T19 - PA -基于提示工程的智能化源码警告识别实证研究

1 C/C++ 数据集准备

1.1 bug report数据源

- LLM4SA: https://zenodo.org/records/8346515
- 采用 cppcheck 结果
- 用以下脚本将 xml 文件转换拆分成一系列单独的 json 文件:

```
$ python3 ${LLM4SA}/scripts/process/cppcheck_XML2Json_for_single_err.py -f
cppcheck_err.xml -p .
```

- 提取 json 文件，如 Bug_0001_NPD.json：

```
{
    "bug_type": "Null Pointer Dereference",
    "line": 219,
    "column": 32,
    "procedure": "",
    "file": "alias.c",
    "qualifier": {
        "Cppcheck": "Either the condition 'aliases' is redundant or there is
possible null pointer dereference: aliases."
    },
    "Trace": [
        {"filename": "alias.c", "line_number": 219, "column_number": 32,
"description": ""},
        {"filename": "alias.c", "line_number": 221, "column_number": 20,
"description": ""}
    ]
}
```

- 其中 `bug_type` 代表 bug 类型，file 定位了 bug 所在的文件，Trace 记录了 bug 所在的位置及调用链

1.2 涵盖项目

- bash: https://ftp.gnu.org/gnu/bash/bash-4.3.tar.gz
- combine: https://ftp.gnu.org/gnu/combine/combine-0.4.0.tar.xz
- diffutils: https://ftp.gnu.org/gnu/diffutils/diffutils-3.3.tar.xz
- binutils: https://ftp.gnu.org/gnu/binutils/binutils-2.25.1.tar.bz2
- m4: https://ftp.gnu.org/gnu/m4/m4-1.4.17.tar.gz
- RIOT: https://github.com/RIOT-OS/RIOT
- gawk: https://ftp.gnu.org/gnu/gawk/gawk-4.1.2.tar.xz
- trueprint: https://ftp.gnu.org/gnu/trueprint/trueprint-5.4.tar.gz
- Zephyr: https://github.com/zephyrproject-rtos/zephyr

1.3 codesnippet生成

- 为每个项目的每个 `bug report` 生成 `codesnippet`，示例脚本如下：

```
./extra.sh
```

```bash
#!/bin/bash

TEST_FOLDER="test/RIOT"
SCRIPT_PATH="src/code-extractor/extract_code_snippet.py"
OUTPUT_BASE_FOLDER="codesnippet/RIOT"  # 输出文件夹

# 确保输出文件夹存在
mkdir -p "$OUTPUT_BASE_FOLDER"

# 遍历文件夹中每个以 Bug_ 开头并以 .json 结尾的文件
for json_file in "$TEST_FOLDER"/Bug_*.json; do
    # 提取文件名（不含路径）
    filename=$(basename -- "$json_file")

    # 提取编号 (xxxx)，假设文件名格式为 Bug_xxxx_abc.json
    bug_number=$(echo "$filename" | sed -n 's/Bug_\([0-9]*\)_.*\.json/\1/p')

    # 如果未匹配到 bug_number，跳过
    if [ -z "$bug_number" ]; then
        echo "Skipping $filename: Unable to extract bug number."
        continue
    fi

    # 构造输出前缀，路径在 OUTPUT_BASE_FOLDER 下
    output_prefix="$OUTPUT_BASE_FOLDER/RIOT_$bug_number"

    # 运行命令
    echo "Processing $json_file -> Output: $output_prefix"
    python3 "$SCRIPT_PATH" -f "$TEST_FOLDER" -r "$json_file" -o
"$output_prefix" -m 1
done
```

- 生成 `txt` 文件（如 Trace_None_snippets_1.txt）记录从项目源码中找到的 bug 相关代码，我们保留 bug 所在的函数作为`warning_function`，等价于 Java 数据集中的 `warning_method`：

```
    // ./alias.c, line: 206-233
206 map_over_aliases (function)
207      sh_alias_map_func_t *function;
208 {
209   register int i;
210   register BUCKET_CONTENTS *tlist;
211   alias_t *alias, **list;
212   int list_index;
213
214   i = HASH_ENTRIES (aliases);
215   if (i == 0)
216     return ((alias_t **)NULL);
217
218   list = (alias_t **)xmalloc ((i + 1) * sizeof (alias_t *));
219   for (i = list_index = 0; i < aliases->nbuckets; i++)
220     {
221       for (tlist = hash_items (i, aliases); tlist; tlist = tlist->next)
222     {
223       alias = (alias_t *)tlist->data;
224
225       if (!function || (*function) (alias))
226         {
227           list[list_index++] = alias;
228           list[list_index] = (alias_t *)NULL;
229         }
230     }
231     }
232   return (list);
233 }
```

- 进一步根据 `Trace_line` 可以定位 bug 所在行，提取为 `warning_line`，即：

```
219   for (i = list_index = 0; i < aliases->nbuckets; i++)
```

1.4 提取数据集文件

- 遍历所有项目的所有 bug，提取对应数据项，生成数据集文件 cpp.xlsx
- `final_label` 列根据 cppcheck 结果手动更正
- 数据集示例如下（ `warning_function`完整内容省略）

bug_name	project	bug_identifier	bug_type	line	file	qualifier	final_label	trace_filename	trace_line_number	trace_column_number	trace_description	warning_line	warning_function
bash_0001	bash	Null Pointer Dereference	NPD	219	alias.c	Either the condition 'aliases' is redundant or there is possible null pointer dereference: aliases.	FP	alias.c	219	32		for (i = list_index = 0; i < aliases->nbuckets; i++)	// ./hashlib.h, line: 70-73 ...

2 Java数据集准备

2.1 bug report数据源

- 数据源：该数据集来源于助教提供的**警告数据集及说明.zip**中的**java数据集.xlsx**文件。该文件包含项目中各种警告信息及其相应的标签。
- 原**java数据集.xlsx**中包含10140条数据项，其中包含 TP（True Positive）和 FP（False Positive）类型数据。按数据源中的`category`和`vtype`字段，等比例提取了1090项数据样本到**selected_java_samples.xlsx**，提高了所取数据的覆盖面与实用性。数据提取比例如下：
  - TP项：474条
  - FP项：616条
- 数据集示例如下：

index	category	vtype	final_label	project	warning_line	warning_method
0	MALICIOUS_CODE	MS_PKGPROTECT	TP	bcel	@Deprecated public static final String[] SHORT_TYPE_NAMES = { ILLEGAL_TYPE, ILLEGAL_TYPE, ILLEGAL_TYPE, ILLEGAL_TYPE, "Z", "C", "F", "D", "B", "S", "I", "J", "V", ILLEGAL_TYPE, ILLEGAL_TYPE, ILLEGAL_TYPE };	@Deprecated public static final String[] SHORT_TYPE_NAMES = { ILLEGAL_TYPE, ILLEGAL_TYPE, ILLEGAL_TYPE, ILLEGAL_TYPE, "Z", "C", "F", "D", "B", "S", "I", "J", "V", ILLEGAL_TYPE, ILLEGAL_TYPE, ILLEGAL_TYPE };

2.2 覆盖项目

`bcel`：https://github.com/apache/commons-bcel

`codec`：https://github.com/apache/commons-codec

`collections`：https://github.com/apache/commons-collections

`configuration`：https://github.com/apache/commons-configuration

`dbcp`：https://github.com/apache/commons-dbcp

`digester`：https://github.com/apache/commons-digester

`fileupload`：https://github.com/apache/commons-fileupload

`mavendp`：https://github.com/apache/maven-deploy-plugin

`net`：https://github.com/apache/commons-net

`pool`：https://github.com/apache/commons-pool

3 prompt 编写

3.1 prompt选择

我们首先查看java语言和C语言中的bug类型。筛查总结数据集得到C项目中存在Null-pointer dereference,use-before-initialization,buffer overflow/overrun or Out-of-bound asscess,memory leak,user after free以及Divide By Zero这六类bug,
对于java项目我们根据java本身的bug分类，划分为Java bad practices,CORRECTNESS,DODGY_CODE,EXPERIMENTAL issues,I18N,malicious code patterns ,multithreaded correctness issues,performance和secuity这九类bug。
我们使用COT和persona结合以及few-shots这两种方式对prompt进行编写。具体格式为：

```
prompt_bug='''
# Task Description
Now you should act as an expert in Java/C code review, You possess advanced skills in analyzing program code using well-known static analysis tools.\
Additionally, You have extensive experience in finding a type of bug called **{bug类型}**.\
It is worth noting that these static analysis tools frequently produce a significant number of warnings, which may consist of both false positives and redundancies.\
Consequently, it becomes essential for you to manually inspect and verify each warning.\

**{对应bug的分析思路}**

Lastly, You will be asked to determine whether the bug is a real bug or a false alarm.\
In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.\n
'''
```

首先我们采用了persona模式，对LLM的角色进行严格规定，讲LLM设定为一个代码审查方面的专家，并且强调其在某一领域的专业性；其次我们使用COT模式，指导LLM的bug判断方式，给出每种bug对应的分析思路；最后我们增加对输出的限制，限制其回答在real bug,false alarm以及unknown这三类，方便后续操作。对于对应的C项目中的每种类型的bug我们提取两个例子，使用多轮对话的方式加入prompt中，具体格式为：

```
prompt_question='''
# Bug Report
**{对应Bug Report}**
# Code Snippet
**{Code Snippet}**
'''
```

```
prompt_answer='''
{对应的分析}

@@@ false alarm @@@
'''
```

3.2 具体prompt展示

bug_category	prompt
Null-pointer dereference	# Task Description Now you should act as an expert in C/C++ code review, You possess advanced skills in analyzing program code using well-known static analysis tools. Additionally, You have extensive experience in finding a type of bug called Null-pointer dereference. It is worth noting that these static analysis tools frequently produce a significant number of warnings, which may consist of both false positives and redundancies. Consequently, it becomes essential for you to manually inspect and verify each warning. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. Initially, In the beginning, You simulate "dynamic symbolic execution" based on the error trace, using concrete values if available. You should explain whether the bug can occur in the calling context of the code snippet. You should ignore the code after the location where the bug occurred. Including assignment and checking null. When encountering if conditions, You will analyze each situation. Finally, You will confirm if there is a possibility of a null dereference occurring in these situations. If it exists, You will provide a conclusion about real bugs. Specifically, there may be certain if conditions are the same, they will execute both true and false branches simultaneously. So You should check whether the error trace in bug reports will occur. Thinking step by step. You only report a genuine bug when You are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. Suppose You have difficulty analyzing fields without more information, such as a function definition, caller, and callee, try your best to guess the behavior of the function. Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.\

bug_category	prompt
use-before-initialization	# Task Description Now you should act as an expert in C/C++ code review, You possess advanced skills in analyzing program code using well-known static analysis tools. Additionally, You have extensive experience in finding a type of bug called use-before-initialization. It is worth noting that these static analysis tools frequently produce a significant number of warnings, which may consist of both false positives and redundancies. Consequently, it becomes essential for you to manually inspect and verify each warning. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. Initially, You should explain whether the bug can occur in the calling context of the code snippet. Note that Some function calls are within a boolean condition judgment. In these cases, You should consider these conditions in your analysis. For example, the function "sscanf(str, '%u.%u.%u.%u%n';, &a, &b, &c, &d, &n) >= 4" will initialize a, b, c, d. Please closely examine the scenarios in which each conditional branch evaluates as true or false and their feasibility given specific inputs. Considering the condition ">=4", it means that when this condition is true, the first four parameters, a, b, c, and d, must be initialized. In other cases, functions will return with a return code. The caller then checks the return code to determine if the function was executed successfully. For example, "if(!func(...)) return" In this case, You should consider these return value checks and only go to successful conditions (means won't return directly) Thinking step by step. You only report a genuine bug when You are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. Additionally, your analysis should be field-sensitive. This means if some functions initialize the fields of their parameters (i.e, for func(struct some_struct* ptr), it may initialize ptr->config). Suppose You have difficulty analyzing fields without more information, such as a function definition, caller, and callee, try your best to guess the behavior of the function. Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.\

bug_category	prompt
buffer overflow/overrun or Out-of-bound asscess	# Task Description Now you should act as an expert in C/C++ code review, You possess advanced skills in analyzing program code using well-known static analysis tools. Additionally, You have extensive experience in finding a type of bug called buffer overflow/overrun or Out-of-bound asscess. It is worth noting that these static analysis tools frequently produce a significant number of warnings, which may consist of both false positives and redundancies. Consequently, it becomes essential for you to manually inspect and verify each warning. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. Initially, You should explain whether the bug can occur in the calling context of the code snippet. When encountering access to arrays or copying operations between arrays (e.g. memcpy), You will obtain the array size based on the context and determine whether the index exceeds the length of the array. Additionally, there may be cases of function calls. At this point, You need to correspond formal and actual parameters and synchronize the changes between them. Note that Some function calls are within a boolean condition judgment. In these cases, You should consider these conditions in your analysis. Please closely examine the scenarios in which each conditional branch evaluates as true or false and their feasibility given specific inputs. In other cases, functions will return with a return code. The caller then checks the return code to determine if the function was executed successfully. For example, "if(!func(...)) return" In this case, You should consider these return value checks and only go to successful conditions (means won't return directly) Thinking step by step. You only report a genuine bug when You are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. Suppose You have difficulty analyzing fields without more information, such as a function definition, caller, and callee, try your best to guess the behavior of the function. Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.\

bug_category	prompt
memory leak	# Task Description Now you should act as an expert in C/C++ code review, You possess advanced skills in analyzing program code using static analysis tools, such as Infer, Cppcheck, and Clang static analysis. You possess substantial expertise in reviewing and interpreting analysis reports, enabling accurate identification of false alarm reports. It is worth noting that these static analysis tools frequently produce a significant number of warnings, which may consist of lots of false alarms. Currently, You are trying to find a type of bug called memory leak. A memory leak occurs when a program allocates memory (typically using functions like malloc, calloc, or new) but fails to deallocate or release that memory when it's no longer needed. You will give you the bug report and its corresponding code snippet, and your responsibility will be to analyze the bug within the calling context of the code snippet by following the steps as shown below. 1. Determine the variable and the location that may have the memory leak bug. You only need to consider the reported memory leak variable provided in the bug report. Please do not consider other pointer variables that may have the memory leak bug.You need to output the memory leak variable and its location in json format: { "mem_leak_var": var, "location": ["file": file, "line": line] }. 2. Extract the path condition. According to the control flow of the code, you need to extract the path conditions from the begining of the function to the location determined in the previous step. For example, for the code "int a = 0; if (b > 3) return 1; a += 1;", the path condition for reaching the statement "a += 1;" is "path_cond": ["int a = 0;", "if (b <= 3)", "a += 1;"]. 3. Analyze whether the determined location is reachable in the "path_cond". In this process, you don't need to consider the branches where the variable "mem_leak_var" failed to apply for memory. For example, for the code "int *p = (int*)malloc(sizeof(int)); if (!p) a = 1; *p += 2;", you should only go to the false condition (means the pointer p had successfuly applied for memory). Additionaly, if the developer's comments indicate that the bug was intentional or confirm that the issue is benign and requires filtering, please report it as a false alarm. In case you are still uncertain or require additional information, your answer should be unknown. You need to output analyze result "report_is_valid" with values "true", "false", or "unknown" , respectively. 4. Analyze whether the variable "mem_leak_var" has been freed (typically using functions like free, FREE, or delete) before returning from function. For example, if there is a free statement before return, "mem_leak_var" will be safely released. If there is no free statement or the free statement occur after returning from the function, memory leak will happen.You need to output analyze result "report_is_valid" with values "true", "false", or "unknown" in json format, respectively. In the last line of your answer, you should conclude with '@@@ real bug @@@', '@@@ false alarm @@@', or '@@@ unknown @@@'.

bug_category	prompt
user after free	# Task Description Now you should act as an expert in C/C++ code review, You possess advanced skills in analyzing program code using well-known static analysis tools. Additionally, You have extensive experience in finding a type of bug called use-before-initialization. It is worth noting that these static analysis tools frequently produce a significant number of warnings, which may consist of both false positives and redundancies. Consequently, it becomes essential for you to manually inspect and verify each warning. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. Initially, You should explain whether the bug can occur in the calling context of the code snippet. When You meet function calls, You need to accurately correspond to each pair of formal and actual parameters and synchronize changes to them. Please closely examine the scenarios in which each conditional branch evaluates as true or false and their feasibility given specific inputs. In other cases, functions will return with a return code. The caller then checks the return code to determine if the function was executed successfully. For example, "if(!func(...)) return" In this case, You should consider these return value checks and only go to successful conditions (means won't return directly) Thinking step by step. You only report a genuine bug when You are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. Additionally, your analysis should be field-sensitive. This means if some functions initialize the fields of their parameters (i.e, for func(struct some_struct* ptr), it may initialize ptr->config). Suppose You have difficulty analyzing fields without more information, such as a function definition, caller, and callee, try your best to guess the behavior of the function. Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.\

bug_category	prompt
Divide By Zero	# Task Description Now you should act as an expert in C/C++ code review. You possess advanced skills in analyzing program code using static analysis tools, such as Infer, Cppcheck, and Clang static analysis. You possess substantial expertise in reviewing and interpreting analysis reports, enabling accurate identification of false alarm reports. It is worth noting that these static analysis tools frequently produce a significant number of warnings, which may consist of lots of false alarms. Currently, you are trying to find a type of bug called Divide By Zero. A divide by zero error occurs when a program attempts to divide a number by zero, which leads to undefined behavior. This type of bug is often hard to detect, especially when the divisor is the result of complex expressions or user inputs. You will be given a bug report and its corresponding code snippet, and your responsibility will be to analyze the bug within the calling context of the code snippet by following the steps as shown below. 1、Determine the variable and the location that may have the Divide By Zero bug. You only need to consider the reported variable and location where the division operation is happening. Do not consider other variables that may cause divide by zero errors. 2、Extract the path condition.According to the control flow of the code, you need to extract the path conditions from the beginning of the function to the location where the divide operation happens. Analyze whether the divide by zero location is reachable in the "path_cond". You need to check if the path condition guarantees that the divisor will never be zero. For example, if there is an if condition checking whether the divisor is non-zero, the error will not happen when the condition is true. However, if the condition is missing or the divisor may still be zero under certain circumstances, a divide by zero error may occur. If the developer's comments indicate that the bug is intentional or benign, please report it as a false alarm. In case you are still uncertain or require additional information, your answer should be "unknown".You need to output the analysis result report_is_valid with values "true", "false", or "unknown" 3、Analyze whether the divisor variable has been checked for zero before the division operation.If the divisor div is checked for zero before performing the division, the bug is avoided. However, if no check is performed or the check occurs after the division, a divide by zero error will happen. Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, you should conclude with @@@ real bug @@@, @@@ false alarm @@@, or @@@ unknown @@@.

表1： C 项目 bug 对应的 prompt

bug_category	prompt
Java bad practices	Now you should act as an expert in Java code review, You possess advanced skills in analyzing program code using well-known static analysis tools. Additionally, You have extensive experience in identifying common Java bad practices, particularly issues like non-compliance with naming conventions, ignoring exception handling, and neglecting return values. These static analysis tools often generate a large number of warnings, including both false positives and redundant findings. As a result, it is crucial for you to manually review each warning in the context of the specific Java code snippet, taking into account the calling context and the broader program flow to assess whether the warning indicates an actual issue or a non-relevant finding. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. Initially, In the beginning, You review the code snippet carefully, focusing on how the method or class is used in the broader program. Check for relevant patterns like null checks, exception handling, and the return value being used appropriately. You should understand where and how the function or method is invoked. Is the code in question called with valid inputs.Finally,you should check for static analysis relevance, cross-reference the static analysis warning with the actual code. Does the warning indicate a potential real bug (e.g., unhandled exception, ignored return value, null pointer dereference)? Or does it reflect an edge case or false positive that doesn't align with the actual program flow? Thinking step by step. You only report a genuine bug when You are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. Suppose You have difficulty analyzing fields without more information, such as a function definition, caller, and callee, try your best to guess the behavior of the function. In case You are still uncertain or require additional information, your answer should be unknown. Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.

bug_category	prompt
CORRECTNESS	Now you should act as an expert in Java code review, You possess advanced skills in analyzing program code using well-known static analysis tools. Additionally, You have extensive experience in identifying CORRECTNESS issues like the logical integrity and functional behavior of Java programs These static analysis tools often generate a large number of warnings, including both false positives and redundant findings. As a result, it is crucial for you to manually review each warning in the context of the specific Java code snippet, taking into account the calling context and the broader program flow to assess whether the warning indicates an actual issue or a non-relevant finding. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. You must first examine the provided code snippet carefully. Focus on understanding the flow of execution—how data is passed through methods, whether proper exception handling is in place, and whether null checks are correctly implemented. Pay special attention to how return values are used and whether any conditions or combinations of conditions could potentially lead to incorrect results, null pointer dereferencing, or program crashes. You should simulate dynamic behavior where necessary, for instance, by using concrete values (or symbolic execution when possible) to understand whether a specific scenario could trigger the issue described. In particular, evaluate how method arguments are passed, whether null values could propagate through the call stack, and how exceptions are caught and handled (or neglected). If there are specific conditions (like null checks, exception handling, or method contract expectations), analyze how these factors interact and check whether the reported bug could actually occur in these situations. Thinking step by step. You only report a genuine bug when You are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. Suppose You have difficulty analyzing fields without more information, such as a function definition, caller, and callee, try your best to guess the behavior of the function. In case You are still uncertain or require additional information, your answer should be unknown. Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.
DODGY_CODE	Now you should act as an expert in Java code review, You possess advanced skills in analyzing program code using well-known static analysis tools. Additionally, You have extensive experience in identifying DODGY_CODE issues such as memory leaks or similar problems in Java. While Java has garbage collection, improper resource handling (like file or database connections) or potential memory leaks (via holding references unnecessarily) can still occur.

bug_category	prompt
	These static analysis tools often generate a large number of warnings, including both false positives and redundant findings. As a result, it is crucial for you to manually review each warning in the context of the specific Java code snippet, taking into account the calling context and the broader program flow to assess whether the warning indicates an actual issue or a non-relevant finding. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. 1、Determine the variable and location with the bug: Identify the variable and location that might have the issue, based on the bug report. Focus only on the variable explicitly mentioned in the bug report. Provide the result in the following JSON format:{"bug_var": "variable_name","location": {"file": "filename","line": line_number}} 2、Extract the path condition: Analyze the code flow from the start of the function to the identified location. Record the conditions and statements that would influence the reachability of the location. The path condition will be in this format:{"path_cond": ["statement1", "statement2", "location_reached"]} 3、Analyze the reachability of the bug location: Check whether the identified location in the path condition is reachable. Consider branches where the bug variable was involved (for example, if the variable has been properly allocated or initialized). If the developer indicates the bug is intentional or benign (e.g., in comments), treat it as a false alarm. If the analysis is unclear or requires further details, mark it as "unknown." 4、Analyze resource release or cleanup: In Java, memory management is handled by the garbage collector, but it's still important to check for proper resource cleanup. For example, if the variable refers to an external resource (e.g., Connection, FileInputStream), verify if there is a corresponding close() method or similar function before the function returns. If not, this could lead to a resource leak. Thinking step by step. You only report a genuine bug when You are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. Suppose You have difficulty analyzing fields without more information, such as a function definition, caller, and callee, try your best to guess the behavior of the function. In case You are still uncertain or require additional information, your answer should be unknown. Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.
EXPERIMENTAL Issues	Now you should act as an expert in Java code review, You possess advanced skills in analyzing program code using well-known static analysis tools. Additionally, You have extensive experience in identifying EXPERIMENTAL issues which could be the naming convention of a specific framework, library, or tool, indicating that this class or feature is in an experimental phase or not fully stable. These static analysis tools often generate a large number of warnings,

bug_category	prompt
	including both false positives and redundant findings. As a result, it is crucial for you to manually review each warning in the context of the specific Java code snippet, taking into account the calling context and the broader program flow to assess whether the warning indicates an actual issue or a non-relevant finding. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. You must begin by thoroughly reviewing the provided code snippet. Focus on understanding the execution flow—how data is passed between methods, whether proper exception handling is in place, and if null checks are correctly implemented. Pay close attention to how return values are used, and consider whether any conditions or combinations of conditions might lead to incorrect behavior, null pointer dereferencing, or potential crashes. When analyzing, consider simulating dynamic execution where applicable. This might include using concrete values (or symbolic execution when necessary) to determine if specific scenarios can trigger the reported issue. Carefully evaluate how arguments are passed to methods, and check if null values might propagate through the call stack. Additionally, observe how exceptions are managed: are they properly caught, or are they ignored, leading to potential issues? If the bug report includes specific conditions (such as null checks, exception handling, or expected method behavior), evaluate how these factors interact within the code. Check if the bug could realistically occur given the program flow and the context in which the function operates. Make sure to think through the issue step by step. You should only label something as a genuine bug if you're confident you've identified a specific code path that causes the issue to trigger. Otherwise, it's likely a false alarm. If you encounter difficulties due to insufficient information (such as missing function definitions, caller or callee contexts), do your best to hypothesize about the behavior of the code based on available context and reasonable assumptions. Always remain cautious and clear about the limits of your analysis. Thinking step by step. You only report a genuine bug when You are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. Suppose You have difficulty analyzing fields without more information, such as a function definition, caller, and callee, try your best to guess the behavior of the function. In case You are still uncertain or require additional information, your answer should be unknown. Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.
I18N	Now you should act as an expert in Java code review, possessing advanced skills in analyzing program code using well-known static analysis tools. Additionally, you have extensive experience in identifying I18N (Internationalization) issues, which could involve issues related to text encoding, locale-specific formats (such as dates, times, currencies), or

bug_category	prompt
	improper handling of language and cultural differences in the code. Static analysis tools may generate a range of warnings concerning I18N, including issues with improper encoding, lack of support for multiple locales, and improper formatting of localized data. These warnings may sometimes be false positives or overly broad, so it is crucial for you to manually review each warning in the context of the specific Java code snippet. Focus on understanding how the application manages various regions, cultures, and languages, and whether the warning represents an actual I18N issue or a non-relevant finding. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. Your review should start with a thorough analysis of how data is passed, stored, and displayed. Pay close attention to: Character Encoding: Ensure that the system handles multi-byte character sets correctly (e.g., UTF-8, UTF-16). Are there any potential issues with string encoding or decoding? Locale Awareness: Verify that methods and classes dealing with user input, date/time, currency, or number formats are properly locale-sensitive. Is there any hardcoding of locale values (such as using MM/DD/YYYY format without considering locale-specific formats)? Resource Bundles: Check for proper usage of ResourceBundle or equivalent mechanisms for loading localized strings based on user locale. Are all user-facing messages or text correctly extracted and placed in resource files, and is there fallback support for missing translations? Date/Time Formats: Ensure that date and time handling in the application takes locale-specific formats into account. Are there potential issues when formatting dates or times for regions with different conventions (e.g., DD/MM/YYYY vs. MM/DD/YYYY)? Number and Currency Formatting: Check that numeric and currency values are formatted according to the locale settings. Are there any hardcoded symbols or separators that could lead to incorrect outputs in different locales? String Comparisons: Look for any string comparison operations that might be locale-sensitive (such as using == instead of .equals() for comparing strings). Are there any string comparison or collation operations that could break for non-English locales? Error Messages and Logging: Ensure that any error messages or logs are properly internationalized. Are the error messages presented to users in their preferred language? Is there any case where the application logs non-localized text that could cause confusion for international users? Additionally, consider how these I18N features interact with other aspects of the application. Are there any areas where improper handling of locale or character sets could lead to bugs, crashes, or incorrect behavior? Simulate dynamic execution where applicable by considering different locale settings (e.g., US vs. FR vs. JP) to check if specific scenarios might trigger issues related to incorrect formatting, text display, or encoding errors. Lastly, evaluate if the bug report includes specific I18N conditions (such as locale issues, character encoding, or translation mishaps). Determine how these factors affect the program flow and whether they could realistically trigger an issue given the context of the function or method. Make sure to think through the issue step by step. You should only label

bug_category	prompt
	something as a genuine I18N bug if you are confident that an actual issue exists and can be triggered within a specific program flow. Otherwise, it is likely a false alarm or a non-relevant finding. In case You are still uncertain or require additional information, your answer should be unknown. If you encounter difficulties due to insufficient information (such as missing locale configurations or incomplete code snippets), try your best to hypothesize the behavior of the application based on the available context and reasonable assumptions. Always remain cautious and clear about the limits of your analysis. Thinking step by step. You only report a genuine I18N bug when you are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. In case You are still uncertain or require additional information, your answer should be unknown. Lastly, you will be asked to determine whether the bug is a real I18N bug or a false alarm. In the last line of your answer, you should conclude with '@@@ real bug @@@', '@@@ false alarm @@@', or '@@@ unknown @@@'.

bug_category	prompt
malicious code patterns	Now you should act as an expert in Java code review, You possess advanced skills in analyzing program code using well-known static analysis tools. Additionally, You have extensive experience in identifying malicious code patterns that can be exploited for security vulnerabilities or unintended behavior. Your task is to carefully analyze the provided code snippet and identify whether there are any indications of malicious behavior, such as intentional security loopholes, unauthorized access, or data manipulation. These static analysis tools often generate a large number of warnings, including both false positives and redundant findings. As a result, it is crucial for you to manually review each warning in the context of the specific Java code snippet, taking into account the calling context and the broader program flow to assess whether the warning indicates an actual issue or a non-relevant finding. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. Your analysis should include: 1、Investigating the calling context and data flow for suspicious patterns. 2、Checking for common malicious coding practices such as unsafe deserialization, unsanitized user input, hardcoded credentials, or insecure access control. 3、Ensuring no sensitive operations (e.g., file manipulation, network access) are exposed or performed inappropriately. 4、Carefully considering code structure to see if there's a hidden malicious payload or logic that can be exploited. When analyzing the code，Focus on the control flow and data manipulation, ignoring parts that are clearly benign (such as logging or basic utility functions). Consider possible attacker behaviors and whether they can exploit the code under typical execution conditions.If you find any path where malicious behavior can be triggered, confirm the bug. In case You are still uncertain or require additional information, your answer should be unknown. ou only report a genuine bug when You are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. Suppose You have difficulty analyzing fields without more information, such as a function definition, caller, and callee, try your best to guess the behavior of the function. In case You are still uncertain or require additional information, your answer should be unknown. Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.

bug_category	prompt
multithreaded correctness issues	Now you should act as an expert in Java code review, You possess advanced skills in analyzing program code using well-known static analysis tools. Additionally, You have extensive experience in identifying multithreaded correctness issues, such as data races, thread safety violations, or deadlocks. Your goal is to ensure that the code behaves as expected in a concurrent environment, without introducing unpredictable or erroneous behavior due to improper synchronization or thread management. These static analysis tools often generate a large number of warnings, including both false positives and redundant findings. As a result, it is crucial for you to manually review each warning in the context of the specific Java code snippet, taking into account the calling context and the broader program flow to assess whether the warning indicates an actual issue or a non-relevant finding. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. Initially,you should reviewing shared variables and ensuring proper synchronization (e.g., `synchronized`, `Lock`, `Atomic`).Identifying potential data races by analyzing how multiple threads interact with the same data. Checking for possible deadlocks, where threads are waiting on each other indefinitely.Looking for thread safety issues when accessing or modifying mutable state in a multi-threaded context. And verifying that critical sections are correctly protected to avoid unintended concurrent modification. Finally, if you find any path where thread safety is compromised, such as race conditions or deadlocks, confirm the bug. You only report a genuine bug when You are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. Suppose You have difficulty analyzing fields without more information, such as a function definition, caller, and callee, try your best to guess the behavior of the function. In case You are still uncertain or require additional information, your answer should be unknown Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.

bug_category	prompt
performance	Now you should act as an expert in Java code review, You possess advanced skills in analyzing program code using well-known static analysis tools. Additionally, You have extensive experience in identifying performance issues. These static analysis tools often generate a large number of warnings, including both false positives and redundant findings. As a result, it is crucial for you to manually review each warning in the context of the specific Java code snippet, taking into account the calling context and the broader program flow to assess whether the warning indicates an actual issue or a non-relevant finding. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. Your analysis should include: - Reviewing time and space complexity for operations, especially in loops, recursive calls, or large data structures. - Identifying inefficient use of collections, such as using `ArrayList` when a `HashMap` or `Set` might be more appropriate. - Checking for repeated or redundant operations that could be avoided by caching, memoization, or optimizing data access. - Analyzing database queries, network calls, or I/O operations for performance bottlenecks, such as unnecessary blocking or expensive operations in tight loops. - Looking for places where thread contention might slow down performance or where resources are not being released properly. When analyzing, consider both algorithmic improvements and implementation-level optimizations. You only report a genuine bug when You are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. Suppose You have difficulty analyzing fields without more information, such as a function definition, caller, and callee, try your best to guess the behavior of the function. In case You are still uncertain or require additional information, your answer should be unknown. Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.

bug_category	prompt
secuity	Now you should act as an expert in Java code review, You possess advanced skills in analyzing program code using well-known static analysis tools. Additionally, You have extensive experience in identifying secuity issues. These static analysis tools often generate a large number of warnings, including both false positives and redundant findings. As a result, it is crucial for you to manually review each warning in the context of the specific Java code snippet, taking into account the calling context and the broader program flow to assess whether the warning indicates an actual issue or a non-relevant finding. You will be provided with the code snippet and the bug report, and your task will be to examine the bug based on the calling context of the code snippet. Initially,you should ensuring that user inputs are sanitized properly and validated to prevent injection attacks (e.g., SQL injection, XSS). And checking the usage of cryptography to ensure sensitive data is encrypted and not exposed in plaintext, verifying proper authentication and authorization checks to prevent privilege escalation or unauthorized access. Then you should reviewing the use of third-party libraries or frameworks for known vulnerabilities, such as outdated dependencies or insecure configurations. Finally , analyzing file handling or network interactions to ensure that sensitive data or resources are not exposed to unauthorized parties. If you find any security flaws, such as missing validation or improper encryption, confirm the bug. You only report a genuine bug when You are highly confident and have accurately identified a specific pathway that triggers it. Otherwise, it is considered a false alarm. Suppose You have difficulty analyzing fields without more information, such as a function definition, caller, and callee, try your best to guess the behavior of the function. In case You are still uncertain or require additional information, your answer should be unknown. Lastly, You will be asked to determine whether the bug is a real bug or a false alarm. In the last line of your answer, You should conclude with '@@@ real bug @@@', '@@@ false alarm @@@' or '@@@ unknown @@@'.

表2： Java 项目 bug 对应的 prompt

4 结果分析

4.1 研究结果概述

本次实验中，我们评估了基于 **ChatGLM-3-Turbo** 的模型在不同数据集 DataSet 和提示词 Prompt 下的表现。总体上，我们的目标是计算分析模型的准确率、精确度、召回率和 F1 分数，以评估其在不同编程语言数据集（Java 和 CPP）上的效果。得到评估表现结果如下：

DataSet	Prompt	Total Test Cases Num	TP	TN	FP	FN	UK	Accuracy	Precision	Recall	F1
Java	BASIC	1074	124	345	132	239	234	55.83%	48.44%	34.16%	40.06%
	COR_COT	932	130	196	218	184	204	44.78%	37.36%	41.40%	39.27%
	BAP_COT	1035	131	211	205	202	286	45.66%	38.99%	39.34%	39.16%
	DOD_COT	1038	156	106	218	62	496	48.34%	41.71%	71.56%	52.70%
	EXP_COT	928	118	299	166	242	103	50.55%	41.55%	32.78%	36.65%
	I18N_COT	908	15	342	31	279	241	53.52%	32.61%	5.10%	8.82%
	MAL_COT	1058	147	288	212	251	160	48.44%	40.95%	36.93%	38.84%
	MUL_COT	1077	139	343	172	281	142	51.55%	44.69%	33.10%	38.03%
	PER_COT	1059	251	214	314	166	114	49.21%	44.42%	60.19%	51.12%
CPP	BOF_COT_FEW_SHOTS	364	4	222	65	8	65	75.59%	5.80%	33.33%	9.88%
	NQD_COT_FEW_SHOTS	249	4	112	100	4	29	52.73%	3.85%	50.00%	7.14%
	UAF_COT_FEW_SHOTS	253	11	121	56	2	63	69.47%	16.42%	84.62%	27.50%
	BOF_COT	431	11	179	150	3	88	55.39%	6.83%	78.57%	12.57%
	NQD_COT	460	9	231	156	7	57	59.55%	5.45%	56.25%	9.94%
	UAF_COT	144	11	39	78	4	12	37.88%	12.36%	73.33%	21.15%

表3： 研究结果

4.2 评估指标

TP（True Positive）：预测为正类且实际为正类的样本数量。

TN（True Negative）：预测为负类且实际为负类的样本数量。

FP（False Positive）：预测为正类但实际为负类的样本数量。

FN（False Negative）：预测为负类但实际为正类的样本数量。

UK（Unknown）：未知或无法分类的数据。

Accuracy（准确率）表示模型预测正确的样本占总样本的比例：

$$\text{Accuracy (准确率)} = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision（精确率）表示所有被预测为正类的样本中，实际为正类的比例：

$$\text{Precision (精确率)} = \frac{TP}{TP + FP}$$

Recall（召回率）表示所有实际为正类的样本中，被模型正确预测为正类的比例：

$$\text{Recall (召回率)} = \frac{TP}{TP + FN}$$

F1-score 是精确率和召回率的调和平均数，用于综合评估模型的性能：

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

4.2 Java数据集分析

在 Java 数据集上，我们测试了多种不同的 `Prompt` 提示词设置。主要的性能指标包括准确率（Accuracy）、精确度（Precision）、召回率（Recall）和 F1 分数。这些指标的计算结果如下：

- **BASIC** 提示词在准确率和精确率上表现较优，准确率达到 **55.83%**，精确度为 **48.44%**。这表明，在基本 Prompt 设置下，模型在分类准确性和精确度方面相对较为均衡，适合应用于对准确度要求较高的任务。
- **DOD_COT** 提示词在召回率和F1分数上表现突出，召回率达到 **71.56%**，F1 分数为 **52.70%**，依然显示出较强的识别能力，尤其适合注重召回的场景。在该提示词下，模型能够较为均衡地在准确性和召回率之间取得良好的平衡，体现出较强的性能。
- **I18N_COT** 提示词的结果相对较差，召回率仅为 **5.10%**，F1 分数为 **8.82%**。

4.3 CPP数据集分析

在 CPP 数据集上，我们也采用了多个不同的 `Prompt` 来进行评估。由于 `CPP` 数据集较高的编程语言复杂度和特异性，模型在该数据集上的表现存在较大的差异。

- **BOF_COT_FEW_SHOTS** 提示词在准确度上表现最强，达到了 **75.59%**，但其精确度非常低，仅为 **5.80%**，这表明模型在识别相关特征时产生了大量的误报。召回率为 **33.33%**，F1 分数为 **9.88%**，表明在提高准确率的同时，还需要进一步优化模型的精确度和召回率。
- **UAF_COT_FEW_SHOTS** 提示词的表现相对均衡，召回率为 **84.62%**，表明该设置对于捕捉正例具有较强的能力。然而，精确度仅为 **16.42%**，这意味着该模型可能存在过多的误报。F1 分数为 **27.50%**，表明其总体表现仍有提升空间。
- **BOF_COT** 在 **CPP** 数据集中的表现偏弱，召回率高达 **78.57%**，但精确度低至 **6.83%**。这一结果显示模型在识别正类时的能力较强，但由于误报较多，导致精确度极低。

4.4 总结

在 **Java 数据集** 中，**DOD_COT** 提示词在召回率上具有显著优势，适用于需要高召回的任务。尽管 **BASIC** 提示词在准确率和精确度上表现较为均衡，但仍有进一步提升的空间。**I18N_COT** 提示词的表现较差，显示了特定场景下模型的适应性问题。

在 **CPP 数据集** 中，尽管 **BOF_COT_FEW_SHOTS** 在准确率上表现突出，但精确度和 F1 分数较低，说明该设置在处理复杂的编程任务时存在较高的误报。**UAF_COT_FEW_SHOTS** 在召回率上表现较好，但精确度较差，表明该模型仍然面临较大的误报问题。**BOF_COT** 设置在 CPP 数据集上的效果较弱，精确度低，误报较多。

针对 **Java 数据集**，推荐使用**DOD_COT** 提示词进行源码警告识别，另外可以进一步优化 **I18N_COT** 提示词，提升其召回率和 F1 分数；而对于 **CPP 数据集**，推荐较为均衡的**UAF_COT_FEW_SHOTS** 提示词，另外需要加强模型对误报的控制，提升精确度，减少假阳性。整体上，提高模型的 **精确度** 和 **召回率** 之间的平衡，以及在不同 **Prompt** 设置下的适应性，是未来优化的关键方向。