POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

White-box Identification of Unsafe Inter-Procedural Interactions using Large Language Models

Tesi di Laurea Magistrale in
COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA INFORMATICA

Author: **Shodai Fujimoto**

Student ID: 242761
Advisor: Prof. Michele Carminati
Co-advisors: Francesco Panebianco
Academic Year: 2024-25

Abstract

Software vulnerabilities are a critical issue in modern development. Directed Greybox Fuzzing (DGF) is a powerful testing technique to find these flaws, but it faces a major bottleneck. It requires precise targets to work effectively. Currently, identifying these targets relies on manual analysis by security experts, which is slow and difficult to scale. While Large Language Models (LLMs) are used for code analysis, existing methods often look at functions in isolation and fail to detect complex bugs that span multiple functions (inter-procedural vulnerabilities).

We propose a method for identifying critical locations in source code where unsafe interactions could occur using LLMs. The goal is to improve the efficiency of selecting analysis targets without requiring the entire codebase as input. Our approach introduces a representation we call *Capability Information*, which explains functional characteristics of each function. This information focuses on the transformation of input data into output. This enables the detection of unsafe interactions, where a parent function misuses a child function.

To measure how well our framework identifies critical locations in C source code, we conducted experiments on two datasets. The first dataset is the FormAI dataset. This dataset includes AI-generated code with mainly intra-procedural vulnerabilities. The second dataset is the InterPVD dataset, which contains real-world inter-procedural vulnerabilities. The results indicate that, for code which includes simple intra-procedural vulnerabilities, our method achieves performance comparable to the baseline. In complex cases within InterPVD (CVE-2014-9728 and CVE-2013-0862), we demonstrate that our method can identify inter-procedural vulnerabilities that the baseline failed to detect.

This study demonstrates that, by leveraging LLMs not merely as classifiers but as reasoning agents that understand and propagate semantic information across functions, it is possible to generate target information to aid experts.

Keywords: Vulnerability Detection, LLM, Inter-Procedural Vulnerability, Directed Greybox Fuzzing

Abstract in lingua italiana

Le vulnerabilità software rappresentano un problema critico nello sviluppo moderno. Il Directed Greybox Fuzzing (DGF) è una potente tecnica di testing per individuare tali difetti, ma presenta un importante collo di bottiglia: per funzionare in modo efficace richiede target precisi. Attualmente, l'identificazione di questi target si basa su un'analisi manuale condotta da esperti di sicurezza, un processo lento e difficile da scalare. Sebbene i Large Language Models (LLM) vengano utilizzati per l'analisi del codice, i metodi esistenti esaminano spesso le funzioni in modo isolato e non riescono a rilevare bug complessi che si estendono su più funzioni (vulnerabilità inter-procedurali).

In questo articolo proponiamo un metodo per identificare le posizioni critiche nel codice sorgente in cui potrebbero verificarsi interazioni non sicure, utilizzando gli LLM con l'obiettivo di migliorare l'efficienza della selezione dei target per il DGF. Il nostro approccio consente all'LLM di comprendere le relazioni tra le funzioni. Il concetto chiave è la *Capability Information*. Essa permette di rilevare interazioni non sicure, in cui una funzione padre utilizza in modo improprio una funzione figlia, senza richiedere l'intero codebase come input.

Per valutare l'efficacia del metodo proposto, abbiamo condotto esperimenti utilizzando il dataset FormAI, che include codice sintetico, e il dataset InterPVD, che contiene vulnerabilità inter-procedurali reali. I risultati sperimentali mostrano che, per vulnerabilità intra-procedurali semplici, il nostro metodo mantiene prestazioni comparabili alla baseline. Inoltre, nei casi complessi presenti in InterPVD (ad esempio, CVE-2014-9728 e CVE-2013-0862), il nostro metodo è in grado di identificare correttamente vulnerabilità inter-procedurali che la baseline non è riuscita a rilevare.

Questo studio dimostra che, sfruttando gli LLM non solo come classificatori ma come agenti capaci di comprendere e propagare informazioni semantiche tra funzioni, è possibile generare informazioni sui target per aiutare l'identificazione successiva da parte di un esperto.

Parole chiave: Rilevamento delle vulnerabilità, LLM, Vulnerabilità inter-procedurale

Contents

Introduction

In recent years, the scale and complexity of software have led to a continuous annual increase in the number of reported vulnerabilities. The Common Vulnerabilities and Exposures (CVE) database [18], managed by the National Institute of Standards and Technology (NIST) [20], records thousands of new vulnerabilities each month, creating an urgent need for the establishment of efficient detection methodologies. Conventionally, vulnerability identification has relied heavily on manual analysis by experts utilizing reverse engineering and debuggers; however, this approach necessitates advanced expertise and entails significant time costs [6]. While large organizations may possess specialized security teams to address these issues, small to medium sized organizations and projects with short development cycles require low-cost, automated detection methods.

Dynamic analysis techniques, such as fuzzing, are widely employed for automated vulnerability detection [17]. Although these methods demonstrate a certain degree of effectiveness, generating inputs that satisfy complex path constraints often remains challenging, resulting in difficulty reaching deep code paths [5]. Directed Greybox Fuzzing [2], which targets specific locations, mitigates this issue; however, it requires the prior selection of potentially vulnerable target locations, and this selection process itself has become a new bottleneck [32]. Therefore, identifying these specific locations constitutes a critical initial step to enable effective dynamic analysis.

Potentially, static analysis utilizing Large Language Models (LLMs) could automate this target identification [25, 28, 34]. However, most existing studies on LLM-based vulnerability detection analyze only single functions in isolation, typically limiting their output to a binary classification of whether the function contains a vulnerability. Such binary outputs are insufficient for guiding Directed Greybox Fuzzing, which requires precise location information. Moreover, real-world software operates through the interaction of multiple functions; consequently, vulnerabilities arising from inter-functional dependencies cannot be detected through single-function analysis alone. Furthermore, technical constraints such as token limitations prevent the input of the entire source code into an LLM at once, leading to potential omissions in analysis.

In this paper, we propose a method to identify critical locations in source code where unsafe interactions could be located within source code using LLMs not as a standalone detector, but as a preprocessing step to optimize downstream security tasks, such as manual code review or automated testing like Directed Greybox Fuzzing (DGF). We specifically focus on detecting potential vulnerabilities across C source code composed of multiple functions and generating detailed explanations for them. This approach enables the identification of necessary targets for Directed Greybox Fuzzing without relying on highly skilled security experts.

Due to the inherent token limitations of Large Language Models (LLMs), vulnerability detection typically necessitates processing source code on a per-function basis. To implement this, we utilize Clang [15] to construct a call graph and extract the source code for individual functions. However, analyzing functions in isolation hinders the detection of inter-procedural vulnerabilities, as the context of interactions between functions is lost.

To address this issue, we introduce "Capability Information", a summary of the callee (child) function, which is injected into the prompt as context during the analysis of the caller (parent) function. Specifically, Capability Information details the inputs and outputs of the child function, as well as how input variables are processed and manipulated internally. This allows the LLM to comprehend the handling of arguments within child functions during the parent function's analysis, thereby facilitating the effective detection of inter-procedural vulnerabilities. To ensure the availability of this information, the LLM-based potential vulnerability detection is performed in a bottom-up manner based on the call graph.

To measure how well our framework identifies critical locations in C source code, we conducted experiments on two datasets. The first dataset is FormAI dataset. This dataset includes AI-generated code with mainly intra-procedural vulnerabilities. We utilized a randomly selected subset of 100 samples from this dataset to evaluate baseline accuracy in pinpointing vulnerable lines within AI-generated code. The second dataset is the InterPVD dataset, which contains real-world inter-procedural vulnerabilities. We use this dataset to validate the system's efficacy in handling complex dependency chains. We utilized 44 real-world instances from this dataset that explicitly involve multi-function interactions.

We evaluated the semantic accuracy of the model's explanations using an *LLM-as-a-Judge* approach [4, 13]. When analyzing the *FormAI* dataset, we evaluated whether the output correctly identified the specific error type and the vulnerable code logic defined in the metadata. When analyzing the *InterPVD* dataset, we implemented a more rigorous

multi-faceted verification, comparing the results against the official CVE description, the CWE [19] classification, the exact vulnerability-triggered statement and critical variable.

We summarize our contributions as follows:

- We proposed a method for extracting and summarizing the inputs, outputs, and internal data manipulations and behaviors of callee functions as "Capability Information" using large language models (LLMs). This approach enables the abstraction of implementation details while efficiently representing only the semantics essential for security analysis, making it possible to utilize this information as contextual input for analyzing other functions.
- We proposed a method for detecting potential vulnerabilities by integrating the extracted Capability Information with the source code of the caller function and performing a bottom-up analysis along the call graph. This approach demonstrates that unsafe inter-procedural interactions arising from inter-function dependencies and inconsistencies can be effectively identified without directly adding the callee function's source code to the prompt.

1 | Background

This chapter provides the necessary theoretical foundation and technical context to understand the proposed approach.

1.1. Software Vulnerabilities

Software vulnerabilities refer to unintended flaws in software systems that may allow attackers to compromise confidentiality, integrity, or availability. These weaknesses can arise during design, implementation, or deployment phases. To analyze and categorize such issues in a systematic manner, prior work has introduced several standardized classification schemes that facilitate consistent reporting and comparison.

Common Vulnerabilities and Exposures (CVE) [18] is a standardized, publicly available dictionary of disclosed information security vulnerabilities and exposures. The CVE program is operated by the MITRE Corporation and provides a common naming scheme that assigns a unique identifier to each vulnerability. This standardized identifier enables consistent reference across security advisories, vulnerability databases, and research publications. Each CVE entry includes a brief description of the reported vulnerability and references to related technical information. By providing a common identifier for each vulnerability, CVE helps organizations manage vulnerabilities more systematically, prioritize risks, and share information effectively.

CVE records focus on concrete vulnerability reports, whereas the Common Weakness Enumeration (CWE) [19] captures broader classes of underlying weaknesses. By abstracting recurring design, implementation, and configuration errors, CWE highlights patterns that transcend individual cases. Such generalization is beneficial for learning-based detection methods, which rely on shared features across multiple vulnerability instances.

Independently of the CWE taxonomy, vulnerabilities can also be classified based on the structural scope of the control and data flow required to trigger them. Intra-procedural vulnerabilities are confined within the scope of a single function, where the source of the malicious input, the propagation path, and the sink all reside within the same function

boundary. Conversely, inter-procedural vulnerabilities span multiple functions and are the primary focus of this research. Typically, a Source enters in a Caller function and propagates through arguments to reach a Sink in a Callee function. Since the Callee itself might be safe in isolation but vulnerable when called with specific arguments, detecting these flaws requires sophisticated analysis of data flow across function boundaries, a task that is significantly more computationally intensive.

1.2. Code Representation and Parsing Infrastructure

Converting raw source code into a structured representation is important for static analysis, as we must capture both syntax and architectural relationships. For this, our system utilizes the Clang compiler infrastructure to generate Abstract Syntax Tree (AST) and Call Graph (CG).

Traditional compilers typically abstract away most of their internal mechanisms. However, Clang exposes a modular architecture that allows direct access to its parsing structures. At the function level, Clang constructs an abstract syntax tree that represents the internal structure of each function. This tree captures control flow constructs as well as variable level elements within a hierarchical representation.

Our analysis extends beyond single functions, so a program level view is also required. For this purpose, we construct a call graph to represent relationships among functions. In this graph, functions correspond to nodes and function calls correspond to directed edges, enabling us to follow the execution flow throughout the program.

In this study, we rely on Clang and its abstract syntax tree traversal interface to identify function boundaries and extract the original source code of each function as an independent unit. The call graph determines the order of analysis. By resolving caller and callee relationships, we apply a bottom up strategy in which callee functions are processed and summarized by the large language model before their callers are analyzed.

1.3. Directed Greybox Fuzzing (DGF)

Fuzzing is a dynamic software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program to monitor for crashes or memory leaks. Among its variations, Greybox fuzzing occupies a middle ground between white-box (full source analysis) and blackbox (no internal knowledge) approaches. It utilizes lightweight instrumentation to gather code coverage information, such as execution paths, to guide input generation with the primary goal of maximizing overall code coverage.

Directed Greybox Fuzzing (DGF) [2] is a variant of fuzzing that aims to steer test case generation toward specific target locations in the program, for example functions suspected to contain vulnerabilities or recently modified code segments.

Despite this advantage, applying DGF in practice can be challenging. The effectiveness of the technique largely depends on how well the target locations are chosen. Unlike conventional fuzzing approaches that explore the program more broadly, DGF requires developers to manually specify functions or code regions that are considered security relevant. Identifying such locations often demands prior knowledge of the program logic and potential weakness patterns.

In this work, we seek to reduce this manual effort by using large language models to automatically identify locations where unsafe interactions may occur.

2 | Motivation

2.1. Problem Statement

Directed Fuzzing has emerged as a powerful technique designed to overcome the limitations of traditional fuzzing, particularly its difficulty in reaching deep code paths [5]. However, its practical application necessitates the automation of selecting specific target locations for analysis. While static analysis approaches utilizing Large Language Models (LLMs) have recently garnered attention as a promising solution for such automation, simply applying existing LLM-based methods poses significant challenges. In this study, we define the technical barriers hindering the automation of target selection through the following three perspectives:

1. **Dependence on Domain Expertise in Target Selection**
   To execute Directed Fuzzing effectively, it is essential to identify regions within complex program structures where vulnerabilities controllable by input values are likely to reside. However, target designation in conventional directed fuzzing still relies heavily on manual source code reviews by security engineers or professional judgment based on static analysis tool outputs. This reliance on human resources constitutes a major bottleneck for automated vulnerability detection in rapid development cycles.

2. **Limitations in Granularity and Precision of LLM-based Detection**
   Most existing studies on vulnerability detection using Large Language Models (LLMs) focus on binary classification (presence or absence of a vulnerability) at the function level. However, guiding directed fuzzing requires precise location information and context, such as "which specific line in which function is vulnerable and why." Existing standalone LLM-based detection models do not yet provide analysis results specific enough to serve as inputs for dynamic analysis, failing to meet the requirements for an automated target selection tool.

3. **Conflict Between Inter-functional Dependencies and Token Constraints**
   Vulnerabilities in real-world, large-scale software are rarely contained within a sin-

gle function; they often arise from data flows and dependencies spanning multiple functions. Conversely, LLMs are subject to strict input token limitations, making it impossible to include an entire large-scale codebase within the context. This constraint prevents LLMs from fully comprehending complex logical structures across functions, which in turn makes it difficult to identify the "deep-seated vulnerabilities" that dynamic analysis aims to uncover.

2.2. State of the art

2.2.1. LLM-based Vulnerability Detection

Shestov et al. [27] adapted WizardCoder [16] to detect vulnerabilities in Java source code by fine-tuning. They proved that training on a balanced dataset vastly improves upon zero-shot baselines, reaching state-of-the-art accuracy for function-level detection but a critical limitation remains. Their approach formulates vulnerability detection strictly as a binary classification task, lacking the granularity to pinpoint specific line numbers or provide semantic root cause explanations.

Zhang et al. [33] conducted a comprehensive empirical study benchmarking LLMs against Small Language Models (SLMs) and Static Application Security Testing (SAST) tools across Python, Java, and JavaScript. Their experiments revealed that while fine-tuning proves to be the optimal strategy when sufficient and balanced training data is available, few-shot learning is more effective in data-limited scenarios. Furthermore, they highlighted that while data downsampling can mitigate issues with class imbalance, simple majority voting ensemble methods fail to significantly improve F1 scores, concluding that Software Vulnerability Detection (SVD) remains a fundamentally challenging task.

Panebianco et al. [25] examine the current readiness of large language models for automated vulnerability detection. Their study compares several quantized zero-shot models, including Mistral [10] and CodeQwen [1], with fine-tuned approaches such as PDBERT [14]. The results reveal weaknesses on both sides. The zero-shot models achieved low precision, in many cases below 0.46. Although the fine-tuned models reported strong benchmark performance, their behavior did not generalize well. In particular, they failed to detect vulnerabilities in curated examples that differed from the training distribution.

The authors argue that smaller models often rely on superficial lexical cues rather than deeper semantic reasoning. As a result, they suggest that these models should not be used as standalone vulnerability detectors without human oversight.

2.2.2. Inter-procedural Vulnerability Detection

Li et al. [12] conducted the first extensive empirical study on the effectiveness of function-level vulnerability detectors against inter-procedural vulnerabilities. They constructed a specific dataset, InterPVD, and demonstrated that state-of-the-art models suffer a drastic performance degradation when vulnerabilities span across multiple functions, highlighting the fundamental inability of isolated function analysis to capture cross-function data and control dependencies.

Wu et al. [30] addressed the critical limitation of single-function analysis by proposing VulnSC, a framework that explicitly captures inter-procedural semantics. They utilize LLMs to generate concise natural language summaries of callee functions (child functions) and append them as comments to the caller (parent function), thereby enhancing the context for downstream classification models like CodeBERT [7] and LineVul [8]. While VulnSC demonstrates that injecting callee summaries significantly improves the accuracy of binary vulnerability classification (determining if a function is vulnerable), it lacks the granularity to pinpoint the **specific line numbers** or provide **detailed root cause explanations**. Obtaining such precise "Target Location Information" is essential for automating target selection for Directed Greybox Fuzzing, which is the core contribution of our proposed method.

2.2.3. Synergy of LLMs and Fuzzing

Xu et al. [31] introduced HGFuzzer, a framework that integrates LLMs into Directed Greybox Fuzzing to address path explosion and exploitation randomness. They employ LLMs to analyze call chains and generate targeted harnesses, reachable initial seeds, and custom mutators, thereby improving the efficiency of reaching specific program states. However, while HGFuzzer effectively optimizes the process of reaching a target, it relies on pre-defined targets derived from existing CVEs or manual expert review, rather than automating the initial identification of these vulnerable locations from scratch. In contrast, our work focuses on the "Target Acquisition" phase, utilizing Capability Information to automatically pinpoint deep-seated vulnerabilities across function boundaries to serve as inputs for such directed fuzzing campaigns.

2.3. Goals and Challenges

Based on the technical barriers mentioned in the previous section, the primary goal of this research is to establish a fully automated methodology for selecting directed fuzzing

targets without relying on domain experts. To realize this objective, we must address specific technical challenges related to the precision of LLM analysis and the handling of large-scale code contexts. This study aims to achieve the following two specific goals:

- **Goal 1: Achieving Fine-Grained Potential Vulnerability Detection**
  To serve as a valid input for directed fuzzing, the analysis must go beyond the binary classification typical of existing LLM-based approaches. Our goal is to develop a mechanism that enables the LLM to output precise "Target Location Information," specifically identifying the vulnerable file, function, and exact line number, along with a semantic explanation of the root cause. The challenge here lies in prompt engineering and context structuring that guides the LLM to perform reasoning grounded in specific code syntax rather than abstract summarization.
- **Goal 2: Reconciling Inter-functional Analysis with Token Constraints**
  The most critical challenge is to detect vulnerabilities arising from complex inter-functional dependencies (e.g., a child function mishandling a buffer passed by a parent) while adhering to the strict token limits of LLMs. Our goal is to establish a method that can propagate semantic information across function boundaries without feeding the entire raw source code. This requires formulating a novel representation—which we define as "Capability Information"—that compresses the functional characteristics of callee functions into a compact format, thereby allowing the LLM to comprehend the global context necessary for accurate detection within a local analysis window.

3 | Approach

3.1. Approach Overview

In this paper, we propose a system that utilizes Large Language Models (LLMs) to identify critical locations in source code where unsafe interactions could occur by considering inter-procedural dependencies across the entire source code, with the aim of efficiently identifying target locations for Directed Greybox Fuzzing. As shown in Fig. 3.1, the proposed system consists of three main phases: (1) structural analysis of source code and function extraction, (2) extraction of Capability Information, and (3) identification of critical interactions. Our approach enables the detection of vulnerabilities spanning function boundaries by propagating information from callee functions (children) to caller functions (parents), rather than limiting analysis to single functions in isolation.

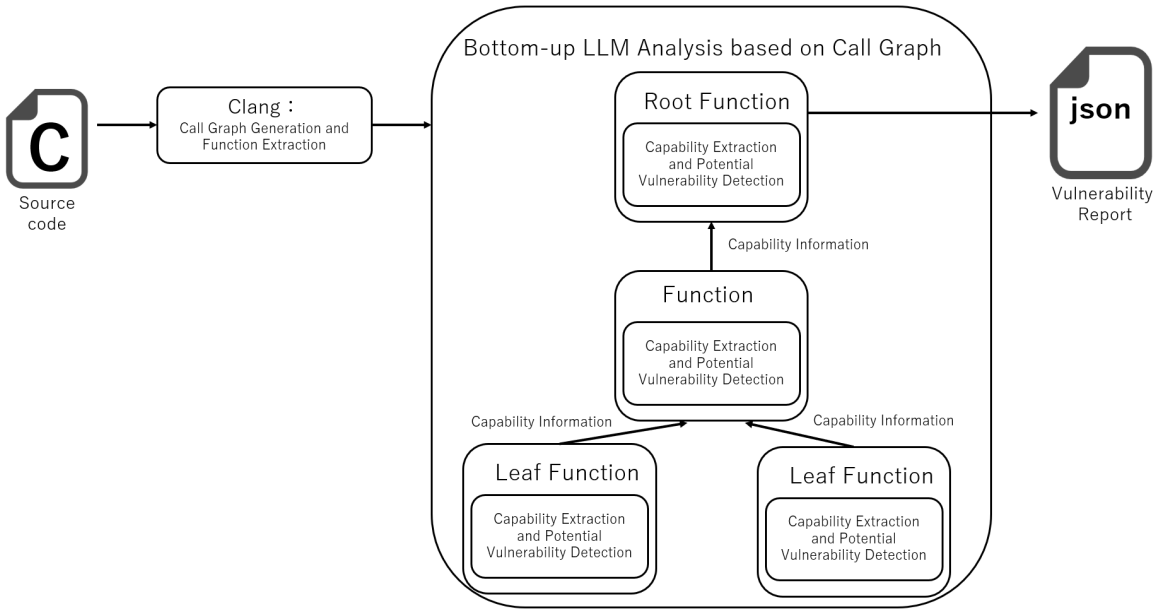

Figure 3.1: Overview of the proposed approach.

3.2. Approach Details

3.2.1. Structural Analysis and Function Extraction

Since our approach leverages Large Language Models (LLMs) to detect potential vulnerabilities at the function level, it is necessary to extract the raw source code of each function as a standalone text segment from the raw C source files. Furthermore, to enable the bottom-up propagation of *Capability Information* from callee functions to their respective callers, constructing an accurate call graph within the source code is imperative. We will define the *Capability information* in the next sections.

Extracting function definitions together with their call relationships is difficult when the source code is incomplete or syntactically invalid. Pure abstract syntax tree traversal is often insufficient in such situations. We therefore complement it with heuristic lexical token analysis, forming a hybrid resolution mechanism.

Identification of Function Definitions

The first task is to isolate valid function definitions from the translation unit. We formalize this as a filtering process over the AST nodes. We iterate through the top-level nodes of the parse tree to identify function entities. A node is selected as a valid analysis target if and only if it satisfies the following conceptual criteria:

1. **Type Validity:** The node conceptually represents a function declaration (as opposed to global variables or type definitions).
2. **Completeness:** The node represents a definitive implementation, meaning it possesses a function body, rather than being a mere forward declaration or prototype.
3. **Scope Locality:** Only functions defined in the main source file are analyzed. If a node refers to code located in external headers or system libraries, it is excluded from processing.

Robust Call Relationship Extraction

Building the edges of the call graph becomes difficult when the input code is incomplete. Complex macros or missing dependencies can prevent the parser from generating proper call expression nodes. In such cases, we rely on token level scanning within each previously identified function to recover potential call relationships.

We first determine the precise start and end points of the function's body to isolate

the analysis area (Scope Determination). Subsequently, we scan the code within this scope as a simple sequence of text to identify potential function calls whenever a name is immediately followed by a left parenthesis (Lexical Analysis). Finally, to minimize false positives, a detected identifier is only recorded as a function call if it matches a name within the set of defined functions previously identified in the global scan of the file (Cross-Reference Validation).

The set of called functions within a function body, denoted as F_{calls}, is approximated as follows:

$$F_{calls} = \{\text{val}(t_i) \mid \text{type}(t_i) = \text{ID}, \text{val}(t_i) \in D_{local}, \text{val}(t_{i+1}) = \text{'('}\} \tag{3.1}$$

Here, t_i denotes the i-th token in the token sequence. This rule is intended to recover call relationships in cases where full semantic analysis cannot be applied, for example when the code contains syntax errors or unresolved dependencies.

Bottom-Up Traversal Strategy

The constructed call graph dictates the order of the subsequent analysis. To ensure that the capability information of callee functions is available when analyzing their callers, we traverse the call graph in a bottom-up topological order. Formally, a function f is queued for analysis only after the capability extraction $\Phi(g)$ has been completed for all $g \in F_{calls}(f)$. This reverse-dependency traversal ensures that the context propagates from the leaf nodes (functions with no callees) upwards to the root nodes.

3.2.2. Extraction of Capability Information

In this phase, we extract the functional characteristics of each function as *Capability Information*, focusing on the transformation of input data into outputs. This process is automated by assigning the role of a security analyst to the LLM and instructing it to execute Data Flow Analysis. To ensure consistency and precision in the LLM's analysis, we employ specific prompt engineering techniques, including Role-Playing [11], One-Shot Learning [3], and Structured Outputs [24].

Semantic Abstraction Model

We formalize the *Capability Information* C_f for a function f not as a generic summary, but as a structured tuple designed specifically for data-flow analysis. The abstraction function Φ is defined as:

$$C_f = \Phi(f) = \langle I_f, O_f, B_f \rangle \tag{3.2}$$

where I_f denotes the Data Inputs, strictly defined as function arguments to clarify the data entry points for the local analysis scope. O_f represents the Data Outputs, defined as return values representing the direct result of the function execution. Finally, B_f encapsulates the Behavior, a descriptive summary of how function arguments are manipulated to generate the output, where the LLM is explicitly instructed to highlight any missing security checks.

As a concrete example, we examine a function that performs a file write operation, which can have security implications. Figure 3.2 shows how the original function code is converted into structured *Capability Information*.

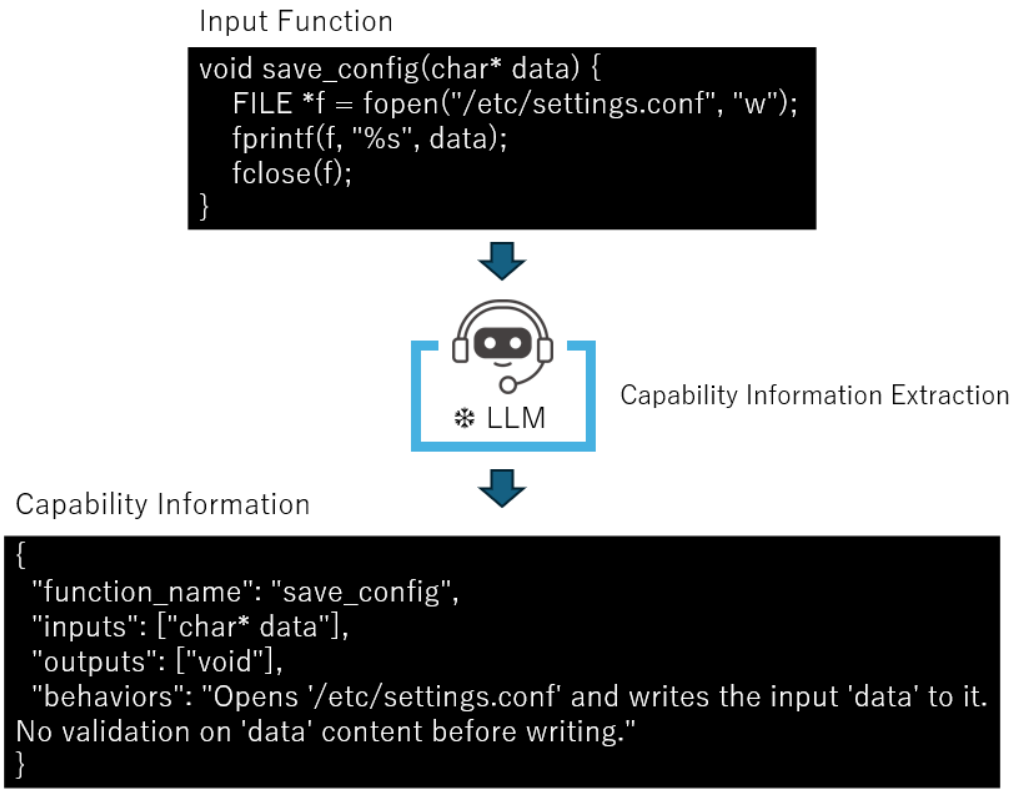

Figure 3.2: Capability Extraction Diagram

The resulting JSON output serves as a summary that can be programmatically injected into the analysis of caller function.

LLM-Based Analysis Strategy

To reliably compute $\Phi(f)$ using an LLM, we employ a guided reasoning strategy comprising three conceptual techniques:

1. **Role-Based Constraint:** The LLM is conditioned with the persona of a security analyst. This constraint ensures that the generated summary prioritizes security

implications over general functional descriptions.

2. **In-Context Learning for Standardization:** To ensure consistency across different functions, we utilize One-Shot Learning by providing an example of the abstraction process. This example serves as a logical template, demonstrating how to map a raw code fragment containing vulnerability patterns to its corresponding semantic summary. This aligns the LLM's latent space to the specific granularity required by our system.
3. **Structured Outputs:** The generated summaries follow a fixed output schema. Using this format, we extract the capability tuple $\langle I_f, O_f, B_{sec} \rangle$ and integrate it into the analysis of parent functions in the call graph.

Context Propagation Mechanism

The extracted *Capability Information* C_f serves as a surrogate for the child function's (callee) source code during the analysis of its parent (caller). By propagating this compressed semantic summary instead of the raw implementation, we achieve two theoretical advantages.

Using the summarized capability representation reduces the amount of contextual information that must be provided to the model. As a result, deeper call sequences can be examined within the available context window.

Moreover, the analysis at the caller level emphasizes how the callee behaves at its interface, instead of inspecting the internal logic line by line.

3.2.3. Identification of Critical Interactions

In the final stage, we integrate the source code of the target function with the extracted *Capability Information* of its child functions for the prompt of LLMs to identify critical interactions.

Context-Aware Interaction Analysis

Rather than classifying a function as vulnerable in isolation, this phase aims to detect specific *interactions* between the caller and its callees that lack necessary security constraints. By presenting the LLM with the "Source Code to analyze" alongside the *Capability Information* of Called Functions (context), the system can reason about inter-procedural logic. For instance, if a callee's capability indicates it "performs a memory copy without length validation," the system checks if the caller sanitizes the length before passing it.

Figure 3.3 illustrates how the *Capability Information* of a callee function interacts with the source code of a caller function.

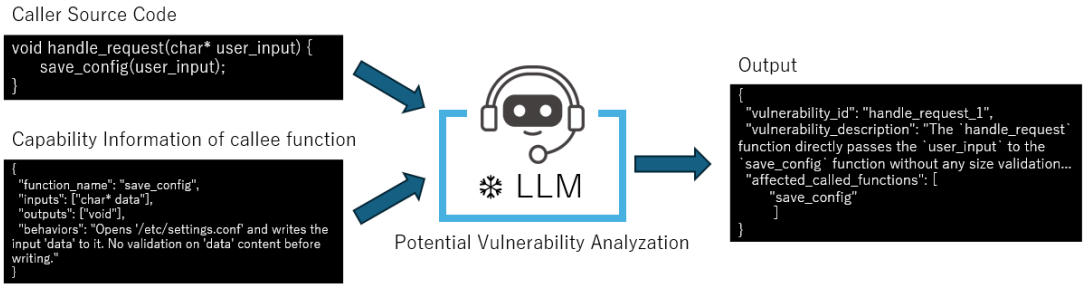

Figure 3.3: Potential Vulnerability Analysis Diagram

Result Formalization

The output of this phase is a structured set of identified critical interactions. Each identified interaction is characterized by a Potential Vulnerability Description, which is a semantic description of the flaw where the LLM is explicitly instructed to mention specific variable names and explain why a security check is missing. Additionally, the output includes a list of affected called functions, which is crucial for tracing the inter-procedural path of the vulnerability, allowing the fuzzer to prioritize paths that traverse these specific functions.

4 | Implementation Details

4.1. System Architecture

We implemented the proposed approach as a comprehensive vulnerability detection tool utilizing Python and local Large Language Models. As illustrated in the system workflow, the process is managed by a central orchestrator script that coordinates data flow between the `Clang`-based analyzer and the `Ollama`-based LLM module.

The implementation runs in a fully local setup, without relying on external API endpoints. This design choice avoids transferring source code outside the analysis environment.

The overall workflow can be divided into two main stages:

1. **Static Analysis Component**: In this stage, the system analyzes the input C source code to locate complete function definitions. The call graph is constructed from the structural information provided by Clang.
2. **LLM Inference Component**: Executes the "Capability Extraction" and "Potential Vulnerability Detection" phases by interacting with a locally hosted LLM via the Ollama framework.

4.2. System Details

First, the source code is parsed into a Translation Unit using the Clang libclang API. We iterate through the top-level cursors of the AST to identify function definitions. To ensure the analysis is confined to the target source file and excludes external headers, we apply a strict file-path normalization and filtering process. A function is indexed if it satisfies the following conditions:

1. The cursor kind matches CursorKind.FUNCTION_DECL.
2. The cursor represents a definitive implementation (is_definition).
3. The source location belongs to the primary input file path.

Robust Call Relationship Extraction via Token Stream

Traditional AST-based call graph construction often fails when the code is partially written or contains complex macros that prevent the formation of a complete CALL_EXPR node. To mitigate this, we implement a token-based scanning mechanism within the scope of each identified function:

1. **Scope Determination:** We locate the COMPOUND_STMT node to define the function body's boundaries for each function definition.
2. **Lexical Analysis:** We extract the linear token stream from the function body. The system identifies a potential function call when it detects an IDENTIFIER token immediately followed by a left parenthesis.
3. **Cross-Reference Validation:** To minimize false positives, a detected identifier is only recorded as a function call if it matches a name within the set of defined functions previously identified in the global scan of the file.

4.2.1. Local Inference and Prompt Engineering

All inference is performed locally through Ollama rather than external API services. This choice avoids transmitting source code outside the analysis environment and makes experimental results easier to reproduce. The models are hosted on a dedicated server equipped with an NVIDIA A100 40GB GPU.

For integration into the automated pipeline, the model outputs follow a predefined JSON schema specified in the system prompt. We also use low temperature sampling to reduce output variance. The resulting responses can therefore be parsed directly by the orchestrator without additional post processing.

Prompt Design for Capability Extraction

The prompt for extracting Capability Information is focusing on the transformation of input data into outputs. This process is automated by assigning the role of a security analyst to the LLM and instructing it to execute Data Flow Analysis. To ensure consistency and precision in the LLM's analysis, we employ specific prompt engineering techniques, including Role-Playing, One-Shot Learning, and Structured Outputs. As shown in Figure 4.1 and 4.2, our prompt explicitly defines the scope of analysis to guide the LLM's reasoning. Unlike general code summarization, we constrain the extraction to three specific aspects:

1. **Inputs:** Strictly defined as "Function arguments" to clarify the data entry points for the local analysis scope.
2. **Outputs:** Defined as "Return values," representing the direct result of the function execution.
3. **Behavior:** A descriptive summary of how function arguments are manipulated to generate the output. Crucially, the LLM is instructed to explicitly highlight "any missing security checks" in this section.

By injecting the raw source code into the `inputCode` placeholder, the system generates a summary that abstracts away safe logic while highlighting potential risk factors.

To improve the quality of the analysis, we utilize One-Shot Learning. We provide a concrete example within the prompt (a create_user_profile function containing a heap buffer overflow) along with its expected JSON output. This example demonstrates to the LLM how to articulate the data flow logic in the summary.

Structured Output and Context Propagation

The extracted Capability Information is strictly enforced to be in JSON format containing function_name, inputs, outputs, and behaviors. This structured format serves two purposes:

1. **Machine Readability and Robustness:** It enables the system to parse and store analysis results programmatically. Crucially, strictly enforcing the schema minimizes decoding errors (e.g., JSON parsing failures) that frequently arise from the generative and unstructured nature of LLM outputs, ensuring stable automation.
2. **Token Efficiency:** When analyzing a parent function (caller), we inject this JSON summary of the child function (callee) instead of its raw source code.

By propagating this compressed "Capability Information," the LLM analyzing the parent function can understand critical context—such as "the child function performs a memory copy without length validation"—without processing the implementation details of the child function. This approach effectively enables the detection of inter-procedural vulnerabilities while minimizing token consumption.

Prompt Design for Vulnerability Detection

For the vulnerability detection phase, the prompt is constructed to synthesize two distinct information sources: the *target function's source code* and the *Capability Information* of

```
### Source Code to Analyze
{InputCode}
### Task Description
You are a security analyst performing Data Flow Analysis on a C function.
Your goal is to extract a summary that focuses on how Data Inputs are transformed into Outputs/Actions,
specifically highlighting any missing security checks.
---
### Definitions for Extraction
1. INPUTS:
- Function arguments
2. OUTPUTS:
- Return values
3. BEHAVIOR:
- Describe how the function arguments are manipulated and how the function's output is generated based
on those inputs.
- Describe the results of the Data Flow Analysis.
---
```

Figure 4.1: Task Prompt for Capability Information Extraction

```
### Output Format
Respond in the following strict JSON format:
{{
   "function_name": string,
   "inputs": [ string, string, ... ],      // Function arguments
   "outputs": [ string, string, ... ],     // Return values
   "behaviors": string   // Summary of data manipulation and security findings
 }}

### Example
Input Code:
char *create_user_profile(char *username) {{
     char *profile_buf = (char *)malloc(64);
     if (profile_buf == NULL) return NULL;
     strcpy(profile_buf, username);
      return profile_buf;
}}
Output:
{{
   "function_name": "create_user_profile",
   "inputs": ["char *username"],
   "outputs": ["char * (pointer to heap buffer)"],
   "behaviors": "Allocates a 64-byte buffer on the heap. Copies the input 'username' into this buffer using 'strcpy' without validating its
length against the buffer size (potential heap buffer overflow). Returns the pointer to the allocated buffer."
       }}
Answer:"""
```

Figure 4.2: One shot Prompt for Capability Information Extraction

its callees. The prompt dynamically injects the target code and the aggregated summaries of child functions (formatted as text) into the context window.

A distinct feature of this prompt design is the adoption of a *Zero-Shot* strategy. Unlike the extraction phase where examples were used to enforce formatting, here we deliberately omit specific vulnerability examples to avoid *pattern-matching bias*. Providing few-shot examples often causes the LLM to overfit to the specific bug patterns shown (e.g., only detecting buffer overflows if a buffer overflow example is given). By withholding these examples, we force the model to identify critical interactions solely based on the logical inconsistencies between the caller's implementation and the callee's requirements.

As shown in Figure 4.3 and Figure 4.4, the instruction explicitly directs the model to "identify security vulnerabilities based on the provided C source code" while considering

the behavior of called functions within this unbiased context. The output is constrained to a JSON list containing:

- **vulnerability_id**: A unique identifier generated by combining the target function name and an index (e.g., `func_name_1`).
- **vulnerability_description**: A semantic description of the flaw. The prompt explicitly instructs the LLM to mention specific variable names and explain *why* a security check is missing, providing the logic necessary for constructing fuzzing targets.
- **affected_called_functions**: A list of child functions involved in the vulnerability. This field is crucial for tracing the inter-procedural path of the vulnerability, allowing the fuzzer to prioritize paths that traverse these specific functions.

```
### Task
Based on the provided C source code, identify security vulnerabilities in the
provided function's source code.

### Function Source Code to analyze
The function source code written in C that is subject to analysis:
{InputCode}

### Capability Information of Called Functions
This section describes the capability information of the functions that are called by
the function above:
{Capability Information}
```

Figure 4.3: Task Prompt for Potential Vulnerability Detection

```
### Output Format
Respond in the following strict JSON format. Do not include any explanatory text or markdown outside the JSON.
affected_called_functions is optional.:
{{
  "vulnerabilities": [
    {{
      "vulnerability_id": "<Function Name of "Function Source Code to analyze">_<Index, starting from 1>",
      "vulnerability_description": "<Detailed description of vulnerability. Explicitly mention the variable name
and why the check is missing.>",
      "affected_called_functions": ["<Names of called functions from "Capability Information of Called
Functions" if the vulnerability is potential Inter procedural vulnerability>"]
    }}
    // ... other vulnerabilities
  ]
}}
###Answer:"""
```

Figure 4.4: Output format prompt for Potential Vulnerability Detection

5 | Experimental Validation

5.1. Goals

The primary objective of our experimental evaluation is to answer the following research questions regarding the effectiveness and reliability of the proposed system. We aim to demonstrate that our approach not only detects vulnerabilities but also provides actionable insights for directed fuzzing.

RQ1: General Detection Performance on AI-Generated Code

The first goal is to assess the baseline performance of our system in identifying vulnerabilities within normal C source code. Using the FormAI-v2 dataset, we aim to measure metrics such as Recall and Distance from the ground-truth vulnerable line. This evaluation verifies the effectiveness of our proposed method on a dataset that does not explicitly specify the inclusion of inter-procedural vulnerabilities. Crucially, a key objective of this experiment is to confirm that the proposed method does not exhibit performance regression compared to the baseline. Since the injection of Capability Information inevitably inflates the context, there is a theoretical risk that this additional information could act as noise. Therefore, we aim to demonstrate that our approach maintains robustness without losing performance, even when such context is potentially redundant. Furthermore, leveraging the provided `vulnerable_line` metadata, which indicates the exact line number where the vulnerability occurs, we also validate the system's performance in precisely extracting the specific line numbers of detected vulnerabilities by error type.

RQ2: Effectiveness of Inter-Procedural Analysis

The core contribution of this paper is the use of *Capability Information* to detect vulnerabilities spanning multiple functions. Our second goal is to validate whether this context propagation mechanism functions as intended. Using the InterPVD dataset, we aim to demonstrate that the system can trace the root cause of a vulnerability from a Vulnerability-triggered function (callee) to the Patch statement function (caller), a task

that traditional single-function analysis would fail to accomplish.

5.2. Datasets

Our experimental evaluation relies on two distinct datasets to measure different aspects of the system's performance. We use the FormAI Dataset to establish a baseline for general vulnerability detection within AI-generated code. In contrast, the InterPVD dataset serves a more targeted purpose: validating how well our approach handles complex, inter-procedural vulnerabilities in real-world software.

5.2.1. FormAI-v2 dataset

To establish our baseline, we utilized the FormAI Dataset [29], a collection of C source code generated by various LLMs (including Gemini-pro, GPT-4, Falcon, and CodeLlama2). This dataset provides a diverse set of coding patterns and potential vulnerabilities introduced during the generation process.

Each sample in the dataset is stored in JSON format and contains detailed ground truth metadata. These samples are characterized by several key attributes: the classification of the sample (e.g., VULNERABLE) is specified in the **category** field, while the **function** attribute identifies the name of the function in which the vulnerability is triggered. Furthermore, the **error_type** provides a description of the specific vulnerability type, such as a buffer overflow on sscanf, and the **vulnerable_line** indicates the exact line number in the source code where the vulnerability occurs.

For this experiment, we randomly selected a subset of **100 samples** from the FormAI Dataset. This selection serves as a baseline to evaluate whether the proposed system can correctly identify critical locations in source code where unsafe interactions could occur.

5.2.2. InterPVD

To evaluate the system's capability in handling inter-procedural dependencies we utilized the Inter-Procedural Vulnerability Dataset (InterPVD) [12]. This dataset is derived from real-world open-source software and focuses specifically on vulnerabilities that span across multiple functions.

Unlike traditional datasets that only point to the patched location, InterPVD annotates the **vulnerability-triggered function** (the root cause location) in addition to the fixed function. It provides comprehensive traceability information, including standard iden-

tifiers such as the **CVE ID and CWE ID**. Furthermore, the dataset specifies the **vulnerability-triggered statements**, which refer to the code segments that trigger the vulnerability without requiring a patch, and the **critical variable**, which refers to the variable related to the vulnerability.

We selected a subset of **44** samples from this dataset. The selection criteria required each sample to explicitly contain both the **patch statement function** and the **vulnerability-triggered function** in a single source file, ensuring that the dataset specifically targets the multi-function analysis capabilities of our proposed method. The source files in this subset exhibit significant variation in size, ranging from approximately 100 to 15,000 lines of code.

5.3. Experimental Setup

5.3.1. Evaluation Methodology: LLM-as-a-Judge

Evaluating the quality of natural language explanations generated by LLMs presents a challenge, as standard string-matching metrics cannot capture semantic correctness. To address this, we adopted an *LLM-as-a-Judge* [4, 13] approach, where a separate, high-capability LLM instance (acting as a security expert) verifies the correctness of the model's outputs against the ground truth. To ensure rigorous evaluation, we filtered the model's outputs based on the dataset metadata, providing the judge LLM only with the results for functions contained in the `List of functions from to-be-patched function to vulnerability-triggered function`.

Since the objective of this study is to improve target selection for directed Greybox fuzzing by identifying potentially vulnerable locations in source code, missing a true vulnerability candidate is more detrimental than over-approximating suspicious locations. In particular, inter-procedural vulnerabilities require capturing root causes, statements, and variable dependencies across function boundaries; failure to identify these elements would directly limit the effectiveness of subsequent fuzzing. Therefore, we prioritize `Recall` as the evaluation metric for each criterion to measure the model's ability to successfully capture ground-truth vulnerability components.

We employed distinct verification strategies tailored to the specific characteristics of each dataset.

Evaluation Criteria for FormAI Dataset

For the FormAI dataset, we evaluated LLMs' vulnerability detection capability by error type and the effectiveness of the proposed pipeline on a dataset that does not explicitly specify the inclusion of inter-procedural vulnerabilities. To this end, we verified the following three criteria. Since the output of our model focuses on generating explanations of potentially vulnerable locations in the source code, we use recall as the evaluation metric for criterion 1 and criterion 2.

Criteria 1: Vulnerability Type Alignment Here, we assess the semantic equivalence between the model's output and the official vulnerability classification. We construct a prompt injecting the Ground Truth Error Type (e.g., "buffer overflow on sscanf") and the vulnerability description generated by the detection model. The Judge LLM is instructed to output `true` only if the candidate explicitly describes the specific type of error, rejecting vague descriptions like bug or error.

Criteria 2: Vulnerable Code Identification Rather than just naming the error type, the system must also locate the correct triggering logic. We extract the source code of the ground truth `vulnerable_line` as the Ground Truth Code Snippet. The Judge LLM determines if the vulnerability description generated by detection model references or implies the logic contained within this snippet.

Criteria 3: Distance from the Ground-Truth vulnerable_line This criterion verifies whether the model correctly detects the vulnerable line. We use the absolute difference between the vulnerable_line generated by the detection model and the ground-truth vulnerable_line as the evaluation metric.

Evaluation Criteria for InterPVD

Because the InterPVD dataset explicitly features inter-procedural vulnerabilities, it is an ideal dataset to evaluate performance of our proposed method. To accurately measure performance in this complex context, we established four distinct verification criteria.

Criteria 1: Semantic Alignment with CVE (Root Cause Verification) This criterion assesses whether the system has correctly identified the fundamental root cause of the vulnerability. We provide the Judge LLM with the official Ground Truth CVE Description and the vulnerability description generated by the detection model. The Judge is instructed to return `true` only if the vulnerability description explicitly describes

the root cause associated with the CVE.

Criteria 2: Taxonomic Correctness (CWE Match) This criterion verifies the technical classification accuracy. The Judge LLM compares the vulnerability description generated by the detection model against the Ground Truth CWE (Common Weakness Enumeration) full name. This step ensures that the system correctly categorizes the specific type of weakness.

Criteria 3: Precision of Statement Identification To validate the inter-procedural analysis capability, we verify if the system pinpointed the exact location where the vulnerability was triggered. The Judge LLM checks if the Ground Truth Vulnerability-Triggered Statement (the specific line of code in the callee or caller) is explicitly cited or described in the vulnerability description output. This is the strictest criterion, confirming that the system successfully traced the dependency chain to the correct source line.

Criteria 4: Precision of Critical Variable Identification This criterion verifies whether the vulnerability explanation correctly points to the specific variable related to the vulnerability. We provide the Judge LLM with the official Ground Truth Critical Variable and the vulnerability description generated by the detection model. The Judge LLM determines whether the vulnerability description explicitly mentions the critical variable.

5.3.2. Experimental Configuration and Baselines

To evaluate the proposed system, we conducted experiments using small size open-weights Large Language Models (LLMs) hosted locally via the **Ollama** [23] framework.

Model Selection

We selected models based on their reasoning capabilities and code understanding performance. The specific model allocations for the detection and evaluation phases are configured as follows:

- **FormAI-v2 Experiments:** For the detection phase, we utilized **Gemma 3:12B** [9]. For the evaluation phase (LLM-as-a-Judge), we employed the larger **Qwen3-Coder:30B** [26] to ensure the judge possesses superior reasoning capabilities compared to the detector.
- **InterPVD Experiments:** To assess performance on complex inter-procedural vul-

nerabilities, we employed two distinct models for detection: **Gemma 3:12B** and **Qwen3-Coder:30B**. For the evaluation phase, **Qwen3-Coder:30B** was consistently utilized as the judge due to its high proficiency in code logic analysis.

Baselines for Ablation Study

To quantify the contribution of our core proposal—the *Capability Information*—we established a baseline configuration for comparison. We compare the following two settings under identical model conditions:

- **With CI**: The proposed method, which injects the summarized Capability Information of callee functions into the analysis of the caller.
- **Without CI**: A baseline approach where each function is analyzed in isolation without external context.

By comparing these two settings, we isolate and measure the specific impact of interprocedural context propagation on detection accuracy.

5.4. Experimental Results and Analysis

In this section, we present the quantitative results of our experiments using the FormAI and InterPVD datasets. We evaluate the impact of Capability Information (CI) on detection Recall and analyze the scalability of the proposed system.

5.4.1. RQ1: Results on FormAI Dataset (AI-Generated Code)

Table 5.1 summarizes the results for the 100 randomly selected samples from the FormAI dataset.

Table 5.1: Comparison of Detection Recall on FormAI Dataset (n=100)

Metric	Without CI	With CI (Proposed)
Error Type Match	0.39	**0.41**
Code Snippet Match	**0.50**	0.49

As indicated in Table 5.1, the performance metrics for the Proposed Method (with Capability Information) are comparable to those of the Baseline (without), showing no significant deviation. We attribute this result to the intrinsic nature of the FormAI dataset. We infer from these results that, unlike benchmarks specifically curated for complex data flows, the AI-generated code snippets in FormAI likely consist predominantly of *intra-*

procedural vulnerabilities confined within a single function scope, suggesting a lack of significant explicit inter-procedural dependencies. Since the dataset appears to lack the complex call chains that our method targets, the injection of Capability Information implies a functionally neutral operation for these samples. Crucially, however, this result demonstrates that our proposed mechanism introduces no negative impact or noise to the detection of standard vulnerabilities. It confirms that our system is capable of maintaining robust baseline performance on simple code, effectively falling back to standard analysis when inter-procedural context is unnecessary.

Table 5.2: Average Distance from Ground-truth vulnerable line by error type

Metric	**Average Distance**
NULL pointer	4.11
buffer overflow on scanf	2.67
invalid pointer	4.58
array bounds violated	4.21
array bounds violated: upper bound	4.8
dereference failure: forgotten memory	9.35
array bounds violated: lower bound	3.54
arithmetic overflow on sub	5.8
arithmetic overflow on add	6.94
arithmetic overflow on mul	6.92
arithmetic overflow on floating-point ieee_mul	12.49
buffer overflow on fscanf	8.06
division by zero	3.71
VLA array size in bytes overflows address space size	3.16
buffer overflow on sscanf	2.48

Table 5.2 presents the average distance (in lines) between the vulnerable location predicted by the LLM and the ground truth by error type. The results reveal a significant disparity in the model's localization performance, strongly correlating with the syntactic locality and semantic complexity of the vulnerability type.

The model demonstrates high precision, indicated by a low average distance, for vulnerabilities that are syntactically localized or manifest within a limited context. For instance, Buffer Overflow on scanf (2.67), sscanf (2.48), and Division by Zero (3.71) show minimal deviation. These defects are often self-contained; a missing width specifier in a scanf call is a syntax-level error that does not require extensive cross-function analysis.

Conversely, the model's performance degrades significantly when the vulnerability involves long-range dependencies or requires tracking state changes over time. Arithmetic Overflow on Floating-point ieee_mul exhibited the worst performance with an average distance of

12.49 lines. This large deviation suggests that the model fails to bridge the gap between the arithmetic operation, which is the root cause, and its distant consequences, such as NaN propagation or logic errors in subsequent conditional branches. A similar trend is observed in Dereference Failure: Forgotten Memory (9.35), which typically corresponds to Use-After-Free (UAF) scenarios. Since UAF bugs involve a temporal sequence of allocation, deallocation, and invalid access, the distance of nearly 10 lines implies the model likely confuses the freeing or allocation site with the actual dereference site, confirming the limitation of current LLMs in tracking variable lifecycles across complex control flows.

5.4.2. RQ2: Results on InterPVD (Real-World Complex Vulnerabilities)

Table 5.3 and Table 5.4 present the results for the 44 instances from the InterPVD dataset.

Table 5.3: Comparison of Detection Recall on InterPVD (Gemma3:12b)

Metric	Without CI	With CI (Proposed)
CVE Description Match	0.409	**0.477**
CWE Match	0.432	**0.477**
Statement Match	**0.364**	0.205
Critical Variable Match	0.318	0.318

Table 5.4: Comparison of Detection Recall on InterPVD (qwen3-coder:30b)

Metric	Without CI	With CI (Proposed)
CVE Description Match	0.455	**0.523**
CWE Match	**0.568**	0.477
Statement Match	0.159	**0.227**
Critical Variable Match	0.432	**0.455**

As presented in Table 5.3 and Table 5.4, the evaluation using LLM-as-a-Judge demonstrates that the proposed method (incorporating Capability Information) achieved higher scores than the baseline (without) across the majority of metrics for both underlying models. However, the magnitude of these improvements remained **modest**, typically limited to the **single-digit percentage range**. We attribute this limited margin of improvement to two primary factors inherent in the evaluation methodology, rather than a lack of efficacy in the proposed mechanism.

First, the **Partial Matching nature** of the LLM-as-a-Judge metric tends to award scores based on semantic similarity even if the reasoning is not strictly precise, potentially

masking the subtle improvements in logic provided by our method.

Second, and more critically, the **Information Equivalence during Evaluation** played a decisive role. To ensure a fair comparison, the Judge LLM was provided with the detection outputs for all functions listed in the dataset metadata (`List of functions from to-be-patched function to vulnerability-triggered function`). Consequently, regardless of whether Capability Information was used during the detection phase, the Judge LLM received a similar aggregate of text data covering the entire function chain at the time of evaluation. This suggests that while Capability Information is crucial for the *Detector* to understand context locally, the *Judge* (having access to global outputs) could reconstruct the context independently, thereby neutralizing the statistical difference in this specific metric.

Manual Evaluation

Given the limitations of the automated metrics discussed in the previous section, specifically the difficulty of capturing subtle logic improvements via LLM-as-a-Judge—relying solely on quantitative scores may fail to fully demonstrate the efficacy of our approach. To address this and verify the impact of Capability Information on the model's reasoning process, we conducted a **manual evaluation**. In this section, we examine two representative CVE instances from the InterPVD dataset and perform an in-depth comparative analysis of the outputs generated by **Gemma 3:12B**. This case study focuses on visualizing how the injection of Capability Information specifically alters the detection logic and enables the identification of complex inter-procedural vulnerabilities that were missed or only vaguely described by the baseline.

CVE-2014-9728 To evaluate the impact of Capability Information on the model's performance, we present a comparative analysis of CVE-2014-9728 [22], a vulnerability found in the UDF filesystem implementation of the Linux kernel. This vulnerability is classified as CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer). The issue stems from an inter-procedural flaw between the caller function, `udf_symlink_filler`, and the callee function, `udf_pc_to_char`. While caller function allocates a memory page for the symbolic link path, it passes this buffer to callee function without a mechanism to enforce the buffer's size limit. Consequently, callee function performs write operations without validating the effective bounds of the destination buffer based on the input size (derived from `inode->i_size`), potentially exceeding the allocated page boundary and causing an out-of-bounds access.

First, we examine the approach using Capability Information. As shown in Fig. 5.1 and

Fig. 5.2, the LLM successfully identified the root cause by stating, "There's no check to ensure that the total size... does not exceed the allocated buffer size." Furthermore, it accurately captured the inter-procedural context by noting the mismatch between the allocation in udf_symlink_filler and the lack of validation in udf_pc_to_char ("The page buffer is allocated..., but ... does not validate..."). This demonstrates that Capability Information enables the model to understand the semantic logic of data flow and correctly identify the page buffer as the critical resource at risk of overflow.


```
{
   "vulnerability_id": "udf_pc_to_char_1",
   "vulnerability_description": "The function `udf_pc_to_char` is vulnerable to a buffer overflow. The `p` pointer is incremented based on the values of `pc->lengthComponentIdent` and fixed sizes (3 for ¥"../¥", 2 for ¥"./¥"). There's no check to ensure that the total size written to `to` does not exceed the allocated buffer size. Specifically, in the case of `componentType == 5`, `udf_get_filename` is called, and the increment of `p` is based on the return value of `udf_get_filename`. If `udf_get_filename` returns a value greater than the remaining space in the `to` buffer, a buffer overflow will occur. The variable `p` is used without bounds checking.",
   "affected_called_functions": []
}
```


Figure 5.1: Analysis results for the inter-procedural vulnerability in CVE-2014-9728 by approach using Capability Information (a) Analysis of the callee function udf_pc_to_char


```
 {
   "vulnerability_id": "udf_symlink_filler_1",
   "vulnerability_description": "The `udf_pc_to_char` function is called with `inode->i_size` as the `fromlen` parameter. There is no check to ensure that the total length of the path constructed by `udf_pc_to_char` does not exceed the size of the `page` buffer. If `udf_get_filename` within `udf_pc_to_char` returns a filename that is too long, or if the total length of the constructed path exceeds the size of the `page` buffer, a buffer overflow can occur. The `page` buffer is allocated based on the page size, but the `udf_pc_to_char` function does not validate the length of the constructed path against this size.",
   "affected_called_functions": [
        "udf_pc_to_char"
    ]
}
```


Figure 5.2: Analysis results for the inter-procedural vulnerability in CVE-2014-9728 by approach using Capability Information (b) Analysis of the caller function udf_symlink_filler

In contrast, Fig. 5.3 shows the result of the approach without Capability Information. While the model attempts to identify a buffer overflow, it fails to distinguish between the

input source and the output destination. The model incorrectly focuses on the input buffer symlink, stating, "If inode->i_size is larger than the available space in symlink ... a buffer overflow can occur." This is a hallucination regarding the vulnerability context; the actual vulnerability lies in the write operation to the page buffer, not the read operation from symlink. This failure highlights that without Capability Information, the LLM struggles to resolve the direction of data flow in complex inter-procedural scenarios, leading to misinterpretation of the vulnerability mechanism.


```
{
   "vulnerability_id": "udf_symlink_filler_1",
   "vulnerability_description": "Potential buffer overflow in udf_pc_to_char. The
function `udf_pc_to_char` is called with `inode->i_size` as the size parameter. If
`inode->i_size` is larger than the available space in `symlink` (which is either `iinfo-
>i_ext.i_data + iinfo->i_lenEAttr` or `bh->b_data`), a buffer overflow can occur.
There's no check to ensure that `inode->i_size` is less than or equal to the size of the
buffer pointed to by `symlink`. The size of `symlink` depends on whether `iinfo-
>i_alloc_type == ICBTAG_FLAG_AD_IN_ICB` or not, and the code doesn't validate
`inode->i_size` against either of these sizes.",
   "affected_called_functions": [
         "udf_pc_to_char"
      ]
}
```


Figure 5.3: Analysis results for the inter-procedural vulnerability in CVE-2014-9728 by approach not using Capability Information (b) Analysis of the caller function udf_symlink_filler

CVE-2013-0862 We present a comparative analysis of CVE-2013-0862 [21], a vulnerability found in the FFmpeg. This vulnerability is classified as CWE-189 (Numeric Errors / Integer Overflow). The issue stems from an inter-procedural flaw between the caller function, `process_frame_obj`, and the callee function, `init_sizes`. While `process_frame_obj` extracts video dimensions from the bitstream, it passes these values to `init_sizes` without sufficient validation. Consequently, `init_sizes` performs a buffer size calculation determining `ctx->buf_size`, which is susceptible to an integer overflow if the dimensions are excessively large.

First, we examine the approach using Capability Information. As shown in Fig. 5.4 and Fig. 5.5, the LLM successfully identified the root cause by explicitly stating that the calculation `ctx->aligned_width * ctx->aligned_height * sizeof(ctx->frm0[0])` involves a data type size multiplication that contributes to the overflow. Furthermore, it accurately captured the inter-procedural context by noting that the unvalidated inputs flow from `process_frame_obj` to `init_sizes` ("The init_sizes function is called with left

+ w and top + h..."). This demonstrates that Capability Information enables the model to understand the precise arithmetic semantics—specifically the impact of the `sizeof` operator—and correctly trace the propagation of tainted data across function boundaries.


```
{
  "vulnerability_id": "init_sizes_2",
  "vulnerability_description": "Integer overflow vulnerability. The calculation `ctx->aligned_width * ctx->aligned_height * sizeof(ctx->frm0[0])` to determine `ctx->buf_size` is susceptible to integer overflow if `ctx->aligned_width` and `ctx->aligned_height` are sufficiently large. This could lead to unexpected behavior or memory corruption. The variables `ctx->aligned_width` and `ctx->aligned_height` are not validated for size before multiplication.",
  "affected_called_functions": []
}
```


Figure 5.4: Analysis results for the inter-procedural vulnerability in CVE-2013-0862 by approach using Capability Information (a) Analysis of the callee function udf_pc_to_char


```
{
  "vulnerability_id": "process_frame_obj_1",
  "vulnerability_description": "The `init_sizes` function is called with `left + w` and `top + h` as width and height respectively. These values are not validated against `SANMVideoContext->width` and `SANMVideoContext->height`. If `left + w` or `top + h` are significantly larger than `SANMVideoContext->width` or `SANMVideoContext->height`, the `init_sizes` function could allocate excessively large buffers, potentially leading to a denial-of-service condition. Specifically, `left + w` is used as width and `top + h` is used as height in `init_sizes`.",
  "affected_called_functions": [
      "init_sizes"
  ]
}
```


Figure 5.5: Analysis results for the inter-procedural vulnerability in CVE-2013-0862 by approach using Capability Information (b) Analysis of the caller function udf_symlink_filler

In contrast, Fig. 5.6 shows the result of the approach without Capability Information. The model hallucinates a non-existent constraint, stating that there is "no check to ensure that w and h are positive values." This indicates a failure to recognize that the variables are strictly defined as unsigned integers (`uint16_t`) in the source code. Additionally, the model failed to detect the inter-procedural dependency, leaving the `affected_called_functions` list empty. This highlights that without Capability Information, the LLM struggles with type awareness and fails to reconstruct the complete call graph required to pinpoint the vulnerability mechanism accurately.

```
{
   "vulnerability_id": "process_frame_obj_2",
   "vulnerability_description": "Missing bounds check on `w` and `h`. The code
calculates `left + w` and `top + h` and uses them to resize the video context. However,
there's no check to ensure that `w` and `h` are positive values. If `w` or `h` are
negative, the calculations `left + w` and `top + h` will be incorrect, potentially leading
to out-of-bounds access or other unexpected behavior.",
   "affected_called_functions": []
}
```


Figure 5.6: Analysis results for the inter-procedural vulnerability in CVE-2013-0862 by approach not using Capability Information (b) Analysis of the caller function udf_symlink_filler

5.4.3. Discussion

We now examine the results of RQ1 and RQ2 to assess the overall behavior of the proposed system as a mechanism for selecting targets. The outcomes on FormAI (Table 5.1) differ from those observed in the InterPVD case studies. In the relatively simple intra procedural programs from FormAI, incorporating Capability Information did not noticeably change the baseline detection results. Performance remained comparable, and no clear degradation was observed.

For the InterPVD dataset, the quantitative evaluation under the LaaS setting showed consistent improvements across several metrics when Capability Information was enabled, compared to the configuration without it. However, the magnitude of these improvements was moderate rather than substantial.

While the quantitative improvement in the LLM-as-a-Judge metric was modest (single-digit percentage), the manual evaluation revealed an enhancement in the quality of reasoning. For Directed Greybox Fuzzing, the ultimate goal is not merely to classify a function as vulnerable (binary classification), but to provide precise vulnerability locations and explicit root causes to guide the fuzzer. The ability of our method to reduce hallucinations about missing checks and correctly identify variable constraints, as seen in the CVE-2013-0862 case, demonstrates that the proposed approach can provide more informative target descriptions to optimize downstream security tasks, such as manual code review or automated testing like Directed Greybox Fuzzing (DGF), even if standard detection metrics struggle to capture this semantic nuance.

6 | Limitations

6.1. Lack of Ground Truth for Vulnerability Explanations

A primary limitation of this study lies in the granularity of existing vulnerability datasets. While our proposed system is designed to generate specific, natural language explanations for identified vulnerabilities (i.e., why the code is dangerous), current standard datasets do not contain corresponding ground truth explanations.

Most existing datasets and benchmarks in this field are annotated primarily for binary classification (vulnerable vs. safe) or multi-class classification (CWE categorization). They typically lack detailed, localized descriptions of the vulnerability's root cause. Consequently, strictly evaluating the semantic accuracy of the LLM-generated explanations is challenging. In our evaluation, we relied on indirect matching using CVE descriptions, CWE definitions, and the location of vulnerable code snippets as proxies for accuracy. However, this approach cannot fully guarantee that the model detected the vulnerability for the correct reasoning, as opposed to relying on spurious correlations.

Addressing this issue would require datasets that include explicit explanations of why a given source code is vulnerable. At present, such resources are largely unavailable. Constructing them would likely involve close collaboration with security practitioners, who could annotate vulnerable samples with detailed descriptions at the code level.

With access to explanation level annotations, researchers could evaluate generated outputs using established NLP metrics such as BERTScore or BLEU, in addition to traditional detection metrics. This would make it possible to assess not only whether a vulnerability is identified, but also whether the reasoning behind the explanation aligns with expert understanding.

6.2. Scalability and Inference Latency

During the evaluation on the InterPVD dataset, we identified a significant challenge regarding execution time and scalability. Our system architecture requires querying the LLM up to $2N$ times for each file (where N is the number of functions) to generate Capability Information bottom-up and then detect vulnerabilities for the entire call tree.

We categorized the target source files into two groups: a **Short Code Group** (files with under 5,000 lines) and a **Long Code Group** (files with over 5,000 lines). Although the Long Code Group contained significantly fewer files (only 8 compared to 37 in the Short group), the total analysis time for these 8 files exceeded **3 hours**, showing a disproportionate increase in latency.

We attribute this not only to the linear increase in API calls as the function count N grows, but also to the intensifying complexity of the call graph in larger codebases. Deeply nested dependencies result in longer prompts (due to accumulated Capability Information from multiple child functions) and increased reasoning complexity for the LLM, thereby prolonging the inference time for each individual query. Consequently, this cost is justifiable for the deep analysis of specific critical components. However, for large source files, real-time verification remains computationally impractical without optimizations such as parallel processing.

7 Future Works

In this study, we proposed a vulnerability detection method that leverages large language models (LLMs) to account for inter-function dependencies, and demonstrated its effectiveness. However, as discussed in Chapter 5, as well as based on new insights obtained through our experiments, the following points are identified as future directions for further research.

7.1. Evaluation with Directed Fuzzing

The primary objective of this study is to automate target identification for Directed Greybox Fuzzing (DGF). In the present experiments, we evaluated the semantic accuracy of the detected vulnerability information using an LLM-as-a-Judge approach. However, we did not measure the actual performance such as target reachability or time to crash detection when using this output to drive DGF.

As future work, it is necessary to integrate our system with existing DGF tools such as AFLGo and quantitatively evaluate to what extent the target location information and vulnerability descriptions generated by the proposed method improve efficiency in real-world fuzzing.

7.2. Construction of Explanatory Vulnerability Datasets

As pointed out in Chapter 5, many existing vulnerability datasets focus primarily on binary classification or CWE categorization, and therefore lack appropriate ground truth for evaluating the detailed reasoning process behind "why" a vulnerability occurs, which is the central aim of this study.

To build a more reliable detection model, it is essential to construct a new dataset in collaboration with security experts, in which the root causes and mechanisms of vulnerabilities are described in detail in natural language. Such a dataset would enable more rigorous and quantitative evaluation using NLP metrics such as BLEU and BERTScore,

and is expected to contribute to further improvements in the reasoning capabilities of large language models.

8 | Conclusions

As software grows larger and more complex, the number of reported vulnerabilities is increasing. This makes Directed Greybox Fuzzing (DGF) a very useful and efficient tool for detecting them. However, the practical application of DGF requires careful target selection. Determining which parts of the code should be prioritized for exploration. Conventional approaches have relied heavily on expert knowledge and manual review for this process, creating a bottleneck in rapid development cycles. In this paper, we propose a method to identify critical locations in source code where unsafe interactions could be located within source code using LLMs not as a standalone detector, but as a preprocessing step to optimizing Directed Greybox Fuzzing.

The core contribution of this study lies in the introduction of the concept of *Capability Information*, which summarizes the functionality of callee functions. This approach makes it possible to propagate complex data flows and missing security requirements across function call hierarchies in a bottom-up manner without directly appending the entire source code of callee functions to the prompt.

In the experimental evaluation using the FormAI dataset, we confirmed that the proposed method maintains performance comparable to the baseline even for simple intra-procedural vulnerabilities, and that the additional contextual information does not act as noise, demonstrating the robustness of the approach.

Another achievement of this study is the case study conducted on real-world complex vulnerabilities using the InterPVD dataset. In cases such as CVE-2014-9728 and CVE-2013-0862, the proposed method successfully identified inconsistencies between callers and callees that baseline approaches failed to detect.

By leveraging *Capability Information*, the LLM was able to correctly recognize variable type definitions and buffer size constraints, demonstrating that it can explain the root causes of vulnerabilities based on logical reasoning rather than simple pattern matching.

In summary, this study establishes an approach that leverages LLMs not merely as classifiers, but as reasoning engines capable of interpreting code semantics and compressing

contextual information. By doing so, it combines the comprehensiveness of analysis with the depth of insight typically provided by expert review.

The detailed vulnerability reports generated by our system—containing specific variable names and explicit logical flaws—constitute precisely the kind of fine-grained target information required for Directed Greybox Fuzzing (DGF). The proposed method thus provides an important foundation for achieving high-precision, low-cost automated vulnerability detection in development environments where access to highly skilled security experts is limited.

Bibliography

[1] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang, B. Hui, L. Ji, M. Li, J. Lin, R. Lin, D. Liu, G. Liu, C. Lu, K. Lu, J. Ma, R. Men, X. Ren, X. Ren, C. Tan, S. Tan, J. Tu, P. Wang, S. Wang, W. Wang, S. Wu, B. Xu, J. Xu, A. Yang, H. Yang, J. Yang, S. Yang, Y. Yao, B. Yu, H. Yuan, Z. Yuan, J. Zhang, X. Zhang, Y. Zhang, Z. Zhang, C. Zhou, J. Zhou, X. Zhou, and T. Zhu. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[2] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3134020. URL https://doi.org/10.1145/3133956.3134020.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

[4] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet?, 2020. URL https://arxiv.org/abs/2009.07235.

[5] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018. doi: 10.1109/SP.2018.00046.

[6] G. Digregorio, R. A. Bertolini, F. Panebianco, and M. Polino. Poster: libdebug, build your own debugger for a better (hello) world. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 4976–4978. ACM, Dec. 2024. doi: 10.1145/3658644.3691391. URL http://dx.doi.org/10.1145/3658644.3691391.

[7] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. URL https://arxiv.org/abs/2002.08155.

[8] M. Fu and C. Tantithamthavorn. Linevul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 608–620, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3528452. URL https://doi.org/10.1145/3524842.3528452.

[9] Google. Ollama gemma3, 2025. URL https://ollama.com/library/gemma3. Accessed: 2025-02-10.

[10] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

[11] A. Kong, S. Zhao, H. Chen, Q. Li, Y. Qin, R. Sun, X. Zhou, E. Wang, and X. Dong. Better zero-shot reasoning with role-play prompting, 2024. URL https://arxiv.org/abs/2308.07702.

[12] Z. Li, N. Wang, D. Zou, Y. Li, R. Zhang, S. Xu, C. Zhang, and H. Jin. On the effectiveness of function-level vulnerability detectors for inter-procedural vulnerabilities, 2024. URL https://arxiv.org/abs/2401.09767.

[13] Y. Liu, D. Iter, Y. Xu, S. Wang, R. Xu, and C. Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment, 2023. URL https://arxiv.org/abs/2303.16634.

[14] Z. Liu, Z. Tang, J. Zhang, X. Xia, and X. Yang. Pre-training by predicting program dependencies for vulnerability analysis tasks, 2024. URL https://arxiv.org/abs/2402.00657.

[15] LLVM Project. Clang: a C language family frontend for LLVM. https://clang.llvm.org/, 2026. Accessed: 2026-01-31.

[16] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang. Wizardcoder: Empowering code large language models with evol-instruct, 2025. URL https://arxiv.org/abs/2306.08568.

[17] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey, 2019. URL https://arxiv.org/abs/1812.00140.

[18] MITRE Corporation. Common vulnerabilities and exposures (cve). https://www.cve.org/, 2026. Accessed: 2026-01-27.

[19] MITRE Corporation. Common weakness enumeration (CWE). https://cwe.mitre.org/, 2026. Accessed: 2026-01-31.

[20] National Institute of Standards and Technology. National institute of standards and technology. https://www.nist.gov/, 2026. Accessed: 2026-01-27.

[21] NIST. National vulnerability database cve-2013-0862, 2013. URL https://nvd.nist.gov/vuln/detail/CVE-2013-0862. Accessed: 2025-02-10.

[22] NIST. National vulnerability database cve-2014-9728, 2015. URL https://nvd.nist.gov/vuln/detail/CVE-2014-9728. Accessed: 2025-02-06.

[23] Ollama. Ollama, 2026. URL https://ollama.com/. Accessed: 2025-02-10.

[24] OpenAI. Introducing structured outputs in the api, 2024. URL https://openai.com/ja-JP/index/introducing-structured-outputs-in-the-api/. Accessed: 2025-02-06.

[25] F. Panebianco, A. Isgrò, S. Longari, S. Zanero, and M. Carminati. Guessing as a service: Large language models are not yet ready for automated vulnerability detection. In *Proceedings of the Joint National Conference on Cybersecurity (ITASEC & SERICS 2025)*, Bologna, Italy, February 2025. URL http://hdl.handle.net/11311/1284205.

[26] Qwen. Ollama qwen3, 2025. URL https://ollama.com/library/qwen3-coder. Accessed: 2025-02-10.

[27] A. Shestov, R. Levichev, R. Mussabayev, E. Maslov, A. Cheshkov, and P. Zadorozhny. Finetuning large language models for vulnerability detection, 2024. URL https://arxiv.org/abs/2401.17010.

[28] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, H. Tong, S. Das, E. T. Barr, and W. Le. A comprehensive study of the capabilities of large language models for vulnerability detection, 2025. URL https://arxiv.org/abs/2403.17218.

[29] N. Tihanyi, T. Bisztray, M. A. Ferrag, R. Jain, and L. C. Cordeiro. How secure is ai-generated code: a large-scale comparison of large language models. *Empirical Software Engineering*, 30(47), 2025. doi: 10.1007/s10664-024-10590-1. URL https://doi.org/10.1007/s10664-024-10590-1.

[30] B. Wu, C. Liu, Z. Li, Y. Cao, J. Sun, and S.-W. Lin. Enhancing vulnerability detec-

tion via inter-procedural semantic completion. *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025. doi: 10.1145/3728912. URL https://doi.org/10.1145/3728912.

[31] H. Xu, Y. Zhao, and H. Wang. Directed greybox fuzzing via large language model, 2025. URL https://arxiv.org/abs/2505.03425.

[32] L. Yu, Y. Lu, Y. Shen, Y. Li, and Z. Pan. Vulnerability-oriented directed fuzzing for binary programs. *Scientific Reports*, 12(1):4271, Mar 2022. ISSN 2045-2322. doi: 10.1038/s41598-022-07355-5.

[33] T. Zhang, C. Yang, Y. Su, M. Weyssow, H. Nguyen, T. Bui, H. J. Kang, Y. Li, E. L. Ouh, L. K. Shar, and D. Lo. Benchmarking large language models for multi-language software vulnerability detection, 2025. URL https://arxiv.org/abs/2503.01449.

[34] X. Zhou, S. Cao, X. Sun, and D. Lo. Large language model for vulnerability detection and repair: Literature review and the road ahead, 2024. URL https://arxiv.org/abs/2404.02525.

A | Appendix A

If you need to include an appendix to support the research in your thesis, you can place it at the end of the manuscript. An appendix contains supplementary material (figures, tables, data, codes, mathematical proofs, surveys, ...) which supplement the main results contained in the previous chapters.

B | Appendix B

It may be necessary to include another appendix to better organize the presentation of supplementary material.

List of Figures

List of Tables

List of Symbols

Variable	Description	SI unit
$\boldsymbol{u}$	solid displacement	m
$\boldsymbol{u}_f$	fluid displacement	m

Acknowledgements

Here you might want to acknowledge someone.