Top-Down Multi-agent LLM System with Runtime Data for MISRA C Violation Detection

Ahcheong Lee
ahcheong.lee@kaist.ac.kr
KAIST
Daejeon, South Korea

Yeongbin Kang
yeongbin.kang@kaist.ac.kr
KAIST
Daejeon, South Korea

Sechang Jang
newwin0198@kaist.ac.kr
KAIST
Daejeon, South Korea

Heechan Yang
heechan.yang@kaist.ac.kr
KAIST
Daejeon, South Korea

Moonzoo Kim*
moonzoo.kim@vpluslab.kr
VPlusLab Inc./KAIST
Sungnam/Daejeon, South Korea

Abstract

Ensuring MISRA C compliance is crucial for the safety and reliability of embedded software. For this purpose, industries utilize static rule checkers but suffer high ratios of both missed violations and false alarms. Although companies also try to utilize LLMs for this purpose, existing LLM-based approaches have suboptimal performance for large, complex programs. We introduce LASIK, the world-first top-down multi-agent LLM compliance-checking framework for MISRA C, which combines on-demand repository-level context provision with runtime execution evidence. This design enables LASIK to report more true violations than state-of-the-art static analyzers and a naive vanilla LLM approach, while maintaining a low false positive ratio.

Our evaluation on 10 embedded projects shows that LASIK detects 1.3-2.7 times more true violations than the static analyzers and the vanilla LLM approach, while achieving the lowest false alarm ratio (26.7%). Furthermore, on an industrial software developed by VPlusLab Inc., LASIK detects up to 2.1 times more true violations with a low false alarm rate (i.e., 10.8%) than the baseline techniques.

CCS Concepts

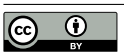
• **Software and its engineering → Software verification and validation**.

Keywords

Runtime information, Static analysis, Multi-agent system, Vulnerability detection, Coding guideline

ACM Reference Format:
Ahcheong Lee, Yeongbin Kang, Sechang Jang, Heechan Yang, and Moonzoo Kim. 2026. Top-Down Multi-agent LLM System with Runtime Data for MISRA C Violation Detection. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834497

*Corresponding author



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834497

1 Introduction

MISRA C [28] is a well-established set of coding guidelines to enhance the safety and reliability of embedded software systems [3, 14]. By restricting the C language to a well-defined subset through both directives and rules, MISRA C enables developers to eliminate undefined and implementation-dependent behaviors and enforce consistent code structure. Especially in safety-critical sectors, adherence to such a standard is a regulatory prerequisite for market entry. For example, the automotive functional safety standard ISO 26262 requires the use of a restricted language subset to ensure unambiguous semantics and to mitigate the risk of system failures.

VPlusLab Inc. (https://www.vpluslab.kr/en), an automated software testing firm in South Korea, maintains strategic research and development collaborations with leading embedded systems manufacturers, including Hyundai Motor Company and LIG Nex1. For these industrial partners, achieving and maintaining MISRA C compliance is a mission-critical objective. Failure to meet these standards can trigger catastrophic system malfunctions caused by undefined C behaviors. Identifying non-compliant code late in development incurs staggering costs, requiring extensive refactoring that leads to severe budget overruns and delivery delays.

Despite its widespread adoption across the automotive, aerospace, and military defense sectors, VPlusLab Inc. has found that maintaining MISRA C compliance in production-grade software remains a significant challenge. The introduction of modern language features, complex refactoring, and the integration of third-party libraries often introduce inadvertent violations into previously verified codebases. This difficulty is compounded by the evolving nature of the standard itself; since its inception in 1998, MISRA C has undergone several major revisions—culminating in MISRA C:2025—to mitigate language ambiguities, compiler-specific variances, and the emerging security risks inherent in modern embedded systems.

While static analyzers serve as the primary gatekeepers for MISRA C compliance, their effectiveness is limited by the constraints of fixed and deterministic heuristics. Traditional tools rely on pre-defined heuristics that frequently encounter a trade-off between soundness and completeness. In other words, to reduce false alarms that consume significant manual human efforts to filter out, static analyzers often suppress alarms in complex code paths and potentially leave critical false negatives undetected. Note that it threatens the safety of target products and significantly increases the total product cost in the long term. Also, since the rule violations

in such complex code paths are difficult to detect, they frequently cause highly dangerous and costly problems.

Also, several bottlenecks hinder adoption of LLM based violation detection in industries. A primary technical hurdle is that LLMs still show limited performance in grasping large-scale semantics, often failing to maintain the necessary depth of understanding required for complex systems; existing approaches lack comprehensive repository-level coverage and the context awareness essential for interpreting inter-procedural dependencies.

In addition, these models typically lack runtime context, which hinders them from filtering out false alarms. Furthermore, the reliance on public LLM services (e.g., ChatGPT) introduces substantial barriers. These include severe security risks regarding proprietary code and prohibitive financial costs; without a carefully designed multi-agent architecture, implementing LLM-based rule violation detection can incur large expenses.

These limitations motivated us to develop a multi-agent LLM framework that incorporates runtime information for repository-level analysis. The main contributions of LASIK are as follows:

(1) LASIK addresses the inherent limitations of both static analyzers and a vanilla LLM approach and detects a significantly higher volume of rule violations while maintaining a lower false alarm ratio.
(2) By leveraging runtime trace data and control-flow insights, unlike most static MISRA C checkers, our method distinguishes between violations that present real safety risks and that are benign under observed usage scenarios.
(3) The framework is implemented based on a local LLM, which removes the need to expose proprietary code to a third party and keeps the recurring analysis cost bounded, making adoption of LASIK smooth in industrial applications.
(4) We conducted extensive experiments comparing LASIK, state-of-the-art static analyzers, and the vanilla LLM approach on 10 open-source embedded projects and an industrial project of VPlusLab Inc.. The results demonstrate that LASIK identifies significantly more valid violations than the baseline techniques while maintaining a low false positive ratio.

The remaining sections are as follows. Sect. 2 shows the motivations for LASIK. Sect. 3 explains the details of LASIK. Sect. 4 describes the experiment setup and Sect. 5 shows the experiment results. Sect. 6 demonstrates the industrial application of LASIK. Sect. 7 discusses the results and areas for improvement. Sect. 8 describes related work. Sect. 9 concludes this paper with future work.

2 Motivation

Through long-standing collaborations with embedded systems manufacturers, VPlusLab Inc. has observed that ensuring MISRA C compliance remains a challenge for industrial partners, primarily due to the following technical limitations.

(1) **Limitations of static analyzers**: Traditional static analysis tools are often limited by pre-defined heuristics, resulting in a failure to detect complex, context-sensitive violations or the generation of excessive false positives that diminish developer productivity. For example, while state-of-the-art static analyzers failed to identify any violations of rule 13.2 in Table 2, LASIK successfully detected 51 cases on the 10 embedded subjects (Fig. 4).
(2) **Scarcity of MISRA C violation analysis research**: Current literature lacks a sophisticated methodology for adapting LLMs to MISRA C violation analysis (see Sect. 2.1).
(3) **Limitation of LLM**: Standard LLM approaches often operate with a limited context window, lacking the repository-level visibility and semantic depth required to verify inter-procedural rules and complex data-flow dependencies (see Sect. 2.2).

Table 1: Classification of Recent MISRA C Literature

Topics	References	Target Standard
Static Analysis Tools	[6, 24, 45]	MISRA C++:2008, MISRA C:2012
LLM-based Code Generation	[19, 37, 41]	MISRA C:2004, MISRA C++:2008
RTOS Compliance Evaluation	[40]	MISRA C:2012
Usability & Efficacy Studies	[16]	MISRA C:2012
Compliance-Driven Development	[21]	MISRA C:2012
LLM-based Compliance Analysis	[32]	MISRA C:2012

Table 2: MISRA C Guideline Examples

ID	Guideline
1.3	There shall be no occurrence of undefined or critical unspecified behavior
2.1	A project shall not contain unreachable code
2.2	A project shall not contain dead code
8.13	A pointer should point to a *const*-qualified type whenever possible
13.2	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders and shall be independent from thread interleaving
21.18	The size_t argument passed to any function in <string.h> shall have an appropriate value

2.1 Literature Review on MISRA C Analysis

Our review of recent MISRA C literature uncovers a significant gap between theoretical research and the functional requirements of industrial compliance. As categorized in Table 1, the majority of existing work remains orthogonal to the challenge of compliance verification. Instead, research has primarily focused on optimizing LLM-based code generation [19, 37, 41] or refining development processes to produce compliant code by construction [21]. Among the few studies addressing violation detection, most proposed static analysis techniques [6, 24, 45] that lack the flexibility of modern LLMs. Notably, the sole identified study utilizing LLMs employed a simple 'vanilla' approach, concluding the necessity of sophisticated multi-agent architectures for accurately analyzing complex, high-integrity rules [32].

Furthermore, a significant gap exists between current research and evolving industry standards. Much of the existing literature remains focused on legacy versions (e.g., MISRA C:2004 and 2012), thereby failing to address recent editions like MISRA C:2025. Because traditional static analysis techniques rely on fixed heuristics, they lack the flexibility required to adapt to these updated guidelines without extensive manual reconfiguration.

```
1  op = OS_STREAM_STATE_READABLE;
2
3  if (impl->selectable) {
4      waitflags   = MSG_DONTWAIT;
5
6      // OS_SelectSingle_Impl may modify the value of op
7      return_code =
8          OS_SelectSingle_Impl(token, &op, abs_timeout);
9  } else {
       //Omitted for simplicity ...
17 }
18
19 if (return_code == OS_SUCCESS) {
20     // The below `then' branch is reachable only if op
21     // is updated by OS_SelectSingle_Impl.
22     if ((op & OS_STREAM_STATE_READABLE) == 0) {
23         // LLM reports unreachable code
24         return_code = OS_ERROR_TIMEOUT;
25     }
26 }
```

Figure 1: A False Alarm Example in OSAL that is Resolvable Using Runtime Context

2.2 Limitations of LLM-based Rule Violation Detection

2.2.1 Limitations. While sophisticated analysis techniques specifically targeting MISRA C violations remain under-researched, significant effort has been directed toward LLM-based vulnerability detection. Two recent surveys [38, 47]—reviewing 58 and 80 papers, respectively—identified several critical observations common to state-of-the-art LLM-based approaches.

(1) LLM-based techniques show suboptimal performance on semantics with a large-scale complexity (i.e., LLMs often fail to understand a project-wide large codebase).
(2) Most studies still analyze functions or snippets in isolation without a global context. Consequently, LLMs are forced to evaluate based on "unseen dependencies", leading to inferences based on incomplete data rather than comprehensive program knowledge.

Besides the aforementioned limitations, we have observed that LLMs can significantly reduce false-positive reports by incorporating runtime context. Due to hallucinations or insufficient contextual knowledge, LLMs often generate inaccurate findings. Fig. 1 illustrates a false report that LASIK successfully identified and filtered out using runtime data. In this instance, the LLM initially reported a Rule 2.1 violation at line 24. As illustrated in Table 2, Rule 2.1 classifies unreachable code as a violation. This error occurred because the model failed to recognize that the *op* variable could be modified during the execution of the *OS_SelectSingle_Impl* function call at line 8. Thus, the LLM incorrectly assumed the branch condition at line 22 would always be false, rendering line 24 unreachable.

To correct such inaccuracies, we leverage the LLM's code generation capabilities to instrument the target code. By inserting strategic probes, we can gather dynamic runtime data to validate or filter

```
1  op = OS_STREAM_STATE_READABLE;
2  fprintf(stderr, "[LOG] op initially : 0x%X", op);
3
4  if (impl->selectable) {
5      waitflags   = MSG_DONTWAIT;
6
7      return_code =
8          OS_SelectSingle_Impl(token, &op, abs_timeout);
9  } else {
       //Omitted for simplicity ...
17 }
18
19 if (return_code == OS_SUCCESS) {
20     fprintf(stderr, "[LOG] op before check = 0x%X", op);
21     if ((op & OS_STREAM_STATE_READABLE) == 0) {
22         fprintf(stderr, "[LOG] Condition ((op &
               READABLE)==0) true");
23         // LASIK does not report unreachable code
24         return_code = OS_ERROR_TIMEOUT;
25     }
26 }
```

Figure 2: Instrumentation Example for Dynamic Data Retrieval

out reported violations. Fig. 2 displays the actual instrumentation LASIK applied to evaluate this report. By placing probes at lines 2, 20, and 22 and executing the instrumented program, LASIK observed that the value of *op* was indeed updated, confirming that line 24 is reachable and the initial violation report was a false positive.

(3) LLMs may fail to reason correctly due to a lack of runtime context.

Furthermore, security concerns prohibit the use of remote APIs that necessitate external code exposure. Consequently, we must rely on local LLMs deployed on on-site hardware. However, these local models typically have lower reasoning capabilities than state-of-the-art cloud alternatives, and local hardware often lacks the massive computational throughput of specialized AI data centers, leading to higher latency. Thus, it is essential to implement a sophisticated multi-agent architecture that can orchestrate these constrained local resources to solve complex tasks efficiently.

(4) Security concerns prohibit the use of public LLM services/APIs.

2.2.2 Solutions. To overcome these limitations and achieve both high effectiveness and efficiency, we designed LASIK as follows:

(1) To analyze complex semantics efficiently, LASIK employs a multi-agent, top-down, 'divide-and-conquer' methodology. The initial agents decompose the target function into a series of suspicious fragments, so the subsequent agents can make higher-fidelity decisions on narrowed snippets.
(2) LASIK employs tool-based agents capable of retrieving contextual data on-demand. This ensures the model evaluates code with a complete dataset, effectively eliminating the issue of 'unseen dependencies' that often plagues isolated analysis.

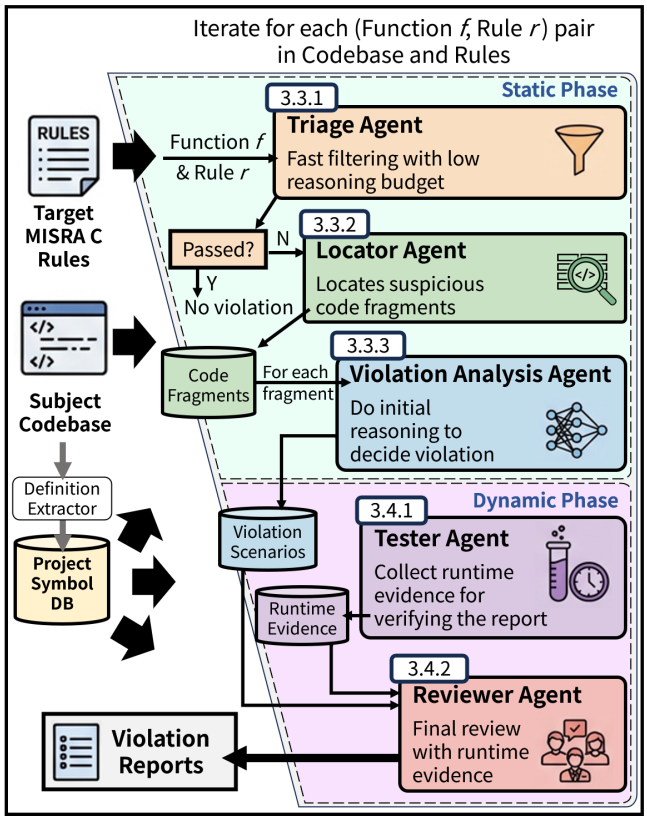


Figure 3: Overall Process of LASIK

(3) LASIK leverages runtime data to validate potential violation scenarios, ensuring high-fidelity analysis.
(4) LASIK is implemented to utilize local LLMs efficiently with asynchronous executions.

3 LASIK Framework

3.1 Overview

Following the suggestions in Sect. 2, we have developed LASIK (LlmbASed mIsra c rule checKer) to automatically detect a large number of MISRA C rule violations in target programs while keeping the false alarm ratio low. Figure 3 shows the overall process of LASIK.

LASIK is a *multi-agent system* designed to detect coding guideline violations using a *top-down, divide-and-conquer* architecture. By decomposing complex compliance checks into narrow, specialized tasks, the system enables individual agents to achieve higher precision and operational efficiency than a monolithic approach [12, 17]. For example, the Triage Agent (located at the top of Fig. 3) focuses on filtering out trivial cases with low reasoning budget, and the Violation Analysis Agent (located at the middle of Fig. 3) focuses on analyzing one suspicious code fragment with high reasoning budget. This divide-and-conquer approach of LASIK improves the analysis performance of LLM by reducing the amount of context information to process.

LASIK comprises five distinct agents, each of which configures LLM with a specialized prompt and a suite of tools (i.e., Python functions). These tools provide necessary global context on-demand, empowering the agents to perform informed, granular reasoning (see Sect. 3.2).

LASIK consists of the two consecutive phases.

(1) **Static Phase** (the upper part of Fig. 3): LASIK makes first decision on the given target function code based on static information. The generated violation scenarios will be investigated and validated further by using dynamic information obtained in the subsequent dynamic phase.
(2) **Dynamic Phase** (the lower part of Fig. 3): LASIK gathers dynamic runtime information by instrumenting and executing the target code with logging probes. These probes record important runtime control flow and state information necessary to analyze the violation reports generated during the static phase.

The following subsections (Sect. 3.2 to Sect. 3.4.2) explain the details of the five agents in LASIK.

3.2 Global Context Provision via Definition Extraction

To overcome the contextual limitations inherent in function-level analysis, LASIK incorporates a mechanism for global repository awareness. This process ensures that agents can resolve external dependencies without relying on speculation.

Prior to agent-based analysis, LASIK constructs a comprehensive global database for symbol definitions in a target project (located at the left middle part of Fig. 3). This repository indexes definition strings for functions, global variables, macros, and type declarations (i.e., Structs, Typedefs, and Unions) in an entire target project scope. To interface with this database, we implemented a suite of retrieval tools available to all five LASIK agents. These tools allow agents to query and retrieve symbol definitions on-demand whenever local information is insufficient for reasoning.

By granting agents the ability to fetch specific context only when needed, LASIK maintains a "context-aware" workflow that is both computationally efficient and semantically accurate, eliminating the ambiguity of unresolved dependencies. It also allows LASIK to analyze rules that require inter-procedural reasoning, such as Rule 2.1 (A project shall not contain unreachable code), whose evaluation depends on definitions outside the target function.

3.3 Static Phase

3.3.1 Triage Agent. To facilitate efficient repository-scale analysis, the Triage Agent acts as a high-throughput filter designed to eliminate clear non-violations with minimal reasoning overhead. The agent systematically iterates through all function-rule pairs, isolating only "suspicious" candidates for deeper investigation.

This top-down strategy is grounded in recent empirical findings [44] that demonstrate a high divergence between accuracy and token consumption across varying reasoning budget settings. For example, on the BrowseComp-Plus benchmark [7], the *gpt-oss-20b* model [33] achieved a 5.3x increase in accuracy when transitioning from a "low" to a "high" reasoning effort; however, this gain incurred a disproportionate 368.2x increase in token overhead.

By leveraging these insights, the Triage Agent performs low-cost filtering of trivial cases, effectively decoupling broad-scale

screening from the resource-intensive reasoning required for complex analysis. This top-down/hierarchical architecture ensures that industrial-scale applications remain cost-effective without sacrificing the rigorous precision essential for safety-critical standards.

3.3.2 Locator Agent. The Locator Agent is designed to pinpoint specific code fragments that potentially violate MISRA C rules. Following a top-down, divide-and-conquer methodology, the agent transforms complex functions into a structured list of "suspicious sites." Rather than attempting to analyze an entire function in a single pass, the agent isolates individual fragments to enable deep, precise reasoning within a narrowed scope.

3.3.3 Violation Analysis Agent. The Violation Analysis Agent determines whether a given code fragment in the target function violates the specified MISRA C rule. If the agent concludes that the code fragment does not violate the rule, LASIK terminates the analysis for that specific code fragment and rule pair without further investigation.

3.4 Dynamic Phase

3.4.1 Tester Agent. The Tester Agent is responsible for gathering dynamic evidence to validate violation scenarios generated by the Violation Analysis Agent. The agent achieves this by instrumenting the source code with probes and executing the resulting binaries compiled from the instrumented source code. To ensure a structured procedure, we designed prompts that provide the agent with four-step guidance:

(1) **Scenario Analysis**: First, the agent analyzes the violation hypothesis generated by the Violation Analysis agent to identify the specific execution paths and logical conditions (e.g., variable states, branch outputs) required to trigger the suspected MISRA C violation.
(2) **Probe Instrumentation**: Based on the identified conditions, the agent generates probe code (e.g., logging hooks or reachability markers) and embeds it into the target program.
(3) **Instrumented code execution**: Utilizing *the runtime tool* (see the description below), the agent executes the instrumented code and captures the runtime logs.
(4) **Post-processing**: Finally, the agent processes the execution logs into a structured evidence report. This report is then passed to the Reviewer Agent to provide a factual basis for the final determination.

This guidance ensures the Tester Agent focuses on designing effective probes (as illustrated in Figure 2) by effectively abstracting away complexities related to the runtime environment, compilation, test suites, and harnessing, which are handled by the Runtime Tool.

Runtime Tool: After the generation of instrumented code, the Tester Agent invokes the runtime tool to compile and execute the instrumented code through the following four-stage pipeline:

(1) **Preparation**: The tool initializes an environment by creating a temporary working directory and copying the original target project into the working directory.
(2) **Compilable instrumented source file generation**: The tool generates a compilable instrumented source file by inserting the given instrumented code into the original source file. Using Clang, the tool parses the given code into a list of declarations and replaces the original segments with their instrumented counterparts. While LLMs could theoretically perform this insertion, they often struggle with large file sizes and are prone to syntax errors or hallucinations.
(3) **Build**: The tool compiles and builds the merged file. If the process fails, the tool feeds the error messages back to the LLM, allowing it to troubleshoot the build problem and regenerate corrected code.
(4) **Execution**: The tool executes the program against the test suite and monitors for output changes. If any new log output is observed, the log is reported back to the LLM for analysis. The tool utilizes the test suite provided by the user.

3.4.2 Reviewer Agent. The Reviewer Agent is responsible for validating the intermediate violation reports produced during the static phase. After the Violation Analysis Agent identifies potential rule violations, the Reviewer Agent re-examines these findings to ensure that the decisions are both consistent and well-supported by the evidence collected during the dynamic phase.

3.5 Implementation

The core multi-agent system of LASIK is implemented in 5,000 lines of Python code with OpenAI Agents SDK. The runtime tool is implemented in 500 lines of Python code, and the code extraction and parsing component is implemented in 2,500 lines of C++ code with Clang visitor and LLVM IRBuilder.

To maximize GPU and VRAM utilization, we implemented LASIK to support the asynchronous LLM executions. To enable each asynchronous instance to independently build and run its own instrumented code, we developed an execution harness that provides isolated environments, preventing execution interference. By leveraging the vLLM [20] engine, LASIK efficiently manages concurrent inference and high-throughput serving at the system level. This architecture ensures that hardware resources remain fully utilized during large-scale repository analysis, enabling the process to complete within demanding industrial timelines.

4 Experiment Setup

4.1 Research Questions

RQ1. How accurately does LASIK identify rule violations in terms of the total number of true violations and the false alarm ratio?

We compare LASIK against three state-of-the-art static analyzers. We selected the following three widely used static analyzers that support MISRA C:2025 rule violation detections.

- **Analyzer-A**: Analyzer-A is a popular commercial static analyzer tool that has been widely adopted in safety-critical domains. It employs data flow analysis, control-flow modeling, and taint tracking to detect issues including buffer overflows . [1]
- **CodeQL**: CodeQL [2] is an open-source static analysis framework maintained by GitHub. CodeQL provides an official repository [2] for various coding standards, including AUTOSAR [1],

[1]We do not disclose the name of the tool to comply with the terms of its user license agreement.
[2]https://github.com/github/codeql-coding-standards

Table 3: Target Subjects

Program	Description	Size (LoC)	Line cov.
cJSON	JSON parser	6,943	88.7%
FreeRTOS	RTOS	16,857	80.0%
libexpat	XML parser	18,094	86.1%
libmetal	HAL	18,467	82.5%
libmicrohttpd	HTTP server	83,049	61.6%
libpng	Image decoding	51,422	69.1%
LittleFS	File system	7,660	79.0%
miniz	Data compression	9,418	79.0%
nanopb	Protocol buffers	4,553	80.6%
OSAL	OS interface	29,094	89.7%
Average		24,555.7	79.6%

SEI CERT [39], and MISRA C. The queries provided in this repository were utilized for our experiments.

- **CppCheck**: CppCheck [27] is a static analysis tool for C and C++ that focuses on detecting programming errors, undefined behaviors, and dangerous coding constructs. It performs bi-directional data flow to reduce false alarms.

In addition, we compare LASIK with a "vanilla" LLM baseline. To ensure a fair comparison, the vanilla LLM approach utilizes the same prompt as LASIK's Violation Analysis Agent but lacks access to LASIK's project symbol database. Additionally, we maintained a consistent reasoning effort of "medium" for both the vanilla LLM baseline and LASIK. We chose "medium" over "high" because the token overhead of higher effort is disproportionately high relative to its accuracy gain at scale (Sect. 3.3.1).

RQ2. How much does LASIK improve the precision of rule violation detection by utilizing the top-down divide-and-conquer approach? To what extent does the Locator Agent of LASIK improve the precision of rule violation detection? To evaluate the impact of the Locator Agent, we implemented LASIK$^{no\text{-}loc}$

- **LASIK$^{no\text{-}loc}$**: This variant excludes the Locator Agent of LASIK. The Violation Analysis Agent receives the whole target function code to determine violations.

RQ3. To what extent does dynamic runtime information generated by LASIK improve the precision of rule violation detection?

To evaluate how the Tester Agent and the Reviewer Agent contribute to LASIK's precision, we conducted an analysis of the violation reports filtered out by the Reviewer Agent during the RQ1 experiments. This allows us to quantify the impact of these agents on filtering false alarms.

4.2 Programs under Analysis

To evaluate LASIK, we utilized the latest versions of 10 widely used, open-source C programs from the embedded systems field that have been actively maintained through January 2026. These subjects span diverse domains (e.g., aerospace, communication, file system, and real-time OS) to mitigate potential domain-specific bias in the results. The details of the programs are provided in Table 3. The program sizes range from 6,943 to 83,049 lines of code, with an average size of 24,555.7.

4.3 Test Suite

LASIK utilizes a test suite to execute target programs and collect dynamic runtime information. To maximize the effectiveness of the Tester Agent, we collaborated with testing experts from VPlusLab Inc. to develop fuzzing drivers for each subject program. We used AFL++ [10] in its default configuration for 24 hours per subject, repeating the process 10 times to ensure stable coverage. As summarized in Table 3, the resulting test suites achieved an average line coverage of 79.6%, providing a sufficient basis for capturing runtime data. The developed fuzzing drivers are publicly available in our research artifact (https://doi.org/10.5281/zenodo.19943523).

Although this campaign incurs a substantial cost, it is a one-time investment per project, and many industrial partners have already made it, since ISO 26262 requires test suites with structural coverage evidence.

4.4 MISRA C Rules under Analysis

The MISRA C:2025 standard comprises 150 decidable rules and 51 undecidable rules. A rule is classified as decidable if compliance can be definitively determined in every case; for instance, Rule 9.3 (Arrays shall not be partially initialized) is easily verified through static analysis. Undecidable rules such as Rule 2.1 (A project shall not contain unreachable code) are inherently complex, since determining code reachability across all execution paths is functionally equivalent to the halting problem, which is undecidable in general.

In this study, we narrowed our scope by excluding decidable rules, as these are effectively handled by static analyzers. Of the 51 undecidable rules, we further excluded 17 that do not need the in-depth reasoning capabilities of an LLM. For instance, Rule 17.8 (A function parameter should not be modified) can be verified by simply identifying assignments to parameter variables. Additionally, we excluded nine rules of the concurrency group, which govern the correct use of threads and mutexes. This selection process resulted in a final set of 25 undecidable rules from MISRA C:2025 [3]. These rules are very difficult to detect and frequently generate false positives, which demand significant manual effort to validate [4, 30].

Since every selected undecidable rule can benefit from runtime data, we configured LASIK to always execute the dynamic phase. For example, Rule 8.13 (A pointer should point to a *const*-qualified type whenever possible) requires runtime data because deciding whether the pointed-to object is ever modified demands pointer aliasing analysis, which a static checker can only approximate conservatively.

4.5 Measurement

To address RQ1-RQ3, we manually classified the violations reported by each technique, comparing the totals of unique true positives and false positives. Detailed classification data is publicly available in our research artifact. For RQ1, we additionally recorded wall-clock execution time and token consumption to evaluate the overall efficiency of LASIK.

[3] 1.3, 2.1, 2.2, 8.13, 12.2, 13.1, 13.2, 13.5, 14.1, 14.2, 14.3, 17.5, 18.1, 18.2, 18.3, 18.6, 19.1, 21.14, 21.17, 21.18, 22.1, 22.2, 22.3, 22.4, and 22.6

4.6 Infrastructure

To avoid security concerns regarding the external exposure of industrial proprietary code through public LLM services, we utilized a state-of-the-art local LLM model *gpt-oss-120b* [33] for our experiments. The experiments were conducted on a workstation equipped with an AMD Ryzen 9 5950X 16-core processor and an NVIDIA RTX PRO 6000 Blackwell GPU (96GB VRAM). To maximize efficiency, we implemented LASIK to support asynchronous execution, configuring it to handle up to 320 concurrent tasks to fully utilize the available GPU VRAM.

4.7 Threats to Validity

Construct. We classify the violations reported by each technique rather than establishing a complete ground truth of all violations in the subject programs. Such a ground truth is infeasible for undecidable rules at this scale, since it would require manually inspecting every function against every rule. To mitigate this threat, we applied all techniques to identical source code, manually classified every report they produced, and published the resulting classification, so that the techniques are compared under identical conditions.

Internal. The manual report classification process may introduce subjectivity or human error, which could affect the reliability of our results. To mitigate this threat, we crosschecked the classification results and made the classification result publicly available. Also, LASIK is non-deterministic and we report a single run per subject, since repeating the experiment on all 10 subjects would require another round of manual classification, which was infeasible.

External. LASIK was evaluated on a limited set of 10 benchmark programs that might not be representative of other programs in general. However, we believe that this threat is limited as we targeted popular open-source embedded programs in different domains.

Reproducibility. The commercial analyzer (Analyzer-A) is anonymized, and its version and configuration cannot be disclosed under its license. The LASIK implementation and CROWN 2.0 are not disclosed as they are proprietary to VPlusLab Inc..

5 Experiment Results

5.1 RQ1. Effectiveness of LASIK Compared to the Baseline Tools

5.1.1 LASIK vs. Static Analyzers. Experimental results clearly show that LASIK significantly outperforms the state-of-the-art static analyzers in both detection effectiveness and precision (Table 4 and 5). Across 10 subjects, LASIK detects 1,345 true violations with a low average false positive ratio of 26.7%. In contrast, the static analyzers detected far fewer true violations (505, 775, 832 for CppCheck, CodeQL, Analyzer-A, respectively) with larger false alarm ratios (28.9%, 28.9%, 63.5% for CppCheck, CodeQL, Analyzer-A, respectively). Overall, LASIK detected between 1.61x (=1,345/832) and 2.66x (=1,345/505) more true violations than existing analyzers while maintaining a lower rate of false positives.

Furthermore, LASIK demonstrates superior stability across diverse software subjects compared to state-of-the-art static analyzers. While the static analyzers exhibit significant volatility in their false positive (FP) ratios, LASIK maintains a consistent performance. For instance, CppCheck achieves a 0% FP ratio on nanopb and OSAL but spikes to 73.9% for LittleFS. Similarly, CodeQL and Analyzer-A also show high variance in FP ratios, with ranges of 8.3%–57.7% and 24.8%–83.4%, respectively. In contrast, LASIK provides a steady FP ratio across all 10 programs, confined to a much narrower and more predictable range of 13.9% to 39.8%. This stability is particularly vital in industrial settings, where LASIK's predictable performance ensures a reliable developer workflow and manageable triage costs, avoiding the disruptive "false alarm spikes" that often lead to tool fatigue and pipeline bottlenecks.

5.1.2 LASIK vs. Vanilla LLM Prompting. The results demonstrate that LASIK significantly outperforms the Vanilla LLM approach, identifying 25.3% (=(1,345-1,073)/1,073) more true violations while reducing false alarms by 48.6% (Table 4 and 5). This superior performance of LASIK clearly demonstrates the advantage of its top-down divide-and-conquer architecture and runtime data utilization (see Sect. 3).

5.1.3 Per Rule Analysis. In addition to detecting the highest number of MISRA C rule violations, LASIK detected rule violations of diverse types. Among the 25 rules under the analysis, LASIK detected violations for the 23 rules, while CppCheck, CodeQL, Analyzer-A, and the Vanilla LLM approach detected violations for only 10, 14, 12, and 22 rules, respectively. Figure 4 shows the distribution of the numbers of true positives per rule reported by the state-of-the-art static analyzers, the Vanilla LLM approach, and LASIK. Among the 25 rules, LASIK detected the highest number of violations for 10 rules [4] and the second best for another 8 rules [5].

5.1.4 Cost Analysis. LASIK achieved high performance while maintaining moderate operational costs, making it well-suited for large-scale industrial applications. The final three columns of Table 5 provide details of wall-clock time and token consumption for each evaluated program. On average, LASIK completed the analysis of a single program in 2 hours and 11 minutes. During this process, it utilized an average of 90.4 million input tokens and 11.98 million output tokens.

Although local LLMs are cost-effective and secure, API-based models from external providers often demonstrate superior reasoning capabilities and performance. To evaluate the impact of model scale and reasoning power on violation detection, we applied LASIK to *LittleFS* using a public LLM service (gpt-5.5). This evaluation incurred a cost of 1,800 USD, processing 242.3M input tokens and 24.9M output tokens. Despite the expense, the API-driven approach identified 192 true violations while maintaining a comparable false positive rate—a 2.6 times increase in true positives over the local *gpt-oss-120b* configuration. These findings demonstrate that LASIK's robust architecture scales effectively with state-of-the-art models, albeit at a higher cost. This highlights the potential for a hybrid deployment strategy: employing local agents for routine, daily analysis while reserving high-reasoning, expensive models for high-risk code segments.

[4] 2.2, 8.13, 13.2, 14.1, 14.3, 19.1, 21.14, 21.18, 22.1, and 22.3
[5] 1.3, 2.1, 12.2, 17.5, 18.3, 18.6, 22.2, and 22.4

Table 4: Number of MISRA C Rule Violations Reported By Static Analyzers and Vanilla LLM Prompting

Project	CppCheck		CodeQL		Analyzer-A		Vanilla LLM prompting	
	TP(%)	FP(%)	TP(%)	FP(%)	TP(%)	FP(%)	TP(%)	FP(%)
cJSON	24 (63.2%)	14 (36.8%)	26 (65.0%)	14 (35.0%)	49 (17.7%)	228 (82.3%)	41 (40.2%)	61 (59.8%)
FreeRTOS	10 (40.0%)	15 (60.0%)	11 (91.7%)	1 (8.3%)	26 (45.6%)	31 (54.4%)	67 (68.4%)	31 (31.6%)
libexpat	41 (59.4%)	28 (40.6%)	81 (66.9%)	40 (33.1%)	118 (20.6%)	456 (79.4%)	144 (68.6%)	66 (31.4%)
libmetal	47 (100.0%)	0 (0.0%)	80 (80.8%)	19 (19.2%)	58 (65.2%)	31 (34.8%)	81 (55.9%)	64 (44.1%)
libmicrohttpd	86 (75.4%)	28 (24.6%)	191 (76.7%)	58 (23.3%)	140 (19.4%)	581 (80.6%)	179 (39.0%)	280 (61.0%)
libpng	103 (66.9%)	51 (33.1%)	85 (42.3%)	116 (57.7%)	194 (36.8%)	333 (63.2%)	223 (46.9%)	252 (53.1%)
LittleFS	42 (26.1%)	119 (73.9%)	38 (90.5%)	4 (9.5%)	54 (24.8%)	164 (75.2%)	44 (38.3%)	71 (61.7%)
miniz	89 (80.2%)	22 (19.8%)	146 (77.7%)	42 (22.3%)	96 (16.6%)	482 (83.4%)	137 (78.3%)	38 (21.7%)
nanopb	8 (100.0%)	0 (0.0%)	10 (55.6%)	8 (44.4%)	21 (42.9%)	28 (57.1%)	32 (44.4%)	40 (55.6%)
OSAL	55 (100.0%)	0 (0.0%)	107 (63.7%)	61 (36.3%)	76 (75.2%)	25 (24.8%)	125 (56.1%)	98 (43.9%)
Total (Avg.%)	505 (71.1%)	277 (28.9%)	775 (71.1%)	363 (28.9%)	832 (36.5%)	2359 (63.5%)	1073 (53.6%)	1001 (46.4%)

Table 5: Number of MISRA C Rule Violations Reported by LASIK and LASIK Variants

Project	LASIK$^{no\text{-}loc}$		LASIK^{filtered_out}		LASIK				
	TP (%)	FP (%)	TP (%)	FP (%)	TP (%)	FP (%)	Time	T^{in}	T^{out}
cJSON	31 (72.1%)	12 (27.9%)	14 (16.9%)	69 (83.1%)	52 (75.4%)	17 (24.6%)	38m	31.8M	3.8M
FreeRTOS	35 (83.3%)	7 (16.7%)	19 (33.9%)	37 (66.1%)	49 (80.3%)	12 (19.7%)	44m	44.2M	5.0M
libexpat	74 (72.5%)	28 (27.5%)	112 (56.3%)	87 (43.7%)	121 (73.3%)	44 (26.7%)	2h19m	92.6M	10.2M
libmetal	55 (73.3%)	20 (26.7%)	79 (58.5%)	56 (41.5%)	99 (70.7%)	41 (29.3%)	1h1m	46.4M	7.3M
libmicrohttpd	145 (66.5%)	73 (33.5%)	86 (16.4%)	437 (83.6%)	291 (61.9%)	179 (38.1%)	4h38m	164.2M	19.6M
libpng	169 (76.1%)	53 (23.9%)	48 (11.3%)	375 (88.7%)	349 (78.4%)	96 (21.6%)	4h33m	193.9M	23.9M
LittleFS	37 (68.5%)	17 (31.5%)	39 (26.7%)	107 (73.3%)	74 (60.2%)	49 (39.8%)	1h57m	78.9M	12.9M
miniz	69 (74.2%)	24 (25.8%)	174 (53.5%)	151 (46.5%)	128 (76.6%)	39 (23.4%)	2h56m	92.4M	14.3M
nanopb	10 (66.7%)	5 (33.3%)	52 (42.6%)	70 (57.4%)	33 (70.2%)	14 (29.8%)	1h21m	43.0M	6.1M
OSAL	92 (81.4%)	21 (18.6%)	42 (25.1%)	125 (74.9%)	149 (86.1%)	24 (13.9%)	1h49m	116.6M	16.7M
Total (Avg.%)	717 (73.5%)	260 (26.5%)	665 (34.1%)	1514 (65.9%)	**1345 (73.3%)**	515 (26.7%)	21h56m	904.0M	119.8M

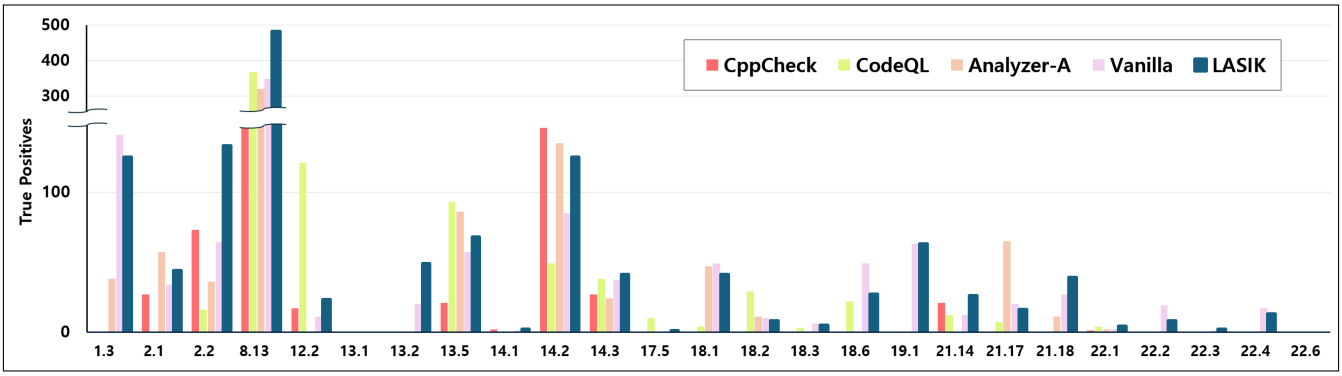


Figure 4: Distribution of Detected True Positives by Rule

5.2 RQ2. Effectiveness of Divide-and-conquer Approach

The results demonstrate that LASIK could detect significantly more valid violations while maintaining a low false positive rate by incorporating the divide-and-conquer approach supported by the

Locator Agent (Table 5). Specifically, LASIK detected 87.6% (=(1,345-717)/717) more true violations than LASIK$^{no\text{-}loc}$, while maintaining similar false positive rate 26.5%. For example, LASIK detected 2.1 (=349/169) times more true violations with lower FP ratio than LASIK$^{no\text{-}loc}$ on `libpng`.

5.3 RQ3. Effectiveness of Runtime Information on Rule Violation Detection

The results show that the Tester Agent and the Reviewer Agent effectively filtered out false reports generated by the Violation Analysis Agent. The fourth and fifth columns of Table 5 show the number of true violations and false violations that are filtered out by the Reviewer Agent with runtime data. By utilizing runtime information, LASIK reduced false positives by 74.6% (=1,514/(1,514+515)) while preserving 66.9% (=1,345/(665+1,345)) of true positives. For instance, on `cJSON` where the Violation Analyzer Agent reported 66 valid and 86 false violations, LASIK retained 78.8% (=52/66) of the valid violations while reducing false positives by 80.2%(=(86-17)/86) with the help of the runtime information of a target program provided by the agents in the Dynamic Phase.

In addition, the performance of Analyzer-A highlights a limitation inherent in reliance on only static information. In our evaluation, Analyzer-A reported significantly more false alarms (2,359) than any other evaluated technique. A granular breakdown of these results reveals that about 69.3% of these errors (1,634 instances) were concentrated within Rule 18.1 and 18.2. These rules are intended to identify buffer overruns and underruns from unsafe pointer arithmetic operations. However, rather than performing a path-sensitive analysis to determine actual risk, the underlying heuristic seems designed to flag all pointer arithmetic operations, including those that are safe. In contrast, LASIK overcomes these limitations through its runtime data utilization.

6 Industrial Application to CROWN 2.0

6.1 Project Background

To evaluate the practical effectiveness and efficiency of LASIK on real-world industrial software, we conducted a case study on CROWN 2.0, an automated white-box unit testing tool developed by VPlusLab Inc.[6].

CROWN 2.0 serves as a critical component in the unit testing life-cycle for industrial embedded software to achieve white-box structural coverage required by ISO 26262 and DO 178A/B/C. Since CROWN 2.0 has been applied for various safety-critical applications such as automotive and military defense systems, industrial partners necessitate CROWN 2.0's adherence to rigorous coding standards to improve its own software quality, such as MISRA C.

Furthermore, VPlusLab Inc. has identified a significant industrial demand for more sophisticated MISRA C violation detection techniques. Traditional analysis tools often suffer from high rates of both false negatives and false positives, which can lead to substantial technical risks and financial liabilities for stakeholders. By applying LASIK to CROWN 2.0, we aim to demonstrate its ability to overcome these limitations in a high-stakes industrial context.

[6] https://www.vpluslab.kr/en/solution

6.2 Experiment Setup

6.2.1 MISRA Rules Applied. One important advantage of the LLM-based approach is its flexibility across different programming languages, as it does not heavily rely on language-specific heuristics like static analyzers. While CROWN 2.0 is written in C++, the LLM-driven nature of LASIK allows it to be seamlessly applied to a C++ codebase without requiring additional implementation effort. For this evaluation, we applied 20 guidelines from among the original 25, specifically those also included in MISRA C++:2023 (excluding rules 13.1, 17.5, 21.14, 22.1, and 22.2).

6.2.2 Target Subject. The target CROWN 2.0 codebase is approximately 6K LoC, and our evaluation utilizes the test suite developed by VPlusLab Inc., which achieves 77.9% line coverage and 67.7% branch coverage. We leverage LASIK to minimize MISRA C violations, thereby mitigating technical risks for stakeholders.

6.2.3 LLM Models. In addition to the *gpt-oss-120b* model used in our experiments, we also utilized *gpt-oss-20b* to evaluate the feasibility of lighter and faster industrial applications. While the 120B model offers superior reasoning depth, the 20B variant provides an advantage in operational efficiency: it significantly reduces inference latency and can be hosted on hardware with far lower VRAM requirements (e.g., *gpt-oss-20b* can run on RTX 5060 TI with 16 GB VRAM whose cost is around 500 USD).

6.3 Experiment Result

The results shown in Table 6 demonstrate that LASIK is both effective and efficient when applied to industrial software. Specifically, LASIK identified the highest number of true violations (157) while maintaining a low false-positive rate of 10.8%. Furthermore, LASIK exhibited a broader detection range than the state-of-the-art static analyzers; it successfully identified violations across eight distinct rules [7], whereas the baseline static analyzers only detected four.

LASIK completed the analysis of CROWN 2.0 in 3 hours and 32 minutes with the 120B model and 45 minutes using the 20B model. This performance profile is well-suited for real-world industrial environments. Furthermore, due to its robust architecture, LASIK with even the 20B model identified a higher number of true violations than state-of-the-art static analyzers, despite a marginal decrease in true positives and a slight increase in false positives compared to the larger model.

6.3.1 Case Study. Figure 5 illustrates a true violation detected by LASIK that state-of-the-art static analyzers failed to identify. The *strncmp* call in line 3 violates Rule 21.18 and may lead to undefined behavior. Rule 21.18 prohibits inappropriate usage of *size_t* argument (i.e., the third argument) on *strncmp* (Table 2). When CROWN 2.0 is deployed for testing, the function name is passed to *FuncName* in line 1. The code fragment attempts to verify if the function name begins with the prefix "ctc_". However, because the logic does not validate that *FuncName* is at least four characters long, the *strncmp* call triggers a buffer overrun when processing a shorter function name.

This example illustrates a key advantage of LASIK. While an LLM can recognize the semantic context of code and infer that the

[7] 1.3, 2.1, 2.2, 8.13, 13.2, 13.5, 21.17, and 21.18

Table 6: Comparison of SOTA Techniques and LASIK on CROWN 2.0

Tool	TP (ratio)	FP (ratio)
CppCheck	105 (60.7%)	68 (39.3%)
CodeQL	129 (97.7%)	3 (2.3%)
Analyzer-A	76 (46.9%)	86 (53.1%)
Vanilla LLM	75 (67.0%)	37 (33.0%)
LASIK-20b	130 (74.7%)	44 (25.3%)
LASIK-120b	**157** (89.2%)	19 (10.8%)

```
string FuncName = f->getNameInfo().getName.getAsString();

if (strncmp(FuncName.c_str(), "ctc_", 4) == 0) {
    return true;
}
```

Figure 5: An Example of Incorrect *strncmp* Usage in CROWN 2.0, Detected by LASIK

FuncName variable refers to a function name that may be shorter than four characters, traditional static analyzers do not. Because these traditional tools rely on fixed heuristics, they lack the ability to grasp developer intent. Consequently, they cannot identify that such a string is valid without performing computationally expensive control flow and data flow analysis.

By addressing reported violations, VPlusLab Inc. could improve the code quality of its product while simultaneously reducing technical debt, ensuring long-term maintainability, and mitigating the security risks.

7 Discussion

7.1 Limitation Study based on False Negative Analysis

To identify areas for improvement, we analyzed LASIK's false negatives, the true violations that LASIK failed to report but were identified by other static analyzer tools or the Vanilla LLM approach. While LASIK uniquely identified 508 violations missed by other tools, the 4 baseline techniques reported a total of 1,416 violations that LASIK failed to report. These false negatives primarily occur with the three MISRA C rules: Rule 8.13 (19.6%=277/1,416), Rule 1.3 (10.0%=142/1,416), and Rule 2.2 (9.5%=135/1,416). These rule guidelines are specified in Table 2.

A primary cause for the observed false negatives is a lack of systemic context. To accurately evaluate violation scenarios for the aforementioned rules, an LLM requires a comprehensive understanding of the system's execution state. For example, Rule 1.3 mandates the absence of undefined behavior, but without external context, the Violation Analysis Agent may erroneously assume that a function's pointer parameter could be NULL. Consequently, it might flag a dereference as a potential violation, even if the broader system logic ensures that the pointer is always initialized.

While LASIK mitigates this by providing global and runtime context and effectively filtering out significantly more false alarms than the Vanilla approach, its performance is still constrained by the two factors: 1) the coverage/reachability of the test suite (Sect. 7.2) and 2) the scalability of context aggregation (Sect. 7.3).

7.2 Need for a New Test Suite Quality Metric

A comprehensive test suite serves as the fundamental basis for any runtime-based analysis. LASIK can not observe runtime behaviors within unexplored executions. By leveraging the testing expertise of VPlusLab Inc., we utilized a high-quality test suite, achieving an average line coverage of 79.6% across 10 open source projects. However, our false negative evaluation revealed that high line coverage does not guarantee in-depth analysis.

Specifically, we found that LASIK lacked access to certain runtime data critical for validating violation scenarios discussed in Sect. 7.1. A vast amount of diverse execution data (such as specific state transitions or edge-case variable values) remains not fully captured by traditional coverage metrics. Consequently, we identified the need for a new runtime execution metric capable of quantifying test suite quality in a way that specifically enhances the contextual reasoning of LLM-based analysis.

7.3 Need for Aggregation of Multi-executions

Aggregating data across multiple execution traces enables the extraction of high-level insights, such as value distributions and dynamic invariants (e.g., verifying that a string consistently maintains a length of 10 characters across 10,000 runs). However, the substantial volume of runtime data generated by each execution often exceeds the context window limitations of current LLMs.

While LASIK is presently restricted to processing a single execution trace per request, implementing a robust summarization or aggregation mechanism will be beneficial. Such an approach would allow the agent to synthesize diverse runtime states into a comprehensive systemic context, significantly enhancing its ability to detect subtle, context-sensitive violations.

8 Related Work

8.1 LLMs and Traditional Static Analysis Tools

Recent research has explored the application of LLMs for software vulnerability detection, often benchmarking their performance against traditional static analysis tools. Multiple studies have demonstrated that LLMs can achieve higher detection rates and F-1 scores across various vulnerability types [9, 11, 29, 34, 35, 46]. However, this improved detection capability is often accompanied by a substantial increase in false positives [34]. LASIK extends this line of comparison to the domain of MISRA C:2025 rule checking. In our experiments, we also observe that direct application of LLMs to MISRA C compliance detection suffers from a high false positive rate, motivating the design of LASIK as a multi-agent system.

8.2 LLM-based Multi-Agent Framework

Multi-agent frameworks have emerged to orchestrate reasoning and reliability in software security. Agents are typically assigned to distinct roles, such as performing repository audits to reduce hallucinations [13] or detecting vulnerabilities [42]. Specifically, AutoPatch [36] performs static analysis on LLM-generated code and employs two separate LLM agents: (1) to verify whether the

code contains any vulnerabilities, and (2) to patch any detected vulnerabilities. Similarly, LASIK adopts a *top-down, divide-and-conquer* approach and utilizes five separate agents, each focusing on a distinct task. This allows each agent to concentrate on a narrow aspect of analysis, enabling more precise evaluations and decisions.

8.3 LLMs with Knowledge Augmentation

Also, researchers have incorporated additional data to ground models with domain-specific knowledge to address limitations of LLMs. Prior work has improved LLM accuracy on vulnerability detection by leveraging historical vulnerabilities and fixes [8], public knowledge bases such as CWE [5], API documentation [48], and relevant code examples identified via semantic, lexical, or structural similarity [25, 26]. While LASIK also utilizes external context, it distinguishes itself by incorporating both static and dynamic runtime information to refine decision accuracy. Similar to LASIK, AutoSafeCoder [31] combines static analysis with a dynamic fuzzing agent to generate safe code. However, AutoSafeCoder observes only crashing outcomes (i.e., encountered errors) from its fuzzing campaign. In contrast, LASIK inserts diverse probes that capture key control flows and memory states, providing richer and more fine-grained data than mere crashing results. By leveraging this detailed runtime information, LASIK can enhance its ability to detect rule violations.

9 Conclusion

We present LASIK, a novel multi-agent MISRA C rule violation detection framework that enhances rule violation detection performance with a top-down, divide-and-conquer architecture and dynamic runtime information. We have applied LASIK to 10 real open-source programs and identified 1,345 valid violations, outperforming state-of-the-art static analyzers by detecting 1.6 to 2.7 times more valid violations while maintaining the lowest false positive ratio at 26.7%. Furthermore, we have demonstrated the high effectiveness and efficiency of LASIK on industrial software CROWN 2.0 developed by VPlusLab Inc. (157 true violations detected with a low false alarm rate 10.8%). For future work, we plan to extend LASIK to detect general vulnerabilities and improve the runtime data utilization technique specialized for LLM. We also plan to integrate state-of-the-art fuzzing techniques [18, 22, 23] and test generation techniques [15] into LASIK, so that it generates inputs targeting a given violation scenario instead of relying on a pre-built test suite. In addition, we plan to apply automated debugging techniques [43] to generate patches for the detected violations.

10 Data Availability

The experiment results for the open-source subjects are publicly available (https://doi.org/10.5281/zenodo.19943523). However, the LASIK implementation and the CROWN 2.0 results are not publicly accessible, as they are the proprietary property of VPlusLab Inc.

Acknowledgments

This research was supported by the National Research Foundation grants funded by the Korean government (RS-2021-NR060080 and NRF-RS-2024-00357348).

References

[1] AUTOSAR Consortium. 2022. *Guidelines for the use of the C++14 language in critical and safety-related systems.* AUTOSAR. https://www.autosar.org/fileadmin/standards/R22-11/AP/AUTOSAR_RS_CPP14Guidelines.pdf Release 22-11.

[2] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:25. https://doi.org/10.4230/LIPICS.ECOOP.2016.2

[3] Roberto Bagnara, Abramo Bagnara, and Patricia M. Hill. 2018. The MISRA C Coding Standard and its Role in the Development and Analysis of Safety- and Security-Critical Embedded Software. In *Static Analysis*, Andreas Podelski (Ed.). Springer International Publishing, Cham, 5–23. https://doi.org/10.1007/978-3-319-99725-4_2

[4] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. *SIGPLAN Not.* 38, 5 (May 2003), 196–207. https://doi.org/10.1145/780822.781153

[5] Daipeng Cao and W Jun. 2024. Llm-cloudsec: Large language model empowered automatic and deep vulnerability analysis for intelligent clouds. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 1–6. https://doi.org/10.1109/INFOCOMWKSHPS61880.2024.10620804

[6] Chih-Yuan Chen, Yung-An Fang, Guan-Ren Wang, and Peng-Sheng Chen. 2023. A GCC-based checker for compliance with MISRA-C's single-translation-unit rules. *Connection Science* 35, 1 (2023), 2222934. https://doi.org/10.1080/09540091.2023.2222934

[7] Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. 2025. BrowseComp-Plus: A More Fair and Transparent Evaluation Benchmark of Deep-Research Agent. *arXiv preprint arXiv:2508.06600* (2025). https://doi.org/10.48550/arXiv.2508.06600

[8] Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, et al. 2024. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *arXiv preprint arXiv:2406.11147* (2024). https://doi.org/10.48550/arXiv.2406.11147

[9] Richard A Dubniczky, Krisztofer Zoltan Horvát, Tamás Bisztray, Mohamed Amine Ferrag, Lucas C Cordeiro, and Norbert Tihanyi. 2025. Castle: Benchmarking dataset for static code analyzers and llms towards cwe detection. In *International Symposium on Theoretical Aspects of Software Engineering*. Springer, 253–272. https://doi.org/10.1007/978-3-031-98208-8_15

[10] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[11] Damian Gnieciak and Tomasz Szandala. 2025. Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection. *arXiv preprint arXiv:2508.04448* (2025). https://doi.org/10.48550/arXiv.2508.04448

[12] Zhouhong Gu, Xiaoxuan Zhu, Yin Cai, Hao Shen, Xingzhou Chen, Qingyi Wang, Jialin Li, Xiaoran Shi, Haoran Guo, Wenxuan Huang, et al. 2025. AgentGroupChat-V2: Divide-and-Conquer Is What LLM-Based Multi-Agent System Need. *arXiv preprint arXiv:2506.15451* (2025). https://doi.org/10.48550/arXiv.2506.15451

[13] Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. 2025. Repoaudit: An autonomous llm-agent for repository-level code auditing. *arXiv preprint arXiv:2501.18160* (2025). https://doi.org/10.48550/arXiv.2501.18160

[14] Les Hatton. 2004. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology* 46, 7 (2004), 465–472. https://doi.org/10.1016/j.infsof.2003.09.016

[15] Robert Sebastian Herlim, Yunho Kim, and Moonzoo Kim. 2022. CITRUS: Automated Unit Testing Tool for Real-world C++ Programs. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 400–410. https://doi.org/10.1109/ICST53961.2022.00046

[16] Alexander Homann, Lisa Grabinger, Florian Hauser, and Jürgen Mottok. 2023. An eye tracking study on MISRA C coding guidelines. In *Proceedings of the 5th European Conference on Software Engineering Education*. 130–137. https://doi.org/10.1145/3593663.3593671

[17] Zican Hu, Wei Liu, Xiaoye Qu, Xiangyu Yue, Chunlin Chen, Zhi Wang, and Yu Cheng. 2025. Divide and Conquer: Grounding LLMs as Efficient Decision-Making Agents via Offline Hierarchical Reinforcement Learning. *arXiv preprint arXiv:2505.19761* (2025). https://doi.org/10.48550/arXiv.2505.19761

[18] Kieun Kim, Seongmin Lee, and Shin Hong. 2025. Refining Fuzzed Crashing Inputs for Better Fault Diagnosis. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering* (Clarion Hotel Trondheim, Trondheim, Norway) *(FSE Companion '25)*. Association for Computing Machinery, New York,

NY, USA, 1248–1249. https://doi.org/10.1145/3696630.3731436
[19] Sven Kirchner and Alois C Knoll. 2025. Generating automotive code: Large language models for software development and verification in safety-critical systems. In *2025 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 813–820. https://doi.org/10.1109/IV64158.2025.11097503
[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. https://doi.org/10.1145/3600006.3613165
[21] Rachid Latif, Moussa Bellaou, Lahoucaine Amenzouy, and Abdessamade Elboudani. 2025. Safety-Oriented Evaluation of Embedded C Coding Methodologies for Automotive Software Using MISRA C and ISO 26262 Standards. In *2025 International Conference on Electrical Systems & Automation (ICESA)*. IEEE, 1–5. https://doi.org/10.1109/ICESA66763.2025.11281351
[22] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. 2022. POWER: Program option-aware fuzzer for high bug detection ability. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 220–231. https://doi.org/10.1109/ICST53961.2022.00032
[23] Ahcheong Lee, Youngseok Choi, Shin Hong, Yunho Kim, Kyutae Cho, and Moonzoo Kim. 2025. ZigZagFuzz: Interleaved Fuzzing of Program Options and Files. *ACM Trans. Softw. Eng. Methodol.* 34, 2, Article 39 (Jan. 2025), 31 pages. https://doi.org/10.1145/3697014
[24] Che-Chia Lin, Wei-Hsu Chu, Chia-Hsuan Chang, Hui-Hsin Liao, Chun-Chieh Yang, Jenq-Kuen Lee, Yi-Ping You, and Tien-Yuan Hsieh. 2025. Support of MISRA C++ Analyzer for Reliability of Embedded Systems. *ACM Trans. Cyber-Phys. Syst.* 9, 1, Article 9 (Jan. 2025), 27 pages. https://doi.org/10.1145/3611390
[25] Zhihong Liu, Qing Liao, Wenchao Gu, and Cuiyun Gao. 2023. Software vulnerability detection with gpt and in-context learning. In *2023 8th International Conference on Data Science in Cyberspace (DSC)*. IEEE, 229–236. https://doi.org/10.1109/DSC59305.2023.00041
[26] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software* 212 (2024), 112031. https://doi.org/10.1016/j.jss.2024.112031
[27] Daniel Marjamäki. 2023. Cppcheck: A Static Analysis Tool for C/C++ Code. https://cppcheck.sourceforge.io/ Accessed: 2025-09-12.
[28] MISRA Consortium Limited. 2023. *MISRA C:2023 - Guidelines for the use of the C language in critical systems*. https://www.misra.org.uk/
[29] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. 2024. Effectiveness of chatgpt for static analysis: How far are we?. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*. 151–160. https://doi.org/10.1145/3664646.3664777
[30] Tukaram Muske and Alexander Serebrenik. 2016. Survey of Approaches for Handling Static Analysis Alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 157–166. https://doi.org/10.1109/SCAM.2016.25
[31] Ana Nunez, Nafis Tanveer Islam, Sumit Kumar Jha, and Peyman Najafirad. 2024. Autosafe coder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing. *arXiv preprint arXiv:2409.10737* (2024). https://doi.org/10.48550/arXiv.2409.10737
[32] Dorin Nuțeanu, Alexandru Guzu, and Georgian Nicolae. 2025. Analyzing Compliance with Safety Standards in C Code via Large Language Models. In *2025 International Symposium ELMAR*. IEEE, 331–335. https://doi.org/10.1109/ELMAR66948.2025.11194021
[33] OpenAI. 2025. gpt-oss-120b & gpt-oss-20b Model Card. https://doi.org/10.48550/arXiv.2508.10925 arXiv:2508.10925 [cs.CL]
[34] Omer Said Ozturk, Emre Ekmekcioglu, Orcun Cetin, Budi Arief, and Julio Hernandez-Castro. 2023. New tricks to old codes: can ai chatbots replace static code analysis tools?. In *Proceedings of the 2023 European Interdisciplinary Cybersecurity Conference*. 13–18. https://doi.org/10.1145/3590777.3590780
[35] Moumita Das Purba, Arpita Ghosh, Benjamin J Radford, and Bill Chu. 2023. Software vulnerability detection using large language models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 112–119. https://doi.org/10.1109/ISSREW60843.2023.00058
[36] Minjae Seo, Wonwoo Choi, Myoungsung You, and Seungwon Shin. 2025. AutoPatch: Multi-Agent Framework for Patching Real-World CVE Vulnerabilities. *arXiv preprint arXiv:2505.04195* (2025). https://doi.org/10.48550/arXiv.2505.04195
[37] Merlijn Sevenhuijsen, Minal Suresh Patil, Mattias Nyberg, and Gustav Ung. 2025. Generating Safety-Critical Automotive C-programs using LLMs with Formal Verification. In *19th International Conference on Neurosymbolic Learning and Reasoning*.
[38] Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. 2025. LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights. *ACM Comput. Surv.* 58, 5, Article 134 (Nov. 2025), 35 pages. https://doi.org/10.1145/3769082
[39] Software Engineering Institute, Carnegie Mellon University. 2016. *SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems*. Software Engineering Institute, Carnegie Mellon University.
[40] Jia Song, Ronisha Shigdel, Aditi Pokharel, and Jim Alves-Foss. 2024. Real-Time Operating Systems' Compliance With MISRA-C Coding Standard: A Comprehensive Study. *IEEE Access* 12 (2024), 151955–151974. https://doi.org/10.1109/ACCESS.2024.3479971
[41] Malik Muhammad Umer. 2025. Comparative Analysis of the Code Generated by Popular Large Language Models (LLMs) for MISRA C++ Compliance. *IEEE Access* 13 (2025), 194815–194831. https://doi.org/10.1109/access.2025.3633086
[42] Ratnadira Widyasari, Martin Weyssow, Ivana Clairine Irsan, Han Wei Ang, Frank Liauw, Eng Lieh Ouh, Lwin Khin Shar, Hong Jin Kang, and David Lo. 2025. Let the Trial Begin: A Mock-Court Approach to Vulnerability Detection using LLM-Based Agents. *arXiv preprint arXiv:2505.10961* (2025). https://doi.org/10.48550/arXiv.2505.10961
[43] Heechan Yang, Ahcheong Lee, Kyutae Cho, and Yunsam Kim. 2026. Industrial Application of Deep Learning based Fault Localization with Mutation Features. In *2026 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 180–190. https://doi.org/10.1109/ICST69053.2026.00032
[44] Jingbo Yang, Bairu Hou, Wei Wei, Yujia Bao, and Shiyu Chang. 2026. Ares: Adaptive Reasoning Effort Selection for Efficient LLM Agents. (2026). https://doi.org/10.48550/arXiv.2603.07915 arXiv:2603.07915 [cs.AI]
[45] Xu Yaozong, Shao Xuebin, Zhou Shuhua, Zhao Qiujun, and Ju Weinan. 2023. Static analysis method of C code based on model checking and defect pattern matching. In *2023 IEEE 5th International Conference on Power, Intelligent Computing and Systems (ICPICS)*. IEEE, 567–573. https://doi.org/10.1109/ICPICS58376.2023.10235566
[46] Recep Yıldırım, Kerem Aydın, and Orçun Çetin. 2024. Evaluating the impact of conventional code analysis against large language models in API vulnerability detection. In *Proceedings of the 2024 European Interdisciplinary Cybersecurity Conference*. 57–64. https://doi.org/10.1145/3655693.3655701
[47] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2025. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 145 (May 2025), 31 pages. https://doi.org/10.1145/3708522
[48] Yuan Zhou, Enze Wang, and Shuoyoucheng Ma. 2025. SSRFSeek: An LLM-based Static Analysis Framework for Detecting SSRF Vulnerabilities in PHP Applications. In *2025 IEEE 6th International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*. IEEE, 939–944. https://doi.org/10.1109/AINIT65432.2025.11035424

Received 2026-05-01; accepted 2026-07-01