iCodeReviewer: Improving Secure Code Review with Mixture of Prompts

Yun Peng[†], Kisub Kim[‡*], Linghan Meng[†], Kui Liu[†]
[†]Huawei Technologies, China
[‡]DGIST, Daegu, Korea
{yun.p, menglinghan2, kui.liu}@huawei.com
falconlk00@gmail.com

***Abstract*—Code review is an essential process to ensure the quality of software that identifies potential software issues at an early stage of software development. Among all software issues, security issues are the most important to identify, as they can easily lead to severe software crashes and service disruptions. Recent research efforts have been devoted to automated approaches to reduce the manual efforts required in the secure code review process. Despite the progress, current automated approaches on secure code review, including static analysis, deep learning models, and prompting approaches, still face the challenges of limited precision and coverage, and a lack of comprehensive evaluation.**

To mitigate these challenges, we propose iCodeReviewer, which is an automated secure code review approach based on large language models (LLMs). iCodeReviewer leverages a novel mixture-of-prompts architecture that incorporates many prompt experts to improve the coverage of security issues. Each prompt expert is a dynamic prompt pipeline to check the existence of a specific security issue. iCodeReviewer also implements an effective routing algorithm to activate only necessary prompt experts based on the code features in the input program, reducing the false positives induced by LLM hallucination. Experiment results in our internal dataset demonstrate the effectiveness of iCodeReviewer in security issue identification and localization with an F1 of 63.98%. The review comments generated by iCodeReviewer also achieve a high acceptance rate up to 84% when it is deployed in production environments.

I. Introduction

Code review has been an important step in the software development process to ensure software quality, as it can help identify various software issues, including coding style issues, performance issues, code smells, and security vulnerabilities, at an early stage of software development [38]. Among all software issues detected in the code review process, security issues could result in severe financial losses and service disruptions. Therefore, secure code review becomes one of the top goals in the entire code review process of many companies.

Traditional code review is quite time-consuming, as reviewers need to thoroughly understand the functionality and potential impacts of the code. To improve the efficiency of software development, researchers explored many **static analysis** tools, such as CppCheck [26] and Flawfinder [7], which are built to detect flaws and dangerous coding constructs. They typically rely on well-designed static rules to identify potential coding patterns that may lead to issues. Given the large number of categories in security issues, **static analysis tools are difficult to cover real-world corner cases as well as new issues** [26].

Recently, deep learning techniques, such as CodeReviewer [15] and T5-Review, proposed fine-tuning a model based on the history of code reviews submitted by code reviewers. The fine-tuned models then take the submitted code as input and generate corresponding code review comments. However, researchers identified that developers commonly overlooked security-related issues when reviewing code in open-sourced projects [5], [40]. For example, Biase et al. [5] found that only approximately 1% of the review comments are related to security. This indicates that, in real-world software development, it is hard to collect sufficient high-quality code examples for each security issue. Therefore, **fine-tuned code review models cannot be quickly adapted to new security issues due to a lack of data for these issues** [2], [37].

In the era of large language models (LLMs), some prompt learning techniques are also studied in code review by carefully designing prompts to ask LLMs to detect issues directly. Theoretically, the prompt-based approaches do not suffer from the insufficient training data of a specific type of security issue; instead, they rely on the prompts from domain experts and the rich knowledge from the LLM. However, when deploying the prompt-based secure code review techniques in our company, we observe the following challenges:

1) **Limited Precision due to False Positives**. When identifying the categories of the security issues with/without prior knowledge (i.e., related categories knowledge in the prompt), LLMs can alarm with the security issues that are impossible to exist in the code (i.e., hallucination). This leads to lower precision of LLMs when compared with current static analysis techniques. However, an approach with high false positive rate will significantly increase the burden of developers in secure code review, as they have to frequently check and discuss the validity of generated code reviews.
2) **Limited Coverage due to False Negatives.** Different security issues may require different analysis of the program state to distinguish and confirm. It is difficult for current LLM-based approaches to cover most real-world security issues with a few fixed prompts. Besides, the performance of LLMs usually drops when it is used in domain-specific data that is rarely included in the

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00264

training datasets. The limited coverage poses great threats to the code review process as its goal is to ensure the quality of software by identifying as many software issues as possible. The missing security issues will require more efforts to identify in the following software testing process.

3) **Lack of Comprehensive Evaluation.** Currently, there is no gold standard for evaluating the quality of comments proposed by code reviewers or automated code review approaches. For example, prior studies evaluated the performance of their approaches on the code review comment generation task with BLEU and ROUGE-L [6], [31]. While useful, such metrics cannot comprehensively reflect the quality of generated review comments. This hinders the direct usage of them in practice.

In this paper, we propose iCodeReviewer, which is a LLM-based secure code review approach built upon a novel mixture-of-prompts architecture. The mixture-of-prompts architecture incorporates many prompt experts, and each of them is a dynamic prompt pipeline designed by experienced developers for one specific security issue. Prompt experts contain rich knowledge from developers to comprehensively identify potential security issues, increasing the coverage of iCodeReviewer. Given the input program, we propose a novel routing algorithm to select applicable prompt experts for code review by analyzing the code features in the program. This indicates that iCodeReviewer only activates highly-related prompt experts and avoids false positives by deactivating most irrelevant prompt experts. iCodeReviewer is also extendable to new security issues by simply adding new prompt experts. To provide a comprehensive evaluation, iCodeReviewer not only outputs the review comments, but also identify the security issue categories and pinpoint the related locations in the program.

We evaluate iCodeReviewer in an internal dataset of the company, which contains programs with different security issues identified by code reviewers in the past. Experiment results show that iCodeReviewer can achieve an F1 of 63.98% in real-world security issue identification, and an accuracy of 47.58% in issue localization. It outperforms current approaches for at least 32.11% in issue identification and 26.51% in issue localization. Furthermore, more than half of the review comments generated by iCodeReviewer are instrumental for developers. iCodeReviewer also significantly improves the acceptance rate of review comments in production lines by at least 36.84%.

We summarize our contributions as follows.

- To the best of our knowledge, we propose the mixture-of-prompts structure for iCodeReviewer, which is a brand new prompt approach for software engineering tasks.
- We design iCodeReviewer, a LLM-based secure code review approach to improve both the precision and coverage of previous approaches.
- Extensive evaluation demonstrates the effectiveness of iCodeReviewer in identifying and localizing security issues.
- iCodeReviewer has been practically deployed inside the company and used in real-world software development.

TABLE I
THE CWE ISSUES COVERED BY ICODEREVIEWER IN SECURE CODE REVIEW.

Category	SubCategory	CWE Issues
Memory Security	Memory Allocation	CWE-131, CWE-401, CWE-789
	Memory Access	CWE-129, CWE-785. CWE-806
	Memory Release	CWE-415, CWE-416, CWE-590, CWE-761, CWE-762
	Pointer Dereference	CWE-476, CWE-690, CWE-823
	Pointer Casting	CWE-587, CWE-588
	Others	CWE-134, CWE-562
Number Processing	Integer Calculation	CWE-128, CWE-191, CWE-193, CWE-369, CWE-1335
	Data Size Calculation	CWE-467, CWE-469
Sensitive Info Exposure	Process Exposure	CWE-214
	Log Exposure	CWE-532
DoS Attack	Untrusted Data	CWE-502
	Improper Control	CWE-119
Injection	Command Injection	CWE-78
	SQL Injection	CWE-89

II. Background

A. Problem Definition

Code review is a common practice in modern software development processes. It is usually formalized as a generation task $p \rightarrow cmt$, which takes the program p as input and outputs natural language comments cmt to indicate potential issues.

In this paper, we define secure code review as a multiple-goal task $p \rightarrow (cat, loc, cmt)$, which takes a program p as input and outputs the category cat of the security issues and its location loc, along with a natural language review comment cmt for further explanation. This task is generally more challenging than regular code review, as it also requires the accurate identification of security issues and locations. With identified security issues and locations, we can evaluate the effectiveness of an approach more objectively.

B. Motivating Example to Reduce False Positives

```
int func(JNIEnv *env, jclass clazz, jstring RootPath, jstring DestPath, jbyteArray data) {
    const char *RootPath = (*env)->GetString(env, RootPath, 0);
    const char *DestPath = (*env)->GetString(env, DestPath, 0);
    size_t buffSize = (*env)->GetArrayLength(env, data);
    uint8_t *buff = (uint8_t *) malloc(buffSize);
    (*env)->GetByteArrayRegion(env,data, 0, buffSize, (jbyte *) buff);
    Error ret = OK;
    ret = SaveBufferToFile(buff, buffSize, RootPath, DestPath);
    (*env)->ReleaseString(env,RootPath, RootPath);
    (*env)->ReleaseString(env,DestPath, DestPath);
    if (ret != OK) {
        LogOutput(LOG_LEVEL_ERR, "...", ret);
        return ret;
    }
    free(buff);
    return 0;
}
```

Code. 1. A motivating example simplified from a C program inside the company.

TABLE II
THE IDENTIFIED CWE ISSUES BY DIFFERENT APPROACHES. CORRECT PREDICTIONS ARE HIGHLIGHTED IN GREEN COLOR.

Approach	Identified CWEs	Location
Cppcheck	None	-
FlawFinder	None	-
CodeReviewer	None	-
T5-Review	None	-
Qwen-2.5 + Instruction Prompt	CWE-762 (Mismatched Memory Management Routines)	15
	CWE-416 (Use After Free)	9, 10
	CWE-476 (NULL Pointer Dereference)	12
	CWE-707 (Improper Enforcement of Message or Data Structure)	5, 6
Qwen-2.5 + CWE Info Prompt	CWE-78 ('OS Command Injection')	8
	CWE-476 (NULL Pointer Dereference)	2
	CWE-532 (Insertion of Sensitive Information into Log File)	8
	CWE-401 (Missing Release of Memory after Effective Lifetime)	15
DeepSeek R1 + Instruction Prompt	CWE-670 (Use of Externally Controlled Input)	2, 3, 4, 5
DeepSeek R1 + CWE Info Prompt	None	-
iCodeReviewer	CWE-690 (Unchecked Return Value to NULL Pointer Dereference)	5
	CWE-476 (NULL Pointer Dereference)	2
	CWE-401 (Missing Release of Memory after Effective Lifetime)	11

To better illustrate the motivation of our approach, we show an example in Code 1. This example is simplified from a real C program in our company, and it has a *Null Pointer Dereference* issue (i.e., CWE-476) at line 2 and a *Unchecked Return Value to NULL Pointer Dereference* issue (i.e., CWE-690) at line 5, where the external function argument env and the pointer $buff$ are not checked before dereference. Besides, it also has a *Missing Release of Memory after Effective Lifetime* issue (i.e., CWE-401) at line 11, where the program does not release the memory $buff$ when it handles an error. Developers have confirmed these issues, which may lead to severe program errors at runtime.

To understand the performance of existing approaches on this example, we apply several popular approaches to it. Table II shows the identified CWEs and their locations. Note that we do not include the natural language reviews in the table to save space.

Static Analysis. Cppcheck [26] and FlawFinder [7] are two static analysis tools designed to scan potential program flaws for C/C++ programs. We do not select advanced static analysis tools that require a compilation database since the compilation database is usually not accessible in the code review process. We observe that both Cppcheck and FlawFinder do not identify any security issues in this example, while their documentation states that they support the identification of CWE-476 and CWE-401 issues. This indicates the limited coverage of current static analysis approaches.

Code Review Models. CodeReviewer [15] and T5-Review [31] do not output CWE categories and locations. However, CodeReviewer just generates a comment "*Please remove this blank line.*" and T5-Review generates a comment "*The logic from here would be easier to understand if we used the same 'tombstone' as: hasUnusedBuffer(env,DestPath, DestPath)*". The both comments do not indicate any security issues and useful suggestions.

LLM-based Approaches. We further evaluate the performance of LLM-based approaches on this example, and design two prompt settings: 1) instructional prompting, where we only query the LLM to list all possible CWE issues, and 2) CWE informed prompting, where we give the LLM the concerned CWE categories (listed in Table I) and prompt the LLM to check if any of them exist. We employ the prompt settings on two LLMs, Qwen-2.5 and DeepSeek R1, to observe the differences between regular LLMs (e.g., Qwen-2.5) and deep thinking LLMs (e.g., DeepSeek R1).

From Table II we can see that Qwen-2.5 and DeepSeek R1 both failed to identify the correct CWE issue based on the instruction prompt. This suggests that **current LLMs are unable to detect the relevant CWE issues without guidance**. However, when given the knowledge of the concerned CWE categories, Qwen-2.5 can identify the existence of CWE-476 and CWE-401. **This indicates that compared with static analysis tools, LLMs have a better performance on secure code review.** This motivates us to use LLMs on the secure code review task with guidance.

Despite the correct identification, Qwen-2.5 misses the CWE-690 issues at lines 5 and it also provides an incorrect location for CWE-401 issue. Furthermore, it outputs two additional CWE categories, which do not exist in the example code, resulting in a high false positive rate. For example, Qwen-2.5 identifies a CWE-78 issue, but there is no interaction with the system via system calls in the example code. Therefore, **LLM-based approaches can produce both false positives and false negatives** if we let them examine the relevant issues one by one.

Our Approach. iCodeReviewer implements a novel mixture-of-prompts architecture to address current challenges. It does not prompt the LLMs to check all concerned security issues; instead, it analyzes the features in the code and identifies the pointer dereferences and memory allocation in the example and only activates the prompt expert for checking both issues. Prompt experts for other security issues are not activated and will not be identified by iCodeReviewer. By doing so, iCodeReviewer can minimize the false positives brought by LLMs. Based on the activated prompt experts, iCodeReviewer can thoroughly examine the existence of related issues and accurately capture all issues and locations, avoiding false negatives.

III. METHODOLOGY

A. Overview

As an LLM-based code review approach, iCodeReviewer adopts a Mixture-of-Prompts (MoP) approach to particularly detect security issues and provide review comments. Fig. 1 shows the overall design of iCodeReviewer that consists of four phases. Similar to the Mixture-of-Experts (MoE) architecture used in LLMs, iCodeReviewer hosts a collection of prompt experts, each of which is manually designed by senior developers in our company to identify specific types of security issues (as detailed in Section III-D). iCodeReviewer first extracts features of the code under review (Phase I). These

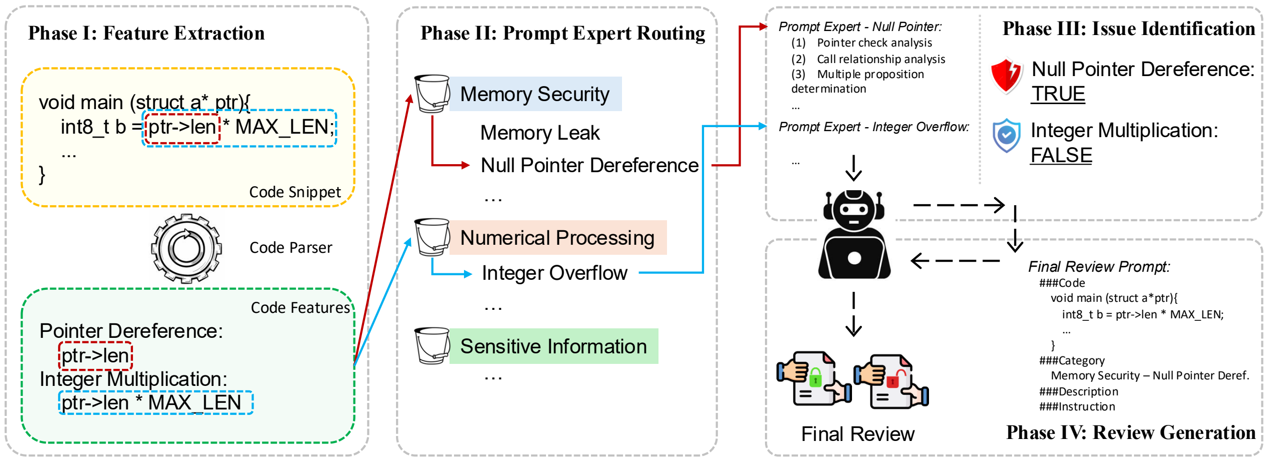
Phase I: Feature Extraction
void main (struct a* ptr){
int8_t b = ptr->len * MAX_LEN;
...
}
Code Snippet
Code Parser
Code Features
Pointer Dereference:
ptr->len
Integer Multiplication:
ptr->len * MAX_LEN
Phase II: Prompt Expert Routing
Memory Security
Memory Leak
Null Pointer Dereference
...
Numerical Processing
Integer Overflow
...
Sensitive Information
...
Prompt Expert - Null Pointer:
(1) Pointer check analysis
(2) Call relationship analysis
(3) Multiple proposition determination
...
Prompt Expert - Integer Overflow:
...
Phase III: Issue Identification
Null Pointer Dereference: TRUE
Integer Multiplication: FALSE
Final Review Prompt:
###Code
void main (struct a*ptr){
int8_t b = ptr->len * MAX_LEN;
...
}
###Category
Memory Security – Null Pointer Deref.
###Description
###Instruction
Final Review
Phase IV: Review Generation

Fig. 1. The overview of iCodeReviewer.

features indicate the existence of potential security issues in the code and then serve as a router in the MoE architecture to select the relevant prompt experts (Phase II). The corresponding prompt experts are then activated to confirm whether the corresponding security issue exists or not (Phase III). iCodeReviewer gathers the identification results from the activated prompt experts and finally generates the final review (Phase IV).

B. Phase I: Feature Extraction

Intuitively, it is unlikely that a code snippet contains all security issues, as the presence of a specific vulnerability is typically associated with certain code features. For example, a code snippet without any memory allocations will never exhibit the memory leak issue. Based on this insight, iCodeReviewer first implements a feature extraction phase to extract the related features in the input code.

To extract the features, iCodeReviewer leverages a code parser like tree-sitter [25] to analyze the input code and transform it into an abstract syntax tree (AST). By traversing the AST, iCodeReviewer gradually builds a symbol table for all symbols in the code and collects four kinds of features as shown below.

Symbol Table. In most programming languages, an identifier could be regarded as a symbol, and a symbol could be a class, a function, a variable, etc. In the AST traversal, iCodeReviewer collects all symbols and tries to infer the necessary properties of them before building the symbol table. To facilitate the secure code review, iCodeReviewer implements the following three lightweight code analysis techniques to infer the properties of symbols:

- **Type Inference.** It tries to get the type for each symbol by identifying the definition of each symbol and extracting the declared types. It also infers types for simple expressions and field accesses.
- **Taint Analysis.** It infers whether a symbol is assigned from untrusted sources, such as parameters of functions.
- **Value Analysis.** It infers the value of a symbol by analyzing initial declarations and direct assignments.

It is worth noting that the goal of lightweight analysis is to assist in feature identification, rather than directly detecting security issues. As such, these analyses are not designed to be sound or complete and may fail when essential information is unavailable. iCodeReviewer does not guess the properties of any symbol to avoid a high false positive rate, and **leaves the unknown property inference for LLMs in the following phases.** The integration of static analysis and deep learning has been proven effective in program analysis [19], [20].

Features. Symbol table records as the state of the input program and cannot be directly used to identify potential security issues. To identify potential security issues, we invite domain experts from the company to manually inspect the definitions of security issues and instances in the codebase to define code features related to the identification of security issues. When determining code features, we only include features that can be explicitly verified based on ASTs. For example, for the *Null Pointer Dereference* issue, we only define *pointer type* and *dereference* as two features, rather than the *pointer check*. This is because *pointer type* and *dereference* can be easily identified based on special syntax patterns, whereas *pointer check* is usually implemented using expressions, external APIs, or macros that require complicated analysis to identify.

iCodeReviewer collects the features of the following four kinds of code based on the symbol table:

- **API**. It identifies all APIs and records the properties of arguments and return values. It also tracks and analyzes the implementation of APIs, provided that their definitions are included in the context.

Algorithm 1 Expert Prompt Routing

```
Input: Symbol table, $ST$; Extracted code features, $F$; Expert prompts, $EP$;
Output: Selected expert prompts, $P$;
1: $F \leftarrow$ macroExpansion($F$, $ST$)
2: ($EP_n$, $EP_w$, $EP_r$) $\leftarrow$ classifyByDependency($EP$, $ST$)
3: for $ep \in EP_n + EP_w + EP_r$ do
4:                                         ▷ Context-free matching
5:     $se \leftarrow$ getSuspiciousEntities($F$, $ep$)
6:     $so \leftarrow$ getSuspiciousOperations($F$, $ep$)
7:     if $se \neq \phi$ & $so \neq \phi$ & matches($se$, $so$) then
8:         $P \leftarrow P + \{ep\}$
9:     end if
10:    $ec \leftarrow$ getExternalCalls($F$, $se$, $so$, $ep$)
11:    if $ec \neq \phi$ then                ▷ Context-sensitive matching
12:        ($st_c$, $se_c$, $so_c$) $\leftarrow$ dynamicRetrieve($ec$, $ep$)
13:        $ST \leftarrow ST + st_c$
14:        $se \leftarrow$ matchSymbol($se$, $se_c$)
15:        $so \leftarrow$ matchSymbol($so$, $so_c$)
16:        if $se \neq \phi$ & $so \neq \phi$ & matches($se$, $so$) then
17:            $P \leftarrow P + \{ep\}$
18:        end if
19:    end if
20: end for
```

- **Statement**. It identifies the loop statements and return statements and records the properties of loop variables and returned arguments.
- **Expression**. It identifies the arithmetic and casting expressions and records the properties of operands and results. It also tracks sub-expressions if multiple sub-expressions contribute to an expression.
- **Special Type**. It identifies the special types, including *pointer*, *array*, *container*, *struct*, *macro*, and *class*. It records the properties of all operations involving these special types, such as pointer dereference and array access.

For example, in Fig. 1, there are two features extracted from the input code snippet: pointer dereference "*ptr→len*" as it is an operation involving pointer types and integer multiplication "*ptr→len * MAX_LEN*" as it is an arithmetic expression. The identified code features are then used to activate prompt experts for certain security issues in the next phase.

C. Phase II: Prompt Expert Routing

Although iCodeReviewer encompasses a wide range of security issue categories through its prompt experts, it only activates those relevant to the current code context for examination. Based on the features extracted in the first phase, iCodeReviewer implements a prompt expert routing algorithm to select prompt experts. We present the algorithm in Alg. 1.

Given the symbol table ST and extracted features F, iCodeReviewer first expands all macros in the code at line 1 to avoid missing critical information in the routing phase. For example, some macros, such as *NODEPTR*, may prevent the identification of pointer types. iCodeReviewer then classifies all prompt experts into three categories according to their dependencies with the symbol table at line 2. iCodeReviewer prioritizes and paralizes the prompt experts with no relation with the symbol table (EP_n), then handles prompt experts that may read and write the symbol table (EP_w), and finally processes prompt experts that only read the symbol table (EP_r). This is to prevent the processing of one prompt expert from interfering with another. For each prompt expert, iCodeReviewer implements both context-free matching and context-sensitive matching to handle both the input program and its associated context.

Context-free Matching. For each prompt expert, we design a corresponding matching pattern consisting of two components: 1) a suspicious entity, which is a code element that may lead to security issues if incorrectly operated, and 2) a suspicious operation, which refers to the operations that may be mishandled. iCodeReviewer activates a prompt expert only when both the suspicious entity and the suspicious operation are detected and matched in the extracted features (line 4-9). For example, to match the prompt expert for the *Null Pointer Dereference* issue, iCodeReviewer first locates all pointers in the current program as suspicious entities and all pointer dereference operations as suspicious operations. If any pointer is identified as a suspicious entity involved in a suspicious operation, iCodeReviewer activates the corresponding prompt expert to conduct further analysis.

Context-Sensitive Matching. iCodeReviewer follows the same methodology to select prompt experts but implements different techniques to collect suspicious entities and operations. Instead of processing all the context, iCodeReviewer first identifies the external calls that are related to collected entities and operations (line 10), and then dynamically retrieves the useful code elements from the context based on the issue categories (line 12). For example, in the *Double Free* issue, iCodeReviewer only identifies the APIs that free the memory in the context and discards all irrelevant contents. This could significantly reduce the code processed by iCodeReviewer in the routing phase, as the context may sometimes be much larger than the input programs. For the newly collected suspicious entities and operations in the context, iCodeReviewer matches them back to the symbols in the current program by handling the argument passing in function calls (lines 14-15). The prompt expert will also be activated if a match is found between the new suspicious entities and operations.

The benefits of the prompt routing phase are twofold: 1) **Pruning.** iCodeReviewer maintains the ability to cover all security issues by keeping all related prompt experts, but it only triggers a few prompt experts in a single review process by filtering out the irrelevant security issues based on the characteristics of the input program. 2) **False Positive Reduction.** LLMs are not guaranteed to be reliable and could make mistakes due to hallucination [10]. The prompt routing phase in iCodeReviewer leverages lightweight static analysis techniques to reduce false positives by preventing the LLM from being queried about security issues that are not applicable to the given code context.

D. Phase III: Issue Identification

In this phase, iCodeReviewer sends all activated prompt experts to the LLMs for security issue identification. This phase is parallelized for each prompt expert, as the identification of each security issue is independent.

Prompt experts. Prompt experts are designed to identify the existence of certain security issues, hence they are essential for the performance of iCodeReviewer. An prompt expert is a prompt pipeline that may contain several sequential prompts $(p_1, p_2, ..., p_n, d)$ to infer the security issue, where $(p_1, ..., p_n)$ is a series of *analysis prompts* aiming to infer the state of the programs, and d is a *determination prompt* that consider the results returned by the analysis prompts and determine the existence of security issues. While different prompt experts have different determination prompts, they may share similar analysis prompts, and the prompt pipeline is dynamically constructed based on the results of previous prompts. Currently, iCodeReviewer supports the following analysis prompts:

- **Value Inference:** This prompt infers the specific value or value range of a variable.
- **Type Inference:** This prompt infers the type of a variable.
- **Value Check Inference:** This prompt infers all checks implemented for a variable.
- **Taint Variable Inference:** This prompt infers whether a variable is assigned from an untrusted source.
- **Data/Control Flow Path Inference:** This prompt infers the data flow or control flow paths from one point to another.
- **Call Relationship Inference:** This prompt infers the call relationships between the functions to facilitate inter-procedure analysis.

The benefit of prompt experts is mainly on the **false negative reduction**. We use a prompt pipeline instead of a single prompt as a prompt expert to simulate the thinking processes of human reviewers, because security issues are generally more difficult to distinguish and require a comprehensive analysis of program states. Furthermore, we break the design of prompt experts into the combination of analysis prompts and determination prompts, where the analysis prompts act like the "*shared experts*" in the MoE structure and can be reused for different security issues, and the domain experts only need to design the determination prompt when adding a new security issue category.

iCodeReviewer contains 38 prompt experts written by senior developers. It covers five major categories of security issues: *memory security*, *number processing*, *sensitive information exposure*, *DoS attack*, and *injection*. Table I presents the security issues that iCodeReviewer currently supports. For instance, the prompt expert for the *Memory Leak* issue in iCodeReviewer generally contains three prompts (p_1, p_2, d). The first analysis prompt p_1 instructs the LLM to analyze the call relationships in the current program and infer the functionality of external APIs. This prompt helps to add missing context information. The second analysis prompt p_2 then instructs the LLM to extract the data flow paths from the point of memory allocation to the end of the function. The extracted paths by p_2 are then used in the last determination prompt d, where the LLM is prompted to determine whether any of them exhibit memory leaks, i.e., at least one path forgets to release the allocated memory.

Multiple Proposition Answer for Determination Prompt. Even with the prompt pipeline $(p_1, ..., p_n, d)$, we still find that LLMs are likely to produce false positives and negatives due to hallucination, especially for security issues with complex identification logic. To alleviate this problem, we further design the multiple-proposition answer for complex security issues in the determination prompt d. The key insight is to decompose complex identification logic into a series of simpler propositions, requiring the LLM to evaluate each proposition individually rather than directly determining the presence of a security issue. As an example, we design two propositions for the *Integer Overflow* issue: *P1: The result of the operation is used as array index/pointer offset/circulation border/argument of memory allocation/length of memory copy.* and *P2: For any of the operands, there exists a value check.* The LLMs are queried to evaluate the truth values of individual propositions. Then, iCodeReviewer finally determines the existence of the security issue by computing the values of all propositions, e.g., *(P1 and not P2)* in this example.

In this phase, LLMs are only queried to give simple results of the propositions without explanations. iCodeReviewer further judges the existence of a security issue based on the results, and discards the security issues that are determined not to be present in the current program. iCodeReviewer then collects all identified security issues and generates a review to indicate them in the next phase.

E. Phase IV: Review Generation

In this phase, iCodeReviewer aggregates all security issues that LLMs confirm their existence via prompt experts and prepares them for review generation. Based on the security issues identified in the previous phase, iCodeReviewer prompts the LLM to generate a review comment that includes the categories, locations, and descriptions of each issue.

When generating the review comment, iCodeReviewer prompts the LLM to double confirm and rank the identified security issues. Specifically, iCodeReviewer first provides the LLM with the input program and all identified security issues, and prompts it to explain the reasons. This is similar to a chain-of-thought process that aims to detect potential errors made by the previous phase. The LLM will remove several identified security issues if it finds them unreasonable. iCodeReviewer then collects the remaining security issues and queries the LLM to generate a complete code review by prioritizing the issues with higher severity. The severity could be either configured by users or determined by the LLM itself.

iCodeReviewer finally outputs the generated review for the input program, which contains the category cat, location loc, and descriptions cmt of the identified security issues. Developers and code reviewers can quickly check and verify the correctness of the generated code review by iCodeReviewer.

TABLE III
THE DISTRIBUTION OF ISSUE CATEGORIES IN OUR INTERNAL DATASET. "BENIGN" INDICATES THAT THE PROGRAMS DO NOT CONTAIN SECURITY ISSUES.

Memory Security	Number Processing	Sensitive Info Exposure	DoS Attack	Injection	Benign
231	9	51	23	31	337

IV. EXPERIMENT SETUP

A. Research Questions

We focus on the following research questions:

- **RQ1:** How effective is iCodeReviewer on secure code issue identification compared with existing approaches?
- **RQ2:** How helpful are the review comments generated by iCodeReviewer in practice?
- **RQ3:** What are the impacts of different components in iCodeReviewer?

B. Datasets

We evaluate iCodeReviewer on an internal dataset collected from the code reviews in multiple production lines of the company. The original data contains the locations and review comments. We invite the developers on the corresponding production line to label the related CWE categories. The dataset includes 345 programs with real-world security issues detected by developers and 337 benign programs that developers confirm to be false positives. The dataset consists of 360 C/C++ programs, 181 Java programs, 101 Python programs, and 40 Shell programs. We show the distribution of different issue categories in Table III.

C. Metrics

As the output of iCodeReviewer contains three parts: identified issue, location, and review comments, we use different metrics to evaluate its performance. For security issue identification, we follow previous work [47] and use multi-class weighted **Precision**, **Recall**, and **F1** to handle the unbalanced distribution of issue categories in our internal dataset. In issue localization, we define **accuracy** as the ratio of correctly identified and located issues to all security issues in the internal dataset. We define a location as correct if the distance between it and the ground truth is within 1. For review comments, we adopt the methodology from Yu *et al.* [41] and classify all review comments into four categories:

- **Instrumental (I):** The generated review comment explicitly indicates the existence of the security issue identified by the reviewer and provides a fully accurate description.
- **Helpful (H):** The generated review comment raises concerns related to the security issue, but may not be entirely accurate or specific enough.
- **Misleading (M):** The generated review comment does not contain helpful information or has misleading information, such as claiming no security issues are found or reporting a false positive.
- **Uncertain (U):** The generated review comment points out other security issues other than the desired one. Due to a lack of context and knowledge, it is hard to confirm the existence of identified security issues.

Based on the above four categories, we evaluate the quality of review comments on the two metrics **I-Score** = $\frac{I}{I+H+M+U}\times$ 100%, **IH-Score** = $\frac{I+H}{I+H+M+U}\times$ 100%, and **M-Score** = $\frac{M}{I+H+M+U}\times$ 100%. We also add a metric **Acceptance Rate**, the ratio of accepted review comments by developers, to evaluate the practical value of review comments.

D. Baselines

We compare iCodeReviewer with the following two widely used static analysis tools:

- **CppCheck** [26]: It is a static analysis tool for C/C++ code. It provides unique code analysis to detect bugs and focuses on detecting undefined behavior and dangerous coding constructs.
- **Flawfinder** [7]: It scans C/C++ source code and reports potential security flaws.

We also evaluate the performance of iCodeReviewer by comparing it with the following deep learning-based approaches:

- **CodeReviewer** [15]: It is a model pre-trained with code change and code review data to support code review tasks.
- **T5-Review** [31]: It is a pre-trained Text-To-Text Transfer Transformer (T5) for automated code review.
- **Instruction Prompt**: We use a simple instruction prompt to reflect the basic performance of LLMs by querying them to output all potential CWE issues.
- **Prompt w/ CWE info** [41]: Yu *et al.* [41] evaluate multiple prompting approaches on secure code review and find that prompts with CWE information perform the best. We use it to represent the performance of a general prompt for various security issues.

In accordance with the information protection policy in the company, we do not use any closed-source LLMs, such as ChatGPT, as the base models for baselines. Instead, we use the two open-source models Qwen-2.5 [22] and DeepSeek R1 [4] for the prompting approaches to observe the performance of regular and deep thinking LLMs.

E. Implementation

iCodeReviewer has two versions for different scenarios of secure code review: 1) a web service to review the code submitted in pull requests, and 2) an IDE plugin to review the code under development. The initial version of iCodeReviewer is written in Python. iCodeReviewer uses the tree-sitter library [25] to parse the code of different programming languages into ASTs in feature extraction and Qwen-2.5 72B [22] as the base LLM for issue identification and review generation. For DeepSeek R1 [4], we choose the *DeepSeek-R1-Distill-Qwen-32B* version.

TABLE IV
PERFORMANCE OF CURRENT APPROACHES IN SECURITY ISSUE IDENTIFICATION AND LOCALIZATION, IN TERMS OF PRECISION, RECALL, MICRO F1, AND ACCURACY. 'W/ INSTR' INDICATES THE INSTRUCTION PROMPT AND 'W/ CWE' INDICATES THE PROMPT WITH CWE INFORMATION. 'DS R1' INDICATES DEEPSEEK R1.

Approach	Issue Identification			Localization
	Precision	Recall	F1	Accuracy
CppCheck	94.36	7.69	13.17	7.41
FlawFinder	90.16	1.14	1.20	1.14
CodeReviewer	**100**	0.28	0.57	-
T5-Review	94.25	2.85	5.37	-
Qwen 2.5 w/ Instr	73.58	41.31	45.59	30.20
DS R1 w/ Instr	67.19	47.29	48.43	35.33
Qwen 2.5 w/ CWE	62.48	30.77	39.04	29.06
DS R1 w/ CWE	54.84	43.87	45.25	37.61
iCodeReviewer	75.48	**62.68**	**63.98**	**47.58**

V. EVALUATION

A. *Effectiveness of iCodeReviewer in Security Issue Identification and Localization*

To evaluate the effectiveness of iCodeReviewer in identifying security issues, we compare it with eight baselines on the internal dataset. We do not limit the number of security issues each approach can output to facilitate the evaluation of their precision. As CodeReviewer and T5-Review only output a review comment without issue categories, we use Qwen-2.5 to label the issue categories based on the comments. We randomly sample 10% of labeled results and find that the labels are 100% correct.

Issue Identification. Based on the predictions of each approach, we then calculate the multi-class weighted precision, recall, and F1 for each approach and present the results in Table IV. From the table, we observe that iCodeReviewer achieves the highest F1 of 63.98%, outperforming the best baseline *DeepSeek R1 with instruction prompts* by 32.11%. This demonstrates the superior performance of iCodeReviewer in real-world security issue identification. Although CodeReviewer achieves the highest precision of 100%, its recall is as low as 0.28%, which indicates that it can hardly capture security issues in practice. iCodeReviewer still achieves the highest precision among all LLM-based approaches, benefiting from the mixture-of-prompts architecture. Furthermore, iCodeReviewer can capture the most real-world security issues with a recall of 62.68%, outperforming the best approach *DeepSeek R1 with instruction prompts* by a large margin.

Issue Localization. Apart from identification, the ability to accurately pinpoint the security issues is also essential to help developers quickly understand the problems. We further calculate the accuracy of locations given by each approach and list it in the fourth column of Table IV. Note that we exclude CodeReviewer and T5-Review as they do not output location information. The results in the table suggest that iCodeReviewer can accurately identify and locate the most real-world security issues with an accuracy of 47.58%. This outperforms the baselines by at least 26.51%. Moreover, we find that the gap between LLM-based approaches in recall for issue identification and accuracy for localization is generally larger than that of static analysis tools. This indicates that LLM-based approaches cannot pinpoint the locations in some cases, even if they can identify the correct issues.

For both issue identification and location, iCodeReviewer significantly outperforms current prompting approaches. This demonstrates the effectiveness of the mixture-of-prompts architecture in iCodeReviewer for secure code review. By routing the input program to only a few prompt experts, iCodeReviewer can reduce the false positives and identify more security issues with accurate location.

Answer to RQ1: iCodeReviewer is effective at identifying and locating real-world security issues with an F1 of 63.98% and an accuracy of 47.58%, outperforming the best baseline by 32.11% and 26.51%, respectively.

TABLE V
HELPFULNESS OF REVIEWS GENERATED BY CURRENT APPROACHES. "↑" INDICATES A HIGHER VALUE IS BETTER AND "↓" INDICATES A LOWER VALUE IS BETTER.

Approach	I-Score ↑	IH-Score ↑	M-Score ↓
CppCheck	1.80	5.58	94.42
FlawFinder	1.33	18.00	82.00
CodeReviewer	0.31	0.31	0.70
T5-Review	2.96	5.62	93.50
Qwen 2.5 w/ Instr	21.69	23.83	76.04
DS R1 w/ Instr	12.75	18.44	72.73
Qwen 2.5 w/ CWE	32.53	40.06	59.94
DS R1 w/ CWE	21.72	33.19	65.24
iCodeReviewer	**53.94**	**59.70**	**40.30**

B. *Helpfulness of Reviews Generated by iCodeReviewer*

Based on the issue categories and locations, we could objectively evaluate the performance of existing secure code review approaches. However, the quality of review comments is still essential as they can help developers quickly understand the problems. To evaluate the quality of review comments, we invite software engineers to manually inspect the comments generated by all approaches for our internal dataset. Furthermore, we also deploy iCodeReviewer in two production lines to observe the changes in the acceptance rate of review comments.

Manual Inspection. We invite three software engineers with at least five years of experience to inspect the comments generated by all approaches manually. We only provide them with the comments and locations so that they can focus solely on the quality of review comments. We ask the engineers to classify all review comments into four categories: *Instrumental*, *Helpful*, *Misleading*, and *Uncertain*, as stated in Sec. IV. Based on the inspection results, we calculate the related metrics and present them in Table V. We identify that iCodeReviewer achieves the highest I-Score

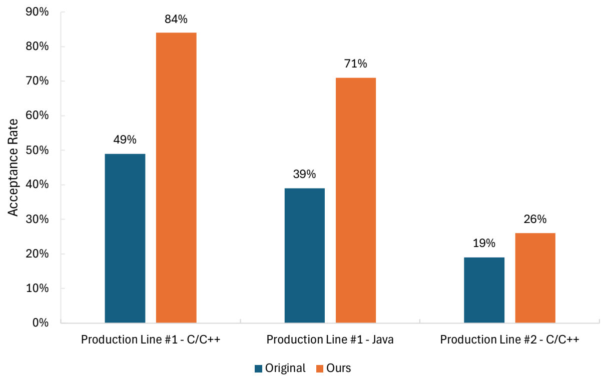

Fig. 2. The acceptance rate before and after deploying iCodeReviewer in two production lines.

of 52.94%, indicating that more than half of the review comments generated by iCodeReviewer are indicative and easy to understand. On the contrary, the best baseline only achieves an I-score of 32.53%, which is significantly lower than that of iCodeReviewer. In addition, we find that LLMs with an instruction prompt obtain much lower IH-Score than those with a CWE information prompt. This is opposite to the F1 in Table IV and suggests that the instruction prompt achieves higher F1 at the cost of producing more misleading information, increasing the burden on developers.

Production Line Deployment. To observe the performance of iCodeReviewer in real-world software development, we deploy it in two production lines for one week and collect the acceptance rates before and after the deployment. We present the results in Fig. 2. For production line #1, we can find that the acceptance rate of iCodeReviewer is 84% for C/C++ and 71% for Java, which outperforms the original tool by 71.43% and 82.05%, respectively. This suggests that developers accept more than 70% of reviews generated by iCodeReviewer in the production lines. As for production line #2, iCodeReviewer achieves an improvement of about 36.84%. However, the acceptance rate in this production line is still low. A possible reason is that the software developed by it is generally more complicated and specific to a single domain, which limits the performance of LLMs.

While better ability in issue identification does not necessarily indicate higher quality of generated review comments, iCodeReviewer still keeps the good quality of review comments. With the analysis information provided by prompt experts, iCodeReviewer can generate informative review comments for specific security issues.

Answer to RQ2: Review comments generated by iCodeReviewer are helpful with an I-score of 53.94% and significantly higher acceptance rates in production lines.

C. Ablation Study

To verify the usefulness of different components in prompt experts, we conduct an ablation study by removing them and observing the performance of iCodeReviewer. Table VI presents the results of issue identification and localization for an objective evaluation. For the ablation, we do not remove the routing mechanism in iCodeReviewer since LLMs with CWE information prompts represent such performance in Table IV and we have already demonstrated the effectiveness of iCodeReviewer in RQ1.

TABLE VI
ABLATION RESULTS OF ICODEREVIEWER IN SECURITY ISSUE IDENTIFICATION AND LOCALIZATION, IN TERMS OF PRECISION, RECALL, MICRO F1, AND ACCURACY.

Approach	Issue Identification			Localization
	Precision	Recall	F1	Accuracy
w/o Analysis Prompts	75.60	60.11	62.23	47.01
w/o Multiple Prop.	78.22	58.69	60.24	43.87
w/o Prompt Experts	75.26	43.30	47.17	37.89
iCodeReviewer	**75.48**	**62.68**	**63.98**	**47.58**

In Table VI, we observe that the highest decrease of F1 from 63.98% to 47.17% occurs when we replace all prompt experts with single questions of whether a security issue exists or not. Without the prompt experts, iCodeReviewer has a similar performance with the instruction prompts. The significant decrease demonstrates that prompt experts are the key component for the mixture-of-prompts architecture. When removing the analysis prompts and multiple proposition answers, the F1 of iCodeReviewer drops from 63.98% to 62.23% and 60.24%, respectively. This verifies the usefulness of analysis prompts and multiple proposition answers in identifying more security issues. The performance drop in both ablations is not as large as the removal of prompt experts because they are designed only for some complex security issue categories. As for the issue localization, the removal of analysis prompts does not cause a significant decrease since analysis prompts mainly help determine whether a security issue exists. Multiple proposition answers contribute more to issue localization since some propositions indicate location guidance.

Answer to RQ3: Prompt experts in iCodeReviewer play the most important roles, and removal of them leads to a significant F1 decrease from 63.98% to 47.17% and an accuracy decrease from 47.58% to 37.89%. Other components in iCodeReviewer also contribute to its final performance.

VI. THREATS TO VALIDITY

A. Internal Validity

Our study may face the following threats to internal validity.

Subjective Evaluation of Reviews. We evaluate the quality of review comments generated by iCodeReviewer based on human judgments. This may introduce subjective factors that threaten the validity of evaluation results. We mitigate this threat by using two evaluation methods. We evaluate iCodeReviewer on the internal dataset by asking several developers to classify the comments into four categories mentioned in Sec. IV-C. Besides, we deploy iCodeReviewer in two production lines and measure the acceptance rates of code reviews generated by iCodeReviewer. We believe that the incorporation of different evaluation metrics and developers could significantly reduce the subjective factors and lead to solid evaluation results.

B. External Validity

Our study may face the following threats to the external validity.

Generalization to Other LLMs. While iCodeReviewer is a prompt framework and could be implemented upon any LLMs, we only implement it on Qwen-2.5, due to the information protection policy in the company and the limited computation budgets. This may threaten the performance of iCodeReviewer if it is adopted in other LLMs. However, we believe the adaptation will not significantly hurt the performance of iCodeReviewer as iCodeReviewer does not require fine-tuning or use any specific features of the Qwen-2.5 model.

Generalization to Other Companies. iCodeReviewer is initially built for checking the security issues of our company. We admit that different companies may target different security issues. The performance of iCodeReviewer may be influenced if it is adopted to check new security issues. However, we believe that iCodeReviewer could be easily adapted for new security issues by designing new prompt experts and adding them into the prompt router. Besides, the existing security issues supported by iCodeReviewer are general issues with CWE categories. We do not include any security issues that are specific to our company in the evaluation.

VII. Related Work

A. LLM-based Code Review

New trends in code review have increasingly focused on leveraging LLMs, due to their flexibility and context-aware reasoning capabilities over both natural and programming languages. Tufano et al. [32] started with a fine-tuned T5 model to generate code review comments, showing that pre-trained Transformers outperform prior neural and statistical baselines. Empirical studies [3], [35] evaluate LLMs such as Codex and ChatGPT in real-world review scenarios. They found that while LLMs can replicate many aspects of human review behavior, they may overgeneralize or hallucinate suggestions without project-specific grounding. Then, researchers also applied agent-based architectures that modularize code review workflows. For example, CodeAgent [29] orchestrates specialized sub-agents (e.g., QA-Checker) to emulate collaborative review dynamics, outperforming baseline generative models such as ChatGPT and Codex. Unfortunately, we could not include this approach as a baseline due to the political matter. Further studies [1], [9], [27] demonstrated that providing additional code structure or execution context improves review accuracy and relevance. Although additional information is already known to be helpful for code review automation, we are the first to propose the mixture-of-prompts approach that routes the prompt experts based on code features.

B. LLM-adapted Vulnerability Detection

Recent studies proposed LLM adaptation techniques for vulnerability detection. There are three major ways to adapt LLMs to vulnerability detection: (1) fine-tuning, (2) prompt engineering, and (3) retrieval augmented generation (RAG). During fine-tuning LLMs, researchers [16], [18], [30], [34], [36], [43] have leveraged various program analysis techniques to extract structural features/relations within code, which can help enhance code understanding. To address the Transformer [33]'s architectural limitations, i.e., sequential token relation, researchers [11], [28], [37] tried to apply deep learning modules such as GNN [23]. There is another study [46] that considers restrictions on the length of input code snippets and applies Bi-LSTM [24] to mitigate the limitation. While fine-tuning techniques have shown measurable improvements in detection performance, the gains are often modest and come with substantial computational cost and limited generalizability across diverse codebases.

To further improve the performance, researchers focused on boosting the LLM's capabilities by engineering the prompts. The possible prompt design considerations are on the Task Descriptions [8], [21], [39], [44], [45], Role Description [8], [13], [39], [45], Auxiliary Information [13], [42], [45], and Chain-of-thought [14], [17], [44]. Moreover, researchers [17], [44] have also studied the capabilities of prompt design using a few examples of input and ground-truth label pairs. Unlike these studies, our approach borrows the Mixture-of-Experts (MoE) concept [12], which routes the input through specialized sub-models based on the characteristics of the code, and proposes the mixture-of-prompts architecture that does not require model training. This allows LLMs to adaptively focus on specific potential security issues initially inferred from the input program, which enhances performance to identify security issues accurately.

VIII. Conclusion

In this paper, we first define secure code review, a more challenging but practical and helpful code review task desired by the industry. To complete this task, we propose iCodeReviewer, a LLM-based approach that leverages a novel mixture-of-prompts architecture to improve both the precision and coverage of previous code review approaches. We compare iCodeReviewer with eight baselines in an internal dataset of the company for an objective evaluation of security issue identification and localization. We also evaluate the quality of review comments generated by iCodeReviewer via manual inspection of senior developers and deployment in two production lines. Results demonstrate the effectiveness of iCodeReviewer in security issue identification and localization, and the helpfulness of review comments generated by iCodeReviewer. iCodeReviewer is now adopted by many production lines in the company.

References

[1] Fannar Steinn Aalsteinsson, Björn Borgar Magnússon, Mislav Milicevic, Adam Nirving Davidsson, and Chih-Hong Cheng. Rethinking code

review workflows with llm assistance: An empirical study. *arXiv preprint arXiv:2505.16339*, 2025.

[2] Larissa Braz and Alberto Bacchelli. Software security during modern code review: the developer's perspective. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '22, page 810–821. ACM, November 2022.

[3] Umut Cihan, Vahid Haratian, Arda İçöz, Mert Kaan Gül, Ömercan Devran, Emircan Furkan Bayendur, Baykal Mehmet Uçar, and Eray Tüzün. Automated code review in practice. *arXiv preprint arXiv:2412.18531*, 2024.

[4] DeepSeek-AI et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[5] Marco di Biase, Magiel Bruntink, and Alberto Bacchelli. A security perspective on code review: The case of chromium. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 21–30, 2016.

[6] Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. Exploring the Capabilities of LLMs for Code Change Related Tasks. *ACM Transactions on Software Engineering and Methodology*, page 3709358, December 2024.

[7] Flawfinder. Flawfinder, 2024. https://dwheeler.com/flawfinder/.

[8] Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. Chatgpt for vulnerability detection, classification, and repair: How far are we? In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pages 632–636. IEEE, 2023.

[9] Md Asif Haider, Ayesha Binte Mostofa, Sk Sabit Bin Mosaddek, Anindya Iqbal, and Toufique Ahmed. Prompting and fine-tuning large language models for automated code review comment generation. *arXiv preprint arXiv:2411.10129*, 2024.

[10] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55, January 2025.

[11] Zhonghao Jiang, Weifeng Sun, Xiaoyan Gu, Jiaxin Wu, Tao Wen, Haibo Hu, and Meng Yan. Dfept: data flow embedding for enhancing pre-trained model based vulnerability detection. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, pages 95–104, 2024.

[12] Michael I Jordan and Robert A Jacobs. Hierarchical mixtures of experts and the em algorithm. *Neural computation*, 6(2):181–214, 1994.

[13] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. Understanding the effectiveness of large language models in detecting security vulnerabilities. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 103–114. IEEE, 2025.

[14] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[15] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. Automating code review activities by large-scale pre-training. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 1035–1047. ACM, 2022.

[16] Zhongxin Liu, Zhijie Tang, Junwei Zhang, Xin Xia, and Xiaohu Yang. Pre-training by predicting program dependencies for vulnerability analysis tasks. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[17] Chao Ni, Liyu Shen, Xiaodan Xu, Xin Yin, and Shaohua Wang. Learning-based models for vulnerability detection: An extensive study. *arXiv preprint arXiv:2408.07526*, 2024.

[18] Tao Peng, Shixu Chen, Fei Zhu, Junwei Tang, Junping Liu, and Xinrong Hu. Ptlvd: Program slicing and transformer-based line-level vulnerability detection system. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 162–173. IEEE, 2023.

[19] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael R. Lyu. Static inference meets deep learning: A hybrid type inference approach for python. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2019–2030. ACM, 2022.

[20] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. Generative type inference for python. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 988–999. IEEE, 2023.

[21] Moumita Das Purba, Arpita Ghosh, Benjamin J Radford, and Bill Chu. Software vulnerability detection using large language models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 112–119. IEEE, 2023.

[22] Qwen. Qwen2.5 technical report, 2025.

[23] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[24] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

[25] Tree Sitter. Tree sitter, 2025. https://tree-sitter.github.io/tree-sitter/.

[26] CppCheck Solutions. Cppcheck, 2024. https://cppcheck.sourceforge.io/.

[27] Tao Sun, Jian Xu, Yuanpeng Li, Zhao Yan, Ge Zhang, Lintao Xie, Lu Geng, Zheng Wang, Yueyan Chen, Qin Lin, et al. Bitsai-cr: Automated code review via llm in practice. *arXiv preprint arXiv:2501.15134*, 2025.

[28] Wei Tang, Mingwei Tang, Minchao Ban, Ziguo Zhao, and Mingjun Feng. Csgvd: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. *Journal of Systems and Software*, 199:111623, 2023.

[29] Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé F Bissyandé. Codeagent: Autonomous communicative agents for code review. *arXiv preprint arXiv:2402.02172*, 2024.

[30] Hoai-Chau Tran, Anh-Duy Tran, and Kim-Hung Le. Detectvul: A statement-level code vulnerability detection for python. *Future Generation Computer Systems*, 163:107504, 2025.

[31] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2291–2302. ACM, 2022.

[32] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *Proceedings of the 44th international conference on software engineering*, pages 2291–2302, 2022.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[34] Huanting Wang, Zhanyong Tang, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. Combining structured static code information and dynamic symbolic traces for software vulnerability prediction. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[35] Miku Watanabe, Yutaro Kashiwa, Bin Lin, Toshiki Hirao, Ken'Ichi Yamaguchi, and Hajimu Iida. On the use of chatgpt for code review: Do developers like reviews by chatgpt? In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 375–380, 2024.

[36] Cheng Weng, Yihao Qin, Bo Lin, Pei Liu, and Liqian Chen. Matsvd: Boosting statement-level vulnerability detection via dependency-based attention. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, pages 115–124, 2024.

[37] Aidan ZH Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. Security vulnerability detection with multitask self-instructed fine-tuning of large language models. *arXiv preprint arXiv:2406.05892*, 2024.

[38] Zezhou Yang, Cuiyun Gao, Zhaoqiang Guo, Zhenhao Li, Kui Liu, Xin Xia, and Yuming Zhou. A survey on modern code review: Progresses, challenges and opportunities. *CoRR*, abs/2405.18216, 2024.

[39] Xin Yin. Pros and cons! evaluating chatgpt on software vulnerability. *arXiv preprint arXiv:2404.03994*, 2024.

[40] Jiaxin Yu, Liming Fu, Peng Liang, Amjed Tahir, and Mojtaba Shahin. Security defect detection via code review: A study of the openstack and qt communities. In *ACM/IEEE International Symposium on Empirical*

Software Engineering and Measurement, ESEM 2023, New Orleans, LA, USA, October 26-27, 2023, pages 1–12. IEEE, 2023.

[41] Jiaxin Yu, Peng Liang, Yujia Fu, Amjed Tahir, Mojtaba Shahin, Chong Wang, and Yangxiao Cai. An insight into security code review with llms: Capabilities, obstacles and influential factors, 2024.

[42] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. Prompt-enhanced software vulnerability detection using chatgpt. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 276–277, 2024.

[43] Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. Vulnerability detection by learning from syntax-based execution paths of code. *IEEE Transactions on Software Engineering*, 49(8):4196–4212, 2023.

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

[45] Xin Zhou, Ting Zhang, and David Lo. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 47–51, 2024.

[46] Noah Ziems and Shaoen Wu. Security vulnerability detection using deep learning natural language processing. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6. IEEE, 2021.

[47] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. Vuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, page 1–1, 2019.