

CognixShield: PoV-Guided Vulnerable API Usage Detection in Large Codebases via LLMs

Quanzhi Fu¹ 

Virginia Tech, Blacksburg, VA, USA

Wenjia Song 

Virginia Tech, Blacksburg, VA, USA

Yi Liu 

Griffith University, Gold Coast, Australia

Ying Zhang² 

Wake Forest University, Winston Salem, NC, USA

Wang Lingxiang¹ 

Independent Researcher, Sunnyvale, CA, USA

Gelei Deng 

Nanyang Technological University, Singapore

Dan Williams 

Virginia Tech, Blacksburg, VA, USA

Abstract

Background. The integration of open-source libraries in Java development introduces severe security risks through vulnerable APIs. Existing program analysis and deep learning tools face challenges in capturing inter-procedural vulnerability semantics at scale. While LLMs show promise for semantic reasoning, they cannot handle large codebases due to context limits, and they lack the vulnerability-specific understanding needed to determine exploitability.

Aim. This work aims to overcome these limitations and enable LLM-based detection of vulnerable API usage in large-scale Java applications.

Method. We present CognixShield, an LLM-powered framework for detecting vulnerable API usage through three core components. First, semantic-preserving AST-based fragmentation partitions large codebases while maintaining syntactic completeness within LLM context windows. Second, vulnerability-aware multi-agent RAG traces relevant program context across these fragments, iteratively assembling security-critical context spanning functions and files. Third, PoV-guided semantic reasoning uses Proof-of-Vulnerability tests that encode precise triggering conditions and exploitation mechanics to determine vulnerability.

Results. CognixShield achieves 84% precision, 95% recall, 84% accuracy, and an 89% F1-score on 57 real-world Java applications, outperforming state-of-the-art tools.

Conclusions. Our results show that vulnerability detection requires specialized architectural innovations beyond generic LLM applications.

2012 ACM Subject Classification Security and privacy → Software security engineering

Keywords and phrases Vulnerable API usage detection, program analysis, LLMs, agentic RAG

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.1

Category Technical Track Paper

Supplementary Material *Software (Source Code and Dataset):* <https://github.com/ying-selab/CognixShield>

Acknowledgements This work has been partially supported by the Wake Forest Pilot Research Grant. We thank the anonymous reviewers for their valuable comments on an earlier version of this paper.

¹ These authors contributed equally.

² Corresponding author.



1 Introduction

Developers rely on many open-source libraries (Lib) and their application programming interface (API) to accelerate development. However, APIs that contain vulnerabilities can be exploited by hackers, turning widely used libraries into powerful attack vectors [51]. Attackers can introduce or discover exploitable vulnerabilities in Libs through (1) contributing flawed code to open-source projects, (2) inspecting library implementation, or (3) consulting publicly available vulnerability databases like Common Vulnerabilities and Exposures (CVE) [55]. By crafting malicious inputs and propagating them from Apps to **vulnerable APIs**, attackers can compromise Applications (Apps) as long as the underlying Libs accept such inputs. The Open Web Application Security Project (OWASP) ranked “software supply chain failures” as the third most critical security risk in 2025 [3].

To mitigate these security risks, existing software composition analysis (SCA) tools [40, 54, 27, 45, 1, 4] identify Apps affected by vulnerable API usage via scanning code bases for (1) dependencies on vulnerable Lib versions and/or (2) calling vulnerable APIs in Libs. However, these tools suffer from low accuracy for two reasons. First, dependency-based analysis is overly coarse-grained and lacks context: an App may depend on a Lib that contains vulnerable APIs, yet never invokes those methods and therefore remains unaffected. As noted previously, “*SCA without context generates too many false positives, this results in your team wasting time chasing ghosts* [39]”. Second, SCA tools overestimate feasible attack paths at scale due to imprecise code semantic reasoning. VAScanner [54], for example, flags a vulnerability whenever a vulnerable API invocation is reachable along an approximate call chain, without verifying whether the usage is actually exploitable in the calling context.

Large Language Models (LLMs) have emerged as a promising alternative solution for vulnerability detection [57, 49]. They offer semantic reasoning over code [15] and can be guided through prompt engineering without large labeled datasets. Nevertheless, applying LLMs to detect vulnerable API usage in Apps has two challenges: **Challenge 1: Maintaining inter-procedural context**. LLMs struggle to track repository-level context across deep function hierarchies [32]. Since vulnerable API usage frequently spans multiple files connected by complex call chains, accurate detection requires preserving inter-procedural context. **Challenge 2: Reasoning about vulnerability-specific semantics**. Trained on general-purpose corpora, LLMs lack the specialized security knowledge needed to reason about vulnerable API usage. Without vulnerability-oriented guidance, they misinterpret program semantics and produce unreliable detections.

In this paper, we introduce COGNIXSHIELD, a high-accuracy framework for detecting vulnerable API usage in Apps using LLMs. Given a vulnerable API from an upstream Lib and an App’s source code, COGNIXSHIELD determines whether the API is exploitable from a public entry point in the app. COGNIXSHIELD combines program analysis with LLM-based reasoning through two key designs. To maintain repo-level program context (**Challenge 1**), we design a multi-agent pipeline in which specialized LLM agents iteratively retrieve missing security-critical context from AST-aligned code blocks and progressively assemble code fragments across functions/ files until the program context from vulnerable API invocation to public entry points is captured. To guide vulnerability-specific semantic reasoning (**Challenge 2**), we leverage Proof-of-Vulnerability (PoV) tests from upstream Libs. PoV tests encode precise triggering conditions, exploitation mechanics, and observable impacts. COGNIXSHIELD extracts these semantics from PoV tests and applies them to the repository-level context to determine whether the vulnerability is reachable and exploitable from public entry points.

We evaluate COGNIXSHIELD on 57 real-world Java projects to assess its detection effectiveness. COGNIXSHIELD achieves a precision of 84%, a recall of 95%, an F1-score of 89%, and an accuracy of 84%. Compared with five state-of-the-art (SOTA) tools, two based on program analysis (Eclipse Steady [40] and VAScanner [54]), two DL-based approaches (DeepDFA [45] and LineVul [19]), and one LLM-agent based approach (Codex [36]), COGNIXSHIELD demonstrates substantial improvements, achieving at least 18%, 210%, 123%, and 100% higher performance, respectively, across four evaluation metrics. To assess robustness, we further evaluate COGNIXSHIELD varying configurations of embedding models, LLMs, and code fragmentation granularities. Across these settings, COGNIXSHIELD maintains better performance against SOTAs, with F1 scores ranging from 64% to 89% and accuracy between 58% and 84%, demonstrating effectiveness across different models and code fragment sizes.

Our research contributions are summarized as follows:

- We propose COGNIXSHIELD, an LLM-powered approach for vulnerable API usage detection in large-scale real-world applications, which overcomes LLMs’ context window limitations through an agent-based RAG that iteratively retrieves inter-procedural code fragments across multiple files.
- We evaluated COGNIXSHIELD on 57 real-world Java projects, demonstrating that it outperforms five SOTA tools across all metrics.
- We conducted an extensive evaluation of different large language models (LLMs) and embedding models for vulnerability detection in code segments of varying lengths. We observed that for vulnerability detection, with OpenAI embeddings and CLAUDE-3.5-SONNET, consistently demonstrated higher performance compared to other tested models across all metrics.

Our code and dataset is available at: <https://github.com/ying-selab/CognixShield>.

2 Background and Motivation Example

In this section, we first present a motivating example that illustrates the challenges of accurately detecting vulnerable API usages in large-scale software projects. We then provide the background on our system design.

2.1 Motivation Example

We use a concrete example to illustrate how the same vulnerable API can be exploitable in one application but not in another. Consider a vulnerability analyst, Alex, who is assigned to audit two large software applications, Apache Kylin [9] and Alibaba Nacos [7], for potential risks caused by a CVE-2020-5408. CVE-2020-5408 reveals a crypto failure in versions 4.2.x prior to 4.2.16. Specifically, the `BCryptPasswordEncoder.encode` method throws a `NullPointerException` when given a null input. This vulnerability was later patched in version 5.4.0-M1 by properly handling null inputs with an `IllegalArgumentException` as shown in Listing 1.

■ **Listing 1** Simplified Fix Commit for CVE-2020-5408 [44].

```

1 public String encode(CharSequence rawPassword) {
2 + if (rawPassword == null) {
3 +   throw new IllegalArgumentException("rawPassword + cannot be null");}
4   ...
5   return BCrypt.hashpw(rawPassword.toString(), salt);}
6
7 //unit test for NPE
8 + @Test(expected = IllegalArgumentException.class)
9 + public void encodeNullRawPassword() {
10 +   BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();
11 +   encoder.encode(null); }
```

Both Apache Kylin and Alibaba Nacos depend on the vulnerable library version and contain large codebases with over 5,000 source files and 500K lines of code. Alex needs to determine whether either application is actually exploitable due to this vulnerable API usage. As shown in Listing 2, Nacos invokes `BCryptPasswordEncoder.encode` within `PasswordEncoderUtil.encode` without a null check (line 11), so an untrusted `null` can propagate through `createUser` (line 4) and trigger an application crash, enabling denial-of-service attacks. Conversely, as shown in Listing 2, Apache Kylin [9] validates all passwords via `checkPasswordLength` before encoding, ensuring `null` cannot reach the vulnerable API. Thus, despite using the same vulnerable library, Kylin is not exploitable.

■ **Listing 2** Usage of `BCryptPasswordEncoder.encode`: (a) vulnerable usage in Nacos [7] – `null` propagates unchecked to the API; (b) secure usage in Kylin [10] – `checkPasswordLength` rejects `null` before encoding.

(a) Nacos (vulnerable).

```
1 public Object createUser(String username,
   ↪ String password) {
2     ...
3     userDetailsService.createUser(username,
4     PasswordEncoderUtil.encode(password));
5     return RestResultUtils.success("create user
   ↪ ok!");
6 }
7
8 public class PasswordEncoderUtil {
9     ...
10    public static String encode(String raw) {
11        return new
   ↪ BCryptPasswordEncoder().encode(raw);}
12 }
```

(b) Kylin (secure).

```
1 public EnvelopeResponse save(
2     @RequestBody PasswdChangeRequest user) {
3     ...
4     checkNewPwdRule(user.getNewPassword());
5     if (existing != null) {
6         existing.setPassword(
7             pwdEncode(user.getNewPassword())); }
8
9     private void checkNewPwdRule(String newPwd) {
10        if (!checkPasswordLength(newPwd)) {...} }
11
12    private boolean checkPasswordLength(String
   ↪ password) {
13        return !(password == null
14            || password.length() < 8); }
```

Given the scale of these projects, manual code review is impractical, so Alex must rely on automated tools. Programming analysis tools [40, 2] encounter scalability challenges to accurately report vulnerable API usage. DL-based approaches [19, 45] may not fully capture vulnerable API usage in Nacos, or overlook safeguard checks in Kylin.

While LLMs provide enhanced semantic reasoning capabilities, applying them directly to large-scale applications faces similar limitations. Their finite context window makes it impractical to analyze entire large codebases [15]. In our preliminary experiments (detailed in Section 5.3), we found that the majority of real-world projects in our dataset exceed the context limits of the tested LLMs. Without an effective mechanism to selectively capture code relevant to vulnerable API usages and guidance for vulnerability-specific semantic reasoning, repo-wide analysis remains infeasible.

2.2 Retrieval-Augmented Generation

RAG [26] is a widely used framework that combines information retrieval with LLMs. A typical RAG architecture consists of a retriever and a generator: RETRIEVER searches an external knowledge source and the GENERATOR integrates the retrieved information into its prompt to produce a more context-aware response. The general workflow involves: (1) embedding the knowledge base into a searchable index [34, 48], (2) generating a query from the task prompt, (3) retrieving relevant entries, and (4) augmenting the prompt with these retrieved results before final response generation. An LLM agent is an autonomous system built on top of the LLM (e.g., GPT-4o [35], Claude [8]). When combined with RAG, such agents form agentic RAG [42], where the model dynamically refines its retrieval queries based on partial context (e.g., reasoning about what information is missing) and performs multiple retrieval iterations.

3 Methodology

3.1 Overview

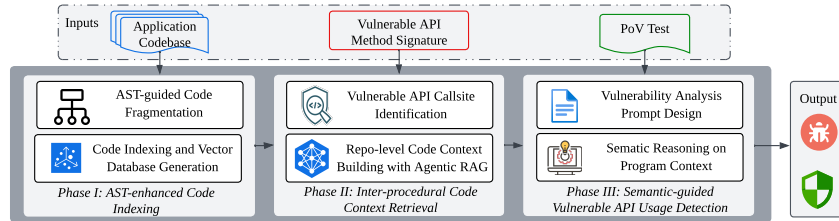


Figure 1 The Overview of COGNIXSHIELD.

The overall workflow of COGNIXSHIELD is depicted in Figure 1. Given an application that depends on a vulnerable library version, the corresponding vulnerable API, and an available PoV patch, COGNIXSHIELD analyzes whether the application context can make that API usage exploitable. The workflow is organized around the two challenges in the introduction: Phase I maintains inter-procedural context for Challenge 1, and Phase II applies vulnerability-specific semantic reasoning for Challenge 2.

Phase I: maintaining inter-procedural context. COGNIXSHIELD first parses the application source code with AST-guided structural segmentation, partitions the repository into semantically self-contained code blocks, and stores each block with its metadata and code embedding. This AST-enhanced code indexing makes repository-scale code searchable while preserving method, class, and file boundaries needed to recover caller/callee relationships around vulnerable library API invocations. Starting from each vulnerable API query, COGNIXSHIELD then drives an Agentic RAG loop that iteratively retrieves and filters security-relevant program context, assembling sufficient repo-level inter-procedural context for the downstream exploitability decision (Section 3.2.2).

Phase II: reasoning about vulnerability-specific semantics. Given the retrieved inter-procedural context and the PoV patch, COGNIXSHIELD extracts PoV-derived vulnerability semantics, including trigger conditions, patch constraints, exploit mechanics, and observable impact. It then checks whether those conditions can propagate from public entry points through application data and control flows to the vulnerable API invocation. The output is an application-level decision that distinguishes exploitable vulnerable API usages from safe usages of the same library API.

3.2 Phase I: Maintaining Repo-level Inter-procedural Context

3.2.1 AST-enhanced Code Indexing

Phase I addresses Challenge 1 by turning the application repository into a searchable context index for vulnerable API usage analysis. A local invocation of a vulnerable library API rarely carries all the information needed to judge whether the usage is safe: the relevant caller, helper callee, field definition, object construction site, or guarding check often lives in another file, and the security-relevant data and control constraints span the structural units that the source code already exposes. The index must therefore preserve these program-defined units so that callers, callees, cross-file definitions, and data/control constraints remain reconnectable during retrieval. COGNIXSHIELD builds the index from the program’s AST rather than

flattening the repository into arbitrary token chunks. The root node (`compilationUnit`¹) corresponds to a source file, and each child node denotes a syntactic element such as a class, method, or statement. Each node contains metadata, including node type and source code location, and hierarchical relationships. Parent nodes structurally encompass their children; for example, a `classDeclaration` node encapsulates modifiers, fields, methods, constructors, and other elements defined at the class level within the class body.

To preserve semantically self-contained and contextually relevant code blocks, we leverage the AST to guide code partitioning. We convert the application source code into an AST forest as $G = \{g_1, g_2, \dots, g_n\}$, where each g_i corresponds to a `compilationUnit`. To obtain contextually relevant code blocks, we perform top-down, structure-guided fragmentation on each compilation unit. If a compilation unit is smaller than the predefined threshold θ , it is preserved as a whole. Otherwise, oversized nodes are recursively divided into child-nodes. Recursion stops when a node falls below θ or reaches a leaf with no further structural decomposition. For instance, a `compilationUnit` can be fragmented as `ImportDeclaration` and `classDeclaration`. This AST-guided structural segmentation keeps repository code aligned with source-level method, class, and file boundaries, so later retrieval can reconnect vulnerable API invocations with surrounding caller/callee paths and security-relevant data/control constraints.

Following the AST-enhanced code segmentation, COGNIXSHIELD converts each code block b_i in $B = \{b_1, b_2, \dots, b_m\}$ into a fixed-length dense vector representation using pre-trained encoders. These embeddings capture the semantics and functionality of the fragments, positioning semantically similar blocks closer together in the vector space, even if their syntactic forms differ. For instance, the following two snippets perform the same task of summing elements despite having different token sequences:

```
a) for (int i = 0; i < n; i++)    sum += arr[i];
b) total = Arrays.stream(list).sum();
```

Although these snippets use different tokens and syntax, once embedded, they yield a high similarity score because they perform the same functionality. Within Phase I, these embeddings serve as retrieval keys rather than standalone predictions. COGNIXSHIELD therefore stores all processed data in a database $D = \{d_1, d_2, \dots, d_m\}$, where each entry $d_i = (b_i, v_i)$ consists of the original code block b_i , which includes the source snippet and associated metadata, and its corresponding vector embedding v_i . This database becomes the searchable substrate for the subsequent Phase I context-retrieval step, where semantically related fragments are assembled into sufficient repo-level inter-procedural context before Phase II performs vulnerability-specific semantic reasoning.

3.2.2 Agentic RAG for Inter-procedural Context Retrieval

To address Challenge 1 for context retrieval, COGNIXSHIELD assembles the required inter-procedural context through an Agentic RAG framework built on LLM-based reflection [11], consisting of two collaborating LLM-based agents: a *Reflector* and a *Grader*. At each iteration, the *Reflector* reasons about missing caller/callee, data-construction, validation, or control-flow links and formulates focused embedding queries against the AST-enhanced vector database D described above, while the *Grader* retains only fragments with semantic relevance to the vulnerable API usage.

¹ We use the terms defined in Tree-sitter [12] for ease of understanding.

The loop iterates over a set of candidate vulnerable API invocation sites. Given a vulnerable library API (API_{vul}), COGNIXSHIELD first locates its invocation sites in the application by matching the method signature API_{vul} , forming the initial statement-level candidates $C_{candidate} = \{c_1, c_2, \dots, c_n\}$.

Given these candidates, the *Reflector* and *Grader* interact as follows:

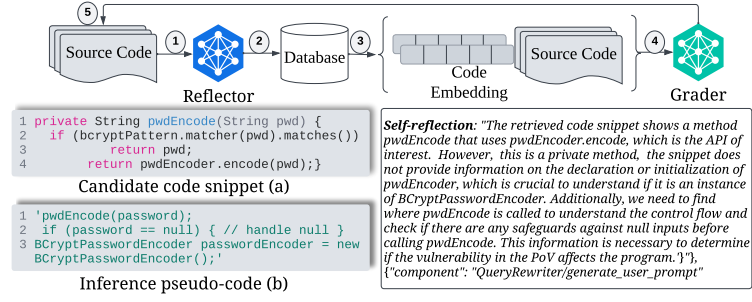
- *Self-Reflection*: For each candidate $c_i \in C_{candidate}$, the *Reflector* evaluates whether the current fragment set provides sufficient inter-procedural context for analyzing the vulnerable API usage. It issues the reflection query $Q(c_i, API_{vul}) \rightarrow (\{\text{Complete}|\text{Incomplete}\}, \text{reason})$. A **Complete** response indicates that the current fragments contain the relevant elements needed for the Phase II Expert agent to reason about the invocation path, while an **Incomplete** response explains which related code is missing and guides subsequent inference. The missing link may be a caller, callee, public entry point, object construction statement, validation check, or control-flow constraint. For example, the *Reflector* examines candidate snippet (a) in Figure 2, where the vulnerable API `pwdEncoder.encode()` is invoked inside a private method. This fragment alone lacks sufficient context because it does not show how the password value is constructed or guarded before the call; the *Reflector* therefore returns an **Incomplete** verdict naming the missing caller, data-construction, and validation links.
- *Code-inference & Retrieval*: When the context is **Incomplete**, the *Reflector* formulates a focused retrieval query for the missing element identified by the reflection verdict. The query takes the form of an inferred code sketch that approximates the missing element's expected signature and surrounding scope. As shown in pseudo-code (b), the *Reflector* sketches how `pwdEncode()` may be invoked by its callers and whether the `password` parameter is checked for null before reaching `pwdEncoder.encode()`. This inferred information drives an embedding-based search over the repository vector database D , allowing COGNIXSHIELD to retrieve AST-aligned fragments from across the repository.
- *Grading*: The *Grader* evaluates the raw retrieval results at the source level and verifies their semantic relevance to the vulnerable API usage. A retained snippet must extend the security-relevant inter-procedural chain, for example by establishing a caller/callee relationship, defining data passed into the API, implementing a validation check, or constraining the control path to the invocation. Snippets that share lexical similarity but do not affect the vulnerable API usage are filtered out. Validated snippets are merged into the candidate set: $C_{candidate_i} = C_{candidate_i} \cup C_{validated}$. This step mitigates false positives that embedding similarity alone may introduce.

The process repeats until the *Reflector* judges that the current fragment set provides sufficient inter-procedural context for vulnerability reasoning, or until retrieval produces no additional non-duplicated validated fragments. The process is shown at Algorithm 1.

3.3 PoV-guided Vulnerable API Usage Detection

To address Challenge 2, COGNIXSHIELD performs vulnerability-specific semantic reasoning over the inter-procedural context assembled in Phase I. After retrieving the sufficient inter-procedural context, we introduce an *Expert* agent that applies PoV-derived vulnerability semantics to that context. The *Expert* receives a structured system prompt and standardized inputs/outputs to ensure reliable reasoning.

The system prompt is specifically designed to enable context-aware detection of vulnerable API usage. Unlike generic prompts that merely classify code as vulnerable or secure, our prompt instructs a two-stage reasoning process: (i) **PoV-derived vulnerability semantics extraction**, where the model extracts trigger conditions, patch constraints, exploit mechanics,



■ **Figure 2** Interaction process between *Reflector* and *Grader* agents on an exemplar candidate for the `BCryptPasswordEncoder.encode()` in Apache Kylin.

■ **Algorithm 1** Iterative Phase-I Inter-procedural Context Retrieval.

Require: $C_{candidates} = \{c_1, c_2, \dots, c_n\}$, API_{vul}

Auxiliary Functions:

`Search(code, scope) → C`: Search for similar code based on embeddings

`Reflector_evaluate( $C$ ,  $API_{vul}$ ) → (hasSufficientContext, missingContext)`: Determines whether C provides sufficient inter-procedural context for analyzing API_{vul}

`Grader_filter( $C_{raw}$ , context) →  $C_{filtered}$` : Filters out irrelevant snippets from raw search results based on context relevance

`Reflector_Inf(missingContext) → ( $c_{missing}$ , scope)`: Generates missing code snippet and structural constraints

```

1: for each  $c_i \in C_{candidates}$  do
2:    $C' \leftarrow \{c_i\}$ 
3:   ( $hasSufficientContext$ ,  $missingContext$ )  $\leftarrow$  Reflector_evaluate( $C'$ ,  $API_{vul}$ )
4:   while not  $hasSufficientContext$  do
5:     ( $c_{missing}$ ,  $scope$ )  $\leftarrow$  Reflector_Inf( $missingContext$ )
6:      $C'_{raw} \leftarrow$  Search( $c_{missing}$ ,  $scope$ )
7:     if  $C'_{raw} == \emptyset$  then
8:       break
9:     end if
10:     $C'_{candidate} \leftarrow$  Grader_filter( $C'_{raw}$ ,  $C' \cup \{API_{vul}\}$ )
11:    if  $C'_{candidate} == \emptyset$  then
12:      break
13:    end if
14:     $C' \leftarrow C' \cup C'_{candidate}$ 
15:     $C_{candidates} \leftarrow C_{candidates} \cup C'_{candidate}$ 
16:    ( $hasSufficientContext$ ,  $missingContext$ )  $\leftarrow$  Reflector_evaluate( $C'$ ,  $API_{vul}$ )
17:  end while
18: end for

```

and observable impact from the provided PoV; and (ii) **semantic propagation analysis**, where multiple related code fragments are analyzed collectively to deduce cross-method path or data constraints, determining whether the exploit condition can propagate to the vulnerable API invocation. For each candidate c_i in the set $C_i = \{c_1, c_2, \dots, c_n\}$ from Phase I, we construct a *task-specific* guidance prompt to the *Expert*:

Vulnerable API Usage Analysis Prompt

You are an expert in Java vulnerability analysis. Your task is to determine if a project is affected by a library vulnerability. Given a project, a vulnerable API, and a library PoV, analyze whether the complete snippet set is affected by the vulnerable API Usage.

Vulnerable API:

Library POV:

Related code snippet: ...

Given this structured input, the *Expert* executes the two stages above and produces an interpretable vulnerability detection result. For instance, based on the code example in the Kylin project (Section 2.1), COGNIXSHIELD analyzes whether variables `pwd` and calls to `pwdEncoder.encode(pwd)` can result in a null value reaching the vulnerable API `BCryptPasswordEncoder`

`Encoder.encode()`, as described in the PoV. The *Expert* generates conclusions such as: “*The code has multiple safeguards preventing null passwords from reaching the encoder.*”, and the project is not vulnerable to the API usage.

Building on this analysis, COGNIXSHIELD evaluates each candidate c_i independently using its corresponding prompt Q_i , and the LLM returns a binary decision: 1 (vulnerable) or 0 (secure). The application is classified as vulnerable if any candidate query is identified as exploitable; it is classified as secure only when all candidates are assessed as non-exploitable.

4 Implementation

In phase I, we employ the tree-sitter [12] to generate an AST representation of the codebase. Subsequently, our tool partition the codebase into code blocks via recursive AST decomposition with a $\theta = 2500$ token threshold, which are then serialized into a single JSON file. Each JSON object encapsulates essential metadata, including the filename, start line, end line, code snippet, and relevant contextual information. After fragmentation, each code block is embedded using the OpenAI text-embedding-3-small model [34], which is widely used in industry AI coding tools, while its metadata fields are preserved. We store the resulting embeddings and metadata in Qdrant [48], a vector database, using cosine distance and a similarity threshold of 0.35 for retrieval. Both the *Reflector* and *Grader* use GPT-4o with temperature 0 to minimize output randomness. We use Tai-e [47] to locate candidate invocation sites of API_{vul} with the vulnerable API signature given as input, e.g., `com.alibaba.fastjson.JSON.parseObject` in the project’s compiled bytecode to initialize the context retrieval; when the signature names an overloaded API, the sites of all overloads are included as candidates. In Phase II, the *Expert* agent is implemented as a GPT-4o-based LLM agent with temperature 0.

5 Evaluation

Our evaluation aims to answer the following Research Questions (RQs):

- **RQ1 (Effectiveness):** *How effectively does COGNIXSHIELD identify vulnerable API usage in large projects?*
- **RQ2 (Component Contribution):** *How does each of COGNIXSHIELD’s three core components contribute to its effectiveness?*
- **RQ3 (Baseline Comparison):** *How well does COGNIXSHIELD compare with SOTA solutions?*
- **RQ4(Robustness):** *How robust is COGNIXSHIELD’s vulnerability detection capability across different LLM-related configurations?*

Datasets. To evaluate our tool, we construct the evaluation dataset based on the following sources. The foundation of our dataset comes from Transfer [23], a third-party benchmark that provides paired Java projects and their vulnerable libraries. Due to compatibility issues in Transfer, many vulnerable libraries lack corresponding client projects. To enlarge the dataset, we search on GitHub using vulnerable library names as keywords and retrieve the top 100 Java projects for each vulnerable library. We then filter these projects based on vulnerable library version and actual API invocation. The resulting dataset includes 57 projects corresponding to 26 distinct vulnerabilities. To establish ground truth, two security experts independently analyze and cross-verify the exploitability of all projects in the dataset. Among them, 40 applications are confirmed to be vulnerable where PoV tests

■ **Table 1** The precision, recall, F1-score and accuracy of tools measured based on 57 projects.

CWE Category	GT		CognixShield(%)				NaiveLLM(%)				Codex(%)				VAScanner(%)				Steady (%)				DeepDFA(%)				LineVul(%)			
	TP	TN	P	R	F	A	P	R	F	A	P	R	F	A	P	R	F	A	P	R	F	A	P	R	F	A	P	R	F	A
CWE-20 (13)	9	4	89	89	89	85	75	25	38	23	82	100	90	85	75	100	86	75	83	45	59	46	50	22	31	31	67	22	33	38
CWE-22 (6)	5	1	83	100	91	83	100	17	29	17	80	80	80	67	0	-	-	0	67	40	50	33	100	20	33	33	100	20	33	33
CWE-78 (7)	6	1	100	100	100	100	-	0	-	14	83	83	83	71	100	100	100	100	100	29	44	29	50	17	25	14	75	50	60	43
CWE-91 (3)	2	1	67	100	80	67	100	33	50	33	67	100	80	67	100	100	100	100	-	0	-	33	-	0	-	33	-	0	-	33
CWE-180 (3)	2	1	67	100	80	67	100	100	100	100	67	100	80	67	100	33	50	33	-	0	-	0	100	50	67	67	-	0	-	33
CWE-330 (4)	2	2	100	100	100	100	100	100	100	100	-	0	-	50	0	-	-	0	-	0	-	0	50	50	50	50	50	50	50	50
CWE-502 (4)	3	1	100	100	100	100	0	-	-	0	100	33	50	50	-	-	-	-	67	100	80	67	-	0	-	25	0	0	-	0
CWE-611 (2)	2	0	100	100	100	100	100	100	100	100	100	100	100	100	-	-	-	-	-	-	-	-	-	0	-	0	100	50	67	50
CWE-770 (6)	4	2	80	100	89	83	100	100	100	100	60	75	67	50	50	100	67	50	50	20	29	17	-	0	-	33	100	75	86	83
CWE-787 (3)	1	2	33	100	50	33	-	-	-	-	33	100	50	33	-	-	-	100	0	0	-	0	-	0	-	67	-	0	-	67
CWE-835 (4)	3	1	100	100	100	100	100	100	100	100	75	100	86	75	50	33	40	25	0	0	-	0	-	0	-	25	-	0	-	25
CWE-1025 (2)	1	1	-	0	-	50	-	-	-	-	-	0	-	0	-	-	-	100	0	-	-	0	-	0	-	50	50	100	67	50
Overall (57)	40	17	84	95	89	84	84	28	42	30	73	80	76	65	64	23	33	25	60	30	40	28	60	15	33	24	71	30	42	42

"-" means that a tool does not report vulnerability of API usage for applications, so the related precision and F-score values cannot be calculated.

are successfully generated to demonstrate vulnerabilities caused by the vulnerable API usage in these applications. The remaining 17 projects are verified as unaffected. The complete project list and labels are provided in the replication package (see Data Availability). On average, each project contains 1,685 Java files and 193,541 lines of code.

Metrics. Following prior work [41], we use four metrics to evaluate COGNIXSHIELD:

Precision (P): The ratio of correctly identified vulnerable projects to the total number of projects flagged as vulnerable by the tool. $P = \frac{\# \text{ of correctly identified vulnerable projects}}{\text{Total \# of projects flagged as vulnerable}}$

Recall (R): The ratio of correctly identified vulnerable projects to the total number of actual vulnerable projects in the ground truth dataset.

$R = \frac{\# \text{ of correctly identified vulnerable projects}}{\text{Total \# of vulnerable projects in ground truth}}$

F1-score (F1) is the harmonic mean of precision and recall; it reflects the trade-off between precision and recall. $F1 = \frac{2 \times P \times R}{P + R}$

Accuracy (A): The ratio of correctly classified projects to the total number of projects in the ground truth dataset. $A = \frac{\# \text{ of correctly classified projects}}{\text{Total \# of projects in ground truth}}$

5.1 RQ1 (Effectiveness of CognixShield)

We evaluate COGNIXSHIELD on the 57 open-source Java projects using the same hyperparameters as Section 4. COGNIXSHIELD analyzed 57 applications that invoke the vulnerable API and correctly classified 48 of them as shown in Table 1. Specifically, among the 40 projects confirmed to be vulnerable, COGNIXSHIELD successfully identified vulnerabilities in 38 of them. For the 17 projects verified as secure, 10 were correctly classified as unaffected. Overall, COGNIXSHIELD achieves a P of 84%, R of 95%, F1 of 89%, and A of 84%. We then manually inspected the running logs on the evaluated applications to understand the effectiveness of COGNIXSHIELD. The nine incorrect detections fall into three categories:

1) *Incomplete context retrieval.* Oversized AST leaf nodes exceeded the fragment size limit, forcing semantic fragmentation that disrupted contextual reconstruction. Three false positives arose when large leaf nodes, like string literals, exceeded the token limit and could not fit within a single fragment. Since leaf nodes are syntactically atomic, COGNIXSHIELD’s hierarchical decomposition had to split them at arbitrary points, disrupting their semantic meaning. For instance, in Findbugs [14], a long statement in `branchOperations` was split mid-expression, preventing COGNIXSHIELD from reconstructing the inter-procedural context. Such fragmentation makes it challenging for COGNIXSHIELD to reconstruct the relevant context needed for accurate analysis.

2) *Relevant code fragments missed.* Large, security-critical code blocks were often missed because their embeddings were dissimilar to the retrieval query. Four failures (2 FPs, 2 FNs) resulted from embedding-based retrieval, missing large but relevant code blocks. For

example, in the Badlion project [24], the 40-50 LOC `URIUtils` class containing the vulnerable API fell below the similarity threshold. Instead, COGNIXSHIELD retrieved the API's library implementation, misinterpreting library code as application logic and producing a false positive.

3) *Reasoning failures for vulnerability semantics.* Two false positives occurred when LLM reasoning introduced assumptions beyond those specified in the PoV. For example, in `ia-recruiter` application [50], the code hardcodes the ZIP file path to prevent directory traversal. However, COGNIXSHIELD assumed attackers could replace the hardcoded file itself, incorrectly flagging it as vulnerable to CVE-2019-10094 [33] despite the PoV indicating otherwise. While prompt engineering with additional constraints helps, LLMs inevitably generate assumptions beyond provided information, making these reasoning errors difficult to eliminate entirely.

Finding 1: *COGNIXSHIELD correctly classifies 38 vulnerable and 10 secure applications, achieving a precision of 84%, recall of 95%, F1-score of 89%, and accuracy of 84% across 57 applications. The unbalanced code structures can disrupt AST-guided fragmentation and degrade the quality of retrieved context.*

Upon further inspection, we observed that the embedding cost increases with project size because the entire codebase must be indexed. For example, the largest project, Hadoop (2.63M LoC), requires 19.8M embedding tokens and takes approximately 76 minutes to generate embeddings. However, the overall analysis cost is primarily determined by the number of vulnerable API invocations rather than project size. Among the two projects exceeding 1M LoC, Hadoop and Druid (1.21M LoC), our approach successfully analyzed both: Hadoop requires 339K input tokens and 7K output tokens for LLM analysis (approximately \$1.30), whereas Druid requires 2.4M input tokens and 360K output tokens (approximately \$10.02). This difference is mainly because Druid contains 28 vulnerable API invocations, compared to only 8 in Hadoop. On average, our approach used 1,772,500 tokens for embedding and 174,196 LLM tokens (88.2% input and 11.8% output), resulting in an average cost of approximately \$0.60 per project analysis.

Finding 2: *COGNIXSHIELD tool analyzes large-scale codebases at low cost, for an average of \$0.60 per project analysis. This cost is primarily determined by the number of vulnerable API invocations rather than project size, keeping COGNIXSHIELD practical for multi-million-LoC repositories.*

5.2 RQ2 (Component Contribution)

Experiment setting. To better understand the contribution of each component, we conducted an ablation study to evaluate the effectiveness of COGNIXSHIELD under the following settings: (1) w/o AST, where the AST-guided code segmentation is replaced with naive line-based chunking; and (2) w/o RAG, where only the target function that invokes the vulnerable API is provided as context; and (3) w/o PoV, where the PoV test is not included.

As shown in Table 2, all three settings led to substantial performance degradation. Specifically, without AST, recall dropped by 41% and accuracy dropped by 27%, while no RAG reduced recall by 42% and accuracy by 34%. Excluding PoV guidance decreased both precision and recall by 12%. We observed that both AST-based indexing and RAG are critical for retrieving security-relevant inter-procedural context. Without AST-based indexing, COGNIXSHIELD loses structural information needed to identify semantically coherent code fragments. For example, when analyzing CVE-2022-25845 in the `commerce` project, the

■ **Table 2** Ablation experiment results of COGNIXSHIELD components.

Model	P (%)	R (%)	F(%)	A(%)
CognixShield	84	95	89	84
CognixShield w/o AST	79	54	64	57
CognixShield w/o RAG	70	53	60	50
CognixShield w/o PoV	72	83	77	65

vulnerability is exploitable only when `JSON.parseObject` deserializes malformed JSON strings. Without AST-based indexing, the retrieved code snippets consist primarily of isolated variable assignments and omit the surrounding control-flow structure.

Inter-procedural Context Retrieval. We observed that COGNIXSHIELD retrieves security-relevant context on demand, adapting its retrieval scope to the vulnerability under analysis rather than reconstructing the entire program. Across 47 projects, COGNIXSHIELD retrieved cross-function and cross-file code fragments to establish exploitability, whereas for the remaining 10 projects, reasoning within a single function was sufficient. Listing 3 shows a simplified prompt automatically generated by *Reflector* for the Nacos project. COGNIXSHIELD performed control-flow related code fragments retrieval across multiple files; it retrieved code blocks from the public interface `createUser` down to the vulnerable API invocation `PasswordEncoderUtil.encode`. While in the Kylin project (Listing 3), COGNIXSHIELD gathered data constraints context instead of the entire control flow. It includes the `checkNewPwdRule` function, which has security-related logic for password validation. For both cases, the retrieved context enabled COGNIXSHIELD to correctly identify the trigger condition, impact, and exploitation flow. When RAG is disabled, COGNIXSHIELD analyzes only the local function containing the vulnerable API, preventing it from determining whether the invocation is reachable under exploitable conditions. These findings indicate that accurate detection depends on retrieving security-relevant inter-procedural context rather than analyzing the function containing the vulnerable API invocation in isolation.

■ **Listing 3** Context retrieved by *Reflector*: (a) Nacos [7] – cross-file control-flow from `createUser` to the vulnerable API; (b) Kylin [10] – data-constraint chain showing null validation before encoding.

(a) Nacos (vulnerable).

```
Related code snippet:
-----
file_path: PasswordEncoderUtil.java
Code:
public class PasswordEncoderUtil {
    ...
    public static String encode(String raw) {
        return new BCryptPasswordEncoder()
            .encode(raw);
    }
}
-----
file_path: UserController.java
Code:
public Object createUser(
    String username, String password) {
    ...
    userDetailsService.createUser(username,
        PasswordEncoderUtil.encode(password));
    return RestResultUtils.success("ok!");
}
```

(b) Kylin (secure).

```
Related code snippet:
-----
file_path: UserController.java, L252-260
Code:
private void checkNewPwdRule(String newPwd) {
    if (!checkPasswordLength(newPwd)) {...}
}
-----
file_path: UserController.java, L262-264
Code:
private boolean checkPasswordLength(
    String password) {
    return !(password == null
        || password.length() < 8);
}
-----
file_path: UserController.java, L201-238
Code:
public EnvelopeResponse save(
    @RequestBody PasswordChangeRequest user) {
    ...
    checkNewPwdRule(user.getNewPassword());
    if (existing != null) {
        existing.setPassword(
            pwdEncode(user.getNewPassword()));
    }
}
```

PoV-guided Vulnerability Semantic Reasoning. With PoV-guided, COGNIXSHIELD interprets the retrieved code context with vulnerability semantics (e.g., vulnerability’s triggering conditions) with the PoV guidance. For example, to analyze the Kylin and Nacos projects, COGNIXSHIELD first deduces from the PoV test (Listing 1, Section 2.1) that exploitation is only possible when a null password is passed to the `BCryptPasswordEncoder.encode()` method. Leveraging this semantic understanding, the *expert* agent reasons over the retrieved code fragments, examining control flow and data constraints surrounding each vulnerable API invocation. In the Kylin project (Listing 4), COGNIXSHIELD identifies built-in safeguards, including password-pattern validation checks (Listing 3), and correctly determines that the API usage is secure despite the presence of the vulnerable call. Conversely, in the Nacos project (Listing 4), the expert agent’s `poc_understanding` field captures a feasible trigger condition, its resulting impact (e.g., throwing an `IllegalArgumentException` when a null password is provided), and the exploitation path. Without PoV guidance, COGNIXSHIELD lacks vulnerability-specific semantics to validate exploitation conditions and paths, causing it to classify both projects as vulnerable based solely on vulnerable API usage.

■ **Listing 4** COGNIXSHIELD’s analysis: (a) insecure usage in Nacos [7]: exploit triggered by null password; (b) secure usage in Kylin [10]: mitigated by input validation.

(a) Nacos (vulnerable).

```
"is_vulnerable": true,
"analysis":
  "poc_understanding":
    "category": "Special Trigger Flow",
    "attack_mechanism":
      "vulnerable_api": "BCryptPasswordEncoder.encode",
      "required_conditions":
        "condition": "raw password is null",
        "explanation": "The encode method of BCryptPasswordEncoder
        ↳ throws an IllegalArgumentException when the raw password
        ↳ is null.",
        "attack_flow": "When a null value is passed to the encode
        ↳ method, it triggers an IllegalArgumentException.",
        "impact": "Potential denial of service if the application does
        ↳ not handle the exception properly."
    "code_analysis":
      "pattern_match":
        "matches": true,
        "explanation": "The code uses BCryptPasswordEncoder.encode, which
        ↳ is the vulnerable API identified in the POC."
      ...
    "conclusion":
      "verdict": true,
      "explanation": "The code is vulnerable because it directly uses
      ↳ the encode method without checking for null inputs, which can
      ↳ lead to an IllegalArgumentException.",
```

(b) Kylin (secure).

```
"is_vulnerable": false,
"analysis":
  ...
  "code_analysis":
    "pattern_match":
      "matches": true,
      "explanation": "The code uses BCryptPasswordEncoder.encode() in
      ↳ pwdEncode() method",
    "mitigations": [
      "type": "Input Validation",
      "effectiveness": "Strong - Multiple layers of validation prevent null
      ↳ passwords"
    ],
    "type": "Pattern Matching",
    "effectiveness": "Strong - Passwords must match specific pattern"
  ]
  "conclusion":
    "verdict": false,
    "explanation": "The code has multiple safeguards preventing null
    ↳ passwords from reaching the encoder.\n1. Password pattern
    ↳ validation (checkNewPwdRule)\n2. BCrypt pattern checking\n3. Input
    ↳ validation in the controller"
```

Finding 3: *Each component plays a critical role in COGNIXSHIELD’s effectiveness. AST-based indexing and RAG enable retrieval of security-relevant inter-procedural context, whereas PoV-guided semantic reasoning allows COGNIXSHIELD to understand vulnerability-specific trigger conditions, impacts, and exploitation paths.*

5.3 RQ3 (Baseline Comparison)

To assess the performance of COGNIXSHIELD relative to existing solutions, we compare it against naive LLM and four baseline tools from two categories: static programming analysis tools and deep learning-based software approaches.

- **Naive LLM** is our baseline, where we use GPT-4o and provide the entire codebase with a system prompt that includes the PoV test.
- **Codex** [36] is an OpenAI LLM agent that autonomously explores and analyzes repositories.

■ **Table 3** Runtime performance of five tools.

Tools	Time (Total)	Time (Average)	Failed Cases
CognixShield	275 mins	5 mins	0
Naive LLM	3 mins	9 secs	38
Codex	122 mins	128 secs	0
Eclipse Steady	722 mins	13 mins	18
VAScanner	41 mins	2 mins	35
DeepDFA	364 mins	6 mins	0
LineVul	361 mins	6 mins	0

- **VAScanner** [2] uses call-graph analysis to determine whether vulnerable APIs from library dependencies are reachable and actually invoked within the application code.
- **Eclipse Steady** [40] performs reachability analysis by combining static and dynamic techniques, tracing execution paths from known vulnerable APIs to their invocation points in the application. It then produces detailed assessments of vulnerability exploitation.
- **LineVul** [19] is a transformer-based model leveraging CodeBERT [17] for code representation and vulnerabilities at line level.
- **DeepDFA** [45] combines dataflow analysis (DFA) with graph neural networks (GNNs) for detecting software vulnerabilities. It encodes dataflow information at each node of a program’s control flow graph and learns data usage patterns.

Experiment settings. To ensure a fair comparison, we run experiments for all tools on the same machine. Naive LLM uses GPT-4o with a 128K context window, packing each codebase’s source code with Repomix 1.6.0 [53]. Codex uses the gpt-5.2 model in full-auto mode, with the PoV test and vulnerable API signature as the task prompt; we disable web search to ensure the agent relies solely on repository exploration. VAScanner runs on its published code [2] with default settings, with program-issue fixes we applied, while Eclipse Steady version 3.2.5 runs with its default configuration; we set a 150-minute per-project analysis time limit for both tools. For the machine-learning-based tools LineVul and DeepDFA, we use their default parameter settings with function-level inputs. Since neither tool was designed for project-level exploitability assessment, we include them as reference. For COGNIXSHIELD, we use the same settings as described in Section 4.

CognixShield achieves an overall F1 of 89% and A of 84%, outperforming Naive LLM (F1: 42%, A: 30%), Codex (F1: 76%, A: 65%), LineVul (F1: 42%, A: 42%), EclipseSteady (F1: 40%, A: 28%), VAScanner (F1: 33%, A: 25%), and DeepDFA (F1: 33%, A: 24%). The naive LLM achieved only 28% recall across 57 projects; due to the agentic RAG design, COGNIXSHIELD uses $5.7\times$ fewer LLM tokens at significantly lower cost.

Although Codex outperforms all other baselines, COGNIXSHIELD is still ahead by 13% in F1 and 19% in accuracy. Codex’s iterative grep-and-read strategy struggles with cross-file reachability: when a vulnerable sink requires traversing multi-file chains (AOP aspects, factories, runtime configuration), it commits to a verdict before resolving the full chain, whereas COGNIXSHIELD’s RAG retrieval co-locates all relevant fragments in a single prompt. For example, in `PLMCodeTemplate` (CVE-2020-26217), the agent found no live callers of `XMLConfig.getBean()` and concluded the sink was dead code. The sink is in fact reachable via a four-file Spring AOP chain: a `@Log`-annotated controller triggers `LogAOP`, which evaluates attacker-controlled SPEL via `SPELUtil`, enabling `T(XMLConfig).toBean(...)` as a gadget. The agent read `XMLConfig.java` but never traversed the annotation-to-aspect-to-SPEL path, while COGNIXSHIELD’s retrieval co-located all four files in a single context, correctly identifying the project as vulnerable.

VAScanner achieves precision of 64%, but yields low recall (23%) due to scalability failures on large codebases. Eclipse Steady exhibits a similar trend, achieving 30% R and an F1 of 40%, failing to analyze 18 projects due to limitations in building call graphs for complex projects. The DL-based tools, LineVul and DeepDFA, achieve low recall rates of 30% and 15%, respectively. These approaches overlook a substantial number of vulnerable API usages. This limitation may be due to their function-level detection strategy, where each function is analyzed in isolation. Detecting vulnerable API usage, however, often requires a broader context that spans across multiple functions and files.

Runtime Performance. As shown in Table 3, COGNIXSHIELD averages 5 minutes per project (275 minutes total) and completed all 57 applications without failures. Eclipse Steady required 722 minutes and failed on 18 projects; VAScanner finished in 41 minutes but failed on 35 cases. DL-based tools averaged 6 minutes per project including training time. COGNIXSHIELD eliminates training requirements and achieves 100% project coverage, making it more practical for real-world deployment.

Finding 4: *CognixShield outperforms all baselines across every metric (F1 89%, accuracy 84%). The strongest baseline, the Codex agent, reaches 76% F1 and 65% accuracy, while the remaining static-analysis, DL-based, and naive-LLM baselines fall below 42% accuracy. It demonstrates better scalability and efficiency on large real-world projects, overcoming memory and dependency challenges faced by existing approaches.*

5.4 RQ4 (Robustness Analysis)

Experiment settings. We evaluate COGNIXSHIELD on two popular text embedding models *voyage-code-3* [6], and *OpenAI text-embedding-3-small* [34]; five state-of-the-art closed-source LLMs and one open-source LLM: GPT-4o [35], GPT-o1 [35], CLAUDE-3.5-SONNET [8], GOOGLE-GEMINI-2.0-FLASH [21] and LLAMA-3.1-405B [5], while varying the maximum code fragment size (θ) over 500, 1000, 1500, 2000, 2500, and 3000. Figure 3 presents the performance results across four key metrics (P, R, F1, A). The first row shows results using the *voyage-code-3* embedding, while the second row shows results using the *OpenAI text-embedding-3-small* embedding.

Embedding Models. COGNIXSHIELD shows minimal sensitivity to the choice of embedding models. The *openAI* embedding achieves more stable precision and the highest F1 of 90% (P: 84%, R: 95%) with GPT-4o at $\theta = 2500$ tokens, while the *voyage* embedding consistently yields higher recall (up to 96% for GPT-4o) but lower precision, with recall differences reaching up to 12% between the two embeddings.

Impact of LLMs. Across five different LLMs, COGNIXSHIELD achieves precision ranging from 74% to 87% and recall from 53% to 96%. GPT-4o achieves the highest overall performance (F1: 89%, A: 84%), while LLAMA-3.1-405B reaches the highest precision (87%) but the lowest recall (53%–66%), resulting in the worst F1 (below 72.5%) and accuracy (below 66%). These findings indicate that the choice of LLM substantially affects detection effectiveness, with GPT-4o demonstrating the strongest capability.

Code Fragmentation Length. Most models maintain stable precision across fragment sizes, while recall improves for GPT-4o and GOOGLE-GEMINI-2.0-FLASH as fragment length increases (e.g., GPT-4o’s recall grows from 91% at 500 tokens to 95% at 2500 tokens). Beyond 2500 tokens, some models show degradation: GPT-o1’s F1 drops from 83% to 72%

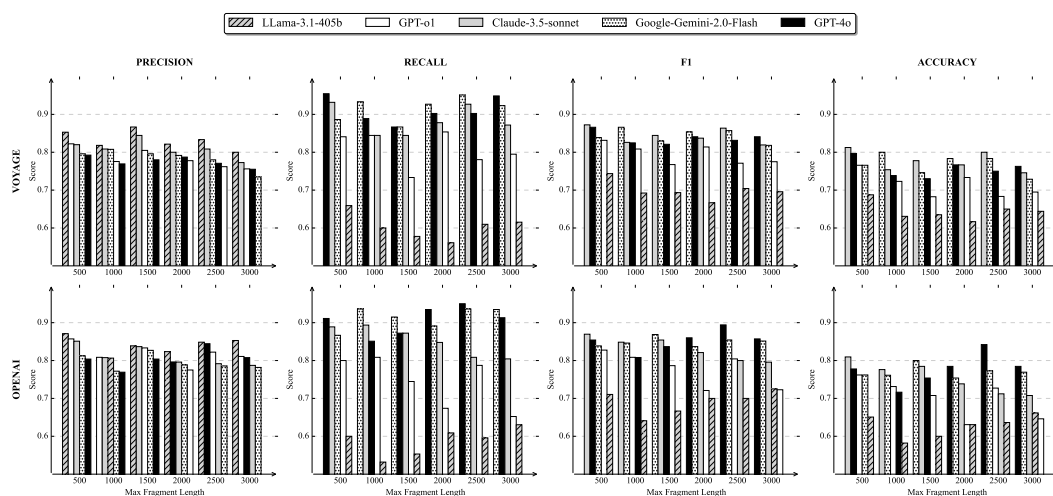


Figure 3 Performance evaluation of LLM-embedding model combinations across six code fragment lengths. The top legend denotes the LLMs, while the left legend indicates the embedding models. Each subplot corresponds to a distinct evaluation metric as listed (Precision, Recall, F1-score, and Accuracy).

when increasing from 500 to 3000 tokens, likely due to overly long fragments introducing noise. This suggests that 2000–2500 tokens represents the optimal balance for large-scale vulnerability detection.

Finding 5: *COGNIXSHIELD demonstrates stable performance across embedding models and fragment sizes, while LLM choice has a substantial impact on detection effectiveness (F1: 64%–89%, A: 58%–84%). GPT-4o consistently achieves the best results (F1: 81%–89%, A: 75%–84%), whereas LLAMA-3.1-405B underperforms (F1: 64%–73%). Code segments of 2,000–2,500 tokens with openAI embeddings yield the highest overall performance.*

6 Threats to validity

Internal Validity. 1) **Parameter Sensitivity:** Our approach’s performance may be sensitive to parameters like similarity thresholds and LLM settings. While we set the temperature to 0 for reproducibility, other parameters (e.g., similarity threshold) may still influence results. 2) **Ground Truth Accuracy:** The manual labeling data requires significant effort. To ensure labeling accuracy, two authors who have industry experience in security domains labeled the data. However, this approach might impact the overall accuracy of the classification. 3) **Prompt Template Limitation:** We used fixed prompt templates for the LLM. This could limit the range of vulnerabilities detected and affect the accuracy of our results. 4) **Implementation Choices:** COGNIXSHIELD’s effectiveness may be influenced by specific implementation choices (e.g., AST parser, encoder models).

External Validity. The generalizability of our findings may be limited by the following factors: 1) The principle of our retrieval-augmented analysis is language-agnostic, but our concrete implementation includes Java-specific components, so the generalizability of our quantitative findings to other languages is not directly established. We chose to instantiate the approach on Java, motivated by Java’s prevalence in open-source software and growing

security concerns over supply chain attacks in popular Java libraries. 2) Our observations and conclusions are based on the third-party datasets [23] complemented by additional data collected from GitHub. While these datasets offer valuable insights, they do not cover all possible scenarios or vulnerabilities in real-world applications. 3) Our study relies on exemplar security tests from libraries to illustrate vulnerability contexts. According to [23], over 51% of vulnerabilities have associated test cases. While this approach is effective for evaluation, it may be limited by the availability of PoV tests.

7 Related Work

Vulnerable API Detection. Researchers and engineers created tools to detect the invocation of vulnerable APIs or misuse of APIs [43, 40, 56, 18]. These tools are designed for identifying insecure cryptographic API usage patterns that may introduce security risk. Specifically, tools such as CryptoTutor [43], and SEADER [56] adopted static analysis to identify whether Java cryptographic APIs are invoked with secure parameter configurations and correct sequential orders. Fischer et al. [18] trained models on labeled code snippets that demonstrate both secure and insecure cryptographic API usage. These models are then applied to detect potentially insecure API usage patterns in new Java code snippets, learning from examples to identify code that deviates from secure usage patterns. For vulnerable APIs introduced through library dependencies, version-based scanners such as DependencyCheck [37] match declared dependency versions against vulnerability databases, whereas reachability-based tools decide whether the vulnerable API is actually invoked: VAScanner [54] and Eclipse Steady [40] build call graphs from the application to the vulnerable API, and ViperScan [46] further uses an LLM to distill exploit conditions from security patches into static reachability rules. These tools rely on syntactic patterns or reachability, and do not reason about whether a reached API is actually exploitable in the surrounding program context.

Deep Learning-based Software Vulnerability Detection. Deep learning-based software vulnerability detector approaches [19, 28, 30, 45, 13, 29] have been proposed to automatically learn vulnerability patterns from various code representations. IVDetect [28] utilized RNN-based models to generate code representations from source code and identify vulnerable code patterns. LineVul [19] leverages CodeBERT, a pre-trained model for programming language, to generate code vector representations and employs BERT’s self-attention layers to capture long-term dependencies in code sequences. DeepDFA [45] encodes dataflow analysis information into graph neural networks to better capture program semantics for vulnerability detection. These methods often focus on detecting vulnerabilities within individual functions or code snippets rather than considering the broader inter-procedural context required for dependency vulnerability detection.

LLM for Vulnerability Detection. Recent work has explored LLMs’ capabilities in security tasks [49, 57, 31]. SecLLMHolmes [49] evaluates the ability of various LLMs to detect security-related bugs under different configurations, concluding that current models are non-deterministic and not robust. Liu et al. [31] introduce GPTAid which adopted LLM to infer API misuse parameter rules and adopts existing tools (CodeQL [20]) for detection. Several studies investigated vulnerable code repair [38, 22, 52] through LLMs. For instance, Pearce et al. [38] examine LLMs in repairing vulnerable code in a zero-shot prompt, finding that while models can successfully repair hand-crafted examples, they struggle with complex, real-world cases. To move beyond single functions, Vul-RAG [16] retrieves distilled vulnerability

knowledge to judge individual functions, and LLMxCPG [25] uses code-property-graph slices to steer the LLM toward vulnerability-relevant code, but both remain function- or slice-level rather than repository-scale. COGNIXSHIELD instead combines AST fragmentation, embedding retrieval, agentic RAG, and LLM reasoning to assemble repository-scale inter-procedural context and determine exploitability.

8 Conclusion

We presented COGNIXSHIELD, a novel Agentic RAG approach for detecting vulnerable API usage in large-scale codebases, which combines inter-procedural context retrieval with LLM-driven vulnerability code semantic reasoning. Unlike traditional analyzers, COGNIXSHIELD constructs repo-level program contexts capturing control flow and data constraints, enabling accurate distinction between exploitable and safe API usages. This work demonstrates that LLM-based techniques, when integrated with domain knowledge, can make vulnerability detection more practical and scalable. Future directions include extending COGNIXSHIELD to support additional languages, such as Golang and JavaScript, developing adaptive context retrieval strategies that adjust to code complexity, and enhancing interpretability by providing actionable fix suggestions alongside detection results.

References

- 1 OWASP Dependency-Check. <https://owasp.org/www-project-dependency-check/>, 2020.
- 2 VAScanner. <https://github.com/VAScanner/VAScanner>, 2024. Accessed: 2024-01-22.
- 3 A03:2025 Software Supply Chain Failures. https://owasp.org/Top10/2025/A03_2025-Software_Supply_Chain_Failures/, 2026.
- 4 Dependency-check vulnerabilities | Snyk. <https://security.snyk.io/package/npm/dependency-check>, 2026.
- 5 Meta AI. Meta llama 3.1 announcement. <https://ai.meta.com/blog/meta-llama-3-1/>, 2024. Accessed: 2024-10-30.
- 6 Voyage AI. Voyage code 3. <https://blog.voyageai.com/2024/12/04/voyage-code-3/>. Published: 2024-12-04, Accessed: 2025-01-20.
- 7 Alibaba. PasswordEncoderUtil.java/UserController.java - Apache Kylin, 2025. Accessed: 2025-01-19. URL: <https://github.com/alibaba/nacos/tree/a19d3fd>.
- 8 Anthropic. Claude sonnet. <https://www.anthropic.com/claude/sonnet>, 2024. Accessed: 2024-10-30.
- 9 Apache Software Foundation. KylinUserService.java - Apache Kylin, 2025. Accessed: 2025-01-19. URL: <https://github.com/apache/kylin/tree/443c252>.
- 10 Apache Software Foundation. UserController.java - Apache Kylin, 2025. Accessed: 2025-01-19. URL: <https://github.com/apache/kylin/tree/443c252>.
- 11 Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Self-reflective retrieval augmented generation. *arXiv preprint*, 2023. [arXiv:2310.11511](https://arxiv.org/abs/2310.11511).
- 12 Max Brunsfeld and contributors. Tree-sitter java. <https://github.com/tree-sitter/tree-sitter-java>, 2024. Accessed: 2024-10-30.
- 13 Sicong Cao, Xiaobing Sun, Xiaoxue Wu, David Lo, Lili Bo, Bin Li, and Wei Liu. Coca: Improving and explaining graph neural network-based vulnerability detection systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3639168.
- 14 FindBugs Project Contributors. Bugtreemodel.java. <https://github.com/tonydamage/findbugs/tree/fd7ec8b>, 2023. Accessed: 2024-10-30.

- 15 Yangruibo Ding, Jinjun Peng, Marcus J. Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. Semcoder: Training code language models with comprehensive semantics reasoning. In *Advances in Neural Information Processing Systems*, volume 37, 2024.
- 16 Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *ACM Transactions on Software Engineering and Methodology*, 2025.
- 17 Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint*, 2020. [arXiv:2002.08155](https://arxiv.org/abs/2002.08155).
- 18 Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136. IEEE, 2017. doi:10.1109/SP.2017.31.
- 19 Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620, 2022. doi:10.1145/3524842.3528452.
- 20 GitHub. Codeql, 2019. Accessed: 2025-08-05. URL: <https://codeql.github.com/>.
- 21 Google. Gemini ai. <https://gemini.google.com/>. Accessed: 2025-01-20.
- 22 Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1162–1174, September 2023. doi:10.1109/ASE56229.2023.00181.
- 23 Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S Păsăreanu, and David Lo. Test mimicry to assess the exploitability of library vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 276–288, 2022.
- 24 Bastian Leicht. Badlion source code - uriutils.java, 2024. Accessed: 2024-10-30. URL: <https://github.com/bastianleicht/badlion-src/tree/93a099e>.
- 25 Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. LLMxCPG: Context-aware vulnerability detection through code property graph-guided large language models. In *34th USENIX Security Symposium (USENIX Security 25)*, 2025.
- 26 Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.
- 27 Qiang Li, Jinke Song, Dawei Tan, Haining Wang, and Jiqiang Liu. Pdgraph: A large-scale empirical study on project dependency of security vulnerabilities. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 161–173, 2021. doi:10.1109/DSN48987.2021.00031.
- 28 Yi Li, Shaohua Wang, and Tien N. Nguyen. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*, pages 292–303, 2021. doi:10.1145/3468264.3468597.
- 29 Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(04):2244–2258, July 2022. doi:10.1109/TDSC.2021.3051525.
- 30 Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Network and Distributed Systems Security (NDSS) Symposium*, 2018.
- 31 Jinghua Liu, Yi Yang, Kai Chen, and Miaoqian Lin. Generating api parameter security rules with llm for api misuse detection. *arXiv preprint*, 2024. doi:10.48550/arXiv.2409.09288.

- 32 Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212:112031, 2024. doi:10.1016/J.JSS.2024.112031.
- 33 National Vulnerability Database (NVD). Cve-2019-10094 - apache tika xml external entity (xxe) vulnerability, 2019. Accessed: 2025-03-14. URL: <https://nvd.nist.gov/vuln/detail/CVE-2019-10094>.
- 34 OpenAI. Embedding models: Openai documentation. <https://platform.openai.com/docs/guides/embeddings/embedding-models>, 2024. Accessed: 2024-10-30.
- 35 OpenAI. Model overview. <https://platform.openai.com/docs/models>, 2024. Accessed: 2024-10-30.
- 36 OpenAI. OpenAI Codex CLI. <https://github.com/openai/codex>, 2025. Accessed: 2025.
- 37 OWASP Foundation. OWASP Dependency-Check. <https://owasp.org/www-project-dependency-check/>, 2024. Accessed: 2024-01-22.
- 38 Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining Zero-Shot Vulnerability Repair with Large Language Models . In *IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356, May 2023. doi:10.1109/SP46215.2023.10179324.
- 39 JULIE PETERSON. The Scariest Things About SCA. <https://devops.com/the-scariest-things-about-sca/>, 2022.
- 40 Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25(5):3175–3215, 2020. doi:10.1007/S10664-020-09830-X.
- 41 Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, pages 2455–2472, 2019. doi:10.1145/3319535.3345659.
- 42 Aditi Singh, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. Agentic retrieval-augmented generation: A survey on agentic rag. *arXiv preprint*, 2025. doi:10.48550/arXiv.2501.09136.
- 43 Larry Singleton, Rui Zhao, Myoungkyu Song, and Harvey Siy. Cryptotutor: Teaching secure coding practices through misuse pattern detection. In *Proceedings of the 21st Annual Conference on Information Technology Education*, pages 403–408, 2020. doi:10.1145/3368308.3415419.
- 44 spring-projects/spring-security. Fix issue (commit d1909ec). <https://github.com/spring-projects/spring-security/commit/d1909ec>, 2025. Accessed: 2025-07-29.
- 45 Benjamin Steenhoek, Hongyang Gao, and Wei Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3623345.
- 46 Fute Sun, Lei Zhang, Zhiyu Wu, Xinyu Chen, Haitao Yu, Tianyang Han, and Min Yang. Fine-grained detection of java cross-library vulnerability propagation by extracting semantic constraints from security patches. *IEEE Transactions on Dependable and Secure Computing*, 23(4):8334–8348, 2026. doi:10.1109/TDSC.2026.3682678.
- 47 Tian Tan and Yue Li. Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1093–1105. ACM, 2023. doi:10.1145/3597926.3598120.
- 48 Qdrant Team. Qdrant: Vector database for the next generation of ai applications. <https://qdrant.tech/>, 2024. Accessed: 2024-10-30.
- 49 Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks . In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 862–880, May 2024. doi:10.1109/SP54263.2024.00210.

- 50 Wlainer. Basicnamefinder.java - ia recruiter, 2025. Accessed: 2025-03-14. URL: <https://github.com/wlainer/ia-recruiter>.
- 51 Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1046–1058. IEEE, 2023. doi:10.1109/ICSE48619.2023.00095.
- 52 Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated Program Repair in the Era of Large Pre-trained Language Models . In *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494, May 2023. doi:10.1109/ICSE48619.2023.00129.
- 53 Kazuki Yamada. Repomix. <https://github.com/yamadashy/repomix>, 2024.
- 54 Fangyuan Zhang, Lingling Fan, Sen Chen, Miaoying Cai, Sihan Xu, and Lida Zhao. Does the vulnerability threaten our projects? automated vulnerable api detection for third-party libraries. *IEEE Transactions on Software Engineering*, 2024.
- 55 Ying Zhang, Wenjia Song, Zhengjie Ji, Danfeng Yao, and Na Meng. How Can ChatGPT Support Human Security Testers to Help Mitigate Supply Chain Attacks? . *IEEE Transactions on Software Engineering*, 52(02):509 – 526, 2025. doi:10.1109/TSE.2025.3632765.
- 56 Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng (Daphne) Yao, and Na Meng. Example-based vulnerability detection and repair in java code. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC ’22*, pages 190–201, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3524610.3527895.
- 57 Xin Zhou, Ting Zhang, and David Lo. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 47–51, 2024. doi:10.1145/3639476.3639762.