Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We?

CHENG WEN, Guangzhou Institute of Technology & ICTT and ISN Laboratory, Xidian University, Guangzhou, China
YUANDAO CAI, Fermat Labs, Huawei Technologies Co., Ltd, Hong Kong, China
BIN ZHANG, College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
JIE SU, Guangzhou Institute of Technology & ICTT and ISN Laboratory, Xidian University, Guangzhou, China
ZHIWU XU, College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
DUGANG LIU, Guangdong Laboratory of Artificial Intelligence and Digital Economy (SZ), Shenzhen University, Shenzhen, China
SHENGCHAO QIN, Guangzhou Institute of Technology & ICTT and ISN Laboratory, Xidian University, Guangzhou, China
ZHONG MING, College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
TIAN CONG, ICTT and ISN Laboratory & Guangzhou Institute of Technology, Xi'an, China

Static analysis tools for capturing bugs and vulnerabilities in software programs are widely employed in practice, as they have the unique advantages of high coverage and independence from the execution environment. However, existing tools for analyzing large codebases often produce a great deal of false warnings over genuine bug reports. As a result, developers are required to manually inspect and confirm each warning, a challenging, time-consuming, and automation-essential task.

This article advocates a fast, general, and easily extensible approach called Llm4sa that automatically inspects a sheer volume of static warnings by harnessing (some of) the powers of Large Language Models (LLMs). Our key insight is that LLMs have advanced program understanding capabilities, enabling them to effectively act as human experts in conducting manual inspections on bug warnings with their relevant code

This work was supported in part by the National Natural Science Foundation of China (Nos. 62372304, 62302375, 62192734), the China Postdoctoral Science Foundation funded project (No. 2023M723736), and the Fundamental Research Funds for the Central Universities.

Authors' addresses: C. Wen, J. Su (Corresponding authors), and S. Qin (Corresponding author), Guangzhou Institute of Technology & ICTT and ISN Laboratory, Xidian University, Guangzhou, China; e-mail: wencheng@xidian.edu.cn, sujie01@xidian.edu.cn, shengchao.qin@gmail.com; Y. Cai, Fermat Labs, Huawei Technologies Co., Ltd, Hong Kong, China; e-mail: ycaibb@cse.ust.hk; B. Zhang, Z. Xu, and Z. Ming, College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China; e-mails: z343831143@163.com, xuzhiwu@szu.edu.cn, dugang.ldg@gmail.com; D. Liu, Guangdong Laboratory of Artificial Intelligence and Digital Economy (SZ), Shenzhen University, Shenzhen, China; e-mail: dugang.ldg@gmail.com; C. Tian, ICTT and ISN Laboratory & Guangzhou Institute of Technology, Xidian University, Xi'an, China; e-mail: ctian@mail.xidian.edu.cn.

ACM 1556-4681/2024/06-ART168
https://doi.org/10.1145/3653718

snippets. In this spirit, we propose a static analysis to effectively extract the relevant code snippets via program dependence traversal guided by the bug warning reports themselves. Then, by formulating customized questions that are enriched with domain knowledge and representative cases to query LLMs, Llm4sa can remove a great deal of false warnings and facilitate bug discovery significantly. Our experiments demonstrate that Llm4sa is practical in automatically inspecting thousands of static warnings from Juliet benchmark programs and 11 real-world C/C++ projects, showcasing a high precision (81.13%) and a recall rate (94.64%) for a total of 9,547 bug warnings. Our research introduces new opportunities and methodologies for using the LLMs to reduce human labor costs, improve the precision of static analyzers, and ensure software trustworthiness

CCS Concepts: • **Software and its engineering**; • **Security and privacy** → **Software and application security**;

Additional Key Words and Phrases: Large language model, static analysis, AI for program analysis, static bug warning, false alarms

ACM Reference Format:
Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Tian Cong. 2024. Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We?. *ACM Trans. Knowl. Discov. Data.* 18, 7, Article 168 (June 2024), 34 pages. https://doi.org/10.1145/3653718

1 INTRODUCTION

Static analysis is a crucial technique for ensuring reliability, security, and maintainability of software, the key characteristics of software trustworthiness. It automatically uncovers potential bugs or vulnerabilities without executing programs. Owing to its high coverage and independence from the runtime execution environment, static bug finding is one of the paramount practices throughout the software development process [13–15, 67, 68, 72, 77, 90]. For instance, static bug-finding tools are widely used in leading internet companies such as Google [64] and Meta/Facebook [5, 21], preventing hundreds of bugs from infiltrating the codebases daily.

Problem and Its Importance. Despite this remarkable progress, however, since highly precise static analysis is generally unscalable for large million-line programs, many industrial-strength tools have to sacrifice high precision, such as context sensitivity [7, 52] and path sensitivity [5, 26, 50, 72], in favor of superior efficiency. As a result, thoroughly scrutinizing the sheer volume of bug warnings to uncover true bugs remains a laborious process for software practitioners due to the significant human effort and expertise required [31, 32]. For instance, when analyzing the *Tmux* project with approximately 40,000 lines of code, the popular static analysis tool SVF [71] produces about 2,000 warnings that are hard to inspect and confirm manually [68]. It is widely observed that software developers often reject static analysis tools, particularly when the bug reports are flooded with a large number of false positives and require excessive scrutiny time [5, 7, 21].

Existing Techniques. Due to the importance of suppressing false positives, numerous researchers [33, 55] have present techniques aimed at automatically identifying genuine bugs from a multitude of static warnings. One promising research direction is to resort to dynamic analysis/testing. Specifically, researchers have tried to explore directed grey-box fuzzing [6, 30, 48, 83, 84] and dynamic symbolic execution [9] to sift through static analysis warnings and trigger the real bugs. However, it is quite time-consuming and challenging to dynamically trigger a single bug across different inputs, which can further deteriorate when faced with thousands of bug reports. For example, directed fuzzing [30, 48] generally requires a few hours or even dozens of hours to trigger a given bug, considerably hindering it from practical adoption. Additionally, constructing an executable environment for testing software is challenging in many scenarios, particularly in embedded software systems, due to their strong dependencies on hardware. Furthermore, dynamic approaches require appropriate test oracles to identify bugs [89, 94], which are often hard to acquire. Another research direction is identifying specific patterns from bug warnings, source code, and software repositories for predicting false positives [44, 80]. For example, machine learning techniques are often used to learn what is likely true and false positives [96]. However, the bug patterns they focus on are generally hard to reflect and capture real-world conditions, and, as a result, these approaches suffer from a low recall rate when applied to real-world programs [35].

Our Insight. This article advocates a novel and complementary approach called Llm4sa,[1] which automatically inspects a sheer volume of static analysis warnings by harnessing the capabilities of ***Large Language Models* (LLMs)** [98], such as ChatGPT [58, 61]. Our basic idea is that the LLMs can act as human experts to perform manual inspections on thousands of static bug warnings based on the relevant calling contexts. Specifically, this is because LLMs have recently demonstrated significant potential in comprehending and reasoning about code [46, 49, 74]. Different from the previous dynamic approaches [9, 30, 48] and static approaches [44, 80, 96], Llm4sa works by inspecting both *the bug reports* and *their corresponding code snippets* through querying LLMs, which explain the code snippets, reason about the reported warnings, and make a decision on whether a warning is a false positive based on expertise. Our approach offers several salient advantages that effectively alleviate the burden of manually confirming numerous static analysis warnings, saving considerable developer efforts: (1) Llm4sa can automatically inspect thousands of static warnings, with an average time of less than 30 seconds per warning. Thus, it is easy and effective to integrate Llm4sa into a static analysis pipeline. (2) Llm4sa is general and can be used to confirm a broad spectrum of different types of static warnings. (3) Llm4sa is easily extensible and orthogonal to (or complementary to) the previous approaches, as it derives only bug-related code snippets via static analysis and inspects warnings with the help of LLMs without executing code.

Our Approach. We develop the Llm4sa framework by identifying and addressing three main challenges when inspecting static warnings using decoder-only LLMs, such as ChatGPT. The first challenge is that LLMs have token limitations, so prompting LLMs with the entire project code is currently impractical. Our basic idea is that manual inspections of bug warnings suffice to focus on a few critical functions around the calling contexts, instead of the entire code. Based on this idea, Llm4sa derives the bug-related code snippets through static program-dependence traversal [25], which enriches the bug warnings with the necessary calling contexts for LLMs to inspect. The second challenge is that even though LLMs can comprehend the related code snippets and natural language in bug warnings (e.g., annotations), their capacity to effectively utilize this knowledge for warning inspection is limited. To improve the effectiveness, we propose prompt engineering techniques, including **Chain-of-Thought (CoT)** [82] and few-shot prompting [8], which significantly help LLMs understand the static warnings. The third challenge is that LLMs often produce unreliable and inconsistent responses [98]. To address this issue, Llm4sa performs pre-processing to convert bug reports from different static analyzers into a uniform format and post-processing to determine the confidence level of the LLMs' answers. To sum up, the three challenges are mitigated by combining *code snippet extraction* based on program dependency graph, *prompt engineering*, and *pre-/post-processing* techniques, respectively.

Evaluation. We have implemented Llm4sa and successfully integrated it into a practical static analysis pipeline. We performed a thorough evaluation of Llm4sa on the Juliet Test benchmark, 3 embedded real-time operating systems, and 11 widely used real-world applications, with three popular open-source static analysis tools (i.e., Cppcheck, Csa, Infer). In total, Llm4sa can inspect a total of 9,547 static warnings, resulting in a high precision rate (81.13%) and recall rate (94.64%). In addition, Llm4sa is characterized by its speed, which averages less than 30 seconds per warning, and affordability, costing only $0.31 per warning.

[1]The acronym of **L**arge **L**anguage **M**odels for(**4**) **S**tatic **A**nalysis

To sum up, this article makes the following contributions:

— *Novelty.* We propose a fast, general, easily extensible approach for the first time, to the best of our knowledge, automating the inspection of a sheer volume of static warnings by harnessing the capabilities of LLMs.
— *Practical Approach.* We identify and address several practical challenges by combining the effective extraction of related code snippets via program dependence traversal, prompt engineering, and customized pre-/post-processing.
— *Evaluation.* We extensively evaluate Llm4sa on a total of 9,547 static bug warnings from the Juliet Test benchmark, 3 embedded real-time operating systems, and 11 widely used real-world applications to show its efficiency, scalability, and practicality.
— *Study and New findings.* To comprehensively understand the false positives, we conduct an in-depth characteristic study for the bug reports produced by three popular static bug-finding tools. We mainly conclude that LLMs are promising in identifying many false alarms in an affordable way.

We have released the implementation and all associated publicly available data to encourage comparable and evidence-based studies on automated static warnings inspection: https://doi.org/10.5281/zenodo.8346515

2 BACKGROUND AND MOTIVATION

In this section, we motivate our approach by employing two real-world examples, providing the necessary background on static analysis as well as LLMs, and presenting a desired LLMs-powered static analysis pipeline.

2.1 Motivating Examples

We present two motivating examples of false positives reported by mainstream static analyzers to illustrate the challenges of employing static analysis in practice.

Figure 1 shows a code snippet from the *Zephyr* project, an open-source embedded real-time system. The code implements a callback function for reading data from a USB endpoint. A null pointer dereference bug is reported by Cppcheck at line `190` in the `acl_read_cb` function, specifically in relation to the `buf` variable. However, this is a false alarm, because the `buf` variable was initialized as `NULL` at line `187` by passing parameters from a function call at line `230`, which also sets the second parameter (i.e., the `size` variable) to zero. Therefore, the condition at line `189` cannot be satisfied, and line `190` is unreachable from the `bluetooth_status_cb` function.

Figure 2 shows another code snippet from the *RIOT* project, an embedded operating system for low-end devices. The code implements a function for dividing two unsigned integers and returning the remainder. The code intentionally causes a division by zero error at line `83`, as the dividend is directly set to zero at line `79`. Almost all mainstream static analyzers are able to report this bug. However, the code is specifically designed for embedded systems that require a divide-by-zero exception handler to run into a situation. The developer's comment at lines `81–82` indicates that this divide-by-zero is intentional, revealing concealed knowledge that is not captured by static analyzers.

2.2 State of the Practice

In this section, we investigate static bug-finding tools. Specifically, we chose three representative and well-known static analyzers for C/C++ code, including Cppcheck, Csa, and Infer. These static analyzers are integrated with various state-of-the-art analysis techniques (e.g., symbolic execution, separation logic, and pattern matching), are popular among practitioners, and are extensively evaluated and used in both the industry and academia [5, 7, 14, 15, 21, 23, 68].

```c
// zephyr-v2.1.0/subsys/usb/class/buluetooth.c:185-194
static void acl_read_cb(unsigned char ep, int size, void *priv)
{
    struct net_buf *buf = priv;

    if (size > 0) {
        buf->len += size;
        bt_buf_set_type(buf, BT_BUF_ACL_OUT);
        net_buf_put(&tx_queue, buf);
        buf = NULL;
    }
... }

// zephyr-v2.1.0/subsys/usb/class/buluetooth.c:210-252
static void bluetooth_status_cb(struct usb_cfg_data *cfg,
                                enum usb_dc_status_code status,
                                const u8_t *param)
{
    ARG_UNUSED(cfg);

    /* Check the USB status and do needed action if required */
    switch (status) {
...
    case USB_DC_CONFIGURED:
        LOG_DBG("USB device configured");
        /* Start reading */
        acl_read_cb(bluetooth_ep_data[HCI_OUT_EP_IDX].ep_addr, 0, NULL);
        break;
...
    default:
        LOG_DBG("USB unknown state");
        break;
    }
}
```

Fig. 1. A *null pointer dereference* false alarm in *zephyr v2.1.0*.

```c
// RIOT-2020.04/sys/quad_math/qdivrem.c:57-83
/* __qdivrem(u, v, rem) returns u/v and, optionally, sets *rem to u%v.
 *
 * We do this in base 2-sup-HALF_BITS, so that all intermediate products
 * fit within u_int.  As a consequence, the maximum length dividend and
 * divisor are 4 `digits' in this base (they are shorter if they have
 * leading zeros).
 */
unsigned int
__qdivrem(unsigned int uq, unsigned int vq, unsigned int *arq)
{
    union uu tmp;
...
    /*
     * Take care of special cases: divide by zero, and u < v.
     */
    if (vq == 0) {
        /* divide by zero. */
        static volatile const unsigned int zero = 0;

        /* cppcheck-suppress zerodiv
         * (reason: division by zero is on purpose here) */
        tmp.ul[H] = tmp.ul[L] = 1 / zero;
...     }
```

Fig. 2. A potential *division by zero* in *RIOT-2020.04*.

(1) Cppcheck [52] is a typical pattern matching-based technique combined with a lightweight data-flow analysis. Specifically, Cppcheck scans C/C++ source code for potential bug patterns with the reports stored in a local database. Although the tool is comprehensive (i.e., armed with multiple bug checkers) and highly efficient, Cppcheck can suffer from high false positives in large codebases, as revealed by the previous work [47].
(2) Csa [41] is based on a typical path-sensitive symbolic execution technique and built on the LLVM/Clang static analysis toolchain, which, however, is restricted to a single translation unit (e.g., a single file). In other words, any function call to a function outside the translation

unit is over-approximated, incurring high false positives. For instance, a variant example of Figure 1 that splits functions `acl_read_cb` and `bluetooth_status_cb` into two different files would lead to a false alarm reported by Csa.

(3) Infer [17] is a typical separation logic-based technique (some checkers may have combined other static analysis approaches [5, 7]). Specifically, Infer utilizes separation logic and bi-adduction for reasoning about memory manipulations to prove certain memory safety conditions and create program state summaries for each function in an analyzed program. Like other static analyzers, Infer is also prone to false positives.

Given a great deal of false positives generated by the static analysis analyzers, developers are faced with the arduous task of manually reviewing numerous bugs specifically in large programs, which is time-consuming. Moreover, the abundance of false positives in reports often leads developers to disregard the use of static analysis, thereby compromising the reliability of their software. As a result, it is crucial and urgent to introduce automated approaches for reviewing bug warnings.

How Does Manual Inspection Typically Perform? Upon receiving a static warning, developers typically begin by reviewing the bug report and attempting to locate the error location in the source code using an **Integrated Development Environment (IDE)**, such as VS Code or Eclipse. To confirm the presence of bugs, developers typically concentrate on specific function bodies, callers, and callees associated with error traces. They determine the feasibility of a bug based on a deep understanding of the code logic and the library implementation, which may not be obvious or easy for developers unfamiliar with the code or the tool.

Actually, most of the static warnings can be easily confirmed or pruned by experienced developers through code review. For instance, in Figure 1, the developers would examine `buletooth.c` and locate line `190` after a quick view of the bug report. It is obvious that the `buf` variable is defined by the third parameter `priv` at line `187`. Therefore, the value of `buf` depends on its call site. Then, they would carefully track the error trace reported by the static analyzer or try to examine all the call site that calls `acl_read_cb`. By capturing the function's arguments at line `230`, they were able to simulate symbolic execution and determined that the condition at line `189` always evaluates to the false branch. This static warning can ultimately be suppressed due to the static analyzer's context or path sensitivity. For another instance in Figure 2, it is easy to dismiss by the developers as long as they understand the high-level code logic. By quickly reviewing the natural language comment, it becomes evident that this is an insignificant warning.

2.3 A Desired LLMs-powered Static Analysis Pipeline

We begin by presenting a fundamental introduction to LLMs [98] that underpin our approach. LLMs are neural models trained on extensive text data that encompasses both natural language and source code, through employing self-supervised learning objectives. Specifically, LLMs have been trained on tremendous and diverse datasets, allowing them to exhibit proficient capabilities in simulating human language skills. As a result, they have brought about significant advancements across multiple domains. Recent advancements in decoder-only LLMs, like ChatGPT [58], a general LLM released by OpenAI, has demonstrated exceptional proficiency in understanding program code and is increasingly employed in the realm of program analysis [20, 29, 45, 46, 74, 75]. One advantageous characteristic of LLMs lies in their adaptability to diverse tasks, facilitated by prompt engineering techniques [8, 82]. These techniques involve designing effective input prompts to elicit desired outputs from LLMs. Inspired by this characteristic, we believe that the adaptability of LLMs provides a promising alternative for comprehending bug reports expressed in natural languages, analyzing code behavior, and assessing the consistency between code behavior and the bug description produced by static analysis tools.

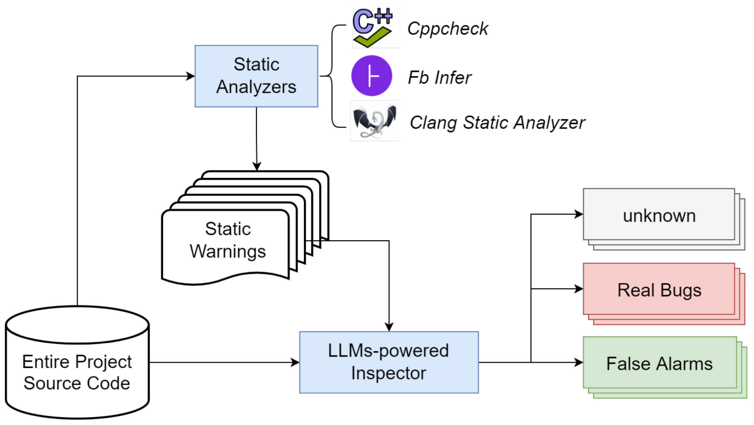


Fig. 3. A static analysis pipeline equipped with Llm4sa, improving the precision of Cppcheck, Infer, and Csa.

Our research is driven by the status quo, where the manual inspection of static analysis warnings is laborious and time-consuming for software practitioners due to the sheer volume of false positives and the discrepancies between tools and their corresponding reports [5, 68]. In light of this, we propose leveraging the natural language processing and code comprehension capabilities of LLMs to automate the manual inspection process for thousands of static warnings. Specifically, we envision an LLM-powered static analysis pipeline called Llm4sa, as depicted in Figure 3, which could fully automatically inspect each static warning, capturing real bugs with high precision and efficiency. In general, the most effective approach for validating a potential bug may vary based on the specific circumstances and the preferences of the developers with their experience. We believe that, when dealing with large codebases that produce plenty of bug reports, traditional approaches (e.g., manual inspection) can be challenging to use and often require extensive expertise to maximize their effectiveness, limiting their practical applicability. Comparatively, Llm4sa provides a more practical and intuitive method for examining bug warnings, effectively substituting human involvement in the process. With its natural language processing and knowledge representation capabilities, Llm4sa can analyze code snippets, offer bug explanations in a developer-friendly manner, scrutinize bug warnings, and confirm the genuineness of reported bugs Specifically, a comparison of the advantages of Llm4sa with other traditional approaches is presented in Table 1, which will be further discussed in Section 5.

To illustrate the responses of LLMs, specifically ChatGPT used in Llm4sa, we provide sample outputs in Figure 4 against the motivating examples mentioned above. Specifically, Llm4sa can explain static bug warnings and code logic, elucidating the specific reasons behind a bug occurrence in a particular piece of code, as well as identifying cases of false alarms. By leveraging its understanding of the code and the connections between the code and the bug report, Llm4sa generates explanations when it detects false alarms or genuine bugs in the code. These explanations are valuable in helping developers comprehend the root cause of the bug and provide guidance on how to fix it. Ultimately, Llm4sa draws conclusions based on these explanations, and the results are output as the format “@@@ result @@@” in the last line, facilitating post-processing and seamless integration into the pipeline.

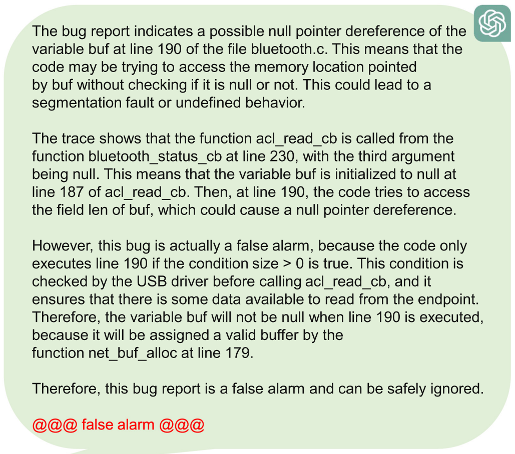
3 METHODOLOGY

3.1 Overview

We first outline the internal workflow of Llm4sa shown in Figure 5. First, Llm4sa conducts the pre-processing to unify and convert the bug reports produced by different static bug-finding tools.

Table 1. Capability of Llm4sa Compared to Other Approaches (e.g., Manual Inspection, Dynamic Approaches, Pattern Matching)

Capability	**Description**
Speed	Llm4sa is fast in reasoning, explaining reports, and making conclusions for bug warnings, compared to traditional approaches like symbolic execution, which generally requires much time in solving path constraints for generating valid inputs [11].
Ease of use (developer-friendly)	The strong natural language generation capabilities of Llm4sa powered by LLMs can facilitate developers in comprehending its results, while traditional approaches are difficult to reach. For example, Infer [21] reports only the program line where the bug finally triggers, rendering it difficult for users to manually confirm.
Integration with existing tools	Llm4sa offers much flexibility and can easily be integrated into a static analysis pipeline, as it solely operates on the outputs (i.e., bug reports) without intruding into the internal workings of static analysis tools.
Scalability	Llm4sa can inspect large reports from analyzing large programs, thanks to extracting relevant code snippets through effective program dependence reasoning. In contrast, traditional approaches like dynamic testing make it difficult to generate specific inputs and configurations that trigger bugs in large programs.
Cost of money	Llm4sa leverages readily available (or affordable) LLMs, which are often accessible as cloud-based services, providing a flexible pricing model. In contrast, traditional approaches like dynamic symbolic execution can require powerful computer servers to store and solve complex path constraints.
Applicability	Llm4sa offers the capability to validate bug warnings generated by static tools without the need for constructing dynamic execution environments. Thus, Llm4sa is applicable to a wide range of applications, including embedded systems, where dynamic testing often requires specific hardware preparation.
Precision	Llm4sa is generally precise due to the strong code understanding of LLMs that can be further improved by using advanced LLMs, superior prompt engineering, and fine-tuning. Note that previous approaches like dynamic symbolic execution can offer higher accuracy but come with prohibitive costs and low efficiency.


The bug report indicates a possible null pointer dereference of the variable buf at line 190 of the file bluetooth.c. This means that the code may be trying to access the memory location pointed by buf without checking if it is null or not. This could lead to a segmentation fault or undefined behavior.

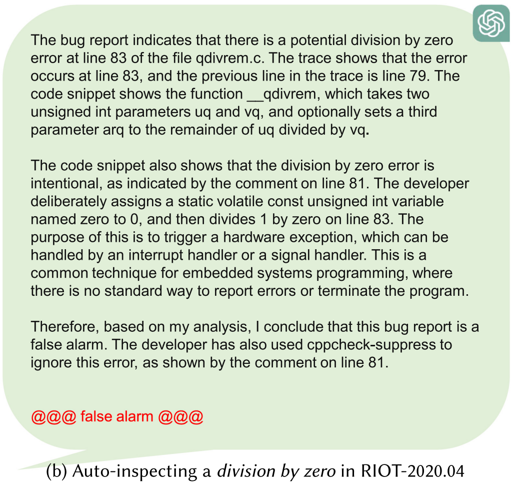
The trace shows that the function acl_read_cb is called from the function bluetooth_status_cb at line 230, with the third argument being null. This means that the variable buf is initialized to null at line 187 of acl_read_cb. Then, at line 190, the code tries to access the field len of buf, which could cause a null pointer dereference.

However, this bug is actually a false alarm, because the code only executes line 190 if the condition size > 0 is true. This condition is checked by the USB driver before calling acl_read_cb, and it ensures that there is some data available to read from the endpoint. Therefore, the variable buf will not be null when line 190 is executed, because it will be assigned a valid buffer by the function net_buf_alloc at line 179.

Therefore, this bug report is a false alarm and can be safely ignored.

@@@ false alarm @@@

(a) Auto-inspecting a *null pointer dereference* in *zephyr v2.1.0*


The bug report indicates that there is a potential division by zero error at line 83 of the file qdivrem.c. The trace shows that the error occurs at line 83, and the previous line in the trace is line 79. The code snippet shows the function __qdivrem, which takes two unsigned int parameters uq and vq, and optionally sets a third parameter arq to the remainder of uq divided by vq.

The code snippet also shows that the division by zero error is intentional, as indicated by the comment on line 81. The developer deliberately assigns a static volatile const unsigned int variable named zero to 0, and then divides 1 by zero on line 83. The purpose of this is to trigger a hardware exception, which can be handled by an interrupt handler or a signal handler. This is a common technique for embedded systems programming, where there is no standard way to report errors or terminate the program.

Therefore, based on my analysis, I conclude that this bug report is a false alarm. The developer has also used cppcheck-suppress to ignore this error, as shown by the comment on line 81.

@@@ false alarm @@@

(b) Auto-inspecting a *division by zero* in RIOT-2020.04

Fig. 4. Sample results from Llm4sa for the code snippets in Figure 1 and Figure 2.

As the tools can generate numerous warnings for a particular project, Llm4sa splits the whole report into a single warning as the fundamental unit of automatic inspection. In addition, as the varying formats of static warnings generated by different tools result in discrepancies in the reported information, Llm4sa converts these static warnings into a unified and comprehensive

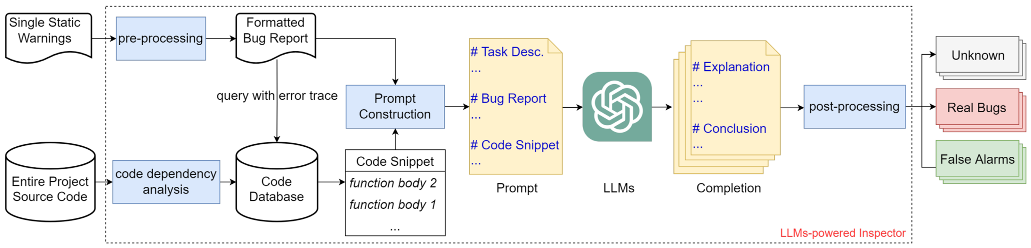


Fig. 5. The workflow of LLM4SA, consisting of the pre-processing, code snippet extraction, prompt engineering, and post-processing.

format, whereby encoding the bug type, description, bug location/trace, and other relevant data. Note that this conversion integrates crucial information that facilitates comprehension for both sides of LLMs and developers.

Second, LLM4SA employs program dependency analysis to create a code database that stores code snippets related to bug reports by analyzing the entire project source code under review. Specifically, extracting the necessary code snippets mitigates the token limitation of LLMs. By utilizing the information from the formatted bug report (e.g., error trace/location), LLM4SA conducts a static program dependency analysis to identify the relevant function bodies associated with the bug. Additionally, LLM4SA identifies the essential calling context, such as callers and callees, in the code snippet. Consequently, the code snippet extraction algorithm in LLM4SA strives to generate a concise and comprehensive code snippet, encompassing sufficient information for LLMs to inspect the bug.

Third, after deriving the code snippets, LLM4SA efficiently constructs prompts that facilitate the querying of LLMs for the purpose of inspecting static warnings. Specifically, the prompts describe the task of confirming whether the bug warning represents a real bug or not via a natural language description. By putting the formatted bug report and the corresponding code snippet together and leveraging prompt engineering techniques, LLMs are guided to provide improved explanations and make precise conclusions. We use prompt engineering techniques (e.g., **Chain-of-Thought (CoT)** and few-shot prompting) to improve the accuracy and consistency of the LLMs' answers.

Finally, LLM4SA performs post-processing, which consists of determining the confidence level of the LLMs' responses by considering factors including the proportion of consistent answers. Specifically, the post-processing aims to mitigate the problem of unreliable and inconsistent responses produced by LLMs. Based on the confidence level, LLM4SA is able to classify the static warning into one of three categories: *False Alarm*, *Real Bug*, or *Unknown*.

We have briefly outlined the internal workflow of LLM4SA. In the following subsection, we will shed light on how LLM4SA works in detail, covering the four aspects: (1) the pre-processing on static bug warnings; (2) the code snippet extraction based on program dependency analysis; (3) the prompt engineering; (4) the post-processing.

3.2 Pre-processing on Static Bug Warnings

At a high level, the initial pre-processing focuses on transforming bug reports generated by various static bug-finding tools into a standardized format. This enables LLM4SA to handle the reports in a uniform and streamlined manner. The standardized format captures basic bug characteristics, which can be easily adapted to accommodate reports from different tools.

Bug type Mapping and Grouping. Static analyzers often use different labels or identifiers to categorize the types of bugs they analyze, such as CPPCHECK employing **Common Weakness Enumerations (CWEs)** while others use their own identifiers. As a result, inconsistent bug

Table 2. Bug Type Mapping and Grouping

Bug Type (abbreviation)	CWE ID	CWE Title	Bug Identifiers in Different Static Analyzer

Null pointer dereference (NPD)	CWE-476
	CWE-690
		NULL Pointer Dereference
		Unchecked Return Value to NULL Pointer Dereference
			Cppcheck: nullPointer;
			Infer: Null Dereference;
			Csa: Dereference of null pointer, Dereference of undefined pointer value.

Uninitialized variable (UVA)	CWE-457
	CWE-824
		Use of Uninitialized Variable
		Access of Uninitialized Pointer
			Cppcheck: uninitvar, uninitdata, uninitStructMember, legacyUninitvar;
			Infer: Uninitialized Value;
			Csa: Uninitialized argument value, Assigned value is garbage or undefined.

Use after free (UAF)	CWE-415
	CWE-416
		Double Free
		Use After Free
			Cppcheck: doubleFree, deallocuse,deallocDealloc;
			Infer: Use After Free, Use After Delete;
			Csa: Use-after-free.

Divide By Zero (DBZ)	CWE-369	Divide By Zero	Cppcheck: zerodiv,; Infer: Divide By Zero; Csa: Division by zero.

Memory leak (ML)	CWE-401	Missing Release of Memory after Effective Lifetime	Cppcheck: memleak, memleakOnRealloc;
			Infer: Memory Leak;
			Csa: Memory leak, Free alloca().

Buffer overflow (BOF)	CWE-121
	CWE-122
	CWE-124
	CWE-125
	CWE-126
	CWE-127
		Stack-based Buffer Overflow
		Heap-based Buffer Overflow
		Buffer Underwrite
		Out-of-bounds Read
		Buffer Over-read
		Buffer Under-read
			Cppcheck: arrayIndexOutOfBounds, bufferAccessOutOfBounds, arrayIndexOutOfBoundsCond;
			Infer: Buffer Overrun L2, Buffer Overrun L3, Buffer Overrun S2;
			Csa: Out of bound memory access.

identifiers and prompts can occur even for the same bug across different tools. To address this issue, we have developed a mapping and grouping approach that assigns each specific bug identifier from different analyzers to a corresponding CWE ID, allowing for automated and easily extensible pre-processing to assess if the bug types identified by the tools align with the actual bugs in the code. An example of this mapping is shown in Table 2, which demonstrates how various bug identifiers from different analyzers are linked to the null-pointer-dereference bug type through the CWE-476 and CWE-690 identifiers.

A Uniform Format in JSON. Different static analyzers generate bug reports in various formats that encode various kinds of program information to help developers locate and understand the bugs. For example, Cppcheck, Csa, and Infer create bug reports in various forms, such as XML, PLIST, and JSON, respectively. To unify the reports, we convert the bug warning to a customized format in JSON, including six general information: the bug type, the file path, the function name, the error location, the error trace, and the corresponding description.

(1) The bug type denotes the category of bugs, such as null pointer dereference, use-after-free, and so on.
(2) The file path indicates the path of the source code file where the bug is detected.
(3) The function name field specifies the function in which the bug can occur.
(4) The error location field provides the exact line and column number in the source code file where the bug is detected.
(5) The error trace field presents a stack trace that showcases the execution flow leading up to the bug.
(6) The description field comprehensively explains the bug in a natural language description for users.

The detailed information mentioned above serves to assist developers in efficiently identifying bugs, locating them within the code and comprehending the associated code segments. Furthermore, static analyzers offer natural language descriptions of bugs, which significantly aid developers in understanding and pinpointing the bugs. In Table 3, various descriptions of a null pointer dereference bug from different static analyzers are consolidated into a single field within the

Table 3. Bug Reports from Different Static Analyzers for the Same Bug

Bug Type	Static Analyzer	Format of Report	Description
Null pointer dereference	CPPCHECK	.xml	subsys/usb/class/bluetooth:190 Possible null pointer dereference: buf.
	CSA	.plist	subsys/usb/class/bluetooth:190 Access to field “len” results in a dereference of a null pointer (loaded from variable “buf”).
	INFER	.json	subsys/usb/class/bluetooth: 190 pointer “buf” last assigned on line 187 could be null and is dereferenced at line 190.

formatted bug report. By converting bug warnings into a standardized JSON format, LLM4SA ensures that all bug-related information is organized consistently. As a result, our pre-processing on static bug warnings facilitates seamless processing, analysis, and sharing of bug data among developers, LLMs, and other bug management tools.

3.3 Code Snippet Extraction Based on Program Dependency Analysis

As mentioned, manual inspections typically do not costly analyze the entire project code; instead, the process typically focuses on a few critical functions around the reported bugs. Based on this observation, LLM4SA derives only bug-related code snippets that are enriched with the necessary calling contexts. To achieve this, LLM4SA extracts code snippets related to bugs from the analyzed program by combining bug reports and the traversal of the program dependency graph. At a high level, the program dependency graph [25] is a directed graph that characterizes the data dependence and control dependence relationships [2] among functions, statements, and variables in a program. To extract the relevant functions, we begin by identifying the relevant statements that are data- or control-dependent on the statements described in the bug reports. The relevant functions are then determined by considering all the functions in which the identified relevant statements are located. We incorporate the concepts of control dependence and data dependence and define the program dependence graph G formally as below.

Control dependence and data dependence are two types of dependencies that affect the execution of program instructions.

Definition 3.1. The program dependence graph of a program can be considered as a triple $G = (N, E_d, E_c)$, where

— N is the node set. Each node is a statement or, equivalently, the variable defined by the statement.
— $E_d \subseteq N \times N$ is a set of directed edges representing data dependence. Each edge is from one statement to the other, which refers to the variable defined in the source statement.
— $E_c \subseteq N \times N$ is a set of directed edges representing control dependence. Each edge is from a statement to an if-statement—the source statement is reachable at runtime if and only if the if-statement is reachable and the branch condition defined in the if-statement is true.

To provide more clarity, our code snippet extraction algorithm first focuses on selecting a specific subset of nodes and edges from the program dependency graph that is directly relevant to the bug warning. In principle, the relevance is determined based on two criteria: (1) the nodes that are explicitly mentioned in the bug warning and (2) the nodes that can be reachable transitively from other relevant nodes (via data-dependent or control-dependent edges). The outcome of our code snippet extraction is a collection of function bodies encompassing the relevant nodes, providing a concise representation of the code snippet associated with the bug warning. For instance, in the example depicted in Figure 1, if a bug warning highlights a null pointer dereference issue involving the variable `buf` within the function `acl_read_cb`, then we collect the nodes that can be

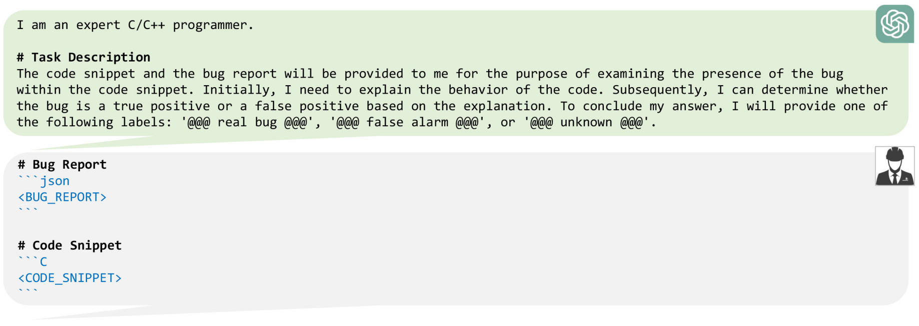
ALGORITHM 1: Code Snippet Extraction through traversal on program dependency graph

```
   Input: A bug warning $W$ and a constructed program dependency graph $G$
   Output: a code snippet $S$ including a set of function bodies that are relevant to $W$
1  $errorTrace$ = extract_trace_info($W$) ;        // extract the error trace from a bug warning $W$
2  $Worklist \leftarrow \emptyset$ ; // initialize a worklist $Worklist$ that is used to collect relevant nodes in $W$
3  for each $l$ in $errorTrace$ do
4      $n$ = find_mapped_node($G, l$) ;        // $l$ denotes a program line; find the node $n$ in $G$ that
         corresponds to the $l$
5      $Worklist \leftarrow Worklist \cup n$ ;      // put the relevant nodes described in a report to $Worklist$
6   $N \leftarrow \emptyset$ ;            // initialize a set $N$ that is used to collect all nodes related to $W$
7  while $Worklist! = \emptyset$ do
8      $n \leftarrow$ select_and_remove_a_node ($Worklist$) ;         // We transitively collect all reachable
         (relevant) nodes
9      if $n$ is not visited then
10         for $n'$ is data-dependent on $n$ in $G$ do
11             $Worklist \leftarrow Worklist \cup n'$ ; // get the reachable node $n'$ through data-dependent edge
                 $e, e \in E_d$
12         for $n'$ is control-dependent on $n$ in $G$ do
13             $Worklist \leftarrow Worklist \cup n'$ ;  // get the reachable node $n'$ through control-dependent
                 edge $e, e \in E_c$
14         $N \leftarrow N \cup n$ ;                              // Collect all reachable $n$ to $N$
15  $S \leftarrow \emptyset$ ;                                          // initialize an empty set $S$
16 for each $n$ in $N$ do
17     $f \leftarrow$ retrive_function ($n$) ;          // Collect all the functions that contain each node
18     if $f$ is not in $S$ then
19         $S \leftarrow f \cup S$ ;                            // Avoid collecting repeated nodes
20 return $S$
```

transitively reached based on the data and control dependence relationships encoded within the graph, outputting the two functions `acl_read_cb` and `bluetooth_status_cb`.

Formally, our code extraction algorithm is presented in Algorithm 1, which takes two inputs: a bug warning W generated by a bug-finding tool and a program dependency graph G. The bug warning W includes details about the bug's type, message, and the specific program location within the source code that is of interest. In our algorithm, we denote a program line as l. The output of the algorithm is a code snippet S, which consists of a collection of relevant function bodies associated with the bug warning W. In detail, the algorithm works as follows:

(1) Lines 1–6: To begin, we gather the error trace $errorTrace$ from the bug warning W, which is a sequence of program locations (each one is denoted as l) indicating where the bug occurs and the path it takes in the source code. It is important to note that, in certain cases, the trace might only provide the error location (i.e., a single l) due to the limitations of the static analyzer. Using this error trace, we gather the corresponding program nodes mentioned in the trace by retrieving the graph G and adding each node n to a worklist $Worklist$.
(2) Lines 6–15: Next, we proceed with traversing the program dependency graph G to gather additional nodes that are relevant to the initial nodes mentioned in the bug report W. This is done by iterating over each node n that is removed from the $Worklist$ later. From node n, we collect all nodes in the graph G that can be reachable through either control-dependent edges (E_c) or data-dependent edges (E_d). The purpose of collecting these reachable nodes is



```
I am an expert C/C++ programmer.

# Task Description
The code snippet and the bug report will be provided to me for the purpose of examining the presence of the bug
within the code snippet. Initially, I need to explain the behavior of the code. Subsequently, I can determine whether
the bug is a true positive or a false positive based on the explanation. To conclude my answer, I will provide one of
the following labels: '@@@ real bug @@@', '@@@ false alarm @@@', or '@@@ unknown @@@'.
```

~~~
# Bug Report
```json
<BUG_REPORT>
```

# Code Snippet
```C
<CODE_SNIPPET>
```
~~~

Fig. 6. Example of prompt used to inspect bug warnings.

to capture any nodes that are connected to the occurrence of the bugs. We also gather all the reachable nodes n and store them in the set N.

(3) Lines 15–20: Finally, we retrieve the functions (each one is denoted as f) that encompass the nodes collected in the previous step and include them in the code snippet S. Intuitively, the retrieved functions are those relevant functions where the relevant statements are located. To ensure that there are no duplicate entries, we avoid adding functions that have already been included in S. The code snippet S thus comprises the relevant function bodies associated with the bug warning, along with any annotations. This code snippet provides the essential information for language models to comprehend the warning effectively.

Performing precise program dependency analysis via pointer analysis on a large-scale codebase is widely recognized as time-consuming and not scalable. To offer a more practical alternative, our observation is that developers often rely on IDE environments for source code comprehension, as the syntax in the source code strongly implies control dependence and data dependencies; these IDE environments take advantage of fast scans to assist in locating relevant function definitions. Based on this insight, we develop a lightweight and efficient approach that shares similarities with the code scanning analysis employed by IDEs. Specifically, *CodeQuery* [63] is a pattern-based static analysis approach [97] that employs regular expressions and various strategies to extract additional information from program texts, including syntax and type information. *CodeQuery* enables us to conduct thorough code searches including symbol references, global definitions, function calls, and reverse function calls. For more details, please refer to our implementation.

Once we have extracted the relevant code snippet corresponding to a bug report, our next step involves constructing a customized question specifically designed to query LLMs for the purpose of validating the bug reports.

3.4 Prompt Engineering

In this section, we describe how Llm4sa harnesses LLMs to validate the code snippet obtained from a bug report by employing prompt engineering. It is worth noting that prior research has shown that the effectiveness of LLMs considerably depends on the way they are prompted to address a particular problem [40]. The process of determining the effective query formulation for a specific task is commonly known as prompt engineering [8, 82].

Llm4sa aims to generate effective prompts for LLMs, thereby guiding them to examine bug warnings and produce reliable and precise answers. Specifically, a prompt in our framework refers to a natural language query containing essential information about the bug warning and code snippets. The prompts typically comprise four key elements: instruction, context, input data, and output indicator, which are depicted in Figure 6, where a prompt is illustrated for inspecting bug warnings in a code snippet. Specifically, Llm4sa creates a Markdown document to construct the prompt. The context is provided in the first line (i.e., “I am an expert C/C++ programmer.”), while the instructions are placed below the “Task Description” section. Besides, the input data is presented in blue text, which is based on templates that can be programmatically populated with specific bug reports and code snippets. Furthermore, the “Task Description” section concludes with a sentence that instructs the desired output format. This output format is critical, as it requires programmatic processing of the LLM’s response.

We introduce two prompt engineering techniques, namely, **Chain-of-Thought (CoT)** [19, 81, 82, 91] and few-shot prompting [8, 66, 87, 93], to construct customized and effective prompts in Llm4sa:

- **Chain-of-Thought.** CoT prompting offers two key advantages as an approach to facilitate reasoning in language models. First, it enables models to break down complex problems into intermediate steps, allocating additional computation to tasks requiring more reasoning steps. Second, a chain of thought provides an interpretable insight into the model’s behavior. It allows us to understand how the model arrived at a specific answer and presents opportunities to debug the reasoning process. To encourage stepwise reasoning, we prompt the language models to “think step by step.” This approach not only helps generate more comprehensive and extended responses but also breaks down intricate problems into manageable steps, allowing for an interpretable view of the code’s behavior. Consequently, we incorporate the CoT strategy into our prompt.
- **Few-shot Prompting.** Few-shot prompting, also referred to as few-shot learning (or in-context learning) with prompt, involves using a limited number of demonstrations that can be accommodated within the model’s context window without allowing any weight updates. This approach allows us to guide LLMs by providing them with a small set of question-answer examples, which helps them generate the desired outputs. By doing so, we can leverage previous knowledge and experiences to effectively handle new and unfamiliar situations. In our specific case, this involves providing code snippets, bug warnings (as questions), and corresponding explanations obtained through manual inspection (as answers). As a result, we incorporate the few-shot prompting strategy into our prompt.

In the workflow of Llm4sa, LLMs are first required to analyze the behavior of the code, present an explanation regarding the presence or absence of a bug, and finally make an appropriate conclusion. By incorporating CoT, LLMs can decompose complex reasoning problems into intermediate steps, facilitating a better understanding of the code’s behavior. In addition, by combining CoT with few-shot prompting, Llm4sa achieves improved results for more intricate tasks that require reasoning prior to generating a response. For example, in the case of inspecting a null pointer dereference bug, Llm4sa provides two code snippets demonstrating null pointer dereference warnings along with their explanations. The LLMs are subsequently prompted to explain a new code snippet that exhibits a similar warning. Through few-shot prompting, LLMs can leverage their knowledge from previous cases and apply it to address novel situations effectively.

3.5 Post-processing

Post-processing plays a crucial role in ensuring the quality and reliability of the answers generated by LLMs. We propose the notion of a confidence level for each bug warning, which represents a numerical score indicating the certainty of the LLMs regarding their answers. This confidence level, ranging from 0 (low confidence) to 1 (high confidence), is derived from the LLMs’ internal probability distribution over the potential answers, reflecting their estimation of uncertainty.

To calculate the confidence level, Llm4sa employs LLMs to generate a set of answers. These answers are then aggregated, and the probability of each answer is computed within the overall answer set. By comparing each answer's probability with a predefined threshold value, we can filter out unreliable or inconsistent answers. In our approach, the threshold is set to 0.7, meaning that only answers with a confidence level of 0.7 or higher are classified as either real bugs or false alarms. Answers below this threshold are labeled as unknown, which ensures that Llm4sa provides reliable and trustworthy answers.

The presence of unknown results can be attributed to two situations: first, when the information within the provided code snippet is insufficient to confirm the presence or absence of a bug, such as missing data structure definitions or calling contexts; second, when LLMs produce inconsistent results. During the post-processing stage, users of Llm4sa have the flexibility to decide whether to prune away the results labeled as "unknown" or include them in the final output, based on their preference for false positives or false negatives. Our experience indicates that adopting a conservative approach by including the unknown results in the final output proves effective, particularly when applying Llm4sa to real-world software. If the objective is to minimize false alarms detected by the static analyzer, then removing all unknown results would be appropriate. However, if the goal is to identify real bugs as precisely as possible, then it is advisable to retain the results marked as unknown for subsequent manual review.

4 IMPLEMENTATION

We implement the prototype of Llm4sa based on *CodeQuery* [63] and OpenAI's API (or Llama-2's API).

Code Extraction. Implementing code extraction faces practical challenges and requires significant engineering efforts. Performing precise pointer analysis and dependency analysis on a large-scale code base is well established as a time-consuming and non-scalable task. In search of a more practical alternative, our solution is based on the observation that developers heavily rely on IDE environments for source code reading. These environments utilize fast scans to assist in locating relevant function definitions. Therefore, we opted to develop our solution utilizing *CodeQuery*, a lightweight implementation that shares similarities with the code scanning analysis employed by IDEs. This part, which involves pre-processing static bug warnings, is implemented in Python and consists of roughly 1,500 lines of code. *CodeQuery* is employed to build a comprehensive code database. This enables us to conduct comprehensive code searches, including symbol references, global definitions, function calls, and reverse function calls. Consequently, managing function call dependencies becomes effortless for us.

Interaction with LLMs. The interaction between Llm4sa and LLMs is facilitated by a basic Python agent, which consists of approximately 900 lines of code. All interactions are fully automated via OpenAI's API. The study utilizes a basic zero-shot prompt, which excludes bug warnings and code snippets, consisting of approximately 500 tokens. It also utilizes six distinct few-shot prompts, each with three examples, addressing bug warning inspection on null pointer dereference, uninitialized variable, use-after-free, divide by zero, memory leak, and buffer overflow, respectively.

Hyper-parameters. There are several hyper-parameters in calling the APIs provided by ChatGPT [39] and Llama-2. For ChatGPT, we set the values of `max_token` and `temperature` to 2,048 and 0.7, respectively. The parameter `max_token` controls the length of the output. Since LLMs predict the next words based on the previous output, longer outputs can provide more comprehensive reasoning. However, using too many tokens can quickly exhaust the context window. Therefore, we chose 2,048 as a balanced option. We also include the result of average prompts tokens in our evaluation. The temperature regulates both randomness and the capacity for reasoning. Ideally, we aim for the analysis to exhibit minimal randomness by decreasing the temperature (ranging from 0 to 2 for GPT models). Nevertheless, excessively low temperatures may lead to repetitive or overly simplistic responses. Following prior work [1, 36, 37, 73], we employed a temperature of 0.7 to adjust the model settings, aiming to generate a wide range of output for the target code snippet and bug warning. For Llama-2, we simply use its default setting.

5 EMPIRICAL EVALUATION

This section presents a comprehensive evaluation of the effectiveness and usefulness of LLM4SA. The experiments conducted aim to answer the following research questions:

RQ1. How effective can LLM4SA be in inspecting different types of static bug warnings in benchmark programs?
RQ2. To what extent can LLM4SA effectively operate alongside static analyzers by automatically inspecting static warnings in real-world software?
RQ3. Can LLM4SA benefit from few-shot prompting?
RQ4. How does LLM4SA compare to other SOTA static warnings scrutinizing methods?
RQ5. What overhead is incurred by LLM4SA in terms of execution time and token cost?

5.1 Evaluation Setup

Static analyzers. We experimentally select three state-of-the-art static analyzers, including CPPCHECK (v2.9), CSA (llvm v12.0.1), and INFER (v1.1.0). The commands for each static analyzer used in our evaluation are shown in Table 10 in the Appendix.

Types of Bugs. Our evaluation examines six representative categories of harmful bugs, including **null pointer dereference (NPD), uninitialized variable (UVA), use-after-free (UAF), divide by zero (DBZ), memory leak (ML)**, and **buffer overflow (BOF)**. As described in Table 2, bug identifiers from different static analysis techniques can be mapped and grouped into distinct categories. We use abbreviations to represent them when presenting the evaluation results.

Benchmark Programs. To understand the capability of LLM4SA in automatically inspecting bug warnings, we evaluate it on both benchmark programs and real-world software. We first evaluate LLM4SA by utilizing a collection of benchmark programs from the Juliet test suite that encompass a diverse range of bugs and for which ground truth data is available. We then use 3 embedded real-time operating systems and 11 well-maintained open-source C/C++ projects to evaluate the bug warning inspection ability of LLM4SA in real-world software.

Large Language Models. All interactions with LLMs, such as sending requests to LLMs or receiving responses from LLMs, are performed via API. The LLama-2 version of our tool, referred to as LLM4SA$_L$, utilizes the *Llama-2-70b* model, which is currently the largest parameter model available in the Llama-2 series. Similarly, the ChatGPT version of our tool employs the *gpt-3.5-turbo-16k-0613* model, which supports the longest input and ensures that the token limit is not exceeded during our evaluation. For comparison, We also include two versions of LLM4SA, one with few-shot prompting (referred to as LLM4SA$_F$) and another with zero-shot prompting (referred to as LLM4SA$_Z$).

5.2 Effectiveness of LLM4SA in Benchmark Programs (RQ1)

The Juliet C/C++ Test Suite[2] released by NIST contains a collection of test cases, which are classified based on MITRE's **Common Weakness Enumeration (CWE)** classification system. This test suite has been widely used to evaluate both static analysis and dynamic testing approaches. Each test case can run as an independent program and contains two variants: a *bad* variant that contains a flaw and a *good* one that does not. All *bad* variants can be used to evaluate the bug

[2]https://samate.nist.gov/SARD/test-suites/112

Table 4. The Results of Bug Warnings Inspection on the *Juliet Test Suites* (ChatGPT)

Bug	**LLM4SA$_Z$ on CPPCHECK's warnings**					**LLM4SA$_Z$ on CSA's warnings**					**LLM4SA$_Z$ on INFER's warnings**				
Type	TP+TN	FP+FN (UK)	Accuracy	Precision	Recall	TP+TN	FP+FN (UK)	Accuracy	Precision	Recall	TP+TN	FP+FN (UK)	Accuracy	Precision	Recall
NPD	158	10 (10)	94.05%	100.00%	94.05%	156	25 (3)	86.19%	87.64%	100.00%	227	34 (2)	86.97%	87.64%	99.13%
UVA	326	12 (1)	96.45%	96.74%	100.00%	279	132 (9)	67.73%	39.34%	99.28%	145	223 (1)	39.24%	69.25%	100.00%
UAF	14	0 (0)	100.00%	100.00%	100.00%	18	0 (0)	100.00%	100.00%	100.00%	–	–	–	–	–
DBZ	35	3 (3)	92.11%	100.00%	92.11%	49	1 (1)	98.00%	100.00%	98.00%	–	–	–	–	–
ML	39	13 (1)	75.00%	76.47%	97.50%	507	187 (3)	73.08%	73.40%	99.80%	–	–	–	–	–
BOF	935	383 (81)	70.94%	84.28%	83.30%	–	–	–	–	–	2,604	389 (245)	87.00%	95.30%	92.59%
All	1,507	421 (96)	78.16%	88.43%	88.22%	1009	345 (16)	74.52%	75.39%	99.60%	2976	646 (248)	82.25%	88.46%	93.41%

*The bug type follows the abbreviations shown in Table 2; the special symbol "–" indicates that this item is not available due to either meaningless or nonexistent data.

detection rate of a tool, while *good* variants can be used to evaluate the false positive rate of a tool. Thus, we have ground truth to evaluate the effectiveness of LLM4SA in these benchmark programs. We analyze each test with CPPCHECK, CSA, and INFER and get a total of 6,904 bug warnings (for a more comprehensive breakdown of the results from the static analysis tools, please refer to Table 11 in the Appendix).

LLM4SA automatically processes each bug warning one-by-one. The bug warnings can be classified by LLM4SA as real bugs, false alarms, or unknown. Table 4 shows the classification results of LLM4SA, where all the unknown results are conservatively considered as wrong answers, that is, **false negatives (FN)** if the unknown warnings are real or **false positives (FP)** if not. Note that the "Accuracy" column measures the accuracy of bug warning classifications by LLM4SA, capturing the proportion of correctly identified real bugs and false alarms. The "Precision" column quantifies the percentage of all real bugs correctly identified by LLM4SA among all positives. The "Recall" column quantifies the percentage of all real bugs correctly identified by LLM4SA among all real bugs.

In total, LLM4SA inspected 6,904 bug warnings and correctly identified 5,492 of them as either real bugs or false alarms. This corresponds to an accuracy rate of 79.5%, surpassing the precision of the static analyzer itself. LLM4SA has also demonstrated its effectiveness in accurately identifying nearly all legitimate bugs, resulting in a high recall rate. While this result is encouraging, it does not imply that LLM4SA can completely replace human inspection of bug warnings. Detailed data analysis reveals that: (1) There is a significant difference in the performance of LLM4SA on static warnings generated by different static analyzers. For example, in the UVA category, LLM4SA achieves precision rates of 96.74%, 39.34%, and 69.25%. (2) LLM4SA can achieve high precision and recall rates on certain bug types, specifically NPD, UAF, and DBZ. However, it performs relatively poorly on ML and BOF bugs, as analyzing long traces or loops is often required in these cases. For example, on static warnings generated by the CPPCHECK, LLM4SA achieves a high accuracy of 94.05% for NPD, while the accuracy for ML is 75%. This also appears on static warnings generated by CSA.

We have also included the number of **unknown results (UK)** in parentheses within the FP+FN column. In some cases, false positives can arise when LLM4SA categorizes a bug warning as "unknown" because it needs additional context for analysis or due to inconsistent results. Thus, in practice, unknown results can be approximated as either real bugs or false alarms, depending on the individual's objective to improve precision or recall rate. If we consider those unknown results as bugs, then the precision in inspecting bug warnings reported by CPPCHECK, CSA, and INFER is up to 81.95%, 74.74%, and 87.30%, while the recall can be up to 92.50%, 99.90%, and 99.39%, respectively.

A similar result is also obtained by running LLM4SA$_L$, which utilizes the open-source large language model *Llama-2*. LLM4SA$_L$ can correctly identify 4,847 bug warnings as either real bugs or false alarms, achieving an accuracy rate of 70.2%, as shown in Table 5. The results demonstrate that LLM4SA can easily be generalized to other popular open-source LLMs, even if this result (70.2%) is lower than the accuracy rate of 79.5% achieved by LLM4SA$_Z$. Note that we can also observe that when examining the warnings provided by CPPCHECK, *Llama-2-70b* achieved a high precision

Table 5. The Results of Bug Warnings Inspection on the *Juliet Test Suites* (LLama-2)

Bug	**LLM4SA$_L$ on CPPCHECK's warnings**					**LLM4SA$_L$ on CSA's warnings**					**LLM4SA$_L$ on INFER's warnings**				
Type	TP+TN	FP+FN (UK)	Accuracy	Precision	Recall	TP+TN	FP+FN (UK)	Accuracy	Precision	Recall	TP+TN	FP+FN (UK)	Accuracy	Precision	Recall
NPD	81	87 (84)	48.21%	100.00%	48.21%	114	67 (15)	62.98%	91.23%	66.67%	147	114 (36)	56.32%	98.43%	54.59%
UVA	305	33 (3)	90.24%	96.21%	93.56%	232	179 (44)	56.45%	74.14%	77.06%	95	273 (73)	25.82%	26.20%	49.31%
UAF	14	0 (0)	100.00%	100.00%	100.00%	18	0 (0)	100.00%	100.00%	100.00%	–	–	–	–	–
DBZ	30	8 (1)	78.95%	100.00%	78.95%	29	21 (3)	58.00%	100.00%	58.00%	–	–	–	–	–
ML	41	11 (0)	78.85%	78.43%	100.00%	501	193 (10)	72.19%	82.51%	79.25%	–	–	–	–	–
BOF	978	340 (43)	74.20%	87.88%	80.89%	–	–	–	–	–	2,274	719 (422)	75.98%	96.51%	77.83%
All	1,437	491 (143)	75.16%	90.29%	90.66%	894	460 (72)	66.03%	81.70%	76.11%	2,516	1,106 (531)	69.46%	89.29%	74.80%

Table 6. Comparison Results of Different Variants of LLM4SA on IoT Embedded OSes

Project Name		**CPPCHECK** (npd,uva,bof,dyz,uaf)	**+ LLM4SA$_Z$** (npd,uva,bof,dyz)	**+ LLM4SA$_F$** (npd,uva,bof,dyz)	**INFER** (npd,uva,bof,dyz)	**+ LLM4SA$_Z$** (npd,uva,bof,dyz)	**+ LLM4SA$_F$** (npd,uva,bof,dyz)
Zephyr	Found bugs	95 (24,48,22,1,0)	62 (18,31,12,1,0)	39 (16,15,7,1,0)	141 (23,118,0,0)	86 (14,72,0,0)	72 (8,64,0,0)
	Real bugs	**2** (1,0,1,0,0)	**2** (1,0,1,0,0)	**2** (1,0,1,0,0)	**1** (1,0,0,0)	**1** (1,0,0,0)	**1** (1,0,0,0)
RIOT	Found bugs	57 (33,10,13,1,0)	21 (10,1,10,0,0)	15 (5,1,9,0,0)	32 (5,10,17,0)	13 (3,1,9,0)	11 (2,0,9,0)
	Real bugs	**2** (2,0,0,0,0)	**2** (2,0,0,0,0)	**2** (2,0,0,0,0)	**1** (1,0,0,0)	**1** (1,0,0,0)	**1** (1,0,0,0)
TencentOS-tiny	Found bugs	424 (110,277,26,9,2)	247 (53,177,10,5,2)	151 (49,77,17,4,2)	–	–	–
	Real bugs	**7** (3,4,0,0)	**7** (3,4,0,0)	**7** (3,4,0,0)	–	–	–
Real bugs/Found bugs (TPR%)		11/576 (1.72%)	11/330 (3.33%) ⇑	11/205 (5.37%) ⇑	2/173 (1.20%)	2/99 (2.00%) ⇑	2/83 (2.40%) ⇑
Precision/Recall of staticAnalyzer +LLM4SA		–	44.62% / 100%	66.32% / 100%	–	43.90% / 100%	53.20% / 100%

*The bug type follows the abbreviations shown in Table 2; The special symbol "–" indicates that this item is not available due to either meaningless or nonexistent data.

rate and recall rate (over 90%), slightly surpassing the performance of *ChatGPT-3.5*. Comparatively, when evaluating the warnings from INFER and CSA, *Llama-2-70b* has lower performance. We anticipate that through the advancement of open-source LLMs and the optimization of prompts, the outcomes of this experiment will be improved.

▶ ***Finding 1:*** *Static analyzers demonstrate an impressive bug detection rate, ranging from 27.67% to 54.63% across various analyzers, while maintaining a low rate of false positives, ranging from 4.08% to 9.76%, in benchmark programs.*
▶ ***Finding 2:*** *The reasoning ability of LLM4SA for different types of bugs is not the same, as described previously.*

Answer to RQ1: LLM4SA demonstrates potent capabilities in automatically inspecting six types of bug warnings that are generated by three static analyzers, showcasing high accuracy, precision, and recall rates.

5.3 Effectiveness of LLM4SA in Real-world Software (RQ2)

In this part, we evaluate the static bug warnings inspection capability of LLM4SA in real-world software.

Target projects. Table 12 provides comprehensive details of all the selected target projects. All of these targets have been extensively studied and are commonly used in the static analysis and dynamic testing community. These projects encompass a diverse range of functionalities, such as binary file analyzers, multimedia file processing, programming language implementations, network packet analyzers, compression algorithms, and so on. The sizes of these projects vary from 12k to 5,647k **source lines of code (SLoC)**, highlighting their extensive diversity.

Results on embedded OSes. Table 6 shows the comparison results of different variants of LLM4SA on three open-source IoT-embedded OSes (`Zephyr`, `RIOT`, and `TencentOS-tiny`). Note that CSA reports many compilation errors when checking IoT-embedded OSes, as their compilation scripts are unsuitable to the Makefiles of IoT OSes. Similarly, Infer reports many compilation errors when checking the `TencentOS-tiny`. Both CPPCHECK and INFER identified a substantial number of potential bugs (a total of 749). Following our comprehensive review, which involved a Ph.D. student who invested six hours in manually inspecting the bug warnings, we determined that only 13 of

Table 7. Comparison Results of Different Variants of Llm4sa on Real-world Applications

Id	Project Name		Cppcheck (n,u_1,b,d,u_2,m)	+Llm4sa$_Z$ (n,u_1,b,d,u_2,m)	+Llm4sa$_F$ (n,u_1,b,d,u_2,m)	Csa (n,u_1,b,d,u_2,m)	+Llm4sa$_Z$ (n,u_1,b,d,u_2,m)	+Llm4sa$_F$ (n,u_1,b,d,u_2,m)	Infer (n,u_1,b,d,u_2,m)	+Llm4sa$_Z$ (n,u_1,b,d,u_2,m)	+Llm4sa$_F$ (n,u_1,b,d,u_2,m)
1	gawk	F	7 (1,2,0,0,0,2)	4 (1,2,0,0,0,1)	6 (1,2,0,0,0,2)	67 (59,0,0,1,6,1)	58 (50,0,0,1,6,1)	30 (26,0,0,1,2,1)	91 (68,23,0,0,0,0)	69 (49,20,0,0,0,0)	16 (14,2,0,0,0,0)
		R	0	0	0	1 (1,0,0,0,0,0)	1 (1,0,0,0,0,0)	1 (1,0,0,0,0,0)	1 (1,0,0,0,0,0)	1 (1,0,0,0,0,0)	1 (1,0,0,0,0,0)
2	tiff	F	4 (1,1,0,0,1,1)	4 (1,1,0,0,1,1)	3 (0,1,0,0,1,1)	56 (6,27,0,17,0,6)	47 (5,21,0,15,0,6)	37 (4,13,0,14,0,6)	30 (13,17,0,0,0,0)	16 (9,7,0,0,0,0)	10 (4,6,0,0,0,0)
		R	0	0	0	0	0	0	0	0	0
3	binutils	F	686 (673,8,5,0,0,0)	279 (272,4,3,0,0,0)	216 (205,7,4,0,0,0)	4 (0,0,0,0,0,4)	4 (0,0,0,0,0,4)	4 (0,0,0,0,0,4)	195 (60,135,0,0,0,0)	121 (41,80,0,0,0,0)	123 (24,99,0,0,0,0)
		R	0	0	0	0	0	0	0	0	0
4	diffutils	F	5 (1,2,0,0,0,2)	3 (1,1,0,0,0,1)	3 (0,1,0,0,0,2)	16 (8,3,0,0,1,4)	15 (8,3,0,0,1,3)	15 (8,2,0,0,1,4)	57 (0,51,6,0,0,0)	35 (0,34,1,0,0,0)	22 (0,20,2,0,0,0)
		R	1 (0,1,0,0,0,0)	1 (0,1,0,0,0,0)	1 (0,1,0,0,0,0)	1 (0,1,0,0,0,0)	1 (0,1,0,0,0,0)	1 (0,1,0,0,0,0)	1 (0,1,0,0,0,0)	1 (0,1,0,0,0,0)	1 (0,1,0,0,0,0)
5	sed	F	6 (3,1,0,0,0,2)	6 (3,1,0,0,0,2)	3 (1,1,0,0,0,1)	12 (1,2,0,0,5,4)	12 (1,2,0,0,5,4)	10 (1,1,0,0,4,4)	10 (7,3,0,0,0,0)	7 (5,2,0,0,0,0)	2 (0,2,0,0,0,0)
		R	0	0	0	0	0	0	0	0	0
6	apr	F	10 (3,1,0,0,0,6)	6 (0,0,0,0,0,6)	6 (0,0,0,0,0,6)	34 (12,18,0,0,2,2)	22 (11,7,0,0,2,2)	10 (6,0,0,0,2,2)	6 (2,4,0,0,0,0)	2 (1,1,0,0,0,0)	3 (2,1,0,0,0,0)
		R	0	0	0	0	0	0	0	0	0
7	bash	F	25 (2,13,4,0,0,6)	15 (2,5,3,0,0,5)	14 (1,5,4,0,0,4)	66 (36,18,0,0,0,9)	47 (23,15,0,0,0,9)	47 (28,10,0,0,0,9)	296 (62,164,70,0,0,0)	211 (44,120,47,0,0,0)	155 (62,36,57,0,0,0)
		R	3 (1,0,0,0,0,2)	3 (1,0,0,0,0,2)	3 (1,0,0,0,0,2)	2 (1,0,0,0,0,1)	2 (1,0,0,0,0,1)	2 (1,0,0,0,0,1)	1 (1,0,0,0,0,0)	1 (1,0,0,0,0,0)	1 (1,0,0,0,0,0)
8	combine	F	20 (6,6,0,0,0,8)	20 (6,6,0,0,0,8)	17 (6,3,0,0,0,8)	13 (0,2,0,0,0,11)	9 (0,2,0,0,0,7)	12 (0,2,0,0,0,10)	36 (10,20,6,0,0,0)	31 (5,20,6,0,0,0)	13 (4,5,4,0,0,0)
		R	5 (0,2,0,0,0,3)	5 (0,2,0,0,0,3)	4 (0,1,0,0,0,3)	5 (0,2,0,0,0,3)	2 (0,1,0,0,0,1)	5 (0,2,0,0,0,3)	1 (0,1,0,0,0,0)	1 (0,1,0,0,0,0)	1 (0,1,0,0,0,0)
9	grep	F	6 (2,2,0,0,0,2)	2 (1,1,0,0,0,0)	4 (0,2,0,0,0,2)	25 (19,2,0,0,0,4)	22 (16,2,0,0,0,4)	18 (13,1,0,0,0,4)	16 (6,10,0,0,0,0)	12 (6,6,0,0,0,0)	3 (1,2,0,0,0,0)
		R	0	0	0	0	0	0	0	0	0
10	m4	F	6 (1,3,0,0,0,2)	4 (1,1,0,0,0,2)	5 (0,3,0,0,0,2)	12 (7,0,0,0,1,4)	8 (3,0,0,0,1,4)	9 (4,0,0,0,1,4)	55 (2,45,8,0,0,0)	44 (2,37,5,0,0,0)	21 (2,16,3,0,0,0)
		R	3 (0,3,0,0,0,0)	1 (0,1,0,0,0,0)	3 (0,3,0,0,0,0)	0	0	0	9 (0,9,0,0,0,0)	3 (0,3,0,0,0,0)	9 (0,9,0,0,0,0)
11	trueprint	F	3 (0,0,0,0,0,3)	3 (0,0,0,0,0,3)	3 (0,0,0,0,0,3)	7 (0,1,0,0,0,6)	6 (0,0,0,0,0,6)	6 (0,0,0,0,0,6)	12 (7,4,1,0,0,0)	8 (5,3,0,0,0,0)	4 (2,1,1,0,0)
		R	3 (0,0,0,0,0,3)	3 (0,0,0,0,0,3)	3 (0,0,0,0,0,3)	3 (0,1,0,0,0,2)	2 (0,0,0,0,0,2)	2 (0,0,0,0,0,2)	1 (0,1,0,0,0,0)	1 (0,1,0,0,0,0)	1 (0,1,0,0,0,0)
R/F (TPR%)			15/778 (1.93%)	13/346 (3.76%) ⇑	14/279 (5.02%) ⇑	12/312 (3.85%)	8/250 (3.20%)	11/198 (5.56%) ⇑	14/804 (1.74%)	8/555 (1.44%)	13/372 (3.49%) ⇑
Precision/Recall			–	56.94% / 86.67%	65.81% / 93.33%	–	21.54% / 66.67%	39.74% / 91.67%	–	31.22% / 57.14%	55.22% / 92.86%

*F means "Found bugs," R means "Real bugs," n means **n**ull-pointer dereference, $u1$ means **u**ninitialized variable, b means **b**uffer overflow, d means **d**ivide by zero, $u2$ means **u**se-after-free, m means **m**emory leak.

them were real bugs. Specifically, 11 real bugs were found by Cppcheck with a precision of 1.72%, and another two bugs were found by Infer with a precision of 1.20%, respectively.

We utilize Llm4sa to filter the bug warnings produced by static analyzers. If Llm4sa identifies the bug warnings as false alarms, then we exclude them from the reported bugs. Otherwise, they will be included in the reported bugs for further manual inspection. For instance, Cppcheck reports a total of 576 bug warnings across these three operating systems. Llm4saіs able to deduce that 371 of these warnings are false alarms, leaving only 205 warnings that are likely to be real bugs. This significantly reduces the need to inspect the remaining warnings manually, thus minimizing human labor costs. Importantly, it should be noted that the automatic inspection process does not miss any real bugs, resulting in a 100% recall rate for those warnings produced by Cppcheck and Infer. Hence, Llm4sa is highly effective and valuable in its ability to inspect static warnings in real-world software.

Results on real-world applications. Table 7 shows the comparison results of different variants of Llm4sa on 11 real-world applications. In contrast to the three previously mentioned embedded **operating systems (OSes)**, which may not be able to undergo checks with the Infer and Csa tools due to numerous compilation errors, these 11 applications can be successfully analyzed by all three static analyzers through appropriate modifications to the Makefile scripts. The three static analyzers produced a total of 1,894 bug warnings, requiring an investment of 8 hours from a Ph.D. student to confirm the existence of real bugs. From the table, we can observe that: (1) The bug detection rates of the three static analysis tools are relatively low. Cppcheck, Csa, and Infer all reported a large number of warnings, with Infer reporting the most at 804. However, only 1.74% (i.e., 14 warnings) of the reports are real bugs. (2) Llm4sa has the ability to reduce the number of warnings that necessitate manual inspection significantly. On average, Llm4sa$_Z$ reduces the number of warnings by 38.28%, while Llm4sa$_F$ reduces the number of bug warnings by 53.73%. Notably, for the warnings generated by Cppcheck, Llm4sa$_F$ filters out 64.14% of the warnings. (3) Llm4sa$_F$ achieves a high recall rate of more than 90% across three static analyzers. However, this result, although encouraging, does not necessarily imply that Llm4sa$_F$ is effective. The cost of missing real bugs could exceed the expense of inspecting bug warnings with human labor, especially if these real bugs result in serious issues. In summary, Llm4sa has consistently shown practicality in inspecting static warnings across various functionalities and levels of complexity in real-world applications.

Table 8. Zero-shot Prompting vs. Few-shot Prompting of LLM4SA on the *Juliet Test Suite*

Bug Type	Bug Numbers	LLM4SA$_Z$							LLM4SA$_F$						
		TP+TN	FP+FN(UK)	Accuracy	Precision	Recall	Average tokens	Average time (s)	TP+TN	FP+FN (UK)	Accuracy	Precision	Recall	Average tokens	Average time (s)
NPD	610	541	69 (15)	88.69%	90.92%	97.83%	3,107	26.97	551	59 (12)	90.33%	97.15%	92.59%	6,527	7.68
UVA	1,117	750	367 (11)	67.14%	67.75%	99.73%	3,542	21.09	1,014	103 (45)	90.78%	98.83%	90.39%	4,721	7.92
UAF	32	32	0	100.00%	100.00%	100.00%	3,636	18.64	32	0	100.00%	100.00%	100.00%	5,148	7.92
DBZ	88	84	4 (0)	95.45%	100.00%	95.45%	3,277	21.44	85	3 (0)	96.59%	100.00%	96.59%	5,354	10.05
ML	746	546	200 (4)	73.19%	73.61%	99.63%	3,540	20.79	638	108 (1)	85.52%	97.74%	94.87%	7,486	12.04
BOF	4,311	3,539	772 (326)	82.09%	92.06%	89.89%	3,536	34.72	3,868	443 (209)	89.72%	95.52%	95.65%	6,541	15.96
All	6,904	5,492	1,412 (360)	79.55%	85.70%	95.10%	3,508	29.06	6,188	716 (265)	89.63%	96.36%	94.64%	5,975	10.25

► ***Finding 3:*** *Static analyzers demonstrate a low rate of bug detection, typically ranging from 1.2% to 3.85% across different analyzers while producing a substantial number of false alarms in real-world software that are extremely larger than in benchmark programs.*

Answer to RQ2: LLM4SA significantly improves the precision of static analyzers by reducing nearly half of the false alarms, all the while upholding high recall rates.

5.4 Improvement through Prompt Engineering (RQ3)

Table 6 and Table 7 provide a comparative analysis between two versions of LLM4SA in real-world software: LLM4SA$_F$, which utilizes few-shot prompting, and LLM4SA$_Z$, which employs zero-shot prompting. Both LLM4SA$_Z$ and LLM4SA$_F$ achieve a recall rate of 100% in the bug warnings of embedded OSes. Moreover, LLM4SA$_F$ significantly improves the precision of LLM4SA$_Z$ across all three static analyzers. LLM4SA$_F$ also demonstrates significant improvement over LLM4SA$_Z$ regarding both precision and recall across 11 real-world applications. LLM4SA$_Z$ improved the true positive rate from 1.93% to 3.76% on reports generated by the CPPCHECK tool. In contrast, LLM4SA$_F$ enhances the analysis precision of all three static analysis tools with six different few-shot prompts. In addition, LLM4SA significantly reduces the number of warnings that require manual inspection. LLM4SA$_Z$ reduces the number of warnings by an average of 38.28%, and LLM4SA$_F$ reduces the number of bug warnings by 53.73%. Notably, LLM4SA$_Z$ filters out 53.21% of the warnings generated by CPPCHECK, while LLM4SA$_F$ filters out 64.14%. Table 8 presents the experimental results of LLM4SA$_Z$ and LLM4SA$_F$ on the Juliet Test Suites, respectively. LLM4SA$_F$ can still improve the precision of LLM4SA$_Z$ while maintaining a high recall rate. Based on the above analysis, we can positively conclude that LLM4SA can greatly benefit from few-shot prompting.

Answer to RQ3: LLM4SA can benefit from few-shot prompting, as it significantly improves the precision of LLM4SA while maintaining a high recall rate compared to zero-shot prompting.

5.5 Compared with SOTA (RQ4)

Table 9 gives a comprehensive comparison of LLM4SA with three **state-of-the-art (SOTA)** techniques. Note that GPT-C [38] and DSE [10] are not publicly available, and Helium [33], which is a dynamic approach, requires 15 minutes to confirm each warning, which is difficult to scale to a large number of warnings. Therefore, we directly compare these techniques with LLM4SA in five key aspects using the data statistics provided in their papers.

Techniques. DSE [10] utilizes error traces generated by static analyzers to guide the search process of dynamic symbolic execution for efficient bug confirmation. Helium [33] generates semantically equivalent executable code fragments and verifies if the corresponding bug warnings can be triggered using established testing tools. In contrast to these two dynamic methods, LLM4SA accomplishes code inspection by utilizing LLMs equipped with natural language processing and

Table 9. Compared Llm4sa with SOTA

SOTA	**LLM4SA**	**GPT-C (ICSE'22)**	**DSE (ISSTA'22)**	**Helium (ISSTA'21)**
Techniques	LLMs-powered	Machine learning	Dynamic symbolic	Testing Code Fragments
Benchmarks	Juliet Test Suite + 3 embedded OSes + 11 real-world applications	5 open-source projects + 2 proprietary projects	10 applications from GNU Coreutils Suite (synthetic bugs)	12 real-world projects
Evaluated Warnings	6,904 + 749 + 1,894	539 + 108	55	1,955
Precision	Llm4sa$_Z$: 69.09% Llm4sa$_F$: 81.13%	RL: 60.6%, NPD: 85.4%	–	83%
Recall	Llm4sa$_Z$: 92.82% Llm4sa$_F$: 94.64%	RL: 64.5%, NPD: 83.7%	–	unknown (only successfully built 68% code fragments)
Time (s)	<30 s/warning	–	9.15 min/warning	16.8 s/warning (unit test generation) 15 min/warning (dynamic testing)
Tool Accessibility	✓	✗	✓	✗

*RL means "**R**esource **L**eak warnings," NPD means "**N**ull-**P**ointer **D**eference warnings."

code comprehension capabilities. This enables it to easily apply to real-world software, while also enhancing its efficiency in handling thousands of bug warnings. The GPT-C [38] utilizes both feature-based and Transformer-based learning approaches to identify false alarms. However, this approach primarily emphasizes model training and has limited analytical capabilities due to the utilization of a small-scale dataset that only consists of null-pointer deference and memory leak reports. In contrast, Llm4sa focuses on harnessing the powerful capabilities of LLMs to facilitate bug report inspection. It achieves this by combining code extraction, prompt engineering, and pre-/post-processing. This approach provides greater flexibility in expanding the inspection capabilities to different types of bug warnings.

Benchmarks & Evaluated Warnings. The benchmarks utilized by Llm4sa offer a more comprehensive and in-depth evaluation compared to GPT-C, DSE, and Helium. We conduct a thorough evaluation of Llm4sa on a total number of 9,547 warnings across six types of bugs from the three major categories of datasets (i.e., the Juliet test suite with ground truth data available, 3 embedded real-time operating system, and 11 well-maintained open-source C/C++ projects). DSE [10] failed to apply to all the real-world projects they have evaluated (among which, 9 projects were also used to evaluate Llm4sa). Specifically, DSE only analyzed 55 injected synthetic bugs of two types (null-pointer dereference and use-after-free) from the GNU Coreutils Suite. However, these injected synthetic bugs are not publicly available. GPT-C [38] examined 647 warnings across two bug types (null-pointer dereference and resource leak) within 7 proprietary projects. Comparatively, Llm4sa can be easily extended to support other types of bugs, such as use-after-free, buffer overflow, and so on. Helium [33] analyzed 1,955 bug warnings from 12 projects, whose focus was on dynamic checking, specifically excluding the warnings that were not yet supported by KLEE, Valgrind, or their respective assertions. Comparatively, Llm4sa is a static and LLM-empowered method that can analyze warnings without the need for environment setup and runtime execution. In addition, Llm4sa can provide support for a wide range of warning types.

Precision & Recall. The analysis precision of Llm4sa shows substantial variations across different benchmarks. For instance, its variant Llm4sa$_F$ achieves an average precision of 81.13% on 9,547 bug warnings. For instance, when considering the ML and NPD bug warnings, Llm4sa attained a minimum precision of 73.40% and 87.64% respectively. These values are slightly higher compared to the 60.60% (RL) and 88.46% (NPD) achieved by the GPT-C tool. This difference can be attributed to the variation in evaluation datasets. However, it is important to note that our evaluation dataset is more diverse and comprehensive, and we obtained these results based on a large quantity of bug warnings. Additionally, it can be observed from Table 9 that the recall rate of Llm4sa is notably higher than that of GPT-C. Excluding the lowest recall rate of 88.22% on the Juliet test suite benchmark, Llm4sa$_F$ consistently achieves recall rates of 94.64% on all other benchmarks.

Time. Regarding the analysis time for each warning, DSE takes an average of 9.15 minutes to identify each injected synthetic bug, while Helium requires 16.8 seconds for generating unit test cases and 15 minutes for dynamic testing. Compared to approaches that typically require a significant amount of time to solve path constraints to generate valid inputs, LLM4SA demonstrates high efficiency in reasoning, interpreting, and drawing conclusions for bug warnings. On average, it analyzes each warning in less than 30 seconds.

Tool Accessibility. We have made the implementations of LLM4SA and associated experimental data publicly available, except for the querying of LLMs, which is currently done via ChatGPT's API call. This design choice allows for easy integration with other out-of-the-box LLMs. Among the three SOTA techniques that we compared, only DSE has released their implementations, enabling comparisons between different tools. However, utilizing DSE requires performing instrumentation based on CSA's error trace and employing KLEE for symbolic execution. This approach is not suitable for real-world applications, which is why DSE was only evaluated on 55 injected synthetic bugs. In contrast, GPT-C and Helium have not released their implementation.

Answer to RQ4: LLM4SA outperforms the state-of-the-art techniques in terms of the comprehensiveness and diversity of evaluation benchmarks, the recall rate of analysis results, as well as the time cost required for inspecting bug warnings.

5.6 Overhead Analysis (RQ5)

Time. On average, LLM4SA is capable of automatically inspecting bug warnings in less than 30 seconds per warning, as shown in Table 8, which is substantially faster than a human engineer. Specifically, the code extraction and running scripts can be completed in a few seconds. Since LLM4SA queries LLMs for every bug warning and completion in a single round, the query time remains relatively constant. In our experiments, the exact time for querying LLMs may vary, depending on the workload of the OpenAI infrastructure.

Cost. The current cost of the *gpt-3.5-turbo-16k-0613* model is \$0.003 per 1,000 tokens for prompts and \$0.004 per 1,000 tokens for outputs. Note that the number of prompt tokens in the zero-shot version average consumes 4,200 tokens (LLM4SA_Z ranges from 1,609 to 4,560 tokens) in our evaluation, while in the few-shot version LLM4SA_F, it average consumes 7,800 tokens (ranged from 4,560 to 14,000 tokens). We set the maximum number of output tokens for LLMs to 2,048 in response. On average, the cost of analyzing each bug warning is \$0.2 for LLM4SA_Z and \$0.31 for LLM4SA_F.

Answer to RQ5: LLM4SA is characterized by its speed, as it averages less than 30 seconds per warning, and affordability, costing only \$0.31 per warning.

5.7 Case Studies

In this section, we pick three interesting cases demonstrating the effectiveness of LLM4SA in inspecting bug warnings. All of these cases are uninitialized value bugs that were reported as bugs by the static analyzers, but for various reasons.

Insensitive to mutex condition. Figure 7 shows a code snippet from the *sed* application, a non-interactive command-line text editor. The code implements a function for getting a line from a stream. INFER and CSA, two static analyzers, report uninitialized variable access false alarms at line `273` in the `ck_getline` function. They make this error because they assume that the two mutex conditions at line `267` and line `270` are both false. However, LLMs can correctly determine that the variable `result` is only assigned a value when the `stream` has no errors, and remains uninitialized otherwise. This is possible because LLMs can analyze the code context under mutually exclusive

```
// sed-4.2/sed/utils.c:261-274
261: size_t ck_getline(char **text, size_t *buflen, FILE *stream) {
...
266:   int result;
267:   if (!ferror(stream))
268:     result = getline(text, buflen, stream);
270:   if (ferror(stream))
271:     panic(_("read error on %s: %s"), utils_fp_name(stream), strerror(errno));
273:   return result;
274: }
```

Fig. 7. An *uninitialized value* false alarm in *sed-4.2*.

```
// apr-1.5.2/strings/apr_strings.c:123-175
123: char *apr_pstrcat(apr_pool_t *a, ...) {
125:     char *cp, *argp, *res;
126:     size_t saved_lengths[MAX_SAVED_LENGTHS];
127:     int nargs = 0;
...
129:    /* Pass one --- find length of required string */
136:     while ((cp = va_arg(adummy, char *)) != NULL) {
137:         size_t cplen = strlen(cp);
138:         if (nargs < MAX_SAVED_LENGTHS)
139:             saved_lengths[nargs++] = cplen;
...
140:     }
...
151:    /* Pass two --- copy the argument strings into the result space */
155:     nargs = 0;
156:     while ((argp = va_arg(adummy, char *)) != NULL) {
157:         if (nargs < MAX_SAVED_LENGTHS)
158:             len = saved_lengths[nargs++];
...
166:     }
175: }
```

Fig. 8. An *uninitialized value* false alarm in *apr-1.5.2*.

conditions. This demonstrates that LLMs have strong contextual summarization, analysis, and reasoning abilities.

Imprecise analysis in path condition. Figure 8 shows a code snippet from the apr project, which creates and maintains software libraries. The code implements a function for finding the length of the required string from inputs and then copying the argument strings into the result space. The array `saved_lengths` is not initialized at line 126, which causes an uninitialized variable access error at line 158 in the `apr_pstrcat` function. This false alarm is reported by Infer, Csa, and other mainstream static analyzers, because they assume that the condition at line 136 is false. However, this assumption is incorrect, as the conditional statements in lines 136∼138 and 156∼157 are identical, except for the assigned variables. This means that if line 158 is reachable, then line 139, which initializes the array `saved_lengths`, is also reachable. Unlike mainstream static analysis tools, which only construct counterexamples based on the control flow of a program in imprecise contexts, LLMs can leverage the code context, the comments from lines 129 and 151, and the domain knowledge about the semantics of functions like `va_arg`. This enables LLMs to divide the function `apr_pstrcat` into two phases, namely, "Pass one" and "Pass two," and analyze them separately. As a result, it can verify that the array `saved_lengths` has been correctly initialized in the "Pass one" phase before its usage.

Difficulty in handling complex code. Figure 9 shows a code snippet from bash, a popular Unix shell and **command-line interface (CLI)** program. The function `glob_vector` is reported to have an uninitialized variable access false alarm at line 748. However, this is incorrect, as the variable `isdir` is initialized at line 738, which is always executed before line 748 according to the code in lines 732 and 746. The function `glob_vector` is very long and complex, with over 360 lines of code, numerous conditional statements, and bitwise operations. As a result, LLMs face difficulty in comprehending and reasoning about the pertinent code, since the zero-shot prompting query would exceed 3,800 tokens. Furthermore, the complex program control flow hinders LLMs from

```
// bash-4.3/lib/glob/glob.c: 552-912
552: char ** glob_vector(pat, dir, flags) char *pat; char *dir; int flags; {
...
564:  int isdir;
...
732:         if (flags & (GX_MATCHDIRS | GX_ALLDIRS)) {
738:           isdir = glob_testdir (subdir, flags);
...
744:         }
746:         if (flags & GX_ALLDIRS) {
748:           if (isdir == 0)
...
787:         }
912: }
```

Fig. 9. An *uninitialized value* false alarm in *bash-4.3*.

comprehending the crucial aspects of the code, consequently impeding their ability to generate precise inspection results.

▶ ***Finding 4:*** *False positives generated by static analysis can occur due to multiple reasons. LLMs are effective in addressing issues such as their insensitivity to mutex conditions and imprecise analysis in path conditions, However, they still lack the ability to handle long and complex program control flow.*

6 DISCUSSION

6.1 Threats to Validity

Our evaluation can be subject to several validity threats.

Threats to Internal Validity. One possible threat could be data leakage. Data leakage refers to the problem that an evaluation is conducted on a dataset that has been used in the training dataset of LLMs. This can lead to overfitting and introduce bias in the evaluation results. We believe that the data leakage threat in our case should not be a significant concern. Specifically, bug warnings generated by different static analysis tools suffer from low quality and variations, rendering them unsuitable for training LLMs. Our research also tackles the data leakage issue from four distinct perspectives: (1) We have evaluated LLM4SA on two different LLMs (ChatGPT and Llama-2); (2) We have used a diverse range of benchmarks in our evaluation; (3) We have evaluated bug warnings from three different static analyzers; (4) Our specific template format makes it highly unlikely that the prompts we generated exist word-for-word within the training data of LLMs. By doing so, we have, to some extent, mitigated the data leakage concerns. We also plan to evaluate LLM4SA on more open-source or closed-source commercial software in our future work.

Threats to External Validity. LLMs generate varied answers for identical code snippets and bug reports with the same prompts due to their inherent randomness. One threat to the validity of our study is that conclusions drawn from random results may be misleading. To mitigate this threat, On one hand, we calculate the confidence level and perform post-processing to ensure output quality. On the other hand, we conduct our experiments on a large dataset that includes both benchmarks and real-world software, thereby mitigating the influence of randomness to some extent.

Threats to Construct Validity. LLMs are currently under active development. The LLMs we selected to evaluate are based on the GPT-3.5 models, a closed-source API subject to frequent updates. The reproducibility of LLM4SA might be potentially compromised due to the future deprecation of the model, which could lead to the inability to reproduce the presented evaluation results. We believe that LLM4SA's improvement over existing approaches is not limited to specific LLMs. The results presented in this article may have underestimated the potential, as they could be further enhanced by more powerful LLMs. Therefore, the results and findings of our article can still serve as valuable references, even in the event of *gpt-3.5-turbo-16k-1613* being deprecated. In future studies, we plan to investigate other promising LLMs and apply fine-tuning techniques to enhance their performance.

6.2 Limitations and Future Directions

Static bug warning inspection via large LLMs is still a rich research field. Our approach reveals some limitations, indicating the potential for further advancements in automated bug warning inspection.

Fine-tuning. With current LLMs typically embodying millions of parameters and further expected to grow in scale, it is crucial to gather a large amount of diverse training data to retrain LLMs for specific tasks [78]. Our study employs representative and widely recognized examples for prompt engineering instead of relying on large datasets for fine-tuning. This choice is influenced by the significant costs associated with analyzing extensive data for fine-tuning and analysis purposes. Furthermore, the challenge of addressing the insufficiency of training datasets for retraining LLMs and improving their capability in static warning inspection persists. Our future research will revolve around this topic.

Combining with dynamic analysis. While LLMs exhibit promise in precisely inspecting bug warnings, there is a slight decrease in recall rate, resulting in the potential omission of real bugs. We believe that combining LLMs with traditional dynamic analysis methods can help mitigate this recall reduction. Dynamic analysis methods, such as DSE [10] and Helium [34], can theoretically provide a soundness guarantee. Using an automatic dynamic analysis method is a possible approach to validate bug warnings. Then, Llm4sa can be employed on these undecided warnings. This advantage of using Llm4sa is easily extendable and complementary to dynamic analysis.

Code extraction filled with enough context. Our code snippet extraction scheme prioritizes practicality and efficiency by sacrificing precise data dependency analysis. This tradeoff is made to prevent performance penalties while accepting a certain degree of precision loss. This can make it challenging to accurately manage dependencies in code fragments that involve lengthy paths and complex function calls, resulting in code snippets that lack sufficient context. As a result, this difficulty in making decisions based on insufficient code snippets can lead to unknown outcomes when using LLMs. In addition to utilizing more precise dependency analysis techniques, another viable solution is to implement an iterative process. This process enables LLMs to request supplementary information, such as function definitions.

Deploying on Open-source LLMs. Because of limited access to ChatGPT weights, we employ a black-box analysis approach utilizing LLMs, like recent studies conducted on "LLMs for software engineering." At the time of writing, Meta introduced Llama 2 [22]. Additionally, other open-source language models such as ChatGLM and Alpaca have also been developed, and these models may possess capabilities that rival GPT-3.5 in certain scenarios. Based on our initial assessments, Llama 2 demonstrates an understanding of our prompts and appears suitable for supporting Llm4sa. Moreover, the open-source nature of Llama 2 offers us opportunities to deploy further and refine the model. Our future studies will involve leveraging these opportunities.

7 RELATED WORK

Static Bug Finding. Static bug finding has achieved significant success in both industry and academia over the past few decades. Existing static analysis techniques can be generally categorized into data flow analysis [3, 28, 62, 65, 69, 70], value flow analysis [14, 23, 67, 71, 72, 90], symbolic execution [41], model checking [42], and pattern matching [57, 86, 97]. For instance, many data-flow analyses are embodied into the **IFDS (Interprocedural, Finite, Distributive, Subset)** framework, which has a wide range of application areas, including bug detection [51], security analysis [56], and taint analysis [3, 88]. However, data-flow analyses generally have performance issues due to the dense data flow propagation along control flows [18, 27]. Most recently, sparse value flow analysis [14, 23, 68, 72, 79] has been proposed to improve the performance, which tracks how the values of variables are propagated efficiently along data dependence. Despite the seminal progress, efficient static analysis is still imprecise, where thousands of analysis reports can be generated when dealing with large codebases. Therefore, we believe that Llm4sa can enhance the effectiveness and precision of these static analysis techniques as a cheap and fully automatic post-processing step.

Notably, recent studies [5, 43] have investigated under-approximate static bug analysis. These approaches embrace the generation of high-confidence bug reports at the expense of potentially missing some bugs. For instance, RacerD [5] favors high-confidence data races to prevent developers from disregarding static analysis reports due to excessive false warnings. Similarly, Pulse-X [43] employs under-approximate abstractions of Incorrectness Separation Logic to uncover high-confidence bugs. In contrast, our work focuses on improving the precision of over-approximate static bug-finding tools. While these tools may lack precision, they offer higher code coverage and the ability to uncover more bugs. It is shown that these two merits have been a significant driving force for developers to prefer static analysis over dynamic analysis and have led to the widespread adoption of over-approximate static analysis in the industry [4, 53].

Static Warning Validating. Several research works have tried to perform automatic static warning validating by using different methods and features to identify and reduce false positives [10, 33, 35, 38, 54, 80]. Muske et al. [55] reviewed 130 studies on postprocessing of alarms generated by static analysis tools and categorized them into six main approaches (i.e., *clustering*, *ranking*, *pruning*, *automated elimination of false positives*, *combination of static and dynamic analyses*, and *simplification of manual inspection*), which provide guidelines and directions for users and researchers. Joshy et al. [33] proposed a method to validate static warnings by conducting dynamic unit testing on code fragments that encapsulate the reported buggy paths. This method is effective but not scalable due to the challenging and costly nature of constructing code fragments that can be compiled and executed. Busse et al. [10] propose converting potential bug reports into concrete test inputs that can trigger the bugs, using a form of dynamic symbolic execution that explores paths compatible with the trace obtained from the static analyzer. However, the authors were unable to find real-world bugs for evaluating the proposed technique, so they only evaluated it by injecting faults into applications, resulting in negative results. Kharkar et al. [38] augmented static analyzers with a variety of machine learning models, including both feature-based model and Transformer-based neural models, to identify false positive bug warnings. However, they primarily focus on training various machine learning models, and their applicability is constrained by the small-scale dataset comprising NPD and memory leak, thereby restricting its generalizability.

Assisting Program Analysis with LLMs. In recent years, there has been a growing interest in applying LLMs to assist program analysis tasks, such as software testing [20, 45, 73, 76], static analysis [6, 74], bug reproduction [36, 37], and bug repair [1, 24, 59, 85]. Ye et al. [92] use LLMs to generate interrupt specifications for interrupt-based deadlock detection. Li et al. [46] investigated the potential of LLMs in enhancing static analysis by posing relevant queries. They specifically focused on UBITest [95], a bug-finding tool for detecting use-before-initialization bugs. The study revealed that those false positives can be significantly reduced by asking precisely crafted questions related to function-level behaviors or summaries. In contrast, Llm4sa incorporates LLMs directly for various bug report inspection tasks and successfully integrates them into a static analysis pipeline that supports multiple static analyzers. Ma et al. [49] and Sun et al. [74] explore the capabilities of LLMs when performing various program analysis tasks such as control flow graph construction, call graph analysis [12, 16], and code summarization. Pei et al. [60] use LLMs to reason about program invariants with decent performance. These diverse applications underline the vast potential of LLMs in program analysis. Llm4sa complements these efforts by demonstrating the efficacy of LLMs in understanding and automatically inspecting thousands of static analysis reports in the real world.

8 CONCLUSION

In this article, we have studied the problem of excessive false positives in static analysis tools, hindering the effectiveness of practical adoption. Specifically, we have described the causes and characteristics of the false positives reproduced by four popular open-source static analysis tools and presented the studies of representative cases to illustrate the limitations of these tools. To address this problem, we have proposed a novel platform called Llm4sa that harnesses **Large Language Models (LLMs)** to sift through numerous static warnings automatically, thereby significantly saving valuable developer time. Our solution can construct related code snippets based on the calling context and design questions with many typical case studies to query LLMs, which can provide accurate and reliable answers for static warnings. We evaluated our solution on a large set of static warnings from Juliet benchmark programs and 11 real-world C++ projects, demonstrating that our solution can automatically identify bug warnings into real bugs or false alarms with a high precision and recall rate. Our work opens up new possibilities for enhancing static analysis tools with the power of LLMs. It provides valuable insights for developers and researchers interested in applying LLMs to static program analysis.

A APPENDIX

A.1 Data Availability

We provide a reproduction package with a unique DOI to facilitate future research. The package includes (1) an available tool; (2) detailed information on the evaluation dataset; (3) bug warning inspection results; (4) experimental scripts. A standard X86 Linux machine with Ubuntu 18.04 LTS or a newer operating system is necessary to evaluate this artifact.

A.1.1 Artifact Check-list (Meta-infomation).

— *Dataset:* A list of download addresses for evaluation datasets with a specific version or commit ID.
— *Runtime environment:* Linux.
— *Hardware:* X86.
— *How much disk space is required (approximately)?:* 500 MB for our uploaded package, and an additional 30 GB for the static analyzers and evaluation dataset.
— *Publicly available?:* Yes.
— *Code licenses (if publicly available)?:* MIT.
— *Archived (provide DOI)?:* Yes.

A.1.2 Description.

— *How to access:* The artifact can be downloaded from the following link: https://doi.org/10.5281/zenodo.8346515
— *Hardware dependencies:* A standard X86 machine.
— *Software dependencies:*
  – *Packages:* cscope codequery libjansson-dev libjansson4 wget autoconf automake pkg-config cmake unzip tzdata libncurses5
  – *Python:* xmltodict openai.

A.1.3 Installation.

— Please refer to the `README.md` file in the artifact.

A.2 Additional Experimental Information

We attach some additional experimental information related to the article here.

Table 10. Commands for Static Analyzer Used in Our Evaluation

Static Analyzer	Executed commands
Cppcheck	cppcheck −enable=warning $project_folder (./) −output-file=cppcheck_err.xml −xml −force
Csa	scan-build −plist -o report $build_command (make/cmake/...)
Infer	infer −keep-going −biabduction −bufferoverrun −liveness −quandary −siof −uninit run − project $build_command (make/cmake/...)

Table 11. Bug Warnings Reported by Different Static Analyzers on the *Juliet Test Suite*

Bug Type	**Cppcheck**				**Csa**				**Infer**			
	TP	FP	TPR (%)	FPR (%)	TP	FP	TPR (%)	FPR (%)	TP	FP	TPR (%)	FPR (%)
NPD	168	0	84.00%	0.00%	156	25	87.15%	13.97%	229	32	90.51%	12.03%
UVA	326	12	50.00%	3.06%	279	132	72.47%	32.27%	144	224	90.00%	86.82%
UAF	14	0	93.33%	0.00%	18	0	45.00%	0.00%	0	0	0.00%	0.00%
DBZ	38	0	52.78%	0.00%	50	0	12.66%	0.00%	0	0	0.00%	0.00%
ML	40	12	16.60%	1.18%	506	188	79.81%	27.01%	0	0	0.00%	0.00%
BOF	1,120	198	24.32%	5.23%	0	0	0.00%	0.00%	2,738	255	64.82%	6.44%
All	1,706	222	27.67%	4.08%	1,009	345	29.20%	9.76%	3,111	511	54.63%	9.22%

*The bug type follows the abbreviations shown in Table 2. TP, FP, TPR, and FPR are abbreviations of true positive, false positive, true positive rate, and false positive rate, respectively. The false positive rate is of incorrect reports (or false alarms) out of all reports produced by a tool.

Table 12. Details of Selected Target Projects

Id	Projcet Name	Type	Version	SLoC	Description
01	Zephyr	Embedded OS	2.1.0	493k	A scalable real-time operating system supporting multiple hardware architectures.
02	RIOT	Embedded OS	2020.04	1,689k	A real-time multi-threading operating system that supports a range of IoT devices.
03	TencentOS-tiny	Embedded OS	23313e	5,674k	A real-time IoT operating system developed by Tencent.
04	gawk	Applications	4.1.2	81k	A special-purpose programming language that handles data-reformatting jobs.
05	tiff	Applications	3.9.7	77k	This software provides support for the Tag Image File Format (TIFF).
06	binutils	Applications	2.25.1	1,988k	The GNU Binary Utilities, or binutils, are a collection of binary tools.
07	diffutils	Applications	3.3	95k	A package of several programs related to finding differences between files.
08	sed	Applications	4.2	35k	A non-interactive command-line text editor
09	apr	Applications	1.5.2	89k	A project to create and maintain software libraries.
10	bash	Applications	4.3	152k	Bash is the GNU Project's shell—the Bourne Again SHell.
11	combine	Applications	0.4.0	36k	A tool for working with record-oriented data files.
12	grep	Applications	2.21	101k	Grep searches one or more input files for lines containing a match to a specified pattern.
13	m4	Applications	1.4.17	115k	GNU M4 is an implementation of the traditional Unix macro processor.
14	trueprint	Applications	5.4	12k	GNU Trueprint takes C source files and other text files and prints them on postscript printers.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] Toufique Ahmed and Premkumar Devanbu. 2023. Better patching using LLM prompting, via self-consistency. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE'23)*. IEEE, 1742–1746.

[2] John R Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 177–189.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, NY, 259–269. DOI: https://doi.org/10.1145/2594291.2594299

[4] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75. DOI: https://doi.org/10.1145/1646353.1646374

[5] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional static race detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (Oct. 2018), 28 pages. DOI: https://doi.org/10.1145/3276514

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2329–2344. DOI: https://doi.org/10.1145/3133956.3134020

[7] James Brotherston, Paul Brunet, Nikos Gorogiannis, and Max I. Kanovich. 2021. A compositional deadlock detector for Android Java. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*. IEEE, 955–966. DOI: https://doi.org/10.1109/ASE51524.2021.9678572

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS'20)*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). Retrieved from https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[9] Frank Busse, Pritam Gharat, Cristian Cadar, and Alastair F. Donaldson. 2022. Combining static analysis error traces with dynamic symbolic execution (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22)*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 568–579. DOI: https://doi.org/10.1145/3533767.3534384

[10] Frank Busse, Pritam Gharat, Cristian Cadar, and Alastair F. Donaldson. 2022. Combining static analysis error traces with dynamic symbolic execution (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 568–579. DOI: https://doi.org/10.1145/3533767.3534384

[11] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. Retrieved from http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[12] Yuandao Cai, Yibo Jin, and Charles Zhang. 2024. Unleashing the power of type-based call graph construction by using regional pointer information. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security'24)*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. Retrieved from https://www.usenix.org/system/files/sec23winter-prepub-350-cai.pdf

[13] Yuandao Cai, Peisen Yao, Chengfeng Ye, and Charles Zhang. 2023. Place your locks well: Understanding and detecting lock misuse bugs. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security'23)*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association. Retrieved from https://www.usenix.org/conference/usenixsecurity23/presentation/cai-yuandao

[14] Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: Practical static detection of inter-thread value-flow bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1126–1140. DOI: https://doi.org/10.1145/3453483.3454099

[15] Yuandao Cai, Chengfeng Ye, Qingkai Shi, and Charles Zhang. 2022. Peahen: Fast and precise static deadlock detection via context reduction. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22)*. Association for Computing Machinery, New York, NY, 784–796. DOI: https://doi.org/10.1145/3540250.3549110

[16] Yuandao Cai and Charles Zhang. 2023. A cocktail approach to practical call graph construction. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023). DOI: https://doi.org/10.1145/3622833

[17] Cristiano Calcagno and Dino Distefano. 2011. Infer: An automatic program verifier for memory safety of C programs. In *Proceedings of the NASA Formal Methods Symposium*. Springer, 459–465. DOI: https://doi.org/10.1007/978-3-642-20398-5_33

[18] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 480–491. DOI: https://doi.org/10.1145/1250734.1250789

[19] Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. 2023. A survey of chain of thought reasoning: Advances, frontiers and future. *arXiv preprint arXiv:2309.15402* (2023).

[20] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'23)*, René Just and Gordon Fraser (Eds.). ACM, 423–435. DOI: https://doi.org/10.1145/3597926.3598067

[21] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. DOI: https://doi.org/10.1145/3338112

[22] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR* abs/2307.09288 (2023).

[23] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. SMOKE: Scalable path-sensitive memory leak detection for millions of lines of code. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE/ACM, 72–82. DOI: https://doi.org/10.1109/ICSE.2019.00025

[24] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23)*. IEEE, 1469–1481. DOI: https://doi.org/10.1109/ICSE48619.2023.00128

[25] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349. DOI: https://doi.org/10.1145/24039.24041

[26] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering memory corruption in Linux. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*. The Internet Society. Retrieved from https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_05A-1_Gens_paper.pdf

[27] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 226–238. DOI: https://doi.org/10.1145/1480881.1480911

[28] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. IEEE, 267–279. DOI: https://doi.org/10.1109/ASE.2019.00034

[29] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *CoRR* abs/2308.10620 (2023).

[30] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed grey-box fuzzing with provable path pruning. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP'22)*. IEEE, 36–50. DOI: https://doi.org/10.1109/SP46214.2022.9833751

[31] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. 2019. How do developers act on static analysis alerts? An empirical study of coverity usage. In *Proceedings of the IEEE 30th International Symposium on Software Reliability Engineering (ISSRE'19)*. IEEE, 323–333. DOI: https://doi.org/10.1109/ISSRE.2019.00040

[32] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. IEEE, 672–681. DOI: https://doi.org/10.1109/ICSE.2013.6606613

[33] Ashwin Kallingal Joshy, Xueyuan Chen, Benjamin Steenhoek, and Wei Le. 2021. Validating static warnings via testing code fragments. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 540–552. DOI: https://doi.org/10.1145/3460319.3464832

[34] Ashwin Kallingal Joshy, Xueyuan Chen, Benjamin Steenhoek, and Wei Le. 2021. Validating static warnings via testing code fragments. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*. Association for Computing Machinery, New York, NY, 540–552. DOI: https://doi.org/10.1145/3460319.3464832

[35] Hong Jin Kang, Khai Loong Aw, and David Lo. 2022. Detecting false alarms from automatic static analysis tools: How far are we? In *Proceedings of the 44th International Conference on Software Engineering*. 698–709. DOI: https://doi.org/10.1145/3510003.3510214

[36] Sungmin Kang, Juyeon Yoon, Nargiz Askarbekkyzy, and Shin Yoo. 2023. Evaluating diverse large language models for automatic and general bug reproduction. *arXiv preprint arXiv:2311.04532* (2023).

[37] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23)*. IEEE, 2312–2323. DOI: https://doi.org/10.1109/ICSE48619.2023.00194

[38] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin B. Clement, and Neel Sundaresan. 2022. Learning to reduce false positives in analytic bug detectors. In *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering (ICSE'22)*. ACM, 1307–1316. DOI: https://doi.org/10.1145/3510003.3510153

[39] J. Kocoń, I. Cichecki, O. Kaszyca, M. Kochanek, D. Szydło, J. Baran, J. Bielaniewicz, M. Gruza, A. Janz, K. Kanclerz, A. Kocoń, B. Koptyra, W. Mieleszczenko-Kowszewicz, P. Miłkowski, M. Oleksy, M. Piasecki, Ł. Radliński, K. Wojtasik, S. Woźniak, P. Kazienko, ChatGPT: Jack of all trades, master of none. *Information Fusion.* 99 (2023) 101861.

[40] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS'22)*. Retrieved from http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[41] Ted Kremenek. 2008. Finding software bugs with the clang static analyzer. *Apple Inc* (2008), 2008–08.

[42] Daniel Kroening and Michael Tautschnig. 2014. CBMC - C bounded model checker—(competition contribution). In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'14) (Lecture Notes in Computer Science)*, Erika Ábrahám and Klaus Havelund (Eds.), Vol. 8413. Springer, 389–391. DOI: https://doi.org/10.1007/978-3-642-54862-8_26

[43] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–27. DOI: https://doi.org/10.1145/3527325

[44] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. 2019. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST'19)*. IEEE, 391–401. DOI: https://doi.org/10.1109/ICST.2019.00048

[45] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23)*. IEEE, 919–931. DOI: https://doi.org/10.1109/ICSE48619.2023.00085

[46] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Poster: Assisting static analysis with large language models: A ChatGPT experiment. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (SP'23)*. IEEE. Retrieved from https://www.ieee-security.org/TC/SP2023/downloads/SP23-posters/sp23-posters-paper39-final_version_2_page_abstract.pdf

[47] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2022. Path-sensitive and alias-aware typestate analysis for detecting OS bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 859–872. DOI: https://doi.org/10.1145/3503222.3507770

[48] Changhua Luo, Wei Meng, and Penghui Li. 2023. SelectFuzz: Efficient directed fuzzing with selective path exploration. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (SP'23)*. IEEE, 2693–2707. DOI: https://doi.org/10.1109/SP46215.2023.10179296

[49] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2023. The scope of ChatGPT in software engineering: A thorough investigation. *CoRR* abs/2305.12138 (2023)

[50] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. CHECKER: A soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 1007–1024. Retrieved from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry

[51] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. PSE: Explaining program failures via postmortem static analysis. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Richard N. Taylor and Matthew B. Dwyer (Eds.). ACM, 63–72. DOI: https://doi.org/10.1145/1029894.1029907

[52] Daniel Marjamäki. Cppcheck: A tool for static C/C++ code analysis. Retrieved ACCESSED: 2024 from https://cppcheck.sourceforge.io

[53] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. 2013. Scalable and incremental software bug detection. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 554–564. DOI: https://doi.org/10.1145/2491411.2501854

[54] Aniruddhan Murali, Noble Saji Mathews, Mahmoud Alfadel, Meiyappan Nagappan, and Meng Xu. 2023. FuzzSlice: Pruning false positives in static analysis warnings through function-level fuzzing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE'24)*. IEEE Computer Society, 767–779.

[55] Tukaram Muske and Alexander Serebrenik. 2022. Survey of approaches for postprocessing of static analysis alarms. *ACM Comput. Surv.* 55, 3, Article 48 (Feb. 2022), 39 pages. DOI: https://doi.org/10.1145/3494521

[56] Damien Octeau, Patrick D. McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium*, Samuel T. King (Ed.). USENIX Association, 543–558. Retrieved from https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/octeau

[57] Mads Chr. Olesen, René Rydhof Hansen, Julia L. Lawall, and Nicolas Palix. 2014. Coccinelle: Tool support for automated CERT C secure coding standard certification. *Sci. Comput. Program.* 91 (2014), 141–160. DOI: https://doi.org/10.1016/j.scico.2012.10.011

[58] OpenAI. ChatGPT: Optimizing language models for dialogue. Retrieved ACCESSED: 2024 from https://chat.openai.com

[59] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (SP'23)*. IEEE, 2339–2356. DOI: https://doi.org/10.1109/SP46215.2023.10179420

[60] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants? In *Proceedings of the 40th International Conference on Machine Learning (ICML'23)*. JMLR.org, Honolulu, Hawaii, USA.

[61] Partha Pratim Ray. 2023. ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope. *Internet Things Cyber-Phys. Systems*. https://doi.org/10.1016/j.iotcps.2023.04.003

[62] Thomas W. Reps. 1997. Program analysis via graph reachability. In *Proceedings of the International Symposium on Logic Programming*, Jan Maluszynski (Ed.). MIT Press, 5–19.

[63] Ruben. A code-understanding, code-browsing or code-search tool. This is a tool to index, then query or search C, C++, Java, Python, Ruby, Go and JavaScript source code. Retrieved ACCESSED: 2024. from https://github.com/ruben2020/codequery

[64] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Commun. ACM* 61, 4 (2018), 58–66. DOI: https://doi.org/10.1145/3188720

[65] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.* 167, 1&2 (1996), 131–170. DOI: https://doi.org/10.1016/0304-3975(96)00072-2

[66] Timo Schick and Hinrich Schütze. 2022. True few-shot learning with prompts—A real-world perspective. *Trans. Assoc. Computat. Ling.* 10 (2022), 716–731.

[67] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. 2020. Conquering the extensional scalability problem for value-flow analysis frameworks. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)* , Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 812–823. DOI: https://doi.org/10.1145/3377811.3380346

[68] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 693–706. DOI: https://doi.org/10.1145/3192366.3192418

[69] Johannes Späth, Karim Ali, and Eric Bodden. 2017. IDEal: Efficient and precise alias-aware dataflow analysis. In *Proceedings of the International Conference on Object-Oriented Programming, Languages and Applications (OOPSLA/SPLASH'17)*. ACM Press. Retrieved from https://bodden.de/pubs/sab17ideal.pdf

[70] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'16)*. Retrieved from https://www.bodden.de/pubs/sna+16boomerang.pdf

[71] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC'16)*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 265–266. DOI: https://doi.org/10.1145/2892208.2892235

[72] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'12)*, Mats Per Erik Heimdahl and Zhendong Su (Eds.). ACM, 254–264. DOI: https://doi.org/10.1145/2338965.2336784

[73] Maolin Sun, Yibiao Yang, Yang Wang, Ming Wen, Haoxiang Jia, and Yuming Zhou. 2023. SMT solver validation empowered by large pre-trained language models. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE'23)*. IEEE, 1288–1300.

[74] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, Hanwei Qian, Yang Liu, and Zhenyu Chen. 2023. Automatic code summarization via ChatGPT: How far are we? *CoRR* abs/2305.12865 (2023).

[75] Nigar M. Shafiq Surameery and Mohammed Y Shakor. 2023. Use Chat GPT to solve programming bugs. *Int. J. Inf. Technol. Comput. Eng.* 3, 01 (2023), 17–22.

[76] Christos Tsigkanos, Pooja Rani, Sebastian Müller, and Timo Kehrer. 2023. Large language models: The next frontier for variable discovery within metamorphic testing? In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'23)*. IEEE, 678–682.

[77] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empir. Softw. Eng.* 25 (2020), 1419–1457. DOI: https://doi.org/10.1007/s10664-019-09750-5

[78] Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022. Bridging pre-trained models and downstream tasks for source code understanding. In *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering (ICSE'22)*. ACM, 287–298. DOI: https://doi.org/10.1145/3510003.3510062

[79] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 999–1010.

[80] Junjie Wang, Song Wang, and Qing Wang. 2018. Is there a "golden" feature set for static warning identification?: An experimental evaluation. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'18)*, Markku Oivo, Daniel Méndez Fernández, and Audris Mockus (Eds.). ACM, 17:1–17:10. DOI: https://doi.org/10.1145/3239235.3239523

[81] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (2022).

[82] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS'22)*. Retrieved from http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[83] Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. 2022. Controlled concurrency testing via periodical scheduling. In *Proceedings of the 44th International Conference on Software Engineering*. 474–486.

[84] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 765–777.

[85] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23)*. IEEE, 1482–1494. DOI: https://doi.org/10.1109/ICSE48619.2023.00129

[86] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). USENIX Association, 163–176. Retrieved from http://www.usenix.org/events/osdi10/tech/full_papers/Xiong.pdf

[87] Mengdi Xu, Yikang Shen, Shun Zhang, Yuchen Lu, Ding Zhao, Joshua Tenenbaum, and Chuang Gan. 2022. Prompting decision transformer for few-shot policy generalization. In *Proceedings of the International Conference on Machine Learning*. PMLR, 24631–24645.

[88] Zhiwu Xu, Cheng Wen, and Shengchao Qin. 2018. State-taint analysis for detecting resource bugs. *Sci. Comput. Program.* 162 (2018), 93–109.

[89] Zhiwu Xu, Bohao Wu, Cheng Wen, Bin Zhang, Shengchao Qin, and Mengda He. 2024. RPG: Rust library fuzzing with pool-based fuzz target generation and generic support. In *Proceedings of the 46th International Conference on Software Engineering*.

[90] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 327–337. DOI: https://doi.org/10.1145/3180155.3180178

[91] Mengjiao Sherry Yang, Dale Schuurmans, Pieter Abbeel, and Ofir Nachum. 2022. Chain of thought imitation with procedure cloning. *Adv. Neural Inf. Process. Syst.* 35 (2022), 36366–36381.

[92] Chengfeng Ye, Yuandao Cai, and Charles Zhang. 2024. When threads meet interrupts: Effective static detection of interrupt-based deadlocks. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security'24)*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association.

[93] Xi Ye and Greg Durrett. 2022. The unreliability of explanations in few-shot prompting for textual reasoning. *Adv. Neural Inf. Process. Syst.* 35 (2022), 30378–30392.

[94] Joobeom Yun, Rustamov Fayozbek, Juhwan Kim, and Youngjoo Shin. 2023. Fuzzing of embedded systems: A survey. *ACM Comput. Surv.* 55, 7 (2023), 137:1–137:33. DOI: https://doi.org/10.1145/3538644

[95] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul L. Yu. 2020. UBITect: A precise and scalable method to detect use-before-initialization bugs in linux kernel. In *ESEC/FSE'20: Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 221–232. DOI: https://doi.org/10.1145/3368089.3409686

[96] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective interactive resolution of static analysis alarms. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 57:1–57:30. DOI: https://doi.org/10.1145/3133881

[97] Xiaowen Zhang, Ying Zhou, and Shin Hwei Tan. 2023. Efficient pattern-based static analysis approach via regular-expression rules. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'23)*, Tao Zhang, Xin Xia, and Nicole Novielli (Eds.). IEEE, 132–143. DOI: https://doi.org/10.1109/SANER56733.2023.00022

[98] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A survey of large language models. *CoRR* abs/2303.18223 (2023).

Received 15 September 2023; revised 30 January 2024; accepted 3 March 2024