reasoning is disabled. When internal reasoning is enabled, the highest F1 score is obtained when knowledge and CVE data are retrieved. Results are averaged over three repetitions.

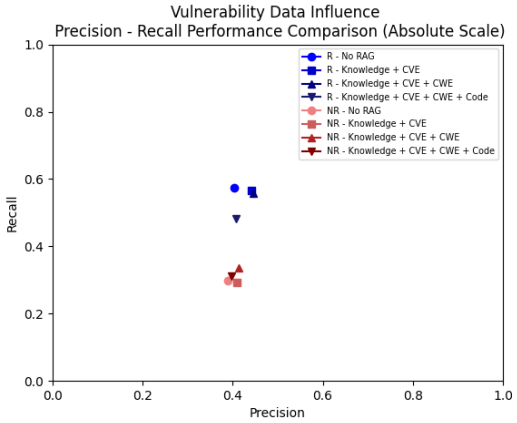


Figure 10: Per-sample CWE classification absolute precision-recall performance comparison. Red markers represent internal reasoning disabled, blue markers internal reasoning enabled. Represents the Relaxed - Combined precision and recall scores.

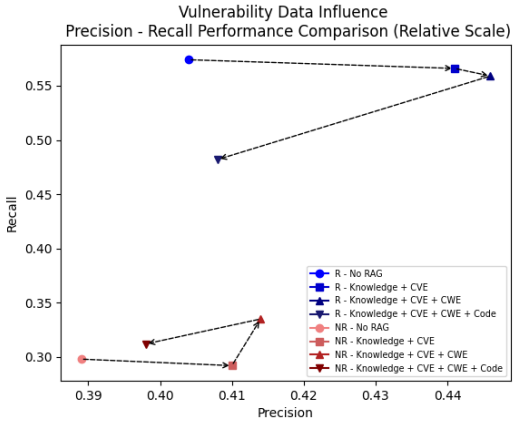


Figure 11: Per-sample CWE classification relative precision-recall performance comparison. Red markers represent internal reasoning disabled, blue markers internal reasoning enabled. Represents the Relaxed - Combined precision and recall scores.

Per-label CWE classification. Finally, Table 15 presents the per-label CWE classification results. When internal reasoning is disabled, we observe an increase in precision when RAG is enabled compared to the no RAG baseline. The retrieval of knowledge, CVE data, and CWE data leads to the highest precision (from 0.152 to 0.214; a relative improvement of 40.8%). Removing CWE data or adding code differences decreases the precision. The effect of the incremented configurations is inconsistent for the recall. However, the highest recall is observed when knowledge, CVE data, and CWE data are retrieved (from 0.140 to 0.173; a relative improvement of 23.6%); the same configuration that has the highest precision. This shows that retrieving knowledge, CVE data, and CWE data leads to the absolute highest precision and recall, subsequently leading to the highest F1 score (F1 score of 0.191). This suggests that RAG has a positive influence on fine-grained CWE label classification, and is in line with the per-sample CWE classification results. When internal reasoning is enabled, we also observe a trend in line with the per-sample CWE classification results: the highest precision is observed when knowledge, CVE data, and CWE data are retrieved (from 0.123 to 0.176; a relative improvement of 43.1% compared to the no RAG baseline). The recall generally decreases the more data is retrieved from the VKB, suggesting that the addition of more data increases the influence of the retrieved entries on the model. However, the highest observed precision is a larger increase than the decrease in recall when knowledge, CVE data, and CWE data are retrieved, leading to this configuration having the highest F1 score (F1 score of 0.209).

Overall, we observe that RAG has a larger positive influence on both the precision and recall when internal reasoning is disabled. Especially, the addition of knowledge, CVE data, and CWE data leads to the largest increases in both. When internal reasoning is enabled, the effects are less impactful and more inconsistent. The precision slightly increases with the addition of knowledge, CVE data, and CWE data, but the recall slightly decreases when more data is retrieved. The addition of code differences for both internal reasoning disabled and enabled shows a smaller increase or even a decrease in precision and recall across the evaluation perspectives. This indicates that the retrieval of code differences does not positively influence the vulnerability detection and classification task.

Per-Label CWE Classification	Reasoning Disabled			Reasoning Enabled		
	P	R	F1	P	R	F1
No RAG						
Exact	0.091	0.084	0.087	0.066	0.139	0.089
Relaxed - Family	0.121	0.112	0.116	0.093	0.194	0.125
Relaxed - Causal	0.104	0.096	0.100	0.085	0.178	0.115
Relaxed - Categorical	0.121	0.112	0.117	0.090	0.190	0.122
Relaxed - Combined	**0.152**	**0.140**	**0.146**	**0.123**	**0.257**	**0.166**
RAG Knowledge + CVE						
Exact	0.104	0.082	0.092	0.082	0.139	0.104
Relaxed - Family	0.143	0.114	0.126	0.125	0.211	0.157
Relaxed - Causal	0.114	0.090	0.101	0.099	0.167	0.124
Relaxed - Categorical	0.137	0.106	0.118	0.113	0.191	0.142
Relaxed - Combined	**0.173**	**0.137**	**0.153**	**0.162**	**0.275**	**0.204**
RAG Knowledge + CVE + CWE						
Exact	0.088	0.071	0.078	0.077	0.113	0.091
Relaxed - Family	0.126	0.103	0.113	0.118	0.174	0.140
Relaxed - Causal	0.127	0.102	0.113	0.109	0.162	0.130
Relaxed - Categorical	0.150	0.122	0.135	0.115	0.170	0.137
Relaxed - Combined	**0.214**	**0.173**	**0.191**	**0.176**	**0.258**	**0.209**
RAG Knowledge + CVE + CWE + Code						
Exact	0.091	0.068	0.077	0.071	0.100	0.083
Relaxed - Family	0.136	0.099	0.115	0.097	0.136	0.133
Relaxed - Causal	0.127	0.095	0.109	0.096	0.136	0.113
Relaxed - Categorical	0.133	0.099	0.114	0.117	0.163	0.136
Relaxed - Combined	**0.204**	**0.151**	**0.173**	**0.171**	**0.239**	**0.199**

Table 15: Results per-label CWE classification for GPT-OSS 120B with reasoning disabled compared to reasoning enabled, measuring the effect of VKB data retrieval. The results indicate that retrieving knowledge, CVE data, and CWE data results in the highest precision and recall scores when internal reasoning is disabled. When internal reasoning is enabled, the highest F1 score is also obtained when knowledge, CVE data, and CWE data are retrieved. Results are averaged over three repetitions.

5.3.3 Statistical Significance

To determine the reliability of our results, we perform McNemar's test on the results of the binary classification and per-sample CWE classification. The test is explained in Section 5.1.5. We observe in Table 16 that none of the comparisons lead to consistently significant differences. This suggests that for the implementation of RAG, and for the incremental addition of different retrieved data, no significant effect between the configurations is found. Additionally, in the majority of cases, the direction of the observed effects is not consistent either. Thus, our results in Section 5.3 are indicative rather than reliable results.

Observing Table 17, no statistically significant results are shown. Nevertheless, the directions per RAG configuration remain consistent for all iterations and reasoning modes. This indicates that RAG, with the retrieval of knowledge and CVE data, shows a tendency to decrease in CWE

	Reasoning Disabled		Reasoning Enabled	
Class	Vulnerable	Safe	Vulnerable	Safe
No RAG ↔ RAG Knowledge + CVE				
Iteration 1	$\rightarrow 1.0$	$\leftarrow 8.04 \times 10^{-1}$	$\leftarrow 2.89 \times 10^{-1}$	$\rightarrow 2.67 \times 10^{-1}$
Iteration 2	$\leftarrow 3.52 \times 10^{-2}$	$\rightarrow 1.0$	$\leftarrow 4.54 \times 10^{-1}$	$\rightarrow 4.24 \times 10^{-1}$
Iteration 3	$\leftarrow 1.67 \times 10^{-1}$	$\leftarrow 2.89 \times 10^{-1}$	$\leftarrow 6.07 \times 10^{-1}$	$\rightarrow 5.81 \times 10^{-1}$
RAG Knowledge + CVE ↔ + CWE				
Iteration 1	$\rightarrow 7.74 \times 10^{-1}$	$\leftarrow 7.74 \times 10^{-1}$	$\leftarrow 3.44 \times 10^{-1}$	$\rightarrow 1.46 \times 10^{-1}$
Iteration 2	$\rightarrow \mathbf{1.27 \times 10^{-2}}$	$\leftarrow 8.04 \times 10^{-1}$	$\rightarrow 1.0$	$\rightarrow 1.0$
Iteration 3	$\rightarrow 1.0$	$\rightarrow 1.0$	$\leftarrow 7.74 \times 10^{-1}$	$\leftarrow 3.44 \times 10^{-1}$
RAG Knowledge + CVE + CWE ↔ + Code				
Iteration 1	$\leftarrow 5.49 \times 10^{-1}$	$\rightarrow 1.09 \times 10^{-1}$	$\leftarrow 1.0$	$\leftarrow 1.09 \times 10^{-1}$
Iteration 2	$\leftarrow 5.81 \times 10^{-1}$	$\rightarrow 1.0$	$\leftarrow 2.27 \times 10^{-1}$	$\rightarrow 7.74 \times 10^{-1}$
Iteration 3	$\rightarrow 1.0$	$\leftarrow 7.74 \times 10^{-1}$	$\rightarrow 1.0$	$\rightarrow 3.52 \times 10^{-2}$

Table 16: P-values for the statistical tests performed on binary classification to determine statistically significant differences between several RAG configurations in experiment 2. The significance threshold is set to $\alpha = 0.05$. The P-values are given per class, per iteration, and statistical significance is determined after performing Holm-Bonferroni correction. Bold values are statistically significant. Three comparisons between RAG configurations are evaluated. Results show that for all RAG configurations, no consistent significant results are found. The left arrow ($\leftarrow$) indicates that the first configuration in the corresponding comparison performs better, while the right arrow ($\rightarrow$) indicates that the second configuration in the corresponding comparison performs better.

	Reasoning Disabled	Reasoning Enabled
Class	Vulnerable	Vulnerable
No RAG ↔ RAG Knowledge + CVE		
Iteration 1	$\leftarrow 6.88 \times 10^{-1}$	$\leftarrow 5.08 \times 10^{-1}$
Iteration 2	$\leftarrow 3.75 \times 10^{-1}$	$\leftarrow 1.0$
Iteration 3	$\leftarrow 1.0$	$\leftarrow 1.0$
RAG Knowledge + CVE ↔ + CWE		
Iteration 1	$\rightarrow 6.25 \times 10^{-1}$	$\rightarrow 1.0$
Iteration 2	$\rightarrow 3.75 \times 10^{-1}$	$\rightarrow 1.0$
Iteration 3	$\rightarrow 1.0 \times 10^{-1}$	$\rightarrow 1.0$
RAG Knowledge + CVE + CWE ↔ + Code		
Iteration 1	$\rightarrow 1.0$	$\leftarrow 1.0$
Iteration 2	$\rightarrow 1.0$	$\leftarrow 6.88 \times 10^{-1}$
Iteration 3	$\rightarrow 1.0$	$\leftarrow 3.75 \times 10^{-1}$

Table 17: P-values for the statistical tests performed on per-sample CWE classification to determine statistically significant differences between several RAG configurations in experiment 2. The significance threshold is set to $\alpha = 0.05$. The P-values are given per class and per iteration. Bold values are statistically significant. Three comparisons between RAG configurations are evaluated. Results show that no RAG configurations lead to significant differences. The left arrow ($\leftarrow$) indicates that the first configuration in the corresponding comparison performs better, while the right arrow ($\rightarrow$) indicates that the second configuration in the corresponding comparison performs better.

label classification performance for both reasoning modes, but no statistical evidence is found for this. However, when comparing this configuration with the addition of CWE retrieval, the results show a non-significant tendency that this addition improves CWE classification when reasoning is disabled. When reasoning is enabled, inconsistent effects are observed. Additional code difference retrieval indicates an inconsistent effect as well when reasoning is disabled, but a possible indicative negative effect is observed when reasoning is enabled, without any statistical evidence.

5.3.4 Costs

Table 18 presents the total costs per component spent for experiment 2. The code description and LLM re-ranking remain consistent across all vulnerability detection configurations. The vulnerability detection represents the total cost of all configurations tested. Table 19 shows the vulnerability detection costs broken down into separate configurations. We observe that the addition of RAG increases the costs when both internal reasoning is disabled (by approximately 22.8%) and enabled (by approximately 19.7%). This is likely due to GPT-OSS 120B spending more tokens to use these relevant entries in its classification process. We see that retrieving and processing different relevant data in the prompt does not have a substantial influence on the cost of the model.

Component Costs	Costs
Code Description	$1.13
LLM Re-ranking	$2.56
Vulnerability Detection	$5.63
Total	**$9.32**

Table 18: Total cost of every LLM-based component in the vulnerability detection pipeline for experiment 2.

Configuration Costs	Costs Reasoning Disabled	Costs Reasoning Enabled
No RAG	$0.48	$0.76
RAG Knowledge + CVE	$0.59	$0.95
RAG Knowledge + CVE + CWE	$0.60	$0.87
RAG Knowledge + CVE + CWE + Code	$0.57	$0.91
Total	**$2.24**	**$3.49**

Table 19: Total cost per configuration for the vulnerability classification for experiment 2.

5.4 Experiment 3: LLM Re-ranker Influence

In this experiment, we examine the effects of RAG and the influence of the LLM re-ranking step on vulnerability detection. We compare these effects when reasoning is disabled and when reasoning is enabled. The experiments are performed using GPT-OSS 120B. First, we cover the setup we are using for this experiment. Afterwards, we analyze and interpret the results. This experiment aims to uncover an answer to **SRQ3**.

5.4.1 Setup

The majority of the setup is as discussed in Section 5.1. The important configuration in this experiment is the RAG implementation. For GPT-OSS 120B, we test the effect of RAG with only BM25 and RAG with both BM25 and LLM re-ranking. We compare the effects with reasoning disabled compared to reasoning enabled, and use the results to answer **SRQ3**.

There is a total of 6 configurations: 1 model run with 3 different RAG modes, with 2 reasoning modes. To isolate the effects of RAG, the other components of the pipeline are consistent for all configurations, including all non-deterministic LLM outputs in previous stages (consistent per iteration) of the pipeline. All configurations are repeated three times for consistency purposes. All results are averaged over these three iterations.

5.4.2 Results

In this section, we discuss the results of the third experiment. We analyze the results averaged over three repetitions. Across all configurations, per iteration, the total number of code snippets resulting in failures ranged between 6 and 12. These failures are excluded from our results. A detailed analysis of the failures is provided in Appendix B.2. We interpret the results per evaluation perspective.

Binary classification. First, the results presented in Table 20 show the effects of using RAG on GPT-OSS 120B on binary classification. We observe that with reasoning disabled, using RAG has a positive influence; both with BM25 only, and with BM25 with LLM re-ranking, the precision increases compared to not using RAG. The addition of the LLM re-ranker leads to

Binary Classification	Reasoning Disabled			Reasoning Enabled		
	P	R	F1	P	R	F1
No RAG	0.526	0.384	0.443	0.542	**0.741**	**0.626**
RAG BM25 Only	**0.600**	0.341	0.434	**0.545**	0.638	0.587
RAG BM25 + LLM Re-ranking	0.593	**0.445**	**0.508**	0.541	0.658	0.594

Table 20: Results of binary classification for model GPT-OSS 120B with reasoning disabled compared to reasoning enabled, measuring the effect of BM25 and LLM re-ranking. The results indicate that without internal reasoning, LLM re-ranking has a positive effect on both the precision and recall compared to not using RAG. When reasoning is enabled, we see that the RAG pipeline negatively influences the recall of the model, while keeping the precision stable. This shows that RAG has a different effect on a model when reasoning is disabled compared to when enabled. Repetitions are averaged over three repetitions.

Per-Sample CWE Classification	Reasoning Disabled			Reasoning Enabled		
	P	R	F1	P	R	F1
No RAG						
Exact	0.045	0.032	0.037	0.008	0.011	0.009
Partial	0.138	0.100	0.116	0.144	0.196	0.166
Relaxed - Family	0.239	0.174	0.201	0.268	0.365	0.309
Relaxed - Causal	0.198	0.148	0.169	0.268	0.365	0.309
Relaxed - Categorical	0.204	0.147	0.171	0.237	0.323	0.273
Relaxed - Combined	**0.360**	**0.263**	**0.303**	**0.419**	**0.571**	**0.483**
RAG BM25 Only						
Exact	0.092	0.052	0.069	0.055	0.064	0.059
Partial	0.138	0.079	0.100	0.124	0.145	0.133
Relaxed - Family	0.269	0.152	0.194	0.257	0.300	0.276
Relaxed - Causal	0.222	0.126	0.161	0.243	0.284	0.262
Relaxed - Categorical	0.222	0.126	0.160	0.234	0.273	0.252
Relaxed - Combined	**0.463**	**0.262**	**0.334**	**0.438**	**0.515**	**0.473**
RAG BM25 + LLM Re-ranking						
Exact	0.077	0.058	0.066	0.056	0.068	0.062
Partial	0.133	0.100	0.114	0.134	0.163	0.147
Relaxed - Family	0.245	0.183	0.209	0.251	0.305	0.275
Relaxed - Causal	0.252	0.189	0.215	0.251	0.305	0.275
Relaxed - Categorical	0.237	0.178	0.203	0.238	0.289	0.261
Relaxed - Combined	**0.433**	**0.325**	**0.371**	**0.437**	**0.531**	**0.479**

Table 21: Results per-sample CWE classification for GPT-OSS 120B with RAG disabled, RAG using only BM25, and RAG using both BM25 and LLM re-ranking, all with reasoning disabled compared to reasoning enabled. Results indicate that using RAG with either BM25 only or BM25 with LLM re-ranking improves precision in both reasoning modes, but improves recall only when reasoning is disabled. Results are averaged over three repetitions.

a larger increase in precision than using only BM25. The unique effect that LLM re-ranking has over only using BM25 is that it increases the recall (from 0.384 to 0.445; a relative increase of 15.9%) compared to not using RAG. This indicates that using an LLM re-ranker has a positive influence when reasoning is disabled. However, when reasoning is enabled, we see the opposite effect: the recall decreases both when only BM25 is used (from 0.741 to 0.638; a relative decrease of 13.9%) and when BM25 is used in combination with LLM re-ranking (from 0.741 to 0.658; a relative decrease of 11.2%), in comparison to not using RAG. The precision stays relatively consistent between only using BM25, and using BM25 with an LLM re-ranker. This indicates that using RAG decreases the number of identified vulnerable samples, while precision remains stable.

Per-sample CWE classification. When examining the per-sample CWE classification perspective in Table 21, visualized in Figure 12 (absolute scale) and Figure 13 (relative scale), an initial observation is that the exact matching between all predicted CWE labels and all assigned CWE labels per sample leads to low precision and recall scores. This holds for all

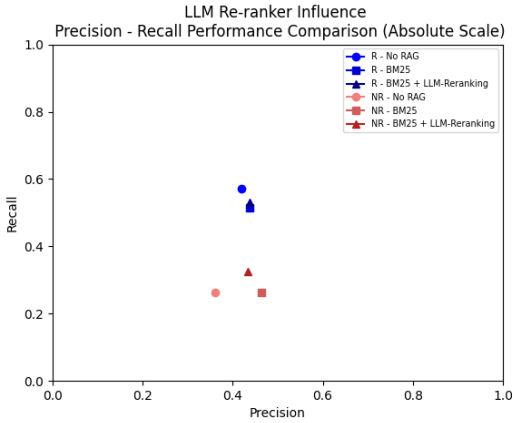


Figure 12: Per-sample CWE classification absolute precision-recall performance comparison. Red markers represent internal reasoning disabled, blue markers internal reasoning enabled. Represents the Relaxed - Combined precision and recall scores.

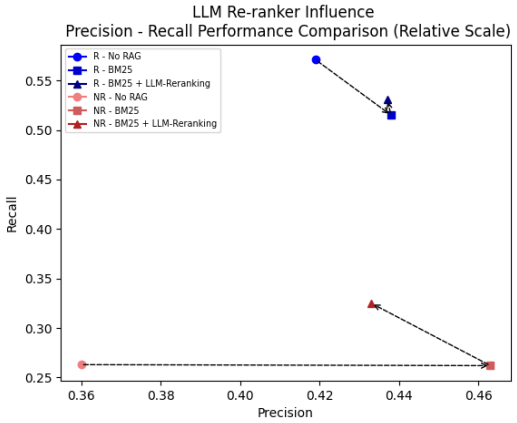


Figure 13: Per-sample CWE classification relative precision-recall performance comparison. Red markers represent internal reasoning disabled, blue markers internal reasoning enabled. Represents the Relaxed - Combined precision and recall scores.

RAG implementations for both reasoning disabled and reasoning enabled. However, when observing the partial matching between the predicted and assigned CWE labels, we see an increase in both precision and recall scores across all configurations. The resulting values of partial matching show that when reasoning is disabled, there is no substantial difference between these increases when comparing the different RAG configurations. There is a slightly lower increase in recall when RAG is used with only BM25 compared to not using RAG. When reasoning is enabled, not using RAG appears to reach a slightly higher precision and recall score than when using RAG with both configurations. However, when viewing the results under our relaxed framework, we see more substantial differences. When reasoning is disabled, when using RAG with only BM25, we see a relative increase of 28.6% in precision and a stable recall. We observe that when using BM25 with LLM re-ranking, there is a relative increase of 20.3% in precision and a relative increase of 23.6% in recall compared to not using RAG. This indicates that using only BM25 leads to a slightly larger gain in precision, but using an LLM re-ranker leads to increases in both precision and recall, contributing to a larger gain in F1 score. The F1 score of using RAG with BM25 and with LLM re-ranking increases (0.303 to 0.371; a relative increase of 22.4%) more than only using BM25 (0.303 to 0.334; a relative increase of 10.2%) in comparison to not using RAG. This shows the positive effect of using BM25 with an LLM re-ranker with reasoning disabled. When reasoning is enabled, we see a similar increase in precision when using RAG with only BM25 and when using RAG with BM25 with an LLM re-ranker, albeit the difference in precision is more stable across the different configurations. The opposite effect can be observed when comparing the recall. There is a decrease in recall when using RAG with only BM25 in comparison with not using RAG (0.571 to 0.515; a relative decrease of 9.8%). There is also a decrease in recall when using RAG with BM25 and an LLM re-ranker, albeit it is a smaller decrease (0.571 to 0.531; a relative decrease of 7%). This leads to a relatively stable F1 score (a difference of at most 0.010 across the three configurations), where using RAG leads to a higher precision score, and not using RAG leads to a higher recall score.

Per-Label CWE Classification	Reasoning Disabled			Reasoning Enabled		
	P	R	F1	P	R	F1
No RAG						
Exact	0.086	0.075	0.080	0.072	0.147	0.096
Relaxed - Family	0.117	0.103	0.109	0.103	0.210	0.138
Relaxed - Causal	0.090	0.079	0.084	0.096	0.194	0.128
Relaxed - Categorical	0.095	0.083	0.088	0.083	0.171	0.112
Relaxed - Combined	**0.126**	**0.111**	**0.118**	**0.134**	**0.274**	**0.180**
RAG BM25 Only						
Exact	0.107	0.059	0.076	0.082	0.108	0.093
Relaxed - Family	0.151	0.083	0.107	0.123	0.164	0.141
Relaxed - Causal	0.107	0.059	0.076	0.094	0.124	0.107
Relaxed - Categorical	0.130	0.071	0.092	0.109	0.144	0.124
Relaxed - Combined	**0.174**	**0.095**	**0.122**	**0.157**	**0.208**	**0.179**
RAG BM25 + LLM Re-ranking						
Exact	0.100	0.075	0.085	0.088	0.123	0.103
Relaxed - Family	0.146	0.110	0.126	0.122	0.171	0.142
Relaxed - Causal	0.121	0.091	0.103	0.108	0.151	0.126
Relaxed - Categorical	0.136	0.102	0.117	0.125	0.174	0.145
Relaxed - Combined	**0.193**	**0.146**	**0.166**	**0.179**	**0.250**	**0.208**

Table 22: Results per-label CWE classification for GPT-OSS 120B with RAG disabled, RAG using only BM25, and RAG using both BM25 and LLM re-ranking, all with reasoning disabled compared to reasoning enabled. Results indicate that using RAG with either BM25 only or BM25 with LLM re-ranking improves precision in both reasoning modes, but improves recall only when reasoning is disabled. Results are averaged over three repetitions.

Per-label CWE classification. Finally, when examining Table 22, we observe that both RAG with only BM25, and BM25 with LLM re-ranking have a positive influence on the precision when reasoning is disabled and enabled compared to not using RAG. In both cases, the precision gain is larger when using RAG with BM25 with an LLM re-ranker than when using RAG with only BM25, compared to not using RAG. A different effect can be seen for recall. When reasoning is disabled, only using BM25 compared to using no RAG leads to a decrease in recall (from 0.111 to 0.095, a relative decrease of 14.4%). However, when using BM25 with LLM re-ranking in comparison with not using RAG, we see an increase in recall (from 0.111 to 0.146, a relative improvement of 31.5%). This suggests that the retrieval of more semantically relevant VKB entries may improve recall, while the retrieval solely based on term overlap appears to be weaker and retrieves less relevant entries, and therefore leads to a decrease in recall. Viewing the results when reasoning is enabled, we see a similar effect in the increase in precision across the three configurations. However, in contrast to when reasoning is disabled, there is a decrease in recall when using both RAG with only BM25 (0.274 to 0.208; a relative decrease of 24.1%) and RAG with BM25 with an LLM re-ranker (0.274 to 0.250; a relative decrease of 8.8%) compared to not using RAG. Nevertheless, when using only BM25, the gains in precision and losses in recall balance each other out and lead to an F1 score consistent with not using RAG (0.180 to 0.179; a relative decrease of 0.6%). The larger

increase in precision and smaller decrease in recall that come with RAG with BM25 with LLM re-ranking lead to a higher F1 score compared to not using RAG (0.180 to 0.208; a relative increase of 15.6%). This shows that the retrieval of semantically more relevant CWE labels leads to better fine-grained CWE label prediction with both reasoning disabled and reasoning enabled.

Overall, we observe the positive effect of using RAG with BM25 and an LLM re-ranker compared to both not using RAG and using RAG with only BM25. The effects are stronger when reasoning is disabled, with substantial increases in F1 scores for binary classification and per-sample and per-label CWE classification. Even though the performance of binary classification slightly decreases with the use of RAG when reasoning is enabled, we see that the vulnerable predictions that are made more often semantically align with the assigned CWE labels, which is reflected by the gain in F1 score in the relaxed framework of per-label CWE classification when using RAG with BM25 with an LLM re-ranker in comparison to not using RAG.

5.4.3 Statistical Significance

To determine the reliability of our results, we perform McNemar's test on the results of the binary classification and per-sample CWE classification. The test is explained in Section 5.1.5. We observe in Table 23 that there is no statistical evidence that using RAG with solely BM25 leads to improved binary classification compared to not using RAG. Additionally, no significant results are shown that indicate that using RAG with BM25 and an LLM re-ranker improves binary classification over RAG with only BM25.

	Reasoning Disabled		**Reasoning Enabled**	
Class	Vulnerable	Safe	Vulnerable	Safe
No RAG $\leftrightarrow$ RAG BM25				
Iteration 1	$\leftarrow \mathbf{2.15 \times 10^{-2}}$	$\leftarrow 6.29 \times 10^{-1}$	$\leftarrow 1.18 \times 10^{-1}$	$\rightarrow 2.63 \times 10^{-1}$
Iteration 2	$\leftarrow 7.91 \times 10^{-1}$	$\rightarrow 1.80 \times 10^{-1}$	$\leftarrow \mathbf{1.06 \times 10^{-2}}$	$\rightarrow 1.0$
Iteration 3	$\leftarrow 4.90 \times 10^{-2}$	$\rightarrow 4.90 \times 10^{-2}$	$\leftarrow 2.38 \times 10^{-1}$	$\rightarrow 3.59 \times 10^{-1}$
RAG BM25 $\leftrightarrow$ RAG BM25 + LLM Re-ranking				
Iteration 1	$\rightarrow 3.86 \times 10^{-2}$	$\leftarrow 6.29 \times 10^{-1}$	$\rightarrow 4.54 \times 10^{-1}$	$\leftarrow 2.10 \times 10^{-1}$
Iteration 2	$\rightarrow 6.07 \times 10^{-1}$	$\leftarrow 3.02 \times 10^{-1}$	$\rightarrow 3.02 \times 10^{-1}$	$\rightarrow 7.73 \times 10^{-1}$
Iteration 3	$\rightarrow 5.74 \times 10^{-2}$	$\leftarrow 5.74 \times 10^{-2}$	$\leftarrow 6.48 \times 10^{-1}$	$\leftarrow 3.02 \times 10^{-1}$

Table 23: P-values for the statistical tests performed on binary classification to determine statistically significant differences between several RAG configurations in experiment 3. The significance threshold is set to $\alpha = 0.05$. The P-values are given per class, per iteration, and statistical significance is determined after performing Holm-Bonferroni correction. Bold values are statistically significant. Two comparisons between RAG configurations are evaluated. Results show that for all RAG configurations, no consistent significant results are found. The left arrow ($\leftarrow$) indicates that the first configuration in the corresponding comparison performs better, while the right arrow ($\rightarrow$) indicates that the second configuration in the corresponding comparison performs better.

	Reasoning Disabled	**Reasoning Enabled**
Class	Vulnerable	Vulnerable
No RAG $\leftrightarrow$ RAG BM25		
Iteration 1	$\leftarrow 1.80 \times 10^{-1}$	$\rightarrow 1.0$
Iteration 2	$\rightarrow 2.19 \times 10^{-1}$	$\leftarrow 3.44 \times 10^{-1}$
Iteration 3	$\rightarrow 1.0$	$\leftarrow 5.08 \times 10^{-1}$
RAG BM25 $\leftrightarrow$ RAG BM25 + LLM Re-ranking		
Iteration 1	$\rightarrow 6.88 \times 10^{-1}$	$\rightarrow 7.54 \times 10^{-1}$
Iteration 2	$\leftarrow 1.25 \times 10^{-1}$	$\leftarrow 1.0$
Iteration 3	$\leftarrow 1.0$	$\leftarrow 1.0$

Table 24: P-values for the statistical tests performed on per-sample CWE classification to determine statistically significant differences between several RAG configurations in experiment 3. The significance threshold is set to $\alpha = 0.05$. The P-values are given per class and per iteration. Bold values are statistically significant. Two comparisons between RAG configurations are evaluated. Results show that no RAG configurations lead to significant differences. The left arrow ($\leftarrow$) indicates that the first configuration in the corresponding comparison performs better, while the right arrow ($\rightarrow$) indicates that the second configuration in the corresponding comparison performs better.

Table 24 shows that there is no statistical evidence that CWE classification quality is improved or decreased due to the usage of RAG with BM25 and RAG with BM25 and an LLM re-ranker; no statistically significant results are found, and directions are inconsistent across iterations.

5.4.4 Costs

Table 25 presents the total costs per component spent for experiment 3. The code description and LLM re-ranking remain consistent across all vulnerability detection configurations. The vulnerability detection component in the table represents the total cost of all configurations that are tested. Table 26 shows the vulnerability detection costs broken down into the separate configurations. We observe that providing additional relevant entries results in higher costs both when reasoning is disabled and enabled. This is likely due to GPT-OSS 120B spending more tokens to use these relevant entries in its classification process. Differences between only using BM25 and using BM25 with an LLM re-ranker remain small, showing that higher-quality retrieved relevant entries do not necessarily increase the cost of the final vulnerability classification.

Component Costs	Costs
Code Description	$1.14
LLM Re-ranking	$2.53
Vulnerability Detection	$4.24
Total	**$7.91**

Table 25: Total cost of every LLM-based component in the vulnerability detection pipeline for experiment 3.

Configuration Costs	Costs Reasoning Disabled	Costs Reasoning Enabled
No RAG	$0.49	$0.69
RAG Only BM25	$0.59	$0.95
RAG BM25 + LLM Re-ranking	$0.65	$0.87
Total	**$1.73**	**$2.51**

Table 26: Total cost per configuration for the vulnerability classification for experiment 3.

5.5 Experiment 4: VKB Data Availability Influence

In this experiment, we investigate the effect of data availability within the VKB on the RAG process in the vulnerability detection pipeline. We order the CVEs in the VKB by their publication date, and split the VKB into three equal parts. We then test whether incrementing the VKB with more recent data improves the final vulnerability detection and classification. The experiments are performed using GPT-OSS 120B. First, we cover the setup we are using for this experiment. Afterwards, we analyze and interpret the results. This experiment aims to uncover an answer to **SRQ4**.

5.5.1 Setup

The majority of the setup is as discussed in Section 5.1. The important configuration of this experiment is the data availability within the VKB. For GPT-OSS 120B, we test the effects when it is provided with relevant entries from the VKB with data availability restrictions. We use these results to answer **SRQ4**.

There is a total of 8 configurations: 1 model ran with 4 different VKB sizes, with 2 reasoning modes. The baseline in our experiment is not using RAG. The four different VKB sizes are, chronologically, no VKB (no RAG), the first third of the VKB, the first and second third of the VKB, and the full VKB. All other components of the pipeline are consistent for all configurations, including all non-deterministic LLM outputs in previous stages (consistent per iteration) of the pipeline. All configurations are repeated three times for consistency purposes. All results are averaged over these three iterations.

5.5.2 Results

In this section, we discuss the results of the fourth experiment. We analyze the results averaged over three repetitions. Across all configurations, per iteration, the total number of code snippets resulting in failures ranged between 6 and 14. These failures are excluded from our results. A detailed analysis of the failures is provided in Appendix B.4. We interpret the results per evaluation perspective.

Binary Classification	Reasoning Disabled			Reasoning Enabled		
	P	R	F1	P	R	F1
No RAG	0.563	**0.401**	0.468	0.517	**0.781**	**0.623**
RAG First Third VKB	0.573	0.350	0.434	**0.565**	0.661	0.609
RAG First and Second Third	0.579	0.376	0.456	0.544	0.688	0.608
RAG Full VKB	**0.588**	0.391	**0.469**	0.545	0.684	0.607

Table 27: Results of binary classification for model GPT-OSS 120B with reasoning disabled compared to reasoning enabled, measuring the effect of VKB data availability. The results indicate that when reasoning is disabled, increasing the VKB data availability increases the precision and recall slightly. When reasoning is enabled, we observe that the precision and recall also increase slightly. However, for both reasoning enabled and disabled holds that the recall is higher when no RAG is used. Repetitions are averaged over three repetitions.

Binary classification. The results of binary classification are presented in Table 27. First, when reasoning is disabled for GPT-OSS 120B, we observe small increases in precision every time we add more new data. The precision grows from 0.563 with no RAG to 0.588 with the full VKB, a relative increase of 4.4%. When RAG is enabled, the model's recall benefits from more recent data in the VKB as well. However, the recall when RAG is disabled is slightly higher, a difference of 0.01. The slight increase in precision and slight decrease in recall result in a relatively consistent F1 score for no RAG in comparison to RAG with the full VKB. When only a part of the VKB is available, the F1 scores are lower. When internal reasoning is enabled, we observe a different effect. Using the first third of the VKB leads to a higher precision than

Per-Sample CWE Classification	Reasoning Disabled			Reasoning Enabled		
	P	R	F1	P	R	F1
No RAG						
Exact	0.060	0.043	0.050	0.018	0.027	0.021
Partial	0.135	0.097	0.113	0.137	0.207	0.165
Relaxed - Family	0.278	0.198	0.231	0.278	0.420	0.334
Relaxed - Causal	0.248	0.177	0.207	0.253	0.383	0.304
Relaxed - Categorical	0.188	0.137	0.156	0.225	0.340	0.271
Relaxed - Combined	**0.398**	**0.284**	**0.331**	**0.394**	**0.596**	**0.474**
RAG First Third VKB						
Exact	0.043	0.027	0.033	0.024	0.027	0.025
Partial	0.102	0.064	0.079	0.103	0.120	0.111
Relaxed - Family	0.277	0.169	0.210	0.299	0.350	0.322
Relaxed - Causal	0.180	0.112	0.138	0.220	0.257	0.237
Relaxed - Categorical	0.215	0.133	0.164	0.234	0.273	0.252
Relaxed - Combined	**0.388**	**0.238**	**0.295**	**0.439**	**0.514**	**0.473**
RAG First and Second Third VKB						
Exact	0.097	0.063	0.077	0.051	0.064	0.057
Partial	0.131	0.085	0.103	0.140	0.177	0.157
Relaxed - Family	0.219	0.143	0.173	0.281	0.355	0.313
Relaxed - Causal	0.188	0.122	0.148	0.265	0.338	0.295
Relaxed - Categorical	0.195	0.127	0.154	0.255	0.323	0.285
Relaxed - Combined	**0.359**	**0.233**	**0.283**	**0.421**	**0.532**	**0.470**
RAG Full VKB						
Exact	0.110	0.074	0.088	0.046	0.058	0.051
Partial	0.141	0.095	0.113	0.130	0.163	0.145
Relaxed - Family	0.246	0.164	0.196	0.281	0.353	0.313
Relaxed - Causal	0.215	0.143	0.171	0.261	0.327	0.290
Relaxed - Categorical	0.198	0.132	0.158	0.243	0.305	0.270
Relaxed - Combined	**0.396**	**0.264**	**0.317**	**0.441**	**0.553**	**0.490**

Table 28: Results per-sample CWE classification for GPT-OSS 120B with RAG disabled and RAG enabled using the first third of the VKB, the first and second third of the VKB, and the full VKB. All configurations are tested with reasoning disabled and reasoning enabled. Results indicate that VKB data availability has a small and inconsistent effect on per-sample CWE classification. Results are averaged over three repetitions.

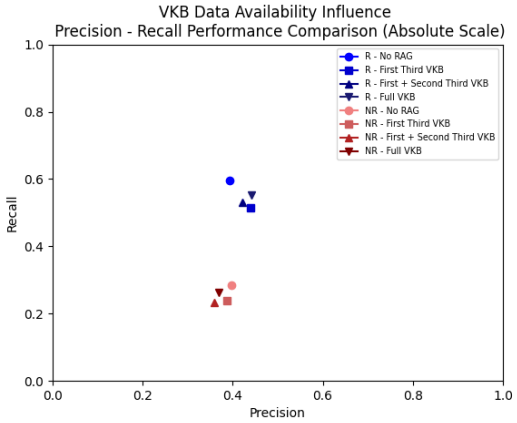


Figure 14: Per-sample CWE classification absolute precision-recall performance comparison. Red markers represent internal reasoning disabled, blue markers internal reasoning enabled. Represents the Relaxed - Combined precision and recall scores.

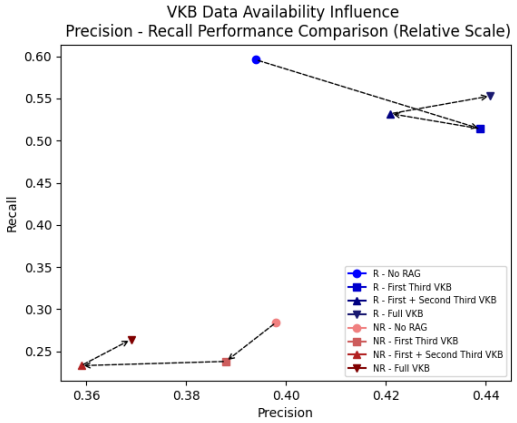


Figure 15: Per-sample CWE classification relative precision-recall performance comparison. Red markers represent internal reasoning disabled, blue markers internal reasoning enabled. Represents the Relaxed - Combined precision and recall scores.

using RAG with the full VKB. The recall is higher when the first and second third of the VKB are available than when the full VKB is available. This suggests that increasing the VKB does not consistently improve the usefulness of the retrieved VKB entries in the final vulnerability detection and classification. Using the full VKB in comparison to not using RAG leads to a slightly higher precision, but a lower recall.

Per-sample CWE classification. The results of per-sample CWE classification in Table 28, visualized in Figure 14 (absolute scale) and Figure 15 (relative scale), show that the VKB data availability does not strongly affect the exact and partial CWE labeling. We observe a larger influence in the prediction of semantically aligning CWE labels. When reasoning is disabled, not using RAG outperforms using RAG; both precision and recall are lower with the different VKB data availability configurations. Focusing on the difference between these configurations, both the precision and recall slightly fluctuate; using the first third of the data in the VKB leads to a slightly higher precision and recall than using the first and second third of the data, while using the full VKB leads to the highest precision and recall among the RAG configurations. This suggests that temporally incrementing the VKB does not have a strong influence on the relevance of the retrieved VKB entries. When reasoning is enabled, we observe a different effect: precision when using RAG is higher than when not using RAG. Between the RAG configurations, the precision fluctuates similarly as when reasoning is disabled. However, the recall slightly increases for every configuration that increments the VKB with more recent data. This leads to the highest precision of 0.441 (a relative improvement of 11.9% compared to the precision when RAG is disabled) and the highest F1 score of 0.490 (a relative improvement of 3.4% when RAG is disabled) when RAG is enabled, and the vulnerability detection pipeline has access to the full VKB. Possible reasons for this include: 1) recent vulnerabilities are more relevant for the experimental dataset code snippets, or 2) a higher quantity of data increases the chance of more relevant VKB entries.

Per-Label CWE Classification	Reasoning Disabled			Reasoning Enabled		
	P	R	F1	P	R	F1
No RAG						
Exact	0.088	0.073	0.080	0.067	0.157	0.094
Relaxed - Family	0.122	0.101	0.110	0.091	0.213	0.127
Relaxed - Causal	0.108	0.089	0.097	0.089	0.209	0.125
Relaxed - Categorical	0.097	0.080	0.088	0.087	0.205	0.122
Relaxed - Combined	**0.146**	**0.121**	**0.132**	**0.127**	**0.297**	**0.178**
RAG First Third VKB						
Exact	0.083	0.048	0.061	0.067	0.091	0.077
Relaxed - Family	0.153	0.088	0.111	0.125	0.169	0.144
Relaxed - Causal	0.111	0.064	0.081	0.091	0.124	0.105
Relaxed - Categorical	0.083	0.048	0.061	0.089	0.119	0.102
Relaxed - Combined	**0.181**	**0.104**	**0.132**	**0.171**	**0.230**	**0.196**
RAG First and Second Third VKB						
Exact	0.107	0.064	0.080	0.088	0.134	0.106
Relaxed - Family	0.148	0.088	0.110	0.126	0.190	0.151
Relaxed - Causal	0.140	0.085	0.105	0.113	0.170	0.136
Relaxed - Categorical	0.120	0.072	0.090	0.126	0.190	0.152
Relaxed - Combined	**0.194**	**0.116**	**0.145**	**0.185**	**0.279**	**0.223**
RAG Full VKB						
Exact	0.109	0.072	0.086	0.082	0.123	0.099
Relaxed - Family	0.134	0.087	0.106	0.119	0.178	0.143
Relaxed - Causal	0.129	0.084	0.101	0.114	0.171	0.137
Relaxed - Categorical	0.130	0.084	0.102	0.127	0.190	0.152
Relaxed - Combined	**0.174**	**0.111**	**0.136**	**0.188**	**0.282**	**0.225**

Table 29: Results per-label CWE classification for GPT-OSS 120B with RAG disabled, RAG with the first third of the VKB, RAG with the first and second third of the VKB, and RAG with the full VKB. All configurations are tested with reasoning disabled and reasoning enabled. Results indicate that VKB data availability has a small and inconsistent effect on per-label CWE classification. Results are averaged over three repetitions.

Per-label CWE classification. The results of per-label CWE classification in Table 29 show that the precision of the prediction of individual CWE labels increases when RAG is enabled, both when reasoning is disabled and when reasoning is enabled. This indicates that when RAG is enabled, more semantically aligned CWE labels are predicted. The recall decreases with both reasoning disabled and reasoning enabled compared to not using RAG. When reasoning is disabled, both precision and recall fluctuate, resulting in the F1 score of 0.145 for RAG when using the first two-thirds of the VKB, being the highest. When reasoning is enabled, precision and recall both consistently but slightly increase. This leads to the highest F1 score of 0.225 for RAG when using the full VKB. Both of these highest F1 scores outperform those of not using RAG for the respective reasoning mode. This suggests that RAG slightly improves CWE label classification in general, but that the availability of more recent data does not have a consistent effect on the quality of the classification.

Overall, we observe a positive effect on the precision of binary classification when RAG is enabled with both reasoning disabled and enabled. However, the effect of the availability of more recent VKB data is small and inconsistent. Similar small and inconsistent effects are perceived when observing the per-sample and per-label CWE classification evaluation perspectives.

5.5.3 Statistical Significance

To determine the reliability of our results, we perform McNemar's test on the results of the binary classification and per-sample CWE classification. The test is explained in Section 5.1.5. We observe in Table 30 that there is one statistically significant effect for the binary classification: when comparing the no RAG baseline to using the first third of the VKB when reasoning is enabled, the binary classification of vulnerable code snippets significantly decreases in quality.

This is consistent with the decrease in recall between these configurations observed in Table 27, which drops from 0.781 to 0.661: fewer vulnerable code snippets are correctly classified. No other consistent statistically significant results are observed. This is in line with the conclusion of our result analysis: availability of more recent vulnerability data does not consistently affect our results.

	Reasoning Disabled		**Reasoning Enabled**	
Class	Vulnerable	Safe	Vulnerable	Safe
No RAG ↔ RAG First Third VKB				
Iteration 1	$\rightarrow 1.0$	$\rightarrow 2.89 \times 10^{-1}$	$\leftarrow \mathbf{1.27 \times 10^{-2}}$	$\rightarrow \mathbf{7.39 \times 10^{-3}}$
Iteration 2	$\leftarrow \mathbf{1.95 \times 10^{-2}}$	$\leftarrow 3.44 \times 10^{-1}$	$\leftarrow \mathbf{4.02 \times 10^{-4}}$	$\rightarrow 9.23 \times 10^{-1}$
Iteration 3	$\leftarrow 1.0$	$\rightarrow 1.09 \times 10^{-1}$	$\leftarrow \mathbf{2.35 \times 10^{-3}}$	$\rightarrow 7.91 \times 10^{-1}$
RAG First Third VKB ↔ RAG First and Second Third VKB				
Iteration 1	$\leftarrow 4.81 \times 10^{-1}$	$\leftarrow 2.10 \times 10^{-1}$	$\rightarrow 1.0$	$\leftarrow 5.81 \times 10^{-1}$
Iteration 2	$\rightarrow 7.03 \times 10^{-2}$	$\rightarrow 1.0$	$\rightarrow 3.88 \times 10^{-1}$	$\rightarrow 1.0$
Iteration 3	$\leftarrow 3.88 \times 10^{-1}$	$\leftarrow 1.46 \times 10^{-1}$	$\rightarrow 9.23 \times 10^{-2}$	$\rightarrow 7.91 \times 10^{-1}$
RAG First and Second Third VKB ↔ RAG Full VKB				
Iteration 1	$\rightarrow 7.68 \times 10^{-2}$	$\rightarrow 7.74 \times 10^{-1}$	$\rightarrow 7.74 \times 10^{-1}$	$\leftarrow 7.91 \times 10^{-1}$
Iteration 2	$\rightarrow 1.0$	$\rightarrow 4.24 \times 10^{-1}$	$\rightarrow 1.0$	$\leftarrow 3.02 \times 10^{-1}$
Iteration 3	$\rightarrow 1.0$	$\rightarrow 7.54 \times 10^{-1}$	$\leftarrow 6.07 \times 10^{-1}$	$\rightarrow 7.74 \times 10^{-1}$

Table 30: P-values for the statistical tests performed on binary classification to determine statistically significant differences between several RAG configurations in experiment 4. The significance threshold is set to $\alpha = 0.05$. The P-values are given per class, per iteration, and statistical significance is determined after performing Holm-Bonferroni correction. Bold values are statistically significant. Three comparisons between RAG configurations are evaluated. Results show that for all RAG configurations, the only consistent statistically significant result is the difference between RAG disabled and using the first third of the VKB when internal reasoning is enabled. Not using RAG leads to significantly improved vulnerable code classification. The left arrow (←) indicates that the first configuration in the corresponding comparison performs better, while the right arrow (→) indicates that the second configuration in the corresponding comparison performs better.

	Reasoning Disabled	**Reasoning Enabled**
Class	Vulnerable	Vulnerable
No RAG ↔ RAG First Third VKB		
Iteration 1	$\rightarrow 1.0$	$\leftarrow 2.67 \times 10^{-1}$
Iteration 2	$\leftarrow 3.75 \times 10^{-1}$	$\leftarrow 5.08 \times 10^{-1}$
Iteration 3	$\leftarrow 6.88 \times 10^{-1}$	$\leftarrow 1.0$
RAG First Third VKB ↔ RAG First and Second Third VKB		
Iteration 1	$\leftarrow 1.0$	$\leftarrow 7.74 \times 10^{-1}$
Iteration 2	$\leftarrow 1.0 \times 10^{-1}$	$\rightarrow 4.53 \times 10^{-1}$
Iteration 3	$\rightarrow 1.0$	$\rightarrow 1.0$
RAG First and Second Third VKB ↔ RAG Full VKB		
Iteration 1	$\rightarrow 1.0$	$\rightarrow 5.08 \times 10^{-1}$
Iteration 2	$\rightarrow 6.25 \times 10^{-1}$	$\leftarrow 6.25 \times 10^{-2}$
Iteration 3	$\leftarrow 2.50 \times 10^{-1}$	$\leftarrow 2.19 \times 10^{-1}$

Table 31: P-values for the statistical tests performed on per-sample CWE classification to determine statistically significant differences between several RAG configurations in experiment 4. The significance threshold is set to $\alpha = 0.05$. The P-values are given per class and per iteration. Bold values are statistically significant. Three comparisons between RAG configurations are evaluated. Results show that no RAG configurations lead to significant differences. The left arrow ($\leftarrow$) indicates that the first configuration in the corresponding comparison performs better, while the right arrow ($\rightarrow$) indicates that the second configuration in the corresponding comparison performs better.

Table 31 shows that no statistically significant results are observed. This suggests that the significant difference between not using RAG and using RAG with the first third of the VKB when reasoning is enabled is not necessarily caused by a decrease in CWE classification quality.

5.5.4 Costs

Table 32 presents the total costs per component for experiment 4. The code description remains consistent across all vulnerability detection configurations per iteration. The LLM re-ranking is performed separately for each configuration, but stays consistent for the two reasoning modes. The LLM re-ranker component and vulnerability detection component in the table represent the total cost per configuration tested. Table 33 shows a breakdown of the cost of the LLM re-ranking and vulnerability detection components per configuration. We observe that providing the vulnerability classification component with retrieved VKB entries increases the costs. This is especially visible when reasoning is enabled: in total, \$0.20-\$0.22 more is used in this component.

The effects of the different sizes of the VKB do not consistently affect the costs of the LLM re-ranker, and the small differences may be caused by the LLMs' non-deterministic token usage and output generation.

Component Costs	Costs
Code Description	$1.14
LLM Re-ranking	$7.45
Vulnerability Detection	$5.89
Total	**$14.48**

Table 32: Total cost of every LLM-based component in the vulnerability detection pipeline for experiment 4.

Configuration Costs	**LLM Re-ranking**	**Vulnerability Classification**	
		Costs Reasoning Disabled	Costs Reasoning Enabled
No RAG	$0	$0.51	$0.75
RAG First Third VKB	$2.53	$0.56	$0.95
RAG First and Second Third VKB	$2.52	$0.58	$0.97
RAG Full VKB	$2.40	$0.60	$0.97
Total	**$7.45**	**$2.25**	**$3.64**

Table 33: Total cost per configuration for the vulnerability classification for experiment 4.

6 Discussion

In this chapter, we discuss the results we obtained with our experiments and use them to answer our research questions. Additionally, we describe limitations and potential threats to validity applicable to our research.

6.1 Discussion of Results

In this section, we discuss the results from our experiments, and use them to answer the sub-research questions. We discuss the effects of internal reasoning and of RAG.

6.1.1 Reasoning Influence

Experiment 1, explained and analyzed in Section 5.2, presents the effects of internal reasoning on the final vulnerability detection and classification within our constructed pipeline. We use these results to answer **SRQ1**: "How does internal reasoning in LLMs, in combination with a RAG pipeline, influence vulnerability detection and classification?".

We observe for binary classification that for both GPT-OSS 120B and Gemini 2.5 Flash-Lite, precision decreases slightly and recall increases substantially. The influence of reasoning on binary classification is shown to be statistically significant using McNemar's test on these two models: enabling reasoning increases the number of correctly predicted vulnerable code snippets (and thus increases recall), but decreases the number of correct safe predictions (and thus decreases precision). A potential reason for this is that internal reasoning leads to overthinking: for a safe method in code, it may hallucinate certain weaknesses. This reason is consistent with the non-significant results on per-sample CWE classification for these two models, indicating that there is no improvement in CWE classification. Gemma 4 26B A4B shows a different result: both precision and recall decrease slightly when internal reasoning is enabled. The model's recall without internal reasoning is already relatively high, which is hurt by the potential overthinking that a model could do with internal reasoning. This corresponds to the results of the statistical test, which shows that neither the binary classification nor the per-sample CWE classification is significantly influenced by enabling reasoning. Thus, this suggests that internal reasoning has a model-dependent effect, which is dependent on the initial performance of the model without internal reasoning enabled and may not consistently be the result of improved CWE classification.

Per-sample CWE classification and per-label CWE classification both show slight fluctuations in precision when internal reasoning is enabled; they may be improved or decreased slightly. However, increases in recall across all models for both evaluation perspectives are observed. This is, nevertheless, not a reliable result, as the statistical tests for per-sample CWE classification show no significant differences in per-sample CWE classification for the vulnerable code snippets when reasoning is enabled.

To answer **SRQ1**, focusing on the effects of internal reasoning on vulnerability detection and classification using RAG, we observe an overall slight decrease in precision, and a model-dependent effect on the recall. The differences in the results of binary classification when reasoning is enabled are statistically significant for both GPT-OSS 120B and Gemini 2.5 Flash-Lite, but not for Gemma 4 26B A4B. However, this effect is likely not caused by overall

improved CWE classification, since there is no consistent statistically significant result in the per-sample CWE classification. This suggests that internal reasoning primarily affects binary classification rather than fine-grained CWE labeling quality.

6.1.2 RAG Influence

Experiments 2, 3, and 4 analyze different effects of the RAG pipeline responsible for retrieving relevant CVEs from the VKB. We cover the effects of the components retrieved from the VKB, the influence of the LLM re-ranker, and the influence of the temporal availability of data. For these experiments, GPT-OSS 120B is the LLM used for the final vulnerability classification.

Effect of VKB Data Type Retrieval Experiment 2, explained and analyzed in Section 5.3, aims to determine the effects of retrieving different data types from the VKB. We use the results to answer **SRQ2**: "What elements from the vulnerability knowledge base lead to the highest quality vulnerability detection and classification?". Under the performed McNemar's tests, no statistically significant results are observed. Therefore, this discussion is indicative rather than based on reliable results.

An initial observation that is made is that the usage of RAG has a possible tendency to increase the F1 score over not using RAG, both with and without internal reasoning. However, these potentially positive influences are solely observed when knowledge, CVE data, and CWE data are retrieved from the VKB. When code differences of the relevant entries are also retrieved, the F1 scores decrease. A possible explanation for this is that the LLM focuses on the code patterns that cause a weakness that is observed in the code differences. Therefore, to classify the judged code snippet as a certain weakness, the LLM may need to see that exact, or a similar, pattern. When code differences are not included, the LLM may more freely classify this weakness.

The results of CWE classification in both evaluation perspectives support this explanation; both show that the addition of code differences slightly lowers the precision and recall scores. When internal reasoning is disabled, retrieving knowledge, CVE data, and CWE data shows a tendency to increase the precision and recall compared to when RAG is disabled. When internal reasoning is enabled, solely retrieving knowledge and CVE data, as well as the addition of CWE data, indicate a tendency to improve precision but slightly lower recall. Nevertheless, our experiments still show a slight increase in F1 scores in these cases.

To answer **SRQ2**, the observed patterns show a tendency to increase vulnerability detection and classification quality when RAG is enabled, and knowledge, CVE data, and CWE data are retrieved. Additionally, the retrieval of code differences generally reduces these improvements, or tends to lower the precision and recall overall compared to the no RAG baseline. However, these results are not supported by significant evidence.

Effect of LLM Re-ranker Experiment 3, explained and analyzed in Section 5.4, aims to determine the effect of the LLM re-ranker. We use the results to answer **SRQ3**: "How does an LLM re-ranker affect the retrieval process of RAG for vulnerability detection and classification?". Under the performed McNemar's tests to determine statistical significance, no statistically significant results are observed. Therefore, this discussion is solely indicative.

Our results on binary classification show that the influence of RAG with an LLM re-ranker differs when internal reasoning is enabled for the model. When reasoning is disabled, we observe a slight positive effect on precision when RAG is used with BM25 as a baseline retriever. However, when we compare this to RAG with both BM25 and an LLM re-ranker, there is an additional increase in recall, albeit it is not statistically significant. When internal reasoning is enabled, we observe a different result. For both RAG with the BM25 baseline retriever and BM25 with an LLM re-ranker, the precision stays relatively consistent, but the recall drops. This non-significant negative effect, when internal reasoning is enabled, is potentially caused by the LLM overthinking the present weaknesses that are retrieved for the LLM through the RAG pipeline. When internal reasoning is disabled, this risk is smaller, since the model is less likely to overthink.

Per-sample CWE classification follows the trends of binary classification. However, the results observed for per-label CWE classification show improved scores in fine-grained CWE labeling, although these results are also not significant. We observe that precision scores for both internal reasoning disabled and enabled increase with the usage of RAG with the BM25 baseline retriever, and increase even further when an LLM re-ranker is present.

To answer **SRQ3**, the observed results show tendencies towards improved binary classification quality when RAG with BM25 and an LLM re-ranker is used with internal reasoning disabled. When reasoning is enabled, a negative effect is observed instead. However, in both cases, our results indicate that fine-grained CWE label predictions improve in quality when RAG is used with BM25, and that the LLM re-ranker improves the detection and classification quality further. This could indicate that semantically more relevant CWE labels in the few-shot prompt result in improved prediction of CWE labels. However, no statistical evidence is found for this.

Effect of VKB Data Availability Experiment 4, explained and analyzed in Section 5.5, aims to determine the effect of VKB data availability. We use the results to answer **SRQ4**: "How does vulnerability knowledge base data availability for the RAG pipeline influence vulnerability detection and classification?".

Our results on the binary classification evaluation perspective show that when internal reasoning is disabled, not using RAG compared to using only the first third of the VKB leads to a decrease in the F1 score. When incrementally adding more thirds of the VKB, both the precision score and recall score slightly increase. The full VKB scores approximately the same F1 score as when no RAG is used. This indicates that increasing the VKB does not consistently improve vulnerability detection. This is in line with the lack of statistically significant results observed when internal reasoning is disabled. However, when internal reasoning is enabled, a stronger effect is observed. When comparing not using RAG to using RAG with the first third of the VKB, there is a strong decrease in recall. Our statistical tests show that this decrease in vulnerability detection is consistently statistically significant across the three performed iterations. Incrementing the VKB with the second third, and last third, we observe increases in both precision and recall, but none of the RAG configurations reach the same performance as the LLM reaches when RAG is disabled. However, the effects of these increments are not statistically significant.

The results of per-sample CWE classification show that CWE classification is not substantially impacted by the introduction of new VKB data when internal reasoning is disabled. We observe that there is a slight drop in recall when comparing the no RAG baseline to RAG using the first third of the VKB. After incrementing the VKB with more data, inconsistent fluctuations are observed, leading to a slightly lower F1 score for RAG with the full VKB compared to the no RAG baseline. When internal reasoning is enabled, we observe a similar result, albeit that the precision score increases more strongly when the first third of the VKB is available compared to the no RAG baseline. This leads to fewer fluctuations in the F1 scores overall. When using the full VKB, a slightly higher F1 score is observed compared to the no RAG baseline. However, none of these results are statistically significant. Similar patterns emerge for per-label CWE classification. Though when internal reasoning is enabled, the increase in precision when incrementing the VKB with more data leads to a slightly larger increase in F1 score.

The inconsistent and non-significant effects are potentially caused by the type of vulnerabilities on which we perform our experiments. A substantial part of the found C/C++ vulnerabilities are related to memory safety issues. Consequently, this class of CWEs also dominates the VKB, as is shown in Section 4.4.4. Therefore, the new vulnerabilities that are retrieved in new increments of the VKB are often of the same CWE types. Subsequently, these do not target different or newer underlying weaknesses, potentially causing the inconsistent effects that we observe.

To answer **SRQ4**, the observed results show mostly inconsistent and non-significant effects on vulnerability detection and classification. The only consistent statistically significant effect is found when internal reasoning is enabled, and we compare the no RAG baseline to using the first third of the VKB. However, this is unlikely due to decreased CWE classification performance, since these results are not statistically significant. Overall, these findings suggest that increasing the VKB with new vulnerability data has a limited effect on the vulnerability detection and classification of C/C++ vulnerabilities.

6.2 Discussion of RQ

Using the answers to the SRQs discussed in the previous section, we can answer the main research question (**RQ**) of this thesis: "How do internal reasoning and RAG affect vulnerability detection and classification performance of LLMs?"

We observe in our experiments that internal reasoning has a model-dependent, potentially statistically significant effect on vulnerability detection and classification. More specifically, vulnerable code snippets are more often correctly predicted when internal reasoning is enabled, and safe code snippets less often. Nevertheless, no statistical evidence is found that these effects are caused by an increased or decreased performance in CWE classification.

Additionally, the implementation of a RAG pipeline shows inconsistent, non-significant results. However, we do cautiously observe slight improvements in both binary classification and CWE classification when solely retrieving knowledge, CVE data, and CWE data, without retrieving code differences, both when internal reasoning is enabled and disabled. Our results further indicate that using an LLM re-ranker has a reasoning-dependent effect on vulnerability detection and classification. When internal reasoning is disabled, our results show stronger increases in

precision and overall increases in recall, whereas when internal reasoning is enabled, precision increases only slightly, and recall decreases. Finally, we have observed that data availability in the VKB has inconsistent effects, indicating that incrementing the VKB with more recent data does not necessarily improve vulnerability detection and classification.

Overall, we can therefore state that internal reasoning and RAG both have distinct effects, albeit that for most of the effects of RAG, we have only observed non-significant patterns. However, we also found that these two adaptation techniques influence each other, leading to further increases or decreases in some observed effects when combined.

These findings are partially consistent with Vul-RAG [30], which also uses knowledge-level RAG. Vul-RAG shows that the accuracy, balanced precision, and balanced recall increase when using RAG compared to the base LLMs. Although the experimental evaluation differs substantially, the findings are partially consistent with our observations in the third experiment. Here, we find that using RAG with BM25 and an LLM re-ranker leads to increases in F1 score over not using RAG, both when internal reasoning is disabled and enabled. However, these improvements were not statistically significant. Additionally, the effects of internal reasoning are partially consistent with the findings of the effects of CoT in Vul-RAG. The authors of Vul-RAG find that the use of CoT brings limited improvements, dependent on the model used. This fits the model-dependent effects that we observe in experiment 1, although the effects of internal reasoning on precision and recall in this experiment are larger than those reported for CoT in Vul-RAG.

The large variability between the different models also impacts the generalizability of our results; our experiments investigating the effects of RAG are solely performed with GPT-OSS 120B, although other models could lead to substantially different results. Consequently, identifying general patterns in incorrectly classified vulnerabilities, and the effects of internal reasoning and RAG on these patterns requires future research. Nonetheless, the rigorous design of `VulnReason-RAG` allows us to systematically examine these effects on the selected models. Additionally, `VulnReason-RAG` is open-sourced to facilitate extensions and further experimentation in future research.

6.3 Limitations and Threats to Validity

In this section, we examine the limitations and possible threats to validity that are encountered in this thesis. We discuss the risks and, where applicable, the mitigations we take to minimize these risks.

Prompt Optimization LLMs are sensitive to small prompt differences [62]. Therefore, slight word changes and formatting choices can have an impact on the quality of vulnerability detection and classification, and at the time of writing, we cannot state that we use optimal prompts. To limit the influence of prompt sensitivity, we base the prompts that we use on a prompt pattern framework, explained in Section 2.2.3. Additionally, for all experiments, we use the same prompts. Therefore, differences in performance are not a consequence of different prompts, but rather isolated effects of the differences in the tested configurations.

Deprecated CWEs We construct a VKB with real CVEs identified since 1999. However, some of these CVEs have CWEs assigned that are deprecated or no longer assignable due to other reasons. Simsek et al. have found that 55% of CVEs up until December 2024 are mapped to invalid CWEs (i.e., prohibited, discouraged, or placeholder CWEs) [98]. Consequently, some of the CWEs retrieved from the VKB are no longer allowed to be assigned. Nevertheless, we expect that the LLMs we use are trained on recent versions of the CWE List, and therefore will not predict invalid CWEs, even if they are provided through the RAG pipeline.

Dataset Size Our experimental dataset consists of 128 code snippets. 64 of these code snippets are vulnerable, and the other 64 are safe. In this dataset, the quality of the data is prioritized. As explained in Section 4.4.1, we filter the available data from the MoreFixes database on several constraints to ensure that the entire vulnerability is encapsulated within a single code snippet, and that no LLM contamination takes place. However, we acknowledge that the small size of the dataset partially contributes to the small number of statistically significant results we obtain in this thesis.

LLM Non-Determinism We observed several inconsistent results within experiments and inter-experimental variability between the results of different experiments. These inconsistencies are partially due to the non-deterministic nature of LLMs. As explained in Section 2.2.1, LLMs generate text by predicting tokens. Predictions are based on probabilities, and consequently, the same prompt can lead to different outputs. Therefore, the same experimental configuration executed several times can lead to different detection and classification results. To mitigate this from affecting our results substantially, we perform three repetitions of all configurations in all our experiments. We average the results over these three repetitions.

Retrieval Quality Our results show inconsistent, non-significant behavior between different RAG configurations. Besides the non-deterministic nature of LLMs, an additional potential reason for this inconsistent behavior may be the quality of the retrieved entries from the VKB. As discussed in Section 4.4.4, the constructed VKB contains 1942 CVEs. However, it is not certain if for every experimental code snippet, a relevant CVE is stored in the VKB, or if the most relevant CVE is correctly retrieved.

Complex Vulnerabilities To ensure that all vulnerabilities in the VKB and all vulnerable code snippets in our experimental dataset are fully captured by the LLM when tested or used, we determine suitable vulnerabilities to have exactly one file change. Consequently, more complex vulnerabilities, e.g., vulnerabilities that span across several files and programming languages, are not covered in this thesis. The configurations experimented with in our research may behave differently when more complex vulnerabilities are tested. Future research is required to investigate the effects of `VulnReason-RAG` on more complex vulnerabilities.

Safe Code Snippets The safe code snippets in the experimental dataset used to evaluate `VulnReason-RAG` consist of code snippets in which vulnerabilities were previously present, but are now mitigated. Therefore, we assume that these code snippets are free of the target vulnerabilities. Nevertheless, we cannot guarantee these code snippets to be fully free of other vulnerabilities, as other potentially undiscovered vulnerabilities may still be present.

Budget Limitations Due to budget limitations and the complexity of the `VulnReason-RAG` pipeline, performing more extensive incremental ablation studies was beyond the scope of this thesis. Instead, the experiments were designed to maximize the reliability of the findings. Therefore, we selected cost-efficient LLMs for our experiments, with which we were able to perform three iterations per experiment. Additionally, we conducted statistical tests to further strengthen confidence that our results are not due to random chance.

7 Conclusion

In this thesis, we investigate the influence of LLMs' internal reasoning and RAG on vulnerability detection and classification. We develop `VulnReason-RAG`, a tool with two pipelines to construct a vulnerability knowledge base, and to detect and classify vulnerabilities in an experimental dataset of code snippets. We experiment with models that have an internal reasoning toggle to allow fair comparison between disabling and enabling internal reasoning. Our RAG process is constructed as a two-stage retrieval pipeline using exact-term matching and semantic matching. Using the results of the performed experiments, we determine the influence of internal reasoning and RAG, both separately and combined. We evaluate the performance of `VulnReason-RAG` on both binary classification and CWE classification, per sample and per label. This gives us a detailed overview of the vulnerability detection and classification performance by enabling comparison between the correctness of binary predictions and fine-grained, semantically aligned CWE label predictions. Additionally, to determine the reliability of our results, we perform McNemar's test to test for statistical significance.

We show that internal reasoning has a model-dependent effect on vulnerability detection and classification; some models predict significantly more vulnerable code snippets correctly, but also predict significantly fewer safe code snippets correctly when internal reasoning is enabled. Other models may show inconsistent effects. However, we find no statistical evidence that CWE classification is improved or decreased when internal reasoning is enabled.

We observe that RAG has a mostly inconsistent effect on vulnerability detection and classification. Nevertheless, we cautiously observe beneficial, but non-significant, configurations that can be used. First, we find that the retrieval of knowledge, CVE data, and CWE data leads to slight improvements in vulnerability detection and classification, both when internal reasoning is disabled and enabled, compared to not using RAG. However, retrieving code differences diminishes these improvements, highlighting the influence that the type of data that is retrieved has on the RAG process. Second, we show that adding an LLM re-ranker to the RAG process has a non-significant positive influence on vulnerability detection and classification when internal reasoning is disabled. When internal reasoning is enabled, these influences are weaker and mostly negative, suggesting that the effect of the LLM re-ranker on the vulnerability detection and classification process is influenced by the internal reasoning of the vulnerability detector. Lastly, we observe that the addition of more recent vulnerabilities in the VKB does not necessarily consistently lead to improved vulnerability detection and classification.

With these results, we can answer our **RQ**: "How do internal reasoning and RAG affect vulnerability detection and classification performance of LLMs?" We observe that internal reasoning leads to an increase in code snippets that are predicted to be vulnerable; the LLMs are more likely to determine code to be vulnerable. RAG provides the LLM with relevant vulnerabilities, which in some cases can positively influence the detection and classification process, but at other times can negatively influence the thought process of the LLM. Especially when internal reasoning is enabled, this can lead to inconsistent effects.

`VulnReason-RAG` is released as open source [44], allowing for several extensions and modifications to be made to the pipeline. Future research on `VulnReason-RAG` could inspect the

functionality of flagship LLM models, instead of the cost-efficient models that we use in this research. This also supports extending the analysis of the generalizability of the effects of internal reasoning and RAG on vulnerability detection. Furthermore, future research could perform more extensive (incremental) ablation studies to further investigate the effects of the distinct components of `VulnReason-RAG`. Additionally, `VulnReason-RAG` could be expanded to support more complex vulnerabilities, e.g., vulnerabilities that cover several files, and vulnerabilities in different programming languages. The former would also allow for more CVEs to be suitable for both the VKB and the experimental dataset, without an increased risk of LLM contamination. The latter supports further generalization of the effects of internal reasoning and RAG on broader vulnerability detection, rather than only on C/C++ vulnerabilities. Other methods of retrieval could also be researched. Currently, an LLM re-ranker, which re-ranks VKB entries based on semantic overlap in knowledge and code descriptions, does not show to be statistically more beneficial than not using RAG. However, using a different LLM (re-)ranking approach, such as embedding-based retrieval, could potentially affect vulnerability detection and classification differently.

References

[1] *1.3.6.6.18. Binomial distribution*. URL: https://www.itl.nist.gov/div898/handbook/eda/section3/eda366i.htm.

[2] Jafar Akhoundali et al. "MoreFixes: A Large-Scale Dataset of CVE Fix Commits Mined through Enhanced Repository Discovery". In: *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2024. Porto de Galinhas, Brazil: Association for Computing Machinery, 2024, pp. 42–51. ISBN: 9798400706752. DOI: 10.1145/3663533.3664036. URL: https://doi.org/10.1145/3663533.3664036.

[3] Jafar Akhoundali et al. *MoreFixes: Largest CVE dataset with fixes*. Zenodo, Sept. 2024. DOI: 10.5281/zenodo.13983082. URL: https://doi.org/10.5281/zenodo.13983082.

[4] Sotiris Anagnostidis and Jannis Bulian. *How Susceptible are LLMs to Influence in Prompts?* 2024. arXiv: 2408.11865 [cs.CL]. URL: https://arxiv.org/abs/2408.11865.

[5] Jaehyeon Bae, Seoryeong Kwon, and Seunghwan Myeong. "Enhancing Software Code Vulnerability Detection Using GPT-4o and Claude-3.5 Sonnet: A Study on Prompt Engineering Techniques". In: *Electronics* 13.13 (2024). ISSN: 2079-9292. DOI: 10.3390/electronics13132657. URL: https://www.mdpi.com/2079-9292/13/13/2657.

[6] Ivan Belcic and Cole Stryker. "LLM Parameters". In: (Nov. 2025). URL: https://www.ibm.com/think/topics/llm-parameters.

[7] Dave Bergmann and Cole Stryker. *Attention mechanism*. Nov. 2025. URL: https://www.ibm.com/think/topics/attention-mechanism.

[8] Guru Bhandari, Amara Naseer, and Leon Moonen. "CVEfixes: automated collection of vulnerabilities and their fixes from open-source software". In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 30–39. ISBN: 9781450386807. DOI: 10.1145/3475960.3475985. URL: https://doi.org/10.1145/3475960.3475985.

[9] Paul Black. *SARD: A Software Assurance Reference Dataset*. en. 1970.

[10] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. *Fuzzing: Challenges and Reflections*. Tech. rep. 3. 2021, pp. 79–80. URL: https://abhikrc.com/pdf/IEEE-SW-Fuzzing.pdf.

[11] Tim Boland and Paul E. Black. "Juliet 1.1 C/C++ and Java Test Suite". In: *Computer* 45.10 (Oct. 2012), pp. 88–90. DOI: 10.1109/mc.2012.345. URL: https://doi.org/10.1109/mc.2012.345.

[12] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. San Diego, CA: USENIX Association, Dec. 2008. URL: https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems.

[14] Daipeng Cao and W. Jun. "LLM-CloudSec: Large Language Model Empowered Automatic and Deep Vulnerability Analysis for Intelligent Clouds". In: *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2024, pp. 1–6. DOI: 10.1109/INFOCOMWKSHPS61880.2024.10620804.

[15] Ira Ceka et al. *Can LLM Prompting Serve as a Proxy for Static Analysis in Vulnerability Detection*. 2025. arXiv: 2412.12039 [cs.CR]. URL: https://arxiv.org/abs/2412.12039.

[16] Saikat Chakraborty et al. "Deep Learning Based Vulnerability Detection: Are We There Yet?" In: *IEEE Transactions on Software Engineering* 48.9 (2022), pp. 3280–3296. DOI: 10.1109/TSE.2021.3087402.

[17] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: (2021). arXiv: 2107.03374 [cs.LG].

[18] Yinan Chen et al. *GPTVD: Vulnerability Detection and Analysis Method Based on LLM's Chain of Thoughts*. Mar. 2025. DOI: 10.21203/rs.3.rs-6275325/v1.

[19] Yizheng Chen et al. "DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection". In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID '23. Hong Kong, China: Association for Computing Machinery, 2023, pp. 654–668. ISBN: 9798400707650. DOI: 10.1145/3607199.3607242. URL: https://doi.org/10.1145/3607199.3607242.

[20] Yuxing Cheng, Yi Chang, and Yuan Wu. *A Survey on Data Contamination for Large Language Models*. 2025. arXiv: 2502.14425 [cs.CL]. URL: https://arxiv.org/abs/2502.14425.

[21] The MITRE Corporation. *CVE List Downloads*. URL: https://www.cve.org/Downloads.

[22] The MITRE Corporation. *CVE Record User Guide*. URL: https://www.cve.org/CVERecord/UserGuide/#cve-cvss.

[23] The MITRE Corporation. *Overview - About the CVE Program*. URL: https://www.cve.org/About/Overview.

[24] *CPE - Common Platform Enumeration: about CPE*. URL: https://cpe.mitre.org/about/.

[25] *CPPCheck: a tool for static C/C++ code analysis*. URL: https://cppcheck.sourceforge.io/.

[26] *CWE - New to CWE*. URL: https://cwe.mitre.org/about/new_to_cwe.html.

[27] Doaa Dalaq et al. “A Systematic Literature Review on Static Application Security Testing (SAST) Tools: Evaluation, Benchmarks, Challenges, and Future Directions”. In: *Proceedings of the 2025 29th International Conference on Evaluation and Assessment in Software Engineering Companion*. EASE Companion ’25. Association for Computing Machinery, 2025, pp. 162–168. ISBN: 9798400718328. DOI: 10.1145/3727967.3756838. URL: https://doi.org/10.1145/3727967.3756838.

[28] Yangruibo Ding et al. “Vulnerability Detection with Code Language Models: How Far are We?” In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2025, pp. 1729–1741. DOI: 10.1109/ICSE55347.2025.00038.

[29] Zican Dong et al. *Exploring Context Window of Large Language Models via Decomposed Positional Vectors*. 2024. arXiv: 2405.18009 [cs.CL]. URL: https://arxiv.org/abs/2405.18009.

[30] Xueying Du et al. *Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG*. 2025. arXiv: 2406.11147 [cs.SE]. URL: https://arxiv.org/abs/2406.11147.

[31] Allen L. Edwards. “Note on the “Correction for Continuity” in Testing the Significance of the Difference between Correlated Proportions”. In: *Psychometrika* 13.3 (Sept. 1948), pp. 185–187. DOI: 10.1007/bf02289261. URL: https://doi.org/10.1007/bf02289261.

[32] Jiahao Fan et al. “A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR ’20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 508–512. ISBN: 9781450375177. DOI: 10.1145/3379597.3387501. URL: https://doi.org/10.1145/3379597.3387501.

[33] *Find the exact p value for McNemar’s test*. URL: https://statkat.com/find-p-value/mcnemars-test-value.php.

[34] *FlawFinder home page*. URL: https://dwheeler.com/flawfinder/.

[35] Ontotext Fundamentals. *Fundamentals: What is a Knowledge Base?* Aug. 2018. URL: https://www.ontotext.com/knowledgehub/fundamentals/what-is-a-knowledge-base/.

[36] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. arXiv: 2312.10997 [cs.CL]. URL: https://arxiv.org/abs/2312.10997.

[37] *Gemini 2.5 Flash-Lite*. URL: https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-flash-lite.

[38] *Gemini 3.1 Flash-Lite Preview*. URL: https://ai.google.dev/gemini-api/docs/models/gemini-3.1-flash-lite-preview.

[39] *Gemma 4 model overview*. URL: https://ai.google.dev/gemma/docs/core.

[40] Github. *GitHub - github/advisory-database: Security vulnerability database inclusive of CVEs and GitHub originated security advisories from the world of open source software*. URL: https://github.com/github/advisory-database.

[41] *GPT-5 nano Model — OpenAI API*. URL: https://developers.openai.com/api/docs/models/gpt-5-nano.

[42] *GPT5 Nano*. URL: https://platform.openai.com/docs/models/gpt-5-nano.

[43] 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*. CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 1865–1879. ISBN: 9798400700507. DOI: 10.1145/3576915.3623175. URL: https://doi.org/10.1145/3576915.3623175.

[44] Jayme Hebinck. *VulnReason-RAG*. URL: https://github.com/JaymeHebinck/VulnReason-RAG.

[45] Sture Holm. "A Simple Sequentially Rejective Multiple Test Procedure". In: *Scandinavian Journal of Statistics* 6.2 (1979), pp. 65–70. ISSN: 03036898, 14679469. URL: http://www.jstor.org/stable/4615733 (visited on 05/26/2026).

[46] *Infer Static Analyzer — Infer — Infer*. URL: https://fbinfer.com/.

[47] Christian Janiesch, Patrick Zschech, and Kai Heinrich. "Machine learning and deep learning". In: *Electronic Markets* 31.3 (Apr. 2021), pp. 685–695. ISSN: 1422-8890. DOI: 10.1007/s12525-021-00475-2. URL: http://dx.doi.org/10.1007/s12525-021-00475-2.

[48] Bo Jiang, Ye Liu, and Wing Kwong Chan. "Contractfuzzer: Fuzzing smart contracts for vulnerability detection". In: *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 2018, pp. 259–269.

[49] Sabrina Kaniewski et al. *A Systematic Literature Review on Detecting Software Vulnerabilities with Large Language Models*. 2025. arXiv: 2507.22659 [cs.SE]. URL: https://arxiv.org/abs/2507.22659.

[50] Zixuan Ke et al. *A Survey of Frontiers in LLM Reasoning: Inference Scaling, Learning to Reason, and Agentic Systems*. 2026. arXiv: 2504.09037 [cs.AI]. URL: https://arxiv.org/abs/2504.09037.

[51] Mete Keltek et al. *Boosting Cybersecurity Vulnerability Scanning based on LLM-supported Static Application Security Testing*. 2024. arXiv: 2409.15735 [cs.CR]. URL: https://arxiv.org/abs/2409.15735.

[52] Vasileios Kouliaridis, Georgios Karopoulos, and Georgios Kambourakis. "Assessing the Effectiveness of LLMs in Android Application Vulnerability Analysis". In: *Attacks and Defenses for the Internet-of-Things*. Springer Nature Switzerland, 2025, pp. 139–154. ISBN: 9783031855931. DOI: 10.1007/978-3-031-85593-1_9. URL: http://dx.doi.org/10.1007/978-3-031-85593-1_9.

[53] Ahmed Lekssays et al. *LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models*. 2025. arXiv: 2507.16585 [cs.CR]. URL: https://arxiv.org/abs/2507.16585.

[54] Patrick Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7Paper.pdf.

[55] Yikun Li et al. *CleanVul: Automatic Function-Level Vulnerability Detection in Code Commits Using LLM Heuristics*. 2025. arXiv: 2411.17274 [cs.SE]. URL: https://arxiv.org/abs/2411.17274.

[56] Yue Li et al. *Everything You Wanted to Know About LLM-based Vulnerability Detection But Were Afraid to Ask*. 2025. arXiv: 2504.13474 [cs.CR]. URL: https://arxiv.org/abs/2504.13474.

[57] Zhen Li † et al. *VulDeePecker: A Deep Learning-Based System for Vulnerability Detection*. Tech. rep. Feb. 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_03A-2_Li_paper.pdf.

[58] Zhen Li et al. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection". In: *Proceedings 2018 Network and Distributed System Security Symposium*. NDSS 2018. Internet Society, 2018. DOI: 10.14722/ndss.2018.23158. URL: http://dx.doi.org/10.14722/ndss.2018.23158.

[59] Wang Lingxiang et al. *SAVANT: Vulnerability Detection in Application Dependencies through Semantic-Guided Reachability Analysis*. 2025. arXiv: 2506.17798 [cs.SE]. URL: https://arxiv.org/abs/2506.17798.

[60] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. "An empirical study on the effectiveness of static C code analyzers for vulnerability detection". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, pp. 544–555. ISBN: 9781450393799. DOI: 10.1145/3533767.3534380. URL: https://doi.org/10.1145/3533767.3534380.

[61] Peter J. Liu et al. *Generating Wikipedia by Summarizing Long Sequences*. 2018. arXiv: 1801.10198 [cs.CL]. URL: https://arxiv.org/abs/1801.10198.

[62] Yang Liu and Chenhui Chu. *Understanding the Prompt Sensitivity*. 2026. arXiv: 2604.18389 [cs.CL]. URL: https://arxiv.org/abs/2604.18389.

[63] Zhihong Liu et al. "Software Vulnerability Detection with GPT and In-Context Learning". In: *2023 8th International Conference on Data Science in Cyberspace (DSC)*. 2023, pp. 229–236. DOI: 10.1109/DSC59305.2023.00041.

[64] Guilong Lu et al. "GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning". In: *Journal of Systems and Software* 212 (2024), p. 112031. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2024.112031. URL: https://www.sciencedirect.com/science/article/pii/S0164121224000748.

[65] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. July 2008. URL: http://nlp.stanford.edu/IR-book/pdf/11prob.pdf.

[66] Noble Saji Mathews et al. *LLbezpeky: Leveraging Large Language Models for Vulnerability Detection*. 2024. arXiv: 2401.01269 [cs.CR]. URL: https://arxiv.org/abs/2401.01269.

[67] $maximum_matching|NetworkX3.6.1documentation$. URL: https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.bipartite.matching.maximum_matching.html.

[68] Quinn McNemar. “Note on the sampling error of the difference between correlated proportions or percentages”. In: *Psychometrika* 12.2 (June 1947), pp. 153–157. DOI: 10.1007/bf02295996. URL: https://doi.org/10.1007/bf02295996.

[69] Mend.io. *Open source vulnerability database — Mend.io*. Nov. 2025. URL: https://www.mend.io/vulnerability-database/.

[70] Richard Meyes et al. *Ablation Studies in Artificial Neural Networks*. 2019. arXiv: 1901.08644 [cs.NE]. URL: https://arxiv.org/abs/1901.08644.

[71] Barton P. Miller, Lars Fredriksen, and Bryan So. “An empirical study of the reliability of UNIX utilities”. In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[72] *moonshotai/Kimi-K2-Instruct-0905 · Hugging Face*. URL: https://huggingface.co/moonshotai/Kimi-K2-Instruct-0905.

[73] Humza Naveed et al. “A Comprehensive Overview of Large Language Models”. In: *ACM Trans. Intell. Syst. Technol.* 16.5 (Aug. 2025). ISSN: 2157-6904. DOI: 10.1145/3744746. URL: https://doi.org/10.1145/3744746.

[74] Chao Ni et al. “MegaVul: A C/C++ Vulnerability Dataset with Comprehensive Code Representations”. In: *Proceedings of the 21st International Conference on Mining Software Repositories*. MSR ’24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 738–742. ISBN: 9798400705878. DOI: 10.1145/3643991.3644886. URL: https://doi.org/10.1145/3643991.3644886.

[75] Georgios Nikitopoulos et al. “CrossVul: a cross-language vulnerability dataset with commit data”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1565–1569. ISBN: 9781450385626. DOI: 10.1145/3468264.3473122. URL: https://doi.org/10.1145/3468264.3473122.

[76] *NVD - Home*. URL: https://nvd.nist.gov/.

[77] *NVD - Search and Statistics*. URL: https://nvd.nist.gov/vuln/search#/nvd/home?resultType=statistics.

[78] OpenAI. *gpt-oss-120b gpt-oss-20b Model Card*. 2025. arXiv: 2508.10925 [cs.CL]. URL: https://arxiv.org/abs/2508.10925.

[79] *OpenRouter*. Apr. 2026. URL: https://openrouter.ai/.

[80] OpenRouter. *Parameters*. URL: https://openrouter.ai/docs/api/reference/parameters.

[81] Venkatesh Balavadhani Parthasarathy et al. *The Ultimate Guide to Fine-Tuning LLMs from Basics to Breakthroughs: An Exhaustive Review of Technologies, Research, Best Practices, Applied Research Challenges and Opportunities*. 2024. arXiv: 2408.13296 [cs.LG]. URL: https://arxiv.org/abs/2408.13296.

[82] Elijah Pelofske, Vincent Urias, and Lorie M. Liebrock. *Automated Software Vulnerability Static Code Analysis Using Generative Pre-Trained Transformer Models*. 2024. arXiv: 2408.00197 [cs.CR]. URL: https://arxiv.org/abs/2408.00197.

[83] Henning Perl et al. “VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS ’15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 426–437. ISBN: 9781450338325. DOI: 10.1145/2810103.2813604. URL: https://doi.org/10.1145/2810103.2813604.

[84] Moumita Das Purba et al. “Software Vulnerability Detection using Large Language Models”. In: *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2023, pp. 112–119. DOI: 10.1109/ISSREW60843.2023.00058.

[85] Wisam A. Qader, Musa M. Ameen, and Bilal I. Ahmed. “An Overview of Bag of Words;Importance, Implementation, Applications, and Challenges”. In: *2019 International Engineering Conference (IEC)*. 2019, pp. 200–204. DOI: 10.1109/IEC47844.2019.8950616.

[86] *Qwen3-VL - a Qwen Collection*. URL: https://huggingface.co/collections/Qwen/qwen3-vl.

[87] Alec Radford and Karthik Narasimhan. “Improving Language Understanding by Generative Pre-Training”. In: 2018. URL: https://api.semanticscholar.org/CorpusID:49313245.

[88] Anand Rajaraman and Jeffrey David Ullman. “Data Mining”. In: *Mining of Massive Datasets*. Cambridge University Press, 2011, pp. 1–17.

[89] Stephen Robertson. “Understanding Inverse Document Frequency: On Theoretical Arguments for IDF”. In: *Journal of Documentation - J DOC* 60 (Oct. 2004), pp. 503–520. DOI: 10.1108/00220410410560582.

[90] Stephen E Robertson et al. *Okapi at TREC-3*. British Library Research and Development Department, 1995.

[91] Rebecca Russell et al. “Automated Vulnerability Detection in Source Code Using Deep Representation Learning”. In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2018, pp. 757–762. DOI: 10.1109/ICMLA.2018.00120.

[92] Rebecca Russell et al. “Automated Vulnerability Detection in Source Code Using Deep Representation Learning”. In: Dec. 2018, pp. 757–762. DOI: 10.1109/ICMLA.2018.00120.

[93] Pranab Sahoo et al. *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. 2025. arXiv: 2402.07927 [cs.AI]. URL: https://arxiv.org/abs/2402.07927.

[94] Minjae Seo et al. *AutoPatch: Multi-Agent Framework for Patching Real-World CVE Vulnerabilities*. 2025. arXiv: 2505.04195 [cs.CR]. URL: https://arxiv.org/abs/2505.04195.

[95] Ze Sheng et al. *LProtector: An LLM-driven Vulnerability Detection System*. 2024. arXiv: 2411.06493 [cs.CR]. URL: https://arxiv.org/abs/2411.06493.

[96] Samiha Shimmi et al. "Software Vulnerability Detection Using LLM: Does Additional Information Help?" In: *2024 Annual Computer Security Applications Conference Workshops (ACSAC Workshops)*. 2024, pp. 216–223. DOI: 10.1109/ACSACW65225.2024.00031.

[97] Mohammed Latif Siddiq and Joanna C. S. Santos. "SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques". In: *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*. MSR4PS 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 29–33. ISBN: 9781450394574. DOI: 10.1145/3549035.3561184. URL: https://doi.org/10.1145/3549035.3561184.

[98] Sevval Simsek, Varsha Athreya, and David Starobinski. *FixV2W: Correcting Invalid CVE-CWE Mappings with Knowledge Graph Embeddings*. 2026. arXiv: 2604.22176 [cs.CR]. URL: https://arxiv.org/abs/2604.22176.

[99] Matilda Q. R. Pembury Smith and Graeme D. Ruxton. "Effective use of the McNemar test". In: *Behavioral Ecology and Sociobiology* 74.11 (Oct. 2020). DOI: 10.1007/s00265-020-02916-y. URL: https://doi.org/10.1007/s00265-020-02916-y.

[100] Evangelia Spiliopoulou et al. *Play Favorites: A Statistical Method to Measure Self-Bias in LLM-as-a-Judge*. 2025. arXiv: 2508.06709 [cs.CL]. URL: https://arxiv.org/abs/2508.06709.

[101] Yuqiang Sun et al. *LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning*. 2025. arXiv: 2401.16185 [cs.CR]. URL: https://arxiv.org/abs/2401.16185.

[102] Joshua Henrina Sundjaja, Rijen Shrestha, and Kewal Krishan. *McNemar and Mann-Whitney U tests*. July 2023. URL: https://www.ncbi.nlm.nih.gov/books/NBK560699/.

[103] Yongjian Tang et al. "The Few-shot Dilemma: Over-prompting Large Language Models". In: *2025 3rd International Conference on Foundation and Large Language Models (FLLM)*. IEEE, Nov. 2025, pp. 134–141. DOI: 10.1109/fllm67465.2025.11391015. URL: http://dx.doi.org/10.1109/FLLM67465.2025.11391015.

[104] Kimi Team et al. *Kimi K2: Open Agentic Intelligence*. 2026. arXiv: 2507.20534 [cs.LG]. URL: https://arxiv.org/abs/2507.20534.

[105] Norbert Tihanyi et al. "The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification". In: *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2023. San Francisco, CA, USA: Association for Computing Machinery, 2023, pp. 33–43. ISBN: 9798400703751. DOI: 10.1145/3617555.3617874. URL: https://doi.org/10.1145/3617555.3617874.

[106] Fouad Trad and Ali Chehab. *Retrieval-Augmented Few-Shot Prompting Versus Fine-Tuning for Code Vulnerability Detection*. 2025. arXiv: 2512.04106 [cs.SE]. URL: https://arxiv.org/abs/2512.04106.

[107] Tielei Wang et al. "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection". In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 497–512. DOI: 10.1109/SP.2010.37.

[108] Xinchen Wang et al. "ReposVul: A Repository-Level High-Quality Vulnerability Dataset". In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. ICSE-Companion '24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 472–483. ISBN: 9798400705021. DOI: 10.1145/3639478.3647634. URL: https://doi.org/10.1145/3639478.3647634.

[109] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[110] Xin-Cheng Wen et al. *VulEval: Towards Repository-Level Evaluation of Software Vulnerability Detection*. 2024. arXiv: 2404.15596 [cs.SE]. URL: https://arxiv.org/abs/2404.15596.

[111] Peter H. Westfall, James F. Troendle, and Gene Pennello. "Multiple McNemar tests". In: *Biometrics* 66.4 (Mar. 2010), pp. 1185–1191. DOI: 10.1111/j.1541-0420.2010.01408.x. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC2902578/.

[112] *What is a CVE?* URL: https://www.redhat.com/en/topics/security/what-is-cve.

[113] J White. "A prompt pattern catalog to enhance prompt engineering with ChatGPT". In: *arXiv preprint arXiv:2302.11382* (2023).

[114] Fang Wu et al. "Vulnerability detection with deep learning". In: *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. 2017, pp. 1298–1302. DOI: 10.1109/CompComm.2017.8322752.

[115] xAI. *Announcing Grok-1.5*. Mar. 2024. URL: https://x.ai/news/grok-1.5.

[116] Fengli Xu et al. *Towards Large Reasoning Models: A Survey of Reinforced Reasoning with Large Language Models*. 2025. arXiv: 2501.09686 [cs.AI]. URL: https://arxiv.org/abs/2501.09686.

[117] Yanjing Yang et al. *DLAP: A Deep Learning Augmented Large Language Model Prompting Framework for Software Vulnerability Detection*. 2024. arXiv: 2405.01202 [cs.SE]. URL: https://arxiv.org/abs/2405.01202.

[118] Kaiyan Zhang et al. *A Survey of Reinforcement Learning for Large Reasoning Models*. 2025. arXiv: 2509.08827 [cs.CL]. URL: https://arxiv.org/abs/2509.08827.

[119] Ting Zhang et al. *Benchmarking Large Language Models for Multi-Language Software Vulnerability Detection*. 2025. arXiv: 2503.01449 [cs.SE]. URL: https://arxiv.org/abs/2503.01449.

[120] Xinran Zheng et al. *Learning to Focus: Context Extraction for Efficient Code Vulnerability Detection with Language Models*. 2025. arXiv: 2505.17460 [cs.SE]. URL: https://arxiv.org/abs/2505.17460.

[121] Yaqin Zhou et al. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. 2019. arXiv: 1909.03496 [cs.SE]. URL: https://arxiv.org/abs/1909.03496.

[122] Yuan Zhou, Enze Wang, and Shuoyoucheng Ma. *SSRFSeek: An LLM-based Static Analysis Framework for Detecting SSRF Vulnerabilities in PHP Applications*. Apr. 2025. DOI: 10.1109/AINIT65432.2025.11035424.

[123] Hao Zhu et al. *Specification-Guided Vulnerability Detection with Large Language Models*. 2025. arXiv: 2511.04014 [cs.SE]. URL: https://arxiv.org/abs/2511.04014.

Appendices

A Prompts

In this section of the appendix, we cover the full prompts used in `VulnReason-RAG`.

A.1 Vulnerability Knowledge Generation Prompt

In Figure 16, the full prompt used to generate vulnerability knowledge for the VKB can be found. The prompt assigns a role to the LLM, explains the task, supplies the code, and gives a clear output template.

Prompt for Vulnerability Knowledge Generation

You are a vulnerability knowledge generator.
Given the following vulnerable code, patched code and code difference, explain what the vulnerability is and how it was fixed. If you really do not know, state it as short as possible. Keep it concise. Do not repeat the code snippet in your response! After you generated the knowledge, reflect on your process to confirm it is correct, but do not generate output during this process. If you determine your final answer is not correct, regenerate it.

The following vulnerable code, patched code, and code difference need analysis:

Vulnerable code:
{Vulnerable code snippet}

Patched code:
{Patched code snippet}

Code difference:
{Code difference snippet}

Please use the exact structure and section names in your output as given below:

[Functionality]: What is the original functionality of the code?
[Vulnerability]: What is the vulnerability in the vulnerable code?
[Impact]: What is the impact when this vulnerability is exploited?
[Patch]: How was this vulnerability fixed in the patched code?
[Security Rule (specific)]: Provide in 1 sentence the specific security rule that was violated leading to this vulnerability.
[Security Rule (general)]: Provide in 1 sentence the general security rule that was violated leading to this vulnerability.

Figure 16: The full prompt used for the vulnerability knowledge generation for the VKB.

A.2 LLM-as-a-Judge Prompt

See Figure 17 for the full prompt used for the LLM-as-a-Judge. The prompt assigns a role to the LLM, explains the task, supplies the code difference and knowledge, and gives a clear output template.

Prompt for Vulnerability Knowledge Validation (LLM-as-a-Judge)

You are an LLM-as-a-Judge. Your task is to compare the code difference between vulnerable code and patched code with knowledge that is generated about this vulnerability. When comparing, check if every single component ([Functionality], [Vulnerability], [Impact], [Patch], [Security rule (specific)], [Security rule (general)]) is present in the generated knowledge and if it matches the code difference.

After the comparison, reflect on your thought process and change your final response if necessary (without generating any further output!).

Code difference:
{Code difference}

Generated knowledge:
{Generated knowledge}

Please use the exact JSON structure as given below:
{"response": "Yes"/"No", "explanation": "Your thought process and reasoning behind the answer."}

Figure 17: The full prompt used for the vulnerability knowledge validation (LLM-as-a-Judge) for the VKB.

A.3 Textual Representation Input Code Snippet Prompt

See Figure 18 for the full prompt used to generate a textual representation of the input code snippet and a list of potential weaknesses that may be present in this code snippet. The prompt assigns a role to the LLM, explains the task, supplies the input code snippet, and gives a clear output template.

Prompt for Code Description Input Code Snippet

You are a code analyzer. Your task is to generate a textual representation of the following code snippet, including its functionality and structure. Additionally, you should generate a list of potential weaknesses that may exist within this code. You do not have to be 100% certain these weaknesses are present, though you should only list weaknesses that can realistically be present within this code snippet.

Keep your answer concise, make sure you do not end up in repetitions, and for both the textual representation and potential weaknesses, make sure you explain your reasoning. Reflect on the textual representation and make any edits if you deem necessary.

Code snippet:
{code snippet}

Please use the following output template:

[Textual representation]:

[List of potential weaknesses]:
- Potential weakness 1:
- Potential weakness 2:
...
- Potential weakness n:

Figure 18: The full prompt used for generating a textual representation and a list of potential weaknesses for the input code snippets used in the vulnerability detector.

A.4 LLM Re-ranker Prompt

See Figure 19 for the full prompt used to re-rank the retrieved top-l relevant entries from the VKB into the top-k relevant entries. The prompt assigns a role to the LLM, explains the task, supplies the input code snippet, and gives a clear output template.

Prompt for LLM Re-ranker

You are an LLM re-ranker. You receive a description of a code snippet and a set of l retrieved vulnerability reports about relevant code snippets that contain vulnerabilities. Based on the description, re-rank the retrieved vulnerability reports such that the vulnerability report that may be most relevant to the given description is in position 1. You must retrieve exactly k entries. The least relevant vulnerability report that you still include in the top-k, should be on position k. Do not retrieve more or less than k entries! After generating your answer, make sure that the entries you selected are the most relevant and that you did not retrieve more or less than k entries. Do not add 'ID' as prefix, or any other text. Just the numbers separated by commas.

[Description]:
{Textual representation and list of weaknesses of input code snippet}

[Retrieved knowledge]:
{entry 1}
{entry 2}
...
{entry l}

Please structure your output like this:
1,2,...,k

In which 1,2,...,k represent the IDs of the vulnerability reports, in order of relevance.

Figure 19: The full prompt used for the LLM re-ranking process.

A.5 Vulnerability Detection Prompt

See Figure 20 for the full prompt used to determine whether there is a vulnerability in the given code snippet. The LLM is supplied with the code snippet, a textual representation, a list of potential weaknesses, and the top-k retrieved relevant entries from the VKB. The prompt assigns a role to the LLM, explains the task, supplies the input code snippet, and gives a clear output template.

Prompt for Vulnerability Detector

You are a vulnerability detector. Given the following code snippet, a textual representation of the code snippet, a list of potentially present weaknesses, and the top-k retrieved relevant code snippets with knowledge, determine if the given code snippet is vulnerable or not.

Code snippet:
{code snippet}

Textual representation + List of potential weaknesses:
{textual representation and potential weaknesses}

Top k relevant knowledge entries:
{entry 1}
{entry 2}
...
{entry k}

Please use the following JSON template as output format. Do not deviate from this template, and only generate valid JSON objects! When you determine a code snippet to be vulnerable, give the CWE ID(s) (one or multiple), as strings (with singular quotation marks!) within a list. This list must be within double quotation marks! If you determine a code snippet to not be vulnerable, state "N/A" instead of a list. Finally, as value for "explanation", give a brief explanation of how you came to your answer. After generating the JSON object, reflect on it and determine if it correctly judges the code snippet you have received, and if the structure and quotation marks are exactly following the template.

{"vulnerable": "Yes"/"No", "cwe": "['CWE-001', 'CWE-002', ...]"/"N/A", "explanation": "Your thought process and reasoning behind your answer."}

Figure 20: The full prompt used for the vulnerability detection.

B Experiment Evaluation Failure Analysis

In this section of the appendix, we cover a more detailed overview of failures during the experiment execution. For each experiment, per iteration for each configuration, between 3 and 18 input code snippets did not lead to a successful vulnerability classification. This occurred due to various reasons, including API errors, JSON construction errors and context window limitations.

B.1 Failure Analysis Experiment 1

Table 34 presents the number of failures per configuration and per iteration for experiment 1. We observe that using GPT-OSS 120B leads to the same number of failures when internal reasoning is disabled and enabled. This is different when using Gemini 2.5 Flash-Lite and Gemma 4 26B A4B. Enabling reasoning here leads to more failures. Especially with Gemma 4 26B A4B, an increase of 193.3% in failures is observed.

	Reasoning Disabled	Reasoning Enabled
GPT-OSS 120B		
Iteration 1	8	9
Iteration 2	7	8
Iteration 3	8	6
Total	**23**	**23**
Gemini 2.5 Flash-Lite		
Iteration 1	5	6
Iteration 2	3	6
Iteration 3	4	8
Total	**12**	**20**
Gemma 4 26B A4B		
Iteration 1	5	13
Iteration 2	4	13
Iteration 3	6	18
Total	**15**	**44**

Table 34: Number of failures per configuration and per iteration for experiment 1.

B.2 Failure Analysis Experiment 2

Table 35 presents the number of failures per configuration and per iteration for experiment 2. We observe no large differences between configurations; every iteration has between 6 and 10 failures.

	Reasoning Disabled	Reasoning Enabled
No RAG		
Iteration 1	8	7
Iteration 2	8	8
Iteration 3	8	9
Total	**24**	**24**
RAG Knowledge + CVE		
Iteration 1	7	7
Iteration 2	6	10
Iteration 3	6	8
Total	**19**	**25**
RAG Knowledge + CVE + CWE		
Iteration 1	6	9
Iteration 2	8	8
Iteration 3	7	10
Total	**21**	**27**
RAG Knowledge + CVE + CWE + Code		
Iteration 1	8	8
Iteration 2	8	10
Iteration 3	8	9
Total	**24**	**27**

Table 35: Number of failures per configuration and per iteration for experiment 2.

	Reasoning Disabled	Reasoning Enabled
No RAG		
Iteration 1	8	8
Iteration 2	7	8
Iteration 3	9	7
Total	**24**	**23**
RAG Only BM25		
Iteration 1	8	7
Iteration 2	7	8
Iteration 3	7	9
Total	**22**	**24**
RAG BM25 + LLM Re-ranking		
Iteration 1	7	12
Iteration 2	6	7
Iteration 3	7	8
Total	**20**	**27**

Table 36: Number of failures per configuration and per iteration for experiment 3.

B.3 Failure Analysis Experiment 3

Table 36 presents the number of failures per configuration and per iteration for experiment 3. We generally observe no large differences between configurations; every iteration for every configuration has between 6 and 9 failures. However, there is an outlier of 12 failures for iteration 3 of configuration BM25 + LLM re-ranker with internal reasoning enabled. Since the other two iterations remain consistent with the other configurations, this suggests that this effect is likely not configuration-specific.

B.4 Failure Analysis Experiment 4

Table 37 presents the number of failures per configuration and per iteration for experiment 4. We observe no large differences between most configurations; every iteration for most configurations has between 6 and 11 failures.

However, a notable observation is that when only the first third of the VKB is used, and internal reasoning is enabled, more failures occur. A potential explanation for this is API errors due to unstable API behavior.

	Reasoning Disabled	Reasoning Enabled
No RAG		
Iteration 1	10	7
Iteration 2	7	7
Iteration 3	9	9
Total	**26**	**23**
RAG First Third VKB		
Iteration 1	6	11
Iteration 2	7	14
Iteration 3	11	12
Total	**24**	**37**
RAG First and Second Third VKB		
Iteration 1	7	7
Iteration 2	8	10
Iteration 3	9	9
Total	**24**	**26**
RAG Full VKB		
Iteration 1	7	7
Iteration 2	8	8
Iteration 3	8	8
Total	**23**	**23**

Table 37: Number of failures per configuration and per iteration for experiment 4.