CE 815 – Secure Software Systems

Modern Vulnerability Detection Methods (VulInstruct)

Mehdi Kharrazi
Department of Computer Engineering
Sharif University of Technology

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

Specification-Guided Vulnerability Detection with Large Language Models, Zhu H., Li J., Gao C., Qian J., Dong Y., Liu H., Wang L., Wang Z., Hu X., Li G., Arxiv, Nov 2025.

Introduction

- Recent studies have explored using LLMs for automated vulnerability detection.
- Although their performance on vulnerability detection benchmarks remains unsatisfactory.
- Authors 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.

Key Insight

- Historical vulnerabilities implicitly encode the security specifications defined by developers and security teams.
- Every vulnerability can be traced to the violation of at least one underlying security specification.
  - Examples of violated expectations: state consistency, pointer lifecycle, protocol boundary enforcement.
- Security specifications are framed as a unifying representation of implicit expert knowledge, interpretable to humans, and transferable across projects.
- VulInstruct extract[s] reusable security specifications from historical vulnerabilities and uses them to instruct the detection of new ones.

Approach

- Two automatic pipelines to construct a specification knowledge base:
- **General security specifications**: extracted from high-quality patch datasets.
  - By comparing vulnerable code with its fixed version and restating the underlying expected behaviors as explicit, reusable specifications.
- **Domain-specific security specifications:** extracted from our comprehensive CVE database.
  - Derived from frequently exploited vulnerabilities within the same repository or related projects
- Two types are complementary: general specifications provide broad coverage while domain-specific specifications capture context-aware expectations.

Learning General Specifications

- Hostname verification in TLS has long-standing requirements (RFC 2595; RFC 6125).
- These requirements rarely appear in project documentation and remain security experts' implicit knowledge base.
- CVE-2013-4488 (libgadu): retrieved/logged the server certificate but failed to verify it against the target hostname, enabling impersonation and undermining TLS security.
- Automated Specification Learning:
  - From the patch for CVE-2013-4488, automatically extracts the underlying security principles.
  - HS-SEC-001: TLS implementations must perform complete certificate chain validation including hostname verification.
  - HS-PROTOCOL-002: TLS handshakes must enforce strict verification of all X.509 certificate fields.

Case Study: TLS Hostname Verification

- Specification-guided Detection
  - Extracted specifications become reusable knowledge.
  - When analyzing e2guardian (CVE-2021-44273), retrieve HS-SEC-001 and detected a similar omission in Socket::startSslClient() with no hostname verification.
  - The system flaggs the violation, explains impact, and this shows specs learned from patches can generalize across projects.

(a) Learning Security Specifications from CVE-2013-4488 (libgadu)

Vulnerable Code

```
// gg_handle_tls_negotiation()
SSL_connect(ssl);
peer = SSL_get_peer_certificate(ssl);
// Extract CN for logging only
if(X509_NAME_get_text_by_NID(X509_get_sub
ject_name(peer), NID_commonName, buf,
sizeof(buf)) == -1)
    // ✗ NO hostname verification!
    buf[0] = 0;
// Just logs the CN without checking
gg_debug_session("CN=%s", buf);
// Connection proceeds without validation
```

Fixed Code:

```
// gg_handle_tls_negotiation()
SSL_connect(ssl);
peer = SSL_get_peer_certificate(ssl);
// ☑ Proper hostname verification added
valid_hostname = verify_hostname(peer, sess->
connect_host);
if (!valid_hostname){
    gg_debug_session("Hostname mismatch!");
    e->event.failure = GG_FAILURE_TLS;
    return GG_ACTION_FAIL;
}
// Only proceed if hostname matches
```

Learning from Patch

HS-SEC-001: TLS implementations must perform complete certificate chain validation including hostname verification against the target server.

HS-PROTOCOL-002: TLS handshake implementations must enforce strict verification of all X.509 certificate fields including subjectAltName

(b) Knowledge-Guided Detection of CVE-2021-44273 (e2guardian)

Reasoning Process in Target Code Analysis: e2guardian SSL Client

I am currently analyzing the function `Socket::startSslClient()` in e2guardian v5.4.x version, which is responsible for initiating an SSL/TLS connection. The key code in this function is as follows: It use `SSL_set_fd(ssl, this->getFD());` binds the SSL object to the socket's file descriptor. This connects the SSL/TLS layer to the underlying network socket, enabling the secure communication setup ......

Reasoning Process in Target Code Analysis in Specification-Guided Vulnerability Detection

Security specification HS-SEC-001 and HS-PROTOCOL-002 have been given. I notice that the code fails to configure hostname: HS-SEC-001 VIOLATED: 1) ✗ No call to X509_VERIFY_PARAM_set1_host() 2) ✗ No manual hostname verification implementation 3) ✗ Certificate accepted without hostname check
Detection Result: *VULNERABILITY CONFIRMED* Confidence: 0.95 | Match: HS-SEC-001 | Fix Required: *`Add X509_VERIFY_PARAM_set1_host()`*
LLM distinguishes vulnerable code from patch Code in detecting patch code in new version: Based on given specification, the target function (fixed version) uses explicit call to verify hostname. I don't notice the common vulnerabilities or code logic violate the given specifications

Learning Domain-specific Specifications

- Attackers rarely invent entirely new exploits.
  - Google Project Zero: among 18 zero-day vulnerabilities disclosed in 2022, at least half were variants of previously patched vulnerabilities.
  - Each recurring exploitation mechanism corresponds to the violation of the similar security specification.
- In CVEs (2015–2016) in ImageMagick image parsers, for different formats (SUN, BMP, RLE), share a recurring exploitation mechanism:
  - Untrusted size fields from image headers were directly used in allocation or offset calculations.
  - Across cases, the common thread is the same: failure to validate external size metadata before use.

Case Study: Image Parsing in ImageMagick

- Retrieve historical CVEs with similar characteristics where untrusted dimension metadata flowed directly into memory access.
- Identify the consistent mechanism: Untrusted Size Field exploitation pattern and matches AS-IMG-001 specification.
- Apply the spec to the target code, constructing a threat model where manipulated alpha channel offsets in a crafted TIFF file could trigger out-of-bounds access.

(a) Historical CVEs with Untrusted Size Field Pattern

CVE-2015-8957 — SUN
- *Vulnerable Function:* `ReadSUNImage()`
- *Attack Vector:* width × height → integer overflow
- *Trust Issue:* No validation of image dimensions from file header
- *Impact:* Buffer overflow → DoS

CVE-2016-6823 — BMP
- *Vulnerable Function:* `WriteBMPImage()`
- *Attack Vector:* width × height × bpp → overflow
- *Trust Issue:* BMP header values used directly in allocation
- *Impact:* Out-of-bounds write → DoS

CVE-2016-10049 — RLE
- *Vulnerable Function:* `ReadRLEImage()`
- *Attack Vector:* columns × rows → overflow
- *Trust Issue:* Trusts RLE header dimensions without bounds check
- *Impact:* Buffer overflow → RCE

CVE-2016-10050 — RLE
- *Vulnerable Function:* `ReadRLEImage()`
- *Attack Vector:* pixel offset → negative value
- *Trust Issue:* No check for negative offset calculations
- *Impact:* Heap overflow → RCE

AS-IMG-001: Image processing functions MUST validate all dimension fields from untrusted file headers before any memory allocation operations.

(b) Specification-Guided Vulnerability Detection in target code (**CVE-2022-0284**)

Potential Recurring Exploitation Mechanism Recognition

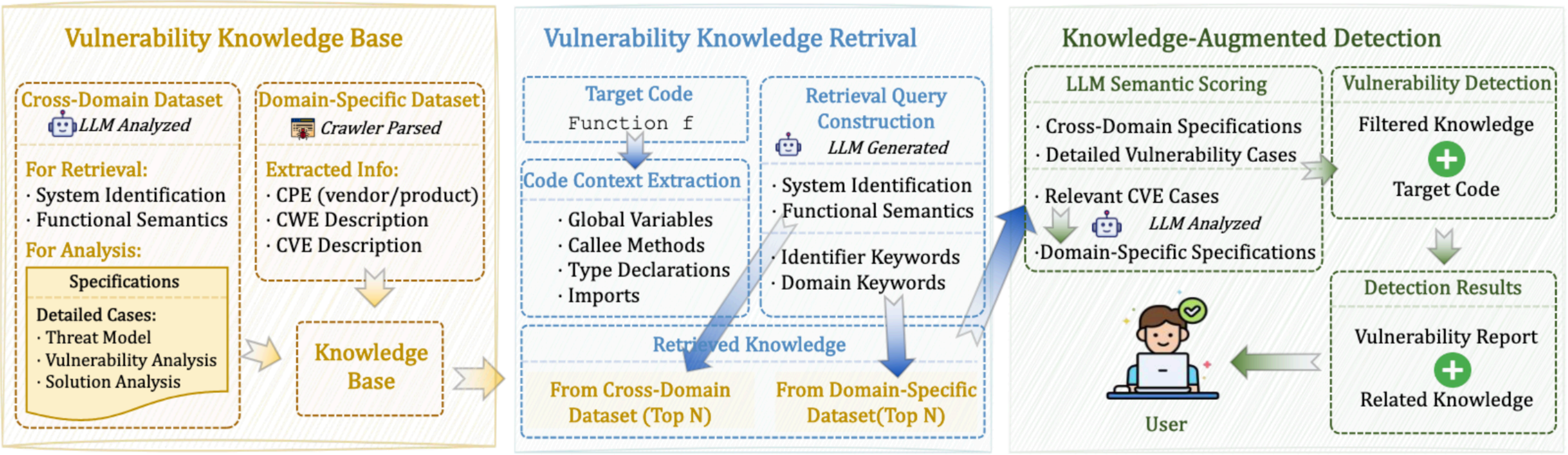
Function Analysis: The `GetPixelAlpha()` function retrieves alpha channel values during TIFF to PICON format conversion, directly accessing pixel data via array indexing *without checking if the offset is within allocated memory bounds*. This occurs during the transparency check in the `WritePICONImage()` conversation process.

Given AS-IMG-001, I have notice the "Untrusted Size Field" explotation mechanism occurs *when image parsers use unvalidated dimension-related metadata*, allowing attackers to craft files with malicious values that cause …….

Trust Boundary Analysis: The critical trust boundary violation occurs *when untrusted TIFF metadata (specifically alpha channel specifications) crosses into the pixel accessor module*. The system implicitly trusts that the TIFF file's declared channel offsets are valid, creating an assumption that external input has been properly sanitized—which it has not. An attacker can exploit *manipulated alpha channel offsets in a crafted TIFF file*, causing GetPixelAlpha() to read from arbitrary memory locations, potentially exposing sensitive data or causing crashes.

Applying our specification-guided detection, we identify that `GetPixelAlpha()` **exhibits all characteristics of** the historical attack pattern: (1) It processes metadata from an external file format, (2) ……

Methodology



Specification Knowledge Base

- General specifications are pre-extracted, reusable rules of expected safe behavior
- Domain-specific specifications are not pre-built; instead, maintain a domain evidence base and dynamically abstracts domain-specific specifications.
- This design allows general specifications to provide broad, cross-project coverage, while domain-specific specifications capture context-aware expectations.

General Specification Knowledge Base

- A **general security specification** is designed as follows:
  - HS-[DOMAIN]-[ID] : [Expected Safe Behavior].
  - DOMAIN indicates the security domain, ID provides a unique index.
- For **Detailed vulnerability cases**, each specification is linked with a structured case that grounds the abstract rule in concrete evidence from real code.
  - Documented in three forms: Threat model, Vulnerability analysis, Solution analysis
- Each vulnerability instance is represented as vulnerable and fixed functions, commit message, CVE description, CWE type and the broader program context (i.e. callees, types, imports, globals)
- Attach two retrieval keys for efficient matching with new code: (i) a system identification key and (ii) a functional semantics key
- These keys allow VulInstruct to retrieve relevant specifications based on both program role and semantics, rather than relying on surface similarity.

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.

Prompt: System Prompt for General Specifications (Con't)

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.]

Prompt: System Prompt for General Specifications (Con't)

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

Prompt: System Prompt for General Specifications (Con't)

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

Prompt: System Prompt for General Specifications (Con't)

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

Prompt: System Prompt for General Specifications (Con't)

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;
+}
```

Prompt: System Prompt for General Specifications (Con't)

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

Prompt: System Prompt for General Specifications (Con't)

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 Boundarie**s: 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})

Prompt: Detailed Vulnerability Cases in General Specifications (Con't)

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

Prompt: Detailed Vulnerability Cases in General Specifications (Con't)

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>

Prompt: Detailed Vulnerability Cases in General Specifications (Con't)

<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>

Prompt: Detailed Vulnerability Cases in General Specifications (Con't)

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: 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**

Prompt: Domain-specific Specification Extraction (Con't)

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>
```

Knowledge Retrieval

- Extract TC surrounding program context
- General Specification:
  - Prompt LLM to generate system identification and functional semantics.
  - Using embedding similarity search, get the specifications and detailed vulnerability analysis for top-N cases.
    - Identifier keywords anchor the function to its concrete software context.
    - Domain keywords characterize the broader functionality exposed to attackers and capture recurring attack surfaces.
- Specific Specification:
  - Filtering and ranking process to retrieve the most relevant top-N CVE cases.

Specification-guided Detection

- Prompt the LLM to evaluate the retrieved top-N specifications, detailed vulnerability cases, and top-N CVE cases and filter out low-scoring items
- By analyzing multiple related cases, Domain-specific specifications are dynamically abstracted and expressed as: AS-[DOMAIN]-[ID] : [Expected Safe Behavior].
- Using the filtered knowledge as context, 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.

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: 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.

Prompt: Vulnerability Detection (Con’t)

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

Datasets

- Two datasets with distinct roles:
- CORRECT provides rich contextual knowledge for building VKB.
  - Contains 2000 vulnerable functions with comprehensive contextual information (callee functions at multiple depths; type declarations/data structures; global variables/constants; imports/module dependencies).
- PrimeVul serves as evaluation benchmark.
  - Key feature is temporal data splitting (train: before cutoff; test: after cutoff) to prevent learning future vulnerabilities and ensure realistic generalization.
- After temporal filtering, the VKB construction dataset for general security specifications includes 1,338 vulnerable-patched pairs from CORRECT that predate the PrimeVul test cutoff.

Baselines

- Compare against state-of-the-art LLM-based vulnerability detection approaches
- Prompting Methods (Chain-of-Thought): follow Ding et al.'s approach with step-by-step reasoning; baseline of direct prompting without enhancements.
- Fine-tuning Methods (ReVD): state-of-the-art fine-tuned approach; uses Qwen2.5-Coder; synthesizes 28,000 vulnerability analysis reasoning data; achieves SOTA on PrimeVul.
- VulTrial: agent-based vulnerability analysis with four role-specific agents; fine-tunes GPT-4o with role-specific instructions; SOTA among agent-based approaches on PrimeVul.
- GPTLens: multi-agent framework that automates vulnerability analysis workflows and iterative reasoning to enhance LLM reasoning ability.
- Vul-RAG: retrieves vulnerability knowledge based on functional semantics to augment detection; extracts multi-dimensional knowledge including vulnerability causes and fixing solutions.

Evaluation Metrics

- A model may predict the correct label while relying on spurious code patterns that do not reflect the actual vulnerability scenario.
- Adopt the CORRECT evaluation metrics, which assess both the vulnerability label and the underlying reasoning process.
  - Verified by an LLM-as-a-Judge.
- Also employ pair-wise prediction metrics to assess whether models can distinguish vulnerable functions from their patched functions.
  - P-C: where (1,0) denotes ideal detection (i.e. the model correctly identifies the vulnerability in the original code but not in its patched version.)
  - VP-S: P-C - P-R, where P-R (0,1) represents reversed prediction.

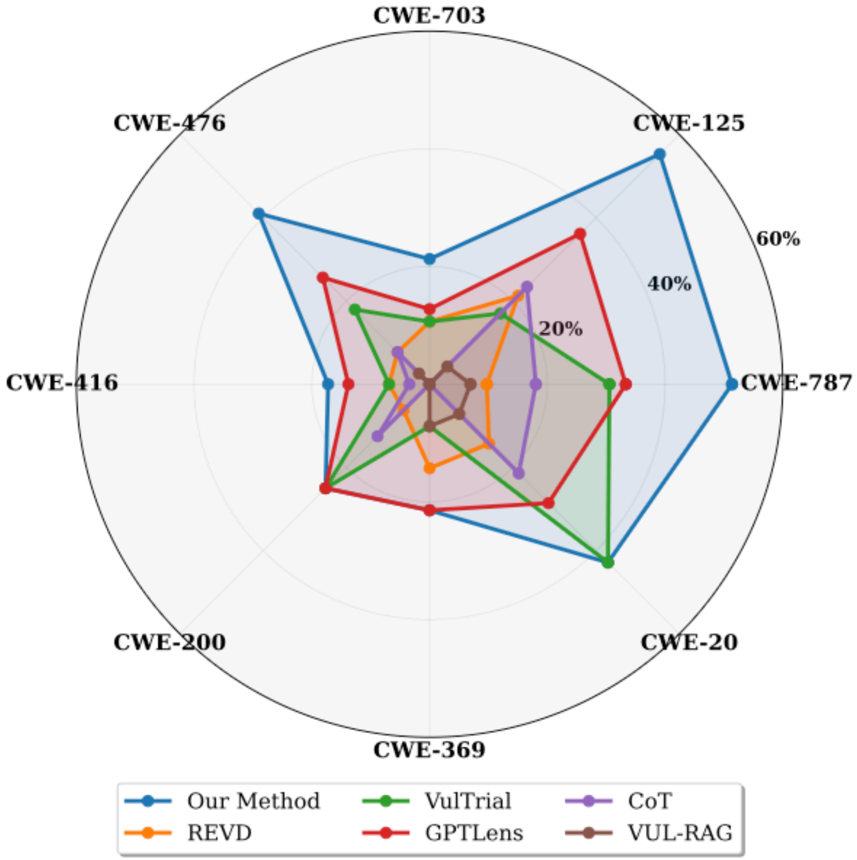
Implementation Details

- Qwen3-Embedding-0.6B to generate embeddings for retrieval queries and stored system identification and functional semantics.
- For both types of security specifications, we retrieve top-N = 10 candidates initially; in knowledge scoring we apply a unified threshold of 6 points (6, 6, 6) to filter three types of knowledge.
- For knowledge extraction, keyword generation, and knowledge scoring, we use DeepSeek-V3.
- For CORRECT evaluation, we use GPT-5 as the LLM-as-a-Judge model, replacing GPT-4o used in the original CORRECT implementation.

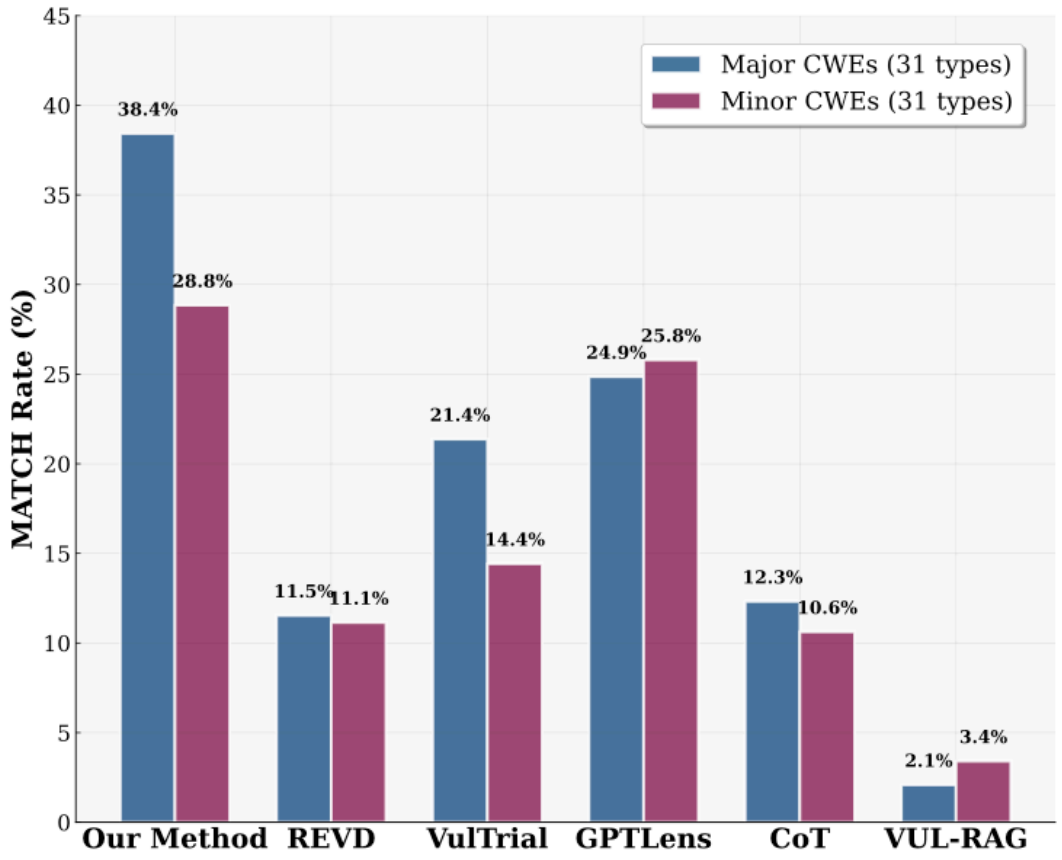
How effective is VulInstruct in vulnerability detection compared to state-of-the-art approaches?

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	✓	<u>53.4</u>	<u>59.9</u>	20.8	<u>10.2</u>	30.9	12.3	<u>6.9</u>
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	<u>25.0</u>	4.4	<u>33.9</u>	<u>13.0</u>	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%

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



(a) CWE Radar Chart

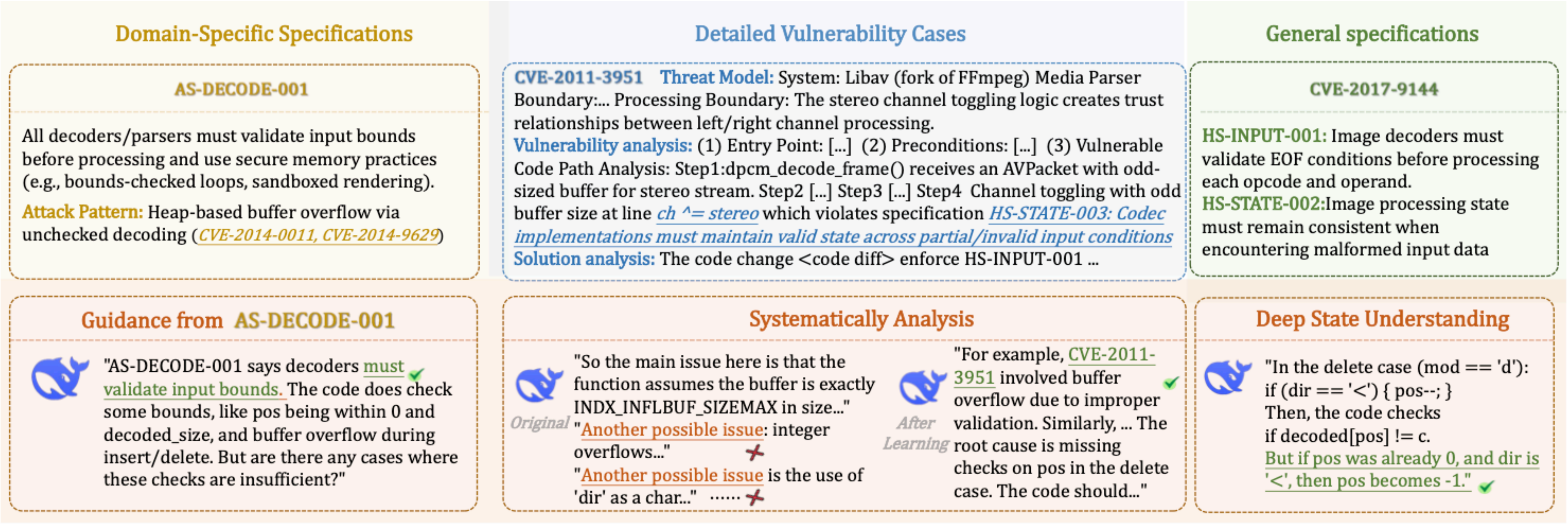


(b) Head/Tail Performance

How well does VulInstruct generalize across different vulnerability types and models? (Con't)

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%* ↑

What makes VulInstruct more effective than existing approaches?



Conclusion

- 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, it helps LLMs reason and align with implicit expectations of safe behavior.
- On PrimeVul, evaluation shows substantial improvements in F1-score, recall, and pair-wise discrimination vs. SOTA baselines.
- Demonstrated practical utility by discovering a previously unknown high-severity vulnerability in real-world software.

Acknowledgments

- [VulInstruct] Specification-Guided Vulnerability Detection with Large Language Models, Zhu H., Li J., Gao C., Qian J., Dong Y., Liu H., Wang L., Wang Z., Hu X., Li G., Arxiv, Nov 2025.