SAVANT: Vulnerability Detection in Application Dependencies through Semantic-Guided Reachability Analysis

Wang Lingxiang* *Independent Researcher*
Quanzhi Fu* *Virginia Tech*
Wenjia Song *Virginia Tech*
Gelei Deng *Nanyang Technological University*
Yi Liu *Nanyang Technological University*
Dan Williams *Virginia Tech*
Ying Zhang *Wake Forest University*

Abstract

The integration of open-source third-party library dependencies in Java development introduces significant security risks when these libraries contain known vulnerabilities. Existing Software Composition Analysis (SCA) tools struggle to effectively detect vulnerable API usage from these libraries due to limitations in understanding API usage semantics and computational challenges in analyzing complex codebases, leading to inaccurate vulnerability alerts that burden development teams and delay critical security fixes.

To address these challenges, we proposed SAVANT by leveraging two insights: proof-of-vulnerability test cases demonstrate how vulnerabilities can be triggered in specific contexts, and Large Language Models (LLMs) can understand code semantics. SAVANT combines semantic preprocessing with LLM-powered context analysis for accurate vulnerability detection. SAVANT first segments source code into meaningful blocks while preserving semantic relationships, then leverages LLM-based reflection to analyze API usage context and determine actual vulnerability impacts. Our evaluation on 55 real-world applications shows that SAVANT achieves 83.8% precision, 73.8% recall, 69.0% accuracy, and 78.5% F1-score, outperforming state-of-the-art SCA tools.

1 Introduction

Software supply chain attacks through vulnerable open-sourced third-party library APIs pose a growing threat to Java applications [3, 36, 69]. Although these libraries offer various APIs to accelerate functionality implementation, they often contain known vulnerabilities [48, 77] (e.g., log4 shell vulnerability [11]). These vulnerabilities are publicly documented [5], making the information accessible to anyone, including potential attackers. Therefore, when developers integrate these APIs into their projects, they may unintentionally introduce security weaknesses into their applications, creating potential attack vectors for system exploitation [34, 36, 75].

Recent studies have shown that more than 80% of Java applications contain at least one known vulnerability inherited from their third-party dependencies [48,77]. Moreover, attackers can exploit these vulnerabilities through vulnerable APIs, propagating the attack through the software supply chain and further compromising applications built on these dependencies. Such attacks have increased by 1300% [62] since 2020.

Existing Software Composition Analysis (SCA) approaches detect vulnerabilities in library dependencies to mitigate Open Source Software (OSS) supply-chain attacks in software applications [2, 4, 6, 18, 28, 30, 39, 54, 74, 77]. For instance, DependencyCheck [52], which analyzes dependency configuration files and cross-reference library versions against databases of known vulnerabilities. It flags any found matches based on the library version without considering the API usage in the codebase, which leads to false positives - incorrect identification of vulnerabilities when vulnerable APIs are not used. Consequently, these false positives can erode developers' trust [33, 75] and delay critical security updates when vulnerable APIs are not actively used in the codebase.

To address the limitations of version-based vulnerability detection, researchers have developed fine-grained approaches [22, 47, 54, 58, 59, 63, 66, 74] that combine static and dynamic analysis. Eclipse Steady [22] traces execution paths from potential vulnerability points to application entry points to determine if vulnerable APIs are actually reachable. However, comprehensive data-flow and control-flow analysis of complex codebases face computational barriers [22, 66], often causing incomplete analysis or excessive processing times. These limitations result in critical vulnerabilities being overlooked in production systems.

Additionally, assessing the impact of vulnerable APIs from the library extends far beyond theoretical reachability analysis: it requires domain knowledge to understand the API usage semantics within the client application's context. For instance, when a vulnerable API is susceptible to malformed input, evaluating the actual risk of an API susceptible to malformed input requires expert knowledge to determine if proper defensive mechanisms are implemented downstream. Without consider-

*Equal contribution.

ing such code semantics, tools may generate security alerts for non-exploitable scenarios [44, 45]. This combination of computational limitations and semantic analysis gap leads to both miss-reporting and over-reporting of vulnerabilities [34, 69]. This undermines the credibility of vulnerability assessments and places a significant burden on developers, who must manually verify each alert while delaying dependency upgrades due to concerns about breaking changes. Consequently, actual vulnerabilities can persist in the codebase [71, 74], potentially exposing applications to security risks.

Our work. To overcome the limitations in the existing approaches, we propose a novel approach SAVANT leveraging two key insights. First, proof-of-vulnerability (PoV) test cases included in security patches represent a not fully explored yet valuable source that precisely specifies vulnerable API usage patterns. These test cases explicitly demonstrate vulnerable conditions and effectively capture the semantic contexts where vulnerabilities can be exploited. Second, Large Language Models (LLMs) have demonstrated their capabilities in understanding code semantics and programming patterns [46, 65, 67, 78], thereby providing a complementing approach to the program analysis approaches. Building on these insights, SAVANT leverages LLM's semantic understanding and patch-provided PoV tests to assess vulnerable API impacts in client applications through two phases: Phase I *semantic-preserving preprocessing* prepares the application for context-aware analysis by chunking source code into meaningful blocks based on AST structure while preserving metadata like line numbers and file locations. Each block is encoded into vector embeddings and stored with metadata for efficient retrieval, maintaining both semantic and structural relationships. It helps SAVANT to overcomes the limitations of real-world project sizes and provides an interface for SAVANT to quickly search the code by semantic. Phase II *reflection-based vulnerability detection* analyzes API usage by combining PoV test and API information as vectors to locate potential vulnerabilities. It iteratively expands context through reflection-based queries until sufficient information is gathered. The LLM then analyzes this program context with API test functions to determine vulnerability impacts, enabling precise semantic-guided reachability analysis.

Evaluations. We evaluate SAVANT on a third-party benchmark dataset [34] containing 55 Java projects. We test SAVANT's detection effectiveness at the project level and compare it with two state-of-the-art SCA tools, Eclipse Steady [22] and VAScanner [73]. SAVANT achieves a precision of 83.8%, recall of 73.8%, accuracy of 69.0%, and F1 score of 78.5%, outperforming the baselines by at least 14%, 182%, 155%, and 103%, respectively, for these metrics. Through extensive ablation studies, we demonstrate that SAVANT maintains F1 scores between 0.72 and 0.87 across different LLMs and embedding models, with optimal performance achieved at code segment sizes of 2,000-2,500 tokens.

Contributions. In summary, our paper makes the following contributions as belows:

- We develop SAVANT - a novel two-phase approach that combines semantic-preserving preprocessing and reflection-based vulnerability detection. It enables efficient and accurate vulnerability localization in real-world projects through semantic-guided reachability analysis powered by LLM.
- SAVANT supports context-aware analysis of API usage by maintaining both semantic and structural relationships within the Application. We conduct a comprehensive evaluation of SAVANT on a third-party benchmark with 55 real-world applications, our approach outperforms existing approaches across all evaluated metrics.
- We extensively evaluated various LLMs and embedding models for vulnerability detection, testing code segments of different maximum length limits. Our experiments revealed that CLAUDE-3.5-SONNET with OpenAI embeddings achieved the highest F1 score.

2 Threat Model

Our research addresses whether Java applications become vulnerable when using APIs from libraries with known vulnerabilities. When an application uses a vulnerable library API, the application might still be secure if it implements proper security checks. However, determining whether these security checks are adequate requires analyzing the application's implementation. Modern applications are large and complex, containing millions of lines of code and many library dependencies, making manual security analysis infeasible.

We assume attackers have access to public vulnerability information, including library and application source code, CVEs, and proof-of-concept demonstrations. While current version checkers and static analysis tools only check library versions and code patterns, attackers can analyze how vulnerable library APIs interact with application-level security controls to potentially exploit the library vulnerabilities.

Our analysis focuses specifically on vulnerabilities that arise from insufficient or inconsistent security checks around vulnerable library APIs. We assume the application code itself is correctly implemented and not vulnerable to common issues like injection or unsafe deserialization. Our scope is limited to cases where attacker-controlled inputs reach vulnerable library code due to missing or inadequate application checks rather than vulnerabilities in the application's implementation.

3 Motivation Example

We illustrate our approach using a real-world vulnerability in Spring Security, a widely-used framework that provides security functionality APIs for Java applications.

CVE-2020-5408 [1] reveals that in versions 4.2.x prior to 4.2.16, the `BCryptPasswordEncoder.encode` method throws an `NullPointerException` when given a `null` input. Since neither the method's signature nor its implementation indicates this exception, developers may be unaware of the need to handle null inputs, leading to unexpected application crashes that could be exploited for denial-of-service attacks. This vulnerability was later patched in version 5.4.0-M1 by properly handling null inputs with an `IllegalArgumentException`.

[!hbtp]

Listing 1: Proof of Vulnerability of Spring Security Library

```
@Test(expected = IllegalArgumentException.class)
public void encodeNullRawPassword() {
    BCryptPasswordEncoder encoder = new
        BCryptPasswordEncoder();
    encoder.encode(null);
}
```

[!hbtp]

Listing 2: Code snippet from Apache Kylin [13] demonstrating safe usage of `BCryptPasswordEncoder.encode` due to input validation.

```
public EnvelopeResponse save(@RequestBody
    PasswdChangeRequest user) {
    ...
    ManagedUser existing = get(user.getUsername());
    checkUserName(user.getUsername());
    checkNewPwdRule(user.getNewPassword());
    if (existing != null) {
        if (!this.isAdmin() && !pwdEncoder.matches(user.
            getPassword(), existing.getPassword())) {
            throw new BadRequestException("pwd update
                error");
        }
        existing = userService.copyForWrite(existing);
        existing.setPassword(pwdEncode(user.
            getNewPassword()));
        ...
    }
}

private String pwdEncode(String pwd) {
    if (bcryptPattern.matcher(pwd).matches())
        return pwd;
    return pwdEncoder.encode(pwd);
}
```

Listing 2 demonstrates a case where CVE-2020-5408 (null pointer vulnerability in `BCryptPasswordEncoder.encode`) does not affect a project despite using the vulnerable API. This example comes from Apache Kylin [12], a large-scale analytics project with over 2,000 Java files and 600,000 lines of code (commit "443c2523"). While Kylin uses Spring Security version 4.2.14, which contains this vulnerability, the code remains secure because Kylin's implementation guarantees that all user passwords are initialized with non-null values before being passed to the encoder, as shown in Listing 2.

However, existing approaches yield unreliable results when analyzing Kylin's security. Library version checkers produce false positives by missing defensive code patterns, while static analyzers face scalability issues when tracking complex control and data flows across large codebases.

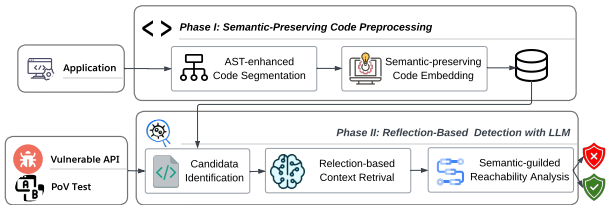

Figure 1: The Overview of SAVANT

Our work. To overcome these limitations, we propose SAVANT, which aims to address two fundamental challenges: (1) efficiently analyzing large codebases without relying on complete call graph construction, and (2) understanding the semantic context of API invocations to determine their true security implications. SAVANT leverages LLMs for semantic code analysis, enabling accurate vulnerability assessment even in a complex codebase.

4 Design

4.1 Overview

To address the limitations of existing approaches in handling large-scale projects, accurately identifying vulnerable APIs, and understanding the semantic context of code, we propose SAVANT, a novel system that leverages the power of LLMs to enhance the detection and impact assessment of vulnerable API usages. As shown in Figure 1, we design a two-phase process to effectively identify and evaluate the impact of vulnerable API usages within an application. Phase I, Semantic-Preserving Code Preprocessing, addresses the challenges of large codebases and variable naming conventions by transforming the application's source code into a structured, semantically rich representation. This involves dividing the code into blocks and generating vector embeddings that capture the underlying meaning of each block, regardless of specific identifiers used. These embeddings, along with the original code and metadata, are stored in a database for efficient retrieval. Phase II, Reflection-Based Detection with LLM, tackles the crucial aspect of semantic understanding by leveraging the capabilities of LLMs. Given a vulnerable API and the corresponding PoV test function, SAVANT uses its vector representation to identify potentially relevant code blocks from the database. It then applies a novel reflection-based querying mechanism, allowing the LLM to iteratively gather contextual information and refine its understanding of how the API is used within the application. This iterative process enables the LLM to determine whether the identified API usage actually leads to a vulnerability in the application. The combination of these two phases allows SAVANT to perform a comprehensive analysis that goes beyond simple pattern matching, providing a deeper understanding of the code's behavior and the true impact of potential vulnerabilities.

The following Sections 4.2 and 4.3 detail our technical approach and demonstrate how SAVANT addresses these challenges through its two-phase architecture.

4.2 Phase I: Semantic-preserving Code Preprocessing

In this phase, SAVANT transforms the App's source code into a format suitable for effective vulnerability analysis. This process involves two key steps: (1) AST-enhanced code segmentation to break down the source code into semantically meaningful blocks, and (2) semantic-preserving code embedding to convert these blocks into dense vector representations. These steps enable SAVANT to capture and preserve the semantic structure of the code while preparing it for efficient LLM-based analysis.

4.2.1 AST-enhanced Code Segmentation

Code segmentation is crucial for processing large codebases with LLMs due to their limited context window size (typically not more than 200,000 tokens) and the performance degradation observed with longer sequences [23]. Therefore, effective code segmentation is essential for maintaining model performance and ensuring comprehensive code analysis. However, fixed-size chunking and line-based splitting can disrupt syntactic units and structural dependencies. By adopting Abstract Syntax Tree (AST)-based segmentation, we preserve the code's hierarchical structure and semantic relationships, enabling analysis of semantically cohesive blocks rather than arbitrary text segments.

Specifically, SAVANT converts the App's source code into an AST represented as $G = \{g_1, g_2, \ldots, g_n\}$, where each g_i corresponds to a compilation unit in the AST. A naive approach might treat each AST node as an independent block. However, this would lead to an excessive number of small, fragmented blocks, making it difficult to understand the relationships between them. For instance, a single AST node representing a complete if-statement block provides more context than numerous smaller nodes representing individual components of that same if-statement. This fragmentation hinders the ability to capture the overall logic and semantic meaning of the code. Conversely, when nodes are too large (containing multiple methods or classes), they may include substantial irrelevant code that obscures the key vulnerability-related information, potentially causing LLM hallucination. Additionally, oversized nodes limit the number of code segments that can fit within the LLM's context window, reducing our ability to analyze multiple related code sections simultaneously. To identify semantically meaningful code blocks, we use a heuristic based on node size and inner structure within each compilation unit. SAVANT segments the AST from the compilation unit using the following rule:

$$g_i \rightarrow \begin{cases} b_i, & \text{if } f_{size}(g_i) < \theta, \\ \{b_{i1}, b_{i2}, \ldots, b_{ik}\}, & \text{otherwise.} \end{cases}$$

Here, θ is a predefined maximum size threshold. For larger nodes ($f_{size}(g_i) \geq \theta$), SAVANT performs structural segmentation on the compilation unit node, further dividing it into sub-nodes, specifically focusing on *ImportDeclaration*, *FieldDeclaration*, *MethodDeclaration*, and *ConstructorDeclaration*. Intuitively, this approach maintains semantic relationships while creating blocks suitable for LLM-based analysis. The specific thresholds used in this process are further discussed in Section 5.4.

4.2.2 Semantic-preserving Code Embedding:

Following the AST-enhanced code segmentation, SAVANT transforms the resulting code blocks into a format suitable for efficient similarity analysis. Given the set of code blocks $B = \{b_1, b_2, \ldots, b_m\}$ obtained from the previous step, SAVANT embeds each block into a dense vector that captures its semantic meaning and functionality. Formally, we define this embedding process as a function f:

$$f : B \rightarrow V, \text{where } V = \{v_1, v_2, ..., v_m\} \text{ and } v_i \in \mathbb{R}^t \quad (1)$$

Here, t is the dimensionality of the embedding space. This transformation ensures that semantically similar code blocks are positioned closer together in the t-dimensional vector space. To generate these embeddings, we apply pre-trained encoders, such as OpenAI V3 embedding models [50], that process entire code blocks as input:

$$v_i = \text{Encoder}(b_i), \text{ where } ||v_i|| = t \text{ for all } i \quad (2)$$

This approach ensures fixed-length embedding vectors regardless of the original block size. We chose this fixed-length block-level embedding over token-level embedding for two primary reasons: 1) *Comprehensive representation:* block-level embedding captures the overall functionality of a code block. For example, consider the following code snippets:

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

Although these snippets have different tokens, they perform semantically similar functionality (i.e., summing elements). Block-level embedding would likely position these blocks closer together in the vector space compared to token-level embedding, which might focus more on the syntactic differences. 2) *Standardization for consistent comparisons:* Fixed-length embeddings provide a uniform format for representing code snippets in a vector space. This standardization enables direct and consistent comparisons between any pair of code

embeddings, regardless of the original code's length or structure. By maintaining a constant dimensionality across all embeddings, this approach avoids additional preprocessing or size adjustments when computing similarities, thus preserving the integrity of the encoded semantic information. Finally, SAVANT stores all processed information in a database D, represented as: $D = \{d_1, d_2, \ldots, d_n\}$, where each entry d_i is a tuple containing $di = (b_i, v_i)$, b_i represents the original code block along with its associated metadata (filename, line number range, the original source code snippet, and the AST node type), and v_i is its corresponding vector embedding.

4.3 Phase II: Reflection-Based Detection

This phase focuses on accurately identifying and analyzing vulnerable API usages through a reflection-based approach. The process consists of three primary steps: (1) Candidate Identification, which finds code blocks related to the vulnerable API and the PoV test function; (2) Context-Complete Code Retrieval, which iteratively expands the context around each candidate for a comprehensive understanding of its usage; and (3) Semantic-Guided Reachability Analysis, where the LLM, acting as a vulnerability expert, decides if the vulnerability can actually be exploited. This approach addresses a key challenge in vulnerability analysis: the inherent difficulty for static analysis tools to accurately understand code semantics, particularly in complex and dynamic applications. Static analysis primarily relies on pattern matching and struggles to interpret the nuanced meaning and relationships within code. In contrast, our reflection-based method leverages the LLM's ability to reason about code semantics and context, enabling a more accurate assessment of whether a vulnerable API usage is truly exploitable. This allows SAVANT to identify vulnerabilities that static analysis might miss, while also reducing false positives.

4.3.1 Candidate identification

Identifying precise API invocation locations is crucial as they serve as starting points for subsequent semantic analysis of vulnerable API usage patterns. SAVANT begins by leveraging a dual-seed approach: (1) vulnerable APIs and (2) PoV test cases. Vulnerable APIs provide a direct target for identifying potentially unsafe usage, while security test cases offer insights into how these APIs are used in practice, especially in security-sensitive contexts.

Using these seeds, SAVANT employs a two-stage semantic-guided code matching process to accurately pinpoint where the API is actually called within the application code. First, it embeds the code blocks from the project under analysis, the vulnerable API signatures, and the code from security test cases into a shared vector space. Then, for each code block embedding v_i, SAVANT calculates its cosine similarity to each vulnerable API embedding v_{api} and each security test case embedding v_{test}. We select cosine similarity over other metrics, such as Euclidean distance, as it produces equivalent results given that embedding vectors are normalized to a unit length, while the dot product operation in cosine similarity is more efficient [50].

The similarity between a code block embedding v_i and the vulnerable API embedding v_{api} is defined as Equation 3:

$$S_c(v_{api}, v_i) = \frac{v_{api} \cdot v_i}{||v_{api}||||v_i||}, \tag{3}$$

The similarity between a code block embedding v_i and a security test case embedding v_{test} is defined as Equation 4:

$$S_c(v_{test}, v_i) = \frac{v_{test} \cdot v_i}{||v_{test}||||v_i||}, \tag{4}$$

To ensure we identify actual API invocation points rather than semantically similar but non-calling code blocks, we implement two filtering stages:

- (F1) **Embedding Similarity Filter** $S_c(v_{api}, v_i) > \tau$ or $S_c(v_{test}, v_i) > \tau$, where τ is the minimum similarity threshold. This filter considers a code block relevant if it shows strong semantic similarity to either the API's signature or a security test case. This dual-seed approach broadens the search, increasing the likelihood of capturing relevant code blocks that might be missed by relying solely on API signatures.

- (F2) **LLM-based Source Code Verification** For code blocks that pass (F1), we apply a second filter (F2) that leverages an LLM grader to verify actual API invocation at the source code level: $LLM_{grade}(b_i, b_{api}) = \text{"yes"}$. The LLM grader analyzes the code block b_i and API signature b_{api} at the source code level to verify actual API invocation, handling cases where semantic similarity alone may not accurately reflect API usage.

This two-stage approach combines the efficiency of embedding-based similarity matching with the precision of LLM-based source code verification, ensuring accurate identification of API invocation locations for subsequent vulnerability analysis. By default, SAVANT considers a code block c_i as a candidate for further analysis if:

$$\begin{aligned} C_{candidate} = \{c_i = (v_i, b_i, v_{api}) \mid c_i \in D \\ \wedge (S_c(v_{api}, v_i) > \tau \vee S_c(v_{test}, v_i) > \tau) \\ \wedge LLM_{grade}(b_i, b_{api}) = \text{"yes"}\} \end{aligned} \tag{5}$$

Figure 2 illustrates the candidate identification process for projects using BCryptPasswordEncoder, where the POV demonstrates a potential null pointer exception in the `encode()` method. Starting from the POV, SAVANT retrieves code snippets from its vector database that are semantically related to both the `BCryptPasswordEncoder.encode()` method and the

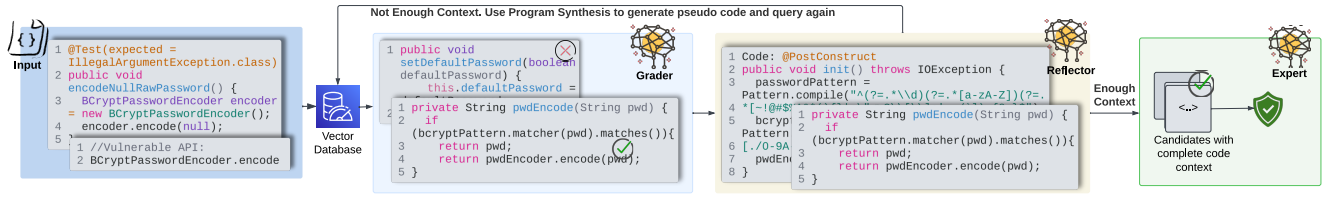

Figure 2: An example of context-complete code retrieval for BCryptPasswordEncoder.encode() method.

vulnerability pattern (F1). During the grading and filtering process (F2), SAVANT identifies that the second snippet contains the core password encoding implementation with semantic checking (marked with a green checkmark), while filtering out unrelated method definitions (marked with red 'X's). When the initially retrieved context is insufficient, SAVANT uses code inference to generate additional queries, ensuring complete context for vulnerability analysis. This procedure effectively combines semantic retrieval with targeted code inference to build a comprehensive code context.

4.3.2 Context-complete Code Retrieval

Accurate vulnerability assessment requires analyzing the complete context of code implementation and API usage patterns. Incomplete context can lead to misinterpretations and inaccurate assessments. To address this, SAVANT employs an LLM-powered iterative reflection technique to achieve comprehensive contextual coverage for each candidate. This technique is inspired by self-reflection methods [14], which iteratively refine generated responses by learning from prior outputs. In SAVANT, we adapt this approach to the specific needs of context-complete vulnerability analysis.

Given a security test case function f_{test}, a vulnerable API API_{vul}, and an initial candidate set $C_{candidate} = \{c_1, c_2, \ldots, c_n\}$, SAVANT iteratively expands the context through the following steps:

- **Reflection Query:** For each candidate $c_i \in C_{candidate}$, SAVANT querys $Q(c_i, API_{vul}, f_{test})$ the LLM: $Q(c_i, API_{vul}, f_{test}) \rightarrow (\{Yes|No\}, \text{reason})$. "Yes" indicates sufficient context for vulnerability assessment, while "No" is accompanied by a reason explaining the missing context. This feedback guides subsequent steps.

- **Code Inference:** If $Q(c_i, API_{vul}, f_{test}) \rightarrow (No, \text{reason})$, SAVANT generates a code inference query $P(c_i, API_{vul}, f_{test}, \text{reason})$ to the LLM: $P(c_i, API_{vul}, f_{test}, \text{reason}) \rightarrow (c_{missing}, scope)$. Here, $c_{missing}$ represents the inferred code snippet to search for, while $scope$ defines required metadata constraints, like class, method, and filepath, ensuring retrieved code meets both functional similarity and structural specifications.

- **Iterative Context Expansion via Code Retrieval:** While the initial candidate identification in Section 4.3.1 uses only code similarity, here SAVANT conducts retrieval using both $c_{missing}$ for embedding similarity search and $scope$ to filter results based on structural constraints. This enhanced retrieval process yields a new set of candidates $C_{missing}$ that are both syntactically similar and structurally relevant. This set is then merged with the existing candidates: $C'_{candidate} = C_{candidate} \cup C_{missing}$.

These three steps work synergistically to iteratively build a more complete code context. The LLM's feedback in the Reflection Query drives the Code Inference step, which in turn informs the Context Expansion. This iterative process continues until one of the termination conditions is met:

- The LLM assesses vulnerability by $Q(c_i, API_{vul}, f_{test})$ and confirms the current context adequacy.

- The candidate identification process yields $C_{missing} = \emptyset$, confirming no new relevant blocks.

Algorithm 1 provides a detailed outline of this context-complete code retrieval process. The algorithm initializes with a set of candidates and iteratively refines the context for each candidate using the Q (Reflection Query) and P (Code Inference) functions, along with the Search function (from Section 4.3.1). The loop continues until the context is deemed complete or no new candidates are found.

Figure 2 illustrates SAVANT's iterative context-building process. Starting with an initial code snippet showing a potential null pointer exception in `BCryptPasswordEncoder.encode()`, SAVANT first identifies a method containing the password encoding implementation. Through reflection analysis, SAVANT determines that this context alone is insufficient to verify the null-check mechanisms for the `pwd` parameter. When an incomplete context is detected, SAVANT generates targeted queries to retrieve additional code segments, focusing on how `pwd` is constructed and validated. This iterative retrieval process continues until sufficient context is gathered to perform a comprehensive vulnerability assessment, as indicated by the feedback loop in the figure. The analysis proceeds to the next stage after obtaining complete context.

Algorithm 1 Context-Complete Code Reflection-Based Retrieval

Require: $C_{candidates} = \{c_1, c_2, \ldots, c_n\}$, API_{vul}, f_{test}
Auxiliary Functions:
`Search(code, scope)` $\rightarrow C$: Search for similar code and filter by scope constraints. Return a code set C.
`Q(`C`,` API_{vul}`,` f_{test}`)` $\rightarrow$ (boolean, reason): Determines if the code set C is semantically complete for API_{vul} and f_{test}.
`P(reason)` $\rightarrow (c_{missing}, scope)$: Code Inference function that generates the code snippet $c_{missing}$ and structural constraints $scope$ based on the provided reason.

for each $c_i \in C_{candidates}$ **do**
  $C' \leftarrow \{c_i\}$
  $(isComplete, reason) \leftarrow Q(C', API_{vul}, f_{test})$
  while not $isComplete$ **do**
    $(c_{missing}, scope) \leftarrow P(reason)$
    $C'_{candidate} \leftarrow$ `Search`$(c_{missing}, scope)$
    if $C'_{candidate} == \emptyset$ **then**
      break
    end if
    $C' \leftarrow C' \cup C'_{candidate}$
    $C_{candidates} \leftarrow C_{candidates} \cup C'_{candidate}$
    $(isComplete, reason) \leftarrow Q(C', API_{vul}, f_{test})$
  end while
end for

4.3.3 Sematic-Guided Reachability Analysis

After obtaining the context-complete candidates, SAVANT performs a semantic-guided reachability analysis to determine if the identified vulnerabilities are exploitable within the target application. This phase leverages an LLM's ability to understand code semantics and reason about potential exploitability, going beyond traditional static analysis techniques that rely on pattern matching.

For each context-complete candidate c_i in the set $C = \{c_1, c_2 ..., c_n\}$ SAVANT constructs a query for Q_i for each candidate c_i: $Q_i = (c_i, API, f_{test}, R, T)$. Here, c_i represents the candidate code snippet, API is the vulnerable API, and f_{test} is the security test case. SAVANT instructs the LLM to act as a "Java vulnerability analysis expert" (R) and tasks it with determining if the App is affected by the vulnerability (T). This query structure provides the LLM with the full context necessary for accurate assessment, including the vulnerable API usage, the security test case, and the contextually relevant code snippets identified during the retrieval phase.

For instance, in the secure code example discussed in Section 3, the LLM integrates information from Listing 1 (PoV test) and Figure 2 (the context-complete code snippets) into the query Q_i to perform the semantic-guided reachability analysis. It evaluates whether the usage of `pwd` and `encoder.encode(pwd)` could potentially allow setting the null value, as demonstrated in the test function. The LLM provides insights such as: *"The encode method of BCryptPasswordEncoder throws an IllegalArgumentException when a null value is passed as the raw password."*. This detailed semantic analysis provides crucial information for both security experts and developers when conducting vulnerability assessments.

Building on this analysis, SAVANT processes each query Q_i independently, returning a binary decision: 1 (vulnerable) or 0 (secure). This independent evaluation allows for potential parallelization, enabling efficient analysis of large-scale projects. The final vulnerability assessment VD for App is determined as follows:

$$VD(App) = \begin{cases} 1 \text{ (vulnerable)}, & \text{if } \exists i : LLM(Q_i) = 1 \\ 0 \text{ (secure)}, & \text{if } \forall i : LLM(Q_i) = 0 \end{cases} \quad (6)$$

The system assigns a vulnerable label when any query Q_i indicates potential exploitation while requiring all queries to indicate safety for a secure classification.

While this LLM-powered approach offers significant advantages in understanding code semantics, it is important to acknowledge potential limitations. The accuracy of the assessment depends on the LLM's ability to comprehend complex code logic and subtle security nuances, and there is a possibility of false positives or negatives.

5 Evaluation

This section presents a comprehensive evaluation of SAVANT's ability to identify vulnerabilities arising from library dependencies. We begin by describing the datasets used in our experiments and the metrics used to quantify the effectiveness of vulnerability detection (Section 5.1). Subsequently, we analyze SAVANT's performance, including comparative results against State-Of-The-Art (SOTA) tools (Sections 5.3–5.4). Finally, we present an ablation study investigating the impact of different model configurations on SAVANT's performance. Our evaluation aims to answer the following Research Questions (RQs):

- **RQ1 (Effectiveness):** *How effectively does* SAVANT *identify vulnerabilities in real-world projects caused by library dependencies?*
- **RQ2 (Tool Comparison):** *How well does* SAVANT *compare with SOTA solutions?*
- **RQ3 (Ablation Study):** *How do different LLMs and embedding models affect* SAVANT*'s ability to detect vulnerable API usage?*

5.1 Benchmark Dataset And Metrics

To evaluate the effectiveness of our tool, we utilized a third-party benchmark [34] comprising 25 distinct vulnerabilities across 55 open-source applications. We list the vulnerable libraries and its corresponding CVEs in Table 1. Within this

dataset, 42 applications were identified as vulnerable due to dependencies on vulnerable libraries, while the remaining 13 were confirmed as unaffected. To enhance the accuracy of the ground truth labels, we conducted a manual code audit of all 55 applications. This process involved tracing from the known vulnerable APIs to identify all potentially vulnerable execution paths. The project list and label are listed in Table 2.

Following prior work [56], we leverage four metrics to evaluate SAVANT:

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.

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

Accuracy (A): The ratio of correctly classified projects (both vulnerable and non-vulnerable) to the total number of projects in the ground truth dataset.

$$A = \frac{\text{\# of correctly classified projects}}{\text{Total \# of projects in ground truth}}$$

F-score (F) is the harmonic mean of precision and recall; it reflects the trade-off between precision and recall.

$$F = \frac{2 \times P \times R}{P + R}$$

5.2 RQ1 (Effectiveness)

To assess the effectiveness of SAVANT in identifying vulnerabilities stemming from library dependencies in real-world projects (RQ1), we conducted extensive empirical evaluation using a comprehensive benchmark dataset of 55 open-source Java projects [34]. Table 2 presents the vulnerability analysis results for each project. SAVANT demonstrates strong detection capability by successfully identifying vulnerabilities in 31 out of 42 vulnerable projects, achieving a recall of 73.8%. With the default configuration, SAVANT flagged 6 additional projects as potentially vulnerable, resulting in a precision of 83.8%, indicating its conservative approach to security assessment.

A representative example of SAVANT's effectiveness is its analysis of the HTTPCLIENT-1803 [26] vulnerability in project gobblin [10]. SAVANT precisely detected a security flaw where malformed URL paths without a leading forward slash could override specified host settings in URIBuilder. Specifically, when analyzing code patterns like `new URIBuilder("@notexample.com/mypath").setHost("example.com")`, SAVANT identified that the host parameter fails to be properly enforced, potentially enabling unauthorized host redirection. Through project-wide analysis, SAVANT confirmed that gobblin's dependency on this vulnerable API could lead to security risks in its network access controls.

SAVANT also demonstrates sophisticated security context analysis, as shown in the Apache Kylin project evaluation. When analyzing BCryptPasswordEncoder usage, SAVANT identified potential null pointer exceptions in the encode method. However, through comprehensive data flow analysis, SAVANT verified that the project implements null-check mechanisms before password encoding operations, ensuring that null values cannot reach the encoder. This case demonstrates SAVANT's ability to consider project-specific implementation safeguards when assessing potential vulnerabilities.

Further analysis reveals SAVANT's adaptive handling of code complexity at the node level. For example, in the findbugs project [20], individual classes like `BugTreeModel` and methods like `branchOperations` contain extensive code blocks within single nodes. SAVANT automatically manages such dense code concentrations through its hierarchical decomposition mechanism, though the current tree-sitter parsing implementation may produce fragmented statements when processing these exceptionally large single nodes. This observation provides valuable insights for future optimizations of SAVANT's AST-enhanced code segmentation.

> **Finding 1 (response to RQ1:)** *With the labeled ground truth,* SAVANT *identifies the client project that has been impacted by vulnerable API with 83.8% precision, 73.8% recall, and 78.5% F1.* SAVANT *has the ability to understand the code semantics with the given sufficient context.*

5.3 RQ2 (Tool Comparison)

To assess the performance of SAVANT relative to existing solutions, we compare it against two representative tools from the domain of traditional SCA:

- **VAScanner** [66] utilizes call graph analysis to trace the invocation of vulnerable APIs from library dependencies to the application code. By analyzing the call graph, VAScanner determines whether vulnerable APIs are actually reachable and invoked within the application.

- **Eclipse Steady** [22] combines static and dynamic analysis techniques to perform reachability analysis. It traces the execution paths from a known vulnerable API (the root cause) to its potential invocation points within the application, providing a more detailed assessment of vulnerability exploitation.

To ensure a fair comparison, we run experiments for all tools on the same machine. The OS is Ubuntu 24.04.1 LTS with Intel Xeon Silver 4214R and 64GB memory. We implement

CVE ID	Library	CWE	CVE ID	Library	CWE
CODEC-134	Apache Commons Codec	CWE-20 (Improper Input Validation)	CVE-2020-13973	json-schema-validator	CWE-20 (Improper Input Validation)
CVE-2017-7525	jackson-databind	CWE-502 (Deserialization of Untrusted Data)	CVE-2020-26217	xstream	CWE-78 (OS Command Injection)
CVE-2017-7957	xstream	CWE-20 (Improper Input Validation)	CVE-2020-28052	bcprov-jdk14	CWE-1025 (Comparison Using Wrong Factors)
CVE-2018-1000632	Dom4j	CWE-91 (XML Injection)	CVE-2020-28491	jackson-dataformat-cbor	CWE-770 (Resource Allocation Without Limits or Throttling)
CVE-2018-1000873	jackson-modules-java8	CWE-20 (Improper Input Validation)	CVE-2020-5408	spring-security	CWE-330 (Use of Insufficiently Random Values)
CVE-2018-1002200	plexus-archiver	CWE-22 (Path Traversal)	CVE-2021-23899	json-sanitizer	CWE-611 (Improper Restriction of XML External Entity Reference)
CVE-2018-1002201	plexus-utils	CWE-22 (Path Traversal)	CVE-2021-27568	spring-security-oauth	CWE-287 (Improper Authentication)
CVE-2018-11761	tika-parsers	CWE-611 (Improper Restriction of XML External Entity Reference)	CVE-2021-29425	apache-commons-io	CWE-22 (Path Traversal)
CVE-2018-12418	junrar	CWE-835 (Loop with Unreachable Exit Condition)	CVE-2021-30468	Apache CXF	CWE-835 (Loop with Unreachable Exit Condition)
CVE-2018-1274	spring-data-commons	CWE-770 (Resource Allocation Without Limits or Throttling)	CVE-2022-25845	fastjson	CWE-502 (Deserialization of Untrusted Data)
CVE-2018-19859	openrefine	CWE-22 (Path Traversal)	CVE-2022-45688	hutool-json	CWE-787 (Out-of-bounds Write)
CVE-2019-10093	tika-parsers	CWE-770 (Resource Allocation Without Limits or Throttling)	CVE-2023-34454	jackson-databind	CWE-502 (Deserialization of Untrusted Data)
CVE-2019-12402	commons-compress	CWE-835 (Loop with Unreachable Exit Condition)	HTTPCLIENT-1803	httpcomponents-client	CWE-180 (Incorrect Behavior Order: Validate Before Canonicalize)
CVE-2020-13956	httpcomponents-client	CWE-20 (Improper Input Validation)	TwelveMonkeys-595	twelvemonkeys-imageio	CWE-20 (Improper Input Validation)
			Zip4J-263	zip4j	CWE-20 (Improper Input Validation)

Table 1: Library Vulnerabilities Overview

VAScanner based on their published code [66] and fix program issues based on the paper's description, adhering to their default settings. We run Eclipse Steady version 3.2.5. We set a 150-minute analysis time limit per project for both Eclipse Steady and VAScanner.

Table 3 presents a comparative analysis of SAVANT, Eclipse Steady, and VAScanner across key performance metrics. SAVANT achieved the highest precision, recall, accuracy, and F1-score, indicating a greater ability to identify vulnerabilities caused by library dependencies within the evaluated Java projects. The lower performance of Eclipse Steady and VAScanner is further illustrated in Table 2. SAVANT successfully analyzed all 55 projects, while Eclipse Steady and VAScanner failed to analyze 17 and 35 projects, respectively.

For VAScanner, its applicability is limited to Maven projects with standard classpath configurations, and it frequently encounters out-of-memory errors when analyzing large codebases. Examination of the source code reveals a reliance on hard-coded paths for classpath resolution within a Maven-specific design. These constraints, combined with insufficient memory management for large-scale analysis, explain its failure to process a significant portion of the dataset.

Analysis of Eclipse Steady's results indicates that even when successfully analyzing projects, Eclipse Steady failed to identify vulnerabilities in cases where SAVANT succeeded, such as in the *netarchivesuite* and *PLMCodeTemplate* projects. This observation suggests potential limitations in Eclipse Steady's static analysis engine, particularly in accurately constructing call graphs and performing comprehensive data flow analysis, which may limit its ability to detect all vulnerable API invocations.

Despite its improved performance compared to the baseline tools, SAVANT still exhibits false positives and false negatives, highlighting the inherent challenges in automated vulnerability detection. These findings emphasize the need for continued research and development to enhance the precision and recall of automated vulnerability detection techniques.

Finding 3 (response to RQ2:) SAVANT *outperforms state-of-the-art solutions in the real-world projects benchmark as SOTAs has faced challenges with the memory limitation and the complex dependencies when building the call graph.*

5.4 RQ3 (Ablation Study)

To evaluate the performance of various LLMs within the SAVANT framework, we selected five state-of-the-art closed-source models and one open-source model: GPT-4O [51], GPT-O1 [51], CLAUDE-3.5-SONNET [9], GOOGLE-GEMINI-2.0-FLASH [29] and LLAMA-3.1-405B [7]. We utilized *voyage-code-3* [8] and *OpenAI text-embedding-3-small* [49] as the embedding models. Furthermore, we explored the im-

Project	Version	Ground Truth	SAVANT	Eclipse Steady	VAScanner
hadoop	0dbe1d32	✗	✓	✗	✗
pay-java-parent	f4f5b8f8	✗	✗	✗	-
druid	078d5ac5	✗	✗	✗	-
pmq	86e2d931	✗	✗	-	-
ole	9f7e33c6	✗	✗	✓	-
findbugs	fd7ec8b5	✗	✓	✓	✗
netarchivesuite	01b069f8	✗	✗	✓	✓
tcpser4j	7a3dbd8d	✗	✗	-	-
roubsite	34a2d22d	✗	✗	✗	✗
core-ng-demo-project	f5e39ffb	✓	✓	-	-
light-4j	68233ba2	✗	✗	✗	✗
rtds-test	07e7e175	✓	✗	✓	✗
elasticsearch-maven-plugin	51706d75	✗	✗	✗	-
tomcat-maven-plugin	32f8302e	✗	✗	-	✗
tycho	a7cdf96c	✗	✗	✗	✗
arraybase	11ac4730	✗	✓	-	-
jlogstash	abe9fb44	✗	✗	✓	-
spring-data-commons	23776034	✗	✗	✗	-
spring-data-mongodb	c2fc09e3	✗	✗	✓	-
OpenRefine Authenticator	6ba959d3	✗	✗	✓	-
bysj	db19a910	✓	✗	-	-
library-of-alexandria	c22277a1	✗	✓	-	-
graphicsfuzz	897a7d49	✓	✓	✗	✗
james-project	056af8c6	✗	✓	✓	✗
alluxio	a16bc958	✓	✓	✗	-
Java-9-Cookbook	d3109e55	✓	✓	-	-
AxonFramework	58fd4d2d	✗	✗	✗	✗
curso-fundamentos-java	4a638622	✗	✗	-	-
knetbuilder	24cc998e	✗	✗	-	-
PLMCodeTemplate	85b7d744	✗	✗	✓	✗
powertac-core	1167f29a	✗	✓	✗	-
tiny	e9180d10	✗	✗	-	-
communote-server	e6a35410	✗	✓	-	-
Openfire	3cd2f68a	✓	✓	✗	✓
CodeDefenders	bdd9ff93	✗	✓	✓	-
kylin	443c2523	✓	✓	-	✗
nacos	a19d3fd0	✗	✓	✓	-
SpringBoot-Learning	fb41583a	✓	✓	-	-
anet	2657f909	✗	✗	-	-
ets-wfs20	3fdf6e41	✗	✗	✗	-
pmd	411be4ac	✗	✗	✓	-
filedossier	2d393042	✗	✗	✓	✓
commerce	899f81c2	✗	✗	✗	-
wxzm	d4aeba96	✗	✗	✗	-
corese	27ad57ca	✓	✗	✗	✓
metersphere	729d7954	✓	✗	✗	-
swim-jumpstart	9a7402cb	✗	✗	✓	-
aem-caching	8a5d4dd9	✗	✗	✓	✓
badlion-src	93a099e7	✗	✗	✓	✗
cantci	16bef3c6	✓	✗	✓	-
ia-recruiter	95567676	✓	✗	✗	✗
storm	ae259205	✗	✗	✓	✗
backend	912aa83a	✗	✓	✗	-
collect	d292f59c	✗	✓	-	-
gobblin	44a7e1a2	✗	✗	-	-

Table 2: Experiment Results per project. ✓: project is secure; ✗: project is insecure; -: the tool failed to run.

Tool	Precision	Recall	Accuracy	F1
SAVANT	0.838	0.738	0.691	0.785
Eclipse Steady	0.700	0.241	0.242	0.359
VAScanner	0.73	0.262	0.271	0.386

Table 3: Performance Metrics of Different Tools

pact of varying the maximum code segment size (θ) by evaluating performance with values of 500, 1000, 1500, 2000, 2500, and 3000. Figure 3 presents the comparative results across four key metrics: precision, recall, accuracy, and F1 score.

Impact of LLMs: Among the tested models, GPT-4O and GOOGLE-GEMINI-2.0-FLASH consistently demonstrate superior performance across all metrics. Particularly, GPT-4O achieves highest F1 scores of 0.87 with OpenAI embeddings at 2,500 tokens, while GOOGLE-GEMINI-2.0-FLASH maintains stable performance with F1 scores above 0.85 across various code lengths. LLAMA-3.1-405B demonstrates competitive precision scores around 0.85 but shows lower recall rates around 0.60, resulting in F1 scores between 0.64 and 0.74. GPT-O1 shows relatively lower performance, with F1 scores ranging from 0.72 to 0.83.

Embedding Model Comparison: The choice of embedding model shows notable impact on performance. OpenAI embeddings generally yield marginally better results compared to Voyage embeddings, particularly evident in precision metrics. This advantage is most pronounced when paired with GPT-4O and GOOGLE-GEMINI-2.0-FLASH, where OpenAI embeddings contribute to approximately 2-3% higher precision scores. However, the performance gap between embedding models narrows at larger code segment sizes, suggesting that the choice of embedding model becomes less critical with increased context. For LLAMA-3.1-405B, both embedding models show similar precision performance, though Voyage embeddings demonstrate slightly better recall rates at smaller code segment sizes.

Code Length Analysis: The maximum size of the code segment (θ) significantly influences the model performance. We observe that:

- Performance generally improves as θ increases from 500 to 2,500 tokens, with optimal results typically achieved around 2,000-2,500 tokens.
- Beyond 2,500 tokens, performance plateaus or slightly decreases, indicating a potential sweet spot for context window size.
- Smaller code segments (500-1,000 tokens) show more variance in performance across different model-embedding combinations, suggesting that larger segments provide more stable results.

Trade-offs: Our results reveal important trade-offs between model performance and computational efficiency. While larger code segments generally improve accuracy, they also increase processing time and resource requirements. The optimal configuration appears to be using GPT-4O or GOOGLE-GEMINI-2.0-FLASH with OpenAI embeddings at a code segment size of 2,500 tokens, balancing performance with computational efficiency. While LLAMA-3.1-405B shows promising precision metrics, its lower recall rates suggest it may be better suited for scenarios where precision is prioritized over comprehensive vulnerability detection.

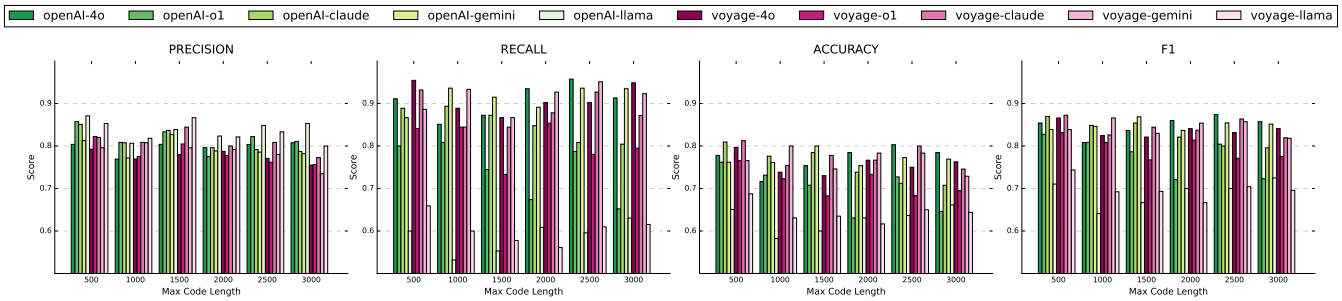

Figure 3: Performance comparison of different LLMs and embedding models across varying maximum code lengths. The legend follows the format "embedding model-LLM model" (e.g., "openAI-4o" indicates OpenAI embedding model with GPT-4O, and "voyage-claude" represents Voyage embedding model with CLAUDE-3.5-SONNET). Each subplot shows a different evaluation metric (precision, recall, accuracy, and F1 score). Different colors represent different LLM models, with green bars indicating OpenAI embeddings and purple bars indicating Voyage embeddings.

Finding 4 (response to RQ3:) *Different LLMs and embedding models show varying performance in* SAVANT. *Models achieve F1 scores ranging from 0.64 to 0.87, with* GPT-4O *and* GOOGLE-GEMINI-2.0-FLASH *performing consistently better (F1: 0.82-0.87) than* LLAMA-3.1-405B *(F1: 0.64-0.74) and* GPT-O1 *(F1: 0.72-0.83). The highest scores are observed at code segment sizes between 2,000-2,500 tokens, and OpenAI embeddings generally show a 2-3% higher precision compared to Voyage embeddings for most models.*

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. Future work will include comprehensive sensitivity analyses. 2) Ground Truth Accuracy: The manual labeling data requires significant effort. Our evaluation's accuracy relies on the correctness of ground truth data. To ensure labeling accuracy, three authors who have industry experience in security domains labeled the data. However, it is important to note that this approach might impact the overall accuracy of the classification. 3) Prompt Template Limitation: We used only fixed prompt templates for the LLM. This could limit the range of vulnerabilities detected and affect the accuracy of our results. 4) Implementation Choices: SAVANT's effectiveness may be influenced by specific implementation choices (e.g., parser, encoder models).

External Validity. The generalizability of our findings may be limited by the following factors: 1) Our observations and conclusions are based on the third-party datasets [34]. While these provide valuable insights, they may not include all possible scenarios or vulnerabilities present in real-world applications. 2) We evaluated our approach only on Java programs, motivated by both Java's widespread adoption in OSS and the increasing security concerns surrounding Java supply chain attacks in widely used libraries. Our proposed approach is theoretically language-agnostic. However, this Java-focused evaluation limits the generalizability of our findings, as the effectiveness of our approach may vary across languages with different syntax structures, programming paradigms, or vulnerability patterns. 3) Our current investigation relies on exemplar security tests from libraries to demonstrate vulnerability context. This approach, while effective for our study, may not capture all of the vulnerability manifestations in diverse software ecosystems.

7 Related Work

Vulnerable API Detection Researchers and engineers created tools to detect the invocation of vulnerable APIs or the insecure use of APIs [19, 24, 25, 35, 54, 55, 57, 60, 61, 72, 76]. These tools are essential for identifying code that may introduce security risks through flawed API usage patterns, particularly within Java cryptographic libraries. Specifically, tools such as FindSecBugs [24], SonarQube [61], Xanitizer [57], CogniCrypt [35], CryptoGuard [55], CryptoTutor [60], and SEADER [76] utilize static analysis techniques to verify whether Java cryptographic APIs are invoked with secure parameter configurations and correct sequential orders. Fischer et al. [25] and Xu et al. [72] 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.

Our approach distinguishes itself from prior work in two key aspects: it generalizes vulnerability detection across diverse API types beyond specific cases like cryptographic APIs, and incorporates broader program context to accurately assess security implications.

Deep Learning based Software Vulnerability Detec-

tion Deep learning-based software vulnerability detector approaches [16, 27, 37, 38, 40, 41, 63, 79] have been proposed to automatically learn vulnerability patterns from various code representations. IVDetect [38] utilized RNN-based models to generate code representations from source code and identify vulnerable code patterns. LineVul [27] 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 [63] 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 dependency vulnerabilities. Differ from these approaches, SAVANT detects the vulnerable API usage from library dependency. Furthermore, most of existing deep learning approaches primarily target C/C++ vulnerabilities [16, 27, 37, 38, 40, 41, 79], SAVANT specifically targets vulnerable API usage detection in Java projects.

LLM for Vulnerability Detection Recent work has explored LLMs' capabilities in security tasks [15, 21, 31, 42, 46, 53, 64, 78]. SecLLMHolmes [64] evaluates the ability of various LLMs to detect security-related bugs under different configurations, concluding that current models is non-deterministic and not robust. Liu *et al.* [42] and Deng *et al.* [21] investigate LLMs' potential in vulnerability management and penetration testing processes, respectively. Several studies have also investigated vulnerable code repair [17, 32, 43, 53, 68, 70] through LLMs. For instance, Pearce et al. [53] examine LLMs in repairing vulnerable code in a zero-shot setting, finding that while models can successfully repair hand-crafted examples, they struggle with complex, real-world cases.

These studies employ either a single-step approach, querying the LLM once for a decision, or require human intervention. In contrast, SAVANT adopts an iterative, reflection-based method where LLMs automatically refine contextual information through multiple distinct roles. While most works focus on local code context [32, 43, 53, 64, 68, 70], and RLCE [17] considers repository-level context, our approach uniquely uses an LLM reflection-based method for efficient context retrieval and precise vulnerability detection in complex programs.

8 Conclusion

We created SAVANT —a novel framework to assess vulnerable API impacts through semantic preprocessing and reflection-based detection, integrating LLMs for context-aware vulnerability analysis. Compared with existing SCA tools, SAVANT offers more precise vulnerability assessment by understanding semantic context in large projects and provides developers with accurate and actionable API impact analysis. Our evaluation demonstrates SAVANT's effectiveness in identifying actual vulnerable API usage with high precision, recall, and accuracy across various codebases and scenarios. In the future, we will enhance SAVANT's analysis capabilities by implementing heuristic-based code block splitting to identify contextual vulnerability patterns more effectively. We will also extend support to C and Python codebases, enabling comprehensive vulnerability assessment across diverse software supply chains.

References

[1] CVE-2020-5408 Detail. https://nvd.nist.gov/vuln/detail/CVE-2020-5408, May 2020. Accessed: 2025-01-19.

[2] OWASP Dependency-Check. https://owasp.org/www-project-dependency-check/, 2020.

[3] Supply chain attacks show why you should be wary of third-party providers. https://www.csoonline.com/article/3191947/supply-chain-attacks-show-why-you-should-be-wary-of-t.html, 2021.

[4] npm-audit. https://docs.npmjs.com/cli/v9/commands/npm-audit, 2023.

[5] NVD. https://nvd.nist.gov, 2023.

[6] Test - Snyk User Docs. https://docs.snyk.io/snyk-cli/commands/test, 2023.

[7] Meta AI. Meta llama 3.1 announcement. https://ai.meta.com/blog/meta-llama-3-1/, 2024. Accessed: 2024-10-30.

[8] Voyage AI. Voyage code 3. https://blog.voyageai.com/2024/12/04/voyage-code-3/. Published: 2024-12-04, Accessed: 2025-01-20.

[9] Anthropic. Claude sonnet. https://www.anthropic.com/claude/sonnet, 2024. Accessed: 2024-10-30.

[10] Apache Software Foundation. MySqlJdbcUrl.java. https://github.com/apache/gobblin/blob/44a7e1a27cc73387cf309487f45895801984059d/gobblin-metastore/src/main/java/org/apache/gobblin/metastore/util/MySqlJdbcUrl.java#L51, 2023. Accessed: 2024-01-22.

[11] Apache Software Foundation. Apache Log4j™ 2. https://logging.apache.org/log4j/2.x/, 2024. Accessed: 2024-01-22.

[12] Apache Software Foundation. KylinUserService.java - Apache Kylin. https://github.com/apache/kylin/blob/443c2523e27e86ed397c526f741db62a805b95c4/server-base/src/main/java/org/apache/kylin/

rest/service/KylinUserService.java, 2025. Accessed: 2025-01-19.

[13] Apache Software Foundation. UserController.java - Apache Kylin. https://github.com/apache/kylin/blob/443c2523e27e86ed397c526f741db62a805b95c4/server-base/src/main/java/org/apache/kylin/rest/controller/UserController.java, 2025. Accessed: 2025-01-19.

[14] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-rag: Self-reflective retrieval augmented generation. *arXiv preprint arXiv:2310.11511*, 2023.

[15] Heewon Baek, Minwook Lee, and Hyoungshick Kim. Cryptollm: Harnessing the power of llms to detect cryptographic api misuse. In *European Symposium on Research in Computer Security*, pages 353–373. Springer, 2024.

[16] 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*, ICSE '24, 2024.

[17] Yuxiao Chen, Jingzheng Wu, Xiang Ling, Changjiang Li, Zhiqing Rui, Tianyue Luo, and Yanjun Wu. When Large Language Models Confront Repository-Level Automatic Program Repair: How Well They Done? . In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 459–471, April 2024.

[18] Xiao Cheng, Xu Nie, Ningke Li, Haoyu Wang, Zheng Zheng, and Yulei Sui. How about bug-triggering paths?-understanding and characterizing learning-based vulnerability detectors. *IEEE Transactions on Dependable and Secure Computing*, 21(2):542–558, 2022.

[19] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Code-based vulnerability detection in node.js applications: How far are we? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pages 1199–1203, New York, NY, USA, 2021. Association for Computing Machinery.

[20] FindBugs Project Contributors. Bugtreemodel.java. https://github.com/tonydamage/findbugs/blob/fd7ec8b5cc0b1b143589674cdcdb901fa5dc0dda/findbugs/src/gui/edu/umd/cs/findbugs/gui2/BugTreeModel.java, 2023. Accessed: 2024-10-30.

[21] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. {PentestGPT}: Evaluating and harnessing large language models for automated penetration testing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 847–864, 2024.

[22] Eclipse Foundation. Eclipse Steady. https://projects.eclipse.org/projects/technology.steady, 2024. Accessed: 2024-01-22.

[23] Xiaoning Feng, Xiaohong Han, Simin Chen, and Wei Yang. Llmeffichecker: Understanding and testing efficiency degradation of large language models. *ACM Transactions on Software Engineering and Methodology*, 2024.

[24] Find Security Bugs Project. Find Security Bugs. https://find-sec-bugs.github.io/, 2021. Accessed: 2024-01-22.

[25] 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.

[26] Apache Software Foundation. Jira issue: Httpclient-1803. https://issues.apache.org/jira/browse/HTTPCLIENT-1803, 2017. Accessed: 2024-10-24.

[27] 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.

[28] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Detecting suspicious package updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 13–16. IEEE, 2019.

[29] Google. Gemini ai. https://gemini.google.com/. Accessed: 2025-01-20.

[30] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. Linevd: statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories*, pages 596–607, 2022.

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

[32] 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.

[33] Md Mahir Asef Kabir, Ying Wang, Danfeng Yao, and Na Meng. How do developers follow security-relevant best practices when using npm packages? In *2022 IEEE Secure Development Conference (SecDev)*, pages 77–83, Los Alamitos, CA, USA, oct 2022. IEEE Computer Society.

[34] 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.

[35] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. Cognicrypt: supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936. IEEE, 2017.

[36] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. Towards the detection of malicious java packages. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 63–72, 2022.

[37] Runhao Li, Chao Feng, Xing Zhang, and Chaojing Tang. A lightweight assisted vulnerability discovery method using deep neural networks. *IEEE Access*, 7:80079–80092, 2019.

[38] 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, page 292–303, 2021.

[39] 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(4):2244–2258, 2021.

[40] 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.

[41] 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.

[42] Peiyu Liu, Junming Liu, Lirong Fu, Kangjie Lu, Yifan Xia, Xuhong Zhang, Wenzhi Chen, Haiqin Weng, Shouling Ji, and Wenhai Wang. Exploring {ChatGPT's} capabilities on vulnerability management. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 811–828, 2024.

[43] Peng Liu, He Wang, Chen Zheng, and Yuqing Zhang. Prompt Fix: Vulnerability Automatic Repair Technology Based on Prompt Engineering . In *2024 International Conference on Computing, Networking and Communications (ICNC)*, pages 116–120, February 2024.

[44] Jeremy Long and contributors. Dependency-check issue #4629: False positive detection of cve-2020-5408. https://github.com/jeremylong/DependencyCheck/issues/4629, September 2023. Accessed: 2025-01-19.

[45] Jeremy Long and contributors. Dependency-check issue #6685: False positive detection of cve-2023-4759. https://github.com/jeremylong/DependencyCheck/issues/6685, January 2025. Accessed: 2025-01-19.

[46] 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.

[47] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. VulChecker: Graph-based vulnerability localization in source code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6557–6574, Anaheim, CA, August 2023. USENIX Association.

[48] Marc Ohm, Arnold Sykosch, and Michael Meier. Towards detection of software supply chain attacks by forensic artifacts. In *Proceedings of the 15th international conference on availability, reliability and security*, pages 1–6, 2020.

[49] OpenAI. Embeddings guide. https://platform.openai.com/docs/guides/embeddings. Accessed: 2025-01-20.

[50] OpenAI. Embedding models: Openai documentation. https://platform.openai.com/docs/guides/embeddings/embedding-models, 2024. Accessed: 2024-10-30.

[51] OpenAI. Model overview. https://platform.openai.com/docs/models, 2024. Accessed: 2024-10-30.

[52] OWASP Foundation. OWASP Dependency-Check. https://owasp.org/www-project-dependency-check/, 2024. Accessed: 2024-01-22.

[53] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining Zero-Shot Vulnerability Repair with Large Language Models . In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356, May 2023.

[54] 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.

[55] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng 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*, pages 2455–2472, 2019.

[56] 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, page 2455–2472, 2019.

[57] RIGS IT. Xanitizer. https://www.rigs-it.com/xanitizer/, 2021. Accessed: 2024-01-22.

[58] Miaomiao Shao and Yuxin Ding. FVD-DPM: Fine-grained vulnerability detection via conditional diffusion probabilistic models. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7375–7392, Philadelphia, PA, August 2024. USENIX Association.

[59] Samiha Shimmi, Ashiqur Rahman, Mohan Gadde, Hamed Okhravi, and Mona Rahimi. VulSim: Leveraging similarity of Multi-Dimensional neighbor embeddings for vulnerability detection. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1777–1794, Philadelphia, PA, August 2024. USENIX Association.

[60] 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.

[61] SonarSource. SonarQube. https://www.sonarqube.org/, 2021. Accessed: 2024-01-22.

[62] Sonatype. State of the software supply chain report 2024: 10-year review. https://www.sonatype.com/state-of-the-software-supply-chain/2024/10-year-look, 2024. Accessed: 2024-01-13.

[63] 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.

[64] 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.

[65] 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 *IEEE Symposium on Security and Privacy*, 2024.

[66] VAScanner Project Contributors. VAScanner. https://github.com/VAScanner/VAScanner, 2024. Accessed: 2024-01-22.

[67] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 172–184, 2023.

[68] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1282–1294, 2023.

[69] 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.

[70] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated Program Repair in the Era of Large Pre-trained Language Models . In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494, May 2023.

[71] Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. Tracer: Finding patches for open source software vulnerabilities. *arXiv preprint arXiv:2112.02240*, 2021.

[72] Zhiwu Xu, Xiongya Hu, Yida Tao, and Shengchao Qin. Analyzing cryptographic api usages for android applications using hmm and n-gram. In *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 153–160. IEEE, 2020.

[73] 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.

[74] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. Mitigating persistence of open-source vulnerabilities in maven ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 191–203. IEEE, 2023.

[75] Ying Zhang, Md Mahir Asef Kabir, Ya Xiao, Danfeng Yao, and Na Meng. Automatic detection of java cryptographic api misuses: Are we there yet? *IEEE Transactions on Software Engineering*, 49(1):288–303, 2022.

[76] 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.

[77] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. Software composition analysis for vulnerability detection: An empirical study on java projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 960–972, 2023.

[78] 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.

[79] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. muVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection . *IEEE Transactions on Dependable and Secure Computing*, 18(05):2224–2236, September 2021.