Enhancing Vulnerability Detection via Inter-procedural Semantic Completion

BOZHI WU, Nanyang Technological University, Singapore
CHENGJIE LIU, Peking University, China
ZHIMING LI*, Nanyang Technological University, Singapore
YUSHI CAO, Nanyang Technological University, Singapore
JUN SUN, Singapore Management University, Singapore
SHANG-WEI LIN, Nanyang Technological University, Singapore

Inspired by advances in deep learning, numerous learning-based approaches for vulnerability detection have emerged, primarily operating at the function level for scalability. However, this design choice has a critical limitation: many vulnerabilities span multiple functions, causing function-level approaches to lose the semantics of called functions and fail to capture true vulnerability patterns. To address this issue, we propose *VulnSC*, a novel framework designed to enhance learning-based approaches by complementing inter-procedural semantics. *VulnSC* retrieves the source code of called functions for datasets and leverages large language models (LLMs) with well-designed prompts to generate summaries for these functions. The datasets, enhanced with these summaries, are fed into neural networks for improved vulnerability detection. *VulnSC* is the first general framework to integrate inter-procedural semantics into existing learning-based approaches for vulnerability detection while maintaining scalability. We evaluate *VulnSC* on four state-of-the-art learning-based approaches using two widely used datasets, and our experimental results demonstrate that *VulnSC* significantly enhances detection performance with minimal additional computational overhead.

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

Additional Key Words and Phrases: Vulnerability detection, inter-procedural semantics, LLMs.

ACM Reference Format:
Bozhi Wu, Chengjie Liu, Zhiming Li, Yushi Cao, Jun Sun, and Shang-Wei Lin. 2025. Enhancing Vulnerability Detection via Inter-procedural Semantic Completion. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA037 (July 2025), 23 pages. https://doi.org/10.1145/3728912

1 Introduction

Vulnerabilities in software systems pose significant threats to software security and reliability, as they can be exploited by attackers, leading to severe financial and data losses [10], compromising critical infrastructure, and even endangering human lives. To mitigate these risks, various static analysis tools have been developed to detect vulnerabilities early in the development lifecycle, reducing the effort required for remediation. These tools include both open-source options like CPPCheck [4], Clang Static Analyzer [8], CodeChecker [2], Infer [6], and Flawfinder [5], as well

*Zhiming Li is the corresponding author.

Authors' Contact Information: Bozhi Wu, Nanyang Technological University, Singapore, Singapore, bozhi001@e.ntu.edu.sg; Chengjie Liu, Peking University, Beijing, China, cheng@stu.pku.edu.cn; Zhiming Li, Nanyang Technological University, Singapore, Singapore, ZHIMING001@e.ntu.edu.sg; Yushi Cao, Nanyang Technological University, singapore, Singapore, YUSHI002@e.ntu.edu.sg; Jun Sun, Singapore Management University, Singapore, Singapore, JunSun@smu.edu.sg; Shang-Wei Lin, Nanyang Technological University, Singapore, Singapore, shang-wei.lin@ntu.edu.sg.

ACM 2994-970X/2025/7-ARTISSTA037
https://doi.org/10.1145/3728912

```
1  static int vhost_vdpa_probe(struct vdpa_device *vdpa)
2  {
3          const struct vdpa_config_ops *ops = vdpa->config;
4          struct vhost_vdpa *v;
5          int minor;
6          int i, r;
7          ...
8          v = kzalloc(sizeof(*v), GFP_KERNEL | __GFP_RETRY_MAYFAIL);
9          if (!v)
10         return -ENOMEM;
11         ...
12         put_device(&v->dev);
13 -       ida_simple_remove(&vhost_vdpa_ida, v->minor);
14         return r;
15 }
```

Fig. 1. A Vulnerability "Double Free" from Linux with Commit ID *e07754e*.

as commercial solutions such as Coverity [3] and Checkmarx [1]. However, in the current era of rapid software development, known as the "big code" era, numerous unknown vulnerabilities continue to emerge [11]. Static analysis tools, often rely on manual rule definition by security experts, struggling to keep pace with the increasingly emerging vulnerabilities.

Inspired by the rapid advancements in deep learning [33], numerous learning-based approaches have been developed to automate and enhance the process of vulnerability detection without the need for manually created rules [53]. These approaches aim to learn effective code representations from programs to uncover vulnerability patterns [54]. They primarily operate at two levels of granularity: Some concentrate on learning from entire single functions, exemplified by approaches like Draper [48], VGDetector [21], Devign [57], and REVEAL [18]. Others, such as VulDeePecker [37], SySeVR [36], and DeepWukong [20] aim to learn from a set of statements sliced from a single function. These approaches typically detect vulnerabilities below function level rather than across multiple functions primarily due to scalability concerns, since a single sample with multiple functions may be too large to be processed by learning models [34].

However, these intra-procedural approaches overlook a crucial fact: in practice, many software vulnerabilities span multiple functions, where the context of function calls is essential for a comprehensive understanding of the vulnerabilities. For example, as depicted in Figure 1, a real-world "double free" vulnerability in the Linux kernel is fixed in commit *e07754e*. In line 12, the put_device() function is invoked, which calls vhost_vdpa_release_dev(), subsequently invoking ida_simple_remove() and freeing "v". Then, in line 13, ida_simple_remove() is invoked again to free "v", resulting in a double free. If the source code in the called functions put_device() and ida_simple_remove() is overlooked, it becomes difficult to identify the vulnerability. This omission of inter-procedural information can hinder the effectiveness of learning-based approaches, as demonstrated by previous work [35]. Neglecting this inter-procedural context may diminish the ability to identify intricate vulnerabilities that extend beyond individual functions.

To tackle this challenge, learning-based approaches should incorporate inter-procedural semantics to learn effective code representation for vulnerability detection. To capture inter-procedural semantics, various traditional techniques have been proposed, including context-sensitive analysis [50, 52], context-insensitive analysis [47], inter-procedural data flow analysis [16, 41], inter-procedural control flow analysis [40], and summary-based analysis [49]. However, techniques such

as context-sensitive and context-insensitive analysis, as well as inter-procedural data flow and control flow analysis, require analyzing the source code of called functions and managing intricate data or control flow information for each sample, which may not scale effectively for large-scale programs. In contrast, summary-based analysis abstracts the behaviors of called functions into concise summaries. This technique minimally increases the sample size by representing the source code of called functions with their concise summaries. Inspired by this, we propose generating summaries for called functions to complement the inter-procedural semantics of function-level datasets, which can enhance the function-level dataset while maintaining scalability for learning-based approaches.

Therefore, in this paper, we propose a simple yet effective framework named *VulnSC* for vulnerability detection. It incorporates code summarization techniques with large language models (LLMs) to complement the semantics of called functions and then leverages various state-of-the-art neural networks to learn code representations from the enhanced semantics after inter-procedural semantic completion. This framework aims to enable learning-based approaches to capture comprehensive vulnerability patterns for vulnerability detection.

Initially, *VulnSC* retrieves the source code of called functions for function-level datasets. Most existing real-world datasets used for learning-based approaches are sourced from open-source projects on GitHub and include information about the project name and commit version, which allows us to obtain the source code of called functions. For a given function-level sample, we first navigate to the repository using the project name and check out the specified commit with the commit ID. We then construct the call graph for the sample to identify its called functions. Using the names of these functions, we retrieve their source code within the repository. This process is repeated for each called function to gather the source code of these functions it calls. Thus, *VulnSC* effectively retrieves both directly and indirectly called functions for all samples.

Subsequently, *VulnSC* leverages LLM with well-designed prompts to generate summaries for the called functions. Since LLMs have demonstrated inspiring performance in code understanding and summarization [13, 42], we utilize LLMs to summarize the called functions by designing appropriate prompts with various prompting strategies to capture essential behaviors related to vulnerabilities within the called functions. In this way, we obtain and use summaries to represent the called functions, effectively capturing inter-procedural semantics for vulnerability detection while maintaining the scalability of learning-based approaches.

Finally, *VulnSC* enhances function-level datasets by incorporating summaries of the called functions and leverages learning-based approaches to learn code representations from these enriched samples for vulnerability detection. Specifically, *VulnSC* first generates summaries for the called functions and appends them as comments to the corresponding code samples. These enhanced samples are then used as input to various state-of-the-art learning-based models to perform vulnerability detection.

We implement *VulnSC* using four representative LLMs (i.e., GPT-4o, CodeLlama, DeepSeek, and Mixtral) along with four well-designed prompting strategies and evaluate it on four state-of-the-art learning-based approaches (i.e., CodeBERT, GraphCodeBERT, UniXcoder, and LineVul) across two widely used real-world datasets: Devign[57] and Big-Vul[23]. Experimental results demonstrate that *VulnSC* enhances these four learning-based approaches, with accuracy improvements by up to 4.73 percentage points and F1-score increasing by up to 5.09 percentage points, which surpasses the gains achieved by employing more advanced models. Moreover, *VulnSC* outperforms existing LLM-based vulnerability detection methods, further highlighting its effectiveness in capturing inter-procedural semantics for vulnerability detection. Additionally, *VulnSC* introduces only a minimal increase in total processing time, while GPU memory usage remains unchanged across all learning-based approaches, ensuring high scalability and computational efficiency.

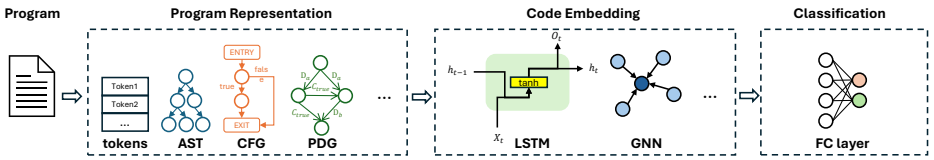


Fig. 2. General Workflow of Learning-Based Approaches for Vulnerability Detection.

In summary, this paper makes the following contributions:

- We propose a novel framework *VulnSC* to address the limitation that existing function-level learning-based approaches overlook inter-procedural semantics for vulnerability detection. *VulnSC* enables learning-based approaches to effectively capture inter-procedural semantics while maintaining scalability. To the best of our knowledge, *VulnSC* is the first work to propose a general framework for integrating inter-procedural analysis into existing learning-based approaches.
- We enhance two widely used open-source vulnerability dataset Devign and Big-Vul by incorporating concise summaries generated by *VulnSC*. Both the source code and the datasets for this work are publicly available on our website [15], providing a solid foundation for further research and development in the field of vulnerability detection.
- We propose four prompting strategies for LLMs to generate concise summaries that effectively capture essential behaviors related to vulnerabilities within called functions.
- We conduct comprehensive evaluations on four state-of-the-art learning-based approaches using the real-world datasets Devign and Big-Vul, and demonstrate that *VulnSC* enhances the performance of learning-based approaches in vulnerability detection while ensuring their scalability.

2 Workflow of Learning-Based Approaches

In this section, we introduce the general workflow of learning-based approaches for vulnerability detection, which typically encompass three phases: program representation, code embedding, and classification, as depicted in Figure 2.

In the first phase, a program is converted into a structured format that neural networks can effectively process. These formats, known as program representations pr, include abstract syntax trees (ASTs), control flow graphs (CFGs), program dependency graphs (PDGs), and token sequences. These representations capture essential semantics such as control flow, data dependencies, and variable interactions, which are crucial for identifying vulnerability patterns and ensuring accurate detection. For example, Draper [48] represents a program as a set of lexical tokens: $pr = \{tk_i\}_{1 \le i \le N}$. Code2Seq [14] employes a set of AST paths as its program representation: $pr = \{< n_i, path_k(n_i, n_j), n_j >\}_{1 \le k \le N}$. IVDetect [32] uses PDGs to represent the program: $pr = (V_p, E_p)$, where V_p and E_p denote the nodes and edges in the program dependency graph, respectively.

In the second phase, the program representation is transformed into a compact vector representation vr. Initially, program representation is embedded into numerical vectors using various techniques such as Word2Vec [22] and GloVe [45]. This embedding step is essential, as machine learning algorithms operate on numerical data rather than raw code or structured representations. Subsequently, based on the program representation pr obtained in the first phase, various neural networks, such as BiLSTM (Bidirectional Long Short-Term Memory), Graph Neural Networks (GNNs), and transformer-based models (e.g., BERT, GPT), are employed to derive a compact vector representation vr from the embedded numerical vectors. For instance, VulDeePecker [37] employs

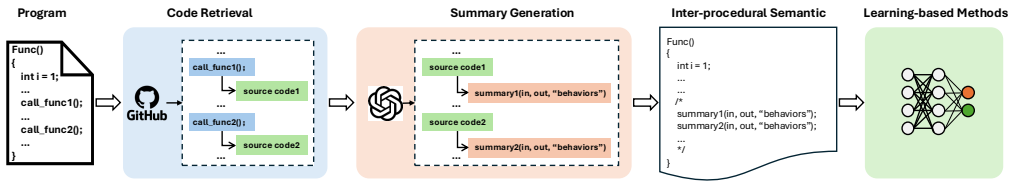


Fig. 3. Framework of *VulnSC*.

BiLSTM to capture sequential dependency information from a sequence of statements as follows,

$$\boldsymbol{h}_{v_1}, \cdots, \boldsymbol{h}_{v_l} = \text{BiLSTM}(E_{v_1}, \cdots, E_{v_l}) \tag{1}$$

$$\boldsymbol{vr} = [\overrightarrow{\boldsymbol{h}_{v_l}}; \overleftarrow{\boldsymbol{h}_{v_1}}] \tag{2}$$

where v denotes statements in the program, l represents the total number of statements, and E denotes an embedding matrix used to encode the nodes v into numerical vectors. Besides, REVEAL [18] leverages GNNs to capture complex relationships within program graphs. LineVul [26] adopts transformer to analyze code sequences, and VulBERTa [30] utilizes BERT for contextual understanding.

In the final phase, learning-based approaches typically use a fully-connected layer as a classification model and map the compact vector representation vr to a probability score p, which indicates the likelihood that the program contains vulnerabilities, as follows:

$$p = \text{Sigmoid}(\text{FC}(vr)) \tag{3}$$

3 Approach

In this paper, we propose a simple yet effective framework *VulnSC*, aimed at enhancing learning-based approaches for vulnerability detection by incorporating inter-procedural semantic completion, as illustrated in Figure 3. The process begins with code retrieval, where the source code of called functions is obtained directly from GitHub using commit ID and project name. After retrieving the source code, LLMs generate concise summaries of the called functions through tailored prompts, capturing their essential behaviors. These summaries, which encapsulate the inputs, outputs, and behaviors, are used to represent the called functions. By incorporating these summaries into the dataset, *VulnSC* enables learning-based approaches to effectively capture inter-procedural semantics within the samples, thereby enhancing their ability to identify vulnerabilities. A detailed description of each phase of the framework is provided below.

3.1 Code Retrieval

The primary objective of code retrieval is to augment function-level datasets used in vulnerability detection by complementing the source code of called functions. Unlike function names, which provide limited information, the source code reveals a detailed implementation of called functions. Therefore, retrieving the source code of called functions establishes a foundational step for subsequent code summary generation, enhancing the inter-procedural semantics of the datasets.

We study the papers published in top-tier conferences and journals up to 2024 and search for all public vulnerability datasets in C/C++ that are used in learning-based approaches, as depicted in Table 1. We find that most existing vulnerability datasets from the real world are function-level and contain metadata including project names and commit IDs.

Therefore, we propose retrieving the source code of called functions from GitHub for the function-level datasets. The process for the code retrieval is outlined in Algorithm 1. Specifically, given a sample from function-level datasets, as depicted in Figure 4, it may call multiple other functions.

Table 1. Existing Datasets for Vulnerability Detection.

Dataset	Source	Granularity	Project Name	Commit ID
Juliet [7]	synthetic	Statement	-	-
Draper [48]	real-world	Function	✓	✓
Devign [57]	real-world	Function	✓	✓
DiverseVul [19]	real-world	Function	✓	✓
Big-Vul [23]	real-world	Function	✓	✓
D2A [56]	real-world	Multi-function	✓	✓

Note: Since Juliet is synthetic, the project name and commit ID are missing.

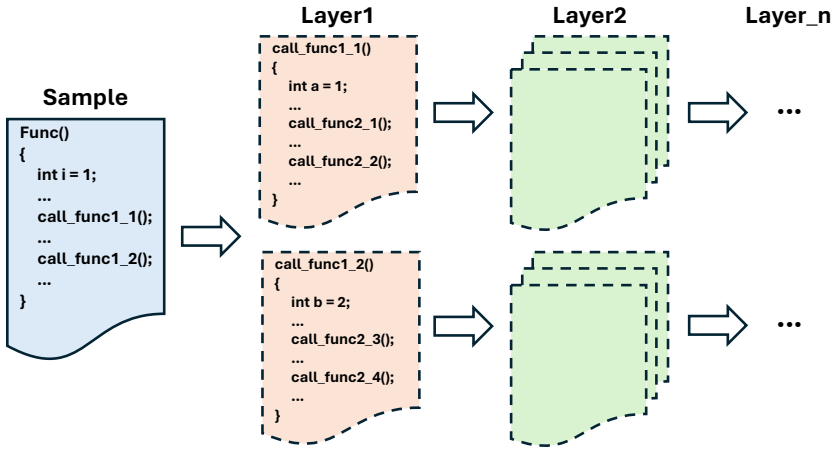


Fig. 4. A Given Sample.

These called functions, in turn, might invoke additional subsequent functions of their own. Initially, *VulnSC* uses the project name P to locate the specific repository on GitHub. Once the repository is identified, the commit ID CID is used to navigate to the exact state of the repository at that commit, ensuring the code snapshot matches the state at the time of the commit. This is achieved by checking out the repository to the specified commit. With the repository set to the correct version, we extract the source code of the functions called by the given sample, which can be determined by the call graph. This procedure is repeated iteratively: for each called function $CurFunc$, the procedure identifies and retrieves the source code of its called functions $CurCallFuncs$. Finally, we extract the source code of all directly and indirectly called functions CF and construct the call graph CG for the given sample.

3.2 Summary Generation

In this phase, *VulnSC* aims to generate concise summaries for called functions to capture inter-procedural semantics. Traditionally, summary-based techniques generate concise summaries to capture the semantics of individual functions within a program. These summaries can then be reused for inter-procedural analysis, eliminating the need to reanalyze the source code. Inspired by these techniques, we propose generating summaries for called functions, thereby enhancing function-level datasets. Given the remarkable success of LLMs in code-related tasks such as code summarization [13], particularly for high-resource programming languages like Python and

Algorithm 1: Retrieving Called Functions from GitHub

Input: Vulnerable Function F, Project name P, Commit ID CID
Output: Call graph CG, All Called Functions CF

Locate the GitHub repository using ProjectName P;
Check out repository P to commit ID CID;
Initialize $VisitedFuncs$ as an empty set;
Initialize $FuncQueue$ with Vulnerable Function F;
Insert Vulnerable Function F into $VisitedFuncs$;
Initialize CF as an empty set;
while *FuncQueue is not empty* **do**
    $CurFunc \leftarrow$ Dequeue $FuncQueue$;
    Build call graph for $CurFunc$, and update CG;
    $CurCallFuncs \leftarrow$ Retrieve called functions in $CurFunc$ from repository P;
    Add $CurCallFuncs$ to CF;
    for *CalledFunc in CurCallFuncs* **do**
        if *CallFunc is not in VisitedFuncs* **then**
            Enqueue $CallFunc$ into $FuncQueue$;
            Insert $CallFunc$ into $VisitedFuncs$;
return CG, CF;

Java [17], which benefit from extensive and diverse training datasets, enabling them to effectively understand and summarize code, we leverage LLMs with well-designed prompts to generate code summaries for called functions. This approach not only provides more comprehensive code semantics for subsequent learning-based vulnerability detection methods but also maintains the scalability of these approaches without significantly increasing the sample size.

3.2.1 Summary. Traditionally, summaries serve as structured representations that capture the essential details of a function including its inputs, outputs, and a description of its operations, simplifying the analysis of interactions between different procedures. This helps in tracking how data flows between functions and understanding the overall behavior of a program. The format of a summary can vary: *textual summaries* offer a detailed narrative explanation, while *vector-based summaries* use binary vectors to indicate the presence or absence of specific attributes. Additionally, *formula-based summaries* employ mathematical expressions to precisely represent the calculations of a function. An example is illustrated in Figure 5 to demonstrate how the summary works.

By abstracting away the detailed implementation of called functions, summary-based analysis can efficiently and scalably handle larger programs. In this paper, we propose generating summaries for called functions via LLMs. To achieve this, tailored summaries should be designed to capture critical function behaviors relevant to vulnerability detection tasks. Our framework focuses on generating summaries of called functions that highlight vulnerable behaviors. We begin by analyzing the vulnerabilities listed in the 2023 CWE (Common Weakness Enumeration) Top 25 Most Dangerous Software Weaknesses [9] and summarizing the common behaviors associated with these vulnerabilities, as detailed in Table 2. This analysis identifies 13 behavior types associated with vulnerabilities in the C programming language. We then use these behavior types to guide the design of prompts and leverage LLMs to generate summaries that emphasize these critical behaviors.

Table 2. Behavior Types of 2023 CWE Top 25 Most Dangerous Software Weaknesses.

CWE ID	Vulnerability Type	Behavior Type	Vulnerable Behavior
CWE-787	Out-of-bounds Write	Memory Management	Writing data beyond the bounds ...
CWE-79	Cross-site Scripting (XSS)	Input Validation	Injection of untrusted data into ...
CWE-89	SQL Injection	Input Validation	Improper neutralization of special ...
CWE-416	Use After Free	Memory Management	Accessing memory after it ...
CWE-78	OS Command Injection	Command Injection	Injection of special elements ...
CWE-20	Improper Input Validation	Input Validation	Failing to properly validate input ...
CWE-125	Out-of-bounds Read	Memory Management	Reading outside the boundaries of ...
CWE-22	Path Traversal	Path Management	Improper limitation of a pathname to ...
CWE-352	Cross-Site Request Forgery (CSRF)	Session Management	Tricking a user's browser into executing ...
CWE-434	Unrestricted Upload of File with Dangerous Type	File Handling	Allowing files to be uploaded without ...
CWE-862	Missing Authorization	Access Control	Failing to check if a user has ...
CWE-476	NULL Pointer Dereference	Memory Management	Dereferencing a null pointer, leading to application ...
CWE-287	Improper Authentication	Authentication	Allowing an attacker to bypass authentication ...
CWE-190	Integer Overflow or Wraparound	Arithmetic Operations	Performing arithmetic operations that exceed ...
CWE-502	Deserialization of Untrusted Data	Data Serialization	Deserializing data from an untrusted source ...
CWE-77	Command Injection	Command Injection	Injecting commands into a program through special ...
CWE-119	Improper Restriction of Operations within Memory Buffer	Memory Management	Allowing operations that exceed ...
CWE-798	Use of Hard-coded Credentials	Authentication	Embedding credentials directly in ...
CWE-918	Server-Side Request Forgery (SSRF)	Network Communication	Causing the server to make requests ...
CWE-306	Missing Authentication for Critical Function	Authentication	Allowing critical functions to be ...
CWE-362	Race Condition	Concurrency	Improper synchronization when ...
CWE-269	Improper Privilege Management	Access Control	Incorrectly managing user ...
CWE-94	Code Injection	Code Execution	Injecting code into an application ...
CWE-863	Incorrect Authorization	Access Control	Failing to properly enforce user ...
CWE-276	Incorrect Default Permissions	Access Control	Setting overly permissive default permissions ...

```c
int find_max(const int *array, size_t length) {
    if (length == 0) {
        return -1;
    }
    int max = array[0];
    for (size_t i = 1; i < length; ++i) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
```

(a) A Function Example

```
Input: const int *array, size_t length
Output: int max
Behaviors:
• Checks if length is zero. If so, returns -1.
• Initializes max with the first element of the array.
• Iterates through the array, comparing each element with max.
• Updates max if a larger value is found.
• After processing all elements, returns max.
```

(b) A Summary

Fig. 5. A Function Example and Its Summary.

3.2.2 Prompt Design. Given a function with its source code, we aim to instruct LLMs to generate summaries that concisely abstract potential behaviors related to vulnerabilities for the function using a well-formed prompt. To achieve this, we adopt the prompt design approach from previous work [27, 31], utilizing the system prompt to establish the context for the conversation and the user prompt to provide specific details.

Specifically, we design the system prompt by assigning it the persona of an advanced code summarization tool, describing the task, and ensuring the LLMs provide responses in the expected structure, as detailed in Listing 1.

```
You are an advanced code summarization tool.

Given the source code of a function, generate a summary for the function.

Provide response in the following format: input:<INPUTS> | output:<OUTPUTS> |
    behavior:<BEHAVIOR1,>, <BEHAVIOR2>, ... where <INPUTS> lists the
    parameters the function accepts, <OUTPUTS> lists the return values of the
    function, and <BEHAVIOR> concisely describes the behavior of the function
    in conjunction with the variables of input and output, without any other
    details other than the operations performed.
```

Listing 1. System Prompt for Summary Generation.

For the user prompt, we utilize the vulnerable behavior types outlined in Table 2 and the source code obtained from code retrieval to develop four prompting strategies, described as follows:

Basic prompting: We design a basic prompt that simply takes the source code of a given function as input and instructs the LLMs to generate a concise summary. The prompt begins with the message "Only provide the response in the format mentioned before for the following code snippet" followed by the target code snippet.
Behavior-guided prompting: As we mentioned above, the key to the summary is to capture behaviors related to our tasks. To achieve this, 13 vulnerable behavior types in Table 2 are summarized from the vulnerabilities in the 2023 CWE (Common Weakness Enumeration) Top 25 Most Dangerous Software Weaknesses. We use them to design a behavior-guided prompt that instructs LLMs to focus on summarizing these behavior types. Hence, the prompt starts with the message "The behaviors should fall into the following behavior types" followed by the list of 13 vulnerable behavior types and their descriptions. Similar to the basic prompt, the prompt also ends with the message "Only describes in detail the behavior of the function along with the input and output variables, without specifying the behavior types. Only provide the response in the format mentioned before for the following code snippet" followed by the target code snippet to clarify the goal of generating a summary for the target code.
One-shot prompt prompting: To design the one-shot prompt, we incorporate an example at the beginning of the basic prompt, which includes a vulnerable code snippet and its corresponding behavior. This example helps LLMs better understand the task by providing a clear reference.
Chain-of-thought prompting: In the chain-of-thought prompt, we include the statement "Let us think step by step" at the beginning of the basic prompt. This approach encourages LLMs to reason through the task and generate the summary in a step-by-step manner.

3.2.3 Bottom-Up Algorithm. In this paper, we aim to generate summaries for the called functions of a given sample and use them to enhance the inter-procedural semantics of the sample. However, as illustrated in Figure 4, called functions may also invoke additional functions. To address this, we propose a bottom-up algorithm to generate summaries for the called functions, progressing from the functions in the bottom layer to those in the top layer, as depicted in Figure 6. Here, a layer refers to the depth level of a function, where Layer 0 contains entry-point functions, Layer N contains functions invoked by those in Layer (N-1). The algorithm follows these steps:

Step 1: Given a function-level sample f, we complement its called functions and construct its call graph cg through the code retrieval process, where edges represent the call relationships between functions, while nodes correspond to the functions of callers and callees, e.g., the called functions, as depicted in step 1 of Figure 6.

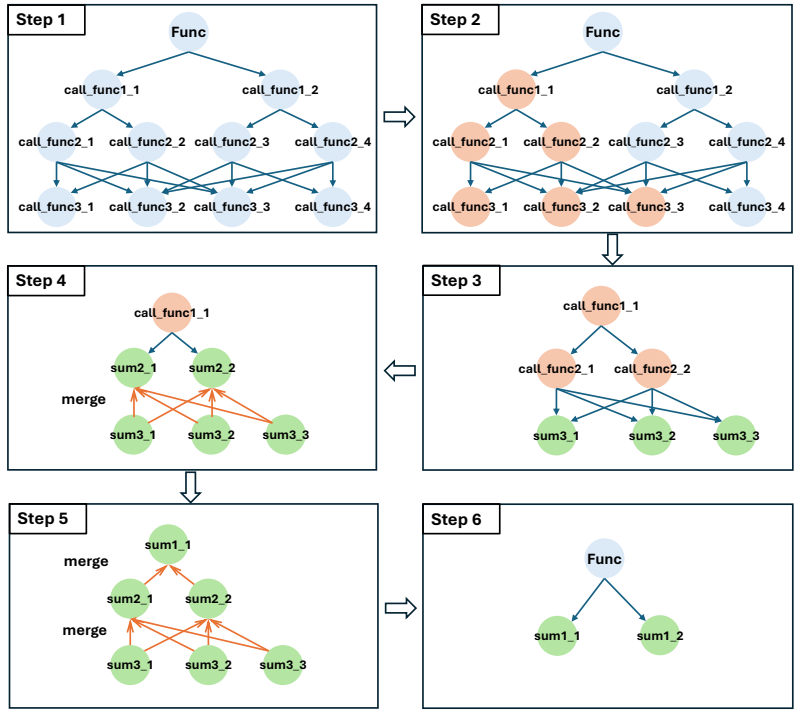


Fig. 6. Bottom-Up Algorithm for Summary Generation.

Step 2: We identify all directly called functions of the sample. For each directly called function cf, we perform a depth-first search on cg to retrieve all its directly and indirectly called functions, forming a tree structure tr as illustrated in the orange part of step 2 in Figure 6.
Step 3: For each directly called function cf, we identify all leaf nodes ln in its tree tr and use LLMs with the designed prompt to generate summaries sum for these leaf nodes, as depicted in step 3 of Figure 6.
Step 4: For each leaf node ln in its tree tr, we first identify its parent node pn using the call graph cg. We then merge the summary sum of ln into the source code of pn, resulting in an updated parent node pn_m. This procedure is repeated for all leaf nodes to obtain the updated parent nodes. Subsequently, we use LLMs with the designed prompt to generate summaries sum for these updated parent nodes, as shown in step 4 of Figure 6.
Step 5: We repeat the procedure outlined in Step 3 until the parent node pn is cf, obtaining the summary sum for each directly called function cf.
Step 6: We repeat the procedures from Steps 3 to 5 to obtain the summaries for all directly called functions. Finally, we update the statements of all directly called functions in f with their corresponding summaries, resulting in the updated sample f_m that is enhanced with inter-procedural semantics, as depicted in step 6 of Figure 6.

3.3 Learning-Based Approaches

Learning-based approaches utilize neural models to learn representations from code snippets for vulnerability detection. In this paper, we enhance these approaches by generating summaries that supplement the inter-procedural semantics of the given code snippets. Since the generated summaries are in natural language, we ensure compatibility with existing learning-based methods

by embedding the summaries of the called functions as comments appended to the corresponding samples, as illustrated in Figure 3. These enhanced samples can then be seamlessly fed into existing learning-based models for vulnerability detection. Our framework, *VulnSC*, formats the generated summaries into function-specific comments and integrates them into the original samples. *VulnSC* not only enriches the inter-procedural semantics but also maintains compatibility with existing learning-based approaches. By leveraging these enhanced samples, existing models can achieve more effective and scalable vulnerability detection.

4 Experimental Setup

4.1 Dataset

We select two widely used open-source vulnerability dataset Devign [57] and Big-Vul [23] as our evaluation datasets. Devign is a balanced dataset containing more than 20,000 examples, collected from two large C-language projects: Qemu and FFmpeg, and Big-Vul is an unbalanced dataset containing 253,096 functions tagged as non-vulnerable and 11,823 functions tagged as vulnerable. We conduct the following steps to preprocess the datasets for the evaluation:

1) Enhancing the dataset. In this paper, *VulnSC* initially retrieves the source code of the called functions for the samples in Devign and Big-Vul. During this process, we discover that the commit IDs in some samples do not match their corresponding functions, leading to failures in retrieving the called functions. Consequently, we exclude these samples from our evaluation, as they cannot be used to assess the effectiveness of *VulnSC* in enhancing inter-procedural semantics. Subsequently, *VulnSC* utilizes LLMs with four prompting strategies, as detailed in Section 3.2, to generate concise summaries for these functions. These summaries are then used to augment the original dataset, resulting in four new versions of Devign: devign-basic, devign-behavior, devign-oneshot, and devign-cot, as well as four new versions of Big-Vul: Big-Vul-basic, Big-Vul-behavior, Big-Vul-oneshot, and Big-Vul-cot. This augmentation yields eight new dataset versions in total, in addition to the original datasets. Each version of Devign contains 19,857 samples, including 9,338 samples tagged as vulnerable and 10,519 samples tagged as non-vulnerable, while each version of Big-Vul comprises 62,016 samples, with 4,086 samples tagged as vulnerable and 57,930 samples tagged as non-vulnerable.

2) Splitting the dataset. The original datasets Devign and Big-Vul, do not include predefined splits for training, testing, and validation. Consequently, we partition both vulnerable and non-vulnerable samples into training, validation, and test sets using an 80%:10%:10% ratio. These partitions are then combined to form the final datasets for training, validation, and testing.

4.2 Baseline

To evaluate the effectiveness of *VulnSC* in enhancing existing function-level learning-based approaches for vulnerability detection, we conduct experiments on four state-of-the-art approaches: CodeBERT [24], GraphCodeBERT [29], UniXcoder [28], and LineVul [26]. These baselines utilize advanced transformer-based pre-training models for identifying vulnerabilities and have demonstrated impressive results. In addition, we compare *VulnSC* with three existing LLM-based vulnerability detection approaches: LLM4Vuln [51], Avishree [31], and GRACE [39]. LLM4Vuln leverages large language models to enhance vulnerability detection through prompt engineering and knowledge integration, Avishree focuses on understanding the effectiveness of LLMs in detecting software vulnerabilities, and GRACE enhances LLM-based software vulnerability detection by leveraging contextual understanding and reasoning capabilities.

4.3 Implementation

For the model implementation, we implement the models in the baselines using PyTorch 2.4.0 with CUDA version 12.4. All models are fine-tuned on three Tesla V100 GPU, paired with an Intel® Xeon® 6248 CPU and 188GB of RAM. The models are trained with our training data using a learning rate of 2e-5 and a batch size of 16. We train the models using our training data and select the best fine-tuned weights based on the accuracy achieved on the validation set. These optimal weights are then used for evaluating the performance on our test set.

For the LLM configuration, we select four representative models—GPT-4o, CodeLlama, DeepSeek, and Mixtral—for our experiments. These models have demonstrated strong performance across various coding benchmarks, making them well-suited for our evaluation. Specifically, we use the OpenAI API to query GPT-4o-2024-08-06 and the Replicate API to query CodeLlama-13B, DeepSeek-V3, and Mixtral-8x7B, employing the prompts designed in Section 3.2 to generate code summaries. To ensure consistency in responses, we set the temperature to 0. Additionally, we limit the maximum number of tokens to 1024 while keeping all other parameters at their default values.

5 Evaluation

In this section, we first assess the quality of code summaries generated by *VulnSC*. We then investigate whether incorporating these summaries into existing learning-based approaches enhances vulnerability detection performance. Additionally, we perform an ablation study to analyze the impact of different LLM selections and prompt strategies on effectiveness. Finally, we examine the scalability of *VulnSC* when integrated with existing learning-based approaches for vulnerability detection. The research questions are as follows:

5.1 Quality of Generated Summaries (RQ1)

Experimental Design: In this research question, we aim to assess the quality of code summaries generated by *VulnSC*. A key challenge in this evaluation is the lack of reference summaries in Devign and Big-Vul datasets used in our study. To address this, we design two complementary experiments to systematically evaluate the quality of summaries generated by *VulnSC*.

First, we assess whether *VulnSC* can generate high-quality summaries by evaluating it on a publicly available code summarization dataset[38], which includes ground-truth summaries for each function. We compare *VulnSC* against four state-of-the-art code summarization approaches (i.e., Transformer[12], Rencos [55], SeqGNN [25], and Shang [38]) using two widely adopted evaluation metrics, BLEU-4 and METEOR. This experiment provides a quantitative assessment of the ability of *VulnSC* to generate summaries that align with reference annotations.

Second, to evaluate the quality of summaries generated for functions in our target datasets (Devign and Big-Vul), we conduct a human study. We recruit 15 volunteers with extensive C programming experience, including 5 PhD students, 5 master students, and 5 professional software engineers. Each participant is asked to rank code summaries on a five-point scale (1: Poor, 2: Marginal, 3: Acceptable, 4: Good, 5: Excellent) based on their relevance to the corresponding source code. To ensure an objective assessment, we randomly select 80 functions with summaries generated by *VulnSC* and 80 functions from the public dataset [38] with reference summaries. The final evaluation is based on the average rating, where a higher score indicates better summary quality.

Experimental Result: From Table 3, we observe that *VulnSC* significantly outperforms traditional baselines such as Transformer, Rencos, SeqGNN, and Shang. Among the tested models, DeepSeek achieves the highest BLEU-4 score (17.42) when using the basic prompt, while CodeLlama attains the highest METEOR score (17.03) under the same setting. This suggests that different LLMs

Table 3. Code Summary Evaluation.

Method	LLM	Prompt	BLEU-4	METEOR
Transformer	-	-	5.75	9.89
Rencos	-	-	7.54	10.35
SeqGNN	-	-	4.94	9.50
Shang	-	-	7.85	11.05
VulnSC	GPT-4o	basic	8.95	13.53
		behavior	8.18	12.94
		one-shot	**11.52**	13.14
		CoT	8.43	13.08
	CodeLlama	basic	**17.08**	17.03
		behavior	13.50	14.87
		one-shot	15.03	14.70
		CoT	15.81	16.22
	DeepSeek	basic	**17.42**	14.67
		behavior	9.56	10.54
		one-shot	15.72	12.40
		CoT	17.30	15.34
	Mixtral	basic	9.73	16.16
		behavior	8.02	15.08
		one-shot	**11.36**	14.07
		CoT	9.07	15.91

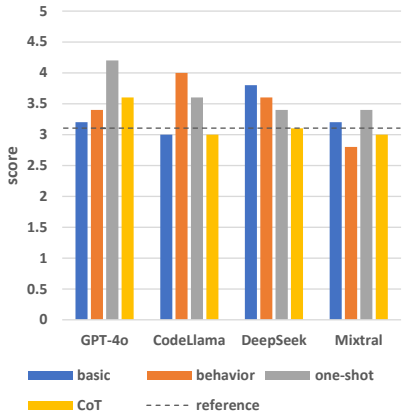


Fig. 7. Human Study.

exhibit distinct strengths in code summarization, with DeepSeek excelling in fluency and CodeLlama in semantic alignment with reference summaries. Notably, the behavior prompt consistently yields lower BLEU-4 scores across all LLMs, likely due to its explicit focus on vulnerability-related information. Since the reference summaries in the evaluation dataset are more general and not tailored specifically to security aspects, the additional vulnerability-specific details in behavior-prompted summaries may introduce semantic deviations, resulting in lower BLEU-4 scores. Nevertheless, the strong METEOR scores indicate that behavior-prompted summaries still maintain good semantic relevance. Overall, these results highlight that *VulnSC* achieves better performance in code summarization compared to traditional learning-based approaches.

Figure 7 presents the evaluation results from the human study. The reference score, represented by the gray dashed line, corresponds to the average rating of 80 functions from the public dataset as assessed by human participants, serving as a benchmark for comparison. The results show that summaries generated by *VulnSC* achieve ratings consistently between "acceptable" and "good", validating their quality. Among the prompting strategies, the one-shot prompt receives the highest human evaluation scores, particularly for GPT-4o, indicating that providing in-context examples significantly enhances human-perceived relevance and clarity. The behavior prompt, despite lower BLEU-4 scores, receives relatively high ratings in human evaluation for some models, suggesting that while it differs from reference summaries, it may provide useful security insights. Additionally, GPT-4o and DeepSeek consistently achieve high ratings across different prompting strategies, reinforcing their effectiveness in code summarization.

Overall, these findings confirm that *VulnSC* generates high-quality summaries across different LLMs and prompting strategies, consistently achieving strong performance in both automated and human evaluations. This establishes a solid foundation for leveraging these summaries as inter-procedural semantics to enhance learning-based vulnerability detection approaches.

Answer to RQ1: Automated metrics and human evaluations confirm that *VulnSC*, leveraging different LLMs and prompting strategies, generates high-quality summaries, effectively serving as inter-procedural semantics for vulnerability detection.

Table 4. Detection Performance of Different Approaches.

Method	LLM	Prompt	Devign			Big-Vul		
			Acc(%)	Pre(%)	Rec(%)	F1(%)	Pre(%)	Rec(%)
LLM4Vuln	GPT-4	raw	52.37	53.75	72.15	8.56	5.44	20.05
		pre-CoT	51.56	53.01	75.29	10.93	6.92	25.92
		post-CoT	52.17	53.56	72.91	9.82	6.15	24.45
	CodeLlama	raw	52.32	53.56	75.10	8.12	4.98	22.00
		pre-CoT	51.16	53.25	63.78	8.62	4.98	32.03
		post-CoT	52.22	54.14	64.07	8.26	4.80	29.83
	Mixtral	raw	52.92	52.98	98.95	10.43	5.58	80.68
		pre-CoT	53.22	53.17	98.19	10.23	5.47	78.97
		post-CoT	47.53	52.38	10.46	10.20	5.46	78.48
Avishree	GPT-4	basic	52.22	54.57	58.46	10.18	5.90	36.92
		CWE	51.06	54.31	47.91	10.12	5.81	39.12
		CoT	49.95	53.25	45.15	9.04	5.23	33.25
		analysis	49.95	52.40	60.17	9.48	5.28	46.70
	CodeLlama	basic	50.35	52.71	61.03	10.19	5.91	37.16
		CWE	49.24	51.79	60.46	10.31	5.75	49.88
		CoT	50.50	52.87	60.46	10.35	5.79	49.14
		analysis	50.05	52.99	50.48	10.07	5.85	36.19
GRACE	-	-	59.78	53.94	82.13	35.50	32.52	39.08
CodeBERT	-	-	58.16	59.95	63.31	33.56	57.39	23.71
GraphCodeBERT	-	-	58.86	64.35	50.10	34.73	55.97	25.18
UniXcoder	-	-	60.57	62.24	65.02	36.66	53.52	27.87
LineVul	-	-	61.48	59.88	54.82	81.90	92.33	73.59
VulnSC (CodeBERT)	GPT-4o	basic	61.98	65.02	61.12	36.42	52.53	27.87
		behavior	**62.89**	62.05	77.09	37.00	49.38	29.58
		one-shot	62.49	62.51	72.91	37.13	49.19	29.82
		CoT	62.19	62.47	71.67	**37.36**	52.19	29.09
VulnSC (GraphCodeBERT)	GPT-4o	basic	60.93	62.06	67.49	36.90	56.56	27.38
		behavior	61.73	61.51	74.14	38.04	56.79	28.60
		one-shot	**62.69**	63.82	68.25	**38.48**	58.79	28.60
		CoT	62.03	62.87	69.20	37.28	60.77	26.89
VulnSC (UniXcoder)	GPT-4o	basic	63.49	62.00	80.32	40.93	46.70	36.43
		behavior	63.65	65.00	67.97	41.58	42.10	41.07
		one-shot	62.74	63.93	68.06	**41.75**	41.70	41.80
		CoT	**63.85**	65.58	66.83	41.20	46.60	36.91
VulnSC (LineVul)	GPT-4o	basic	63.29	65.28	46.90	**84.29**	93.45	76.77
		behavior	62.74	62.56	51.71	84.21	93.97	76.28
		one-shot	62.59	62.68	50.54	83.67	93.37	75.79
		CoT	**63.54**	65.00	48.72	84.16	94.24	76.03

5.2 Improvement in Detection Performance (RQ2)

Experimental Design: In this research question, we aim to assess the effectiveness of *VulnSC* in enhancing existing learning-based approaches for vulnerability detection. To this end, we evaluate *VulnSC* across four state-of-the-art learning-based models: CodeBERT, GraphCodeBERT, UniXcoder, and LineVul. For this experiment, we select GPT-4o as the representative LLM to demonstrate the improvements in detection performance brought by *VulnSC*. The performance of other LLMs (CodeLlama, DeepSeek, and Mixtral) is further analyzed in RQ3 to examine their impact on vulnerability detection.

To conduct a fair and comprehensive evaluation, we first re-implement each of the four learning-based approaches using their default settings and evaluate their performance on five versions of the Devign and Big-Vul datasets. The original versions of these datasets remain unchanged, while the four additional versions are enhanced by *VulnSC* using GPT-4o with four different prompting strategies. By comparing the results between the original and enhanced datasets, we can quantify the effectiveness of *VulnSC* in improving vulnerability detection.

We measure performance using four widely adopted metrics: accuracy, precision, recall, and F1-score. Each model is trained and evaluated five times with different random seeds, and the final performance is reported as the average score over these runs. Notably, accuracy is the most

appropriate evaluation metric for balanced datasets, whereas for imbalanced datasets, F1-score is preferred, as it provides a more reliable measure of overall model performance.

Additionally, to further validate the effectiveness of *VulnSC*, we compare it with existing LLM-based vulnerability detection approaches. Specifically, we re-implement three representative LLM-based methods—LLM4Vuln, Avishree, and GRACE—as baselines, and compare their performance against *VulnSC* to assess its relative advantages in detecting vulnerabilities.

Experimental Result: From Table 4, we observe that *VulnSC* significantly improves the effectiveness of all learning-based approaches for vulnerability detection across all prompting strategies. Specifically, on Devign dataset, *VulnSC* increases the accuracy of the original CodeBERT by 3.82 to 4.73 percentage points, GraphCodeBERT by 2.07 to 3.83 percentage points, UniXcoder by 2.17 to 3.28 percentage points, and LineVul by 1.11 to 2.06 percentage points with the four prompting strategies. Among these models, CodeBERT with the behavior-guided prompting strategy experiences the highest increase of 4.73 percentage points in accuracy, while UniXcoder with the Chain-of-Thought prompting strategy achieves the highest accuracy of 63.85%. On Big-Vul dataset, *VulnSC* improves the F1-score of all models by 1.77 to 5.09 percentage points across all prompting strategies. In particular, by leveraging the four prompting strategies, CodeBERT improves F1-score by 2.86 to 3.80 percentage points, GraphCodeBERT by 2.17 to 3.75 percentage points, UniXcoder by 4.27 to 5.09 percentage points and LineVul by 1.77 to 2.39 percentage points.

We also observe that the lower the performance of the original models, the greater the improvement *VulnSC* provides. Specifically, the original CodeBERT, which has the lowest baseline accuracy of 58.16% on Devign dataset, benefits the most from *VulnSC*, achieving a significant accuracy increase of 4.73 percentage points. In contrast, the original LineVul, which starts with the highest baseline accuracy of 61.48% on Devign dataset, shows the least improvement among all models, with a maximum increase of only 2.06 percentage points. For the remaining models, GraphCodeBERT, with an original accuracy of 58.86%, experiences a more notable increase compared to UniXcoder, which has an original accuracy of 60.57%. The accuracy gains for GraphCodeBERT and UniXcoder with *VulnSC* are less than 3.83 percentage points and 3.28 percentage points, respectively. Similar to the performance improvement on the Devign dataset, CodeBERT, which has the lowest baseline F1-score of 33.56% on the Big-Vul dataset, a significant improvement of 3.8 percentage points, marking the second-largest increase among all models. Conversely, the original LineVul with the highest F1-score of 81.90% shows a more modest improvement, ranging from 1.77 to 2.39 percentage points, which is smaller than the increases seen with other approaches. This pattern suggests that the *VulnSC* has a more pronounced effect on models with lower initial performance, providing a larger relative gain where there is more room for improvement.

Finally, the performance gains achieved with *VulnSC* using GPT-4o are even more significant than those achieved by improving the complicated model architectures. As indicated in Table 4, GraphCodeBERT, which builds upon CodeBERT by incorporating additional graph-based information like data flow into the model, results in only a 0.7 percentage points increase in accuracy and a 1.17 percentage points increase in F1-score. Similarly, UniXcoder and LineVul show improvements of 2.41 percentage points and 3.32 percentage points in accuracy over CodeBERT, respectively. In contrast, *VulnSC* offers a more significant enhancement, increasing the accuracy of existing models by up to 4.73 percentage points and F1-score by up to 5.09 percentage points. This notable improvement is attributed to the inter-procedural semantic completion by *VulnSC*, which adds valuable contextual information to the datasets. Despite the advancements in model architectures, predicting vulnerabilities remains challenging without sufficient inter-procedural context. *VulnSC* addresses this by enriching the dataset with inter-procedural information, offering a cost-effective and broadly applicable solution. By significantly enhancing the effectiveness of existing models, *VulnSC* proves to be a valuable and effective addition to vulnerability detection approaches.

Additionally, learning-based approaches enhanced by *VulnSC* significantly outperform the three existing LLM-based vulnerability detection approaches (LLM4Vuln, Avishree, and GRACE) in terms of accuracy and F1-score. LLM4Vuln, which utilizes GPT-4, CodeLlama, and Mixtral with raw, pre-CoT, and post-CoT prompts, achieves a maximum accuracy of 53.22% on Devign and an F1-score of 10.93% on Big-Vul, demonstrating limited effectiveness in vulnerability detection. Avishree, which incorporates CWE, CoT, and dataflow analysis-based prompting strategies, performs similarly, with a maximum accuracy of 52.22% on Devign and an F1-score of 10.35% on Big-Vul. GRACE achieves slightly better results, with an accuracy of 59.78% on Devign and an F1-score of 35.50% on Big-Vul, yet still falls short in delivering substantial performance gains. In contrast, *VulnSC* delivers a significant performance boost across all learning-based approaches and datasets, far surpassing the results of LLM4Vuln, Avishree, and GRACE. Specifically, when applied to LineVul with the CoT prompt, *VulnSC* achieves an accuracy of 63.54% on Devign and an F1-score of 84.16%, surpassing GRACE by 3.76 percentage points in accuracy and 48.66 percentage points in F1-score. These results demonstrate the effectiveness of *VulnSC* in leveraging code summaries as inter-procedural semantics to enhance vulnerability detection, making it a far superior alternative to existing LLM-based methods.

Answer to RQ2: *VulnSC* enhances the effectiveness of learning-based approaches across all prompting strategies. Compared to existing LLM-based vulnerability detection approaches, *VulnSC* demonstrates clear superiority, outperforming them across both accuracy and F1 score.

5.3 Impact of LLM Selection and Prompt Strategies (RQ3)

Experimental Design: In this research question, we conduct an ablation study to assess the impact of LLMs and prompting strategies on the performance improvements achieved by *VulnSC*. To analyze the effect of different LLMs, we fix the prompt to basic prompting strategy and evaluate the performance enhancement brought by *VulnSC* using GPT-4o, CodeLlama, DeepSeek, and Mixtral across the Devign and Big-Vul datasets. Subsequently, we fix the LLM to GPT-4o and examine the impact of different prompting strategies, focusing on accuracy and F1-score, to determine their effectiveness in improving vulnerability detection performance.

Experimental results: Table 5 presents the detection performance of *VulnSC* with different LLMs under the basic prompt. The results indicate that all tested LLMs improve performance, but the magnitude of improvement varies. DeepSeek and GPT-4o consistently achieve the highest performance improvements, while Mixtral exhibits relatively smaller gains. On Devign, *VulnSC* achieves the highest accuracy improvement (ΔAcc=4.02 percentage points) when applied to CodeBERT with DeepSeek. On Big-Vul, UniXcoder combined with GPT-4o achieves the best performance improvement, with an improvement in F1-score of 4.27 percentage points. These results suggest that LLM choice significantly influences the effectiveness of *VulnSC*, with GPT-4o and DeepSeek leading to the greatest improvements.

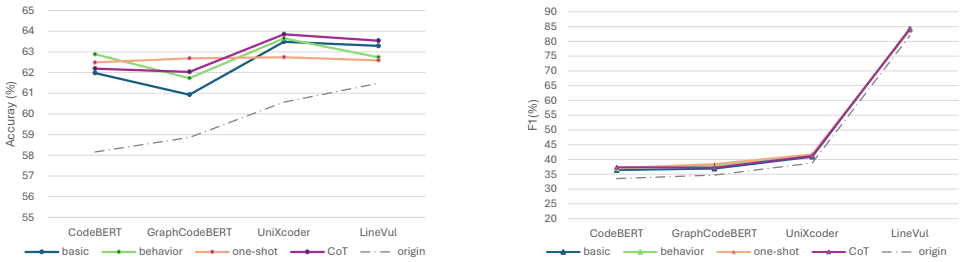
Figure 8 illustrates that different prompting strategies under GPT-4o result in varying performance increases in accuracy and F1 score across all models. In general, the Chain-of-Thought prompting strategy and behavior-guided prompting strategy achieve relatively higher increases in accuracy on Devign dataset, whereas the basic prompting strategy results in the lowest increase, and the one-shot prompting strategy maintains stable accuracy across all models. Regarding the F1-score from Big-Vul dataset, all strategies show similar performance across models, as depicted in Figure 8(b), with the exception of the basic prompting strategy, which exhibits a relatively small increase for CodeBERT and a relatively higher increase for LineVul. This performance aligns with the design of the prompting strategies: the basic prompting strategy provides simple instruction for

3

Table 5. Performance of *VulnSC* with Different LLMs Using the Basic Prompt on Devign and Big-Vul.

Method	LLM	Devign				Big-Vul			
		Acc(%)	ΔAcc(%)	Pre(%)	Rec(%)	F1(%)	ΔF1(%)	Pre(%)	Rec(%)
VulnSC(CodeBERT)	GPT-4o	61.98	3.82	65.02	61.12	36.42	2.86	52.53	27.87
	CodeLlama	61.67	3.51	61.12	75.36	35.85	2.29	56.91	26.16
	DeepSeek	62.18	**4.02**	62.80	69.61	37.31	**3.75**	59.68	27.14
	Mixtral	61.97	3.81	61.44	75.17	36.07	2.51	54.73	26.89
VulnSC(GraphCodeBERT)	GPT-4o	60.93	2.07	62.06	67.49	36.90	2.17	56.56	27.38
	CodeLlama	61.27	2.41	62.61	66.16	37.74	**3.01**	55.45	28.61
	DeepSeek	61.52	**2.66**	64.19	61.36	37.17	2.44	56.78	27.63
	Mixtral	61.16	2.30	62.30	67.02	36.89	2.16	54.55	27.87
VulnSC(UniXcoder)	GPT-4o	63.49	**2.92**	62.00	80.32	40.93	**4.27**	46.70	36.43
	CodeLlama	63.09	2.52	63.80	69.61	39.06	2.40	54.87	30.32
	DeepSeek	63.19	2.62	65.61	63.66	39.45	2.79	52.00	31.78
	Mixtral	62.78	2.21	63.85	68.07	38.34	1.68	51.44	30.56
VulnSC(LineVul)	GPT-4o	63.14	1.66	63.34	71.72	84.29	**2.30**	93.45	76.77
	CodeLlama	62.48	1.00	60.37	84.27	83.11	1.21	93.85	74.57
	DeepSeek	63.59	**2.11**	62.80	76.22	83.63	1.73	95.60	74.33
	Mixtral	61.97	0.49	60.85	78.52	82.88	0.98	93.27	74.57

Note: ΔAcc and ΔF1 denote the performance improvement with *VulnSC* over the original approaches.



(a) Accuracy under Different Prompting Strategies

(b) F1 under Different Prompting Strategies

Fig. 8. Detection Performance of Different Prompting Strategies under GPT-4o.

LLMs to generate summaries of called functions, while the behavior-guided and one-shot strategies supply LLMs with vulnerable behaviors and an example, respectively. The Chain-of-Thought prompting strategy directs LLMs to generate summaries through a step-by-step reasoning process.

Interestingly, when analyzing the relationship between detection performance improvement and the quality of summaries generated by different LLMs, we find no direct correlation. From RQ1, CodeLlama achieves the highest BLEU-4 and METEOR scores, while DeepSeek receives the highest human evaluation score under the basic prompting strategy. However, in this section, we observe that GPT-4o and DeepSeek—rather than CodeLlama—yield the most significant improvements in vulnerability detection performance. This suggests that higher lexical similarity to reference summaries (as measured by BLEU-4 and METEOR) does not necessarily lead to better detection performance. Instead, other factors, such as the informativeness and relevance of generated summaries, play a more critical role in enhancing detection effectiveness. This is further reinforced by the observation that the behavior-guided prompting strategy, despite having lower BLEU scores, leads to relatively higher improvements in detection performance, highlighting the importance of security-relevant information over textual similarity in vulnerability detection.

Table 6. Computational Overhead Before and After Enhancement by GPT-4o.

			Devign					Big-Vul				
Prompt Strategy			origin	basic	behavior	one-shot	CoT	origin	basic	behavior	one-shot	CoT
data size (MB)			45.27	65.18	**68.63**	64.61	63.92	210	229	244	232	230
CodeBERT	time	train (s)	889.49	916.11	**922.37**	914.25	916.03	2724.65	2784.46	2809.55	2790.75	2797.51
		test (s)	37.11	41.86	**42.02**	41.40	40.34	78.94	88.23	90.05	88.06	87.95
	GPU(MiB)		14154	14154	14154	14154	14154	25284	25284	25284	25284	25284
GraphCodeBERT	time	train (s)	893.74	917.34	**923.89**	917.59	920.30	2732.07	2790.05	2818.18	2797.15	2800.36
		test (s)	37.48	41.09	**42.18**	41.29	41.34	78.20	90.61	93.32	88.88	89.91
	GPU(MiB)		14154	14154	14154	14154	14154	25284	25284	25284	25284	25284
UniXcoder	time	train (s)	895.37	918.63	**969.99**	959.80	918.21	2728.79	2796.47	2811.00	2799.57	2795.51
		test (s)	37.12	42.18	**42.30**	41.67	40.32	77.10	87.75	90.71	88.35	88.11
	GPU(MiB)		14200	14200	14200	14200	14200	25574	25574	25574	25574	25574
LineVul	time	train (s)	1478.33	1503.49	**1515.05**	1499.55	1506.97	4337.87	4689.81	4702.80	4683.96	4695.71
		test (s)	37.35	39.39	**39.94**	39.64	39.54	77.25	88.09	90.79	88.35	88.25
	GPU(MiB)		20532	20532	20532	20532	20532	32670	32670	32670	32670	32670

Answer to RQ3: LLM selection and prompting strategies significantly impact the effectiveness of *VulnSC*, with GPT-4o and DeepSeek yielding the greatest improvements, while the Chain-of-Thought and behavior-guided prompting strategies lead to relatively higher gains in accuracy and F1-score in most cases.

5.4 Scalability with Learning-Based Approaches (RQ4)

Experimental Design: To assess the scalability of *VulnSC* with existing learning-based approaches, we measure computational time and GPU resource usage before and after integrating *VulnSC* with four different prompting strategies under GPT-4o. Specifically, we assess four baseline models using both the original dataset and four enhanced versions augmented by *VulnSC*. For each model and dataset combination, we record the total training and testing time as well as GPU memory utilization. By comparing these metrics between the original and enhanced datasets, we evaluate how the integration of *VulnSC* influences both time efficiency and GPU resource demands.

Experimental Results: Table 6 demonstrates that the total training and testing times experience only minimal increases across all learning-based approaches after applying *VulnSC* with the four prompting strategies, while GPU memory usage remains unchanged. Specifically, For Devign dataset, the training time increases for CodeBERT by 24.77 to 32.88 seconds and the testing time by 3.23 to 4.91 seconds; for GraphCodeBERT, the increases are 23.6 to 30.15 seconds and 3.61 to 4.7 seconds; for UniXcoder, the increases are 22.84 to 74.63 seconds and 3.2 to 5.18 seconds; and for LineVul, the increases are 21.22 to 36.72 seconds and 2.04 to 2.59 seconds. Similarly, for Big-Vul dataset, CodeBERT increases the training time by 59.81 to 84.9 seconds, and the test time by 9.01 to 11.11 seconds. Compared to the total processing time of over 900 seconds, these increases are minimal. Additionally, there is no increase in GPU memory usage after applying *VulnSC*. These findings demonstrate that the *VulnSC* is highly scalable and suitable for real-world applications.

Different prompting strategies introduce varying training and testing time overheads. Among them, the behavior-guided prompting strategy consumes the most computational resources across all learning-based approaches. This may be attributed to the fact that the behavior-guided prompting strategy provides LLMs with detailed information on vulnerable behaviors, leading to more extensive summaries compared to other strategies. As illustrated in Table 6, the data size for the behavior-guided prompting strategy is the largest among the strategies, at 68.63 MB and 244MB.

We also observe that the increase in total training and testing time as well as GPU memory usage caused by *VulnSC* can be much smaller than the increase caused by model upgrade. For instance, upgrading from CodeBERT, with a baseline accuracy of 58.16%, to LineVul, which has a baseline accuracy of 61.48%, results in an increase of 588.84 seconds in training time and a rise of 6,378 MiB in GPU memory usage. In contrast, applying *VulnSC* with the behavior prompting strategy to CodeBERT—achieving a higher accuracy of 62.89%—only increases training time by 32.88 seconds and GPU memory usage remains unchanged. This comparison demonstrates that *VulnSC* introduces minimal overhead compared to the significant resource demands associated with model upgrades.

Answer to RQ4: *VulnSC* introduces minimal increases in total training and testing times, as well as GPU memory usage, across all models. The behavior-guided prompting strategy incurs the highest cost due to more detailed vulnerability descriptions. Compared to model upgrades, *VulnSC* may cause less increase in training and testing time, as well as GPU memory usage.

6 Threats to Validity

Internal validity: (1) Incomplete coverage of vulnerable behaviors: Our behavior-guided prompting considers 13 types of vulnerable behaviors derived from the vulnerabilities in the 2023 CWE Top 25 Most Dangerous Software Weaknesses. These behavior types can be used to assist LLMs in summarizing vulnerability-related actions relevant to vulnerability detection tasks. However, this selection may not encompass all possible vulnerable behaviors. As new types of vulnerabilities emerge, we can address this limitation by incorporating these new behaviors into the prompt, thereby guiding the LLM in generating more comprehensive summaries. (2) Potential deadlock in Algorithm 1 for code retrieval: The algorithm is designed to extract called functions based on the call graph. When the call graph contains cycles—where lower-layer functions call higher-layer functions—the algorithm may enter a repetitive loop extracting functions within the cycle. To mitigate this risk, we have empirically set the retrieval depth to 6 layers, ignoring called functions deeper than 6 layers. This approach ensures that most samples can retrieve all relevant called functions while minimizing the risk of deadlock in cases where the call graph contains cycles. (3) Potential deadlock in bottom-up algorithm for summary generation: The bottom-up algorithm for summary generation may face challenges with cycles in the call graph, When the call graph contains cycles—where lower-layer functions call higher-layer functions—the algorithm may enter a repetitive loop, and summaries for functions within this cycle may continuously update without termination. To address this, our bottom-up algorithm uses a depth-first traversal to extract a call tree from the call graph, effectively removing cycles. Although this approach may reduce accuracy, it ensures feasibility.

External validity: (1) Impact of LLM Variations on Results. In this work, we utilize the LLM model from GPT-4o-2024-08-06, CodeLlama-13B, DeepSeek-V3 and Mixtral-8x7B to generate summaries for called functions. The performance of our framework may be influenced by the specific LLM version and vendor used. Variations in LLM versions or models from different vendors may lead to differences in the generated summaries and in the effectiveness of vulnerability detection.

7 Related Work

Inter-procedural analysis techniques. Inter-procedural analysis techniques are essential for understanding and reasoning about programs that involve multiple functions or procedures. These techniques aim to analyze the interactions and data flow between different functions to improve software quality and detect vulnerabilities. Generally, inter-procedural analysis can be categorized

into several types, including context-sensitive and context-insensitive analyses, inter-procedural data flow and control flow analyses, and summary-based analysis. Context-sensitive analysis, which considers the call context of functions, provides precise information but is computationally expensive. It has evolved significantly, as seen in seminal works like those of Sharir and Pnueli [46]. In contrast, context-insensitive analysis simplifies the problem by ignoring the call context, leading to more scalable but less precise results. Notable advancements include the work of Ruf et al. [47]. Inter-procedural data flow analysis focuses on how data values propagate through functions, with foundational contributions by Myers et al. [41]. Inter-procedural control flow analysis examines the control structures and execution paths across functions, with key developments by Nielson et al. [43]. Lastly, summary-based analysis aggregates information from function summaries to facilitate efficient inter-procedural reasoning, building upon techniques introduced by Nystrom et al. [44]. Inspired by summary-based techniques for inter-procedural analysis, we propose generating summaries for called functions to enhance function-level datasets, addressing the limitation of ignoring inter-procedural semantics while preserving the scalability of learning-based approaches.

Learning-based approaches for vulnerability detection. Learning-based approaches aim to learn effective code representations from programs to reveal vulnerability patterns for vulnerability detection. As neural network architectures have advanced, learning-based approaches have progressed significantly, evolving from basic models to more advanced architectures. Initially, Russell et al. [48] utilize Convolutional Neural Network (CNN) and Gated Recurrent Unit (GRU) to learn vector representations from lexical tokens of source code for vulnerability detection. As Graph Neural Networks (GNNs) gained prominence, Devign [57] propose a novel approach utilizing GNNs to derive code representations from various graph types, including Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), Data Flow Graphs (DFGs), and Natural Code Sequences (NCSs). The advent of pre-trained models further advanced the field, with models like CodeBERT [24], GraphCodeBERT [29] and UnixCoder [28] making significant impacts on vulnerability detection. Recent work, such as LineVul [26], has achieved state-of-the-art performance by leveraging these advancements. Therefore, in this paper, we conduct experiments on these state-of-the-art approaches to evaluate the effectiveness of our framework.

8 Conclusion

In this paper, we introduce *VulnSC*, a novel framework designed to address a critical limitation in existing learning-based vulnerability detection approaches—their inadequate handling of inter-procedural semantics due to scalability challenges when processing large samples. To overcome this, *VulnSC* retrieves the source code of called functions and leverages large language models (LLMs) to generate concise summaries using four well-designed prompting strategies. These summaries encapsulate inter-procedural semantics and are integrated into function-level samples to enhance vulnerability detection datasets. Extensive experiments on four state-of-the-art learning-based approaches across two widely used datasets demonstrate that *VulnSC* significantly improves detection effectiveness while incurring minimal computational overhead. This work not only advances vulnerability detection but also lays the foundation for future approaches that leverage inter-procedural semantics to further improve accuracy and efficiency.

9 Data Availability

All source code and data from this paper are openly licensed and accessible on our website [15].

Acknowledgments

This research is supported by the Ministry of Education, Singapore under its Academic Research Fund Tier 2 (Award ID: T2EP20222-0037).

References

[1] 2022. *checkmarx*. https://www.checkmarx.com
[2] 2022. *Codechecker*. https://codechecker.readthedocs.io
[3] 2022. *coverity*. https://scan.coverity.com
[4] 2022. *Cppcheck*. https://cppcheck.sourceforge.io
[5] 2022. *Flawfinder*. https://dwheeler.com/flawfinder
[6] 2022. *infer*. https://fbinfer.com
[7] 2022. *Juliet*. https://samate.nist.gov/SARD
[8] 2023. *Clang Static Analyzer*. https://clang-analyzer.llvm.org
[9] 2023. *CWE*. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html
[10] 2024. *report*. https://www.imf.org/en/Blogs/Articles/2024/04/09/rising-cyber-threats-pose-serious-concerns-for-financial-stability
[11] 2024. *status*. https://www.infosecurity-magazine.com/news/zeroday-surged-50-annually-google/
[12] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4998–5007.
[13] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.
[14] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*.
[15] Authors. 2024. Enhancing Vulnerability Detection via Inter-procedural Semantic Completion. https://sites.google.com/view/vulnsc-issta.
[16] Jeffrey M Barth. 1978. A practical interprocedural data flow analysis algorithm. *Commun. ACM* 21, 9 (1978), 724–736.
[17] Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge transfer from high-resource to low-resource programming languages for code llms. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 677–708.
[18] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).
[19] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 654–668.
[20] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33.
[21] Xiao Cheng, Haoyu Wang, Jiayi Hua, Miao Zhang, Guoai Xu, Li Yi, and Yulei Sui. 2019. Static detection of control-flow-related vulnerabilities using graph embedding. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 41–50.
[22] Kenneth Ward Church. 2017. Word2Vec. *Natural Language Engineering* 23, 1 (2017), 155–162.
[23] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.
[24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *Findings of the Association for Computational Linguistics: EMNLP 2020* (2020).
[25] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured neural summarization. (2018).
[26] Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 608–620.
[27] Zeyu Gao, Hao Wang, Yuchen Zhou, Wenyu Zhu, and Chao Zhang. 2023. How far have we gone in vulnerability detection using large language models. *arXiv preprint arXiv:2311.12420* (2023).
[28] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7212–7225.
[29] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.
[30] Hazim Hanif and Sergio Maffeis. 2022. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International joint conference on neural networks (IJCNN)*. IEEE, 1–8.

[31] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2023. Understanding the effectiveness of large language models in detecting security vulnerabilities. *arXiv preprint arXiv:2311.16169* (2023).

[32] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 292–303.

[33] Zhiming Li, Junzhe Jiang, Yushi Cao, Aixin Cui, Bozhi Wu, Bo Li, Yang Liu, and Danny Dongning Sun. 2025. Logic-Q: Improving Deep Reinforcement Learning-based Quantitative Trading via Program Sketch-based Tuning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 18584–18592.

[34] Zhiming Li, Yanzhou Li, Tianlin Li, Mengnan Du, Bozhi Wu, Yushi Cao, Junzhe Jiang, and Yang Liu. 2024. Unveiling Project-Specific Bias in Neural Code Models. In *LREC/COLING*.

[35] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the Effectiveness of Function-Level Vulnerability Detectors for Inter-Procedural Vulnerabilities. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[36] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021).

[37] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society.

[38] Shangqing LIU, Yu CHEN, Xiaofei XIE, Jingkai SIOW, and Yang LIU. 2021. Retrieval-augmented generation for code summarization via hybrid GNN.(2021). In *Proceedings of the Ninth International Conference on Learning Representations: ICLR*. 4–8.

[39] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software* 212 (2024), 112031.

[40] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary hunting with inter-procedural control flow. In *International Conference on Information Security and Cryptology*. Springer, 92–109.

[41] Eugene M Myers. 1981. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 219–230.

[42] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[43] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis*. springer.

[44] Erik M Nystrom, Hong-Seok Kim, and Wen-Mei W Hwu. 2004. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Static Analysis: 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004. Proceedings 11*. Springer, 165–180.

[45] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[46] M Pnueli and Micha Sharir. 1981. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications* (1981), 189–234.

[47] Erik Ruf. 1995. Context-insensitive alias analysis reconsidered. *ACM SIGPLAN Notices* 30, 6 (1995), 13–22.

[48] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.

[49] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 264–274.

[50] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Notices* 41, 6 (2006), 387–400.

[51] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Yang Liu, and Yingjiu Li. 2024. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185* (2024).

[52] Robert P Wilson and Monica S Lam. 1995. Efficient context-sensitive pointer analysis for C programs. *ACM Sigplan Notices* 30, 6 (1995), 1–12.

[53] Bozhi Wu, Shangqing Liu, Ruitao Feng, Xiaofei Xie, Jingkai Siow, and Shang-Wei Lin. 2022. Enhancing Security Patch Identification by Capturing Structures in Commits. *IEEE Transactions on Dependable and Secure Computing* (2022).

[54] Bozhi Wu, Shangqing Liu, Yang Xiao, Zhiming Li, Jun Sun, and Shang-Wei Lin. 2023. Learning Program Semantics for Vulnerability Detection via Vulnerability-Specific Inter-procedural Slicing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1371–1383.

[55] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1385–1397.
[56] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: a dataset built for AI-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 111–120.
[57] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).

Received 2024-10-31; accepted 2025-03-31