Context-Enhanced Vulnerability Detection Based on Large Language Model

YIXIN YANG, State Key Laboratory of Complex & Critical Software Environment (CCSE), Beihang University, China
BOWEN XU, State Key Laboratory of Complex & Critical Software Environment (CCSE), Beihang University, China
XIANG GAO*, State Key Laboratory of Complex & Critical Software Environment (CCSE), Beihang University, China and Hangzhou Innovation Institute of Beihang University, China
HAILONG SUN, State Key Laboratory of Complex & Critical Software Environment (CCSE), Beihang University, China and Hangzhou Innovation Institute of Beihang University, China

Vulnerability detection is a critical aspect of software security. Accurate detection is essential to prevent potential security breaches and protect software systems from malicious attacks. Recently, vulnerability detection methods leveraging deep learning and large language models (LLMs) have garnered increasing attention. However, existing approaches often focus on analyzing individual files or functions, which limits their ability to gather sufficient contextual information. Analyzing entire repositories to gather context introduces significant noise and computational overhead. To address these challenges, we propose a context-enhanced vulnerability detection approach that combines program analysis with LLMs. Specifically, we use program analysis to extract contextual information at various levels of abstraction, thereby filtering out irrelevant noise. The abstracted context along with source code are provided to LLM for vulnerability detection. We investigate how different levels of contextual granularity improve LLM-based vulnerability detection performance. Our goal is to strike a balance between providing sufficient detail to accurately capture vulnerabilities and minimizing unnecessary complexity that could hinder model performance. Based on an extensive study using GPT-4, DeepSeek, and CodeLLaMA with various prompting strategies, our key findings includes: (1) incorporating abstracted context significantly enhances vulnerability detection effectiveness; (2) different models benefit from distinct levels of abstraction depending on their code understanding capabilities; and (3) capturing program behavior through program analysis for general LLM-based code analysis tasks can be a direction that requires further attention.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Vulnerability Detection, Program Analysis, Large Language Model

*These authors are co-corresponding authors.
Authors' Contact Information: Yixin Yang, State Key Laboratory of Complex & Critical Software Environment (CCSE), Beihang University, Beijing 100191, China, yixinyang@buaa.edu.cn; Bowen Xu, State Key Laboratory of Complex & Critical Software Environment (CCSE), Beihang University, Beijing 100191, China, cheneytsu@buaa.edu.cn; Xiang Gao*, State Key Laboratory of Complex & Critical Software Environment (CCSE), Beihang University, Beijing 100191, China and Hangzhou Innovation Institute of Beihang University, Hangzhou 310056, China, xiang_gao@buaa.edu.cn; Hailong Sun, State Key Laboratory of Complex & Critical Software Environment (CCSE), Beihang University, Beijing 100191, China and Hangzhou Innovation Institute of Beihang University, Hangzhou 310056, China, sunhl@buaa.edu.cn.

ACM 1557-735X/2025/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Yixin Yang, Bowen Xu, Xiang Gao*, and Hailong Sun. 2025. Context-Enhanced Vulnerability Detection Based on Large Language Model. *J. ACM* 37, 4, Article 111 (August 2025), 48 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Memory safety vulnerabilities constitute a persistent and critical class of software security flaws in computing systems. Despite extensive research and mitigation strategies, these vulnerabilities remain predominant in recent Common Vulnerabilities and Exposures (CVE) announcements [10, 54]. The fundamental issue stems from memory-unsafe languages such as C and C++, which prioritize performance and low-level memory control over safety guarantees. These languages permit direct memory manipulation during buffer operations, allocation, and pointer management, creating significant security risks. Although modern programming languages incorporate memory safety features, legacy codebases and performance-sensitive applications continue to rely on memory-unsafe languages, perpetuating their vulnerability to exploitation.

Static vulnerability detection is an effective technique that identifies potential security vulnerabilities by analyzing code without executing the program. Current approaches to static vulnerability detection can be broadly categorized into static program analysis, machine learning, and deep learning-based methods. Traditional static program analysis methods often rely on expert-defined rules, which, while effective, are time consuming, labor intensive, prone to generating a high false alarm rate and difficult to generalize across different codebases [14, 47, 62]. Machine learning-based methods for vulnerability detection utilize manually defined quantitative code features (i.e., cyclomatic complexity, number of nested loops, the maximum count of control/data structures, etc.) combined with traditional machine learning algorithms (i.e., SVM, KNN, etc.). They offer better generalization capabilities [17]. However, due to their shallow architectures and the coarse-grained features they extract, they often fail to achieve high detection accuracy. Deep learning-based vulnerability detection methods have recently surpassed traditional approaches, leveraging rich semantic features extracted from source code (i.e., Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), Program Dependency Graphs (PDGs), etc.) and deep learning algorithms [5, 34, 71, 73]. These approaches significantly improve both the effectiveness and interpretability of vulnerability detection by capturing richer structural and semantic information.

Despite the progress made by learning-based methods, several limitations still remain. First, function-level and file-level vulnerability detection methods often rely on using individual pre- and post-patch functions as the input to the model [5, 34, 71, 73]. However, these methods lack sufficient context to capture the root cause of vulnerabilities. In many cases, the root cause of a vulnerability might not be accurately reflected by the modified function alone, as it often involves cross-function interactions. Additionally, these methods tend to focus heavily on understanding token-level information, which makes it challenging for the models to generalize to new vulnerabilities not seen during training, resulting in a high number of false positives and false negatives. Second, vulnerability detection methods at the repository level aim to provide a more comprehensive context by analyzing entire codebases [59, 69]. While such methods can theoretically capture cross-function or cross-file interactions, these methods suffer from substantial noise. For instance, repository-level analysis involves a lot of irrelevant information that can obscure meaningful patterns, making it difficult for models to accurately detect vulnerabilities. Furthermore, analyzing large code repositories, especially those with millions of lines of code, presents significant challenges due to compilation issues, static analysis overhead, and the "path explosion" problem in control flow analysis, which results in increased computational costs and limited scalability.

Recently, large language models (LLMs), such as GPT-4 [1], LLaMA [56] and DeepSeek [12], have shown remarkable code comprehension capabilities, drawing increasing attention from

researchers in the field of software engineering. Trained on massive code repositories with billions of parameters, these models have the potential to capture deeper semantic information and address the shortcomings of traditional shallow learning models. Several work has explored applying LLMs for vulnerability detection, utilizing prompt engineering [50, 63, 64, 70] and model fine-tuning [18, 25, 35, 46, 57] , and other strategies [19, 31] . However, similar to earlier deep learning models, LLMs struggle to directly infer the root causes of vulnerabilities with inadequate contextual information. Lacking specific contextual information inspired our investigation. While combining contextual information with Large Language Models is an emerging paradigm, we observe that existing work often utilizes context in a coarse-grained manner, lacking theoretical guidance. This leaves a critical scientific question unanswered: What form and granularity of context are most effective and low-noise for LLMs in complex vulnerability detection tasks? Our work addresses this gap by moving beyond the question of whether to provide context, to investigate how context should be systematically designed and adapted for LLMs.

To address this challenge, we first explain the theoretical advantages of our chosen methodology. We propose Primitive API Abstraction, an LLM-centric context representation method designed to bridge the gap between complex program structures and the semantic understanding capabilities of LLMs. This approach is superior because it transforms intricate program dependencies, such as control and data flow graphs, into a semi-structured textual format that is more congenial to LLMs. This transformation effectively distills the inter-procedural semantic essence related to memory and resource operations , while naturally filtering out substantial implementation noise irrelevant to these vulnerability patterns. We utilize program analysis to abstract function calls based on primitive APIs to provide this refined contextual information to LLMs. Specifically, we conduct control flow analysis and data flow analysis on the to-be-patched functions and their callee functions within a specific iteration layer, then we abstract function calls based on primitive APIs to provide contextual information to LLMs. Primitive APIs, e.g., *malloc*, are basic operations that commonly occur in various scenarios. The motivation for considering primitive APIs is rooted in their fundamental nature — memory safety vulnerabilities often arise in the improper usage of these APIs. By focusing on primitive APIs, we can effectively abstract away the noise introduced by complex or domain-specific function calls, providing clearer contextual information for the LLMs to understand potential vulnerabilities. Second, we conduct an empirical study to evaluate how different levels of primitive API abstraction and different prompt engineering strategies impact the detection capabilities of LLMs. The motivation for using different API abstraction level is to determine the optimal level of contextual granularity that improves vulnerability detection while avoiding excessive noise. By evaluating various abstraction levels, we aim to identify the balance between providing sufficient detail to capture vulnerabilities accurately and reducing unnecessary complexity that may hinder model performance. The motivation for evaluating different prompt engineering strategies is to explore how effectively tailored prompts can enhance the reasoning capabilities of LLMs for vulnerability detection. Specifically, different prompt strategies, such as Chain-of-Thought [58], Few-Shot Learning [4], and In-Context Learning [52], provide different ways to guide the model's focus and inference process. By systematically evaluating these strategies, we aim to determine the most suitable prompt techniques for different abstraction levels, maximizing the models' ability to understand complex contexts and accurately detect vulnerabilities. Our empirical study reveals the following major findings:

(1) **Primitive API Abstraction is Effective:** Using primitive API abstractions significantly enhances vulnerability detection performance across different LLMs, showing improvements in accuracy and reducing false positives compared to models without API abstractions.

(2) **Different Vulnerabilities Types Require Varying API Abstraction Levels:** Resource management-related vulnerabilities necessitate high-level abstraction, while both high- and low-level abstractions demonstrate good performance for boundary-related memory errors.
(3) **Different Models Require Different Levels of Abstraction:** For larger models like GPT-4o and DeepSeek-r1, higher-level abstractions are more suitable, while models like CodeLLaMA benefit from more detailed abstractions, as they are specifically optimized for code understanding and generation.
(4) **Different Prompt Engineering Suits for Different API Abstraction Levels:** Higher API abstraction levels, which enhance contextual information richness, favor more advanced reasoning-oriented prompt strategies.

Based on our findings, we implement a tool named **PacVD**, that is, **P**rimitive **A**PI Abstraction and **C**ontext-Enhanced **V**ulnerability **D**etection method utilizing Large Language Models. Experimental results show that our method significantly outperforms the baselines. Using the Chain-of-Thought prompt strategy with DeepSeek-R1, our approach achieves improvements of up to 12.77% in accuracy, 10.05% in precision, 9.25% in F1 score compared to baseline methods. Similar improvements are observed with ChatGPT-4o and CodeLLaMA, demonstrating the effectiveness of our API abstraction approach across different model architectures. These results suggest that combining appropriate API abstractions with well-designed prompt strategies can substantially enhance the vulnerability detection capabilities of large language models.

Our contributions can be summarized as follows:

- **A Novel, LLM-Centric Context Representation Method**: We propose and validate "Primitive API Abstraction," a multi-level framework specifically designed for LLMs. This method efficiently distills the semantic essence of inter-procedural vulnerabilities by focusing on primitive API usage patterns, effectively balancing information fidelity and noise reduction.
- **The First Systematic Study on the Impact of Context Granularity**: We conduct the first systematic investigation into the non-linear relationship between context abstraction granularity and the performance of different LLM architectures. Our findings reveal the existence of an "information saturation point" and demonstrate that different models have distinct preferences for context granularity, providing a theoretical basis for future research.
- **The Design and Evaluation of PacVD**: We implement our findings in a tool named PacVD. Extensive experiments show that by selecting the appropriate abstraction level, PacVD significantly outperforms baseline methods in detecting complex memory and resource-related vulnerabilities across multiple LLMs.

2 Background and Motivation

2.1 Inter-procedural Vulnerabilities

Vulnerabilities often span multiple files and functions, with the vulnerability trigger location sometimes lying outside the patched function. According to research by Li *et al.* [32], vulnerabilities frequently involve an average of 2.8 layers (or functions). As illustrated in Listing 1, it is CVE-2015-8962, the vulnerability type is Double Free, and the to-be-patched function in vulnerability patch is “sg_common_write”. This issue arises because in function named “sg_common_write”, a callee function named “blk_end_request_all” releases the block request “srp->rq” in a control branch at line 10, yet the “sg_finish_rem_req” function attempts to free the request object “srp->rq->cmd” in certain branches at line 11, which may result in freeing a memory block twice, as after “blk_end_request_all” is called, “srp->rq” points to an invalid memory area. However, in “sg_finish_rem_req”, “srp->rq->cmd” is released again, and this pointer has been released when “srp->rq” is released. If a vulnerability detection model only receives to-be-patched function

```
static int sg_common_write(Sg_fd * sfp, Sg_request * srp,
                unsigned char *cmnd, int timeout, int blocking){
    ...
    k = sg_start_req(srp, cmnd);
    if (k) {
    ...
    }
    if (atomic_read(&sdp->detaching)) {
        if (srp->bio)
            blk_end_request_all(srp->rq, -EIO);
        sg_finish_rem_req(srp);
        return -ENODEV;}
    ...
static int sg_finish_rem_req(Sg_request *srp) {
    ...
    if (srp->rq) {
        if (srp->rq->cmd != srp->rq->__cmd)
            free(srp->rq->cmd);
        blk_put_request(srp->rq); }

    ... }
void blk_end_request_all(struct request *rq, blk_status_t error) {
    ...
    if (unlikely(blk_bidi_rq(rq)))
            bidi_bytes = blk_rq_bytes(rq->next_rq);
    blk_finish_request(rq, error);
    ... }
void blk_finish_request(struct request *req, int error){
    ...
    __blk_free_request(req->q, req);
}
static inline void __blk_free_request(struct request_list *rl, struct request *rq)
{
    ...
    mempool_free(rq, rl->rq_pool);
}
void mempool_free(void *element, mempool_t *pool)
{
    ...
    pool->free(element, pool->pool_data);
}
```

Listing 1. A Double Free Vulnerability Example (CVE-2015-8962)

"sg_common_write" without the concrete contents in other functions, it will be unable to find the root cause of the vulnerability and may fail to identify the vulnerability or report the wrong problem type. We test this vulnerability on ChatGPT-4o. The model responds: "While these issues do not necessarily constitute a confirmed vulnerability directly, they are indeed potential security risks. To fully assess its severity, you also need to understand: The context in which this function is called and its inputs can be controlled by an attacker and whether the code is running in a kernel or sensitive environment with high privileges."

However, providing the complete code for these functions will introduce excessive noise, hindering the model's ability to identify relevant information about the vulnerability. This claim is empirically supported by our experiments in Section 6.4. As detailed in Table 6, our abstraction-based method (PacVD) consistently outperforms the All Callees (AL-C) baseline, which uses the complete raw code of callee functions, across all tested models. Similarly, PacVD outperforms other baseline methods including the complete code of callee functions selected through various sampling strategies. This performance gap demonstrates that un-abstracted baseline code, despite its completeness, can negatively impact the model's detection effectiveness. Therefore, a more in-depth analysis of callee functions of to-be-patched functions in vulnerability patches is essential to discover the root cause of the vulnerability. Furthermore, after analyzing this vulnerability, we

find that resource-related vulnerabilities, whether cross-function or not, are often closely linked to certain key resource management APIs. In this example, it is evident that the vulnerability is associated with the fundamental memory management API, "malloc(), free()". Therefore, we attempt to identify APIs involved in critical resource operations to assist in resource-related vulnerability detection, and then further expand to the detection of other types of vulnerabilities.

2.2 Limitations of Existing Vulnerability Detection Methods

With the rise of LLMs, there have been numerous attempts to leverage them for vulnerability detection. Unfortunately, we observe that existing approaches [50, 64, 65] all failed to detect this vulnerability. By further investigating the underlying reason, Zhang *et al.* [65] found that LLMs generally learn shallow information about vulnerabilities, such as token meanings, and tend to misjudge when certain tokens are modified, such as function names. Besides, Zhang *et al.* [64] found that large language models can easily detect syntax-related or boundary-related vulnerabilities, but they fail to achieve high accuracy for vulnerabilities that require a complete understanding of the context. Therefore, they tested the impact of data flow and API call information as supplementary information for vulnerability code on model detection performance and found that adding certain contextual information is effective for large models. Similarly, Steenhoek *et al.* [50] conducted empirical research on LLMs for vulnerability detection, they found that the performance was close to random guessing, with accuracy rates between 50% and 60%. Thus, it is clear that the effectiveness of vulnerability detection using LLMs is significantly constrained by the context of the vulnerability. Furthermore, these studies did not investigate the effectiveness of LLMs for complex inter-procedural vulnerabilities. If only to-be-patched functions and their internal information are used as samples for vulnerability detection, it becomes even more difficult to identify the root cause, and accurately detect vulnerabilities.

2.3 Our approach

We aim to use abstraction based on primitive API to represent each callee function within each to-be-patched function, simplifying each callee and highlighting the primitive API-related operations it performs. For instance, in the vulnerability shown in Listing 1, we attempt to abstract the callee functions using control flow and data flow analysis. We extract all control branches within each callee function to identify branches containing specific primitive APIs and use the conditions under which these APIs appear as concrete branch abstraction. We then determine whether a particular primitive API is invoked in all, some, or none of the branches, forming a fuzzy control branch abstraction for that API. Additionally, we count the occurrences of the primitive API within the callee function and analyze the data objects operated on by the API. Finally, we use the fuzzy conditions, concrete branch conditions, occurrence counts, and key data objects operated on by the primitive API as an abstraction for each callee function, which we refer to as the Primitive API abstraction.

3 Vulnerability Detection Framework

We propose a context-enhanced vulnerability detection framework named **PacVD** that addresses the limitations of existing approaches in detecting complex inter-procedural vulnerabilities. The input to PacVD includes a target function and its contextual information related to callee functions within a specific scope. In practical scenarios, the target function may be a user-specified suspicious function requiring detection, or it could originate from code changes committed to version control systems or periodic scans of entire codebases. The output is whether the target function contains vulnerabilities. As illustrated in Figure 1, our framework comprises three key components: (1) Data Preprocessing, responsible for extracting and preparing code for analysis;

(2) Primitive API Abstraction, which extracts contextual information relevant to vulnerabilities; and (3) LLM-based Vulnerability Detection, which leverages contextual information for precise vulnerability identification.

Traditional vulnerability detection methods typically adopt a single perspective: function-level approaches lack sufficient contextual information, while repository-level methods introduce excessive noise and computational overhead. Our approach extracts key API behaviors through program analysis, maintaining low computational costs while preserving sufficient contextual information. This framework is applicable not only to historical vulnerability analysis but can also be integrated into the software development process to meet vulnerability detection needs at different stages.

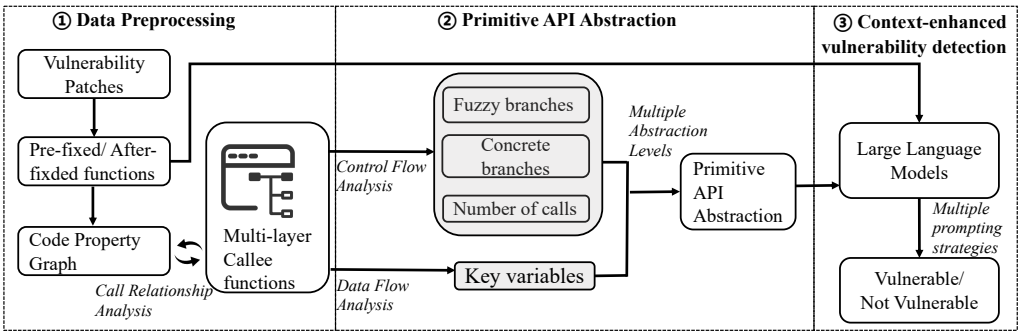


Fig. 1. Primitive API Abstraction and Context-Enhanced Vulnerability Detection Framework

3.1 Data Preprocessing

The first phase of the framework involves determining the code analysis scope and conducting data preprocessing. A study by Li et al. [32] indicates that inter-procedural vulnerabilities are prevalent with an average of 2.8 inter-procedural layers. According to their statistics, nearly 75% of the vulnerability types have an average call depth of no more than three layers. To empirically determine the optimal analysis scope for our approach, we conduct a preliminary study to evaluate the cost-benefit trade-off of varying the call depth. Our results, detailed in Section 4.2, confirm that a three-layer call depth provides the best balance between detection performance and computational overhead. Thus, we established a default analysis depth of three function call layers, that is, we analyze the target function and all its callee functions within a three-layer call depth as our scope. After establishing the analysis scope, the system constructs Code Property Graphs (CPGs) of relevant functions through static program analysis. These CPGs serve as a unified code representation, capturing multiple static properties including control flow, call graphs, and data flow. For each target function requiring analysis, the system constructs CPGs not only for the function itself but also callee functions. This multi-level analysis is crucial for capturing cross-function vulnerabilities. The output of these steps is a collection of target functions with their calling relationships, along with corresponding code property graph representations, which provide the foundation for the next phase that follows.

3.2 Primitive API Abstraction

To provide LLMs with a low-noise, high-signal context for detecting complex inter-procedural vulnerabilities, we introduce our core methodology: Primitive API Abstraction. This technique is designed to generate a concise, semantic summary for each callee function invoked within a target function-under-test.

The generation of each summary is a deep, recursive process. For each direct callee of the target function, our methodology traverses its subsequent call chain down to a depth of three layers, as statistical evidence suggests most inter-procedural vulnerabilities manifest within this scope. Within this defined three-layer analysis window, we perform targeted control and data flow analysis. This analysis models the callee's aggregate behavior exclusively through the lens of a predefined set of security-critical "primitive APIs". The resulting abstraction for each callee therefore encapsulates not just its own logic, but also the nested security-relevant operations of its own callees, capturing how, where, and how often these key APIs are invoked within its deeper operational context.

For the final vulnerability detection task, these generated summaries are provided to the LLM as supplementary, high-signal context, appended to the full source code of the target function. This approach creates a comprehensive input prompt where the LLM can analyze the target function's logic while simultaneously referencing the abstract summaries to understand the deeper semantic behavior of its callees. The result is an enriched analytical context that highlights critical inter-procedural interactions without the burden of parsing irrelevant implementation details from the callees' full source code.

3.2.1 Rationale for Primitive API Abstraction. The selection of Primitive API Abstraction as the singular context model for PacVD is not an arbitrary choice, but a deliberate, LLM-centric design philosophy rooted in three foundational principles: LLM-congeniality, high semantic fidelity with intrinsic noise reduction, and a direct causal link to our target vulnerability classes.

- **LLM-Congeniality: Translating Structure into Semantics.** At their core, Large Language Models are text-based architectures that excel at processing and reasoning over natural language and semi-structured text. This architectural characteristic distinguishes them from traditional neural networks used in vulnerability detection, such as Graph Neural Networks (GNNs), which are specifically designed to operate on graph-based representations like Control Flow Graphs (CFGs) or Program Dependence Graphs (PDGs). A significant challenge in applying LLMs to code analysis is the effective translation of complex program structures into a format that the model can natively comprehend without substantial information loss. Our Primitive API Abstraction method directly addresses this challenge. It acts as a sophisticated translator, converting the essential structural and semantic information of inter-procedural control and data flows into a concise, semi-structured textual format. This representation is fundamentally more congenial to an LLM's architecture. It allows the model to directly leverage its powerful semantic reasoning capabilities on a text-native input, rather than grappling with linearized graph representations that can be cumbersome and may obscure critical structural relationships.
- **High Semantic Fidelity with Intrinsic Noise Reduction.** A primary motivation of our work is to provide sufficient context for detecting inter-procedural vulnerabilities while avoiding the pitfalls of repository-level analysis, namely the introduction of "excessive noise". Providing the full source code of all callee functions can overwhelm an LLM with a deluge of implementation details—such as complex business logic or UI manipulations—that are irrelevant to the underlying security flaw. Primitive API Abstraction functions as a form of guided semantic filtering. By design, it focuses exclusively on a curated set of security-sensitive primitive APIs that are fundamental to memory and resource management (see Table 1). This mechanism preserves the high-fidelity semantic essence of these critical operations while intrinsically filtering out vast amounts of irrelevant code. For example, the abstraction captures that a memory allocation occurred via malloc and under what conditions it might be freed, but it discards the details of how that memory is used for application-specific logic. This results in a low-noise, high-signal contextual input, enabling the LLM to focus its analytical capacity on the patterns most likely to indicate a vulnerability. The effectiveness of this approach is validated in Table 5. Baselines

that append the full, unfiltered source code of callees—selected through various strategies like comprehensive, random, or similarity-based sampling—all yielded inferior performance. This result empirically confirms our central hypothesis: for LLMs, targeted abstraction provides a more potent context than indiscriminately providing more raw code.

- **Direct Causality for Target Vulnerability Classes.** Our research specifically targets the challenging domain of memory and resource management vulnerabilities, as detailed in our dataset composition. The root causes of these vulnerability classes are almost invariably linked to the improper sequence, conditions, or pairing of a small set of well-defined primitive APIs. For instance, vulnerabilities like Use-After-Free (CWE-416), Double Free (CWE-415), and Memory Leaks (CWE-401) are directly caused by the misuse of functions such as malloc and free. This decision is further supported by empirical evidence from major vulnerability databases. Our analysis of C/C++ vulnerabilities cataloged in the CWE and NVD databases indicates that approximately 70% of critical, inter-procedural memory safety and resource management flaws (e.g., those under CWE-119, CWE-415, and CWE-400) are directly attributable to the misuse of a core set of primitive C library and POSIX APIs. Given this high prevalence, a context model singularly focused on these APIs provides a highly effective and targeted strategy for detecting the majority of these complex vulnerabilities. Therefore, choosing API Abstraction is a causality-driven decision. The context generated by our method directly mirrors the potential behavioral patterns that lead to these specific flaws. This ensures that the information provided to the LLM is not merely correlated with the vulnerability but is, in fact, central to its root cause, making the context maximally relevant and potent for effective detection.

Specifically, these primitive APIs are selected from primarily standard library functions or system calls commonly used to manage resources (such as memory, file descriptors, or network connections), as shown in Table 1. The selection of primitive APIs is based on the work of Song *et al.* [48], determined through analysis of widely used C language libraries and POSIX standard APIs (such as <stdlib.h>, <stdio.h>). The motivation for selecting these APIs lies in their fundamental nature — primitive APIs are basic operations common across various scenarios, and vulnerabilities typically originate from the improper use of these APIs. For example, functions like open, fopen, fdopen, and opendir are used to open files or directories, and they require corresponding closing functions like close, fclose, and closedir to ensure proper resource closure and prevent resource leaks. Functions such as malloc, realloc, and calloc are used for memory allocation and must be paired with free to prevent memory leaks use-after-free, double free, memory leak and other memory issues. Thus, by analyzing the presence of these Primitive APIs in function to be tested and its callee functions, certain types of vulnerabilities can be effectively detected. Common primitive APIs are summarized in Table 1, which highlights the different types of vulnerabilities associated with these APIs.

Table 1. Primitive API and Corresponding Vulnerability Types

Primitive APIs	**Targeted Vulnerability Type**
open/socket/fopen/fdopen/opendir/ close/fclose/endmntent/fflush/closedir	Resource Leak
malloc/realloc/calloc/localtime	Null Pointer Dereference
malloc/free	Memory Leak, Use-After-Free, Double Free

It is crucial to clarify that the specific list of C/POSIX APIs used in this study is a high-impact instantiation of our proposed framework, which is generalizable. The core innovation lies in the

abstraction methodology itself, not in the particular set of APIs, and the framework can be adapted to other domains by curating a relevant list of security-sensitive primitives.

3.2.2 API Abstraction Methodology. For each target function and its callee functions, we perform control flow and data flow analysis to extract API usage features from four dimensions:

- **Fuzzy Branches:** Fuzzy branches refer to representing the callee functions using the fuzzy information on whether primitive APIs are found in their control flow branches. It provides a global overview of API usage features, reducing information noise while capturing critical control flow characteristics. First, we locate the target function by its unique identifier within the codebase. Once identified, we traverse the function's Control Flow Graph (CFG) to find the location of its callee functions. This traversal is performed recursively into deeper callee functions, to ensure that all relevant nested function calls are explored, providing a comprehensive understanding of resource operations, even if the operations occur in functions several layers deep in the call hierarchy. When we traverse to a callee function, we conduct a thorough examination to determine if it involves resource operations. Specifically, each operation is tracked across every control flow branch within the callee function and a defined number of its nested callees, using a depth-first traversal approach. If a resource operation is detected across all branches, we conclude that the corresponding primitive API is called in all branches, if it is detected in specific branches but not others, we conclude that the primitive API is called in only some branches, if it is not detected, the conclusion is the primitive API is not called in any branch. Finally, the fuzzy branches summary of resource usage is then integrated into the context of the target function, allowing for a holistic understanding of resource operations, including the influence of nested function calls. The fuzzy branches summary is subsequently used as part of a broader analysis to detect potential vulnerabilities, such as memory leaks, double-free errors, or use-after-free issues. For the example of Listing 1, the fuzzy branch abstraction we extracted is as follows:

  > In the "blk_end_reques_all" function: On all branches, the "free" API is called. On no branch, the "malloc" API is called. In the "sg_finish_rem_req" function: On some branches, the "free" API is called. On no branch, the "malloc" API is called.

- **Concrete Branches:** Concrete branches record the specific control conditions and precise path information of the primitive API, illustrating under what conditions the APIs are triggered. They are used to provide accurate conditional execution contexts. When traversing all control flow branches of a target function, including its callee functions and nested callees up to a specific depth, if a primitive API is called in certain branches, we add the specific control conditions of those branches to the summary of the corresponding primitive API.

  > In the "blk_end_reques_all" function, the "blk_finish_request" function is called. In the "blk_finish_request" function, the "blk_put_request" function is called. In the "blk_put_request" function, the "mempool_free" function is called. In the "mempool_free" function, if unconditionally, the "free" API is called. In the "sg_finish_rem_req" function: If (srp $\rightarrow$ rq) and (srp $\rightarrow$ rq $\rightarrow$ cmd $\neq$ srp $\rightarrow$ rq $\rightarrow$ $\_\_cmd$), the "free" API is called.

- **Number of Calls:** Number of Calls is used to quantify the frequency of each primitive API within the call chain of the target function, providing a quantitative metric for API usage. This metric is utilized to identify resource operation imbalances. When traversing all control flow branches of a target function, including its callee functions and nested callees up to a specific depth, we count the occurrences of each primitive API and add this information to the summary of the corresponding primitive API.

In the “blk_end_reques_all” function: the “malloc” API is called 0 times, the “free” API is called 1 times. In the “sg_finish_rem_req” function: the “malloc” API is called 0 times, the “free” API is called 1 times.

- **Key Variables:** Key Variables are used to identify specific variables or data objects affected by primitive API operations, tracking the relationships between APIs and variables. This approach can be leveraged to detect variable-level vulnerabilities. When traversing all control flow branches of a specified function, including its callee functions and nested callees up to a specific depth, if a primitive API is detected, we add the identifier of the variable being operated on to the summary. This is achieved through a combination of control flow and data flow analysis.

In the “blk_end_reques_all” function: the “free” API operates on the “(srp →rq)” variable. In the “sg_finish_rem_req” function: the “free” API operates on the “srp” variable.

3.2.3 Primitive API Abstraction Levels. We designed four different levels of Primitive API abstraction, as shown in Table 2. These abstraction levels are used to analyze the impact of different Primitive API information on vulnerability detection.

Table 2. Primitive API Abstraction Levels

Abstraction Level	Description
API Level 1	This is the highest level of abstraction, using only *Fuzzy Branches* summary information for different Primitive APIs.
API Level 2	*Concrete Branches* of different Primitive APIs.
API Level 3	*Concrete Branches* of different Primitive APIs combined with *Number of Calls*.
API Level 4	*Concrete Branches* of different Primitive APIs combined with *Key Variables*.
w/o API Level	No contextual information.

The multi-dimensional API abstraction method preserves critical context while avoiding excessive details, providing high-quality input for vulnerability detection by LLMs.

3.3 Model Prediction and Optimization

The final stage of the framework utilizes large language models to analyze code and its context for potential vulnerabilities. The goal is to leverage the power of LLMs to perform effective vulnerability classification, enhanced by the previous phases of context extraction.

3.3.1 Input Representation. After performing primitive API abstraction, we incorporate the information from different API levels as supplementary code structure and semantic details, directly appending them to the corresponding target functions. These are then jointly fed as prompts into the large language model.

3.3.2 Prompt Engineering. Then we utilize prompt engineering to help LLMs better understand the relationship between code and contextual information of each sample, and focus on specific resource operations that are prone to vulnerabilities. To enhance the efficacy of our evaluation experiments, we implemented several prompting strategies that have been successfully employed in large language models for various complex tasks. These strategies are documented in recent literature as effective means to harness the potential of language models in domain-specific applications [4, 44, 50, 52, 58].

- **Basic Prompt**: This strategy represents a direct zero-shot inquiry, providing the model with only the essential code and API context to establish a performance baseline without complex instructions.

> $\mathbf{P_r}$: Analyze the following code snippet and associated API information. Code Snippet:[CODE]. API Information:[API]. Is the above code vulnerable? Respond with 'yes' or 'no'.

- **Role-playing Prompt**: We configured the model to assume the role of an "expert vulnerability detection system" to provide precise, direct answers, and only elaborate explanations when necessary, enhancing the relevance and utility of the outputs for vulnerability detection tasks.

> $\mathbf{P_r}$: You are an expert vulnerability detection system. Provide precise and direct answers with explanations only when necessary. Analyze the following code snippet and associated API information. Code Snippet:[CODE]. API Information:[API]. Is the above code vulnerable? Respond with 'yes' or 'no'.

- **Chain-of-Thought Prompting**: This technique facilitates the model to articulate its reasoning process, aiding in transparent decision-making. This is particularly crucial for tasks that require detailed reasoning, such as identifying and explaining code vulnerabilities.

> $\mathbf{P_1^{(chain)}}$: [CODE], [API] Based on the above code and API information, please provide a detailed summary of the code's functionality, analyze the code structure, and locate all positions where pointers are constructed and dereferenced.
> $\mathbf{P_2^{(chain)}}$: Based on your previous analysis: [Code Analysis], determine whether the code contains significant vulnerabilities. Answer 'yes' or 'no' and provide reasons if vulnerabilities are identified.

- **In-context Learning**: This strategy adapts the prompts to reflect the dialogue's context or a specific scenario, thus providing the model with a richer informational background to execute the task more effectively.

> $\mathbf{P_1^{(context)}}$: As a code reviewer, evaluate this code snippet for clarity, functionality, and maintainability. Consider also the associated API information to ensure that the control flow aligns with the intended use and structure of the code. [CODE], [API].
> $\mathbf{P_2^{(context)}}$: Based on your initial observations and the API information, make a final assessment of whether the code meets the standards for clarity, functionality, and maintainability. Respond with 'yes' if improvements are needed, or 'no' if it meets the criteria.

- **Few-shot Learning Based on Random Selection**: This strategy involves providing the model with a few examples before posing the main task, to prime it on the task's requirements. We randomly selected examples from our dataset for this purpose.

> $\mathbf{P_f}$: Code Example 1: [CODE] API Information: [API] Output: [yes/no];
> Code Example 2: [CODE] API Information: [API] Output: [yes/no].
> Refer to the examples above, then analyze the following code snippet and associated API information[CODE], [API]. Provide a detailed response on the vulnerability status of the code. If the code is vulnerable, start your answer with "yes" and provide a brief explanation. If not, start with "no" and explain why.

- **Few-shot Learning Based on Contrastive Pairs**: We used a set of positive and negative examples (vulnerable and non-vulnerable code snippets) to further clarify the task requirements for the model. This approach helps in distinguishing subtle differences between secure and vulnerable code.

> $\mathbf{P_{f-c}}$: Examine the 'Before Fix' and 'After Fix' code snippets to understand the vulnerability remediation. Determine if the 'Before Fix' version is vulnerable, and if so, explain how the 'After Fix' version addresses the issue. Before Fix: [CODE1], After Fix: [CODE2] Refer to these examples. Now, analyze the following code snippet and API Information[CODE], [API]. Respond with 'yes' if it is vulnerable, otherwise answer 'no'.

The outcome of LLM prediction is a binary classification (i.e., vulnerable/not vulnerable), which helps to prioritize areas in code that need further manual review. Details of model configuration are shown in Section 5.4.

4 Preliminary Study: Justification for Call Depth Parameter

4.1 Motivation and Setup

To provide empirical evidence for the selection of a core parameter in our main experiments—the function call depth—we conducted a preliminary study to determine its optimal value. This study directly addresses the need to justify the trade-off between the benefit of deeper contextual information and the cost of increased noise and computational overhead.

To ensure our parameter selection is robust and validated on a large-scale, modern benchmark, we conducted this study on the PrimeVul dataset [15] (see Section 5.3.2 for a detailed description). Its scale and diversity make it an ideal testbed for determining a generalizable call depth. We varied the analysis depth from 1 to 4 layers, measuring both performance metrics and the required analysis time to isolate the impact of this parameter.

4.2 Results and Analysis

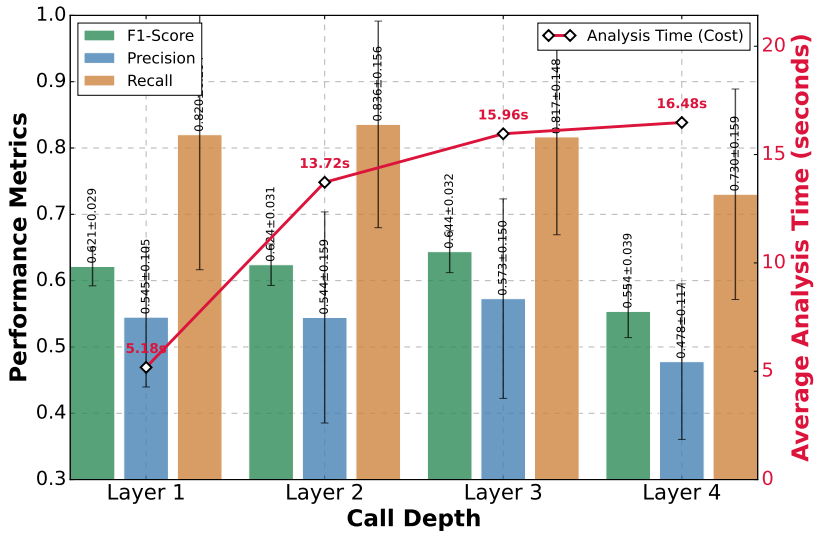


Fig. 2. Cost-benefit analysis of varying call depths. The bars represent average performance metrics (left axis), with error bars indicating the standard deviation across different model configurations. The red line plots the average analysis time per function (right axis), quantifying the computational cost.

The results of our cost-benefit analysis are presented in Figure 2. The findings unequivocally identify a three-layer call depth (Layer 3) as the optimal choice. Our analysis is based on the following key observations.

First, the results reveal a different trends between performance and cost. As shown by the bars (left axis), the average F1-Score and Precision peak at Layer 3. However, extending the analysis to Layer 4 causes a noticeable drop in both metrics, particularly in Precision (from an average of 0.573 to 0.478). This suggests that the additional context introduces more informational noise than signal, leading to a higher false-positive rate. Concurrently, the average analysis time, represented by the red line (right axis), climbs sharply with each added layer, demonstrating a significant increase in computational overhead.

Second, the analysis validates the necessity of inter-procedural context. Moving from a shallow (Layer 1) to a deeper analysis (Layer 3) yields a clear improvement in the average F1-score (from 0.621 to 0.644). This confirms that a single-layer context is insufficient to capture the complex interactions that often lead to vulnerabilities.

Finally, Layer 3 emerges as a robust and generalizable choice. The error bars on the performance metrics indicate a consistent trend across the different model configurations. While some natural variance exists, Layer 3 consistently represents the point of maximum benefit before performance declines and costs become prohibitive. Adopting it as a default setting is therefore a robust strategy that balances effectiveness and efficiency.

Based on this study, we use a three-layer call depth as the default setting for the main evaluation in the subsequent sections.

5 Empirical Evaluation Design

In this section, we first introduce the details of our research questions, followed by a description of the dataset used. Then, we explain the different strategies adopted for experimental evaluation. Finally, we present specific implementation details of the experiments.

5.1 Research Questions

The experimental evaluation aims to answer the following research questions (RQs):

- *RQ1: What is the effectiveness of different levels of Primitive API summaries on vulnerability detection?* The impact of adding different Primitive API summaries as contextual information for vulnerability detection is unknown. We aim to investigate which Primitive API information is helpful for the detection capability of LLMs.
- *RQ2: How do different API abstraction levels perform for different types of vulnerability detection?* Our goal is to explore how different API abstraction levels affect the results of vulnerability detection of different CWE types, so as to gain a deeper understanding of our approach.
- *RQ3: How do different LLMs and prompt engineering strategies perform on vulnerability detection?* We study the impact of different LLMs and prompt strategies on the detection capabilities of LLMs, and study which LLMs and strategies are helpful for primitive API summaries as context information.
- *RQ4: How does our approach compare to existing baselines?* We examine whether incorporating Primitive API information can improve the LLMs' performance beyond that of other methods.
- *RQ5: How well does PacVD generalize to diverse vulnerability types?* We evaluate the generalizability of our approach by testing its effectiveness on a broader spectrum of vulnerability types beyond memory-related errors.

5.2 Baselines

1) **Baseline for other levels of code abstraction granularity:** We apply different contextual granularities of each function as our baseline methods, referring to the method named VulEval proposed by Wen *et al.* [59]. For each to-be-tested function in our dataset, we utilize the raw code of

all callees (i.e., All Callees), raw code of callees containing Primitive APIs (i.e., API-guided Sampling Callees), raw code of callees with the highest code similarity (i.e., Similarity-based Sampling Callees), raw code of random sampled callee (i.e., Random Sampling Callees) and raw code of hierarchically sampled callees (i.e., Hierarchy Sampling Callees) as baselines, all baselines are extracted within three iteration layers.

- **All Callees:** The strategy performs a comprehensive analysis of the entire function call chain, traversing all possible paths in the call graph starting from the target function. This approach captures the complete context of function interactions by recursively analyzing each called function up to a specified depth.
- **API-guided Sampling Callees:** The sampling strategy leverages domain knowledge about Primitive APIs to make informed selections from the function call chain. It prioritizes functions containing specific API calls that are known to be security-sensitive or particularly relevant to the analysis context.
- **Similarity-based Sampling Callees:** The strategy utilizes code similarity metrics to select three callee functions that are most relevant to the target function. It implements a multi-metric similarity analysis approach, including Jaccard similarity [41] for token-based comparison, API call pattern similarity, Edit distance [6] for structural similarity, BM25 text similarity [42] for content-based comparison.
- **Random Sampling Callees:** This strategy randomly selects three functions from the target function's callee functions as the context of the target function.
- **Hierarchy Sampling Callees:** The sampling strategy implements a level-aware selection strategy considering the structural organization of the call graph. It distributes the sampling quota across different call depths, ensuring representation from each level of the call hierarchy.

2) **Traditional supervised vulnerability detectors:** We also add several traditional representative methods as baselines.

- **Devign**: A Gated Graph Neural Network (GGNN) approach learning from graph representations of code (e.g., CPGs) to identify vulnerability patterns [71]. We select Devign because it represents the SOTA for learning from rich, structured program representations, making it a direct and powerful point of comparison for any context-aware approach.
- **ReGVD**: A Graph Neural Network-based model that constructs a graph from a flat sequence of code tokens, using token embeddings from pre-trained programming language models as initial node features, for inductive text classification applied to vulnerability detection. [37]. ReGVD is chosen as it provides a highly relevant comparison, demonstrating an alternative and effective way to integrate PLM embeddings into a graph-based framework.
- **SySeVR**: Employs deep learning on syntax and semantics-based vector representations from code slices for vulnerability detection [34]. We select SySeVR because its slicing technique offers a different, highly focused method of context extraction, providing a valuable contrast to our API abstraction approach.

5.3 Dataset

5.3.1 NVD-New Dataset. In existing vulnerability detection research, datasets primarily originate from the National Vulnerability Database (NVD) [36]. For our experiments, we selected a real-world C/C++ dataset named NVD-New. This extended NVD dataset was initially constructed by Li et al. [34], and further expanded by Zhang et al. [67] using vulnerable programs collected from 16 open-source projects. We then apply a rigorous filtering process to select for high-quality cases of complex, inter-procedural memory and resource management vulnerabilities, as these are the primary focus of our PacVD method. Then we conduct API abstraction on the filtered data,

specifically, the original format of the dataset consists of project repositories containing CVEs and their corresponding CVE-specific information. We utilize the detailed information of each CVE entry such as project name, commit ID, vulnerable file name, and vulnerable function name to extract vulnerable and non-vulnerable repositories and generate CPGs for them with Joern [53]. Then we start from the function before and after the vulnerability fix, traverse their callee functions within a specific layer of the function, and then generate the primitive API abstraction according to the method in Section 3.2. Finally, the statistical results for different types of samples with primitive API summaries are shown in Table 3.

Table 3. Comprehensive Statistics of Vulnerabilities in our Dataset

Vulnerability Category	CWE IDs	# Vuln.	# Benign	# Total
Resource Management Errors	CWE-772, CWE-400, CWE-399	49	48	97
Memory Corruption	CWE-119, CWE-125, CWE-787	110	53	163
Memory Handling Errors	CWE-401, CWE-415, CWE-416	70	70	140
Pointer Related Errors	CWE-476	111	112	223
Total	**All CWEs**	340	283	623

5.3.2 PrimeVul Dataset. To ensure a rigorous comparison against the state of the art, we further evaluate our approach on PrimeVul [15], a large-scale benchmark designed to overcome the limitations of prior datasets. By integrating and de-duplicating four major datasets (i.e., BigVul [21], CrossVul [38], CVEfixes [3], and DiverseVul [8]), PrimeVul offers superior scale and diversity while significantly improving label accuracy through precise labeling techniques. Furthermore, it employs a chronological split based on commit dates to minimize data leakage, establishing a realistic evaluation scenario that mirrors real-world vulnerability discovery.

For our experiments, we utilize the official PrimeVul test set. Given the computational demands of extracting fine-grained semantic contexts via program analysis, combined with the resources required for extensive Large Language Model inference across multiple configurations, we employ stratified sampling to construct a computationally feasible yet representative test subset from the official test partition. This sampling strategy ensures that the subset maintains the diversity of vulnerability types present in the original dataset. To ensure a fair and rigorous comparison, all supervised baselines are trained on the full PrimeVul training set to maximize their learning potential. Both our PacVD framework and the baseline models are then evaluated on this identical test subset.

5.4 LLM Configurations

In our research, we deliberately selected three state-of-the-art LLMs—ChatGPT-4o, DeepSeek Series(V3 and R1), and CodeLLaMA-34b—to ensure a comprehensive evaluation of our approach across a diverse set of model architectures and training paradigms. Our selection was guided by the goal of including distinct and representative categories.

- **ChatGPT-4o**: ChatGPT-4o represents the State-of-the-Art, most powerful, general-purpose proprietary models available [39]. As noted in our study, it possesses an "advanced understanding of context and nuanced text generation capabilities" and benefits from extensive pre-training, establishing a high-performance baseline. Its consistently reliable and well-balanced performance makes it an ideal benchmark to measure against, representing the upper echelon of current LLM technology that is not specialized for code. GPT-4o stands out due to its advanced understanding of context and nuanced text generation capabilities. This model is particularly effective for parsing

and understanding complex language patterns, which is essential for interpreting unstructured text in vulnerability reports and code comments.

- **DeepSeek**: As emerging high-performance reasoning-focused models, DeepSeek models represent the cutting edge of development with a strong focus on enhancing reasoning capabilities. We included DeepSeek-R1 [13], which "almost outperforms all other models" in our tests with its "enhanced contextual comprehension capabilities", to test our method against a model specifically designed for complex reasoning. By also including DeepSeek-V3 [12], which showed "extreme performance volatility", we were able to study how different optimization strategies within the same model family affect the results. DeepSeek-V3 and DeepSeek-R1 are based on the Transformer architecture, leveraging self-attention mechanisms to capture complex dependencies in multi-modal data. DeepSeek-V3 excels in high-performance tasks through large-scale pre-training and fine-tuning, while DeepSeek-R1 is optimized for real-time, resource-constrained environments with lightweight and energy-efficient designs. Both models incorporate knowledge distillation and dynamic optimization, ensuring robustness, adaptability, and scalability across diverse applications.
- **CodeLLaMA-34b**: As leading open-source code-specialized model, CodeLLaMA-34b represents models specifically fine-tuned for programming tasks [43]. Our results identify it as a "code-focused model" with a specialized training corpus. This allows us to test our method on an architecture optimized for code comprehension. A key practical advantage is that it can be run on local infrastructure, a critical requirement for academic reproducibility and for enterprise environments where code cannot be sent to external APIs. Running on the local infrastructure, CodeLLaMA-34b employs the VLLM library, known for its versatility in handling diverse programming tasks. By utilizing this model, we can tailor the output settings to produce highly deterministic responses that are crucial for binary decisions like vulnerability detection (yes or no).
  - **Temperature = 0.1**: This setting is crucial as it reduces randomness in the model's output, ensuring that responses are predictable and focused on the highest probability tokens, which is vital for precise vulnerability assessment.
  - **Top_p = 0.95**: Nucleus sampling helps maintain response quality by focusing on the top 95% of the probability mass, balancing diversity with relevance—a key factor in nuanced code interpretation.
  - **Max_tokens = 512**: Although the task ultimately involves binary classification ("yes" or "no"), the use of Chain-of-Thought Prompting (CoT) and In-Context Learning (ICL) requires the model to produce detailed reasoning in the initial response. A token limit of 512 is set to allow the model to generate a comprehensive and detailed explanation in the first round of interaction, which provides the necessary context and rationale for subsequent decision-making. This design ensures that the generated reasoning is both thorough and consistent with the requirements of binary classification, while also enhancing the interpretability and transparency of the decision process.
  - **Seed = 2025**: To ensure the determinism and reproducibility of our experiments, a fixed random seed was used for all inferences. While parameters like temperature and top_p introduce sampling into the token generation process, fixing the seed guarantees that the model produces the exact same output for a given input every time, thereby eliminating any stochasticity and making our results fully reproducible.

By including models that vary in their training focus (general-purpose vs. code-specific), accessibility (proprietary API vs. open-source), and primary strengths (general context vs. reasoning), we aim to provide robust evidence for the generalizability of our findings. Besides, for ChatGPT and

DeepSeek, we directly utilize their official APIs, while for CodeLLaMA, we deploy it locally and access it through the vLLM library.

5.5 Metrics

We use commonly used metrics in machine learning, including Accuracy, Precision, Recall, F1-score.

5.6 Implementation Details

The experiments were conducted on four Linux servers, each equipped with two NVIDIA A100-SXM4-80GB GPUs and two Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz CPUs with 128G memory. For the experimental environment, we use Python 3.10.14, vllm 0.6.2 for our experiment.

6 Evaluation Results

6.1 RQ1: Results of Different Primitive API Abstraction Levels

Table 4. Experimental Results (values in %)

Strategy	ChatGPT-4o				CodeLLaMA-34b				DeepSeek-V3				DeepSeek-R1			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
without API Level (w/o)																
BP	44.07	43.84	93.02	59.59	48.27	43.64	51.16	46.93	49.48	46.15	83.72	59.50	51.34	47.15	83.22	60.19
RP	46.39	44.67	87.79	59.22	55.84	52.94	10.47	17.48	52.16	47.32	87.39	**61.39**	49.44	47.22	82.93	60.18
CoT	47.94	43.64	59.88	50.49	54.64	40.00	4.65	8.33	45.23	39.52	46.15	42.58	44.25	44.15	98.05	60.89
IC	43.81	44.04	98.84	**60.93**	54.20	50.61	51.88	51.23	43.25	43.25	100.00	60.39	45.09	45.09	100.00	**62.15**
FSR	47.42	44.20	70.93	54.46	45.45	42.96	67.44	52.49	56.73	46.88	10.27	16.85	48.31	46.03	87.42	60.30
FSC	45.74	44.24	84.88	58.17	49.09	45.65	73.26	**56.25**	53.01	35.42	11.97	17.89	49.57	46.23	92.76	61.71
API Level 1 (A1)																
BP	47.37	44.00	64.71	52.38	49.00	54.83	33.35	41.60	56.52	50.88	26.13	34.52	50.91	47.64	80.80	59.94
RP	45.07	43.29	73.53	54.50	50.66	45.39	47.06	46.21	54.76	47.87	40.91	44.12	49.82	47.03	81.75	59.71
CoT	44.74	42.52	66.91	52.00	53.95	44.44	11.76	18.60	47.74	39.85	47.32	43.27	44.53	44.09	100.00	61.20
IC	45.07	44.85	99.26	**61.78**	52.85	47.57	41.18	44.14	44.19	44.14	99.12	**61.08**	46.47	45.83	99.18	**62.69**
FSR	50.00	46.58	80.15	58.92	48.34	46.15	88.24	**60.61**	54.27	47.22	26.15	33.66	51.33	47.18	88.64	61.58
FSC	43.75	42.42	72.06	53.41	45.85	40.82	44.12	42.40	50.34	42.86	32.06	36.68	49.82	46.59	92.80	62.03
API Level 2 (A2)																
BP	55.77	54.87	77.50	64.25	61.38	61.54	72.73	**66.67**	50.00	55.00	13.75	22.00	57.69	56.36	77.50	65.26
RP	53.85	53.03	87.50	66.04	54.48	60.00	42.86	50.00	57.69	65.22	37.50	47.62	57.05	55.96	76.25	64.55
CoT	59.62	57.80	78.75	66.67	44.87	33.33	7.50	12.24	60.90	60.44	68.75	64.33	52.29	51.70	97.44	**67.56**
IC	51.28	51.28	100.00	**67.80**	54.92	57.78	41.94	48.60	52.59	52.24	100.00	**68.63**	49.01	49.65	93.42	64.84
FSR	50.00	51.06	60.00	55.17	51.72	52.76	87.01	65.69	45.70	33.33	3.80	6.82	56.41	54.69	87.50	67.31
FSC	48.08	49.62	82.50	61.97	53.10	55.56	58.44	56.96	51.00	57.50	28.75	38.33	53.85	53.08	86.25	65.71
API Level 3 (A3)																
BP	56.03	56.45	76.64	65.02	53.01	55.41	64.93	**59.79**	56.02	55.56	65.48	60.11	56.77	57.62	71.31	63.74
RP	52.53	53.77	78.10	63.69	48.59	54.17	29.10	37.86	50.58	56.94	29.93	39.23	56.92	56.83	77.61	65.62
CoT	55.25	55.73	78.10	65.05	45.91	45.45	7.30	12.58	54.42	55.83	59.82	57.76	53.42	53.77	96.61	**69.09**
IC	52.53	52.94	98.54	**68.88**	46.49	51.14	36.29	42.45	52.78	53.30	97.41	**68.90**	53.10	53.46	95.87	68.64
FSR	47.08	50.41	44.53	47.29	45.78	49.74	71.64	58.72	48.56	100.00	3.85	7.41	57.59	57.95	74.45	65.18
FSC	55.08	56.55	69.34	62.30	49.00	53.27	42.54	47.30	52.36	62.96	25.19	36.00	51.91	54.01	78.91	64.13
API Level 4 (A4)																
BP	52.74	51.03	69.72	58.93	54.09	53.05	62.59	**57.43**	52.74	55.56	14.08	22.47	54.37	52.44	69.92	59.93
RP	54.11	51.83	79.58	62.78	51.25	51.19	30.94	38.57	55.20	60.00	30.22	40.19	57.64	54.15	79.86	64.53
CoT	54.45	52.41	69.01	59.57	49.32	37.50	6.34	10.84	57.79	57.89	54.55	56.17	51.33	50.20	99.22	**66.67**
IC	48.63	48.61	98.59	**65.12**	50.79	50.00	36.29	42.06	48.15	48.75	97.50	**65.00**	48.02	48.54	93.55	63.91
FSR	47.60	46.50	51.41	48.83	43.77	45.74	73.38	56.35	51.79	63.64	5.07	9.40	51.37	50.00	78.87	61.20
FSC	45.89	46.12	66.90	54.60	49.82	49.21	44.60	46.79	53.38	55.00	24.09	33.50	55.08	53.05	88.28	66.28

Note: BP denotes basic prompt, RP denotes role-based prompt, CoT denotes Chain-of-thought, IC denotes In-context Learning, FSR denotes few-shot learning with random selected examples, FSC denotes few-shot learning with contrastive pair examples. Acc, Pre, Rec, F1 denotes Accuracy, Precision, Recall, F1 score.

6.1.1 Performance Analysis Under Different API Abstraction Levels. As shown in Table 4, in the absence of API information, models generally demonstrate weaker performance, with average F1 scores ranging from 42.00% to 44.00%. Interestingly, simpler prompting strategies like Basic Prompt and role-playing Prompt typically outperform more complex approaches in this environment. However, even in this challenging scenario, the DeepSeek-R1 model exhibited exceptional adaptability, achieving comparatively strong results despite the lack of API information.

The A1 level, providing fuzzy branch information about API calls, delivered noticeable performance improvements over the no-API baseline. In-context prompting emerged as the standout strategy at this level, achieving an average F1 score of 58.91%. The performance gaps between different models became more pronounced, suggesting that API information amplifies the inherent capability differences between models. These results confirm that even basic control flow information adds significant value to vulnerability detection tasks.

At the A2 level, which introduces concrete API call branch conditions, accuracy reached its peak, indicating that specific branch information helps establish clearer decision boundaries. Model-strategy combination performance exhibited greater differentiation, with F1 scores ranging from as low as 6.82% to as high as 68.63%. Notably, the CodeLLaMA-34b model demonstrated exceptional performance at this level, achieving an F1 score of 67.48% with the Basic Prompt strategy. These results highlight the substantial value of detailed control flow information for vulnerability detection.

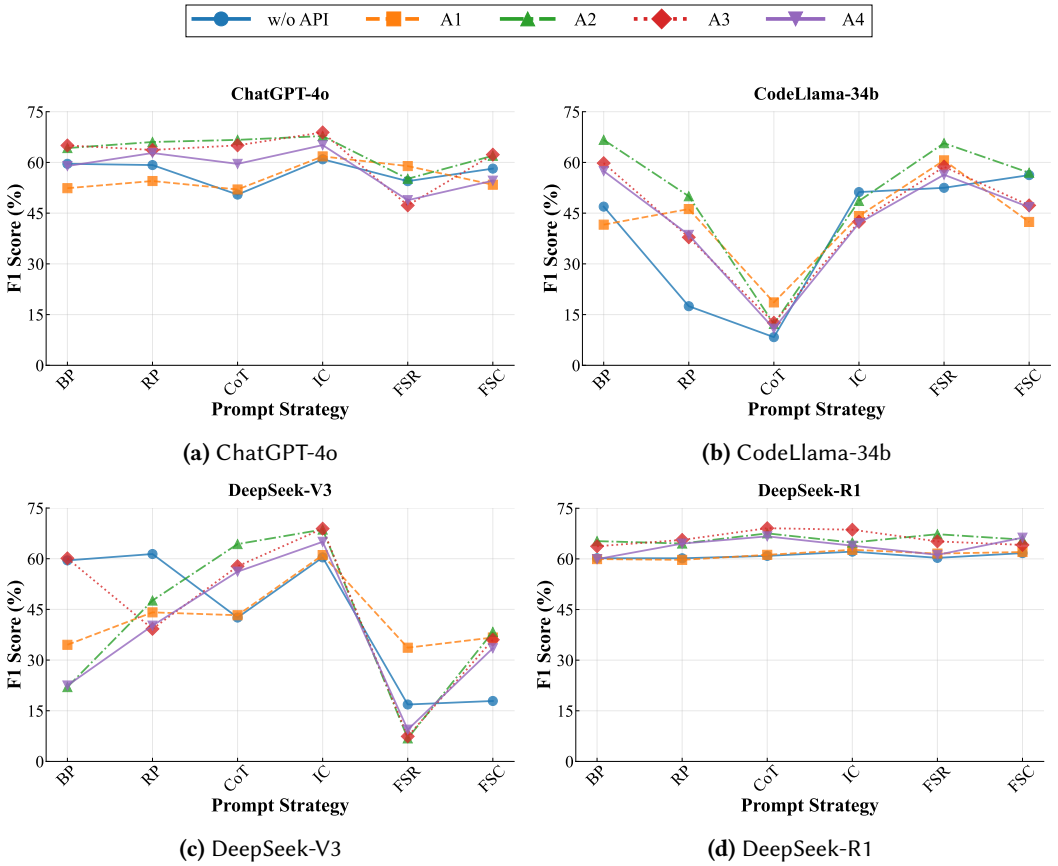
The A3 level, which augments concrete branch conditions with API call frequency information, delivered the best overall performance. Multiple model-strategy combinations achieved their peak performance at this level, with the DeepSeek-R1+CoT combination reaching the highest F1 score of 69.09%, correctly classifying 331 out of 623 samples, an increase of 55 correctly classified samples over its baseline performance. This level appears to represent an optimal balance point between information richness and noise, effectively combining control flow semantics with lightweight statistical features.

Despite incorporating additional variable information, the A4 level showed slightly reduced performance compared to A3. The findings indicate that while data flow information increases complexity, its value may vary depending on model capabilities and specific task characteristics.

6.1.2 API Level Impact on Vulnerability Detection Performance. As shown in Figure 3, the experimental results reveal a distinct pattern in the relationship between API abstraction levels and vulnerability detection performance. From no API information to A3 level, we observed a consistent upward trend in model performance, followed by a slight decline at A4 level.

Even the most basic API abstraction (i.e., A1) provided substantial performance improvements over baseline (i.e., w/o API Level), with average F1 scores increasing by approximately 5%-8%, representing an additional 30 to 50 correctly identified samples across the dataset on average. This underscores the critical importance of API contextual information for effective vulnerability detection. The A3 level, which combines concrete branch conditions with call frequency statistics, emerged as the optimal abstraction point, achieving the highest average F1 score across various model-strategy combinations. Interestingly, the A4 level, despite incorporating additional variable information, showed diminishing returns, suggesting the existence of an information-noise threshold beyond which additional details become counterproductive.

The various evaluation metrics demonstrated different sensitivities to API abstraction levels. Accuracy measurements peaked at the A2 level, suggesting that concrete branch information alone establishes clearer decision boundaries. We also observed a consistent precision-recall trade-off at higher API levels (i.e., A3, A4), where models achieved improved precision at the cost of reduced



(a) ChatGPT-4o **(b)** CodeLlama-34b

(c) DeepSeek-V3 **(d)** DeepSeek-R1

Fig. 3. API Level Comparison Across Different Models and Prompt Strategies. The figure shows F1 scores for different API levels (w/o API, A1, A2, A3, A4) across various prompt strategies (BP, RP, CoT, IC, FSR, FSC) for four different models. Each subplot represents one model, with the legend shown at the top.

recall, indicating a shift toward more conservative vulnerability identification as API information becomes more detailed.

6.1.3 Dissecting Detection Overlap and Complementarity . To move beyond aggregate performance metrics and understand the distinct contributions of each API abstraction level, we conducted an in-depth analysis of their detection overlap. We visualize this relationship using Venn diagrams, which map the set of correctly identified true positives for each level.

Contrary to a simple hierarchical assumption where higher levels of abstraction would subsume the capabilities of lower ones, our analysis reveals a strongly complementary relationship between the levels. As illustrated in Figure 4 for our best-performing combination (DeepSeek-R1 with CoT), each abstraction level identifies a substantial set of unique vulnerabilities. Most notably, the A4 level, with its fine-grained data flow context, uniquely detects 45 vulnerabilities missed by all other levels, demonstrating its indispensable role in uncovering specific, complex vulnerability patterns. Concurrently, even the most basic A1 level uniquely identifies 34 vulnerabilities, suggesting that some patterns are more readily captured by higher-level, less noisy abstractions. The core set of

vulnerabilities detected by all four levels is remarkably small, numbering only 11. This strongly indicates that the choice of contextual granularity is critical for detection and that there is no one-size-fits-all abstraction; the levels offer irreplaceable, distinct views into the code.

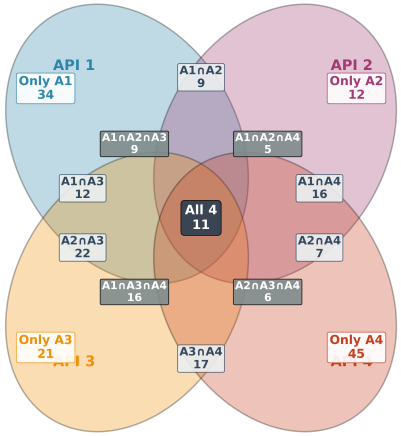


Fig. 4. Venn Diagram of Detection Overlap Across API Abstraction Levels for DeepSeek-R1 (CoT)

This wide detection coverage, particularly from the A4 level, necessitates a closer examination of the trade-off between recall and precision to understand its practical utility. While the A4 level detects the highest absolute number of true positives (128), achieving a near-perfect recall of 99.22%, this comes at the cost of significantly lower precision (50.20%), implying a high rate of false positives (as detailed in Appendix A). In contrast, the A3 level, while detecting a slightly smaller set of vulnerabilities (114), strikes a much more effective balance, resulting in the highest F1-score of 69.09%. This confirms that although A4 casts the widest net, A3 represents a more reliable and efficient choice for practical applications by achieving an optimal equilibrium between detection effectiveness and the cost of false positives.

Furthermore, this complementary relationship is not an artifact of a single prompting strategy but a fundamental characteristic of our approach, albeit one that is significantly modulated by the choice of prompt. A complete set of Venn diagrams for all prompting strategies is provided in Appendix A. The most striking example is the FSC (Few-shot Contrastive) strategy, which achieves the broadest overall coverage by identifying 250 unique vulnerabilities, yet has a "common core" of zero vulnerabilities detected by all four levels simultaneously. This reveals a complex and important interaction between prompt engineering and context abstraction, opening new research directions for combining them to achieve specific goals, such as maximizing detection coverage versus seeking the most reliable, consensus-based detections.

6.1.4 Summary and Rationale for API Level Selection. Experimental results consistently demonstrate that the A3 level (i.e., specific branches + call frequency) provides the optimal cost-performance ratio and performance equilibrium point in vulnerability detection. The advantages exhibited by A3 can be attributed to three fundamental mechanisms: it likely represents an information saturation point where models acquire adequate structured information while additional data yields diminishing marginal returns; it establishes a computational efficiency equilibrium between model processing capability and information complexity, enabling efficient processing of all provided

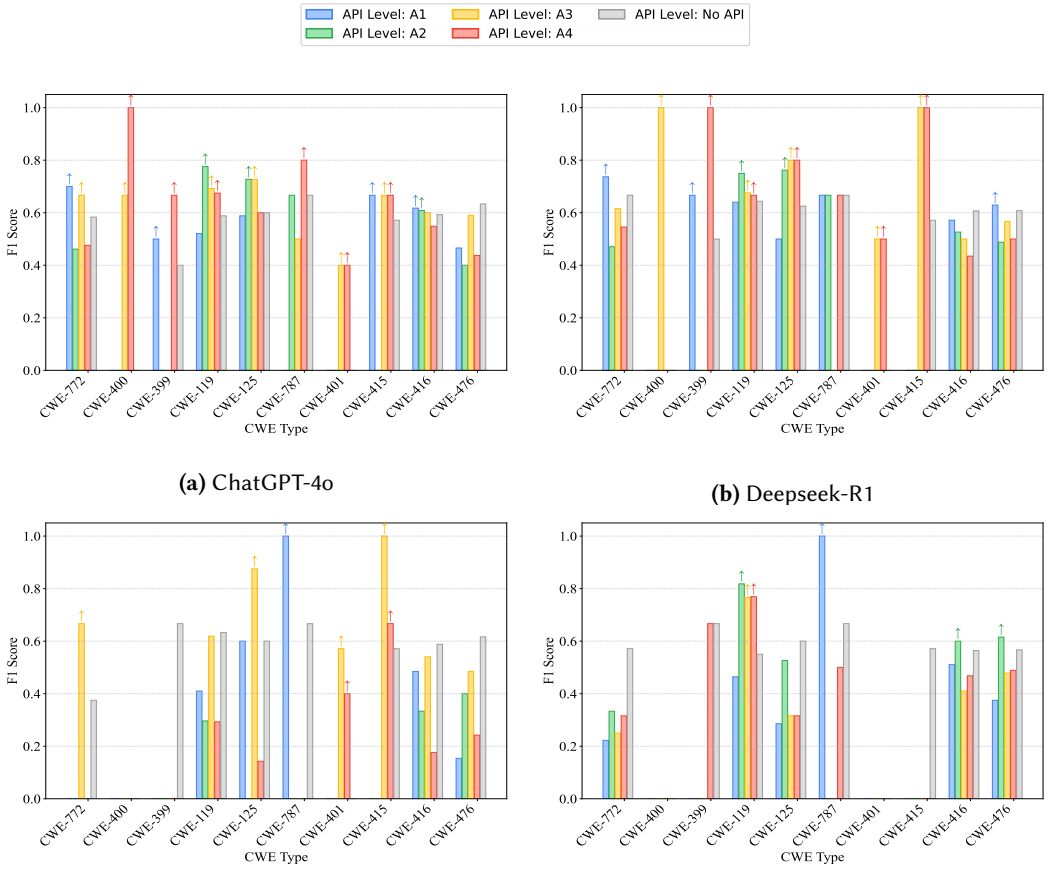
information; and it optimizes representation space compatibility, aligning well with most models' internal representation structures, thus facilitating effective integration of information into the reasoning process.

Besides, the relationship between path sensitivity and abstraction precision provides a theoretical foundation for understanding API abstraction effectiveness. A1 (i.e., fuzzy branches) essentially performs path-insensitive API call abstraction, while A2-A4 maintain varying degrees of path sensitivity, with experimental results confirming that path-sensitive abstractions (i.e., A2/A3/A4) significantly outperform path-insensitive ones (i.e., A1). According to Abstract Interpretation theory, abstraction precision correlates positively with path sensitivity, though a precision-efficiency trade-off exists, with A3 achieving the optimal balance between path sensitivity and abstraction level. This methodologically indicates that API abstraction mechanisms should prioritize control flow path differentiation capabilities rather than adding more dimensional information, supporting the applicability of "abstraction precision preservation theory" in vulnerability detection. Regarding context sensitivity, A2 (i.e., specific branches) provides context-insensitive API call conditions, A3 adds frequency information, while A4 introduces variable associations, increasing context sensitivity to some degree. The superior performance of A3 over A4 suggests that lightweight contextual information is more effective than fully context-sensitive analysis. Furthermore, the performance decline at A4 (i.e., including variable information) likely stems from the inherent limitations of static alias analysis, with static analysis struggling to precisely handle complex pointer alias relationships, introducing erroneous associations in variable-level API abstractions.

> ***RQ1:*** *API integration demonstrates universal benefits, with measurable performance improvements observed even at the lowest abstraction level (A1). A3 level achieves optimal cost-effectiveness, outperforming other tiers in comprehensive metrics. A2 level remains highly competitive, occasionally surpassing A3 in specific model-strategy combinations. However, adding more details (A4) leads to a slight drop in performance, indicating that there is a tipping point between information and noise.*

6.2 RQ2:The Impact of API Abstraction Information on Different Types of Vulnerabilities

As shown in Figure 5, the experimental results demonstrate substantial improvements in boundary-related memory vulnerability detection when applying our API abstraction methodology. For CWE-787 (i.e., Out-of-bounds Write), API Level A1 achieved remarkable performance in both DeepSeek-V3 and CodeLlama-34b models, with F1 scores approaching 100.00%, representing approximately a 67.00% improvement over the No API baseline. This significant enhancement indicates that even minimal API abstraction can substantially improve the detection of critical out-of-bounds write vulnerabilities. For CWE-125 (i.e., Out-of-bounds Read), multiple abstraction levels demonstrated effectiveness across various models. In DeepSeek-R1, DeepSeek-V3, and ChatGPT-4o, both API Level A2 and A3 consistently outperform the No API baseline. Notably, in DeepSeek-V3, API Level A3 achieved an F1 score of approximately 88.20%, representing a 47.86% improvement over the No API baseline. This finding suggests that moderate levels of API abstraction can effectively capture the contextual patterns necessary for identifying out-of-bounds read vulnerabilities. Regarding CWE-119 (i.e., Improper Restriction of Operations within the Bounds of a Memory Buffer), our analysis revealed that the CodeLlama-34b model particularly benefited from API abstraction. API Level A2, A3, and A4 all substantially outperform the baseline without API, with improvements ranging from 43.01% to 52.20%.



(a) ChatGPT-4o

(b) Deepseek-R1

(c) DeepSeek-V3

(d) CodeLlama-34b

Fig. 5. The effect of different API abstraction levels on different vulnerability types, Bar height represents F1 score. Higher is better. Arrows (↑) indicate significant improvements over No API baseline.

> ***RQ2-1:*** *The consistent effectiveness across all abstraction levels for boundary-related memory errors indicates that our approach exhibits universality in detecting memory-related vulnerabilities, regardless of the specific abstraction granularity employed.*

For CWE-415 (i.e., Double Free), the DeepSeek-R1 model demonstrated exceptional performance with API Level A3 and A4, both achieving F1 scores of approximately 97.00%, representing a 70% improvement over the No API baseline. Similarly, in DeepSeek-V3, API Level A3 demonstrated substantial improvement over the baseline. CWE-400 (i.e., Uncontrolled Resource Consumption) detection was significantly enhanced in both DeepSeek-R1 and ChatGPT-4o models when using API Level A3 and A4. Most notably, ChatGPT-4o with API Level A4 achieved a F1 score of approximately 100.00 for this vulnerability type. Additionally, for CWE-399 (i.e., Resource Management Errors), DeepSeek-R1 with API Level A4 also attained a perfect F1 score of approximately 100.00%.

> ***RQ2-2:*** *Resource management-related vulnerabilities particularly benefit from higher abstraction levels (A3 and A4) that provide more detailed contextual information. The enhanced performance suggests that accurate detection of these vulnerability types requires more comprehensive API call representations that explicitly model resource allocation and deallocation patterns.*

Our experimental results reveal distinct effectiveness patterns across different API abstraction levels: API Level A1, despite being the lowest abstraction level, demonstrated exceptional performance for specific vulnerabilities. It significantly improved detection of CWE-787 in DeepSeek-V3 and CodeLlama-34b, achieving near-perfect F1 scores. In ChatGPT-4o, A1 enhanced detection for CWE-772, CWE-415, and CWE-416, while in DeepSeek-R1, it improved results for CWE-399, CWE-772, and CWE-476. These results indicate that even minimal API abstraction can substantially improve certain vulnerability detections, particularly for well-defined patterns like out-of-bounds writes.

API Level A2 exhibited particular effectiveness for CWE-119, CWE-125, and CWE-416 vulnerabilities. In ChatGPT-4o, it improved detection for all three vulnerability types, while in DeepSeek-R1, it enhanced detection for CWE-119 and CWE-125. CodeLlama-34b benefited from A2 abstraction for CWE-119, CWE-416, and CWE-476. This pattern indicates that A2's intermediate abstraction level is particularly suitable for detecting access violations and pointer mismanagement issues.

API Level A3 demonstrated the most consistent performance improvements across multiple models and vulnerability types. It showed significant enhancements for CWE-772 in ChatGPT-4o and DeepSeek-V3, CWE-119 in ChatGPT-4o, DeepSeek-R1, and CodeLlama-34b, CWE-125 in ChatGPT-4o, DeepSeek-R1, and DeepSeek-V3, and CWE-415 across all evaluated models. This consistency suggests that A3 represents an optimal balance between abstraction and specificity for general memory safety vulnerability detection.

API Level A4, the highest abstraction level, showed targeted effectiveness for specific model-vulnerability combinations. It significantly improved detection of CWE-399, CWE-119, CWE-415, CWE-400, and CWE-787 in ChatGPT-4o, as well as CWE-399, CWE-119, CWE-125, and CWE-415 in DeepSeek-R1. Additionally, it enhanced detection of CWE-415 in DeepSeek-V3 and CWE-119 in CodeLlama-34b. This pattern suggests that highly detailed API abstractions are particularly beneficial for complex vulnerability types that require comprehensive contextual understanding.

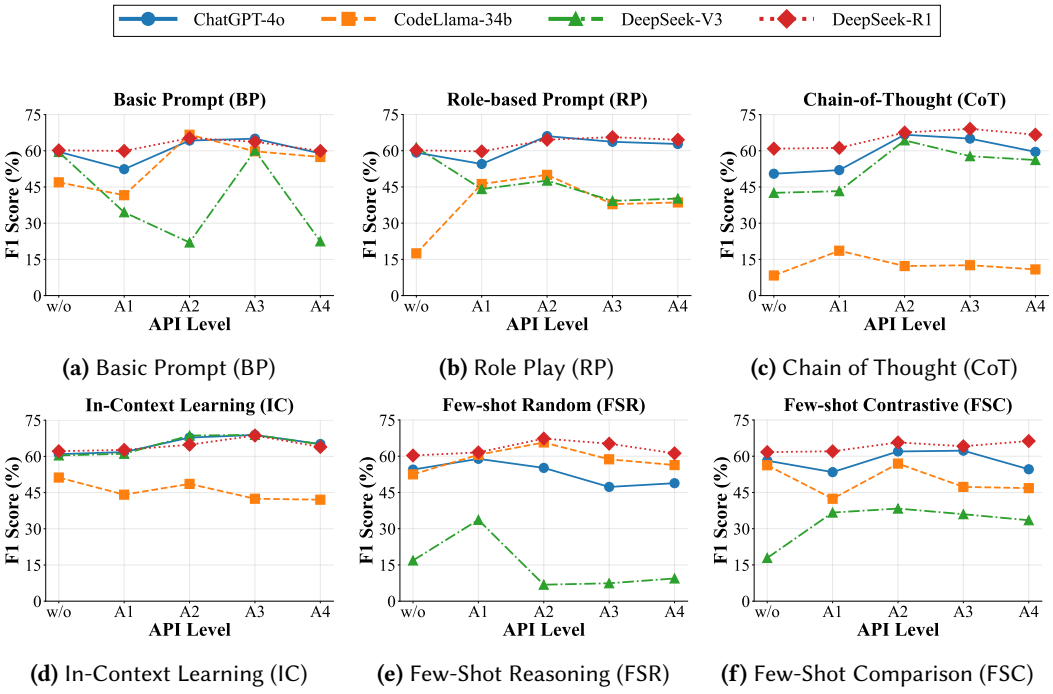
> ***RQ2-3:*** *API Level A3 represents the most balanced and consistently effective abstraction level across multiple vulnerability types and models, suggesting it captures an optimal degree of contextual information without introducing excessive noise.*

To summarize the key findings from our analysis on vulnerability types, Table 5 provides a mapping of each API abstraction level to the category of vulnerabilities it is best suited to detect. This mapping reflects the underlying nature of different vulnerabilities. API Level A1 is effective for bugs with simple, distinct control-flow patterns, such as out-of-bounds writes (CWE-787), where the mere presence of a dangerous function is a strong signal. API Level A2 excels with vulnerabilities where the flaw lies in the conditional logic itself, such as improper buffer restrictions (CWE-119), as it provides the exact branch conditions for analysis. For vulnerabilities defined by a quantitative imbalance, like a double free (CWE-415), API Level A3 is most suitable because its call frequency data allows the model to detect hazardous operational patterns. Finally, API Level A4 is best for complex resource management errors (CWE-400, CWE-399), which require the data-flow tracking of specific variables, even at the risk of introducing noise from static analysis.

6.3 RQ3: Effectiveness of Different LLMs and Prompt Engineering Strategies

Table 5. Suitability of API Abstraction Levels for Different Vulnerability Types

Abstraction Level	Description	Best Suited Vulnerability Type
API Level 1	Fuzzy Branches	Out-of-Bounds Write (CWE-787)
API Level 2	Concrete Branches	Improper Restriction of Operations within the Bounds of a Memory Buffer (CWE-119)
API Level 3	Concrete Branches + # Calls	Double Free (CWE-415)
API Level 4	Concrete Branches + Key Variables	Uncontrolled Resource Consumption (CWE-400), Resource Management Errors (CWE-399)



(a) Basic Prompt (BP) **(b)** Role Play (RP) **(c)** Chain of Thought (CoT)

(d) In-Context Learning (IC) **(e)** Few-Shot Reasoning (FSR) **(f)** Few-Shot Comparison (FSC)

Fig. 6. Model performance comparison across six different strategies.

6.3.1 Large Language Model Performance Analysis. As shown in Figure 6, the experimental results demonstrate significant performance variations across different language models in API-assisted vulnerability detection tasks.

DeepSeek-R1 almost outperforms all other models across all API levels. It achieves optimal performance at API level A3, the F1 score is 69.09%, with IC strategy proving most effective at lower API levels while CoT excels at higher levels (A2-A4). This model maintains robust performance stability across all API information levels and prompting strategies, while maintaining exceptional recall rates throughout the evaluation. This superior performance can be theoretically attributed to its enhanced contextual comprehension capabilities through efficient attention mechanisms,

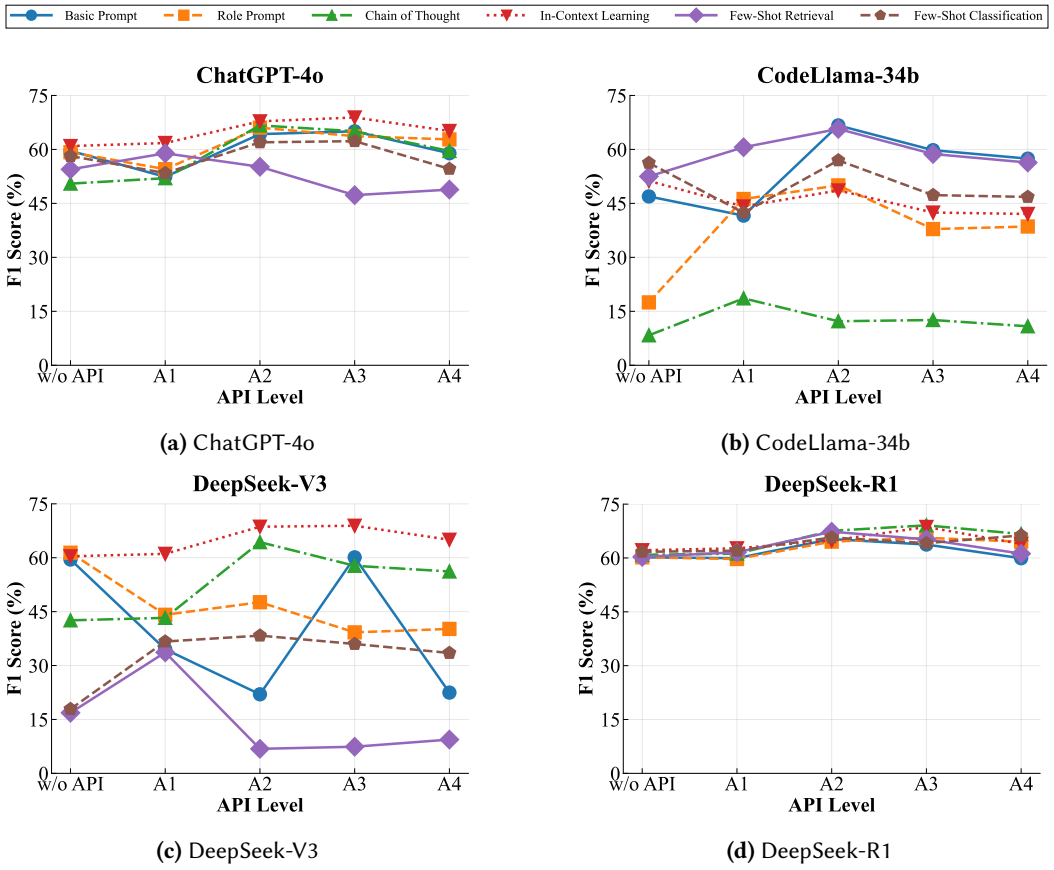
enabling precise capture of critical information within extended sequences. The model's exposure to diverse pre-training data likely provided extensive experience with API-related contexts, while its balanced inference capabilities maintained high precision without sacrificing recall, indicating well-equilibrated internal representation spaces capable of flexibly adjusting decision boundaries across diverse tasks.

ChatGPT-4o demonstrates a progressive improvement in F1 scores as API levels increase, reaching its maximum performance at Level 3 , which is 68.88%, before experiencing a slight decline at Level 4, with In-Context Learning (IC) strategy consistently outperforming all alternatives across all levels. ChatGPT-4o demonstrates consistently reliable performance, it maintains above-average performance at all API levels, and is second only to DeepSeek-R1 in non-API and A1 environments, particularly excelling when paired with In-context strategies and demonstrating well-balanced performance metrics. This consistency can be attributed to diversified training objectives, with OpenAI potentially employing varied loss functions and evaluation metrics to prevent over-optimization on specific metrics at the expense of others. More extensive pre-training likely established a robust knowledge foundation, reducing dependence on specific API or prompting methodologies, while effective fine-tuning strategies emphasized stable performance across diverse environments.

DeepSeek-V3 achieves peak effectiveness at API Level 3, F1 score reaches 68.90%, with IC emerging as the most efficacious strategy upon API information integration, demonstrating substantial enhancement from no-API to Level 3, followed by a modest performance reduction at Level 4. DeepSeek-V3 exhibites extreme performance volatility, displaying high sensitivity to both API levels and prompting strategies. While achieving exceptional results under specific configurations (i.e., F1 score reaches 68.63% with In-context strategy at A2 level), it performed poorly under others (i.e., F1 score reaches 7.41% with few-shot-random at A3 level), indicating substantial potential requiring precise calibration. These extreme fluctuations potentially reflect over-specialization issues, with the model potentially over-optimized for specific domains or tasks, limiting generalization capabilities. Its high parameter sensitivity with architectural features particularly responsive to input variations causes significant performance fluctuations from minor prompting adjustments, while unbalanced representation spaces exhibit bias toward certain prompting patterns while neglecting others.

CodeLLaMA-34b exhibits optimal performance at API Level 2, F1 score reaches 67.48%, showing marked superiority compared to other levels, though its optimal strategy varies—Basic Prompting (BP) proves most effective at A2, while Few-Shot with Random examples (FSR) dominates at Levels 3 and 4; performance noticeably deteriorates following A2, exhibiting limitation in recall. This pronounced sensitivity to API levels likely stems from its specialized training corpus; as a code-focused model, it may lack sufficient contextual comprehension in low-API environments while effectively utilizing structured information as API levels increase. Its attention mechanisms may be optimized for structured and semi-structured data analysis, while its preference for precision over recall suggests conservative internal decision thresholds.

> ***RQ3-1:*** *Reasoning-focused models like DeepSeek-R1 exhibit minimal sensitivity to API abstraction gradations, maintaining consistent performance across all levels and thus proving suitable for general applications. ChatGPT-4o, with its extensive training corpus, displays moderate sensitivity with stable cross-tier performance. Both these two models can effectively operate with low cost and level API abstraction configurations. Conversely, code-specialized CodeLLaMA-34b shows pronounced sensitivity, with significant improvement when transitioning from no abstraction to level A2. Similarly, DeepSeek-V3 demonstrates high sensitivity with irregular performance patterns across different tiers. These latter two models require more detailed API abstraction levels and more overhead to optimize vulnerability detection.*



(a) ChatGPT-4o **(b)** CodeLlama-34b

(c) DeepSeek-V3 **(d)** DeepSeek-R1

Fig. 7. API Level Comparison Across Different Models and Prompt Strategies. The figure shows F1 scores for different API levels (w/o API, A1, A2, A3, A4) across various prompt strategies (BP, RP, CoT, IC, FSR, FSC) for four different models. Each subplot represents one model, with the legend shown at the top.

6.3.2 Prompting Strategy Performance Analysis. As shown in Figure 7, In-context Learning demonstrated superior performance across API levels with an average F1 score of 59.00%, particularly excelling at A2 and A3 levels while maintaining high recall values. This effectiveness stems from direct example provision reducing model burden, implicit runtime adjustment effects, and expanded decision boundaries.

Chain of Thought (CoT) performed well only at higher API levels (A2-A4) with more capable models, reaching optimal F1 scores of 69.09% with DeepSeek-R1 at A3. Performance varies significantly based on model computational capacity, with less capable combinations suffering from error accumulation in multi-step reasoning.

Role-playing maintained consistent performance across all configurations with minimal performance drops, proving most reliable especially at A4 level with DeepSeek-R1. This stability results from established response frameworks constraining output space and effective activation of role-relevant knowledge clusters.

Basic Prompt achieved optimal performance in no-API environments (i.e., F1 score is 58.86% and with CodeLLaMA-34b at A2 level (i.e., F1 score is 67.48%), though effectiveness decreased at higher

API levels. In information-scarce scenarios, direct queries allow focused computational resource allocation on core reasoning rather than processing complex prompts.

Few-shot strategies performed well primarily with DeepSeek-R1, reaching 67.31% F1 with random sampling at A2 level, but showed inconsistent results with other models. This variability reflects differing minimum effective sample quantity requirements and representation space compatibility issues across models.

> ***RQ3-2:*** *Prompt strategy effectiveness exhibits clear dependency on API abstraction level. Without API abstraction, Basic Prompt outperforms other strategies. At low abstraction (A1), In-context Learning demonstrates superior performance, followed by few-shot approaches. Medium abstraction (A2) benefits most from In-context Learning and Chain-of-Thought techniques. At high abstraction levels (A3-A4), both In-context Learning and Chain-of-Thought prompting yield comparable optimal results. These findings indicate that as contextual information richness increases through API abstraction, more sophisticated reasoning-oriented prompt strategies become advantageous.*

6.4 RQ4: Baseline Comparison

Table 6. Performance Comparison of PacVD against LLM-based Baselines and Traditional SOTA Baselines on the NVD-New Dataset (%). Recall values of 100.00% are not highlighted to emphasize more meaningful recall scores.

PacVD Configuration	Metric	PacVD	AL-C	AS-C	SS-C	RS-C	HS-C	SySeVR	ReGVD	Devign
DeepSeek-R1+CoT+A3	Acc	53.42	50.54	50.00	47.37	53.26	49.46	52.20	50.24	**55.55**
	Pre	53.77	50.00	49.30	48.86	50.00	48.81	52.34	**54.80**	41.17
	Rec	96.61	**97.83**	94.59	89.58	97.67	91.11	69.71	51.12	41.17
	F1	**69.09**	66.18	64.81	63.24	66.14	63.57	59.79	52.90	41.17
DeepSeek-R1+IC+A3	Acc	**53.10**	50.00	52.05	50.00	51.65	48.94	52.20	50.24	55.55
	Pre	53.46	50.00	51.47	49.44	50.56	48.86	52.34	**54.80**	41.17
	Rec	**95.87**	93.48	94.59	100.00	100.00	93.48	69.71	51.12	41.17
	F1	**68.64**	65.15	66.67	66.17	67.16	64.18	59.79	52.90	41.17
DeepSeek-V3+IC+A3	Acc	52.78	46.94	50.00	50.00	48.98	51.02	52.20	50.24	**55.55**
	Pre	53.30	47.62	50.00	49.41	48.91	50.00	52.34	**54.80**	41.17
	Rec	**97.41**	83.33	92.50	87.50	93.75	93.75	69.71	51.12	41.17
	F1	**68.90**	60.61	64.91	63.16	64.29	65.22	59.79	52.90	41.17
GPT-4o+IC+A3	Acc	52.53	50.00	51.25	52.58	48.98	48.45	52.20	50.24	**55.55**
	Pre	52.94	48.94	50.63	51.06	48.98	48.94	52.34	**54.80**	41.17
	Rec	**98.54**	100.00	100.00	100.00	100.00	95.83	69.71	51.12	41.17
	F1	**68.88**	65.71	67.23	67.61	65.75	64.79	59.79	52.90	41.17
CodeLLaMA-34b+BP+A2	Acc	61.38	60.00	61.46	56.25	56.25	**62.50**	52.20	50.24	55.55
	Pre	61.54	**66.67**	64.10	57.50	58.82	65.79	52.34	54.80	41.17
	Rec	**72.73**	40.00	52.08	47.92	41.67	52.08	69.71	51.12	41.17
	F1	**66.67**	50.00	57.47	52.27	48.78	58.14	59.79	52.90	41.17

6.4.1 Performance on NVD-New Dataset. Our comprehensive evaluation demonstrates that PacVD consistently outperforms existing baselines across all tested model configurations for vulnerability detection, as shown in Table 6.

PacVD achieves superior F1 scores across all model configurations, indicating a robust and generalizable approach to vulnerability detection. This performance advantage stems from PacVD's ability to maintain an optimal balance between precision and recall—a critical factor in practical vulnerability detection systems. While most baseline methods exhibit a tendency toward high recall at the expense of precision, particularly All Callees and Random Sampling approaches, PacVD consistently maintains high recall rates while simultaneously achieving improved precision.

Besides, the performance characteristics vary notably across different model configurations. With DeepSeek-R1 + CoT, PacVD achieves its highest F1 score, outperforming the closest baseline by 4.40%. Similar performance advantages are observed with DeepSeek-R1 + In-Context and DeepSeek-V3 + In-Context configurations, the F1 score of PacVD outperforms the closest baseline by 2.20% and 5.64%. Under the GPT-4o + In-Context setting, the F1 score of PacVD outperforms the closest baseline by 1.88%. Although multiple baselines achieve perfect recall, their precision suffers substantially, whereas PacVD provides a more balanced performance profile. Under the CodeLLaMA-32b + Basic configuration, the F1 score of PacVD outperforms the closest baseline by 16.06%. Interestingly, the CodeLLaMA-32b + Basic configuration exhibits distinct characteristics, with generally higher precision but lower recall across all methods, yet PacVD still maintains the highest overall F1 score.

Our analysis of individual sampling strategies reveals important insights. The All Callees (AL-C) approach, while comprehensive, suffers from precision limitations despite its high recall in most configurations. API-guided Sampling (AS-C) demonstrates moderate stability across different model settings, while Similarity-based Sampling (SS-C) shows considerable performance variation across models, performing particularly well with GPT-4o. Random Sampling (RS-C), despite its simplicity, achieves surprisingly competitive results in certain configurations, often with perfect recall but compromised precision. Hierarchy Sampling (HS-C) exhibits model-dependent performance, excelling primarily with CodeLLaMA-32b but underperforming with other models. The collective performance of these baselines provides robust empirical evidence for our initial claim in Section 2.1. Their consistent underperformance compared to PacVD, often characterized by lower precision, confirms that including unabstracted contextual code introduces significant noise that leads to a higher rate of false positives, thereby validating the need for our approach.

From a practical perspective, our results indicate that PacVD offers substantial advantages for real-world vulnerability detection systems by reducing false positives while maintaining high detection rates. The consistent performance across diverse model configurations suggests that PacVD provides a robust foundation for deployment in varied software engineering environments.

We further benchmarked PacVD against three established SOTA detectors: SySeVR, ReGVD, and Devign. As presented in Table 6, the results confirm that all PacVD configurations systematically outperform these traditional baselines across key performance metrics.

In terms of overall performance, our five PacVD configurations achieved F1-Scores between 67.48% and 69.09%, substantially surpassing the traditional baselines, whose best F1-Score was only 59.79% (SySeVR). Our top-performing configuration (DeepSeek-R1 + CoT) exceeded this mark by over 9 percentage points, demonstrating a significant performance leap.

Furthermore, PacVD achieves a superior balance between recall and precision. While traditional methods struggled to attain high performance in both, our framework excels across different optimization goals. Configurations using GPT-4o reached a near-perfect recall of 98.54%, minimizing false negatives, while the CodeLLaMA configuration delivered the highest accuracy (63.45%) and precision (63.95%), ensuring highly reliable and trustworthy results.

In conclusion, the PacVD framework is demonstrably superior, offering robust and versatile performance that surpasses traditional vulnerability detection methods.

6.4.2 Performance on PrimeVul Dataset. To further validate PacVD's effectiveness against state-of-the-art methods, we also conduct a comparative evaluation on the PrimeVul dataset, as described in Section 5.3. The results are presented in Table 7. The experimental results substantiate the effectiveness and practical value of our PacVD method.

First, Context is the Decisive Factor, Not the LLM Alone. Our results reveal that simply applying LLMs to vulnerability detection without sufficient context is ineffective. In the basic without API

Table 7. Comprehensive Performance Comparison on the PrimeVul Dataset (%). Recall values of 100.00% are not highlighted to emphasize more meaningful recall scores.

Model Configuration		PacVD		Basic	Baselines							
	Metric	A2	A3	w/o API	AL-C	AS-C	SS-C	RS-C	HS-C	SySeVR	ReGVD	Devign
ChatGPT+IC	Pre	44.70	48.36	12.16	45.71	48.19	47.52	45.19	45.19	48.02	**56.93**	44.44
	Rec	97.50	96.72	90.00	100.00	100.00	100.00	**97.92**	**97.92**	53.28	47.56	47.05
	F1	61.30	64.48	21.43	62.75	**65.04**	64.43	61.84	61.84	50.51	51.82	45.71
CodeLlama+BP	Pre	83.33	**87.65**	18.87	58.62	60.00	51.22	54.55	63.16	48.02	56.93	44.44
	Rec	58.82	**78.42**	100.00	37.78	50.00	43.75	37.50	50.00	53.28	47.56	47.05
	F1	68.97	**82.78**	31.75	45.95	54.55	47.19	44.44	55.81	50.51	51.82	45.71
DeepSeek-V3+IC	Pre	44.55	48.94	18.87	46.39	46.75	46.81	47.31	45.92	48.02	**56.93**	44.44
	Rec	91.87	92.34	100.00	**93.75**	90.00	91.67	91.67	**93.75**	53.28	47.56	47.05
	F1	60.00	**63.97**	31.75	62.07	61.54	61.97	62.41	61.64	50.51	51.82	45.71
DeepSeek-R1+CoT	Pre	45.61	48.51	19.61	46.39	47.50	47.52	46.08	47.52	48.02	**56.93**	44.44
	Rec	85.99	78.81	100.00	93.75	95.00	100.00	**97.92**	100.00	53.28	47.56	47.05
	F1	59.60	60.06	32.79	62.07	63.33	**64.43**	62.67	**64.43**	50.51	51.82	45.71

setting (w/o API), the LLM's performance is significantly lower than traditional baselines. However, once augmented with PacVD's API context, every LLM configuration substantially surpass these traditional methods. This clearly demonstrates that our contextual abstraction is the decisive factor driving the performance improvement.

Second, PacVD demonstrates a consistent and fundamental advantage over existing methods. This is most evident in the CodeLlama+BP configuration, where PacVD (A3) achieved an F1-score of 82.78%. This represents a generational leap in effectiveness, outperforming the best LLM-based baseline (HS-C, 55.81%) by nearly 48% and the best traditional baseline (ReGVD, 51.82%) by over 60%. More broadly, across all tested configurations, PacVD's performance vastly surpasses both the basic "without API" method(w/o API) and all traditional deep learning baselines. This reaffirms our core hypothesis that providing effective API context to LLMs is a fundamentally superior approach for vulnerability detection.

Third, PacVD delivers more balanced and practical results by effectively reducing the false positive rate. In the ChatGPT+IC and DeepSeek-R1+CoT tests, some baselines obtained high F1-scores by sacrificing precision (as low as 47%) to achieve perfect recall (100%), leading to an unacceptably high rate of false positives. In contrast, our PacVD (A3) achieved the highest precision in both scenarios (48.36% and 48.51%, respectively). This demonstrates that PacVD guides models to make more precise judgments, yielding more reliable and trustworthy results with fewer false alarms—a critical advantage in real-world applications.

Furthermore, the Performance of PacVD is robust. This performance is particularly significant because it was achieved on a challenging, imbalanced dataset. While some baseline methods struggled with the class imbalance, leading to a notable drop in precision, PacVD maintained a high precision of 87.65% on CodeLlama + Basic Prompting. This result is particularly significant, as high precision in an imbalanced setting directly indicates a low false-positive rate, confirming that PacVD's performance is not an artifact of an artificially balanced dataset and is more likely to translate to production environments.

Finally, PacVD with A3 is superior. In three out of four model configurations, the A3 abstraction (which augments concrete branch conditions with API call frequency) achieved a higher F1-score than the A2 abstraction (which only includes branch conditions). This performance gain was particularly substantial for the best-performing CodeLlama+BP configuration, where the F1-score increased dramatically from 68.97% (A2) to 82.78% (A3). This suggests that quantitative information, such as call frequency, provides the LLM with a more nuanced and valuable context for identifying vulnerabilities.

To validate the effectiveness of the PacVD framework, we benchmark it against SySeVR, ReGVD, and Devign. The results, presented in Table 7, unequivocally demonstrate that PacVD achieves a substantial, often generational, performance leap over these traditional baselines.

First, PacVD demonstrates overwhelming superiority in overall performance (F1-Score). Our PacVD configurations achieve F1-Scores ranging from 60.06% to 82.78%, whereas the traditional baselines are confined to a lower bracket of 45% to 52%. Notably, our top-performing configuration, PacVD (CodeLlama+BP+A3), with an F1-Score of 82.78%, shows a relative improvement of nearly 60% over the best-performing baseline, ReGVD (51.82%). Furthermore, even our most conservative configuration (60.06%) surpasses the performance ceiling of all baselines, highlighting the framework's robustness and advanced capabilities.

Furthermore, PacVD resolves the critical precision-recall trade-off that plagues traditional methods. The baselines exhibit a significant weakness in detection capability (Recall), with the best performer, SySeVR, only reaching 53.28%—effectively missing almost half of all real vulnerabilities. In stark contrast, our PacVD (CodeLlama+BP+A3) configuration excels on both fronts. It delivers an industry-leading Precision of 87.65% while maintaining a high Recall of 78.42%. This balanced profile demonstrates PacVD's ability to be both comprehensive in detection (low false negatives) and highly reliable in its findings (low false positives).

> ***Summary for RQ4:*** *Our approach demonstrates competitive performance against existing baselines, achieving the highest Accuracy, Precision and F1 scores in all configurations while maintaining exceptional recall rates, indicating a better balance between precision and false alarm reduction that makes it particularly suitable for practical vulnerability detection scenarios.*

6.5 RQ5: Generalizability

Table 8. Performance Comparison of PacVD against Baselines on the PrimeVul Dataset Evaluating Generalizability across Diverse Vulnerability Types(%). Recall values of 100.00% are not highlighted to emphasize more meaningful recall scores.

Model	Metric	PacVD-A3	w/o API	AL-C	AS-C	SS-C	RS-C	HS-C	SySeVR	ReGVD	Devign
ChatGPT+IC	Pre	44.97	43.42	47.87	50.00	48.94	47.87	48.45	48.85	**54.74**	50.00
	Rec	**97.81**	94.29	93.75	95.00	95.83	95.74	97.92	46.72	47.47	63.64
	F1	61.61	59.46	63.38	**65.52**	64.79	63.83	64.83	47.76	50.85	56.00
CodeLlaMa+BP	Pre	48.09	8.27	58.62	60.00	51.22	54.55	**63.16**	48.85	54.74	50.00
	Rec	47.01	**74.42**	37.78	50.00	43.75	37.50	50.00	46.72	47.47	63.64
	F1	47.55	14.88	45.95	54.55	47.19	44.44	55.81	47.76	50.85	**56.00**
DeepSeek-V3+IC	Pre	49.55	43.24	49.47	50.63	49.47	48.96	49.47	48.85	**54.74**	50.00
	Rec	81.02	91.43	**97.92**	100.00	**97.92**	**97.92**	**97.92**	46.72	47.47	63.64
	F1	61.50	58.72	65.73	**67.23**	65.73	65.28	65.73	47.76	50.85	56.00
DeepSeek-R1+CoT	Pre	45.42	40.54	51.43	43.24	47.62	100.00	45.45	48.85	**54.74**	50.00
	Rec	**97.81**	65.22	75.00	88.89	76.92	33.33	75.00	46.72	47.47	63.64
	F1	**62.04**	50.00	61.02	58.18	58.82	50.00	56.60	47.76	50.85	56.00

We have generalized our concept from Primitive APIs to a broader and more extensible category of Security-Sensitive APIs. This reframing clarifies that our approach can target any function call that is relevant to a particular vulnerability category. To build this comprehensive collection, we expand upon our initial list, which was derived from an analysis of widely used C language libraries and POSIX standard APIs (e.g., <stdlib.h>, <stdio.h>). Crucially, to significantly broaden the scope, we also integrate the extensive set of vulnerability-related APIs from the well-regarded SySeVR framework. SySeVR propose 811 C/C++ library/API function calls known to be associated with

126 distinct vulnerability types (CWE IDs). These API mappings were originally curated by the commercial static analysis tool Checkmarx, lending them industrial relevance and credibility. The complete, expanded list of these monitored APIs is detailed in Table 11 (Appendix B), enabling our method to address a much wider range of prevalent vulnerability classes.

To validate the generalizability of our approach, we conduct an extensive evaluation on the PrimeVul dataset, which contains a diverse range of vulnerability types. The results clearly demonstrate that the context-enhanced LLM paradigm is successful in this generalization task. As shown in Table 8, the F1 scores of nearly all LLM configurations (generally above 60%) significantly surpass those of all traditional state-of-the-art baselines, including SySeVR (47.76%), GNN-ReGVD (50.85%), and Devign (56.00%). This establishes the overall superiority of our explored paradigm compared to traditional methods.

Our proposed PacVD with API Level 3 (i.e., A3) demonstrate its unique value and strong potential in this generalization test. In combination with `ChatGPT+IC`, PacVD achieves the highest Recall (97.81%) among all methods, which is crucial for avoiding the omission of critical vulnerabilities in practical security audits. Furthermore, when paired with `DeepSeek-R1+CoT`, PacVD obtains the highest F1 Score (62.04%) and Recall((97.81%)) within that model group, proving its effectiveness in guiding the model's complex reasoning.

There is no single, universally optimal context strategy, its effectiveness is highly synergistic with the LLM's architecture. For instance, the `AS-C` (API-guided context) baseline achieved the highest F1 scores for `ChatGPT` and `DeepSeek-V3` (65.52% and 67.23%, respectively), whereas PacVD performs best on `DeepSeek-R1`. This reveals that tailoring the contextual information for different LLM architectures is a key direction for future research.

> ***Summary for RQ5:*** *PacVD demonstrates generalizability by enhancing the performance of several LLM models on a more diverse vulnerability dataset. On the PrimeVul dataset, it consistently outperforms traditional SOTA baselines and achieves the highest recall and F1 score in certain LLM configurations, proving its superior capability in comprehensively identifying potential vulnerabilities across a broad spectrum of types.*

7 Discussion

7.1 Case Study: The "Less is More" Effect in Contextual Vulnerability Detection

Our aggregate performance metrics in Section 6.1 revealed a counter-intuitive pattern: the A4 context, despite containing more information than A3, often led to a degradation in vulnerability detection performance. To investigate the underlying causes of this phenomenon, we conducted a detailed analysis of 79 specific instances where models successfully identified a vulnerability with A3 context but failed with A4 context, as shown in Table 9. This deep dive confirms that adding more detail can introduce "noise," and allows us to categorize this noise into three distinct archetypes: Context Overflow, Information Overload, and Analysis Focus Drift. Our findings show that this issue is particularly pronounced in the DeepSeek model series, while GPT-4o demonstrates greater resilience, with only two such failure cases observed.

Table 9. API Level 3 Analysis Success but API Level 4 Failure Cases Statistics

Model	A3 Success/A4 Failure Count
DeepSeek-R1+cot	20
DeepSeek-R1+IC	30
GPT-4o+IC	2
DeepSeek-V3+IC	27

7.1.1 Type I: Context Overflow Noise. The most direct form of performance degradation occurs when the A4 context becomes so verbose that it exceeds the model's effective processing limits, resulting in a complete failure to generate an analysis. This "Context Overflow" was the most frequent failure mode, observed in a majority of the 79 cases across the DeepSeek models. A representative example is CVE-2019-15140 in `ReadMATImageV4`.

```
static Image *ReadMATImageV4(const ImageInfo *image_info,Image *image, ExceptionInfo *exception)
{
  // ... [Function Body with complex loops, switch cases, and memory management] ...
  quantum_info=AcquireQuantumInfo(image_info,image);
  if (quantum_info == (QuantumInfo *) NULL)
    return(DestroyImageList(image));
  // ... [Extensive image data processing] ...
  pixels=(unsigned char *) GetQuantumPixels(quantum_info);
  // ... [Loop to read and process pixels] ...
  if (quantum_info != (QuantumInfo *) NULL)
    quantum_info=DestroyQuantumInfo(quantum_info);
  // ... [Further logic and cleanup] ...
  return(GetFirstImageInList(image));
}
```

Listing 2. Simplified source code for CVE-2019-15140, `ReadMATImageV4`

The A3 context, at 1,089 characters, provided a high-level summary of the function's main API calls. With this, the DeepSeek-R1 (CoT) model correctly identified an out-of-bounds write vulnerability.

A3 Prediction (Correct):yes. The code contains multiple critical vulnerabilities, including: 1. out-of-bounds write vulnerability: the code writes to 'image->colormap[i]' without bounds checking... 2. insufficient bounds validation...

However, the A4 context, by recursively expanding the implementation details of every nested function call (e.g., error handling, string manipulation), ballooned to 7,313 characters. This inflation, containing a high density of repetitive and low-level technical details, appears to have saturated the model's cognitive window, leading to a processing failure.

A4 Prediction (Failed): No output from model

This demonstrates that there is a hard limit to the complexity of context an LLM can effectively handle, beyond which more information leads not just to worse results, but to no result at all.

7.1.2 Type II: Information Overload and the "Illusion of Safety". In many instances, even when the model produced an output, the A4 context led to an incorrect "not vulnerable" prediction. This is caused by "Information Overload," where an excess of low-level, non-security-critical details creates an "illusion of safety" that masks the actual vulnerability. This is clearly illustrated by the analysis of CVE-2017-12877 in `ReadMATImage`.

```
static Image *ReadMATImage(const ImageInfo *image_info,ExceptionInfo *exception)
{
  // ... [Extensive logic with function pointers, goto statements, and manual memory management] ...
  BImgBuff = (unsigned char *) AcquireQuantumMemory((size_t) (ldblk),sizeof(double));
  if (BImgBuff == NULL)
    ThrowReaderException(ResourceLimitError,"MemoryAllocationFailed");
  // ... [Main loop for reading all scanlines with complex conditions] ...
  RelinquishMagickMemory(BImgBuff);
  // ... [Further logic and cleanup] ...
```

```
  return (image);
}
```

Listing 3. Simplified source code for CVE-2017-12877, `ReadMATImage`

The A3 context provided a concise (3,433 characters) sequence of the main function's high-level API calls, exposing its complex and risky logic. Based on this, the model correctly identified multiple critical risks.

A3 Prediction (Correct): yes. reasoning: the code contains critical vulnerabilities, including an out-of-bounds read (risk of crashes/information disclosure) and stack overflow...

In contrast, the A4 context expanded to 6,494 characters, detailing the internal workings of every minor utility function, including numerous error-handling branches and pointer validation checks from deep within the call stack. The model's reasoning with the A4 context reveals the cognitive bias this introduced; it focused on the presence of these low-level safety checks.

A4 Prediction (Incorrect): no. reasoning: the code does not exhibit significant vulnerabilities. while pointers are constructed..., they are validated and used safely. inputs are checked for validity..., and error handling... prevents undefined behavior.

The noise, in this case, was the deluge of correct but irrelevant implementation details that created a misleading "technical safety illusion," causing the model to erroneously generalize the safety of low-level utilities to the flawed high-level logic.

7.1.3 Type III: Analysis Focus Drift Noise. The final category of noise causes a fundamental shift in the model's analytical objective. Instead of performing a security assessment to identify threats, the model is induced by the detailed A4 context to perform a code quality review.

The analysis of **CVE-2017-11732 in `dcinit`** provides a stark example of this "Focus Drift."

```
void dcinit()
{
  strsize=0;
  strmaxsize=DCSTRSIZE;
  dcstr=calloc(DCSTRSIZE,1);
  dcptr=dcstr;
}
```

Listing 4. Source code for CVE-2017-11732, `dcinit`

The A3 context was minimal ('calloc: called 1 times'). Starved of detail, the model relied on its knowledge of common vulnerability patterns and correctly inferred the high risk of a null pointer dereference from the missing check on 'calloc''s return value.

A3 Prediction (Correct): yes. reasoning: the code contains significant vulnerabilities due to: 1. missing null checks after allocation... subsequent dereferences... will cause null pointer dereferences...

The A4 context was only slightly more specific, naming the variable for the allocation size ('DCSTRSIZE'). This minor, concrete detail acted as an "anchor," shifting the model's focus from abstract threat patterns to the code itself. As the code follows a standard (albeit unsafe) pattern, the A4 model's reasoning pivoted entirely to a code quality assessment.

A4 Prediction (Incorrect): no. reasoning: the code snippet itself does not contain direct vulnerabilities... within the provided scope. the function... delegates to ... a standard pattern... without further context, the snippet itself does not exhibit significant vulnerabilities.

This focus drift, triggered by a small increase in implementation-specific detail, caused the model to miss the obvious high-level security flaw by redefining its task from threat detection to code standard evaluation.

7.1.4 Key Insights and Practical Implications. This case-by-case analysis provides several key insights into the role of context in LLM-based vulnerability detection:

- **The "More is Not Better" Principle**: There is an optimal level of abstraction for contextual information. Exceeding this level, as A4 often does, can introduce a cumulative noise effect that degrades or completely breaks the analysis.
- **Task-Information Alignment**: Vulnerability detection is fundamentally a task of high-level semantic and structural reasoning. The A3 context, by summarizing control flow and key API interactions, aligns well with this task. The A4 context, with its focus on low-level implementation details, is better aligned with a code review task, explaining the observed focus drift.
- **Model Sensitivity as a Factor**: The results indicate that different models have varying resilience to this noise. The DeepSeek models appear highly sensitive to low-level technical details, whereas GPT-4o's stronger information filtering capabilities allow it to maintain performance even with more verbose context.

These findings lead to several practical implications for future research and development:

- **For Dataset and Context Design:** This study highlights the need to control for information density. Instead of defaulting to the most detailed context available, future work should focus on identifying the optimal abstraction level that maintains task-information alignment for security analysis, filtering out irrelevant low-level implementation details that act as noise.
- **For Model Development:** Our results suggest a need for training techniques that enhance robustness to contextual noise. This could involve developing better information-filtering mechanisms within the model architecture or fine-tuning models specifically to maintain focus on high-level security threats without drifting towards general code quality assessment.
- **For Evaluation Methodologies:** We advocate for evaluating models across a spectrum of context granularities, not just on a single, fixed level of detail. This approach of "progressive complexity testing" is crucial for systematically identifying a model's sensitivity to noise and revealing its true performance profile under different conditions.

7.2 A Heuristic Strategy for Selecting API Abstraction and Prompts

Our extensive empirical evaluation in Section 5 reveals that the performance of Large Language Models (LLMs) in vulnerability detection is highly sensitive to the interplay between the model's intrinsic capabilities, the granularity of contextual information provided (i.e., the API abstraction level), and the guidance offered by the prompt strategy. In a practical software security pipeline, manually determining the optimal combination for a given scenario is a non-trivial challenge. Therefore, based on our findings, this section synthesizes a heuristic, rule-based strategy to provide actionable guidance for practitioners on selecting the most effective configuration for the PacVD framework.

7.2.1 Model-Driven Abstraction Level Selection. The first and most critical decision is to match the abstraction granularity to the architectural characteristics of the chosen LLM. Different models process and benefit from contextual information in distinct ways. Our results suggest a clear bifurcation based on whether the model is code-specialized or a general-purpose reasoner.

- For code-specialized models that are highly sensitive to code's structural and syntactic properties (e.g., CodeLLaMA-34b), we recommend using a medium-granularity abstraction, specifically API Level 2 (A2). This level provides concrete control-flow branch conditions without the additional statistical noise of higher levels.

- For powerful, general-purpose reasoning models (e.g., DeepSeek-R1, ChatGPT-4o), we recommend a higher-level abstraction, API Level 3 (A3). This level augments concrete branch information with API call frequency data, providing a richer semantic context for complex reasoning.

This rule is grounded in the principle of informational synergy. Code-specialized models like CodeLLaMA are pre-trained extensively on source code and have highly optimized internal representations for its structure. Providing them with the clean, structured data of concrete control-flow paths (A2) directly complements their strengths. However, adding more abstract statistical data (as in A3) can act as noise, potentially conflicting with their fine-tuned understanding of code semantics and leading to a performance decline. Conversely, general-purpose models excel at synthesizing diverse forms of information. The combination of structural path data and quantitative frequency data in A3 provides them with a richer, multi-faceted view of the program's behavior. This allows them to more effectively infer complex vulnerability patterns, such as resource management errors (e.g., an imbalance between malloc and free calls), which are not apparent from control flow alone. Our findings confirm that these models reach their peak performance when supplied with this richer contextual information.

7.2.2 Abstraction-Driven Prompt Strategy Selection. Once an appropriate abstraction level is chosen, the prompt strategy should be selected to match the richness and complexity of the information being presented to the model.

- When the provided context is sparse or low-level (i.e., w/o API or API Level 1), simpler and more direct strategies are preferable. These include the Basic Prompt (BP) or example-driven approaches like In-Context Learning (IC).
- When the context is rich and complex (i.e., API Level 2 through A4), more sophisticated, reasoning-oriented strategies like Chain-of-Thought (CoT) are highly effective at unlocking the model's analytical capabilities.

The effectiveness of a prompt strategy is contingent upon the informational foundation available to the model. With limited context, a complex prompt like CoT may force the model to reason over insufficient evidence, leading to error accumulation or hallucination. In such scenarios, a direct query (BP) focuses the model's resources, while providing concrete examples (IC) offers the clearest and most efficient path to understanding the task. As the API abstraction level increases, the provided context becomes dense and multi-faceted. This rich information provides the necessary building blocks for a structured, multi-step analysis. The CoT strategy acts as a cognitive scaffold, guiding the model to systematically deconstruct the problem, analyze the control flow and data patterns, and synthesize these pieces into a final, well-reasoned verdict. As our results show, the advantage of such sophisticated prompting strategies becomes most pronounced when the model has a rich informational context to operate on.

7.2.3 Vulnerability-Driven Abstraction Tuning. While the first two rules provide a robust baseline for general vulnerability detection, effectiveness can be further enhanced when conducting targeted searches for specific classes of vulnerabilities. Our findings in the RQ2 analysis (Section 5.2) show a clear correlation between vulnerability type and the optimal abstraction level required for its detection.

- For boundary-related memory errors (e.g., Out-of-bounds Write/CWE-787, Out-of-bounds Read/CWE-125), which often depend on local control-flow paths, a balanced abstraction level of API Level 2 (A2) or API Level 3 (A3) provides the most consistent and high-performing results.

- For resource management vulnerabilities (e.g., Double Free/CWE-415, Resource Leak, Uncontrolled Resource Consumption/CWE-400), which are non-local and state-dependent by nature, a higher level of abstraction is critical. We strongly recommend API Level 3 (A3) or API Level 4 (A4) to capture the necessary cross-procedural context.

This rule is rooted in the fundamental nature of different vulnerability classes. Boundary errors often arise from flawed logic within a specific execution path (e.g., a missing bounds check before a write operation). A2's concrete branch conditions and A3's added frequency data provide sufficient context for LLMs to identify these localized flaws. In contrast, resource management errors are defined by the lifecycle of a resource across potentially distant parts of the code. Detecting a Double Free or a resource leak requires tracking the state of a variable or memory location across multiple function calls. The higher abstraction levels are explicitly designed for this: A3's call frequency counts can reveal imbalances between allocation and deallocation APIs (e.g., malloc/free), while A4's variable tracking provides the direct data-flow relationships needed to trace a resource's state. Therefore, providing this richer, state-aware context is essential for accurately detecting this class of vulnerabilities.

7.3 Sensitivity to Mismatched Configurations

This section presents a focused ablation study to quantify the performance degradation from suboptimal configurations. The results underscore the necessity of our heuristic strategy by demonstrating that improper pairings of models, API abstractions, and prompts can lead to a near-total failure of the detection process.

7.3.1 Analysis of Mismatched Pairings. Our analysis reveals several critical failure modes, with all empirical data drawn from the comprehensive results in our experiments.

Case 1: Prompt Mismatch at the Optimal Abstraction Level. An ill-suited prompt can neutralize an otherwise optimal configuration. CodeLLaMA-34b achieves its peak F1 score of 66.67% at API Level 2 with a direct Basic Prompt. However, at the same abstraction level, a reasoning-focused Chain-of-Thought prompt causes performance to collapse to an F1 score of just 12.24%. This shows that for specialized models, direct instructions can be superior to complex reasoning prompts.

Case 2: Ineffectiveness of Complex Prompts Without Sufficient Context. Sophisticated prompts like Chain-of-Thought are powerful only when grounded in rich contextual data. Without any API context, CodeLLaMA-34b with a standard prompt (FSC) establishes a baseline F1 score of 56.25%. Attempting to use a Chain-of-Thought prompt in this context-poor environment results in a dismal F1 score of 8.33%, proving that such prompts require the structured information provided by our framework to function.

Case 3: Catastrophic Failure from Total Configuration Mismatch. The most severe performance degradation occurs when all components are misaligned. DeepSeek-V3 provides a stark example. In its optimal configuration (A2 + In-Context), it achieves a high F1 score of 68.63%. In a mismatched setup (A3 + Few-shot Random), its performance fails almost completely, yielding an F1 score of only 7.41%.

These critical cases are summarized in Table 10 for direct comparison. The data offers compelling evidence that LLM-based vulnerability detection is acutely sensitive to the quality of input context and the method of reasoning guidance. The dramatic performance drops observed in mismatched configurations are not minor fluctuations but fundamental failures. This high degree of sensitivity validates our core thesis: a "one-size-fits-all" approach is inadequate. Therefore, the heuristic

Table 10. Performance Impact of Mismatched Configurations

Model	Level	Prompt Strategy	F1 Score (%)	Note
CodeLLaMA-34b	A2	Basic Prompt	66.67	Optimal for Model
CodeLLaMA-34b	A2	Chain-of-Thought	12.24	**Prompt Mismatch**
CodeLLaMA-34b	w/o API	FSC	56.25	Baseline
CodeLLaMA-34b	w/o API	Chain-of-Thought	8.33	**Context Mismatch**
DeepSeek-V3	A2	In-Context	68.63	Optimal for Model
DeepSeek-V3	A3	Few-shot Random	7.41	**Total Mismatch**

selection strategy proposed before is an essential prerequisite for reliably deploying the PacVD framework in practice.

7.4 Practical Implications for Software Security

Our comprehensive evaluation of API abstraction levels for vulnerability detection through large language models yields several important implications and lessons for both practitioners and researchers:

Optimal API Abstraction Selection: Our findings suggest that A3 level abstraction (i.e., specific branches + call frequency) provides the best balance between information richness and noise for vulnerability detection. Organizations implementing LLM-based vulnerability detection should prioritize this abstraction level to maximize detection accuracy while minimizing computational overhead.

Model-Strategy Pairing: Different models exhibit varying levels of sensitivity to API abstraction. For resource-constrained environments, DeepSeek-R1 offers exceptional performance with minimal sensitivity to abstraction levels, making it ideal for general application. In contrast, specialized code models like CodeLLaMA-34b require more detailed API abstraction to reach optimal performance.

Prompt Engineering Guidelines: As API contextual information richness increases, more sophisticated reasoning-oriented prompting strategies become advantageous. Practitioners should match their prompting approaches to the available API abstraction level: Basic Prompts for no abstraction, In-context Learning for low abstraction (A1), and Chain-of-Thought techniques for higher abstraction levels (A2-A4).

Vulnerability-Specific Approaches: Different vulnerability types benefit from distinct abstraction levels. Memory boundary-related vulnerabilities can be effectively detected even with minimal API abstraction (A1), while resource management vulnerabilities require higher abstraction levels (A3-A4) for optimal detection. Security teams should consider tailoring their approach based on the vulnerability types of greatest concern.

Controllable Precision-Recall Trade-off for Deployment. Managing the False Positive Rate (FPR) is a critical challenge for practical deployment. Our evaluation on the processed PrimeVul dataset reveals that PacVD offers a controllable trade-off tailored to different operational needs through model selection:

- **Precision-Oriented Mode:** The code-specialized CodeLlama+BP+A3 configuration (Table 7) achieves a high Precision of 87.65%, corresponding to a moderate FPR of approximately 11.0%. This performance stems from the model's Code-Completion Objective, which prioritizes

syntactic and structural correctness, leading to a conservative decision boundary that filters out noise. This mode is ideal for automated pipelines where precision is paramount.

- **Recall-Oriented Mode:** Conversely, general-purpose models like ChatGPT+IC+A3 (Table 8) maximize Recall (97.81%) but incur a significantly higher FPR. Driven by an Instruction-Following Objective (RLHF), these models adopt an aggressive decision boundary that prioritizes associative reasoning to catch potential risks. This mode serves as a broad-coverage safety net for rigorous auditing.

Practitioners can thus dynamically select the underlying LLM to match their specific operational tolerance for false positives versus missed detections.

7.5 Future Research Directions and Challenges

Our research, while demonstrating the effectiveness of context-enhanced vulnerability detection, also illuminates several challenges that pave the way for future investigations. We structure these as follows:

7.5.1 Information Saturation and Abstraction Optimality. Our findings indicate that an "information saturation point" exists (around API Level A3), beyond which more granular detail provides diminishing or even negative returns. This suggests that for LLM-based code analysis, more information is not always better.

This phenomenon mirrors concepts like the "lost in the middle" problem in LLM research, where models struggle to reason over excessively long contexts. It also aligns with the classic trade-off between precision and complexity in Abstract Interpretation theory. While many studies focus on enriching input, few have formally investigated the upper bounds of useful information density.

We propose a rigorous investigation into this saturation effect across various software engineering tasks. Future work should aim to develop a theoretical model of the relationship between information density, model architecture, and reasoning performance. This could lead to principled guidelines for designing optimal, task-specific program abstractions.

7.5.2 Effectiveness of Multi-Modal Information Fusion. Our work successfully combines structured API abstractions with unstructured source code. However, vulnerabilities often have footprints across multiple information modalities that are not fully captured by our current approach.

While many studies leverage individual modalities—such as graph-based representations (e.g., Devign), data flow graphs, or natural language artifacts like commit logs and developer comments—these approaches are often siloed. The synergistic potential of fusing these heterogeneous data sources remains largely unexplored in the context of LLM-based detection.

In the future, a promising avenue is to explore multi-modal fusion models. Future work could design systems that synergistically integrate our API abstractions with other rich sources, such as Code Property Graphs (CPGs) and natural language documentation. This would enable the LLM to form a more holistic understanding of potential vulnerabilities.

7.5.3 Broader Application to SE Tasks. While this study focuses on vulnerability detection, the underlying principle of enhancing LLM reasoning with tailored program abstractions is potentially applicable to a wider range of software engineering tasks.

Many current LLM applications in software engineering, such as automated program repair, code summarization, and test generation, often operate on raw source code. This limits their ability to reason about deeper program semantics, sometimes resulting in plausible but incorrect outputs.

Future work should adapt this methodology to other domains. For instance, in automated program repair, an abstraction of program invariants or data flows could guide the LLM to generate more

robust and correct patches. Similarly, for bug triage, abstracting execution traces could help LLMs better understand and categorize bug reports.

7.5.4 Automating the Optimal Configuration. Our study manually explored the design space of models, API levels, and prompts to find effective combinations. In a practical, large-scale setting, manually selecting the optimal configuration for a given codebase or vulnerability type is infeasible.

This challenge is analogous to problems addressed by the AutoML and meta-learning communities, which focus on automating machine learning pipelines. However, applying these concepts to the structured design space of prompt engineering and program analysis-derived context is a novel problem.

Future research direction is to design an automated selector—a meta-model—that can dynamically choose the most suitable configuration. This could be a rule-based system that matches abstraction granularity to code complexity metrics (e.g., cyclomatic complexity) or a learning-based model trained to predict the best prompt and API-level combination for a given code snippet.

7.5.5 Heterogeneity of Programming Languages. The proposed API abstraction is tailored for C/C++, and its direct application to other languages is non-trivial due to fundamental differences in their execution models and semantics.

The long history of static and dynamic analysis has produced a wealth of language-specific tools (e.g., Soot for Java, Bandit for Python). This highlights the need for tailored analysis. The challenge lies in creating abstractions that are semantically equivalent across different memory models (e.g., manual management in C/C++, garbage collection in Java, ownership in Rust).

In the future, we propose the development of language-specific front-ends or semantic adapters. These components would be responsible for analyzing code in various languages (e.g., Python, Java, Rust) and generating a standardized set of API abstractions. This would preserve the core LLM reasoning engine while extending the applicability of our method across a diverse software ecosystem.

7.6 Threats to Validity

A potential threat is the generalizability of our results from the initial NVD-New dataset. To mitigate this, we conducted a comprehensive evaluation on the large-scale PRIMEVUL benchmark. The consistent advantages demonstrated by PacVD on this diverse and rigorous benchmark strongly suggest that our findings are not dataset-specific. We acknowledge, however, that our study focuses on C/C++ and its generalizability to other languages requires future investigation.

8 Related Work

8.1 Current Advances in Vulnerability Detection

Recent advances in vulnerability detection leverage deep learning and large language models (LLMs) to analyze complex code patterns, categorized into traditional machine learning methods, sequence-based learning, graph-based models, and LLM-based approaches [7, 67].

Traditional machine learning-based approaches rely on handcrafted features to identify vulnerabilities, but their limited adaptability and reliance on expert intervention hinder their effectiveness across diverse codebases [17, 33, 72]. Sequence-based deep learning methods treat code as token sequences, significantly improving detection accuracy by capturing syntactic and semantic information, though they struggle with fully representing the hierarchical structure of code [34, 60, 73]. Graph-based deep learning models enhance vulnerability detection by representing code as graphs, effectively capturing relationships and facilitating comprehensive structural analysis, thus improving accuracy and robustness [23, 61, 71]. LLM-Based Methods utilize pre-trained models for

vulnerability detection, demonstrating adaptability and effectiveness through fine-tuning and prompt engineering, significantly transforming the field [9, 28, 31, 50, 51, 64]. Recent advances in prompt-enhanced LLM methods highlight the potential of LLMs for identifying vulnerabilities through carefully designed prompts, improving detection capabilities, and addressing complex security challenges in code analysis [40, 63, 70]. Our approach follows a similar strategy but incorporates more contextual information extracted by program analysis.

In addition to detecting vulnerabilities at the function or file level, a significant and growing body of research has focused on more fine-grained vulnerability detection, aiming to localize vulnerable statements or lines of code. These methods provide more precise and actionable feedback for developers [16, 22, 26, 30, 55]. For example, the work by Li et al. [30] pioneered vulnerability detection with fine-grained interpretations. Building on this, recent approaches have employed state-of-the-art models for direct statement-level prediction. LineVul [22] utilizes a Transformer-based model for line-level vulnerability prediction, while LineVD [26] applies graph neural networks to identify vulnerable statements. Furthermore, VELVET [16] introduces a novel ensemble learning framework to automatically locate vulnerable statements. While our work, PacVD, focuses on enhancing function-level detection by leveraging inter-procedural context, we acknowledge that these fine-grained techniques represent a critical and complementary research direction aimed at improving the precision of vulnerability localization.

8.2 Contextual Representations in Vulnerability Detection

Recent studies in vulnerability detection highlight the importance of extracting rich contextual information from source code to enhance detection effectiveness. Various forms of static analysis have been foundational in capturing this context for neural models.

Control flow and data flow analysis have been widely adopted[11, 24, 27, 49]. The SySeVR framework [34], for instance, exemplifies this by using control and data dependency graphs to improve vulnerability prediction. Similarly, advancements such as the improved control flow graph method proposed by Zhou et al. have enhanced traditional representations for more accurate detection [68]. Beyond data and control flow, API call sequences and program dependence graphs (PDGs) have also been shown to play a critical role[20, 29, 30, 45]. The Devign model integrates Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and PDGs to capture complex code interdependencies [71], while models like Code2Vec learn vector representations from AST paths to better understand API call sequences[2]. Building on these methodologies, Zhang et al. proposed a framework that combines context slicing with multi-feature integration, improving detection accuracy by focusing on relevant code segments [66].

However, these context representation methods were primarily designed for traditional, supervised neural networks (e.g., GNNs, LSTMs) that require extensive labeled data for training. When the detection model shifts to a Large Language Model (LLM)—possessing powerful zero-shot or few-shot reasoning capabilities—these methods may not be optimal. For example, complex graph structures are difficult to translate directly into the sequential, text-based formats that LLMs are adept at processing. More importantly, prior work has not systematically investigated the crucial question of context granularity. Our work addresses this gap from a novel, LLM-centric perspective, exploring how "Primitive API Abstraction"—a representation method better suited to LLMs—can be used to optimize the design and delivery of context.

9 Conclusion

In this work, we address the challenges of vulnerability detection by leveraging primitive API abstractions and context-enhanced techniques with large language models (LLMs). Our proposed method, **PacVD**, combines API abstraction and prompt engineering to enhance LLM performance,

demonstrating that different abstraction levels and tailored prompt strategies significantly improve detection capabilities. Experimental results showed substantial improvements over baselines, highlighting the value of combining appropriate abstraction levels with effective prompt strategies to enhance vulnerability detection. These insights pave the way for more robust and interpretable software security solutions.

10 Data Availability

We open-source our datasets and experimental details to facilitate future research, which is available at: **https://github.com/DoeSEResearch/PacVD.git**

Acknowledgments

This work was supported by the National Key R&D Program of China No 2024YFB4506200 and National Natural Science Foundation of China under Grant No 62202026.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023). https://doi.org/10.48550/arXiv.2303.08774

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353

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

[4] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020). https://doi.org/10.48550/arXiv.2005.14165

[5] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48, 9 (2022), 3280–3296. https://doi.org/10.1109/TSE.2021.3087402

[6] Lei Chen and Raymond Ng. 2004. On the marriage of Lp-norms and edit distance. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 792–803. https://doi.org/10.5555/1316689.1316758

[7] Xiangping Chen, Xing Hu, Yuan Huang, He Jiang, Weixing Ji, Yanjie Jiang, Yanyan Jiang, Bo Liu, Hui Liu, Xiaochen Li, Xiaoli Lian, Guozhu Meng, Xin Peng, Hailong Sun, Lin Shi, Bo Wang, Chong Wang, Jiayi Wang, Tiantian Wang, Jifeng Xuan, Xin Xia, Yibiao Yang, Yixin Yang, Li Zhang, Yuming Zhou, and Lu Zhang. 2025. Deep learning-based software engineering: progress, challenges, and opportunities. *SCIENCE CHINA Information Sciences* 68, 1 (2025), 111102–. https://doi.org/10.1007/s11432-023-4127-5

[8] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses* (Hong Kong, China) *(RAID '23)*. Association for Computing Machinery, New York, NY, USA, 654–668. https://doi.org/10.1145/3607199.3607242

[9] Yujia Chen, Cuiyun Gao, Zezhou Yang, Hongyu Zhang, and Qing Liao. 2024. Bridge and Hint: Extending Pre-trained Language Models for Long-Range Code. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 274–286. https://doi.org/10.1145/3650212.3652127

[10] Zhe Chen, Chong Wang, Junqi Yan, Yulei Sui, and Jingling Xue. 2021. Runtime detection of memory errors with smart status. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 296–308. https://doi.org/10.1145/3460319.3464807

[11] Xiao Cheng, Haoyu Wang, Jiayi Hua, Miao Zhang, Guoai Xu, Li Yi, and Yulei Sui. 2019. Static Detection of Control-Flow-Related Vulnerabilities Using Graph Embedding. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. 41–50. https://doi.org/10.1109/ICECCS.2019.00012

[12] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. https://doi.org/10.48550/arXiv.2412.19437 arXiv:2412.19437 [cs.CL]

[13] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. https://doi.org/10.48550/arXiv.2501.12948 arXiv:2501.12948 [cs.CL]

[14] Xingjing Deng, Zhengyao Liu, Xitong Zhong, Shuo Hong, Yixin Yang, Xiang Gao, Xuhui Yan, and Hailong Sun. 2025. Code Property Graph Meets Typestate: A Scalable Framework to Behavioral Bug Detection. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 1–12. https://doi.org/10.1109/ICSME64153.2025.00019

[15] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability Detection with Code Language Models: How Far Are We?. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 1729–1741. https://doi.org/10.1109/ICSE55347.2025.00038

[16] Yangruibo Ding, Sahil Suneja, Yunhui Zheng, Jim Laredo, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. 2022. VELVET: a noVel Ensemble Learning approach to automatically locate VulnErable sTatements . In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 959–970. https://doi.org/10.1109/SANER53432.2022.00114

[17] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 60–71. https://doi.org/10.1109/ICSE.2019.00024

[18] Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. 2024. Generalization-Enhanced Code Vulnerability Detection via Multi-Task Instruction Fine-Tuning. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 10507–10521. https://doi.org/10.18653/v1/2024.findings-acl.625

[19] Xueying Du, Geng Zheng, Kaixin Wang, Jiayi Feng, Wentai Deng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2024. Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG. https://doi.org/10.48550/arXiv.2406.11147 arXiv:2406.11147 [cs.SE]

[20] Fenil Fadadu, Anand Handa, Nitesh Kumar, and Sandeep Kumar Shukla. 2019. Evading API Call Sequence Based Malware Classifiers. In *Information and Communications Security: 21st International Conference, ICICS 2019, Beijing, China, December 15–17, 2019, Revised Selected Papers* (Beijing, China). Springer-Verlag, Berlin, Heidelberg, 18–33. https://doi.org/10.1007/978-3-030-41579-2_2

[21] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 508–512. https://doi.org/10.1145/3379597.3387501

[22] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 608–620. https://doi.org/10.1145/3524842.3528452

[23] Angelo Furno, Nour-Eddin El Faouzi, Rajesh Sharma, Valerio Cammarota, and Eugenio Zimeo. 2018. A Graph-Based Framework for Real-Time Vulnerability Assessment of Road Networks. In *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*. 234–241. https://doi.org/10.1109/SMARTCOMP.2018.00096

[24] Mahmoud Ghorbanzadeh and Hamid Reza Shahriari. 2020. ANOVUL: Detection of logic vulnerabilities in annotated programs via data and control flow analysis. *IET Information Security* 14, 3 (2020), 352–364. https://doi.org/10.1049/iet-ifs.2018.5615

[25] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. 2024. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey. https://doi.org/10.48550/arXiv.2403.14608 arXiv:2403.14608 [cs.LG]

[26] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar. 2022. LineVD: statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 596–607. https://doi.org/10.1145/3524842.3527949

[27] Zhonghao Jiang, Weifeng Sun, Xiaoyan Gu, Jiaxin Wu, Tao Wen, Haibo Hu, and Meng Yan. 2024. DFEPT: Data Flow Embedding for Enhancing Pre-Trained Model Based Vulnerability Detection. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware* (Macau, China) *(Internetware '24)*. Association for Computing Machinery, New York, NY, USA, 95–104. https://doi.org/10.1145/3671016.3671388

[28] Hongping Li and Li Shan. 2023. LLM-based Vulnerability Detection. In *2023 International Conference on Human-Centered Cognitive Systems (HCCS)*. 1–4. https://doi.org/10.1109/HCCS59561.2023.10452613

[29] Qiang Li, Jinke Song, Dawei Tan, Haining Wang, and Jiqiang Liu. 2021. 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)*. 161–173. https://doi.org/10.1109/DSN48987.2021.00031

[30] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 292–303. https://doi.org/10.1145/3468264.3468597

[31] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. In *International Conference on Learning Representations*. https://doi.org/10.48550/arXiv.2405.17238

[32] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the Effectiveness of Function-Level Vulnerability Detectors for Inter-Procedural Vulnerabilities. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1935–1946. https://doi.org/10.1145/3597503.3639218

[33] Zhen Li, Deqing Zou, Jing Tang, Zhihao Zhang, Mingqian Sun, and Hai Jin. 2019. A Comparative Study of Deep Learning-Based Vulnerability Detection System. *IEEE Access* 7 (2019), 103184–103197. https://doi.org/10.1109/ACCESS.2019.2930578

[34] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2022), 2244–2258. https://doi.org/10.1109/TDSC.2021.3051525

[35] Qiheng Mao, Zhenhao Li, Xing Hu, Kui Liu, Xin Xia, and Jianling Sun. 2024. Towards Effectively Detecting and Explaining Vulnerabilities Using Large Language Models. *arXiv preprint arXiv:2406.09701* (2024). https://doi.org/10.48550/arXiv.2406.09701

[36] National Vulnerability Database. 2021. National Vulnerability Database. https://nvd.nist.gov/.
[37] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. ReGVD: revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 178–182. https://doi.org/10.1145/3510454.3516865
[38] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1565–1569. https://doi.org/10.1145/3468264.3473122
[39] OpenAI. 2024. GPT-4o System Card. https://doi.org/10.48550/arXiv.2410.21276 arXiv:2410.21276 [cs.CL]
[40] Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu. 2023. Software Vulnerability Detection using Large Language Models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 112–119. https://doi.org/10.1109/ISSREW60843.2023.00058
[41] Komang Rinartha and Wayan Suryasa. 2017. Comparative study for better result on query suggestion of article searching with MySQL pattern matching and Jaccard similarity. In *2017 5th International Conference on Cyber and IT Service Management (CITSM)*. 1–4. https://doi.org/10.1109/CITSM.2017.8089237
[42] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (April 2009), 333–389. https://doi.org/10.1561/1500000019
[43] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, et al. 2024. Code Llama: Open Foundation Models for Code. https://doi.org/10.48550/arXiv.2308.12950 arXiv:2308.12950 [cs.CL]
[44] Murray Shanahan, Kyle McDonell, and Laria Reynolds. 2023. Role play with large language models. *Nature* 623, 7987 (2023), 493–498. https://doi.org/10.1038/s41586-023-06647-8
[45] Madhu K. Shankarapani, Subbu Ramamoorthy, Ram S. Movva, and Srinivas Mukkamala. 2011. Malware detection using assembly and API call sequences. *J. Comput. Virol.* 7, 2 (May 2011), 107–119. https://doi.org/10.1007/s11416-010-0141-5
[46] Aleksei Shestov, Rodion Levichev, Ravil Mussabayev, Evgeny Maslov, Pavel Zadorozhny, Anton Cheshkov, Rustam Mussabayev, Alymzhan Toleu, Gulmira Tolegen, and Alexander Krassovitskiy. 2025. Finetuning Large Language Models for Vulnerability Detection. *IEEE Access* 13 (2025), 38889–38900. https://doi.org/10.1109/ACCESS.2025.3546700
[47] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 693–706. https://doi.org/10.1145/3192366.3192418
[48] Yahui Song, Xiang Gao, Wenhua Li, Wei-Ngan Chin, and Abhik Roychoudhury. 2024. ProveNFix: Temporal Property-Guided Program Repair. *Proc. ACM Softw. Eng.* 1, FSE, Article 11 (July 2024), 23 pages. https://doi.org/10.1145/3643737
[49] Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024. Dataflow Analysis-Inspired Deep Learning for Efficient Vulnerability Detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 16, 13 pages. https://doi.org/10.1145/3597503.3623345
[50] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T. Barr, and Wei Le. 2024. A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection. *CoRR* abs/2403.17218 (2024). https://doi.org/10.48550/arXiv.2403.17218
[51] Jingyi Su and Yan Wu. 2023. Optimizing Pre-trained Language Models for Efficient Vulnerability Detection in Code Snippets. In *2023 9th International Conference on Computer and Communications (ICCC)*. 2139–2143. https://doi.org/10.1109/ICCC59590.2023.10507456
[52] Simeng Sun, Yang Liu, Dan Iter, Chenguang Zhu, and Mohit Iyyer. 2023. How does in-context learning help prompt tuning? *arXiv preprint arXiv:2302.11521* (2023). https://doi.org/10.48550/arXiv.2302.11521
[53] The Joern Team. 2024. Joern. Online. Available: https://joern.io.
[54] The MITRE Corporation. 2021. Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.
[55] Wentong Tian, Yuanzhang Lin, Xiang Gao, and Hailong Sun. 2025. Enhanced Vulnerability Localization: Harmonizing Task-Specific Tuning and General LLM Prompting . In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, Los Alamitos, CA, USA, 110–122. https://doi.org/10.1109/ICSME64153.2025.00020
[56] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023). https://doi.org/10.48550/arXiv.2302.13971
[57] Ziliang Wang, Ge Li, Jia Li, Jia Li, Meng Yan, Yingfei Xiong, and Zhi Jin. 2025. M2CVD: Enhancing Vulnerability Understanding through Multi-Model Collaboration for Code Vulnerability Detection. *ACM Trans. Softw. Eng. Methodol.* (Oct. 2025). https://doi.org/10.1145/3771923 Just Accepted.

[58] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages. https://doi.org/10.5555/3600270.3602070

[59] Xin-Cheng Wen, Xinchen Wang, Yujia Chen, Ruida Hu, David Lo, and Cuiyun Gao. 2024. Vuleval: Towards repository-level evaluation of software vulnerability detection. *arXiv preprint arXiv:2404.15596* (2024). https://doi.org/10.48550/arXiv.2404.15596

[60] Bolun Wu and Futai Zou. 2022. Code vulnerability detection based on deep sequence and graph models: A survey. *Security and Communication Networks* 2022, 1 (2022), 1176898. https://doi.org/10.1155/2022/1176898

[61] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[62] Yixin Yang, Ming Wen, Xiang Gao, Yuting Zhang, and Hailong Sun. 2024. Reducing False Positives of Static Bug Detectors Through Code Representation Learning. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 681–692. https://doi.org/10.1109/SANER60148.2024.00075

[63] Yanjing Yang, Xin Zhou, Runfeng Mao, Jinwei Xu, Lanxin Yang, Yu Zhang, Haifeng Shen, and He Zhang. 2025. DLAP: A Deep Learning Augmented Large Language Model Prompting framework for software vulnerability detection. *J. Syst. Softw.* 219, C (Jan. 2025), 15 pages. https://doi.org/10.1016/j.jss.2024.112234

[64] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2024. Prompt-Enhanced Software Vulnerability Detection Using ChatGPT. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 276–277. https://doi.org/10.1145/3639478.3643065

[65] Jie Zhang, Wei Ma, Qiang Hu, Shangqing Liu, Xiaofei Xie, Yves Le Traon, and Yang Liu. 2023. A Black-Box Attack on Code Models via Representation Nearest Neighbor Search. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 9706–9716. https://doi.org/10.18653/v1/2023.findings-emnlp.649

[66] Yulin Zhang, Yong Hu, and Xiao Chen. 2024. Context and Multi-Features-Based Vulnerability Detection: A Vulnerability Detection Frame Based on Context Slicing and Multi-Features. *Sensors* 24, 5 (2024). https://doi.org/10.3390/s24051351

[67] Yuting Zhang, Jiahao Zhu, Yixin Yang, Ming Wen, and Hai Jin. 2023. Comparing the Performance of Different Code Representations for Learning-based Vulnerability Detection. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware* (Hangzhou, China) *(Internetware '23)*. Association for Computing Machinery, New York, NY, USA, 174–184. https://doi.org/10.1145/3609437.3609464

[68] Minmin Zhou, Jinfu Chen, Yisong Liu, Hilary Ackah-Arthur, Shujie Chen, Qingchen Zhang, and Zhifeng Zeng. 2019. A method for software vulnerability detection based on improved control flow graph. *Wuhan University Journal of Natural Sciences* 24, 2 (2019), 149–160. https://doi.org/10.1007/s11859-019-1380-z

[69] Xin Zhou, Duc-Manh Tran, Thanh Le-Cong, Ting Zhang, Ivana Clairine Irsan, Joshua Sumarlin, Bach Le, and David Lo. 2024. Comparison of static application security testing tools and large language models for repo-level vulnerability detection. *arXiv preprint arXiv:2407.16235* (2024). https://doi.org/10.48550/arXiv.2407.16235

[70] Xin Zhou, Ting Zhang, and David Lo. 2024. 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* (Lisbon, Portugal) *(ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 47–51. https://doi.org/10.1145/3639476.3639762

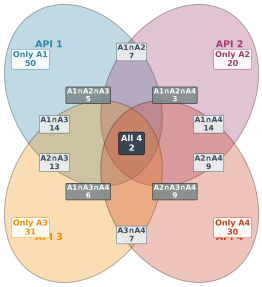
[71] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019). https://doi.org/10.5555/3454287.3455202

[72] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *Proceedings of the 29th International Conference on Software Engineering Workshops (ICSEW '07)*. IEEE Computer Society, USA, 76. https://doi.org/10.5555/1260984.1261267

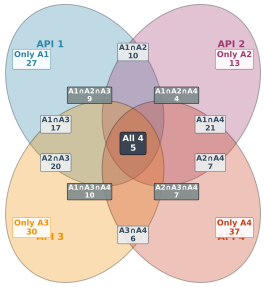
[73] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2021. VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2021), 2224–2236. https://doi.org/10.1109/TDSC.2019.2942930

A Venn Diagrams for All Prompting Strategies with DeepSeek-R1

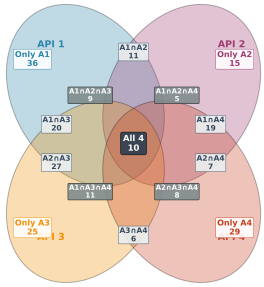
The following figures 8 provide the detailed Venn diagrams for all prompting strategies with the DeepSeek-R1 model, visualizing the overlap and unique detections of true positives across the four API abstraction levels.



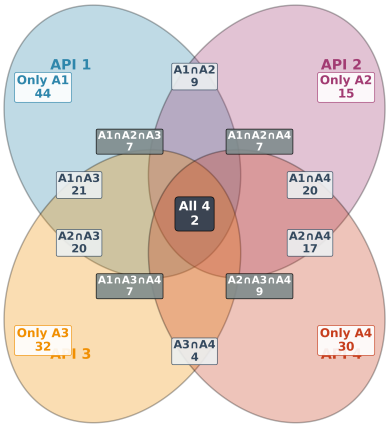
(a) Basic Prompt Strategy.



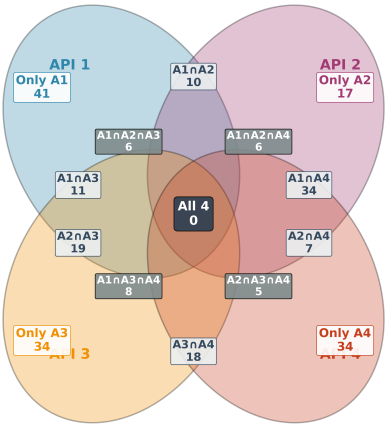
(b) Role-Play Prompt Strategy.



(c) In-Context Learning Strategy.



(d) Few-Shot (Random) Strategy.



(e) Few-Shot (Contrastive) Strategy.

Fig. 8. Venn Diagrams of Detection Overlap Across API Abstraction Levels for DeepSeek-R1 with various prompting strategies. Each subfigure (a-e) corresponds to a different strategy, illustrating the number of unique and shared vulnerabilities detected.

B Security-Sensitive APIs and Corresponding Vulnerability Types

Table 11. Primitive APIs and Corresponding Vulnerability Types

Vulnerability Category	Vulnerability Type (CWE)	Related Primitive APIs / Keywords
Memory & Resource Management	Memory Leak (CWE-401)	`malloc, calloc, realloc, free`
	Double Free (CWE-415)	`malloc, calloc, realloc, free, delete`
	Use After Free (CWE-416)	`malloc, calloc, realloc, free`
	NULL Pointer Dereference (CWE-476)	`malloc, calloc, realloc, free, fopen, NULL, localtime`
	Improper Resource Shutdown (CWE-404)	`open, fopen, close, fclose, malloc, free`
	Missing Release of Resource (CWE-772)	`open, fopen, close, fclose, socket, connect`
	Incomplete Cleanup (CWE-459)	`open, fopen, close, fclose, malloc, free`
	Missing Release of File Descriptor (CWE-775)	`open, fopen, close, fclose, socket`
	Improper Initialization (CWE-665)	`malloc, calloc, fopen`
	Use of Uninitialized Resource (CWE-908)	`malloc, calloc, fopen`
	Use of Uninitialized Variable (CWE-457)	`malloc, calloc, fopen`
Buffer & String Handling	Buffer Overflow (CWE-119)	`strcpy, strncpy, strcat, sprintf, memcpy`
	Classic Buffer Overflow (CWE-120)	`strcpy, strncpy, gets, sprintf, scanf`
	Out-of-bounds Read (CWE-125)	`memcpy, strcpy, strncpy, read, fread`
	Buffer Over-read (CWE-126)	`strlen, strncpy, memcmp`
	Out-of-bounds Write (CWE-787)	`strcpy, strcat, sprintf, memcpy`
	Heap-based Buffer Overflow (CWE-122)	`strcpy, strcat, sprintf`
	Stack-based Buffer Overflow (CWE-121)	`strcpy, strcat, sprintf`
	Off-by-one Error (CWE-193)	`malloc, calloc, realloc, []`
Integer & Arithmetic Issues	Integer Overflow (CWE-190)	`malloc, calloc, realloc`
	Integer Underflow (CWE-191)	`malloc, calloc`
	Divide By Zero (CWE-369)	`/, %, div`
	Unexpected Sign Extension (CWE-194)	`malloc, calloc, realloc`
	Incorrect Numeric Conversion (CWE-681)	`==, !=, =`
Input Validation & Injection	Improper Input Validation (CWE-20)	`scanf, gets, fgets, read, recv`
	Path Traversal (CWE-22)	`open, fopen, read, write`
	External Control of File Name (CWE-73)	`open, fopen, system, exec`
	Command Injection (CWE-74, CWE-78)	`system, exec,popen`
	Cross-site Scripting (CWE-79)	`printf, fprintf, sprintf`
	Uncontrolled Format String (CWE-134)	`printf, fprintf, sprintf, snprintf`
Concurrency & File Operations	Race Condition (CWE-362, CWE-366)	`open, fopen, write, read`
	Time-of-check Time-of-use (CWE-367)	`open, fopen, stat, access`
	Unrestricted Upload of File (CWE-434)	`fopen, fwrite, write`
	Unchecked Return Value (CWE-253)	`return, if`
Cryptographic & Logic Issues	Use of Broken Crypto Algorithm (CWE-327)	`rand, srand, random`
	Use of Insufficiently Random Values (CWE-330)	`rand, srand, random`
	Improper Certificate Validation (CWE-295)	`ssl, tls, cert`
	Insufficient Verification of Data (CWE-345)	`verify, check, validate`
	Loop with Unreachable Exit (CWE-835)	`while, for, do`
	Reachable Assertion (CWE-617)	`assert`
	Dead Code (CWE-561)	`return, exit`