Specification-Guided Vulnerability Detection with Large Language Models

HAO ZHU, Peking University, China
JIA LI, Tsinghua University, China
CUIYUN GAO, Harbin Institute of Technology, China
JIARU QIAN, Peking University, China
YIHONG DONG, Peking University, China
HUANYU LIU, Peking University, China
LECHENG WANG, Peking University, China
ZILIANG WANG, Peking University, China
XIAOLONG HU, New H3C Technologies Co., Ltd, China
GE LI*, Peking University, China

Large language models (LLMs) have achieved remarkable progress in code understanding and analysis tasks. However, state-of-the-art LLMs demonstrate limited performance in vulnerability detection tasks, and even state-of-the-art models struggle to distinguish vulnerable code from patched code. We argue that a key reason for this limitation is that LLMs lack an understanding of **security specifications**—the expectations defined by developers and security teams about how code should behave to remain safe. When the actual behavior of the code differs from these expectations and introduces a security risk, it becomes a potential vulnerability. However, such knowledge is rarely explicit in training data, leaving models unable to reason about the root causes of security flaws. To address this challenge, We propose **VulInstruct**, a specification-guided approach that systematically extracts reusable security specifications from historical vulnerabilities to instruct the detection of new ones. Specifically, VulInstruct designs two automatic pipelines to construct a **specification knowledge base** from complementary perspectives: (i) **General specifications**, extracted from high-quality patches across diverse projects, capturing fundamental safe behaviors accumulated across the open-source ecosystem; and (ii) **Domain-specific specifications**, context-dependent expectations repeatedly violated in particular repositories or domains that are relevant to the target code under analysis. Before analyzing new code, VulInstruct leverages this specification knowledge base to retrieve relevant past cases and their associated specifications, enabling LLMs to reason about expected safe behaviors rather than relying solely on surface patterns. We evaluate VulInstruct under strict evaluation criteria requiring both correct predictions and valid reasoning. On the PrimeVul dataset, VulInstruct achieves 45.0% F1-score (32.7% improvement) and 37.7% recall (50.8% improvement) compared to the strongest baselines, while uniquely detecting 24.3% of all identified vulnerabilities—2.4× more than any baseline. In pair-wise evaluation distinguishing vulnerable from patched code, VulInstruct also achieves a 32.3% relative improvement over the best baseline. Beyond benchmarks, VulInstruct discovered a previously unknown high-severity vulnerability in production code (later assigned CVE-2025-56538) by recognizing violations of extracted specifications, demonstrating its practical value for real-world vulnerability discovery. All code and supplementary materials are available at https://github.com/zhuhaopku/VulInstruct-temp.

*Corresponding author.

Authors' Contact Information: Hao Zhu, Peking University, Beijing, China, zhuhao@stu.pku.edu.cn; Jia Li, Tsinghua University, Beijing, China, jia_li@mail.tsinghua.edu.cn; Cuiyun Gao, Harbin Institute of Technology, Harbin, China, gaocuiyun@hit.edu.cn; Jiaru Qian, Peking University, Beijing, China, qianjiaru77@gmail.com; Yihong Dong, Peking University, Beijing, China, dongyh@stu.pku.edu.cn; Huanyu Liu, Peking University, Beijing, China, huanyuliu@stu.pku.edu.cn; Lecheng Wang, Peking University, Beijing, China, wanglecheng@stu.pku.edu.cn; Ziliang Wang, Peking University, Beijing, China, wangziliang@pku.edu.cn; Xiaolong Hu, New H3C Technologies Co., Ltd, Hangzhou, China, xlhu@h3c.com; Ge Li, Peking University, Beijing, China, lige@pku.edu.cn.

1 Introduction

Software vulnerability detection plays a critical role in software security, aiming to identify potential security flaws in source code that could be exploited by malicious attackers. With the rapid development of large language models (LLMs) and their success in code understanding and analysis tasks [3, 10, 13, 31, 41], recent studies have explored using LLMs for automated vulnerability detection [12, 22, 27, 28, 35, 40, 42]. While these LLM-based approaches have achieved notable improvements over traditional methods, their performance on vulnerability detection benchmarks remains unsatisfactory. Recent evaluation [2] shows that even state-of-the-art LLMs achieve less than 12% accuracy in distinguishing vulnerable from patched code, highlighting a fundamental limitation in understanding the root causes of vulnerabilities.

We argue that one key reason behind the limited performance of LLMs is that models lack an understanding of the **security specifications** in code. A security specification is the expectations defined by developers and security teams about how the code should behave in order to remain safe. When the actual behavior of the code differs from this expectation and introduces a security risk, it becomes a potential vulnerability. The following example illustrates this fundamental challenge:

Example: TLS Certificate Validation

Consider a case where a TLS library fails to verify the hostname of a server during certificate validation. To a developer who is unaware of the relevant specification, this code might appear correct: the function successfully completes a TLS handshake and retrieves the server certificate. However, the expected code behavior requires that the hostname in the certificate must be checked against the intended server identity. Only when this expectation is made explicit does it become clear that the missing check introduces a serious security flaw, since an attacker could impersonate any server with a valid certificate.

However, such specifications are rarely documented explicitly in code or public resources. In practice, they are typically discussed and agreed upon during design or review phases, and prior work [29] has noted that this knowledge is often shared through internal training or reviews. As a result, LLMs trained primarily on code rarely encounter these implicit expectations and thus struggle to reason about the root causes of vulnerabilities.

Our key motivation is that historical vulnerabilities implicitly encode the security specifications defined by developers and security teams (we provide a detailed discussion and illustrative cases in Section 3). Based on this insight, we propose **VulInstruct**, a specification-guided approach for vulnerability detection. The central idea of VulInstruct is to extract **reusable security specifications from historical vulnerabilities** and use them to instruct the detection of new ones. Before analyzing a target code snippet, VulInstruct retrieves similar historical vulnerability cases and provides their associated specifications as explicit security knowledge. Under the support of such specifications, LLMs can go beyond common or surface defect patterns and reason about whether the code violates an expected safe behavior.

A key technical challenge is how to extract security specifications in a systematic way. To address this, VulInstruct designs two automatic pipelines to construct a **specification knowledge base** from complementary perspectives: **(i) General security specifications.** Extracted from high-quality patch datasets across diverse projects, a focus that has long been central to the vulnerability detection community, these specifications capture fundamental expected behaviors representing security expertise accumulated across the entire open-source ecosystem. We then extract them from high-quality patch datasets by comparing vulnerable code with its fixed version and restating

the underlying expected behaviors as explicit, reusable specifications. To move beyond prior works, we further capture surrounding information such as callee functions, type declarations, imported modules, and global variables alongside the vulnerable and patched code, enabling more precise abstraction of the developer's intended safe behavior. **(ii) Domain-specific security specifications**: Dynamically extracted from our comprehensive CVE database, these specifications are derived from frequently exploited vulnerabilities within the same repository or related projects in the same domain of the target detected code. By studying how attackers repeatedly exploit similar weaknesses, we identify context-specific expectations whose violations enable attacks. These specifications capture the attacker's perspective, revealing which expected behaviors matter most in practice for specific codebases. Together, the two types of specifications are complementary: general specifications provide broad coverage of fundamental secure behaviors, while domain-specific specifications capture context-aware expectations that matter most in practice.

We evaluate VulInstruct under the stricter evaluation framework [14], which requires not only correct predictions but also valid reasoning. On the widely adopted PrimeVul dataset [2], VulInstruct achieves 45.0% F1-score, representing a 32.7% relative improvement over the strongest baseline. VulInstruct's 37.7% recall substantially outperforms all baselines, with 24.3% of its detections being unique vulnerabilities that other methods miss entirely. In pair-wise evaluation, where models must correctly distinguish vulnerable code from its patched code, VulInstruct achieves 17.2% P-C accuracy, improving by 32.3% over the best baseline. Our analysis further reveals that effective vulnerability detection requires not just access to security knowledge but careful selection of the most relevant specifications and cases. Beyond controlled evaluations, we demonstrate VulInstruct's real-world utility through a case study where our specification-guided approach successfully identified a previously unknown high-severity vulnerability. The vulnerability, which violated the same security specification as a known CVE, was confirmed and subsequently patched by developers, highlighting VulInstruct's potential for practical vulnerability discovery.

Our contributions are summarized as follows:

- We are the first to systematically extract and apply *security specifications* from historical vulnerabilities, showing that reasoning about expected safe behavior is crucial for detecting root causes of vulnerabilities.
- We propose VulInstruct, a novel specification-guided approach that extracts implicit knowledge from historical vulnerabilities through a dual-layer retrieval design and leverages it to instruct LLMs in vulnerability detection.
- Under more strict evaluation, VulInstruct achieves state-of-the-art vulnerability detection performance, improving F1-score by 32.7% and recall by 50.8% over the strongest baselines.

2 Background and Related Work

The limitations of LLM-based vulnerability detection were most clearly articulated by the PrimeVul benchmark[2]. The authors concluded that even state-of-the-art LLMs achieve less than 12% accuracy in distinguishing vulnerable code from patched code. Their discussion explicitly highlighted three challenges for future research: (i) the need for richer program context, (ii) the importance of teaching models to reason about vulnerability detection beyond binary labels, and (iii) the lack of security awareness in current models. Subsequent work has largely followed these directions.

2.1 Code Context and Reasoning in Vulnerability Detection

Leveraging richer program context. Recent studies [30, 32] have identified code context as a critical factor in vulnerability detection. Risse et al. [23] reported that most of the functions in widely used benchmarks cannot be reliably classified without knowing of how they are invoked

and used in the program. They enumerated several types of important contextual information, such as external functions, type declarations, and global variables. More recent approaches have explored various strategies to leverage richer program context. Yildiz et al. [39] uses ReAct agents for recursive callee retrieval, Yang et al. [37] abstracts primitive API calls at multiple granularity levels. Wang et al. [18] introduced Savant, which uses LLM semantic understanding to detect vulnerable API usages in Java dependencies through reflection-based iterative context expansion. Shemetova et al. [25] leveraged LLMs to automatically generate function annotations for memory allocation/deallocation patterns in code context to integrate with static analyzers.

Teaching models to reason about vulnerability detection. In response to PrimeVul's second challenge, a growing line of work has focused on enhancing models' reasoning capabilities in vulnerability detection. One direction is to use prompting strategies and in-context learning. These approaches [15, 19, 22, 26, 38] typically leverage program analysis techniques such as control flow graphs (CFG), data flow analysis, and program dependency graphs (PDG) to construct prompts that stimulate models' exploration capabilities. Beyond prompting, training-based methods often decompose vulnerability detection into subtasks to improve reasoning quality. Du et al. [4] proposed VulLLM, which uses multi-task learning across detection, localization, and interpretation. Weyssow et al. [34] introduced R2Vul, combining reinforcement learning with structured reasoning distillation to train small LLMs for vulnerability detection with explanations. Wen et al. [33] proposed ReVD, which applies triplet supervised fine-tuning with curriculum preference optimization to capture vulnerability patterns. A third line of work leverages multi-round, multi-agent interactions to simulate structured reasoning. Hu et al. [9] proposed GPTLens with auditor agents generating vulnerabilities and critic agents evaluating them adversarially. Widyasari et al. [36] introduced VulTrial, a courtroom-inspired multi-agent framework featuring four role-specific agents, achieving 102.39% improvement over single-agent baselines with GPT-4o.

Despite these advances, most existing studies still evaluate models using binary labels: whether code is vulnerable or not. Such evaluation is unsatisfactory for LLMs: a model may predict the correct label but rely on spurious correlations rather than genuine understanding of root causes. Consequently, researchers [11, 16] have increasingly advocated for reasoning-aware evaluation. Li et al. [14] proposed the CORRECT framework, which requires both the binary classification label and the reasoning process to be correct. By leveraging sufficient code context, they demonstrated that stricter evaluation can reveal whether models truly understand vulnerability mechanisms. As LLMs' reasoning capabilities continue to improve, more works are shifting toward such stricter metrics. This trend motivates our adoption of CORRECT evaluation in this paper.

2.2 Security Awareness in LLM-based Detection

Augmenting Security Knowledge. PrimeVul also emphasized that current code LMs lack what the authors termed "security awareness." In their discussion, this referred to the observation that models often decide based on textual similarity in training dataset rather than considering the underlying causes of vulnerabilities or fixes of the vulnerabilities. They suggested that researchers should explore ways to teach LMs security knowledge, inspired by how human developers are trained in secure coding.

In practice, however, such knowledge rarely exists explicitly. Security expertise in industry is often implicit: experts accumulate best practices through hands-on experience, and then transfer this knowledge to developers through periodic training sessions or internal guidelines [29]. Yet no comprehensive dataset currently exists that systematically captures and encodes such security knowledge for AI models. Several efforts have begun to make such implicit security knowledge more explicit. BugScope [7] leverages LLMs to summarize curated vulnerability cases, thereby expanding code patterns associated with common vulnerability types. In the formal analysis community,

APHP [17] and Seal [1] recover API usage constraints from security patches, identifying rules such as "`release()` must follow `acquire()`." These studies demonstrate that structured rules can indeed be distilled from developer fixes, turning implicit practices into explicit specifications.

In contrast, most existing methods in vulnerability detection, including Vul-RAG [5] and the training-based approaches discussed above, primarily rely on exposing models to more vulnerability examples through retrieval or fine-tuning. While this can improve recognition of familiar patterns, it does not provide models with the underlying expectations about safe behavior that truly determine whether code is vulnerable. Consequently, models remain limited to pattern matching rather than reasoning with explicit security knowledge. A systematic way to extract and apply such structured security knowledge has been lacking, motivating the design of our approach.

3 Motivation

As introduced in Section 1, vulnerabilities fundamentally arise when an implementation violates a security specification. We define a security specification as the expectation about safe program behavior established by developers and security experts. These expectations are rarely documented explicitly, but can be reconstructed from historical evidence. We identify two complementary perspectives for recovering such expectations: **(i) General specifications:** By analyzing how vulnerable code differs from its fixed version across high-quality patches, we can explicitly restate the expected safe behavior that was violated. **(ii) Domain-specific specifications:** By studying vulnerabilities that share similar exploitation mechanisms within the same domain, we can recover the underlying expectations that were repeatedly violated.

A key *insight* is that every vulnerability—no matter how local or project-specific—can be traced to the violation of at least one underlying security specification. For example, failing to reset session state after authentication errors violates an expectation of state consistency; using freed memory without nullification violates an expectation about pointer lifecycle; neglecting TLS hostname verification violates an expectation about protocol boundary enforcement. These expectations, which we formalize as *security specifications*, serve as a unifying representation of implicit expert knowledge, making them both interpretable to humans and transferable across projects.

3.1 Learning General Specifications: TLS Hostname Verification

Case Study: TLS Hostname Verification. The importance of hostname verification in TLS has been recognized for decades. RFC 2595 (1999) [21] already mandated that "the client MUST check its understanding of the server hostname against the server's identity… to prevent man-in-the-middle attacks", later consolidated in RFC 6125 (2011) [24]. Despite being standardized, such requirements rarely appear in project documentation and remain security experts' implicit knowledge base. In practice, this specification is often overlooked. Figure 1(a) shows CVE-2013-4488 in the `libgadu` library, where the TLS negotiation handler retrieved and logged the server certificate but failed to verify it against the target hostname. This omission allowed attackers with any valid certificate—even self-signed—to impersonate arbitrary servers, completely undermining TLS security.

VulInstruct's Automated Specification Learning. From the patch that fixed CVE-2013-4488 by adding hostname verification, VulInstruct automatically extracts the underlying security principles. Specifically, it inferred two structured specifications: **(i) HS-SEC-001**: TLS implementations must perform complete certificate chain validation including hostname verification. **(ii) HS-PROTOCOL-002**: TLS handshakes must enforce strict verification of all X.509 certificate fields. This illustrates how VulInstruct turns implicit expert rules into explicit, LLM-readable specifications, bridging the gap between human reasoning and model inference.

Specification-guided Detection. Once extracted, these specifications become reusable knowledge. For example, when analyzing `e2guardian` (CVE-2021-44273), VulInstruct retrieved HS-SEC-001

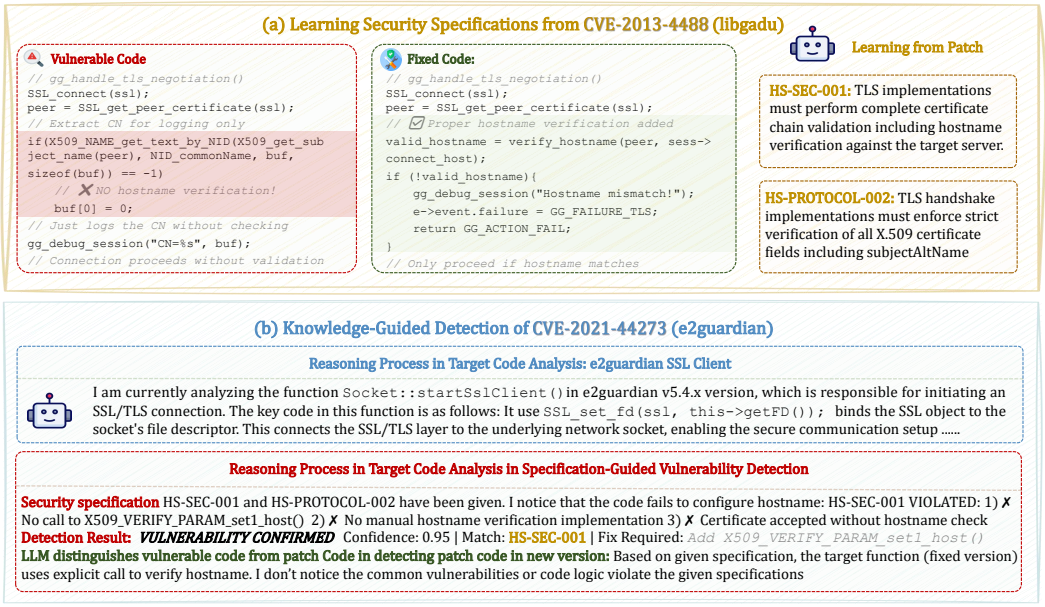


Fig. 1. (a) CVE-2013-4488 vulnerability analysis: The vulnerable code path in `libgadu` showing missing hostname verification, and VulInstruct's automated specification extraction from the patch. (b) VulInstruct using specifications from historical cases to detect new vulnerability

and successfully detected a similar omission in the function `Socket::startSslClient()`, where no hostname verification was performed for OpenSSL (Figure 1b). The system flagged the violation and explained the attack impact, confirming the vulnerability with high confidence. This case demonstrates how specifications distilled from historical patches can generalize across projects, enabling specification-guided detection of previously unseen vulnerabilities.

3.2 Learning Domain-specific Specifications: Image Parser Vulnerabilities

Attackers rarely invent entirely new exploits; instead, they frequently *reuse exploitation strategies* that have succeeded in the past. Google Project Zero reported that among 18 zero-day vulnerabilities disclosed in 2022, at least half were variants of previously patched vulnerabilities [6]. This recurrence shows that vulnerabilities often stem from the same underlying weakness, which attackers repeatedly target across projects in the same domain.

Why can such exploits be reused? Because each recurring exploitation mechanism corresponds to the violation of the similar **security specification**—an implicit expectation about safe behavior that was never properly enforced. In this sense, attack patterns are not separate from security specifications, but evidence of which specifications matter most in practice. By abstracting recurring exploitation mechanisms into explicit specifications, we capture the security expectations needed to prevent the same class of attacks from reappearing and make them reusable knowledge for guiding vulnerability detection.

Case Study: Image Parsing in ImageMagick. Figure 2 illustrates how similar exploitations emerge in practice. Figure (a) summarizes a sequence of historical CVEs (2015–2016) in ImageMagick image parsers. Despite occurring in different formats (SUN, BMP, RLE), these vulnerabilities share a recurring exploitation mechanism: **untrusted size fields from image headers were directly used in allocation or offset calculations.** For example, CVE-2015-8957 trusted width × height

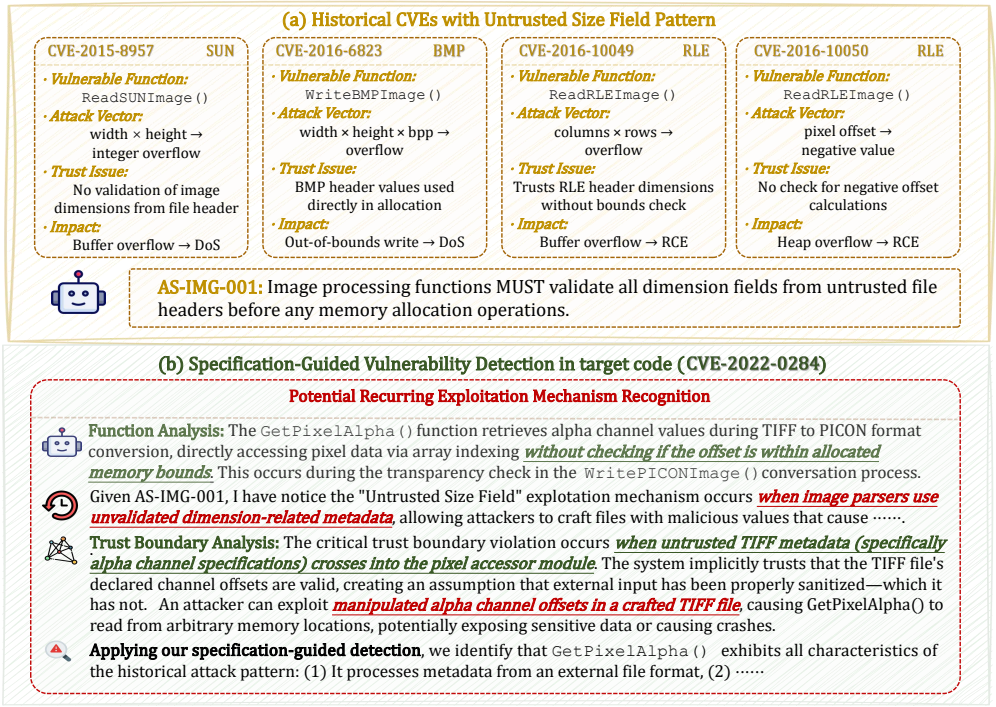


Fig. 2. Domain-specific Recurring exploitation mechanism in ImageMagick: unvalidated length fields across historical and new vulnerabilities.

values in the SUN image parser, leading to buffer overflow, while CVE-2016-6823 and CVE-2016-10049 in BMP and RLE coders suffered similar overflows due to unchecked dimension metadata. Across cases, the common thread is the same: failure to validate external size metadata before use.

Applying Domain-specific Specifications. Figure 2b illustrates how VulInstruct leverages retrieval and abstraction during the analysis of CVE-2022-0284. When examining the function `GetPixelAlpha()` in the TIFF-to-PICON conversion pipeline, the system first retrieved historical CVEs with similar characteristics where untrusted dimension metadata flowed directly into memory access (as shown in Figure 2a). VulInstruct then identified a consistent exploitation mechanism: "Untrusted Size Field" exploitation pattern and concluded AS-IMG-001 specification. Finally, VulInstruct applied this specification back to the target code, constructing a threat model that explained how manipulated alpha channel offsets in a crafted TIFF file could trigger out-of-bounds access.

4 Methodology

Figure 3 shows the workflow of **VulInstruct**, which combines *offline* knowledge construction with *online* specification-guided detection. Offline, VulInstruct designs two automatic pipelines to build a **Specification Knowledge Base (SKB)** that includes (i) a *general specification knowledge base*, consisting of reusable specifications extracted from high-quality patches across diverse projects, and (ii) a large-scale *domain evidence base*, collected from the CVE database, which provides relevant vulnerability cases for deriving domain-specific specifications. *Online*, given a target code snippet, VulInstruct performs dual-path retrieval: spec-level retrieval over general specifications, and case-level retrieval over the domain evidence base. From the retrieved cases, it dynamically induces

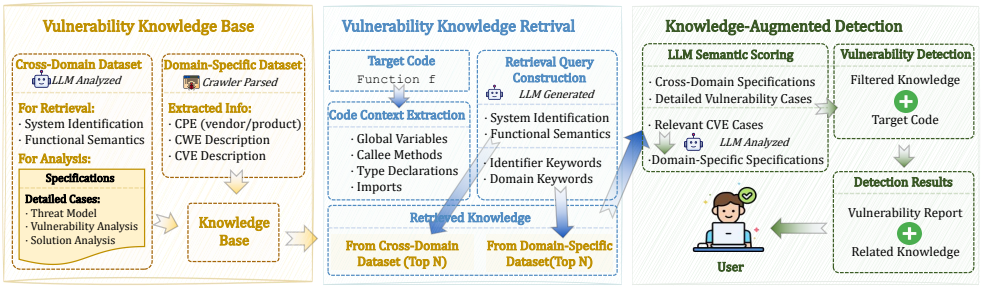


Fig. 3. Overview of VulInstruct

domain-specific specifications. Finally, VulInstruct performs specification-guided reasoning to detect vulnerabilities with explanations.

4.1 Specification Knowledge Base

The SKB consists of two complementary parts. **general specifications** are pre-extracted, reusable rules of expected safe behavior, constructed offline from high-quality patches across diverse projects. In contrast, **domain-specific specifications** are not pre-built; instead, we maintain a domain evidence base collected from the CVE database, and during detection, VulInstruct dynamically abstracts domain-specific specifications from the most relevant retrieved cases. This design allows general specifications to provide broad, cross-project coverage, while domain-specific specifications capture context-aware expectations directly relevant with the target code.

4.1.1 General Specification Knowledge Base. The general specification knowledge base in VulInstruct contains two kinds of knowledge: (i) *general specifications*, reusable rules that abstract the expected safe behavior, and (ii) *detailed vulnerability cases*, structured analyses that ground each specification in concrete vulnerability–patch examples. Together, they capture both general, reusable specifications and the concrete detail from real-world codebases.

General specifications. A general security specification is designed as follows:

$$\texttt{HS-[DOMAIN]-[ID]} : [\text{Expected Safe Behavior}] \tag{1}$$

where `DOMAIN` indicates the security domain (e.g., MEM for memory safety, PROTOCOL for communication protocols, STATE for state consistency), while `ID` provides a unique index. The requirement is always expressed in a *Expected Safe Behavior* form ("must do X" rather than "must not omit X"), ensuring both clarity and reusability.

Detailed vulnerability cases. Each specification is linked with a structured case that grounds the abstract rule in concrete evidence from real code. Each case is documented in three forms:

- *Threat model.* A system-level view that clarifies trust boundaries, attack surfaces, and the vulnerability chain (e.g., how an initial flaw escalates into a critical CWE). This highlights why the specification matters in the broader security context, ensuring that the rule is not only syntactic but rooted in realistic attack scenarios.
- *Vulnerability analysis.* Code analysis that traces the vulnerable execution path, identifies entry points and preconditions, pinpoints the flawed logic, and maps the violation back to the specification, which provides precise evidence of how the specification is broken in practice.
- *Solution analysis.* Patch explanation that describes how the applied fix restores compliance with the violated specification, explicitly linking code changes to expected safe behaviors. This step confirms that the specification not only explains the cause of the vulnerability but also captures the principle underlying its fixing.

Together, these analyses complement the reusable specification itself: the specification captures the expected safe behavior, while the linked case provides concrete evidence of its violation and enforcement in real-world code.

Full automatic pipeline. We extract specifications from high-quality patch datasets. Each vulnerability instance is represented as $D_i = \{c_i^{\text{vul}}, c_i^{\text{fix}}, m_i, d_i, w_i, \text{Ctx}(f_i)\}$, where c_i^{vul} and c_i^{fix} are the vulnerable and fixed functions, m_i is the commit message, d_i the CVE description, w_i the CWE type, and $\text{Ctx}(f_i)$ the broader program context:

$$\text{Ctx}(f) = \{\text{callees}(f, d),\ \text{types}(f),\ \text{imports}(f),\ \text{globals}(f)\}. \tag{2}$$

This representation ensures that each instance is grounded not only in the function body but also in its interaction with the surrounding system.

Then, each general specification is inferred by prompting LLMs to perform structured Chain-of-Thought reasoning on each vulnerability instance. We guide LLM through a carefully designed one-shot prompt, which first classifies vulnerabilities according to 10 core security domains and then encourages reasoning about the vulnerability's root cause and the corresponding expected safe behavior. Given the extracted general specifications, we further prompt the LLM to generate detailed vulnerability cases that link each specification to concrete evidence in the code.

Retrieval keys. Finally, to enable efficient matching with new code, we attach two retrieval keys to each specification: (i) a *system identification key* describing the subsystem, module, and interactions of the function, and (ii) a *functional semantics key* summarizing its intended behavior. Combined with the broader context in Equation 2, these keys allow VulInstruct to retrieve relevant specifications based on both program role and semantics, rather than relying on surface similarity.

4.1.2 Domain Evidence Base. We construct a domain evidence base to capture recurring exploitation patterns in CVE cases. Unlike general specifications, which require high-quality patch datasets, domain-specific cases rely on breadth rather than depth. For this purpose, we collect all publicly available CVEs from the National Vulnerability Database (NVD) [20], which is maintained by the U.S. National Institute of Standards and Technology (NIST) and provides comprehensive vulnerability metadata including severity scores, impact metrics, and reference information for each CVE entry. We design an automatic pipeline that crawls and normalizes CVE data from the National Vulnerability Database. In this work, we collect entries published between 2002 and 2024, excluding rejected ones. This process yields 173,070 valid cases and 15,391 rejected cases. For each CVE, the pipeline automatically normalizes key fields, including the vulnerability description, its associated CWE type, Common Platform Enumeration(CPE) identifiers (vendor and product names), reference links to advisories and patches, and the publication date for temporal filtering.

4.2 Knowledge Retrieval

The Vulnerability Knowledge Base (VKB) contains two complementary layers of knowledge: *general security specifications* and *domain-specific security specifications*. Given a target function f to be analyzed, the retrieval process aims to identify the most relevant knowledge items from the VKB that can guide subsequent detection.

Code Context Extraction. We first implement an automatic tool that, given a target function f, extracts its surrounding program context as defined earlier in Equation 2. The extracted context is then incorporated into the retrieval query construction to improve semantic alignment with historical vulnerabilities in knowledge bases.

Retrieval Query Construction for general specifications. To build effective retrieval queries, we prompt LLM to generate two forms of system-level descriptions for the target function: *system identification*, which specifies the subsystem, module, and component interactions where f resides,

and *functional semantics*, which summarize the purpose and detailed behavior of f. We then use **Embedding-based Retrieval.** We concatenate the target code, the system identification and functional semantics of f, and encode them into dense embeddings. Using embedding similarity search, we retrieve the most relevant *top-k cases*: $\{K_1^{(\text{spec})}, \ldots, K_k^{(\text{spec})}\}$ ranked by cosine similarity between the query embedding and the stored cases. Then we get the *specifications* and *detailed vulnerability analysis* in *top-k cases*.

Retrieval Query Construction for domain-specific specifications. We design a retrieval process that emphasizes identifiers and domain concepts, which capture the most relevant features of a target function likely to correlate with similar exploitation cases. We first use **Identifier Keywords and Domain Keywords.** For each target function, we instruct the LLM to generate two complementary sets of retrieval keywords. The first are *identifier keywords*, which anchor the function to its concrete software context. Following a priority hierarchy, LLM generates CPE vendor or product names (e.g., "ImageMagick"), repository or organization names, and component or module names. These identifiers ensure that vulnerabilities from the same project or closely related codebases are prioritized. The second are *domain keywords*, which characterize the broader functionality exposed to attackers. Here, LLM generates descriptors such as protocol names (e.g., TLS, HTTP, SSH), file formats (e.g., PNG, XML, PDF), or system components (e.g., parser, allocator, validator). Such domain keywords capture recurring attack surfaces even across different projects, enabling the discovery of similar exploitation strategies. Then we **filter and rank the retrieved cases in our comprehensive cve dataset.** To simulate how historical vulnerabilities would realistically assist detection, we adopt a two-stage filtering and ranking process: (i) We apply *temporal constraints*. For a target function associated with CVE x, we retain only cases whose publication date precedes that of x, and whose CVE identifiers were assigned earlier in the official numbering sequence. (ii) We perform *keyword-based filtering* using the identifier and domain keywords generated in the previous step. Candidate CVEs are retained only if their CPE fields or CVE descriptions contain at least one of the generated keywords. To avoid overwhelming matches, keywords that return more than 500 CVEs are discarded as overly broad. (iii) We apply a *naive ranking scheme* to prioritize the most informative candidates. Each case is scored based on three factors: (a) presence of attacker-related terms, like "attack", (b) description length as a proxy for detail richness, and (c) recency, where more recent vulnerabilities are favored. We compute an averaged weight across these dimensions to produce the final ranking. Using these strategies, we retrieve the most relevant top-k CVE cases in the same domain, $\{K_1^{(\text{cve})}, \ldots, K_k^{(\text{cve})}\}$, which constitute the candidate set after filtering and ranking.

4.3 Specification-guided Detection

In the final phase, VulInstruct integrates retrieved knowledge into the detection pipeline. The process consists of three stages:

Knowledge Scoring. For each target function, we prompt the LLM to evaluate the retrieved *top-k specifications*, their corresponding *detailed vulnerability cases*, and candidate *top-k CVE cases*. LLM scores each specification, detailed case, and CVE case on a 1-10 scale. We establish a simple 1-10 scoring rubric, where 10 points indicate highly relevant cases with strong alignment in vulnerability type and code features, and lower scores indicate weaker or only partial similarity. This scoring step ensures that subsequent reasoning is guided primarily by the most relevant knowledge rather than by noisy or tangential cases. We then apply a unified *threshold* to filter out low-scoring items. The surviving cases form the *high-relevance pool* of specifications and CVEs that guide reasoning.

Domain-specific specifications Generation. For the filtered set of relevant domain CVEs in *high-relevance pool*. VulInstruct induces reusable specifications by jointly analyzing multiple

related cases. Unlike general specifications that are pre-extracted at the patch level, domain-specific specifications are *dynamically abstracted* during detection. We leverage information from CVE descriptions, CWE types, and CPE identifiers, together with detailed exploitation notes, to identify recurring exploitation mechanisms. Formally, each domain-specific specification is expressed as:

$$\texttt{AS-[DOMAIN]-[ID]} : \text{[Expected Safe Behavior]} \tag{3}$$

Here, `DOMAIN` and `ID` correspond to the same concepts in Equation 1. In addition to the expected safe behavior, each entry also records its supporting *recurring exploitation mechanism*, derived by jointly analyzing multiple CVEs (e.g., heap-based buffer overflows from unchecked decoding in CVE-2014-0011 and CVE-2014-9629). This ensures that the specification is not only a defensive rule but also explicitly grounded in concrete exploitation patterns repeatedly observed in practice.

Structured Reasoning. Using the filtered knowledge as context, we instruct LLM to follow a structured Chain-of-Thought reasoning process. The final detection decision is then derived by aligning the target function against all the scored specifications, allowing LLM to explain vulnerabilities in terms of established security knowledge.

5 Experiment Design

5.1 Datasets

Our experimental setup uses two datasets with distinct roles: CORRECT provides rich contextual knowledge for building our vulnerability knowledge base, while PrimeVul serves as our evaluation benchmark. **Evaluation Dataset: PrimeVul** [2] is a widely adopted vulnerability detection benchmark. The key feature of PrimeVul is its *temporal data splitting*—training data comes from vulnerabilities before a cutoff date, while test data comes from after. This prevents models from "learning" future vulnerabilities and being tested on past ones, ensuring a realistic evaluation where detection systems must generalize from historical patterns to identify new vulnerabilities. Therefore, we use the PrimeVul vulnerable-patched pairs test set as our primary evaluation benchmark, which is widely used in our baselines. **Knowledge Source: CORRECT** [14] dataset contains 2000 vulnerable functions and provides comprehensive contextual information, including (i) complete callee functions at multiple depths, (ii) type declarations and data structures, (iii) global variables and constants, and (iv) import statements and module dependencies. We leverage this high-quality dataset to build our Vulnerability Knowledge Base.

To ensure fair evaluation and prevent data leakage, we apply strict temporal filtering when constructing our Vulnerability Knowledge Base: We use PrimeVul's test set cutoff date as our boundary. Only vulnerabilities published *before* this date are included in our knowledge base. After temporal filtering, our VKB construction dataset for general security specifications includes: From CORRECT 1,338 vulnerable-patched pairs that predate the PrimeVul test cutoff.

5.2 Baselines

We compare VulInstruct against state-of-the-art LLM-based vulnerability detection approaches across different paradigms: *Prompting Methods.* **Chain-of-Thought** [2]: We follow Ding et al.'s approach, prompting the LLM with step-by-step reasoning for vulnerability analysis. This serves as the baseline of direct prompting without enhancements. *Fine-tuning Methods.* **ReVD** [33] is the current state-of-the-art fine-tuned approach for vulnerability detection. It uses Qwen2.5-Coder as the base model and synthesizes 28,000 vulnerability analysis reasoning data for training, achieving state-of-the-art performance on PrimeVul. **VulTrial** [36]: An agent-based vulnerability analysis framework that employs four role-specific agents (researcher, author, moderator, and review board). It fine-tunes GPT-4o with role-specific instructions and achieves state-of-the-art performance among agent-based approaches on PrimeVul. *Agent-based Methods.* **GPTLens** [9]: multi-agent

framework that automates vulnerability analysis workflows and iterative reasoning to enhance LLMs reasoning ability. *Retrieval-Augmented Methods.* **Vul-RAG** [5] retrieves relevant vulnerability knowledge based on functional semantics and uses it to augment the detection process. The method extracts multi-dimensional knowledge, including vulnerability causes and fixing solutions.

5.3 Evaluation Metrics

As discussed in 2.1, using binary classification alone as the evaluation criterion for LLMs has been widely questioned. A model may predict the correct label while relying on spurious code patterns, which do not reflect the actual vulnerability scenario. This raises concerns that accuracy alone cannot demonstrate true understanding of the root cause of vulnerabilities.

Therefore, we adopt the CORRECT evaluation metrics [14], proposed by Li et al., which assess both the vulnerability label and the underlying reasoning process. LLMs must both predict correctly and provide reasoning that is consistent with ground-truth evidence, including CVE descriptions, patches, and commit messages, as verified by an LLM-as-a-Judge model (i.e., another LLM acting as an evaluator). With the correctness of predictions and reasoning process defined under CORRECT, we compute standard metrics for vulnerability detection, including accuracy $\frac{TP+TN}{TP+TN+FP+FN}$, precision $\frac{TP}{TP+FP}$, recall $\frac{TP}{TP+FN}$, and F1-score $\frac{2\cdot\text{precision}\cdot\text{recall}}{\text{precision}+\text{recall}}$. In addition, we employ *pair-wise prediction metrics* [2, 33] to assess whether models can distinguish vulnerable functions from their patched functions. Formally, let $(p_v, p_f) \in \{0,1\}^2$ denote prediction outcomes, where p_v indicates whether the model correctly identifies the *ground-truth vulnerability* in the original function (based on both label and reasoning process), and p_f indicates whether the model correctly recognizes the fixing in the function. In our evaluation, we adopt the commonly used **P-C** metric, where $(1, 0)$ denotes ideal detection: the model correctly identifies the vulnerability in the original code but not in its patched version. In addition, we employ the **VP-S** score [33], defined as VP-S = P-C − P-R, where P-R $(0, 1)$ represents reversed prediction. VP-S thus provides a stricter measure of pair-wise discrimination by rewarding correct detections while penalizing such reversed errors.

For clarity, we provide a section in the supplementary material (Section A in the Appendix file) that details the formulation in CORRECT evaluation framework and how we apply it in experiments.

5.4 Implementation Details

We provide key implementation details for reproducibility: (i) We use Qwen3-Embedding-0.6B to generate embeddings for retrieval queries and stored system identification and functional semantics. (ii) For both types of security specifications, we retrieve top-$k = 10$ candidates initially. In knowledge scoring, we apply a unified threshold of ≥ 6 points (6, 6, 6) to filter three types of knowledge. (iii) For knowledge extraction in two type specification base, keyword generation, knowledge scoring, we use DeepSeek-V3 model. (iv) For CORRECT evaluation metric, we use GPT-5 as LLM-as-a-Judge model, which represents the current state-of-the-art in evaluation capabilities, replacing the GPT-4o used in the original CORRECT implementation. (v) Details of our prompt formats in extracting specifications, knowledge scoring, and vulnerability detection can be found in the supplementary material (Section B in the Appendix file). We also provide the technical details about our automatic tool in code context extraction (Section F in the Appendix file).

6 Results and Analysis

6.1 RQ1: How effective is VulInstruct in vulnerability detection compared to state-of-the-art approaches?

Table 1 shows the results of vulnerability detection on the PrimeVul dataset. **VulInstruct achieves state-of-the-art performance with 45.0% F1 score, surpassing GPTLens by 32.7%.** More

Table 1. Vulnerability detection performance on PrimeVul.

Method	Model	Trained	Standard (%) Acc.↑	Prec.↑	Rec.↑	Unique↑	F1-Score↑	Pairwise (%) P-C↑	VP-S↑
Prompting-based Methods									
Ding et al.(CoT)	Deepseek-V3	✗	49.9	49.5	12.2	1.8	19.5	3.9	1.4
Fine-tuning Methods									
ReVD	Qwen2.5-Coder	✓	49.5	48.0	11.5	5.8	18.6	7.4	−0.9
VulTrial	GPT-4o	✓	53.4	59.9	20.8	10.2	30.9	12.3	6.9
Agent-based Methods									
VulTrial	DeepSeek-V3	✗	51.9	57.4	10.0	10.2	17.0	6.6	2.7
GPTLens	DeepSeek-V3	✗	51.3	52.8	25.0	4.4	33.9	13.0	2.7
Retrieval-Augmented Methods									
Vul-RAG	DeepSeek-V3	✗	50.5	58.3	3.4	0.4	6.5	2.5	1.0
VulInstruct	DeepSeek-V3	✗	**53.9**	55.8	**37.7**	**24.3**	**45.0**	**17.2**	**7.4**
Relative Improvement over best baseline			+0.9%	-6.8%	+50.8%	+138.2%	+32.7%	+32.3%	+7.2%

critically, VulInstruct achieves 37.7% recall, an improvement of 50.8% over GPTLens, demonstrating the superior ability to discover vulnerabilities. While fine-tuned VulTrial with GPT-4o achieves higher precision (59.9%), its recall remains limited at 20.8%, reflecting a high-confidence but low-coverage detection strategy. In contrast, VulInstruct maintains balanced performance with 55.8% precision while achieving 37.7% recall, demonstrating that VulInstruct can identify more vulnerabilities without sacrificing accuracy. Furthermore, we introduce the Unique metric which represents the percentage of vulnerabilities exclusively detected by that method among all 226 successfully detected CVEs across all systems. VulInstruct achieves 24.3% uniqueness, significantly outperforming all baselines. Notably, we analyze Vul-RAG and ReVD failures in Appendix C, revealing why retrieval-based and reasoning-enhanced approaches underperform.

VulInstruct also demonstrates superior capability in distinguishing vulnerable from patched code. In pairwise evaluation, VulInstruct achieves 17.2% P-C accuracy and 7.4% VP-S score, improving by 32.3% over the best baseline. This indicates our approach not only identifies vulnerabilities but understands how patches address underlying security issues.

Answer to RQ1: VulInstruct establishes new state-of-the-art with 45.0% F1-score (32.7% ↑) and 37.7% recall (50.8% ↑). The specification-guided approach significantly outperforms methods across all paradigms—prompting, fine-tuning, agents, and RAG—demonstrating the effectiveness of mining and applying security knowledge for vulnerability detection.

6.2 RQ2: What is the contribution of each module in VulInstruct?

To understand how different knowledge sources contribute to detection, we analyze VulInstruct under two complementary perspectives: knowledge utilization patterns in the full model and ablation studies where one source is entirely removed.

Knowledge utilization. For each input, VulInstruct retrains 10 relevant general knowledge (general specifications and detailed vulnerability cases) from General Specification Knowledge Base and 10 relevant CVE cases from the Domain Evidence Base. After Knowledge scoring, each test sample in vulnerability detection falls into one of four mutually exclusive categories: (i) both general knowledge and Domain-specific specifications left, (ii) only general knowledge left, (iii) only domain-specific specifications left, or (iv) no knowledge left. As shown in Table 2, most

Table 2. Ablation study and knowledge utilization in VulInstruct. The top part shows performance by subsets of test samples, grouped by retrieved knowledge sources (pair-wise metrics are omitted since subsets are not directly comparable). The bottom part reports ablation results where one source is entirely removed.

Variant / Subset	Ratio (%)	Acc.	Prec.	Rec.	F1	P-C	VP-S
Full VulInstruct (Deepseek-V3)	100.0	53.9	55.8	37.7	45.0	17.2	7.4
General + Domain-specific	73.5	55.5	58.4	41.2	48.3	–	–
General only	25.7	52.1	51.4	36.2	42.5	–	–
No knowledge matched	0.7	50.0	0.0	0.0	0.0	–	–
Domain-specific only	0.1	100.0	0.0	0.0	0.0	–	–
w/o Domain-specific specifications	–	50.6	50.9	33.4	40.4	14.2	1.0
w/o General specifications	–	51.7	52.6	33.7	41.0	16.2	3.7

samples (73.5%) belong to case (i), achieving the highest F1-score of 48.3%. Even when only general specifications or vulnerability detailed cases are left (25.7%), performance remains strong at 42.5% F1. In contrast, cases with only domain-specific specifications (0.1%) or no matched knowledge (0.7%) yield near-zero performance. These results indicate that two types of specifications are complementary: general specifications provide the essential detection capability, while domain-specific specifications enhance it when combined. For most samples, VulInstruct can retrieve at least one type of relevant knowledge, enabling more effective and reliable vulnerability detection.

Ablation. To further isolate contributions, we conduct ablation studies with entire knowledge sources removed. Removing domain-specific specifications causes F1-score to drop from 45.0% to 40.4% (−4.6%), while removing general specifications and detailed vulnerability cases reduces it to 41.0% (−4.0%). Interestingly, while domain-specific specifications appear in fewer samples (73.6% vs. 99.2% for general specifications and detailed vulnerability cases), their contribution to performance is slightly larger. This suggests that domain-specific specifications, when applicable, provide highly valuable domain-specific exploitation knowledge that complements the broader security knowledge captured in general knowledge. The VP-S metric shows a more dramatic difference: dropping from 7.4% to 1.0% without domain-specific specifications versus 3.7% without general knowledge, indicating its importance for distinguishing vulnerable from patched code.

Threshold analysis. To control experimental cost, we randomly sample 100 out of 435 vulnerability–patch pairs and vary the relevance threshold to study the trade-off between knowledge quality and coverage. Figure 4 illustrates this effect. On the left axis, F1-score follows an inverted-U curve: with a lenient threshold (3,3,3), more knowledge is preserved but the additional noise depresses performance (40.2% F1). At the other extreme (9,9,9), almost no knowledge is preserved (on average fewer than two items per type), leading to a sharp drop (31.1% F1). The optimal point occurs at (5,5,5), where sufficient high-quality knowledge is retained (15.4 items on average across both sources) and performance peaks at 43.4% F1. On the right axis, we also observe that the average number of general specifications, Domain-specific specifications, and detailed vulnerability cases items decreases monotonically as the threshold tightens. This confirms that higher thresholds improve knowledge precision but reduce coverage. Together, these results demonstrate that **more retrieved knowledge is not always better**; instead, an intermediate threshold is required to balance quality and quantity for effective retrieval-augmented detection.

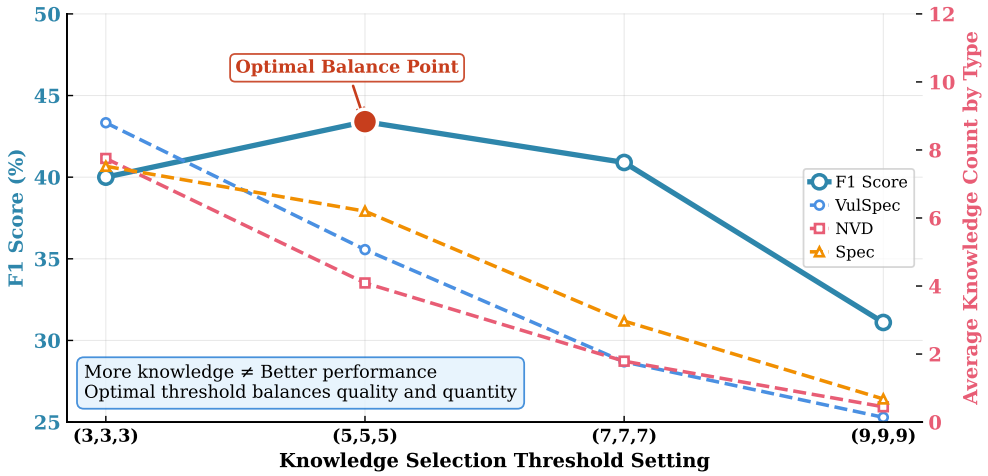


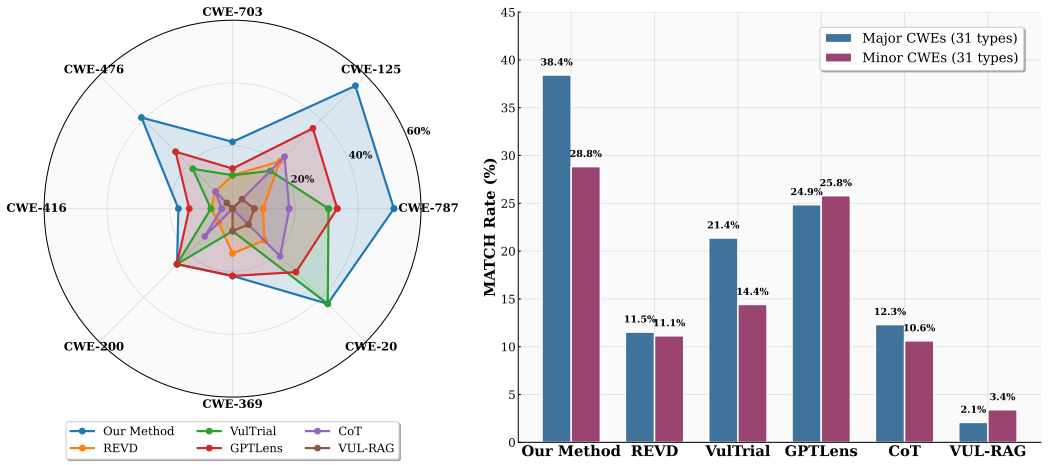
Fig. 4. Knowledge Selection Threshold Ablation: The inverted-U relationship in RAG-based vulnerability detection. *Spec* represents general specifications, *VulSpec* denotes corresponding detailed vulnerability cases, and *NVD* indicates CVE cases (which are subsequently transformed into domain-specific specifications).

Two types of security specifications are complementary for VulInstruct. Utilization analysis shows that combining them yields the highest performance (48.3% F1), while ablation confirms that removing either source consistently degrades results. Threshold experiments further reveal an inverted-U relationship between knowledge quality and coverage, with the optimal balance reached at (5,5,5). These findings highlight that effective retrieval-augmented detection requires not only diverse knowledge sources but also careful selection.

6.3 RQ3: How well does VulInstruct generalize across different vulnerability types and models?

Beyond overall performance, we examine VulInstruct's generalization across diverse vulnerability types and LLMs. Following the strict CORRECT evaluation, we define **MATCH rate** as the percentage of vulnerable samples where the model correctly identifies both the label and the root cause: MATCH rate = (correctly identified root causes) / (total vulnerable samples). This metric is important because simply predicting a "vulnerable" label does not ensure practical usefulness—security analysts need to know *why* the code is vulnerable. MATCH rate thus directly reflects the model's ability to discover and explain vulnerabilities rather than relying on superficial correlations.

Across CWE types. Figure 5a presents a radar chart comparing MATCH rates over the eight most frequent CWE categories in PrimeVul. VulInstruct consistently outperforms all baselines across diverse vulnerability types, including memory safety issues (e.g., CWE-125, CWE-787) and resource management flaws (e.g., CWE-416, CWE-476). While competing methods such as VulTrial and GPTLens exhibit strong performance only in certain categories, our approach achieves more balanced coverage. **Head vs. tail performance.** We evaluate performance on the long-tail distribution of vulnerability types, by ranking all 62 CWE types by their sample count in descending order. We then partition them into two equal groups: the top 31 CWE types (major/head group which contains 91.7% of all vulnerable samples) and the bottom 31 CWE types (minor/tail group with the left 8.3%). Figure 5b compares detection on major (head) CWEs and rare (tail) CWEs. VulInstruct attains the highest MATCH rate on both groups (38.4% on major CWEs and 28.8% on minor CWEs), outperforming GPTLens (25.8% and 24.9%) and VulTrial (21.4% and 14.4%). These



(a) CWE Radar Chart (b) Head/Tail Performance

Fig. 5. Comparison of two experimental results.

results indicate that VulInstruct not only improves detection of common vulnerabilities but also alleviates the long-tail challenge by transferring knowledge to rare vulnerability types.

Across models. Table 3 shows that **VulInstruct consistently improves different families of LLMs compared with their CoT approach** On GPT-OSS-120B, VulInstruct boosts recall by +72.9% and F1 by +45.4%, complementing its already strong precision. On Claude-Sonnet-4, the relative gains are even more striking: VP-S more than triples (from 2.2% to 7.3%) and P-C more than doubles, demonstrating that VulInstruct effectively enhances models with weaker intrinsic security awareness. Finally, DeepSeek-R1 with VulInstruct shows the strongest overall results, achieving the best P-C (22.8%), accuracy (56.6%), and precision (62.8%), establishing a new state of the art.

Table 3. Performance of VulInstruct across different language models on PrimeVul dataset. Improvement row represents relative improvement (%)

Model	Method	Acc. (%)	Prec. (%)	Rec. (%)	F1 (%)	P-C (%)	VP-S (%)
GPT-OSS-120B	Baseline	55.1	**70.5**	17.7	28.2	15.9	10.1
	+ VulInstruct	**56.1**	62.4	**30.6**	**41.0**	**17.6**	**11.8**
	Improvement	*+1.8% ↑*	*-11.5% ↓*	*+72.9% ↑*	*+45.4% ↑*	*+10.7% ↑*	*+16.8% ↑*
Claude-Sonnet-4	Baseline	51.3	52.1	32.0	39.6	3.2	2.2
	+ VulInstruct	**53.6**	**55.6**	**35.8**	**43.5**	**6.9**	**7.3**
	Improvement	*+4.5% ↑*	*+6.7% ↑*	*+11.9% ↑*	*+9.8% ↑*	*+115.6% ↑*	*+231.8% ↑*
DeepSeek-R1	Baseline	53.7	59.2	23.9	34.0	18.9	4.4
	+ VulInstruct	**56.6**	**62.8**	**32.2**	**42.6**	**22.8**	**13.7**
	Improvement	*+5.4% ↑*	*+6.1% ↑*	*+34.7% ↑*	*+25.3% ↑*	*+20.6% ↑*	*+211.4% ↑*

VulInstruct generalizes effectively across vulnerability types and base model. It achieves consistently higher MATCH rates across diverse CWE categories, improves both head and tail vulnerabilities to mitigate the long-tail challenge, and yields robust improvements on different LLMs. Notably, the combination with DeepSeek-R1 shows state-of-the-art performance across important metrics.

6.4 RQ4: What makes VulInstruct more effective than existing approaches?

We analyze VulInstruct's specification-guided approach using DeepSeek-R1. Overall, VulInstruct achieves a 16.7% improvement in detecting actual vulnerabilities (FN→TP) and an 8.0% reduction in false positives (FP→TN). To better understand how these improvements are achieved, we conducted manual analysis of successful cases, with detailed results provided in Appendix file Section D of the supplementary material. Here, we present one representative **case study**: CVE-2022-1533 is a buffer overflow vulnerability in a media decoder, where insufficient input validation on stereo DPCM audio streams leads to memory corruption during decoding. During detection, VulInstruct retained two general specifications and two detailed vulnerability cases from the general specification knowledge base. In addition, six relevant CVE cases were retrieved to induce three domain-specific specifications. Figure 6 highlights three types of knowledge. In particular, the domain-specific specification `AS-DECODE-001` and the detailed analysis of `HS-STATE` from CVE-2011-3951 significantly improved DeepSeek-R1's reasoning. `AS-DECODE-001` provided the basic guidance that all decoders must validate input bounds, while the "channel toggling" pattern from CVE-2011-3951 (`ch ^= stereo`) inspired the model to further track the interaction of `dir` and `pos`, revealing state changes that could trigger the overflow. By focusing on `pos` in depth, the model eventually reached a correct and faithful explanation.

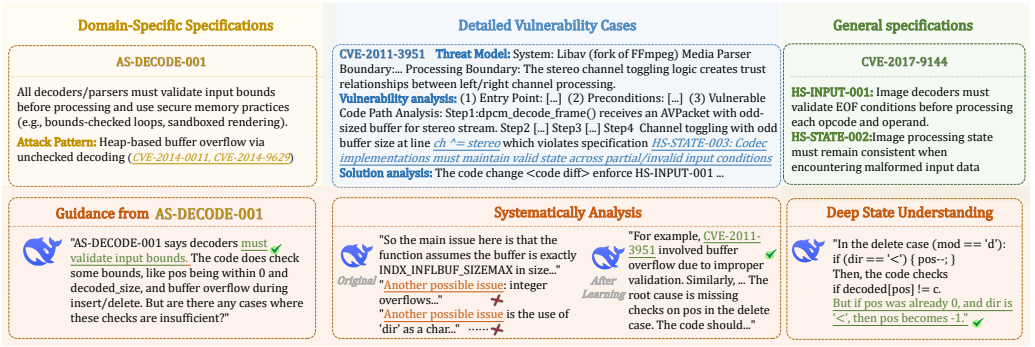


Fig. 6. Case Study in detecting CVE-2022-1533

> VulInstruct is more effective because specifications provide explicit guidance that helps models reason faithfully about the root cause of vulnerabilities, as shown in our case study.

7 Discussion

Potential for Real-World Vulnerability Discovery. We further investigate whether our approach can assist in uncovering real-world vulnerabilities. We conducted a preliminary study using our specification-guided vulnerability auditing framework. We simulate a manual vulnerability auditing workflow: Given the specification associated with a known vulnerability and vulnerable function, we construct an agent that automatically generates a small set of git grep commands to retrieve potentially relevant code segments for further inspection. The retrieved contexts are then analyzed by the agent under the guidance of the specification. Through this process, we successfully identified a high-severity logic vulnerability, which is essentially a variant of the original vulnerability and violates the same specification. We responsibly reported the vulnerability, and the issue was subsequently confirmed by developers, validated via integration testing, and patched. Detailed results are provided in our supplementary material (Section E in the Appendix file).

Nevertheless, we believe there remain at least two deeper directions to explore based on our work: **Richer use of NVD resources.** Current retrieval from NVD is still under-exploited. Each NVD entry typically contains a dynamically updated set of references, including such as patch lists, issue discussions, third-party advisories, developer-provided proof-of-concepts (PoCs), and crash reports. Such information could enrich the model's reasoning by offering concrete entry points and complete exploit paths, thereby enabling a more comprehensive understanding of vulnerabilities. **Improved contextual datasets.** We excluded PrimeVul's training dataset as its early version lacked CVE/CWE mappings and sufficient code context. This risks reducing understanding to single code parts within functions, insufficient for real-world vulnerability reasoning. Richer contextual datasets would significantly improve the quality of specification knowledge base.

8 Threats to Validity

Implementation validity. We did not identify apparent implementation errors in our study. To further validate this, we conducted targeted failure analyses of two representative baselines, Vul-RAG and ReVD. For Vul-RAG, we observed that its retrieval-based decision mechanism tends to prematurely classify samples as secure once a patch match is detected, which leads to a large number of missed vulnerabilities. For ReVD, we manually examined 50 vulnerable cases and found that reasoning failures often arise from zero reasoning, incorrect vulnerability categorization, incomplete mechanism understanding, or over-generalization. Detailed analyses are provided in our supplementary material (Section C in the Appendix file).

Experimental fairness. *The design choice of CWE.* Prior works like CORRECT and Vul-RAG include CWE types as input during detection, which simplifies the task to CWE-specific vulnerability identification. While this is fair within their experimental setup (all compared methods receive CWE information), it does not reflect realistic scenarios where vulnerability types are unknown. In our evaluation, we exclude CWE information from all methods, including VulInstruct and all baselines to ensure both experimental fairness and practical relevance. *The design choice of Dataset.* Furthermore, we did not select the SVEN [8] dataset, which is also a widely adopted high-quality dataset, for evaluation as it lacks the CVE-related metadata required by our evaluation framework.

Temporal validity. To prevent data leakage and ensure realistic evaluation, we enforce strict temporal constraints in our knowledge retrieval. For domain-specific specifications, we only retrieve CVEs whose publication dates and ID assignments precede the target vulnerability in test set. For General specifications, we verify all samples predate PrimeVul's test set cutoff. This temporal ordering ensures we simulate learning from historical vulnerabilities to detect future ones, avoiding the unrealistic scenario of using future knowledge to predict past vulnerabilities.

LLM-as-Judge reliability. Following CORRECT, we employ LLM-as-Judge to evaluate reasoning correctness. We acknowledge that using LLMs to judge other LLMs' outputs may introduce bias, though this approach captures reasoning correctness. To minimize such biases, we use GPT-5 as the judge model, which is distinct from all evaluated models in our experiments, thereby preventing self-evaluation bias where a model might favor its own reasoning patterns.

9 Conclusion

In this work, we introduced VulInstruct, a specification-guided approach that systematically mines security specifications from historical vulnerabilities to enhance vulnerability detection. By combining specifications extracted from patches and CVE records with recurring attack patterns across projects, VulInstruct enables large language models to reason and align with implicit expectations of safe behavior. Our extensive evaluation on the PrimeVul benchmark demonstrates substantial improvements in F1-score, recall, and pair-wise discrimination compared with state-of-the-art baselines. More importantly, VulInstruct has shown practical utility by discovering a previously

unknown high-severity vulnerability in real-world software. These results highlight the value of bridging implicit security expertise with automated analysis, opening new directions for integrating structured security knowledge into LLM-based vulnerability detection.

References

[1] Wei Chen, Bowen Zhang, Chengpeng Wang, Wensheng Tang, and Charles Zhang. 2025. Seal: Towards diverse specification inference for linux interfaces from security patches. In *Proceedings of the Twentieth European Conference on Computer Systems*. 1246–1262.

[2] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability Detection with Code Language Models: How Far Are We?. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 469–481.

[3] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A Survey on Code Generation with LLM-based Agents. arXiv:2508.00083 [cs.SE] https://arxiv.org/abs/2508.00083

[4] Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. 2024. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. *arXiv preprint arXiv:2406.03718* (2024).

[5] Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, et al. 2024. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *arXiv preprint arXiv:2406.11147* (2024).

[6] Google Project Zero. 2022. *2022 0-day In-the-Wild Exploitation...so far*. Technical Report. Google. https://googleprojectzero.blogspot.com/2022/06/2022-0-day-in-wild-exploitationso-far.html

[7] Jinyao Guo, Chengpeng Wang, Dominic Deluca, Jinjie Liu, Zhuo Zhang, and Xiangyu Zhang. 2025. BugScope: Learn to Find Bugs Like Human. *arXiv preprint arXiv:2507.15671* (2025).

[8] Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1865–1879.

[9] Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. 2023. Large language model-powered smart contract vulnerability detection: New perspectives. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 297–306.

[10] Xue Jiang, Yihong Dong, Zhi Jin, and Ge Li. 2024. SEED: Customize Large Language Models with Sample-Efficient Adaptation for Code Generation. arXiv:2403.00046 [cs.SE] https://arxiv.org/abs/2403.00046

[11] Sabrina Kaniewski, Fabian Schmidt, Markus Enzweiler, Michael Menth, and Tobias Heer. 2025. A Systematic Literature Review on Detecting Software Vulnerabilities with Large Language Models. *arXiv preprint arXiv:2507.22659* (2025).

[12] Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. 2025. {LLMxCPG}:{Context-Aware} Vulnerability Detection Through Code Property {Graph-Guided} Large Language Models. In *34th USENIX Security Symposium (USENIX Security 25)*. 489–507.

[13] Jia Li, Hao Zhu, Huanyu Liu, Xianjie Shi, He Zong, Yihong Dong, Kechi Zhang, Siyuan Jiang, Zhi Jin, and Ge Li. 2025. aiXcoder-7B-v2: Training LLMs to Fully Utilize the Long Context in Repository-level Code Completion. arXiv:2503.15301 [cs.SE] https://arxiv.org/abs/2503.15301

[14] Yue Li, Xiao Li, Hao Wu, Minghui Xu, Yue Zhang, Xiuzhen Cheng, Fengyuan Xu, and Sheng Zhong. 2025. Everything You Wanted to Know About LLM-based Vulnerability Detection But Were Afraid to Ask. *arXiv preprint arXiv:2504.13474* (2025).

[15] Yue Li, Xiao Li, Hao Wu, Yue Zhang, Xiuzhen Cheng, Yating Liu, Fengyuan Xu, and Sheng Zhong. 2025. If LLMs Would Just Look: Simple Line-by-line Checking Improves Vulnerability Localization. arXiv:2410.15288 [cs.CR] https://arxiv.org/abs/2410.15288

[16] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. arXiv:2405.17238 [cs.CR] https://arxiv.org/abs/2405.17238

[17] Miaoqian Lin, Kai Chen, and Yang Xiao. 2023. Detecting {API}{Post-Handling} bugs using code and description in patches. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3709–3726.

[18] Wang Lingxiang, Quanzhi Fu, Wenjia Song, Gelei Deng, Yi Liu, Dan Williams, and Ying Zhang. 2025. SAVANT: Vulnerability Detection in Application Dependencies through Semantic-Guided Reachability Analysis. *arXiv preprint arXiv:2506.17798* (2025).

[19] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *J. Syst. Softw.* 212, C (June 2024), 13 pages. doi:10.1016/j.jss.2024.112031

[20] National Institute of Standards and Technology. 2024. National Vulnerability Database. https://nvd.nist.gov/. Accessed: 2024-09-11.

[21] Chris Newman. 1999. *Using TLS with IMAP, POP3 and ACAP*. Technical Report. RFC Editor.

[22] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. 2024. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230* (2024).

[23] Niklas Risse, Jing Liu, and Marcel Böhme. 2025. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 388–410.

[24] Peter Saint-Andre and Jeff Hodges. 2011. *Representation and verification of domain-based application service identity within internet public key infrastructure using X. 509 (PKIX) certificates in the context of transport layer security (TLS).* Technical Report. RFC Editor.

[25] Ekaterina Shemetova, Ilya Shenbin, Ivan Smirnov, Anton Alekseev, Alexey Rukhovich, Sergey Nikolenko, Vadim Lomshakov, and Irina Piontkovskaya. 2025. LAMeD: LLM-generated Annotations for Memory Leak Detection. *arXiv preprint arXiv:2505.02376* (2025).

[26] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. 2024. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185* (2024).

[27] Karl Tamberg and Hayretdin Bahsi. 2025. Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study. *IEEE Access* 13 (2025), 29698–29717. doi:10.1109/access.2025.3541146

[28] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. arXiv:2312.12575 [cs.CR] https://arxiv.org/abs/2312.12575

[29] Shengye Wan, Joshua Saxe, Craig Gomes, Sahana Chennabasappa, Avilash Rath, Kun Sun, and Xinda Wang. 2024. Bridging the Gap: A Study of AI-based Vulnerability Management between Industry and Academia. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*. IEEE, 80–87.

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

[31] Zejun Wang, Jia Li, Ge Li, and Zhi Jin. 2023. ChatCoder: Chat-based Refine Requirement Improves LLMs' Code Generation. arXiv:2311.00272 [cs.SE] https://arxiv.org/abs/2311.00272

[32] Xin-Cheng Wen, Xinchen Wang, Yujia Chen, Ruida Hu, David Lo, and Cuiyun Gao. 2024. Vuleval: Towards repository-level evaluation of software vulnerability detection. *arXiv preprint arXiv:2404.15596* (2024).

[33] Xin-Cheng Wen, Yijun Yang, Cuiyun Gao, Yang Xiao, and Deheng Ye. 2025. Boosting Vulnerability Detection of LLMs via Curriculum Preference Optimization with Synthetic Reasoning Data. In *Findings of the Association for Computational Linguistics: ACL 2025*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 8935–8949. doi:10.18653/v1/2025.findings-acl.467

[34] Martin Weyssow, Chengran Yang, Junkai Chen, Ratnadira Widyasari, Ting Zhang, Huihui Huang, Huu Hung Nguyen, Yan Naing Tun, Tan Bui, Yikun Li, et al. 2025. R2Vul: Learning to Reason about Software Vulnerabilities with Reinforcement Learning and Structured Reasoning Distillation. *arXiv preprint arXiv:2504.04699* (2025).

[35] Ratnadira Widyasari, David Lo, and Lizi Liao. 2024. Beyond ChatGPT: Enhancing Software Quality Assurance Tasks with Diverse LLMs and Validation Techniques. arXiv:2409.01001 [cs.SE] https://arxiv.org/abs/2409.01001

[36] Ratnadira Widyasari, Martin Weyssow, Ivana Clairine Irsan, Han Wei Ang, Frank Liauw, Eng Lieh Ouh, Lwin Khin Shar, Hong Jin Kang, and David Lo. 2025. Let the Trial Begin: A Mock-Court Approach to Vulnerability Detection using LLM-Based Agents. *arXiv preprint arXiv:2505.10961* (2025).

[37] Yixin Yang, Bowen Xu, Xiang Gao, and Hailong Sun. 2025. Context-enhanced vulnerability detection based on large language model. *arXiv preprint arXiv:2504.16877* (2025).

[38] Yanjing Yang, Xin Zhou, Runfeng Mao, Jinwei Xu, Lanxin Yang, Yu Zhang, Haifeng Shen, and He Zhang. 2025. Dlap: A deep learning augmented large language model prompting framework for software vulnerability detection. *Journal of Systems and Software* 219 (2025), 112234.

[39] Alperen Yildiz, Sin G Teo, Yiling Lou, Yebo Feng, Chong Wang, and Dinil M Divakaran. 2025. Benchmarking LLMs and LLM-based Agents in Practical Vulnerability Detection for Code Repositories. *arXiv preprint arXiv:2503.03586* (2025).

[40] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2024. Prompt-Enhanced Software Vulnerability Detection Using ChatGPT. arXiv:2308.12697 [cs.SE] https://arxiv.org/abs/2308.12697

[41] Kechi Zhang, Huangzhao Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023. ToolCoder: Teach Code Generation Models to use API search tools. arXiv:2305.04032 [cs.SE] https://arxiv.org/abs/2305.04032

[42] Xin Zhou, Ting Zhang, and David Lo. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. arXiv:2401.15468 [cs.SE] https://arxiv.org/abs/2401.15468

A CORRECT Evaluation Framework

The CORRECT evaluation introduces rationale-based assessment to verify whether models identify vulnerabilities for the correct reasons: The rationale refers to the model's step-by-step reasoning process and results that lead to its vulnerability detection conclusion. CORRECT evaluates this rationale against ground-truth vulnerability information (CVE descriptions, patches, and commit messages) to determine correctness: The framework operates as follows:

(1) **For vulnerable code**: The model must correctly identify the ground-truth vulnerability in its rationale. If the rationale includes the actual vulnerability cause or key elements in vulnerabilities, it is deemed correct; otherwise, the detection is considered a false negative despite potentially correct binary labeling.
(2) **For patched code**: Two evaluation modes are employed:
   - *Lenient Mode*: Accepts the model's results if it either correctly identifies the code as non-vulnerable, or if it reports a vulnerability but the rationale does not reference the original (now patched) vulnerability.
   - *Strict Mode*: This mode extends Lenient Mode by adding an explicit iterative check. When the model flags a patched function as vulnerable but does not reference the ground-truth vulnerability, the framework instructs the model to disregard the previously reported issues and re-evaluate whether the function still contains a vulnerability. If during this iterative process the model again reports the ground-truth vulnerability, the case is marked as a false positive, since the model fails to recognize that the vulnerability has already been fixed.

These scenarios are summarized in Table 4, which contrasts the evaluation outcomes of Lenient and Strict modes across different prediction cases. In practice, this rationale assessment is implemented by another LLM acting as a judge, which takes the model's explanation as input and decides whether it correctly reflects the ground-truth vulnerability.

Table 4. Evaluation outcomes under Lenient and Strict modes for different prediction scenarios.

Ground Truth	Prediction	Rationale	Lenient	Strict
Vulnerable	Vulnerable	Correct	TP	TP
Vulnerable	Vulnerable	Incorrect	FN	FN
Vulnerable	Non-vulnerable	–	FN	FN
Patched	Non-vulnerable	–	TN	TN
Patched	Vulnerable	References original vuln	FP	FP
Patched	Vulnerable	Other issues only	TN	Feedback→TN/FP

Choice of Evaluation Mode. While CORRECT provides both Lenient and Strict modes, we adopt Lenient Mode in our evaluation based on empirical evidence and practical considerations.

Empirical results in the CORRECT paper, including extensive statistics reported in its appendix, show that the benefits of *Strict Mode* are marginal: only 10.3% of cases are modified after the first feedback round, dropping to 3.6% in round 2, and merely 1.5% by round 4. This diminishing return suggests that the iterative refinement process yields small improvements while significantly increasing computational costs.

In addition, applying *Strict Mode* to existing methods is particularly problematic. Training-based approaches such as **ReVD**, and multi-agent frameworks such as **VulTrial**, rely on carefully designed instructions. Forcing these models to ignore previously reported vulnerabilities and repeatedly

re-analyze the code disrupts their prompt structures, often leading to poor performance. Supporting such feedback loops would require retraining models from scratch with feedback-aware objectives, which introduces substantial overhead in both data construction and training optimization.

Given these issues, we adopt *Lenient Mode* for all evaluations. This ensures a fair comparison across baselines while still keeping evaluation rigor through rationale correctness assessment.

B Prompt Template.

Prompt: System Prompt for General Specifications

A structured threat modeling analysis process where security experts conduct systematic security analysis based on provided information. The expert must:
1. Understand Code Context (within `<understand>` tags)
Thoroughly analyze and describe the system context without revealing the vulnerability itself:
System Identification
- **What system**: Clearly identify the software system, library, or application
- **Domain/Subsystem**: Specify the particular domain or subsystem where the code operates
- **Module/Component**: Identify the specific module, component, or functional unit

Functional Analysis
- **Core functionality**: Describe what this system/module is designed to do in detail: 1. 2. 3.

2. Security Domain Classification (within `<classification>` tags)
Classify vulnerabilities according to 10 core security domains:
Core Security Domains:
(1) **MEM**: Memory Safety [Buffer errors, pointer issues, use-after-free, allocation problems, etc.]
(2) **STATE**: State Management [Inconsistent states, object lifecycle, concurrency issues, etc.]
(3) **INPUT**: Input Validation [Parsing logic, data validation, type checking, encoding, etc.]
(4) **LOGIC**: Program Logic [Arithmetic errors, type confusion, logical mistakes, etc.]
(5) **SEC**: Security Features [Authentication, cryptography, permissions, policy enforcement]
(6) **IO**: I/O Interaction [Filesystem operations, networking, device interaction, etc.]
(7) **CONF**: Configuration Environment [Configuration parsing, environmental variables, etc.]
(8) **TIMING**: Timing & Concurrency [Race conditions, synchronization issues, TOCTOU, etc.]
(9) **PROTOCOL**: Protocol Communication [Message parsing/formatting, session handling, etc.]
(10) **HARDWARE**: Hardware & Low-level [Low-level interfaces, architectural specifics, etc.]

3. Security Specification (within `<spec>` tags)
Security Specification helps understand how vulnerable code violates developer's original expectations and how patches implement fixes.
Security Specification Requirements:
- Each security specification includes a unique identifier (`HS-<DOMAIN>-<NNN>`) in classification results
- Use a **positive action** structure (describe what must be done, not what went wrong)
- For simple vulnerabilities, limit to 2 or fewer specifications, focusing on essential ones
- Ensure traceability to specific vulnerability description or repair commit
- Focus on underlying semantic knowledge and domain rules — e.g., protocol constraints, business logic invariants, or program semantics

Reasoning Pattern Examples:
1. Reverse Reasoning Pattern
State Consistency Specification:
- Vulnerability phenomenon → "XML parser loads external entities in non-validating mode"
- Positive requirement → "Parser behavior must remain consistent with configuration state"
- Security Specification: `HS-STATE-001`: Parser operation privileges must be strictly constrained by validation mode state

2. Causal Analysis Pattern
Pointer Lifecycle Specification:
- Logic issue → "Freed pointer not nullified causing dangling pointer"
- Root requirement → "Pointer release operations must atomically complete state cleanup"
- Security Specification: `HS-MEM-002`: Pointer release must include immediate nullification forming atomic unit

3. Domain Mapping Pattern
Protocol Boundary Specification:
- Domain concept → "HTTP message integrity boundary"
- Security property → "Content-Length must precisely match actual data"
- Security Specification: `HS-PROTOCOL-003`: Protocol implementation must enforce byte-level consistency validation between message headers and body

4. Defensive Design Pattern
Input Normalization Specification:
- Vulnerability source → "Non-canonical input bypasses validation logic"
- Design principle → "All external input must undergo standardization processing"
- Security Specification: `HS-INPUT-004`: Input processing pipeline must implement pre-parse normalization and boundary checking

---

Example:
Input Information
- **Repository**: ksmbd
- **Commit Message**: ksmbd: Fix dangling pointer in krb_authenticate
- **CVE Description**: In the Linux kernel, the following vulnerability has been resolved: ksmbd: Fix dangling pointer in krb_authenticate...
- **CWE Type**: CWE-416 (Use After Free)

Code Diff:

```
-if (sess->state == SMB2_SESSION_VALID)
+if (sess->state == SMB2_SESSION_VALID) {
     ksmbd_free_user(sess->user);
+    sess->user = NULL;
+}
```

Expected Output Format:
<understand>
System Identification
- **What system**: ksmbd - in-kernel SMB server implementation for Linux
- **Domain/Subsystem**: SMB/CIFS network file sharing protocol implementation
- **Module/Component**: Kernel component receives SMB requests, uses netlink IPC...

</understand>
<classification>
<primary>MEM.LIFECYCLE</primary>
<tags>[STATE.CONSISTENCY, SEC.AUTHENTICATION, PROTOCOL.SMB]</tags>
<reasoning>The root cause is the failure to manage the lifecycle...</reasoning>
</classification>
<spec>HS-MEM-001: Pointer release operations require atomic cleanup with immediate nullification</spec>
- Reasoning: Dangling pointer vulnerability → freed but not nullified → atomic release-nullification prevents use-after-free

---

Current Analysis Target:
Repository: {repository}
Commit Message: {commit_message}
CVE Description: {cve_description}
CWE Type: {cwe_type}
Vulnerable Code:

```
{vuln}
```

Solution:

```
{fixed}
```

Please conduct analysis following the above format.

Prompt: Detailed Vulnerability Cases in General Specifications

A structured threat modeling analysis process where security experts conduct systematic security analysis based on provided information.

Analysis Framework

1. System Understanding (provided context)

`{understand}`

2. Security Specifications (provided rules)

`{specification}`

3. System-Level Threat Modeling (within `<model>` tags)

Analyze vulnerability at system design level:

- **Trust Boundaries**: Identify where system components transition between trusted/untrusted states
- **Attack Surfaces**: Focus on realistic attack vectors that led to this specific vulnerability
- **CWE Analysis**: Trace complete vulnerability chain (e.g., initial CWE-X triggers subsequent CWE-Y, where at least one matches: `{cwe_type}`)

4. Code-Level Analysis

Vulnerability Context (within `<vuln>` tags)

Provide a granular, narrative explanation of the vulnerability:

(1) **Entry Point & Preconditions**: Describe how the attack is initiated and what system state is required
(2) **Vulnerable Code Path Analysis**: Step-by-step trace of execution flow, naming key functions and variables. Pinpoint **The Flaw** and its **Consequence**
(3) **Specification Violation Mapping**: Link code path steps to specific `HS-` specifications they violate

Fix Implementation (within `<solution>` tags)

Explain how the patch enforces security specifications:

- Specific code changes and their security impact
- How fixes restore compliance with violated specifications

Example Output Format:

`<model>`

- **trust_boundaries**: User-Kernel boundary during SMB2 session setup; Intra-kernel function contract violation
- **attack_surfaces**: Malicious SMB2 SESSION_SETUP request; Error path exploitation
- **cwe_analysis**: Primary CWE-416 (Use After Free) enabled by state management violation

`</model>`

`<vuln>`

(1) **Entry Point**: Privileged user sends Netlink message with crafted CIPSOV4 tags
(2) **Code Path**: Loop processes tags → **The Flaw**: Off-by-one error in bounds check → **Consequence**: Stack buffer overflow
(3) **Violations**: `HS-MEM-001` (incorrect bounds check), `HS-STATE-002` (incomplete initialization)

`</vuln>`

`<solution>`

Change 1: Bounds Check Correction

```
-if (iter > CIPSO_V4_TAG_MAXCNT)
+if (iter >= CIPSO_V4_TAG_MAXCNT)
```

Compliance: Changes exclusive to inclusive comparison, preventing array overflow

Change 2: Complete Array Initialization

```
-doi_def->tags[iter] = CIPSO_V4_TAG_INVALID;
+while (iter < CIPSO_V4_TAG_MAXCNT)
+    doi_def->tags[iter++] = CIPSO_V4_TAG_INVALID;
```

Compliance: Ensures all array elements initialized to safe values

`</solution>`

Input Information:

- **CVE**: `{cve_description}`

- **CWE**: `{cwe_type}`
- **Commit**: `{commit_message}`
- **Vulnerable Code**: `{vuln}`
- **Fixed Code**: `{fixed}`
- **Code Context**: `{code_context}`

Please conduct analysis following the above framework.

Prompt: VulInstruct Knowledge Scoring Mechanism

You are a security expert. Please evaluate the relevance between the following code and VulInstruct vulnerability cases.

Target Code

`{code_snippet}`

VulInstruct Cases to Evaluate

`{chr(10).join(cases_for_evaluation)}`

Please score the relevance of each case to the target code (1-10 points):

Scoring Criteria:

- **10 points**: Highly relevant, vulnerability type, trigger conditions, and code patterns are almost identical
- **8-9 points**: Strong relevance, main vulnerability features are similar, can provide valuable reference
- **6-7 points**: Moderate relevance, some features are similar, has certain reference value
- **4-5 points**: Weak relevance, only few similarities
- **1-3 points**: Very low relevance, basically no reference value

Please strictly follow the HTML format for output:

```
<vulinstruct_evaluation>
<case_1_score>6</case_1_score>
<case_1_reasoning>Scoring reason</case_1_reasoning>
<case_2_score>8</case_2_score>
<case_2_reasoning>Scoring reason</case_2_reasoning>
...
</vulinstruct_evaluation>
```

Prompt: Domain-specific Specification Extraction

You are a security expert. Analyze these related vulnerabilities and extract reusable security specifications.

Related Historical Vulnerabilities

`{chr(10).join(nvd_descriptions)}`

Task: Extract Attack-Derived Specifications

For each vulnerability pattern you identify:

(1) **Identify the recurring attack mechanism** across these CVEs
(2) **Convert it to a positive security specification** that would prevent such attacks
(3) **Format as defensive requirements** developers must implement

Output Format:

```
<attack_specifications>
  <specification_1>
    <attack_pattern>
      Description of recurring attack mechanism
      in cve-xxx and cve-xxx in detail
    </attack_pattern>
    <defensive_spec>
      AS-DOMAIN-001: Security rule that describes
      the code behavior that prevents this attack
    </defensive_spec>
    <implementation_hint>
```

```
      Specific checks or validations needed
    </implementation_hint>
  </specification_1>
  <specification_2>...</specification_2>
</attack_specifications>
```

Prompt: Vulnerability Detection

You are a senior code security expert. Please perform systematic multi-layer security analysis on the following code.
Analysis Mode: *[Determined by knowledge relevance scoring]*

- **Autonomous Analysis**: Low relevance with knowledge base, perform independent analysis
- **Knowledge-Assisted**: High relevance knowledge filtered through LLM evaluation as reference

Input Components:

- Code Snippet: `{code_snippet}`
- Code Context: `{code_context}`
- LLM-filtered Security Knowledge: `{selected_knowledge}`

Multi-Layer Vulnerability Analysis Framework
1. Surface Symptom Analysis Identify direct suspicious operations.
2. Root Cause Investigation Trace deeper causes that give rise to the surface symptoms, focusing on data/control flow, completeness of input validation, adequacy of error handling, and potential attacker exploitation paths.
3. Architectural & Contextual Analysis Examine broader design-level factors and domain-specific assumptions in the application logic.
Comprehensive Security Assessment. Based on the above LLM-filtered Security Knowledge and three-layer analysis mode framework, please provide your professional judgment:
(i) Analysis Process: [Please describe your three-layer analysis process in detail, including discovered issues and reasoning chains]
(ii) Key Findings: [List the most important security findings]
(iii) Final Conclusion:
Please strictly follow the format below for output:
Output Format:

```
<vulnerability_assessment>\\
Please strictly follow the format below for output:
  <has_vulnerability>yes/no</has_vulnerability>
  <confidence>0-1</confidence>
  <suspected_root_cause>Core findings summary</suspected_root_cause>
</vulnerability_assessment>
```

Format Description:

- `has_vulnerability`: "yes" or "no"
- `confidence`: Confidence level between 0.0 and 1.0
- If a fixing solution has been applied, you may judge "no"
- Focus on analysis quality, avoid over-sensitivity

C Failure Analysis in Vul-RAG and ReVD

We conduct a targeted analysis of the key performance factors in three representative approaches: the retrieval mechanism in Vul-RAG, the reasoning capability in ReVD, and the specifications in VulInstruct.

Failure Analysis of Vul-RAG. Vul-RAG conducts vulnerability detection in an iterative retrieval manner. Given a target code snippet, the model first retrieves the top-k most relevant vulnerability–patch cases ($k = 10$ in our experiments). For each case, it checks (i) whether the same type of vulnerability exists in the target code and (ii) whether the corresponding patch has already been applied. The outcome is a binary signal: $(1, 0)$ indicates a vulnerability without a patch (*classified as vulnerable*), $(0, 1)$ indicates a patch without vulnerability (*classified as secure*), while $(0, 0)$ or $(1, 1)$

Table 5. Failure analysis of baseline methods. Left: iteration distribution of Vul-RAG. Right: categorized failure cases of ReVD.

Iteration Range	Ratio	Vuln.	Secure
1–3	53.1%	12.5%	87.5%
4–6	22.6%	18.0%	82.0%
7–9	6.2%	27.5%	72.5%
10 (final)	5.4%	50.0%	50.0%
10 (no decision)	12.7%	0.0%	100%

(a) Vul-RAG iteration distribution where Vuln and Secure represents the final binary prediction.

Failure Type	Ratio
Zero reasoning	4%
Wrong vulnerability type	30%
Similar type confusion	20%
Mechanism misinterpretation	14%
Over-generalization	26%
Other errors	6%

(b) ReVD failure categories.

are inconclusive and trigger the next iteration. The process continues until a conclusive decision is made; if no decisive signal is found after all iterations, the output is treated as *no decision*. As shown in Table 5, over half of the samples (53.1%) terminate within the first three iterations, while 12.7% end without any decision.

As shown in Table 5(a), over half of the samples (53.1%) terminate within three iterations, while 12.7% never reach a decision. A common failure is prematurely classifying samples as secure once a single patch match is observed, even if other vulnerabilities remain. This reliance on cross-project case matching proves unreliable: few vulnerabilities recur in exactly the same form, resulting in only 3.4% reasoning correctness under CORRECT.

Failure Analysis of ReVD. We manually analyzed 50 vulnerable cases where ReVD produced the correct binary label but failed in reasoning. Four recurring error types emerged (Table 5(b)): **(i) Zero reasoning** where no explanation was provided, **(ii) Incorrect vulnerability categorization** with either wrong CWE families or confusion between closely related types, **(iii) Incomplete mechanism understanding** where the type was identified but the related code or the exploitation results was wrong, and **(iv) Over-generalization** where only vague statements like code quality concerns replaced specific root causes. Although ReVD leverages large-scale reasoning data, its generated explanations often lack the precision and depth which highlights the key limitation of reasoning-distillation approaches: they improve surface-level consistency but do not guarantee faithful identification of vulnerability root causes.

D Manual analysis of successful cases in VulInstruct

We analyze VulInstruct's specification-guided approach using DeepSeek-R1, examining how specifications enhance vulnerability detection. The method achieves 16.7% improvement in detecting actual vulnerabilities (FN→TP) and 8.0% reduction in false positives (FP→TN). To understand these improvements, we randomly sampled 10 cases from each category and performed a manual analysis, identifying four distinct repair mechanisms through which specifications enhance detection capabilities. Table 6 shows four recurring *repair mechanisms*: **(i) Missing Security Dimension (20% of cases)** which occurs when models lack awareness of entire vulnerabilities. For example, in CVE-2021-37848, the model only checked `strncmp` for logical errors. Our specification addressed this by mandating constant-time comparisons for security-sensitive operations. **(ii) Domain-Specific Blindness (30% of cases)** occurs when models detect potential defects but underestimate their severity in specific contexts. For instance, in CVE-2022-24214, the model identified an integer overflow in DNS code but classified it as low risk, failing to recognize that DNS TXT records are attacker-controlled and high-risk vectors. Our domain specification AS-DOMAIN-1 provided this

essential context, upgrading the assessment from "possible issue" to "high-severity vulnerability." **(iii) Deep Reasoning Enhancement (25% of cases)** improves models' ability to connect isolated risks into complete analysis of vulnerability mechanism. We provide a detailed case study in Figure 6. **(iv) Secure Pattern Validation (25% of cases, exclusively FP→TN)** enables models to recognize secure implementations and avoid false positives. For example, in CVE-2022-21654, the model mistakenly flagged the use of `EVP_sha256` as insecure. Our specification HS-CRYPTO-003 confirmed this as correct cryptographic practice, reinforcing that knowledge of secure patterns is as critical as vulnerability detection.

Table 6. Repair mechanisms by which specifications improve vulnerability detection

Repair Mechanism	FN→TP	FP→TN	Total	Primary Knowledge Sources
Missing Security Dimension	4	0	4 (20%)	General + Domain-specific
Domain-Specific Blindness	3	3	6 (30%)	Domain-specific + Detailed cases
Deep Reasoning Enhancement	3	2	5 (25%)	Domain-specific + Detailed cases
Secure Pattern Validation	0	5	5 (25%)	General + Detailed cases

E Using VulInstruct Finding real-world vulnerability

Case Study: From CVE-2021-32056 to a New Access Control Bypass. We provide a detailed case study to illustrate how our specification-guided auditing framework can facilitate the discovery of new real-world vulnerabilities. Starting from CVE-2021-32056 in the Cyrus IMAP Server (`cyrusimap/cyrus-imapd`), a randomly selected vulnerability case from the CORRECT dataset, we successfully identified and reported a previously unknown high-severity flaw in the same codebase. CVE-2021-32056 allowed authenticated users to bypass intended access restrictions on server annotations, leading to potential replication stalls and service disruptions. The root cause was an improperly scoped permission check in `imap/annotate.c`, where the `maywrite` check was nested inside a conditional block and skipped when the mailbox pointer was `NULL`. From this case, our framework distilled three reusable security specifications, including the specification HS-SEC-001 (annotation write operations must always enforce strict privilege-based access control).

Guided by this specification, we constructed an auditing agent workflow that simulated the workflow of a manual security audit: starting from the known flaw, generalizing its pattern, and searching the codebase for other instances where security checks might be incorrectly scoped.

Our workflow first prompted a LLM to generate candidate `git grep` commands that could reveal similar patterns elsewhere in the repository. We employed Gemini-2.5-Pro, whose relatively large context window allowed us to supply extracted code fragments without requiring a more elaborate dynamic windowing design. The generated commands reflected the model's reasoning about how the extracted specification might be violated in other parts of the codebase. In particular, the model focused on core database operations such as `store` and `delete`, which are security-critical under HS-SEC-001, and combined them with contextual information from the repository's application context (e.g., session management, mailbox operations). In doing so, the model hypothesized potential validation points where access control checks might be inconsistently applied. The next stage of our workflow was to feed the retrieved code snippets back into the model for analysis under the extracted specifications. This allowed the model to reason about whether the conditional checks in each match were relevant to security enforcement. Among four generated queries produced, one was especially effective, as shown below:

```
git grep -p --all-match \
```

```
  -e 'if (mailbox)' \
  -e 'cyrusdb_store\|cyrusdb_delete' \
  -- '*.c'
```

This query rediscovered the already patched `write_entry` function in `annotate.c`, thereby validating the search strategy, while simultaneously surfacing additional candidate sites. Furthermore, the model dismiss some false positives, such as matches in `imap/tls.c`, where the conditionals guarded resource management logic rather than access control, and therefore did not violate HS-SEC-001. At the same time, the model highlighted `imap/mboxlist.c` as a high-risk location, noting that several conditional branches in this file surrounded critical database operations such as `cyrusdb_store` and `cyrusdb_delete`. To further investigate, we selected this candidate and applied our Automatic Context Extraction Tool to retrieve the surrounding function bodies together with a depth-3 call chain, ensuring that the model could reason about how access control checks were propagated across related functions. Given this enriched context, the model conducted a more detailed analysis and identified problematic control-flow paths. Guided by this assessment, we performed a closer inspection of the extracted functions—`mboxlist_update`, `mboxlist_update_entry_full`, and `mboxlist_renamemailbox`—which ultimately led to the discovery of a severe privilege bypass. The most critical finding emerged in `mboxlist_renamemailbox`, a function with complex multi-path control flow. Here, we discovered that a `goto` statement transferred execution directly to the database update section, bypassing any privilege checks:

```
if (mbentry->mbtype & MBTYPE_INTERMEDIATE) {
    // ... destination checks ...
    goto dbupdate; // BYPASSES permission check!
}

myrights = cyrus_acl_myrights(auth_state, mbentry->acl);
if (!isadmin && !(myrights & ACL_DELETEMBOX)) {
    return IMAP_PERMISSION_DENIED;
}
```

As a consequence, an authenticated user without `ACL_DELETEMBOX` rights could still rename intermediate mailboxes. This represents a direct violation of HS-SEC-001: security checks were present but misplaced, enabling a privilege bypass. Conceptually, this flaw mirrors CVE-2021-32056—the same fundamental security specification was violated—but here the error manifested through control-flow misordering (`goto`) rather than conditional scoping.

The potential consequences were significant: unauthorized mailbox renaming could disrupt shared mailbox hierarchies, shift shared folders into private namespaces, and cause denial-of-service for legitimate users. We responsibly disclosed the issue to the Cyrus IMAP development team, who confirmed the vulnerability, reproduced it via integration tests, and patched it by relocating the permission check before the special-case handling logic. We ultimately assigned CVE-2025-56538 to this vulnerability, though the detailed information will not be made public during the review process.

F Automatic Context Extraction Tool

We implement a unified Commit URL parser that normalizes repository and patch commit metadata across heterogeneous hosts. Using a small set of hand-written matching rules, the parser recognizes both modern and legacy interfaces (e.g., GitHub, GitLab, Bitbucket, as well as cgit/gitweb deployments such as kernel.org and GNU Savannah), and deterministically maps each commit URL to a canonical triple {`repo_name`, `commit_hash`, `clone_url`}. For example, kernel.org cgit links are remapped to stable GitHub mirrors to enable uniform downstream handling.

Building on this, we resolve—when available via the host's API—the canonical parent repository for each commit, thereby obtaining the repositories for both the patched and vulnerable versions. Following CORRECT, we derive target functions from the change lines and then use `joern` and `cflow` to extract the four context types described in Section 4.1.1. For large repositories (e.g., Linux, TensorFlow), we further augment the default flow with lightweight subsystem/component identification and module-boundary detection, guided by directory structure cues together with staged filtering (includes, calls, symbol-to-file mapping) and shallow depth limits. These heuristics make CPG construction incremental and tractable, while avoiding uncontrolled expansion.

Building on this, we resolve—when available via the host's API—the canonical parent repository for each commit, thereby obtaining the repositories for both the patched and vulnerable versions. Following CORRECT, we derive target functions from the change lines and then use `joern` and `cflow` to extract the four context types described in Section 4.1.1. For large repositories (e.g., Linux, TensorFlow), we further augment the default flow with lightweight subsystem/component identification and module-boundary detection, *seeded by a small set of manually curated anchors* (Linux: `net/fs/kernel`; TensorFlow: `core/python/compiler`) and simple boundary rules (`find_module_root`, `is_module_boundary`). The analysis employs staged filtering (header-include analysis → call-graph edges → symbol-to-file mapping) and adaptive file selection (targets+module, callers/callees, include neighbors, paired `.h`/`.c`), together with conservative caps (up to 3,000 files; depth ≤ 3) and graceful timeouts/recovery. These heuristics keep CPG construction incremental and tractable while preventing uncontrolled expansion.

Our final preprocessed dataset provides substantially richer context than PrimeVul, which only supplies the full file in which a vulnerable function resides. In contrast, our dataset captures the four distinct types of context described in Section 4.1.1, thereby enabling more fine-grained program analysis and downstream tasks.