VERCATION: Precise Vulnerable Open-source Software Version Identification based on Static Analysis and LLM

Yiran Cheng*†, Ting Zhang‡, Lwin Khin Shar‡, Shouguo Yang§, Chaopeng Dong*†, David Lo‡, Shichao Lv*†, Zhiqiang Shi*†, Limin Sun*†
* Beijing Key Laboratory of IOT Information Security Technology,
Institute of Information Engineering, Beijing, China
† School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
‡ Singapore Management University, Singapore
§ Zhongguancun Laboratory, Beijing, China
chengyiran@iie.ac.cn, lkshar@smu.edu.sg, tingzhang.2019@phdcs.smu.edu.sg, {yangshouguo, dongchaopeng}@iie.ac.cn, davidlo@smu.edu.sg, {lvshichao, shizhiqiang, sunlimin}@iie.ac.cn

***Abstract*—Open-source software (OSS) has experienced a surge in popularity, attributed to its collaborative development model and cost-effective nature. However, the adoption of specific software versions in development projects may introduce security risks when these versions bring along vulnerabilities. Current methods of identifying vulnerable versions typically analyze and extract the code features involved in vulnerability patches using static analysis with pre-defined rules. They then use code clone detection to identify the vulnerable versions. These methods are hindered by imprecision due to (1) the exclusion of vulnerability-irrelevant code in the analysis and (2) the inadequacy of code clone detection. This paper presents VERCATION, an approach designed to identify vulnerable versions of OSS written in C/C++. VERCATION combines program slicing with a Large Language Model (LLM) to identify vulnerability-relevant code from vulnerability patches. It then backtracks historical commits to gather previous modifications of identified vulnerability-relevant code. We propose code clone detection based on expanded and normalized ASTs to compare the differences between pre-modification and post-modification code, thereby locating the vulnerability-introducing commit (*vic*) and enabling the identification of the vulnerable versions between the vulnerability-fixing commit and the *vic*. We curate a dataset linking 122 OSS vulnerabilities and 1,211 versions to evaluate VERCATION. On this dataset, our approach achieves an F1 score of 93.1%, outperforming current state-of-the-art methods. More importantly, VERCATION detected 202 incorrect vulnerable OSS versions in NVD reports.**

***Index Terms*—Open-source software security, Vulnerable version, Large Language Model.**

I. INTRODUCTION

Open-source software (OSS) has become increasingly popular in recent years, thanks to its collaboration and cost-effectiveness. In the rapidly evolving world of OSS, numerous versions exist due to continuous evolution. While OSS plays a pivotal role in expediting software development, the integration of particular software versions in development projects can pose security risks, as these versions may contain vulnerabilities. Therefore, having a comprehensive knowledge of vulnerable versions of OSS becomes imperative for software developers.

Public vulnerability repositories collect vulnerability reports of software products and disseminate information regarding the affected versions of the software. The National Vulnerability Database (NVD) [1], recognized as the largest public vulnerability database, employs the Common Platform Enumeration (CPE) format to store information about vulnerable versions. However, the NVD often encompasses all versions before the reported vulnerability or designates only the versions mentioned in the report as vulnerable. For instance, CVE-2018-5785 reported only version v2.3.0 as vulnerable in CPE [2]. Yet, upon manual validation, it was uncovered that versions v2.1.1 to v2.3.0 are all susceptible to the vulnerability. Recent research [3]–[6] indicates that incomplete and incorrect information about vulnerable versions is prevalent in such reports. Therefore, there is a need for an automated method that can more accurately identify vulnerable versions of released OSS vulnerabilities.

Two limitations of current approaches. The approaches for confirming vulnerabilities involve utilizing Proof of Concept (PoC) to trigger the vulnerability. PoC triggers vulnerabilities through dynamic execution, offering conclusive evidence of their existence. However, executing a PoC for each software version is a time-intensive process that requires meticulous environment setup. Additionally, the PoC input may not be universal across all vulnerable versions [7]. Therefore, current researches use static analysis to identify vulnerable versions [8]–[15]. These methods typically consist of two steps: extracting vulnerability features from code snippets and identifying vulnerable versions through code clone detection of these vulnerability features. Existing methods typically extract vulnerability features using static analysis in which vulnerability patterns are pre-defined by human experts: some consider data and control dependencies from the patched code [12], [13]; some consider using hashes of entire functions [16]; and some consider only the patch code [14], [17]. These methods generally produce many false positives or false negatives because of the inherently imprecise nature of static analysis and the incompleteness of pre-defined vulnerability patterns, making it difficult to distinguish vulnerability-relevant code from irrelevant code (❶). The majority of the methods for code clone detection are function-level approaches [18] whereas real-world vulnerability logic is typically confined to (often a few) statements only. There are statement-level code clone detection methods but they are limited to textual signatures (such as Levenshtein algorithm [17] and hash value comparison [12], [16]), without taking into account the code changes that reorganize code while preserving its functionality. These limitations hinder the effectiveness of vulnerability detection in practical scenarios (❷).

These limitations primarily stem from the lack of reasoning about the context of the vulnerable code being analyzed. Recently, large language models (LLMs) have shown remarkable performance in various code-related tasks, including code generation [19], [20] and code summarization [21], [22]. This strong performance demonstrates LLMs' capability to recognize code patterns and correlations at both syntactic and semantic levels. As such, we aim to address the first limitation by leveraging the power of LLMs. Essentially, we hypothesize that LLMs can be leveraged to address the imprecision of static analysis and the incompleteness of human-defined vulnerability patterns (❶). However, there is a challenge of Prompt engineering in leveraging an LLM in our problem domain. The design of prompts is pivotal in directing the LLM to produce desired responses. In vulnerability comprehension tasks, well-crafted prompts should encompass the code for analysis and offer a lucid description of the analysis objectives. However, in the case of current LLMs, merely providing all the vulnerable function codes and directly instructing vulnerability analysis often yields suboptimal outcomes. When the function code is too long, the LLM's ability to understand the relationships between distant parts of the context may diminish [23], [24]. Hence, formulating prompts to guide LLMs in generating desired vulnerability logic analyses poses a challenge.

Due to variations in programming styles, variable naming, and structural differences, code clones may manifest different changes. Furthermore, developers often streamline and refactor code through method outlining. Such code structural changes often mislead code clone detection-based approaches to incorrectly identify the vulnerable versions. Furthermore, prior studies [25], [26] found that vulnerabilities can often be localized to a few key lines, making file- and function-level vulnerability detection overly coarse-grained. The limitation motivated us to propose a statement-level code clone detection approach to address the second limitation of current approaches (❷).

Our approach. We propose VERCATION[1], a novel method to identify vulnerable versions of open-source C/C++ software utilizing a symbiotic combination of static analysis, LLM, and code clone detection. Given a vulnerability fixing commit (*vfc*), VERCATION applies program slicing to extract vulnerability-related statements as vulnerability features, leverages the capability of LLM in code understanding to refine the extracted features, and performs semantic-level clone detection on vulnerability features in code changes. This hybrid method effectively overcomes the practical limitations of existing approaches. More specifically, VERCATION automatically preprocess the fixing commits and construct prompts based on the Few-shot and Chain-of-Thought (CoT) strategies, enabling the LLM to reason with the vulnerability and identify the most probable vulnerable statements as features. Subsequently, VERCATION traces earlier modifications of vulnerability features and applies a clone detection method based on expanded Abstract Syntax Trees (ASTs) to pinpoint the vulnerability-introducing commit *vic*. Finally, VERCATION identifies vulnerable versions between the *vic* and *vfc*.

Evaluation. We meticulously curated a ground-truth dataset for evaluation, encompassing 12 commonly used OSS projects, 122 Common Vulnerabilities and Exposures (CVEs), and a total of 1,211 OSS versions. This dataset comprised every patch released through Git with respect to those 122 CVEs. The first author engaged in manual vulnerability validation with the help of public Proof-of-Concept (PoC), meticulously labeling the presence of vulnerabilities across software versions. On this dataset, VERCATION demonstrated both higher precision (91.8%) and recall (94.5%) compared to state-of-the-art methods (SOTAs), including V-SZZ [17], V0Finder [16], V1SCAN [27], VERJava [28] and Vision [29]. We conducted an ablation study using three different LLMs (GPT-4 [30], CodeLlama [31], and DeepSeek-V3 [32]) to evaluate their vulnerability comprehension capabilities. Utilizing the Few-shot and CoT combined strategy, DeepSeek-V3 achieved an F1 score of 93.1%, significantly improving the F1 score of Joern parser [33] by 92.8%, a commonly-used static analysis tool. More importantly, during the evaluation, we found 202 version errors in the NVD reports. VERCATION has also shown to be efficient, analyzing each vulnerability on an average of 28.61 seconds.

Contributions. The main contributions of this paper are as follows:

- Unlike previous efforts that heavily relied on pre-defined patterns of static analysis tools, We present VERCATION, the first framework to integrate the reasoning capability of LLM for vulnerable version identification tasks, through the use of a multi-strategy universal prompt engineering.
- VERCATION presents a solution based on expanded and normalized AST to address the structural modifications in the clone detection challenge, which was designed for refactoring commit identification during vulnerability backtrack.
- We curated a dataset including 122 published CVEs containing 1,211 versions. This extensive dataset was curated across 12 OSS projects and underwent meticulous labeling through a combination of PoC input validation and manual verification.
- We have implemented a prototype of our approach and assessed its performance using our dataset, achieving the F1 score of 93.1%, improving SOTAs by 8.1% to 108.7%. We also evaluated three contrasting LLMs — a commercial,

[1] VERCATION: Vulnerable version identification

closed model (GPT-4) and open models (CodeLlama and DeepSeek-V3) — in our approach and reported that they achieved similar performances in our context when appropriate prompting strategies were applied. More importantly, by applying our approach, we have detected 202 incorrect vulnerable OSS versions in NVD reports.

The source code of VERCATION and our curated dataset are publicly available at https://github.com/Veronica-L/Vercation.

II. BACKGROUND AND MOTIVATION

In this section, we clarify the target problem, introduce the LLM and discuss the motivation with two examples.

A. Problem Statement

Commits serve as comprehensive records of OSS development, functioning as vital checkpoints for tracking the chronological evolution of code changes. They enable developers to revisit specific points on a particular date or time. This paper centers on vulnerable version identification in OSS vulnerabilities. Within the development of an OSS, certain commits introduce vulnerabilities, referred to as vulnerability-introducing commits (*vic*), which are later fixed in vulnerability fixing commits (*vfc*). The vulnerable version analysis aims to pinpoint the *vic*, allowing us to assess which versions are susceptible to the vulnerabilities.

We posit that the initial function undergoes modification by a *vic* and transforms into a vulnerable function F_v. There probably exists some subsequent commits such as feature-adding commits and refactoring commits to optimize the function code, which is denoted as F_{vr}. Ultimately, software analysts discover the vulnerability and fix the code in the *vfc*, resulting in the final function F_f. An illustrative timeline is presented in Figure 1.

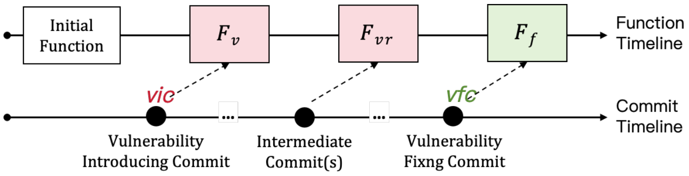


Fig. 1: Function Timeline from Vulnerability Introduction to Vulnerability Fixing.

B. Large Language Model

Large Language Models (LLMs) have revolutionized natural language processing and have demonstrated remarkable capabilities in various code-related tasks. These models, extensively trained on vast amounts of text and code repositories, can understand and generate human-like text and code. Recent advancements in LLMs, such as GPT-3 [34], GPT-4 [30], DeepSeek [35] and CodeLlama [31], have demonstrated impressive performance in code generation [19], [20], code summarization [21], [22] and program repair [36], [37].

The success of LLMs in code-related tasks can be attributed to their ability to capture complex patterns and contextual relationships in code [31], [38]. Through Pattern Recognition, LLMs learn correlations and patterns between code snippets from large-scale training data, enabling them to identify code with different syntax but similar functionality. Additionally, LLMs can infer the purpose of code snippets based on their context. This makes them particularly suitable for tasks that require a deep understanding of code structure and functionality like vulnerability detection. However, the effectiveness of LLMs heavily depends on the quality of prompts used to guide their responses. Prompt engineering, the process of designing effective prompts, has become a crucial area of research in leveraging LLMs for specific tasks [39], [40].

Despite their reasoning capabilities, LLMs also face challenges in code analysis tasks such as handling very long code sequences. Therefore, in our work, instead of blindly applying LLM as a standalone tool, we first use static analysis to extract candidate vulnerability-related codes and then leverage the semantic capturing capabilities of LLMs to improve the accuracy of extracted codes.

C. Motivating Examples

We present two examples of disclosed vulnerable code in two distinct scenarios, which motivate our work. Both of these vulnerabilities have now been effectively resolved by the development team. The key points we wish to underscore are as follows: 1) The significance of understanding vulnerability behavior in the process of discovering *vic*. 2) The challenge of accurately pinpointing *vic* due to the code structural modifications introduced by the intermediate commits.

Example 1) A vulnerability in `FFmpeg` (CVE-2017-14169 [41]) shows the limitation of existing approaches in terms of capturing vulnerability logic. The code snippet of *vfc* is shown in Listing 1. A sanitizing check was added at Line 13 for the `item_num`. Without this check, a remote attacker can make a crafted file with a large `item_num` field such as `0xffffffff`, causing a buffer overflow issue in the `avio_read` function at Line 25 and potentially leads the application to exhibit incorrect behavior or crash.

V-SZZ [17] assumes the deletion lines in the security patch as vulnerable codes and the basic idea of it is to pinpoint the earliest commit introducing the deletion lines as *vic*. However, vulnerability logic is composed of various vulnerable codes, such as Lines 21 and 25 in Listing 1. V-SZZ only considers tracing back deleted line (Line 12) and overlooks the real vulnerability behavior, leading to incorrect identification of *vic*. V0Finder [16] attempts to identify vulnerable versions using code clone detection. It generates hash values for the entire vulnerability function and identifies code clones according to the distance value of two hashes. This method may still introduce excessive vulnerability-unrelated features due to the inclusion of “entire function”.

Observation. Identification of vulnerable versions is hindered by the challenge of comprehending vulnerability logic from the security patch. LLMs can grasp contextual code semantics and achieve human-like understanding in code-related tasks, eliminating the reliance on predefined patterns and rules utilized by traditional static analysis and automated tools.

```
diff --git a/libavformat/mxfdec.c b/libavformat/mxfdec.c
@@ -493,11 +493,11 @@ static int mxf\_read\_primer\_pack
static int mxf_read_primer_pack(void *arg, AVIOContext *pb, int tag, int size, UID uid, int64_t klv_offset)
{
    MXFContext *mxf = arg;
    int item_num = avio_rb32(pb);
    int item_len = avio_rb32(pb);
    if (item_len != 18) {
        avpriv_request_sample(pb, "Primer pack item length %d", item_len);
        return AVERROR_PATCHWELCOME;
    }
-   if (item_num > 65536) {
+   if (item_num > 65536 || item_num < 0) {
        av_log(mxf->fc, AV_LOG_ERROR, "item_num %d is too large\n", item_num);
        return AVERROR_INVALIDDATA;
    }
    if (mxf->local_tags)
        av_log(mxf->fc, AV_LOG_VERBOSE, "Multiple primer packs\n");
    av_free(mxf->local_tags);
    mxf->local_tags_count = 0;
    mxf->local_tags = av_calloc(item_num, item_len);
    if (!mxf->local_tags)
        return AVERROR(ENOMEM);
    mxf->local_tags_count = item_num;
    avio_read(pb, mxf->local_tags, item_num*item_len);
    return 0;
```

Listing 1: Motivating Example of CVE-2017-14169.

```
1   diff --git a/libavformat/avidec.c b/libavformat/avidec.c
2   @@ -350,8 +350,7 @@ static void avi_read_nikon(AVFormatContext *s, uint64_t end)
3     uint16_t tag      = avio_rl16(s->pb);
4     uint16_t size     = avio_rl16(s->pb);
5     const char *name = NULL;
6     char buffer[64]   = { 0 };
7   - if (avio_tell(s->pb) + size > tag_end)
8   -     size = tag_end - avio_tell(s->pb);
9   + size = FFMIN(size, tag_end - avio_tell
10  +     (s->pb));
11    size -= avio_read(s->pb, buffer, FFMIN(size, sizeof(buffer) - 1));
12  
13  
14  /*The definition of method FFMIN in libavutil/common.h*/
15  #define FFMIN(a, b) ((a)>(b)?(b):(a))
```

Listing 2: Motivating Example of Code Refactoring Commit.

Through the integration with an LLM, we can automate vulnerability analysis, enhancing the efficiency of our approach.

Example 2) After extracting the vulnerability logic and its corresponding vulnerable statements (S_v), we can locate the *vic* by backtracing the code changes of S_v in the previous commit, thereby identifying the vulnerable versions. V-SZZ [17] identifies code clones before and after such code changes by the edit distance. However, refactoring commits, prevalent in OSS development, aim to optimize and reorganize code while preserving its functionality [42], [43].

A refactoring commit in FFmpeg (Listing 2) demonstrates how structural changes caused by method encapsulation mislead existing detection methods. In this commit, developers extracted the codes with `if` condition structure into a new method `FFMIN()` to improve code modularity. While this refactoring preserves functionality, it introduces significant syntactic divergence between the original code (Lines 7-8) and the restructured code (Lines 9-10).

Traditional tools like V-SZZ [17] consider edit distance as the metric to detect code clones. The edit distance between the original and refactored code drops to 48% due to the method outlining, causing V-SZZ to incorrectly flag this commit as the vulnerability-introducing commit (*vic*). Other approaches like ReDeBug [14] and V0Finder [16] also fail to recognize the equivalence between inline logic and encapsulated methods, as their coarse pattern matching ignores structural abstraction.

Observation. This case highlights the structural refactoring in OSS development and the limitations of existing clone detection techniques. VERCATION addresses this by expanding function calls during AST generation—inlining the `FFMIN()` method body to reconstruct the original logic. Combined with AST normalization, this allows VERCATION to detect code logic equivalence despite structural variations, accurately tracing the *vic* through refactored commits.

III. DESIGN OF VERCATION

We propose VERCATION, an end-to-end automated approach designed for vulnerable version identification in OSS vulnerabilities. The high-level workflow is illustrated in Figure 2. VERCATION consists of three phases: *Vulnerable code extraction* (P1), *Code clone detection* (P2) and *Vulnerable version range determination* (P3).

In P1, VERCATION combines program slicing and LLM to identify and extract vulnerability-related program statements in a precise manner. Specifically, we utilize the patch code as a slicing criterion to extract *dangerous flows*, defined as program statements that directly or indirectly affect the variables or expressions at the patch code. To mitigate the risk of including statements unrelated to the vulnerability (false positives), we employ prompt engineering with few-shot and Chain-of-Thought (CoT) strategies, empowering LLM to reason with the vulnerability based on the extracted dangerous flows and accurately extract vulnerability-related statements.

In P2, VERCATION retraces historical commits to collect previous modifications of vulnerable statements. For each statement before and after modification, VERCATION expands the functions within the statement, generates ASTs, and normalizes them. Then we utilize an in-order traversal algorithm to compare the AST before and after the modification as code similarity, which determines if the commit is the initial introduction of the vulnerable statements.

In P3, VERCATION identifies the affected versions based on the CVE's *vfc* and the corresponding *vic*.

To note, VERCATION supports patches that span multiple functions and files. In P1, we extract dangerous flows from each affected function independently, then combine all extracted dangerous flows as input to the LLM for unified

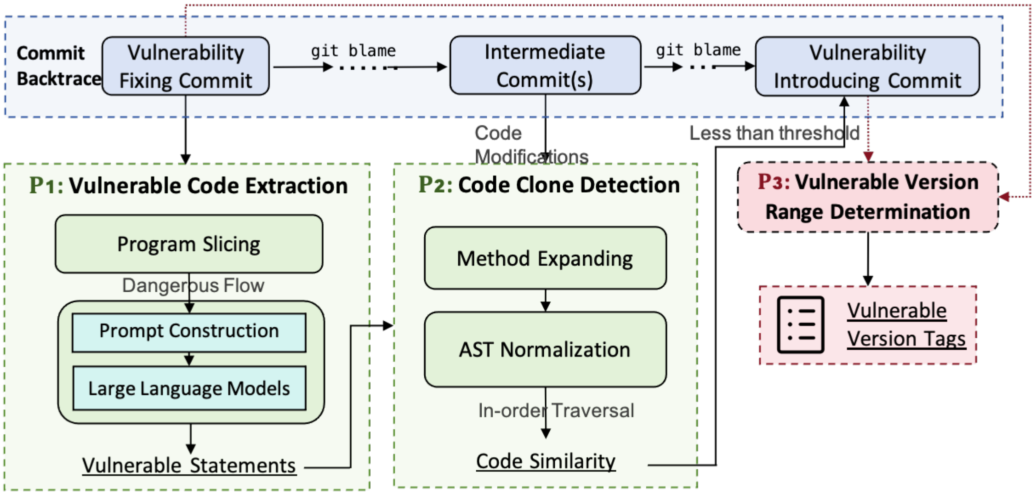


Fig. 2: VERCATION Workflow.

vulnerability logic analysis. In P2, we perform a backtrack analysis for vulnerable statements from all affected functions.

A. Vulnerable Code Extraction

Taking a *vfc* as an input, VERCATION identifies and extracts program statements related to a vulnerability in three steps, as explained in the following subsections respectively.

1) Dangerous Flow Extraction: Firstly, VERCATION conducts program slicing [44] on the source code to extract crucial program statements that may contribute to the vulnerability, which we call *dangerous flow*. The slicing is based on the following criteria: deleted/added statements and patch-related variables. Building upon existing techniques [12], [13], [15], we execute program slicing on the program dependency graph of the patched function according to the slicing criteria. We performed slicing in two directions: backward and forward. *Backward slicing* is used to trace the source of patch-related variables and *forward slicing* is used to find the trigger behavior that causes the vulnerability. For different statement types, we customize different slicing rules:

- *Assignment statement* affects the data-flow values of vulnerability behavior. We conduct normal slicing in the assignment statements and the assigned variables should be added to the slicing criterion. For example, if we take `item_num` of Line 12 and 13 in Listing 1 as slicing criterion and perform forward slicing, Line 21 and 24 are included.
- *Conditional statement* affects the reaching condition of the vulnerability trigger. We aim to slice the condition statement that takes the variables of slicing criterion as condition check, and the result also includes all the subsequent statements of condition statement (e.g., subsequent statement Line 14-15 of condition statement Line 13, subsequent statement Line 23 of condition statement Line 22).
- *Function call statement.* If the function call statement's parameters contain patch-related variables included in the slicing criterion, we conduct slicing on the statement.
- *Return statement.* There is no need for forward slicing because there is no dependency between the return value and the statements following the return statement. For example, there is no need for forward slicing on Line 15 and 23 in Listing 1.

2) Dangerous Flow-based Prompt Construction: Secondly, we leverage an LLM to refine these dangerous flows further since the dangerous flows extracted in the above step (program slicing) may include false positives (unrelated to vulnerability logic). We explore prompting strategies to assist LLM in the vulnerability comprehension task and in refining vulnerable statements from the dangerous flows extracted through program slicing. It is important to note that the LLM's role is to perform code understanding and vulnerability logic analysis rather than to recall specific CVE information from its training data. Most vulnerability reports do not provide detailed vulnerability logic analysis or identify specific vulnerable code statements [45], where the LLM's semantic analysis capabilities become essential.

Our method utilizes an LLM that supports interaction with system prompts and user prompts. The system prompt sets the role and background of the LLM. The user prompt consists of specific instructions issued by the user. At the beginning of the system prompt, we assign the role of a security researcher to the model with the statement, “You are a security researcher an expert in detecting security vulnerabilities,” and indicate that we will provide the CVE information and dangerous flow. The prompt also concludes with a declaration of the fixed output format expected for the model's response. The content of the user prompt includes detailed CVE information including CVE ID, CWE ID, CVE description, and dangerous flow extracted from program slicing. The extracted dangerous flows are formatted with line numbers prefixed to each statement when constructing prompts, such as “`4442 total_size += msec->size; 4444 stash->info_ptr_memory = (bfd_byte *) bfd_malloc (total_size);`”.

Additionally, we use the following prompt strategies to optimize the performance of LLM:

(i) Few-shot prompting is a technique where we provide the LLM with a small number of examples demonstrating the desired task before presenting the actual problem. We carefully select two examples from publicly available and verified CVE cases. To avoid data leakage, the cases are not included in our experimental dataset. The selection of examples follows specific criteria: (1) having complete and clear CVE descriptions, (2) containing explicit vulnerability fix code, and (3) featuring easily understandable vulnerability logic. These examples were created and reviewed through a systematic process: analyzing CVE descriptions and patch code, extracting key vulnerability logic, annotating relevant vulnerable statements, and writing clear vulnerability analysis explanations. The examples were created by the first author and reviewed by the fourth and fifth authors. To ensure experimental reproducibility, we maintain a fixed set of examples throughout all experiments. Each example consists of: a) A sample CVE ID, CWE ID and CVE description, b) Corresponding dangerous flow extracted from program slicing, c) The expected vulnerability logic analysis, and d) The correctly identified vulnerable statements.

(ii) Chain-of-thought (CoT) prompting provides LLM with a prompt that encourages it to generate intermediate reasoning steps before arriving at a final answer [46]. We prompt the model to "Please analyze the code following these steps: 1. Explain the vulnerability logic from the code 2. Indicate which statements are relevant to the vulnerability logic". This structured format naturally implements a chain of thought - requiring the model to first reason about and explain the vulnerability logic (the reasoning step) before identifying the specific vulnerable lines (the identification step). This approach identifies the vulnerable statements using the code understanding capability of LLM rather than superficial pattern matching. Table 3 shows our prompting template, and we take the vulnerable statements from the response as the input for the next step.

You are a security researcher and expert in detecting security vulnerabilities. I will provide you with a CVE ID, CWE ID, CVE description, and dangerous code.

`[Few-shot]`
Example 1:
Input:
CVE ID: CVE-2019-17451
CWE ID: CWE-190 Integer Overflow or Wraparound
Description: An issue was discovered in the...
Dangerous Code: <dangerous code snippet>
Output:
Vulnerability logic:
1. In the original code, there is no check for potential integer overflow...
2. If total_size overflows to a smaller value, it may lead to...
Vulnerable lines: [4442, 4444, 4459, 4460, 4461]

Example 2: ...

`[Chain-of-thought]`
Please analyze the code following these steps:
1. Explain the vulnerability logic from the code.
2. Indicate which statements are relevant to the vulnerability logic.

Provide a response only in the following format:
vulnerability logic: <text>
vulnerable lines : <Line number List>
Do not include the added line number (with +) and anything else in response.

Now, please analyze the following case:
<CVE ID>,<CWE ID>,<CVE description> and <dangerous flows>

Fig. 3: The LLM Prompt Structure.

B. Code Clone Detection

Given the vulnerable statements S_v of a fixing commit (extracted in Section III-A), VERCATION performs a commit backtrack to identify which previous commit introduced S_v by tracing the history of earlier modifications. To achieve this, VERCATION employs the `git blame` command to backtrack through previous commits. Figure 4 illustrates our code clone detection process. For each intermediate commit between the *vfc* and the potential *vic*, we compare the post-modification statements (S) with the pre-modification statements (S') using two levels of similarity detection: a quick syntactic check using edit distance and a deeper structural analysis using ASTs. The results from both comparisons, combined with statement weights assigned in Section III-B3, contribute to a final similarity score. If this score is below the threshold θ_3, we identify the commit as the *vic*. Otherwise, we continue the backtrack process.

1) Syntactic Similarity Analysis: First, we use line mapping to pre-filter highly similar lines in S' with S, where $S \subseteq S_v$. If such lines are found, it indicates that the commit is not the first to introduce S_v. We use the edit distance to calculate line similarity $\text{Similarity}_A(S'_i, S_i)$, where $S'_i \in S'$ and $S_i \in S$. We set a similarity threshold ϑ_1; if $\text{Similarity}_A(S'_i, S_i) \geq \vartheta_1$, we continue to backtrack the previous commit. If $\text{Similarity}_A(S'_i, S_i) < \vartheta_1$, we proceed to the more detailed structural analysis. To note, when encountering merge commits with multiple parents during backtrack, we traverse each parent path independently and select the chronologically earliest *vic* across all branches.

2) Structural Similarity Analysis: As exemplified in Example 2 of Section II-C, Lines 5-6 (deleted pre-commit lines) and 7-8 (added line of post-commit, e.g., included in S_v) exhibit high similarity in code behavior. However, previous research failed to identify this high similarity through simple line mapping, resulting in the commit (version) being erroneously labeled as an introducing commit (vulnerable version). We propose a fine-grained similarity comparison method at the AST-level to address this limitation. AST is a tree representation of code that preserves well-defined components of statements, explicit statement order and the execution logic [47].

Method Expansion. Due to code refactoring often outlining code into a new method, as shown in Listing 2, the codes in Lines 7-8 are outlined into methods `FFMIN`. The standard AST comparison treats the if-condition (Lines 7-8) as completely different from the FFMIN function call, resulting in a false vulnerability identification. Therefore, we utilize the inline technique to expand the method during AST generation to gather more code behavior within the statements. This involves obtaining the method definition at the callsite and expanding the method body inline, resulting in a more comprehensive code representation. We get the AST for the target file using *Clang* without performing actual compilation. Furthermore, through recursive retrieval of line numbers for sub-ASTs, we

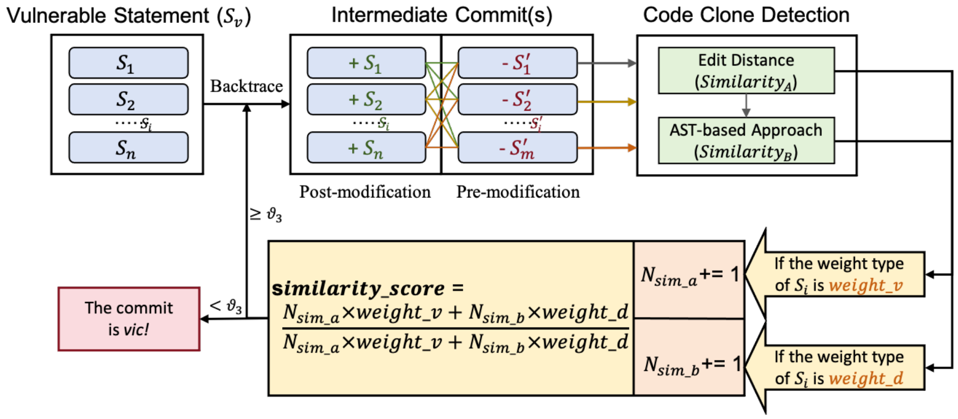

Vulnerable Statement (S_v)
Intermediate Commit(s)
Code Clone Detection
S_1
S_2
$\cdots S_i$
S_n
Backtrace
$+S_1$
$+S_2$
$\cdots S_i$
$+S_n$
$-S'_1$
$-S'_2$
$\cdots S'_i$
$-S'_m$
Edit Distance ($Similarity_A$)
AST-based Approach ($Similarity_B$)
Post-modification
Pre-modification
$\geq \vartheta_3$
$< \vartheta_3$
The commit is vic!
$similarity\_score = \frac{N_{sim\_a} \times weight\_v + N_{sim\_b} \times weight\_d}{N_{sim\_a} \times weight\_v + N_{sim\_b} \times weight\_d}$
N_{sim_a}+= 1
N_{sim_b}+= 1
If the weight type of S_i is $weight\_v$
If the weight type of S_i is $weight\_d$

Fig. 4: Overview of Code Clone Detection Process in VERCATION.

can locate the sub-AST corresponding to S_v.

AST Normalization. To address structural variations arising from different developer coding styles, it is essential to normalize the AST. This process mitigates inconsistencies and reduces analytical noise by standardizing code representations. AST normalization involves conditional structure normalization, loop structure normalization, and relational operation normalization.

- *Conditional structure* can be implemented by *if-else* and *switch-case*. We normalize the structure of *switch-case*: `switch(expression){case value1: statement1; break; case value2: statement2; break; default statement3;}` into a *if-else* structure: `if(exp == value1) statement1; else if(exp == value2) statement2; else statement3`.
- *Loop structure* can be represented as *while*, *do-while* and *for* structures, we normalize all the loop structure to *while* [48]. The *for* structure is `for(initialize; condition; increment) {statements;}`, we transform it into *while* structure: `initialize; while(condition) {statements; increment}`.
- *Relational operation* include commutative and noncommutative operation. Commutative relational operation (i.e., +, !=, &&) refers to the operation where commutate the operands does not change the operation semantics. We normalize the two operands (left and right child nodes) of the operation (parent nodes) by alphabetical order. For example, we transform $b + a$ (b is the left child node) to $a + b$ (a is the left child node). Noncommutative relational operation (i.e., $>$, $<=$) will change the operation semantic when commutating the operands, we normalize them by predefined rules. For example, we transform $b < a$ to $a > b$ (normalize operation $<$ to $>$).

AST Similarity. Ultimately, we utilize an in-order traversal to ascertain whether the commit introduces S by comparing the AST similarity of S_i and S'_i. First, we perform an in-order traversal of the ASTs for S_i and S'_i, converting the results into sequences Sq_i and $S'q_i$. Then, we calculate the edit distance between the two sequences as their AST-based similarity $\text{Similarity}_B(Sq_i, S'q_i)$. We set a similarity threshold ϑ_2, and if $\text{Similarity}_B(Sq_i, S'q_i) \geq \vartheta_2$, S_i and S'_i are considered to be AST-based similar. If $\text{Similarity}_B(Sq_i, S'q_i) < \vartheta_2$, it indicates that there is no statement similar to S_i in the pre-modification commit. During this process, we classify a commit as a refactoring commit when it exhibits low edit-distance similarity ($\text{Similarity}_A(S_i, S'_i) < \vartheta_1$) but high AST-based similarity ($\text{Similarity}_B(Sq_i, S'q_i) \geq \vartheta_2$). This indicates that while the code structure has been significantly modified, the underlying functionality remains unchanged.

3) Statement Weight Allocation: Not all statements identified by an LLM contribute equally to vulnerability in triggering the exploit. To address this, we propose a weight allocation strategy that prioritizes vulnerable statements based on their invocation of known sensitive functions. The core premise is that the presence of specific, high-risk functions within a statement is a strong indicator of its criticality. We followed a systematic process to identify and categorize these sensitive functions. Starting with a comprehensive review of previous studies [49], [50], we collected existing sensitive function enumerations. We then conducted a preliminary study of common C/C++ vulnerabilities to identify the most frequent vulnerability categories, such as buffer overflow, integer overflow, and use-after-free, and examined the functions commonly involved in known vulnerabilities for each type. This process involved three authors with security expertise: the first author proposed the initial categorization, which was then independently reviewed and validated by the fourth and fifth authors, with any discrepancies resolved through discussion. The result is the pre-defined table (Table I) of sensitive functions for various vulnerability types. We analyze each vulnerable statement extracted by the LLM. If a statement contains a direct call to any function listed in our sensitive function table, it is assigned a higher weight, marking it as a probable vulnerability trigger. For instance, in Listing 1, Lines 21 and 25 are critical trigger lines.

Furthermore, we conduct an inter-procedural analysis to identify customized functions that serve as variants of canonical sensitive functions. For instance, the `av_calloc` function on Line 21 is a variant of the `calloc` function. To discern whether the customized function `F` contains sensitive

TABLE I: Sensitive Functions for Some Vulnerability Types.

Vulnerability Type	Sensitive Function
Buffer overflow	strcpy, strncpy, memcpy, memset, read, write, gets, gets_s, strcat
Integer overflow	add, multiple, bit-shifting, memcpy
Null pointer derefrence	malloc, calloc, realloc, strdup
Pointer out-of-bounds access	memcpy, memmove
Use after free	free, malloc, calloc
Format string	sprintf, printf, scanf
Arbitrary command execution	fopen, popen, system

functions, we first analyze the definition and implementation code of function `F`, then check if `F`'s function body calls any sensitive functions from Table I. If such call relationships are found, `F` is also identified as a sensitive function. Taking Listing 1 as an example, considering this is an overflow-type vulnerability, we recognize the `av_calloc` in Line 21 and `avio_read` function in line 25 as sensitive functions.

Parameters. There are two parameters for the weight allocation of statements, i.e., $W_{sensitive}$ and W_{base}, that could affect the effectiveness of VERCATION: 1) W_{base} is the base weight for initializing the weights of all statements. 2) $W_{sensitive}$ is the weight for the statements containing sensitive functions. We set $W_{sensitive} = 2 \times W_{base}$, $W_{base} = 1.0$, the detailed weighting configuration experiment is in Section V-B.

4) Code Clone Determination: We calculate the *similarity_score* for each code modification of the previous commit with the backtrack. We set a threshold ϑ_3, if $similarity\_score < \vartheta_3$, we determine that there is no code clone between the pre-modification code of this previous commit and the vulnerable statements, thus confirming that this commit introduced the vulnerability. Otherwise, we proceed with the commit backtrack process.

$$similarity\_score = \frac{sim\_a \times W_{sensitive} + sim\_b \times W_{base}}{a \times W_{sensitive} + b \times W_{base}}$$

where, $W_{sensitive}$ and W_{base} are predefined weights assigned to different types of statements; a and b are the total counts of the statements in S_v corresponding to these weight types; sim_a and sim_b are the counts of statements that were found to be similar, either through syntactic or structural analysis.

C. Vulnerable Version Range Determination

To precisely determine the range of software versions affected by a vulnerability, we employ a methodology based on the principle of commit reachability. The core premise is that a vulnerability, once introduced in a specific *vic*, is propagated to all subsequent commits in its lineage until it is fixed by a *vfc*.

The basis for our analysis is the concept of reachability, where a commit `A` is considered reachable from another commit `B`, if commit `A` is an ancestor of commit `B` in the project's commit graph [17]. As version tags are essentially pointers to specific commits, they inherit the reachability properties of the commits they represent. This allows us to map the vulnerability's presence from the commit level to software versions.

Our methodology formally defines the set of vulnerable versions (V_v) as follows: First, we identify V_i, the set of all version tags reachable from *vic*. Second, we identify V_f, the set of all version tags reachable from *vfc*. The vulnerable version set V_v is then derived from the set difference between these two sets: $V_v = V_i - V_f$.

We illustrate the procedure by the case of `CVE-2021-20294` in the Binutils project [51]. For this vulnerability, the set of versions containing the *vic* is [`binutils-2_35`, `binutils-2_42`], while the set of patched versions containing the *vfc* is [`binutils-2_36`, `binutils-2_42`], with `binutils-2_42` being the latest version of the Binutils project. Therefore, we consider the version tags [`binutils-2_35`, `binutils-2_36`) as vulnerable. This notation signifies a range that includes `binutils-2_35` but excludes `binutils-2_36`.

IV. EXPERIMENTAL SETUP

A. Dataset

With confirmation from the author of V-SZZ, it is established that V-SZZ assumes the deleted lines in the *vfc* as vulnerable codes and manually labels the first introduction of deleted lines in a *vfc* as the *vic* to construct the dataset. However, as mentioned in Section II-C, deleted lines cannot fully capture the vulnerability logic. Additionally, it cannot be guaranteed that the identified software version will necessarily exhibit the vulnerability. Meanwhile, V-SZZ excludes the *vfc*s that only contain added lines of code in their vulnerability selection, as the SZZ algorithm relies on tracking deleted lines of code to locate *vic* and thus cannot handle such cases. Therefore, we need to construct a ground-truth dataset with labeled vulnerable version or non-vulnerable version of OSS, which are verified by a public PoC.

Vulnerabilities. To build a reliable evaluation dataset, we formulate four vulnerability selection criteria: 1) The vulnerability must be a CVE reported in an OSS project; 2) The vulnerability must have publicly available PoCs and patches; 3) The PoC must be reproducible across different versions of the OSS and 4) The OSS must have a certain number of versions for comprehensive analysis. Based on the selection criteria, we established a dataset comprising 122 publicly disclosed CVEs associated with public PoC and patches, spanning 12 prevalent OSS, as detailed in Table II. The published date of these CVEs from 2016 to 2025. Meanwhile, the collected vulnerabilities cover 13 common CWE types. Some vulnerabilities are classified into multiple CWEs. For example, CVE-2023-1579 belongs to both CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) and CWE-787 (Out-of-bounds Write). Among the dataset, vulnerabilities belonging to CWE-787 (Out-of-bounds Write) are the most numerous, accounting for 20.27%.

Patches. We acquired security patches by crawling *vfc* from the OSS Git repositories. In total, the *vfc*s cover 2,287 modifi-

TABLE II: Ground-truth Dataset Overview.

IDX	Name	#Version	#CVE	Domain
1	Binutils	157	18	Programming tools
2	cJSON	48	3	JSON parser
3	FFmpeg	405	31	Multimedia processing
4	Jasper	99	12	Image coding toolkit
5	Libarchive	51	10	Streaming processing
6	Libcaca	7	3	Graphics library
7	Liblouis	69	6	Braille translator
8	Libming	14	9	Flash library
9	Libtiff	78	7	Image tools
10	Libxml2	233	12	XML toolkit
11	OpenJPEG	25	8	Image codec
12	Pcre2	25	3	Regular expression engine
Total	-	**1,211**	**122**	-

cation lines containing 1,532 insertion lines and 755 deletion lines. Among these, 289 lines are unrelated to vulnerability (e.g., modifications in the ChangeLog file). On average, each *vfc* has 21.8 modification lines. The number of added lines significantly exceeds that of deleted lines, underscoring the critical importance of incorporating additional code into the analysis process. Among these, 10.3% are large *vfc*s (the number of modification lines exceeding 50). Notably, 79 *vfc*s (64.75%) contain both code insertion and deletion, 29 *vfc*s exclusively insertions (23.77%), and 15 *vfc*s solely involve deletions (12.30%).

Verifying & Labeling. We apply the following process to systematically label the ground-truth dataset:

(i) The fixed version can be determined through the official release of *vfc*s. We confirm whether versions following the fixed version are vulnerable by checking if they contain the patch code.

(ii) For the remaining version, we obtained the public PoC of each vulnerability from the Git issues or by referring to OSS sites.

(iii) To validate the PoC on each corresponding OSS version, we establish the dependent environment for each OSS version and compile the libraries.

(iv) During the PoC input testing process for each vulnerability, we analyze the triggering conditions and dangerous behaviors of the vulnerability and label the statements that trigger the vulnerability as $S_{trigger}$. For versions where the PoC cannot trigger the vulnerability (as some PoCs are only applicable to certain specific versions), we check whether the version contains $S_{trigger}$ to label the version.

In our experimental setup, we took 10 days to build the software and version pool. Then we took 30 days to build the ground-truth dataset, including the time taken to establish the dependent environment for each software version, as well as the time to perform vulnerability verification and labeling.

B. Baseline and Metrics

Baseline. For the vulnerable version identification task, we selected the NVD [52], along with SOTAs: SZZ-based methods (AG-SZZ [53], B-SZZ [54], V-SZZ [17]) and clone-based methods (V0Finder [16], V1SCAN [27], VERJava [28] and Vision [29]) for accuracy comparison. We gathered the vulnerable version range of each CVE from CPE to evaluate the NVD's accuracy.

The basic idea of SZZ algorithms is to backtrack the commit history to locate the earliest commit that introduces the deletion lines removed by security patches. We use different SZZ algorithms to identify the *vic*, with the versions between the *vic* and the *vfc* considered vulnerable. Since these algorithms trace deleted lines individually, instances may arise where multiple *vic* are identified, and in such cases, we select the earliest one as the detection outcome.

V0Finder, V1SCAN, VERJava, and Vision are clone-based methods. We use their techniques to detect the code clone for each version and identify the affected versions of the vulnerabilities in the ground-truth dataset. Because VERJava and Vision are applied in Java, we modified the code to make them compatible with C/C++.

For VERCATION, we selected thresholds $\vartheta_1 = 0.9$, $\vartheta_2 = 0.8$ and $\vartheta_3 = 0.7$ (related experiments are introduced in Section V-C). In terms of the performance of vulnerability code extraction, we compared the differences between combining various LLMs (GPT-4, CodeLlama-13B, DeepSeek-V3) and using the static analysis tool Joern parser [33] alone.

Metrics. To evaluate the accuracy, we employ the following three metrics, i.e., true positives (TP), false positives (FP), false negatives (FN), precision ($\frac{\#TP}{\#TP+\#FP}$), recall ($\frac{\#TP}{\#TP+\#FN}$) and F1-score ($\frac{2*precision*recall}{precision+recall}$), to measure the accuracy of the above methodologies. These metrics are also used in previous studies to evaluate the technique performance [9], [17]. Because of significant differences in the version count for distinct OSS, we compute the precision and recall for each OSS and report the average of precisions and recalls finally.

C. Implementation Details

We utilized Joern [33] for program slicing and developed Python scripts to extract control and data dependencies, outputting the sliced statements as dangerous flows. We explored three LLM models, GPT-4 [30], CodeLlama-13B [31], and DeepSeek-V3 [32] for vulnerable code extraction. We use the public API developed by OpenAI to perform the experiment in GPT-4. The API version is `GPT-4-0613` published in 2023. CodeLlama was created through further fine-tuning of Llama 2 on specific code datasets. We selected the 13B parameter model (`codellama/CodeLlama-13b-hf`) from the CodeLlama model series. The API of DeepSeek-V3 was published in July 2024. We set the temperature parameter to 1.0. To account for the stochastic nature of LLM outputs, we generate 10 independent responses for each CVE [55]. We employ a majority voting strategy where statements appearing in ≥ 6 out of 10 runs are included in the final vulnerable statement set for evaluation. The maximum size of tokens is 1,024. We deployed the Codellama model on our server with four NVIDIA RTX A5000 GPUs. Similar to GPT-4, the temperature parameter is set to 1.0, and the maximum size of tokens is set to 1,024. Subsequently, we apply weight allocation strategies to extract vulnerable statements using Python scripts. For threshold selection, we used $\theta_1 = 0.9$, $\theta_2 = 0.8$ and $\theta_3 = 0.7$ based on grid search optimization described in our sensitivity analysis. In the code change detection, we utilize the Clang 10.0.0 tool for function expansion (inline optimization) and AST generation of C/C++ code. Specifically, we use the *-fsyntax-only* option to perform syntax checking without actual compilation. We then develop Python scripts to achieve AST normalization and similarity comparison. Overall, we constructed our system with 6K LoC in Python.

V. Evaluation

In this section, we evaluate the performance of the proposed VERCATION by answering the following research questions (RQs).

- **RQ1.** *Overall Effectiveness:* How does the overall performance of VERCATION compare against state-of-the-art vulnerable version identification methods?
- **RQ2.** *Architectural Contribution:* How do the key architectural choices in VERCATION—the use of an LLM, static analysis, and statement weighting—contribute to its performance?
- **RQ3.** *Component Robustness:* How accurate and robust is VERCATION's statement-level code clone detection?
- **RQ4.** *Real-World Application:* What are the effects of applying VERCATION in the real world scenarios?

A. RQ1: Overall Effectiveness

We first compare VERCATION against the NVD and SOTA methods. Table III shows the vulnerable version identification results of NVD, SZZ algorithms, V0Finder, V1SCAN, VERJava, Vision, and VERCATION. Note that some CVEs do not provide vulnerable versions in NVD, such as CVE-2021-30499 [56]. SZZ algorithms cannot work on the *vfc*s containing solely added lines (i.e., 11 cases), so we exclude these cases in approaches using the SZZ algorithm.

1) Comparison with NVD: The NVD achieves a precision of 66.8% and a recall of 41.8% . False positives often occur because NVD may broadly flag all versions preceding a patch as vulnerable without specific analysis. This situation exists in 18 CVEs (15%) of the ground truth. For instance, security experts found a buffer overflow risk exists in Libtiff 4.4.0, but NVD simply reported the versions of Libtiff<4.4.0 were exposed in vulnerability. In reality, the vulnerable version range is Libtiff<4.4.0 and ≥4.0.0.

The reason for low recall in NVD can be summarized into two types: (i) NVD only reported the version in which vulnerabilities were found. For example, CVE-2017-14152 [57] was found in OpenJPEG 2.2.0, then NVD reported the affected version was only 2.2.0, whereas the real vulnerable version range was 2.2.1-2.2.0. The problem exists in 57 CVEs (47%) of the ground truth. (ii) Some vulnerabilities are not fixed promptly after disclosure but are addressed after several versions have been released. These vulnerable versions can be overlooked if NVD does not update the version information. For example, CVE-2021-33815 [58] was discovered in FFmpeg 4.4 but was fixed in version 5.0. Due to the lack of timely tracking of the fix information, NVD overlooked the vulnerable versions 4.4.1-4.4.4.

TABLE III: Comparison with NVD Database and SOTA works.

Type	Methods	Precision	Recall	F1 score
-	NVD Database	0.668	0.418	0.514
SZZ Algorithm	AG-SZZ [59]	0.372	0.625	0.466
	B-SZZ [54]	0.378	0.543	0.446
	V-SZZ [17]	0.756	0.851	0.801
Clone-based Approach	V0Finder [16]	0.829	0.745	0.785
	V1SCAN [27]	0.863	0.724	0.788
	VERJava [28]	0.891	0.247	0.386
	Vision [29]	0.842	0.881	0.861
-	VERCATION	**0.918**	**0.945**	**0.931**

2) Comparison with SZZ-based Methods: The basic idea of SZZ algorithms is to backtrack the commit history to locate the earliest commit that introduces the deletion lines removed by patches. Regarding F1-score, VERCATION improves the best-performing baseline V-SZZ by 16.2%. And the difference between the F1 of B-SZZ and AG-SZZ is small (0.446 vs. 0.466). We affirmed that there are two main causes of false alarms in these approaches:

(i) The line mapping algorithm based on edit distance does not consider the behavior of the source code. As shown in Listing 2, The deleted lines were encapsulated within the added functions, and in fact, they have the same behavior. Line mapping algorithms failed in these cases, unable to identify the true *vic*.

(ii) The SZZ algorithm treats each deleted line as an independent origin for backtracking, resulting in multiple *vic* being identified. For example, the security patch of CVE-2020-35965 [60] from FFmpeg has 5 deleted lines. The true vulnerability logic lies in the lack of checking the size of ymax before executing the memset zero operation, potentially resulting in out-of-bounds writes to memory. Our method utilizes a weighted allocation approach to increase the weight of memset operation, thus backtracing to the correct *vic*. However, SZZ algorithms separately backtrack different deleted lines, resulting in the identification of 5 different *vic*, significantly reducing the precision.

3) Comparison with Clone-based Methods: In the same dataset, V0Finder achieved a precision of 82.9% and a recall of 74.5%. There are two reasons for the inaccuracies of V0Finder: 1) V0Finder generates a hash value for the entire vulnerable function as a vulnerability fingerprint. This fingerprint is coarse because the vulnerable function contains a lot of vulnerability-irrelevant code, introducing a significant amount of noise. 2) V0Finder considers the function containing all deleted patch code as a vulnerable function clone. This approach also overlooked the inserted code of the security patch.

V1SCAN achieves an F1-score of 0.788, combining version-based detection with code-based detection methods. For version-based detection, V1SCAN obtains CVE-affected

OSS versions from NVD's CPE database and filters out functions belonging to versions outside the CVE's affected range. As we mentioned earlier, there are many errors in NVD's affected version information, which consequently affects V1SCAN's detection results. For code-based detection, V1SCAN primarily relies on text similarity comparisons, overlooking deeper relationships between code snippets. When functions are refactored or outlined into new functions, this approach fails to identify vulnerability-related code, resulting in false negatives.

Table III reports a low recall of 0.247 in VERJava, VERJava has a fundamental issue in determining vulnerable versions: it requires the target version's code to simultaneously meet two strict conditions - perfect matching of deleted lines in the *vfc* (delSim $\geq$ 1.0) and the absence of most added patch code (addSim $\leq$ 0.9). In project development, code frequently undergoes changes, including variable name modifications and function refactoring. This means that even versions that contain vulnerabilities are unlikely to meet such strict matching requirements due to code evolution. As a result, many vulnerable versions are incorrectly identified as safe versions, leading to serious false negatives. Another problem with VERJava is similar to V0Finder, it incorrectly assumes that deleted lines in the patch represent vulnerable features.

Vision achieves an F1 of 0.861, combining critical statement selection with weighted dependency graphs. Vision relies on taint analysis as a first step for critical statement selection. However, this approach often includes many statements that are control or data-dependent but not actually relevant to the vulnerability, which consequently affects Vision's precision in extracting vulnerable code. Moreover, Vision's analysis is limited to Java packages from Maven repositories, which may not fully capture the vulnerability patterns in other languages.

Accuracy of VERCATION. VERCATION demonstrates consistent performance across most projects, achieving high precision ($>$ 0.90) on 9 out of 12 projects and perfect recall (1.00) on several projects including cJSON and Libcaca. Notably, while baseline methods like V-SZZ and V0Finder show significant performance fluctuations across different projects, VERCATION maintains more stable performance. For instance, on the Binutils project, which has complex version management and frequent refactoring, Vercation achieves a 0.95 F1 score, outperforming V-SZZ (0.47 F1 score) and V0Finder (0.88 F1 score).

Though VERCATION performed well in the version identification, it still failed to identify inducing commits for a small number of vulnerabilities. One reason is that in some cases, slight modifications in the AST have little impact on the similarity between the AST before and after the modification, but are crucial in introducing vulnerabilities. Another reason is that despite using the weight allocation strategies, there are still some statements related to patch variables that are not relevant to the vulnerability but have been assigned high weights. The above two reasons may incorrectly identify *vic*, leading to a misidentification of the vulnerable version range.

In our evaluation, 5.7% of traced commit histories involved merge commits with multiple parents. For example, in CVE-2022-1355 [61] from Libtiff, the vulnerability was introduced in a feature branch that was later merged into the main development line, creating a non-linear history with two parent commits at the merge point. For these cases involving non-linear histories, VERCATION achieved a precision of 91.2% and recall of 92.5%, demonstrating that our multi-path traversal strategy effectively handles branching scenarios without significant performance degradation.

TABLE IV: Performance of LLMs in Vulnerability Logic Comprehension.

Method (Tool)	Avg #Vuln Stmts[1]	Precision	Recall	F1
Static Analysis (Joern)	22.64	0.435	0.544	0.483
LLM (DeepSeek)	11.33	0.742	0.796	0.768
Static Analysis + LLM (Joern + DeepSeek)	5.66	**0.918**	**0.945**	**0.931**

[1] The average count of extracted vulnerable statements for each CVE.

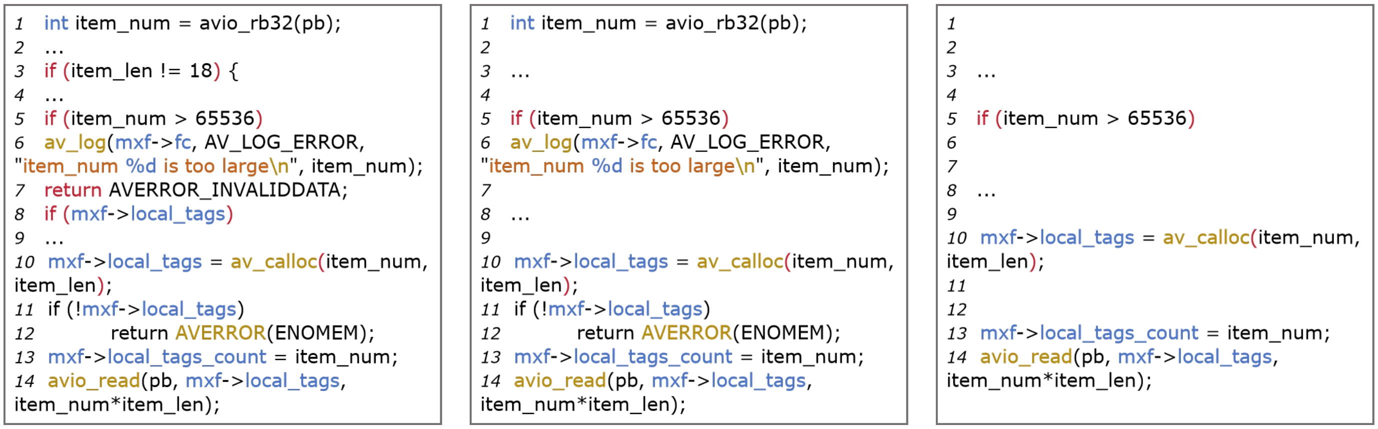
RQ1: VERCATION demonstrates superior performance in vulnerable version identification on our ground-truth dataset, achieving an F1 score of 93.1%. This significantly outperforms SOTA works. The prompt strategy combining Few-shot and CoT techniques performs the best.

B. RQ2: Architectural Contribution

1) Program Slicing & LLM Ablation Study: We conducted ablation experiments by separately removing the program slicing component (implemented by Joern) and the LLM component (DeepSeek) from phase 1.

Static Analysis Only. As depicted in Table IV, the effectiveness of Joern in extracting vulnerability logic is significantly inferior to that harnessed by the power of LLM models, with an F1 score 46.9% less than Joern + DeepSeek (Few-shot + CoT). This discrepancy primarily stems from Joern's reliance on fixed dependency extraction patterns, leading to the retrieval of numerous statements irrelevant to vulnerabilities. As shown in Figure 5a, using only the static analysis tool Joern parser for program slicing extracts statements on Lines 3, 7-8, 11-12. While these statements are related to control flow, they are not directly relevant to the vulnerability. This illustrates how static analysis alone can include extraneous information that is not crucial to understanding the specific vulnerability logic. Meanwhile, we found that Joern cannot accurately address pointer structure dataflows, thus missing some statements that have data or control dependencies. To elaborate, on average, Joern extracts 3.64 times more vulnerable statements per CVE compared to combined DeepSeek.

LLM Only. The approach of LLM Only (DeepSeek) involves inputting the entire function into the model for analysis. As illustrated in Table 5b, DeepSeek alone (with Few-shot + CoT strategies) achieves an F1 score of 0.739, which is 18.7% lower than the Joern + DeepSeek combination (0.926). This performance gap primarily stems from LLM's tendency to include a broader range of contextual information when given an entire function as input, this over-inclusion of context



```
1  int item_num = avio_rb32(pb);
2  …
3  if (item_len != 18) {
4  …
5  if (item_num > 65536)
6  av_log(mxf->fc, AV_LOG_ERROR,
"item_num %d is too large\n", item_num);
7  return AVERROR_INVALIDDATA;
8  if (mxf->local_tags)
9  …
10  mxf->local_tags = av_calloc(item_num,
item_len);
11  if (!mxf->local_tags)
12          return AVERROR(ENOMEM);
13  mxf->local_tags_count = item_num;
14  avio_read(pb, mxf->local_tags,
item_num*item_len);
```

(a) Extracted by Using Only Joern Parser.

```
1  int item_num = avio_rb32(pb);
2
3  …
4
5  if (item_num > 65536)
6  av_log(mxf->fc, AV_LOG_ERROR,
"item_num %d is too large\n", item_num);
7
8  …
9
10  mxf->local_tags = av_calloc(item_num,
item_len);
11  if (!mxf->local_tags)
12          return AVERROR(ENOMEM);
13  mxf->local_tags_count = item_num;
14  avio_read(pb, mxf->local_tags,
item_num*item_len);
```

(b) Extracted by Using Only LLM.

```
1
2
3  …
4
5  if (item_num > 65536)
6
7
8  …
9
10  mxf->local_tags = av_calloc(item_num,
item_len);
11
12
13  mxf->local_tags_count = item_num;
14  avio_read(pb, mxf->local_tags,
item_num*item_len);
```

(c) Extracted by Combining Joern Parser and LLM.

Fig. 5: The Vulnerable Statements Extracted by Different Methods.

TABLE V: VERCATION with Different Strategies.

Model	Strategy	Precision	Recall	F1 score
GPT-4	Zero-shot	0.708	0.823	0.761
	Few-shot	0.892	0.907	0.899
	Few-shot + CoT	0.893	0.946	0.925
CodeLlama	Zero-shot	0.671	0.789	0.725
	Few-shot	0.851	0.895	0.873
	Few-shot + CoT	0.842	0.912	0.876
DeepSeek-V3	Zero-shot	0.769	0.827	0.797
	Few-shot	0.882	0.921	0.901
	Few-shot + CoT	**0.918**	**0.945**	**0.931**

TABLE VI: LLM Consistency for Vulnerable Statement Extraction.

Model	Agreement Rate	Jaccard Similarity
GPT-4	0.454	0.439
CodeLlama	0.517	0.566
DeepSeek-V3	**0.586**	**0.629**

leads to lower precision because some of which are tangential to the core vulnerability logic. As illustrated in Figure 5b, using only LLM improves upon static analysis by reducing the extraction of irrelevant control flow statements. However, when given the entire function as input, the LLM struggles to precisely differentiate between core vulnerability-related code and contextual information. This results in the inclusion of some noise, such as variable definitions (Line 1) and unrelated control flow statements (Lines 4, 7-8).

As shown in Figure 5c, the combination of Joern's structured analysis with DeepSeek's code understanding allows for a more focused and accurate extraction of vulnerability-related statements. Joern provides an initial set of potentially relevant statements based on program structure, which DeepSeek then refines using its contextual understanding, resulting in a more balanced and effective approach to vulnerability logic comprehension.

2) LLM Performance, Consistency, and Generalization: We further evaluated the LLM component by comparing different models and analyzing their stability and ability to generalize.

Comparative Performance. Table V illustrates the performance of combining the dangerous flows extracted by the Joern parser with LLMs on our dataset, with the "Strategy" column delineating different prompting strategies. Our observation reveals that DeepSeek-V3 prompted with Few-shot and CoT surpasses other models. Specifically, DeepSeek-V3 (Few-shot + CoT) exhibits a 0.64% and 6.28% improvement in F1-score compared to the GPT-4 and CodeLlama, respectively. Moreover, the performance trends with DeepSeek remain consistently upward across different prompting strategies. When prompted with the Few-shot prompt, DeepSeek achieves a 13.15% improvement in F1-score over the Zero-shot prompt. This is further improved by 3.33% F1 with the addition of CoT prompting. We observed that the utilization of the Zero-shot prompt often results in an increased generation of false positives (vulnerable statements), including the statements predefining patch variables (e.g., Line 6 in Listing 1). Conversely, in examples prompted by Few-shot, we prioritize understanding the triggering mechanisms of vulnerabilities, thereby facilitating LLM in learning a more accurate method of generating vulnerability logic.

LLM Consistency Analysis. To evaluate the stability of our LLM-based vulnerable statement extraction, we measured two consistency metrics across 10 independent runs with temperature = 1.0. The agreement rate represents the percentage of statements that appeared in all 10 runs relative to the total unique statements extracted across all runs. As shown in Table VI DeepSeek-V3 achieved the highest agreement rate 58.6%, followed by CodeLlama (51.7%). For cases with variation, we calculated average Jaccard similarity by computing pairwise similarities between all run combinations and taking the mean. DeepSeek-V3 achieved the highest Jaccard similarity of 0.629 shown in Table VI, indicating that DeepSeek-V3 maintains the most consistent performance even when exact matches were not achieved. While the consistency levels are moderate across all models, this reflects the inher-

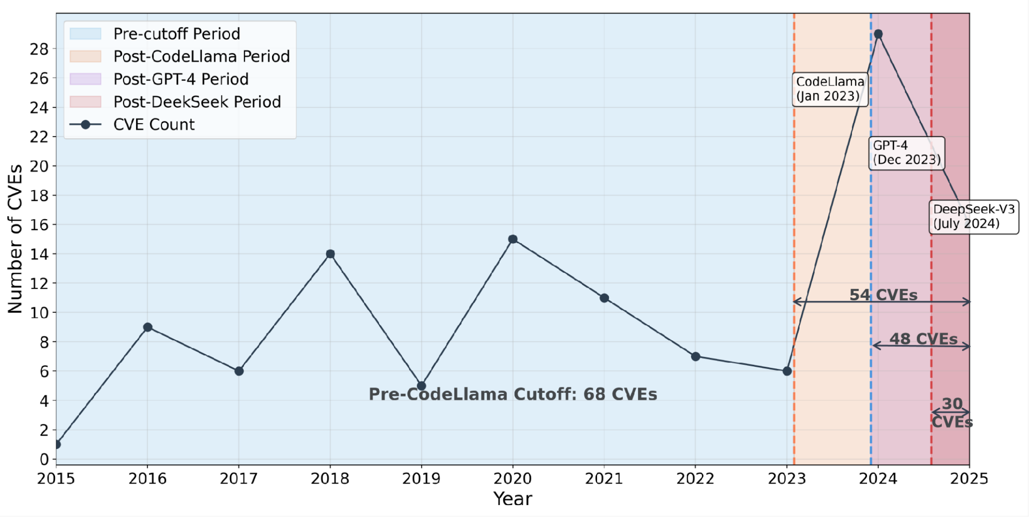


Fig. 6: Temporal Distribution of CVEs with Knowledge Cutoff Boundary.

ent complexity of vulnerability logic analysis where slight variations in reasoning can lead to different but potentially valid interpretations. The superior consistency of DeepSeek-V3, combined with its highest F1-score performance, makes it the most reliable choice for practical deployment.

Generalization Capability. A critical concern regarding the LLM-based approach is whether the models' knowledge cutoff affects performance when analyzing vulnerabilities published after the training data cutoff. To address this concern and demonstrate that VERCATION leverages LLMs' general code understanding capabilities rather than specific vulnerability knowledge, we conducted an ablation study comparing each model's performance on CVEs published before and after their respective knowledge cutoff dates.

The three LLMs in our evaluation have different knowledge cutoff dates: CodeLlama's training data extends to September 2022 (based on Llama 2's cutoff date [62]), GPT-4's knowledge cutoff is December 2023 [63], and DeepSeek-V3's knowledge cutoff is July 2024 (confirmed through direct inquiry with DeepSeek). We partitioned our dataset according to each model's cutoff date and evaluated their performance on both pre-cutoff and post-cutoff CVE subsets as shown in Figure 6.

As shown in Table VII, CodeLlama shows a slight performance decrease of 1.25% (from 0.881 to 0.870) when analyzing post-cutoff CVEs. In contrast, both GPT-4 and DeepSeek-V3 demonstrate slight performance improvements on post-cutoff CVEs, with GPT-4 improving by 0.54% and DeepSeek-V3 by 1.73%. The performance variations across all models are minimal ($< 2\%$), indicating remarkable consistency regardless of knowledge cutoff boundaries. These results validate our core hypothesis that VERCATION's effectiveness stems from leveraging LLMs' fundamental code understanding and reasoning capabilities rather than memorized vulnerability-specific knowledge.

3) Effectiveness of the Statement Weighting Strategy: To validate the effectiveness of our statement weighting strategy, we conducted an ablation study comparing our weighted approach with different weight ratios across our entire dataset. We systematically evaluated weight ratios from 1:1 to 5:1, where the first number represents the weight for sensitive function statements ($W_{sensitive}$) and the second number represents the weight for regular statements (W_{base}). These ratios were tested using the DeepSeek-V3 model, with all other parameters held constant.

TABLE VII: Performance Comparison Across Knowledge Cutoff Periods.

Model	Cutoff Date	Period	F1-score
CodeLlama	2022.09	Pre-cutoff	0.881
		Post-cutoff	0.870 (↓ 1.25%)
GPT-4	2023.10	Pre-cutoff	0.923
		Post-cutoff	0.928 (↑ 0.54%)
DeepSeek-V3	2024.07	Pre-cutoff	0.927
		Post-cutoff	0.943 (↑ 1.73%)

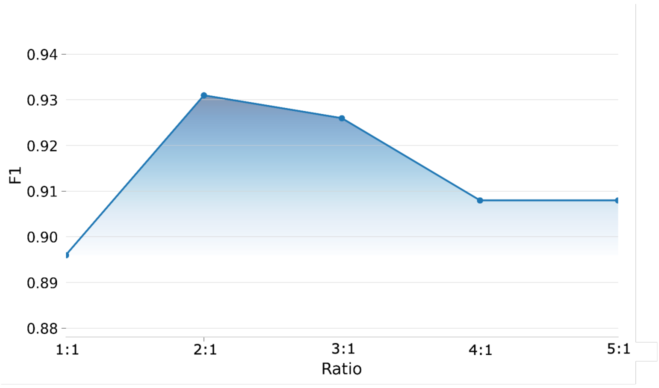


Fig. 7: Impact of Statement Weight Ratios on F1-Score

Figure 7 illustrates the impact of different weight ratios on F1-score. The results demonstrate that equal weighting (1:1 ratio) achieves an F1-score of 0.896, while our proposed 2:1 ratio ($W_{sensitive} = 2.0$, $W_{base} = 1.0$) achieves the optimal

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3599581

TABLE VIII: Accuracy of Code Clone Detection Methods.

Methods	Level	Precision	Recall
Edit distance	statement	0.85	0.84
Hash values	statement	0.82	0.82
CodeBERT	statement	0.84	0.88
FCDetector	function	0.89	0.93
VERCATION	statement	**0.90**	**0.95**

performance of 0.931, representing a 3.91% improvement. Performance gradually decreases as the weight ratio increases beyond 2:1. Higher ratios (3:1 and above) lead to over-emphasis on sensitive functions. This causes the similarity score to be dominated by a small number of statements, reducing the method's ability to distinguish between true vulnerability patterns and coincidental sensitive function usage.

> **RQ2**: VERCATION's effectiveness stems from the synergistic combination of static analysis and LLMs. DeepSeek-V3 with a Few-shot+CoT prompt proves to be the most accurate, consistent, and generalizable model. Furthermore, our 1:2 statement weighting strategy is shown to be optimal for balancing the influence of sensitive functions.

C. RQ3: Component Robustness

Here we evaluate the accuracy of the similarity comparison technique utilized in VERCATION's phase 2.

1) Accuracy of Code Clone Detection: We compare the code clone detection technique in VERCATION's phase 2 with typical statement-level code similarity detection methods: Edit distance [64], Hash value [65], and CodeBERT embedding comparison [66]. Current SOTA code clone detection methods primarily focus on function-level detection [18], whereas our method operates at the statement level. To perform a comparison, during the process of backtracing the previous commits of vulnerable statements, we apply code clone detection to the entire function before and after the modification. If the commit includes changes to other non-vulnerable statements within the function, we normalize these statements to be identical. Table VIII summarizes the accuracy of different code clone detection methods.

Edit distance [64] calculates the minimum number of edit operations required to transform one code statement into another. A smaller edit distance indicates a higher similarity. Hash values comparison [65] generates hash for each code statement through MD5 algorithm and then compares the hash similarity. CodeBERT embedding comparison [66] uses CodeBERT to generate embedding representations for each code statement, then computes the cosine similarity between these embeddings. FCDetection [67] generates AST and CFG representations as features. Word2vec and Graph2vec are used to embed these features. Then the fused feature vectors are input into a deep neural network model for classification of code clones.

For Edit distance, Hash values, and CodeBERT embedding comparisons, we conducted independent threshold optimization. Similar to our threshold sensitivity analysis for Vercation, we evaluated each method across different threshold values from 0.1 to 1.0 with a step of 0.1, and selected the threshold that achieved the best F1 for that specific method. As FCDetector is a specialized code clone detection tool, we used its original pre-trained model and thresholds as specified in their work. The results presented in Table VIII represent each method's optimal performance with their respective optimized thresholds. The results show that the AST-based code clone detection achieves the best performance. Because the edit distance and hash value methods terminate the backtrack in cases of low textual similarity without considering code behavior, thus leads to a large number of false negatives. The results show that our method is robust against low syntactic similarity. CodeBERT embedding comparison, while effective for semantic similarity, cannot capture fine-grained structural information. AST representations preserve the exact syntactic structure of code, allowing for more precise comparisons of code organization and logic flow. This structural analysis can identify clones that CodeBERT might miss, especially in cases where similar functionality is implemented with different coding patterns or variable names, which can be addressed by AST normalization. Although FCDetector also extracts AST as a feature and normalizes variable names and constant values, it cannot identify function outline cases (as shown in Listing 2). Moreover, the CFG features that FCDetector focuses on are not particularly effective when there are only a few vulnerable statements.

2) Analysis of Refactoring Commits: As motivated in Example 2 of Section II-C, refactoring commits can significantly impact vulnerable version identification by introducing structural differences while preserving code behavior. To validate this motivation and evaluate the effectiveness of our code clone detection approach, we analyzed the commits encountered during our evaluation. In the backtrack of our dataset, we conducted a total of 276 comparisons between pre-commit and post-commit. On average, it takes 3.73 comparisons to identify the *vic* for each *vfc*. During the commit backtrack, cases with significant edit-distance differences ($\theta_1 < 0.9$) but minor AST differences ($\theta_2 \geq 0.8$) were classified as refactoring commits. Out of these comparisons, 89 commits (32.2%) were identified as refactoring commits, indicating that refactoring commits are a significant factor in backtracing *vic*. Furthermore, our analysis reveals that cases without any intermediate commit between the *vic* and the *vfc* are comparatively uncommon, constituting only 11% of instances in our dataset.

3) Threshold Sensitivity: We used ϑ_1 and ϑ_2 to represent the threshold for edit-distance similarity and AST-based similarity, respectively. ϑ_3 represents the *similarity_score* used for the backtrack termination condition. To measure the sensitivity of the thresholds, we incrementally increased ϑ_1, ϑ_2 and ϑ_3 by 0.1 from 0 to 1, and evaluated the identification F1-score of the CVEs in ground-truth. Specifically, we first initialized ϑ_2 and ϑ_3 to 0.8, as high similarity thresholds generally provide better precision in code clone detection. Then we varied ϑ_1 from 0.1 to 1.0 with a step of 0.1 and found the value of ϑ_1 that achieved the highest F1-score (0.9). With ϑ_1 fixed at 0.9 and kept ϑ_3 at 0.8, we varied ϑ_2 from 0.1 to 1.0 and found the optimal value (0.8). Finally, with ϑ_1 = 0.9 and ϑ_2 = 0.8, we

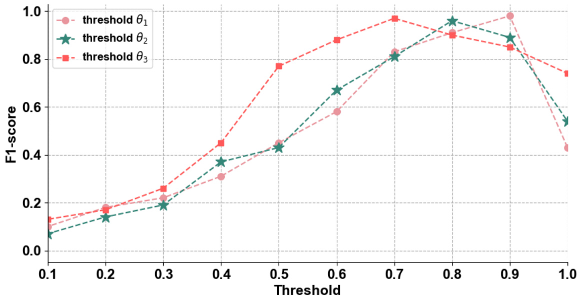


Fig. 8: Threshold Sensitivity of VERCATION.

varied ϑ_3 from 0.1 to 1.0 and determined the optimal value of ϑ_3 (0.7). When the thresholds are greater than the optimal values, the precision is higher while the recall decreases. In contrast, the precision slightly decreases.

RQ3: Refactoring commits occupy a significant proportion of the code changes history. The AST-based similarity comparison in VERCATION performs better in the vulnerability-introducing commit identification task for calculating code similarity, enhancing the overall effectiveness of VERCATION in vulnerable versions detection.

D. RQ4: Real-World Application

We created a new vulnerability dataset consisting of 342 CVEs from the NVD. By developing a simple crawler, we obtained the corresponding CPE and CVE patches. We applied VERCATION to the vulnerability dataset and found the vulnerable versions of 202 CVEs are incorrect. Among them, the CPE for 134 CVEs is incomplete, with 108 CVEs only reporting a single vulnerable version, significantly increasing the risk of vulnerability propagation. Furthermore, the CPE for 68 CVEs is affected by the overinclusion problem, with 56 CVEs considered all versions before the OSS version mentioned in the vulnerability report as susceptible to attack, without conducting a detailed analysis. We have submitted reports to the NVD detailing the CVEs with incorrect affected versions information we identified. The NVD team's response said they plan to address these inaccuracies as part of their ongoing efforts to enhance the quality of their vulnerability information. More importantly, we have discovered 4 CVEs that do not provide information about the vulnerable version.

To understand the practical impact of accurately identifying vulnerable versions, we analyzed the severity of 134 vulnerabilities with incomplete version information in the NVD dataset. The analysis shows that CVEs with CVSS Score of 4.0-6.9 (Medium) and 7.0-8.9 (High) accounted for the most, at 49.7% and 49.1%. Notably, we identified 2 critical vulnerabilities (CVSS Score 9.0-10.0) in *FFmpeg* and *php*. These findings highlight that in many high-severity vulnerabilities, NVD does not report the correct vulnerable version. This inaccuracy has serious implications, as downstream software manufacturers may not recognize the need to update their vulnerable dependencies in time, exposing their systems to significant security risks. Vercation's high accuracy in identifying vulnerable versions therefore helps organizations better improve the accuracy of vulnerability reports.

RQ4: Applying Vercation to real-world CVEs demonstrated practical impact in two key findings: it identified 202 CVEs (59.1%) with incomplete or overincluded versions in NVD reports. Notably, nearly half of the vulnerabilities with incorrect versions were of high or critical severity, emphasizing the importance of accurate version identification.

VI. DISCUSSION

A. Performance and Cost Analysis

In terms of performance efficiency, VERCATION analyzes each vulnerability in an average of 28.61 seconds. This includes the time for program slicing, LLM processing, and AST-level code clone detection. Compared to existing approaches: V-SZZ takes 10.25 seconds on average, V0Finder requires approximately 22 seconds per CVE, Vision needs 1,094.12 seconds per vulnerability, and V1SCAN can detect vulnerabilities within 20 seconds for 99% of projects. While VERJava performs faster at 0.71 seconds per CVE, it achieves much lower recall in version identification. Compared to other methods, Vercation achieves higher accuracy without significant time overhead.

The best LLM DeepSeek-V3 in our evaluation, whose API pricing is $0.30 per million tokens for input and $1.10 per million tokens for output. Based on our experimental setup with 10 independent runs for self-consistency, the average LLM API cost per CVE is only $0.0034. This cost includes: input tokens for CVE descriptions, dangerous flows, and few-shot examples, output tokens for logic explanation, and vulnerable statements. The low cost allows the LLM's application in practical security.

B. Comparison with Dynamic Analysis

While dynamic analysis through PoC testing provides definitive evidence of vulnerability existence, several limitations render it insufficient for comprehensive and efficient identification of vulnerable versions at scale. Our analysis of CVEs published in the NVD from 1999 to 2025 reveals that only 24% of CVEs provide publicly available exploit links, and these exploit reports do not guarantee successful reproduction across different software versions.

The portability of PoCs across different software versions presents another critical challenge for dynamic analysis approaches. Previous studies have demonstrated that PoC reproducibility across different environments and versions is a significant technical hurdle [4], [7]. During the construction of our ground-truth dataset, we encountered these challenges firsthand when attempting to validate vulnerabilities through PoC execution. Of the available PoCs in our dataset, only 55% could be successfully executed across multiple versions without modification. The remaining cases failed to trigger the vulnerability due to environmental dependencies, compilation differences, or version-specific behavioral changes, which required substantial manual code-level analysis for verification.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3599581

The temporal cost of dynamic analysis further limits its practical applicability for large-scale vulnerable version identification. Our experience during dataset construction revealed that each CVE required an average of 3 hours for environment setup, dependency resolution, and PoC execution across different versions, excluding the time needed for manual verification when PoCs failed. For our complete dataset of 122 CVEs spanning 1,111 versions, comprehensive dynamic testing would require approximately 30 days compared to 28.61 seconds per CVE for our static approach. This represents a 742x time difference in favor of static analysis, making dynamic approaches impractical for real-world deployment scenarios where rapid vulnerability assessment is crucial.

C. Threats to Validity

1) Internal Validity: **Static–analysis precision.** VERCATION relies on the Joern parser to build code-property graphs that combine an AST, control-flow graph, and data-dependency graph. Joern struggles with certain C/C++ constructs, notably complex pointer arithmetic, overloaded constructors, and indirect calls, so some data flows and constructor edges are missed. Our AST expansion step further inherits the classic limitations of method outlining: it cannot always disambiguate targets reachable via function pointers, callbacks, or C++ virtual dispatch, which static analysis alone cannot resolve definitively [68], [69]. These imprecisions may hide or mislocate refactoring patterns.

Inter-procedural dependencies. In multi-function patches, VERCATION currently analyses each affected function independently and then merges the results. Although effective for most cases, this strategy may overlook vulnerability patterns that hinge on data/control flows spanning several functions.

LLM variability and reproducibility. As LLMs continue to evolve rapidly, the performance of VERCATION might change with newer versions of these models. This could affect the long-term consistency of our results. Additionally, LLMs may have biases or limitations in their training data that could influence their ability to understand certain types of vulnerabilities or code structures. To mitigate this, we have provided detailed information about the LLM versions and prompts used in our study, but future research may need to account for potential variations in LLM capabilities and performance as these models continue to develop.

2) External Validity: **Language generalisability.** VERCATION's core methodology is language-agnostic and can be adapted to other programming languages. The program slicing tool Joern inherently supports multiple languages, including Java, Python, JavaScript, and PHP. The main adaptations required would be updating the sensitive function table for language-specific vulnerability patterns and modifying AST generation rules according to the target language's syntax and semantics. Future work could involve extending VERCATION to support a broader range of programming languages and verifying its effectiveness across different language paradigms.

Repository branching complexity. While our approach handles common merge [70] and branching patterns effectively, complex branching scenarios, such as when the same vulnerable code is independently modified in multiple parallel feature branches before being merged (creating diamond-shaped merge patterns), may require more sophisticated graph traversal algorithms to accurately determine the earliest vulnerability introduction point [71], [72].

VII. RELATED WORK

A. Vulnerability Fixing Commit Analysis

Several studies have focused on analyzing vulnerability fixing commit (VFC) to extract vulnerability features, which can be used for various purposes such as vulnerability detection, classification, and affected version identification. VulPecker [73] extracts code patterns from VFC to detect similar vulnerabilities in other software versions. SySeVR [74] uses a systematic approach to extract syntax- and semantic-based vulnerability features from patches. VulDeePecker [75] leverages deep learning techniques to learn vulnerability patterns from code gadgets extracted from vulnerability fixing commits. Most existing VFC analysis methods rely heavily on predefined rules or patterns, which may not capture the full context of vulnerabilities. VERCATION incorporates VFC analysis by leveraging LLMs to understand the context of code changes in fixing commits, potentially capturing more nuanced vulnerability features.

B. Vulnerable Version Detection for OSS

Several approaches have been proposed for vulnerable version identification. V-SZZ [17] uses the SZZ algorithm to backtrack *vfc*s and identify vulnerability-introducing changes. MVP [12] employs program slicing to extract vulnerability and patch signatures, then uses these to identify potentially vulnerable functions. V0Finder [16] generates fingerprints for vulnerable functions and uses a clone-based technique to detect vulnerable versions across different software releases. VCCFinder [76] uses machine learning techniques to identify vulnerability-contributing commits. VUDDY [77] proposes a scalable approach for vulnerable code clone detection in large-scale code bases. Existing approaches often rely on predefined patterns and syntactic-level analysis, leading to imprecision in vulnerability characterization and code clone detection. VERCATION combines static analysis with LLM for code understanding and introduces AST-based code clone detection to address the code structure modification problem.

C. Code Refactoring Detection

Some researchers conducted studies on the characteristics of code refactoring not altering the code behavior of the code and have proposed detection methods. MLRefScanner [78] detects refactoring commits in Python projects by analyzing commit history and extracting features that represent refactoring activities without altering code behavior. Abid et al. [79] perform a study outlining different levels of refactoring from code level to architecture and discuss the importance of maintaining behavior during refactoring. Eman et al. [80] focuses on how refactoring is integrated into the code review process while ensuring that the original software behavior is preserved.

Existing code refactoring detection methods primarily focus on high-level changes like renaming and method/class moves. Our approach, however, targets statement-level changes through fine-grained AST expansion and normalization, including inline function expansion during AST generation.

D. Code Clone Detection

Some SOTA techniques utilize static analysis to conduct code clone detection. SCDetector [81] feeds tokens with graph detail into a Siamese architecture neural network to train a code clone detector. Fang et al. [82] propose a joint code representation that applies fusion embedding techniques to learn hidden features of source codes, then train a supervised deep learning model to detect functional code clones. FCDetection [67] generates AST and CFG representations and uses Word2vec to embed these features into vectors. Then the fused feature vectors are input into a deep neural network model for classification of code clones.

VIII. Conclusion and Future Work

This paper proposed VERCATION, an approach designed for vulnerable version identification of open-source C/C++ software. Our approach introduces two key innovations: leveraging LLMs to enhance the extraction of vulnerability-related statements and employing code clone detection based on expanded and normalized ASTs. Experimental results on our dataset of 122 CVEs across 12 popular open-source projects validate that VERCATION surpasses existing techniques for vulnerable version identification, demonstrating notable improvements in both precision and recall. VERCATION's novelty lies not in individual techniques but in the systematic integration of semantic analysis and methodological innovations in LLM application. The combination creates a qualitatively different approach that advances the state-of-the-art in both accuracy and practical applicability, opening new research directions for AI-assisted software security analysis.

In the future, we plan to empirically validate VERCATION on a broader set of languages (Java, Python, JavaScript, and PHP) to confirm its effectiveness. Furthermore, we are interested in exploring integrating inter-procedural analysis techniques to enhance the detection of cross-functional vulnerability patterns.

Acknowledgement

We thank the anonymous reviewers for their insightful comments. This research is supported by Beijing Natural Science Foundation (Grant No. L234033).

References

[1] "Nvd - home," https://nvd.nist.gov/, 2009.

[2] "Cve-2018-5785," https://nvd.nist.gov/vuln/detail/CVE-2018-5785, 2018.

[3] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in 28th USENIX security symposium (USENIX Security 19), 2019, pp. 869–885.

[4] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 919–936.

[5] X. Tan, Y. Zhang, C. Mi, J. Cao, K. Sun, Y. Lin, and M. Yang, "Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 3282–3299.

[6] V. H. Nguyen and F. Massacci, "The (un) reliability of nvd vulnerable versions data: An empirical experiment on google chrome vulnerabilities," in Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, 2013, pp. 493–498.

[7] J. Dai, Y. Zhang, H. Xu, H. Lyu, Z. Wu, X. Xing, and M. Yang, "Facilitating vulnerability assessment through poc migration," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 3300–3317.

[8] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," IEEE Transactions on Software Engineering, vol. 43, no. 7, pp. 641–657, 2016.

[9] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, "Evaluating szz implementations through a developer-informed oracle," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 436–447.

[10] V. H. Nguyen, S. Dashevskyi, and F. Massacci, "An automatic method for assessing the versions affected by a vulnerability," Empirical Software Engineering, vol. 21, pp. 2268–2297, 2016.

[11] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, "How bugs are born: a model to identify how bugs are introduced in software components," Empirical Software Engineering, vol. 25, pp. 1294–1340, 2020.

[12] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou et al., "Mvp: Detecting vulnerabilities using patch-enhanced vulnerability signatures," in 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 1165–1182.

[13] Y. Shi, Y. Zhang, T. Luo, X. Mao, and M. Yang, "Precise (un) affected version analysis for web vulnerabilities," in Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–13.

[14] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in 2012 IEEE Symposium on Security and Privacy. IEEE, 2012, pp. 48–62.

[15] S. Woo, H. Hong, E. Choi, and H. Lee, "Movery: A precise approach for modified vulnerable code clone discovery from modified open-source software components," in 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 3037–3053.

[16] S. Woo, D. Lee, S. Park, H. Lee, and S. Dietrich, "V0finder: Discovering the correct origin of publicly reported software vulnerabilities," in 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 3041–3058.

[17] L. Bao, X. Xia, A. E. Hassan, and X. Yang, "V-szz: automatic identification of version ranges affected by cve vulnerabilities," in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2352–2364.

[18] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, 2020, pp. 516–527.

[19] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.

[20] J. Li, G. Li, C. Tao, H. Zhang, F. Liu, and Z. Jin, "Large language model-aware in-context learning for code generation," arXiv preprint arXiv:2310.09748, 2023.

[21] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.

[22] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–5.

[23] T. Li, G. Zhang, Q. D. Do, X. Yue, and W. Chen, "Long-context llms struggle with long in-context learning," 2024.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3599581

[24] W. Song, S. Oh, S. Mo, J. Kim, S. Yun, J.-W. Ha, and J. Shin, "Hierarchical context merging: Better long context understanding for pre-trained llms," 2024.

[25] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: Statement-level vulnerability detection using graph neural networks," in Proceedings of the 19th international conference on mining software repositories, 2022, pp. 596–607.

[26] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "Vulsniper: Focus your attention to shoot fine-grained vulnerabilities." in IJCAI, 2019, pp. 4665–4671.

[27] S. Woo, E. Choi, H. Lee, and H. Oh, "V1scan: Discovering 1-day vulnerabilities in reused c/c++ open-source software components using code classification techniques," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 6541–6556.

[28] Q. Sun, L. Xu, Y. Xiao, F. Li, H. Su, Y. Liu, H. Huang, and W. Huo, "Verjava: Vulnerable version identification for java oss with a two-stage analysis," in 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2022, pp. 329–339.

[29] S. Wu, R. Wang, K. Huang, Y. Cao, W. Song, Z. Zhou, Y. Huang, B. Chen, and X. Peng, "Vision: Identifying affected library versions for open source software vulnerabilities," in Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 1447–1459.

[30] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat et al., "Gpt-4 technical report," arXiv preprint arXiv:2303.08774, 2023.

[31] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin et al., "Code llama: Open foundation models for code," arXiv preprint arXiv:2308.12950, 2023.

[32] DeepSeek-AI, "Deepseek-v3 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2412.19437

[33] "Joern - home," https://joern.io/, 2014.

[34] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell et al., "Language models are few-shot learners," Advances in neural information processing systems, vol. 33, pp. 1877–1901, 2020.

[35] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li et al., "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," arXiv preprint arXiv:2401.14196, 2024.

[36] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023, pp. 1469–1481.

[37] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, "A deep dive into large language models for automated bug localization and repair," arXiv preprint arXiv:2404.11595, 2024.

[38] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," arXiv preprint arXiv:2305.07922, 2023.

[39] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, "Large language models are human-level prompt engineers," arXiv preprint arXiv:2211.01910, 2022.

[40] B. Chen, Z. Zhang, N. Langrené, and S. Zhu, "Unleashing the potential of prompt engineering in large language models: a comprehensive review," arXiv preprint arXiv:2310.14735, 2023.

[41] "Cve-2017-14169," https://nvd.nist.gov/vuln/detail/CVE-2017-14169, 2017.

[42] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in 11th working conference on reverse engineering. IEEE, 2004, pp. 144–151.

[43] A. Almogahed, H. Mahdin, M. Omar, N. H. Zakaria, Y. H. Gu, M. A. Al-Masni, and Y. Saif, "A refactoring categorization model for software quality improvement," Plos one, vol. 18, no. 11, p. e0293742, 2023.

[44] M. Weiser, "Program slicing," IEEE Transactions on software engineering, no. 4, pp. 352–357, 1984.

[45] E. Aghaei, E. Al-Shaer, W. Shadid, and X. Niu, "Automated cve analysis for threat prioritization and impact prediction," arXiv preprint arXiv:2309.03040, 2023.

[46] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou et al., "Chain-of-thought prompting elicits reasoning in large language models," Advances in neural information processing systems, vol. 35, pp. 24 824–24 837, 2022.

[47] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi, "Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection," in 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2021, pp. 224–236.

[48] Z. Xue, Y. Zhang, and R. Xu, "Clone-based code method usage pattern mining," in Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 543–547.

[49] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, 2017, pp. 269–280.

[50] F. A. Alaba, M. Othman, I. A. T. Hashem, and F. Alotaibi, "Internet of things security: A survey," Journal of Network and Computer Applications, vol. 88, pp. 10–28, 2017.

[51] "Cve-2021-20294," https://nvd.nist.gov/vuln/detail/CVE-2021-20294, 2021.

[52] "Cpe - home," https://nvd.nist.gov/products/cpe, 2009.

[53] S. Kim, T. Zimmermann, K. Pan, E. James Jr et al., "Automatic identification of bug-introducing changes," in 21st IEEE/ACM international conference on automated software engineering (ASE'06). IEEE, 2006, pp. 81–90.

[54] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 2014, pp. 313–324.

[55] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, 2024, pp. 102–106.

[56] "Cve-2021-30499," https://nvd.nist.gov/vuln/detail/CVE-2021-30499, 2021.

[57] "Cve-2017-14152," https://nvd.nist.gov/vuln/detail/CVE-2017-14152, 2017.

[58] "Cve-2021-33815," https://nvd.nist.gov/vuln/detail/CVE-2021-33815, 2021.

[59] Y. Fan, X. Xia, D. A. Da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of mislabeled changes by szz on just-in-time defect prediction," IEEE transactions on software engineering, vol. 47, no. 8, pp. 1559–1586, 2019.

[60] "Cve-2020-35965," https://nvd.nist.gov/vuln/detail/CVE-2020-35965, 2021.

[61] "Cve-2022-1355," https://nvd.nist.gov/vuln/detail/CVE-2022-1355, 2021.

[62] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale et al., "Llama 2: Open foundation and fine-tuned chat models," arXiv preprint arXiv:2307.09288, 2023.

[63] "Gpt-4 turbo," https://platform.openai.com/docs/models/gpt-4-turbo, 2023.

[64] A. Sheneamer and J. Kalita, "Code clone detection using coarse and fine-grained hybrid approaches," in 2015 IEEE seventh international conference on intelligent computing and information systems (ICICIS). IEEE, 2015, pp. 472–480.

[65] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in 2010 IEEE International Conference on Software Maintenance. IEEE, 2010, pp. 1–9.

[66] T. Sonnekalb, B. Gruner, C.-A. Brust, and P. Mäder, "Generalizability of code clone detection on codebert," in Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–3.

[67] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, 2020, pp. 516–527.

[68] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in Proceedings of the 25th international conference on compiler construction, 2016, pp. 265–266.

[69] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in Proceedings of the 31st IEEE/ACM international conference on automated software engineering, 2016, pp. 332–343.

[70] J. C. C. Ríos, S. M. Embury, and S. Eraslan, "A unifying framework for the systematic analysis of git workflows," Information and Software Technology, vol. 145, p. 106811, 2022.

[71] Y. Fan, X. Xia, D. A. Da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of mislabeled changes by szz on just-in-time defect prediction," IEEE transactions on software engineering, vol. 47, no. 8, pp. 1559–1586, 2019.

[72] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, “Evaluating szz implementations through a developer-informed oracle,” in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 436–447.

[73] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, “Vulpecker: an automated vulnerability detection system based on code similarity analysis,” in Proceedings of the 32nd annual conference on computer security applications, 2016, pp. 201–213.

[74] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, “Sysevr: A framework for using deep learning to detect software vulnerabilities,” IEEE Transactions on Dependable and Secure Computing, vol. 19, no. 4, pp. 2244–2258, 2021.

[75] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, “Vuldeepecker: A deep learning-based system for vulnerability detection,” arXiv preprint arXiv:1801.01681, 2018.

[76] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, “Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits,” in Proceedings of the 22nd ACM SIGSAC conference on computer and communications security, 2015, pp. 426–437.

[77] S. Kim, S. Woo, H. Lee, and H. Oh, “Vuddy: A scalable approach for vulnerable code clone discovery,” in 2017 IEEE symposium on security and privacy (SP). IEEE, 2017, pp. 595–614.

[78] S. Noei, H. Li, and Y. Zou, “Detecting refactoring commits in machine learning python projects: A machine learning-based approach,” 2024.

[79] C. Abid, V. Alizadeh, M. Kessentini, T. d. N. Ferreira, and D. Dig, “30 years of software refactoring research: a systematic literature review,” arXiv preprint arXiv:2007.02194, 2020.

[80] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, “Refactoring practices in the context of modern code review: An industrial case study at xerox,” in 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2021, pp. 348–357.

[81] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, H. Liang, and H. Jin, “Scdetector: Software functional clone detection based on semantic tokens analysis,” in Proceedings of the 35th IEEE/ACM international conference on automated software engineering, 2020, pp. 821–833.

[82] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, “Functional code clone detection with syntax and semantics fusion learning,” in Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, 2020, pp. 516–527.