Paradigm International Journal of Mathematics and Computer in Engineering 4(2) (2026) 351–372



International Journal of Mathematics and Computer in Engineering

https://reference-global.com/journal/IJMCE

Original Study

A large language model-based analysis of vulnerability discovery in windows software

Puya Pakshad[1†], Samson Quaye[1], Jamal Al-Karaki[2], Marwan Omar[1], Maurice E. Dawson[1]

[1]Department of Information Technology and Management, Illinois Institute of Technology, Chicago, Illinois, 60616, USA

[2]College of Interdisciplinary Studies, Zayed University, Abu Dhabi, P.O. Box: 144534, United Arab Emirates

Communicated by Haci Mehmet Baskonus; Received: 04.03.2026; Accepted: 01.04.2026; Online: 02.06.2026

Abstract

Source code security auditing is essential before software release in order to identify programming faults that may lead to vulnerabilities and functional failures. In this paper, we present a structured security assessment of the `Windows App SDK` by integrating multiple static analysis tools with a context-aware and disagreement-aware Large Language Model (LLM) interpretation layer. Although static analyzers are effective in reporting potential weaknesses, their raw outputs often contain redundant alerts, limited contextual explanation, and inconsistent severity assignments. To address these limitations, the proposed LLM-based interpretation layer normalizes and de-duplicates alerts, filters context-limited or non-actionable warnings, and refines severity prioritization under inter-tool disagreement without introducing new vulnerability discoveries. Experimental results show the security findings before and after LLM-based interpretation. In particular, the proposed framework reduces static-analysis alerts by 62.5%. In addition, disagreement-aware severity refinement eliminates over-prioritized critical findings and improves prioritization by reducing `Medium` findings from 11 to 7 and `Low` findings from 42 to 28. These results demonstrate the potential of LLM-based interpretation to reduce noise in static-analysis outputs and improve vulnerability prioritization for practical security assessment.

Keywords: Software security, static code analysis, large language models, vulnerability discovery, Windows App SDK.
AMS 2020 codes: 68N30; 68T07; 68M12; 94A60.

1 Introduction

Software applications have become essential tools in modern life, providing a wide range of capabilities, user-friendly interfaces, and extensive options for user utilization. These applications are designed to accommodate a diverse range of user requirements, from routine tasks to intricate technical processes. Numerous companies of varying sizes are actively involved in developing and deploying these software solutions to satisfy a wide array of user needs. Among these companies, Microsoft emerges as the leading company in the software industry, which allows developers to create robust and efficient software [1]. Microsoft is widely recognized not only for its user-friendly software products, but also for providing developers with sophisticated tools for software development.

Despite the developer-friendly capabilities of modern Windows development frameworks, large-scale

†Corresponding author.
Email address: ppakshad@hawk.illinoistech.edu

Paradigm
ISSN 2956-7068
doi:10.2478/ijmce-2026-0019
Paradigm
Open Access.© 2026 Pakshad et al., published by Paradigm.
This work is licensed under the Creative Commons Attribution alone 4.0 License.

platform components such as the `Windows App SDK` remain exposed to security vulnerabilities due to their architectural complexity, rapid evolution, and broad adoption. Over time, Microsoft introduced more advanced development tools, the evolutionary progression of which is illustrated in Figure 1. As shown in Figure 1, following years of evolution in Windows software development tools, Microsoft has taken a significant step toward unifying application development with the introduction of the Windows App SDK. This tool represents the apex of the advances that began with Visual Basic and its visual design approach [2, 3]. These advances progressed through WinForms, WPF, WinRT, and UWP, each contributing to graphical interface capabilities and cross-device compatibility. The Windows App SDK now serves as a unified platform, enabling developers to build modern Windows applications using consistent, modular, and up-to-date APIs [4–6]. Despite the developer-friendly capabilities provided by the Windows App SDK, its open-source nature, extensive adoption, and integration into critical applications inherently expose it to security vulnerabilities, exacerbated by its attractiveness as a prime target for exploitation [7]. Historical security records, such as the vulnerability to remote code execution of the `.NET Framework` identified in WPF (CVE-2023-32030: RCE through XAML parsing, CVE-2022-41089: Malicious XPS execution), required substantial updates to address the associated security risks, emphasizing persistent vulnerabilities present in software developed using the Windows App SDK [8–10]. These vulnerabilities can enable malicious adversaries to execute arbitrary code, compromise sensitive systems, disrupt critical services, and raise business and technical impacts for developers and end users alike [11–13]. Such risks show the imperative necessity of source code auditing and comprehensive vulnerability assessments for Windows App SDK software. The execution of this security audit is critical not only to mitigate potential threats, but also to improve reliability among Microsoft users and developers [8, 14–16].

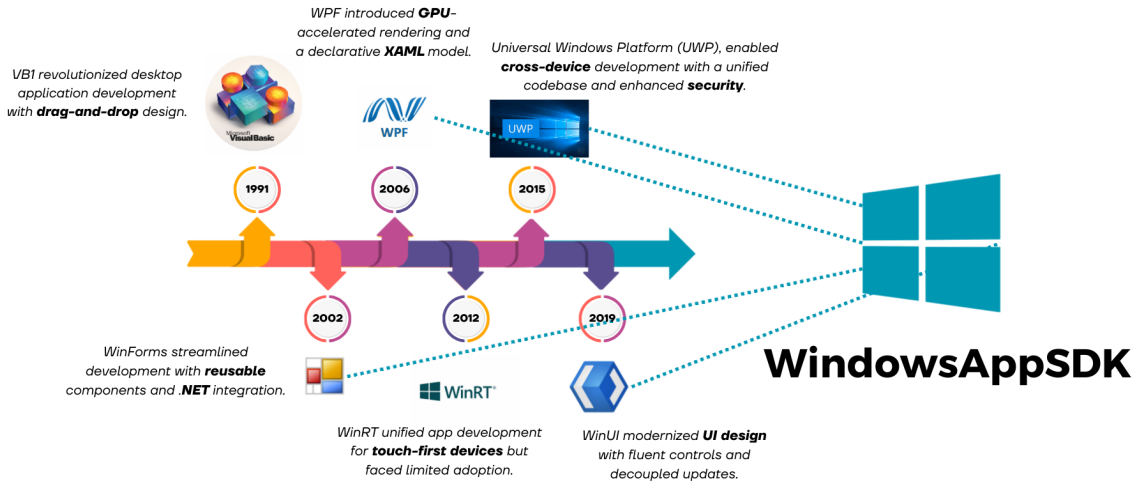

Fig. 1 An historical overview of Windows development frameworks.

To address these concerns, this paper conducts a structured security assessment of the `Windows App SDK` source code using a multi-tool static analysis workflow and a context-aware interpretation layer based on large language models (LLMs). Although static analysis remains an effective first step for vulnerability discovery, raw outputs across multiple analyzers often include substantial redundancy, partially overlapping reports, and coarse severity assignments, which complicate developer decision-making and vulnerability prioritization at scale. Our proposed approach is therefore designed not only to collect findings, but also to transform multi-tool outputs into security insights by consolidating duplicate or unnecessary warnings and improving prioritization under inter-tool disagreement. Specifically, we employ five static analysis tools (open-source and commercial) to maximize coverage, then apply disagreement-aware LLM reasoning to (i) normalize and de-duplicate alerts, (ii) filter non-actionable or context-limited warnings based on code context

and vulnerability characteristics, and (iii) refine prioritization by reinterpreting severity when tool labels are inconsistent or unsupported by contextual evidence. This design is novel in that it treats disagreement between analyzers as an informative signal for prioritization rather than as noise, and it provides an explainable path from raw tool alerts to refined security findings. In line with this goal, we investigate the following research questions: (RQ1) how effectively an LLM can consolidate and filter duplicate or unnecessary warnings across multiple static analysis tools using contextual code understanding, and (RQ2) whether disagreement-aware LLM reasoning can improve prioritization of vulnerabilities reported by static analysis tools. In general, this article makes the following contributions.

- We perform a security analysis of the `Windows App SDK` using multiple static analysis tools to establish a comprehensive baseline of vulnerability findings and severity distributions on a `C/C++` source code.
- We propose and evaluate a context-aware and disagreement-aware LLM-based interpretation layer that normalizes, de-duplicates, and filters security alerts, and improves vulnerability prioritization by refining severity under contextual and inter-tool disagreement.

The remainder of this paper is structured as follows. Section 2 reviews related work on static code analysis, vulnerability detection, and recent efforts to improve security assessment using machine learning and LLMs, showing limitations in existing approaches. Section 3 motivates the selection of the `Windows App SDK` as a case study by analyzing its development characteristics, scale, and evolution, and discussing the challenges that arise in securing large and continuously evolving `C/C++` source code. Section 4 presents the proposed methodology, detailing the multi-tool static analysis workflow and the context-aware, disagreement-aware LLM-based interpretation layer used for alert consolidation and severity refinement. Section 5 reports the experimental results obtained from applying the proposed approach to the `Windows App SDK`, including quantitative analyses of alert consolidation, noise reduction, and severity prioritization. Section 6 discusses the results in relation to the research questions, illustrating the implications of disagreement-aware interpretation for vulnerability prioritization and practical security assessment. Section 7 presents various recommendations. Section 8 outlines the limitations of this paper. Finally, Section 9 concludes the paper.

2 Literature review

The historical evolution of Windows application development reflects Microsofts ongoing attempt to reconcile usability, developer efficiency, extensibility, and security in multiple generations of graphical user interface (GUI) frameworks. This evolution began with the introduction of `Visual Basic 1.0` in 1991, which popularized rapid application development through a drag-and-drop visual programming model that reduced the entry barrier for desktop software development, while providing limited support for scalability and structured application architectures [2]. In response to increasing demands for modularity and reusable components in native Windows software, Microsoft released the `Microsoft Foundation Class Library (MFC)` in 1992, introducing a `C/C++`-based abstraction for window management, event handling, menu systems, and input/output operations. Despite improving structural organization, MFC provided minimal support for modern visual styling and advanced graphical components, constraining its adaptability to evolving user interface requirements [17]. The subsequent introduction of `WinForms` in 2002 as part of the `.NET Framework` represented a notable shift toward improving developer productivity through managed execution, tight integration with `Visual Studio`, and extensive use of visual design tools [18]. Although `WinForms` simplified the development of desktop applications, its dependence on software-based rendering and its limited support for advanced graphics, touch input, and more modern interaction paradigms restricted its suitability for more advanced user interfaces. These shortcomings led to the development of `Windows Presentation`

Foundation (WPF) in 2006, which introduced hardware-accelerated rendering via Direct3D and adopted XAML as a declarative interface definition language.

This architectural shift introduced an explicit separation between presentation concerns and application logic. It also accommodates vector-based graphics, animation support, styling mechanisms, and integrated media functionality, allowing interface design and functional behavior to evolve independently [19, 20]. Despite these improvements, the adoption of WPF in certain enterprise settings remained limited due to hardware dependencies and the added design complexity of its development model, leading many organizations to continue relying on WinForms for less complex applications [19].

As Windows computing environments expanded toward mobile, touch-oriented, and multi-device ecosystems, Microsoft introduced the Windows Runtime (WinRT) alongside Windows 8 in 2012 to provide a unified application programming interface across desktops, tablets, and emerging device classes. However, restrictive API exposure, limited compatibility with legacy Win32 applications, and reliance on a sandboxed execution model impeded enterprise adoption and slowed large-scale migration [6]. Building on this approach, the Universal Windows Platform (UWP) was introduced with Windows 10 in 2015 to further consolidate application development across Windows devices, while incorporating additional security controls such as application sandboxing and constrained API access [21]. In practice, factors including mandatory distribution through the Microsoft Store, reduced access to system-level APIs, and insufficient interoperability with existing Win32 applications continued to limit widespread adoption [22]. To address the fragmentation between legacy desktop software and modern interface requirements, Microsoft evolved WinUI as a modular user interface framework derived from UWP. WinUI allows Fluent Design components, improved responsiveness, and enhanced interoperability with both Win32 and .NET-based applications [4, 5, 22]. Unlike earlier frameworks that were tightly coupled to operating system releases, WinUI supports independent delivery of interface updates, allowing incremental enhancements without reliance on major Windows upgrades. The progression and interdependencies between WinForms, WPF, UWP, and WinUI are summarized in Figure 1, illustrating the architectural convergence that culminated in the introduction of the Windows App SDK. The Windows App SDK embodies Microsofts unification strategy by consolidating the capabilities of preceding frameworks into a modular development platform designed to offer consistent APIs, improved performance at runtime and support for contemporary application design practices [3, 14]. Although this consolidation simplifies development workflows and improves consistency between applications, the increased architectural complexity, open-source development model, and widespread adoption of the Windows App SDK also expand its potential attack surface, making it a high-value target for adversarial exploitation [7, 8]. Historical vulnerability disclosures in closely related Windows frameworks, including remote code execution flaws affecting .NET and WPF components (e.g., CVE-2022-41089 and CVE-2023-32030), demonstrate that even mature and extensively deployed platforms remain exposed to critical security weaknesses. These cases emphasize the need for systematic source code inspection and structured vulnerability assessment methodologies [8–10]. Given the foundational role of the Windows App SDK within the Windows ecosystem, vulnerabilities at this level can propagate across a broad range of dependent applications, amplifying their potential impact.

Previous work has shown that static code analysis is an effective approach for detecting software vulnerabilities and insecure coding practices. In particular, studies report that the combination of multiple analysis techniques such as pattern-based detection, abstract interpretation, symbolic execution, and model checking-leads to more accurate findings while reducing false positive results [23, 24]. Large-scale empirical analyses further reveal that vulnerability prevalence and detection performance vary considerably between programming languages, analysis tools, and configuration settings, highlighting the importance of context-aware and language-specific evaluation strategies [25, 26]. Research on community-driven code reuse, such as insecure C/C++ examples shared on Stack Overflow, illustrates how vulnerabilities can propagate when secure coding practices are not consistently applied [27]. Additional evaluations of static and binary analysis tools for C/C++ systems report significant differences in detection coverage, false positive behavior,

and alignment with established security standards, particularly in large and complex software systems [28, 29]. Further studies have investigated the integration of static analysis techniques into continuous integration environments and large-scale development pipelines, reporting measurable improvements in code quality while also identifying variability in adoption and effectiveness between projects [30, 31]. More recent work has explored the application of machine learning methods and LLMs to improve static analysis by prioritizing alerts, reducing noise, and estimating vulnerability severity using features of the source code and contextual information [32–36].

Table 1 Comparison of research work on bug detection.

Ref.	Semantic Reasoning	Explainability	Hybrid Static + AI	Failure Mode	Evaluation
[37]	✓	×	✓ (LLM)	✓ (shallow reasoning)	High FDR (>50%)
[38]	×	×	✓ (LLM)	✓ (industrial errors)	FP reduced by ≈94–98%
[39]	×	×	×	✓ (count bias)	F1 ≈0.97; Recall <30%
[40]	×	×	×	×	Accuracy ≈0.87; F1 ≈0.86
[41]	✓	×	×	×	Accuracy ≈0.86; F1 ≈0.85
[42]	×	×	×	×	Accuracy ≈0.90; F1 ≈0.91
[43]	✓	×	×	×	Accuracy ≈0.87
[44]	×	✓	×	×	Accuracy ≈91.8%
Our Proposed Model	✓	✓	✓ (LLM)	✓ (tool disagreement)	Alert reduction 62.5%

Recent studies on LLMs have emphasized their potential to support reasoning tasks in software security analysis by allowing semantic interpretation of vulnerability descriptions rather than relying only on syntactic pattern matching. In contrast with traditional machine learning approaches, which typically depend on handcrafted features or statistical code metrics, LLMs can jointly reason over natural language vulnerability reports, source code context, and structured security knowledge such as `CWE` taxonomies to assess vulnerability relevance and severity. Previous work shows that LLMs can assist in filtering non-security findings, prioritizing alerts, and providing explanatory context for static analysis results, thus reducing alert overload and supporting more effective analyst decision-making [32–34].

Despite these advances, most recent studies evaluate LLMs either as standalone vulnerability detection systems or as post-processing components applied to the output of a single static analysis tool. Consequently, several critical challenges in large scale security assessment-most notably cross-tool alert redundancy, inter-tool disagreement, and explainable consolidation of findings remain largely unaddressed. Table 1 provides a structured comparison of recent work in vulnerability and bug detection, including representative studies [37–43]. Although these approaches demonstrate strong quantitative performance in terms of accuracy, F1 score, or false positive reduction, they differ substantially in their handling of semantic reasoning, explainability, hybrid static-AI integration, and analysis of failure modes. Li et al. [37] study LLM-based vulnerability detection at the project level and report high false discovery rates despite incorporating semantic reasoning, highlighting the difficulty of maintaining precision in large and complex code bases. Du et al. [38] focus on false-positive reduction in industrial settings and achieve significant improvements in alert filtering; however, their approach does not explicitly model disagreement in analysis tools. Pushkar et al. [39] identify recall degradation in multi-vulnerability scenarios and characterize count bias as a major failure mode. Park et al. [40] and Sultana et al. [43] evaluate instruction-tuned and fine-tuned LLMs as classification-based detectors, but do not integrate these models within static analysis pipelines. Similarly, Saju et al. [41] and Cao et al. [42] report improved accuracy and F1 scores through prompt engineering and preprocessing techniques, yet their methods operate independently of multi-tool static analysis workflows. Haurogné et al. [44] introduce explainability mechanisms within a deep learning framework, but the approach remains limited to supervised classification rather than hybrid vulnerability consolidation. Collectively, existing research tends to focus on isolated aspects of vulnerability analysis, such as improving detection performance or reducing false positives

within single-tool pipelines. As summarized in Table 1, to the best of our knowledge, no previous research study simultaneously integrates (i) hybrid static and LLM-based analysis across multiple tools, (ii) disagreement-aware semantic reasoning, and (iii) explainable consolidation of findings for large-scale `C/C++` source code. From this point of view, the `Windows App SDK` provides a challenging case study as a platform-level framework under test. In this paper, we propose a framework that goes beyond isolated detection or filtering tasks by integrating multiple static analysis tools with a disagreement-aware LLM-based reasoning layer to support structured vulnerability consolidation, severity refinement, and more explainable security reporting.

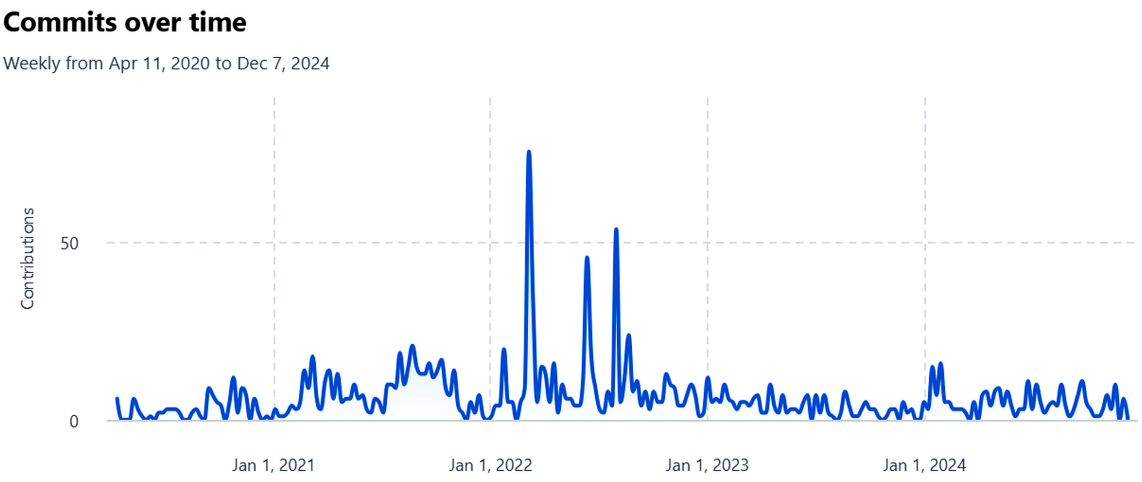

Fig. 2 The analysis of the commit frequency over time.

3 Motivation

The primary motivation for analyzing the Windows App SDK arises from its central role as a core platform for the development of modern Windows desktop applications. As a comprehensive and evolving framework, the Windows App SDK allows developers to build scalable, feature-rich, and cross-platform applications that are widely deployed in Windows platforms. Given its broad adoption and continuous evolution, ensuring the reliability of this software framework is of critical importance. Figure 2 illustrates the commit activity of the Windows App SDK project from April 2020 to December 2024, highlighting temporal trends in development intensity. The figure reveals a substantial surge in developer contributions during 2022, with weekly commit counts exceeding 70 during peak periods. This increase reflects a phase of accelerated development, likely associated with the introduction of new features, architectural enhancements, or significant refactoring efforts. In contrast, the gradual decline in commit frequency observed in 2023 and 2024 suggests a transition toward a stabilization phase, where development efforts increasingly focused on debugging, optimization, and maintenance of existing functionalities. Such alternating phases of rapid development and stabilization are characteristic of large-scale, high-impact software projects supported by active developer communities.

Figure 3, derived from publicly available data from the Microsoft GitHub repository, further illustrates trends in code churn by presenting the volume of code additions and deletions over time. The data indicate a pronounced spike in code additions during 2021, exceeding 100,000 lines within a relatively short period, accompanied by substantial deletions. This pattern suggests extensive development activity, potentially involving major integrations, redesigns, or architectural refactoring. In subsequent years, particularly during 2023 and 2024, the magnitude of code changes becomes less pronounced, indicating a shift toward a more maintenance-oriented development cycle. Together, Figures 2 and 3 demonstrate the inherent variability and

complexity of the Windows App SDK source code over time. Such large-scale and continuous code evolution inherently increases the risk of inadvertently introducing security vulnerabilities during development iterations. Periods of intense development activity, especially those involving rapid feature expansion or structural changes, are particularly susceptible to security regressions and implementation flaws. At the same time, the sustained engagement of the developer community and the regular release cadence underscore the critical role of the Windows App SDK within the broader Windows application ecosystem. These factors collectively motivate the need for systematic and scalable security analysis techniques capable of identifying potential vulnerabilities in evolving versions of the codebase.

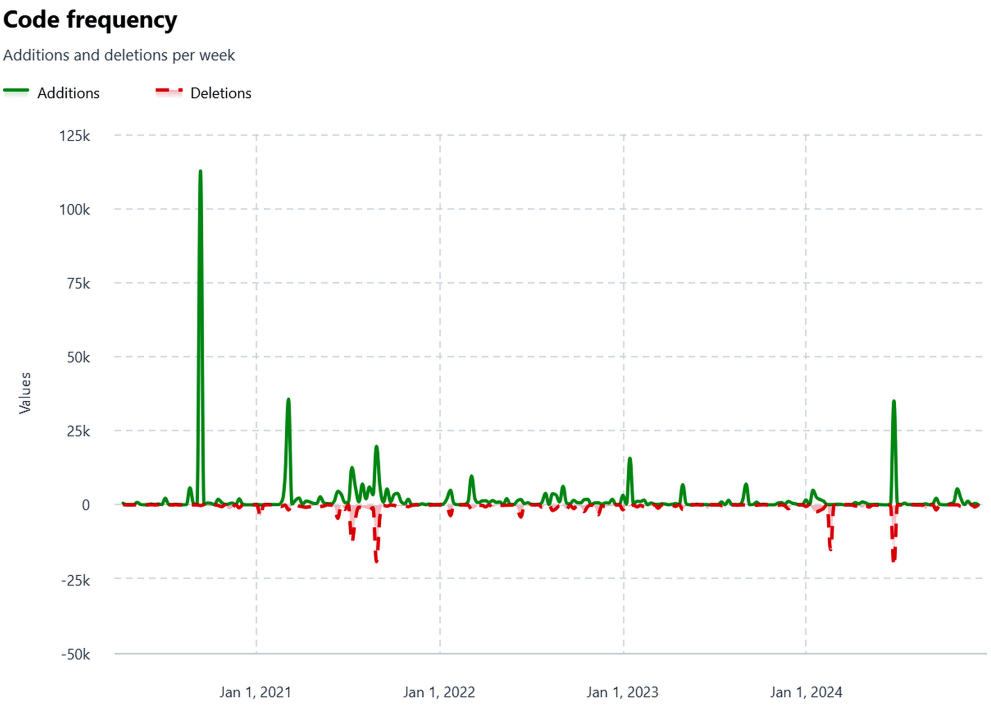

Fig. 3 The code frequency over time.

Although static code analysis provides an effective means to detect a wide range of potential security weaknesses, the scale and complexity of the Windows App SDK pose challenges related to alert volume, contextual interpretation, and vulnerability prioritization. These challenges motivate the exploration of enhanced analysis workflows that combine traditional static analysis with higher-level reasoning capabilities. Consequently, this work is motivated by the need to augment static analysis results with contextual interpretation mechanisms, allowing a more meaningful assessment of security findings and supporting informed security decision-making for large software frameworks. These observations motivate the need for a security analysis methodology that not only detects potential vulnerabilities at scale, but also supports meaningful interpretation and prioritization of security findings to help developers focus remediation efforts on security-relevant issues, reduce analysis overhead caused by noisy alerts, and make informed security decisions within large and continuously evolving source code. In the following section, we present a proposed methodology that integrates multiple static code analysis tools with a LLM to improve vulnerability interpretation, severity refinement, and security reporting for the Windows App SDK.

4 Proposed methodology

The proposed methodology aims to address a key limitation of static code analysis workflows, the lack of contextual interpretation, and prioritization of security findings. Although static analysis tools are effective at identifying potential weaknesses in source code, their output often consists of large volumes of alerts with limited semantic explanation and coarse-grained severity labels.

As illustrated in Figure 4, the proposed architecture integrates multiple static code analysis tools with a LLM reasoning layer to transform raw vulnerability alerts into security insights. The input of the proposed architecture is the Windows App SDK project, which consists of a large C/C++ source codebase. These C/C++ source files are processed by several static code analysis tools, which independently analyze the codebase and generate raw vulnerability alerts characterized by tool-specific formats, severity labels, and descriptive messages. The resulting alerts are subsequently aggregated, normalized, and de-duplicated to form a set of security findings that preserve detection coverage. Rather than replacing existing severity scoring mechanisms, the LLM performs contextual reasoning on code snippets, vulnerability descriptions, and CWE-related knowledge to assess security relevance and refine tool-reported severity levels, allowing for more informed vulnerability prioritization. The output of the proposed architecture is a security report that includes vulnerability findings, refined severity interpretations, CWE distribution analysis, prioritized remediation advice, and an estimate of alert noise and false-positive reduction.

In the following, we first describe how vulnerability-related information is collected through multiple static code analysis tools, including the types of findings reported and the metadata extracted from each tool. We then explain how these security findings are aggregated, normalized, and prepared as structured inputs for the LLM. Subsequently, we detail how the vulnerability data, together with the designed prompts, are injected into the LLM to perform contextual reasoning, severity refinement, and vulnerability interpretation.

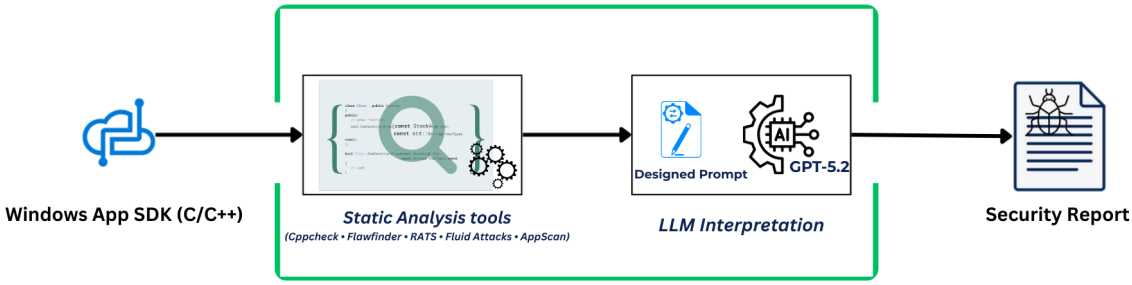

Fig. 4 Proposed architecture.

4.1 Static code analysis tools

The first phase of the proposed methodology consists of selecting static code analysis tools to perform a security assessment of potential vulnerabilities in the C/C++ source code of the Windows App SDK. This phase corresponds to the static analysis stage of the proposed architecture illustrated in Figure 4 and Algorithm 1 (Lines 1–4), where the source code is independently analyzed by multiple static code analyzers before any higher-level interpretation. The effectiveness of static analysis tools in detecting software security weaknesses has been analyzed in previous research. For example, Kaur and Nayyar [45] conducted a comparative study on various static code analysis tools, evaluating their efficiency in identifying vulnerabilities within the C/C++ and Java source code. Their findings show both the strengths and limitations of different tools, providing a foundational basis for the tool selection process adopted in this research study.

To ensure a well-rounded evaluation and broad detection coverage, we utilize a combination of open-source and commercial static analysis tools, including Flawfinder, Cppcheck, RATS, FluidAttacks Tool, and AppScan Static Analyzer. The selection criteria were informed by previous research, particularly the comparison

conducted by Fatima et al. [46], which evaluated sixteen static code analysis tools based on detection precision, false positive rates, and adherence to secure coding standards. These insights reinforced our methodology and guided the choice of tools that best align with the objectives of this research study. Each selected tool was chosen for its specific strengths in identifying security vulnerabilities in C/C++ code, thus ensuring a diverse and robust static analysis stage.

Flawfinder was used to identify potential security vulnerabilities by scanning the source code for patterns associated with known weaknesses and insecure coding practices [47]. A key strength of Flawfinder lies in its ability to assign severity levels to detected issues, allowing initial prioritization of security risks. In this research study, Flawfinder served as a valuable tool for identifying potentially dangerous function usages and common C/C++ weaknesses within the Windows App SDK codebase. Cppcheck, another static analysis tool, was used to identify potential bugs, memory management issues, and undefined behaviors in the C/C++ code [48]. Unlike traditional linters, Cppcheck emphasizes the adherence to coding standards, which facilitates the detection of logical errors and code quality issues that may indirectly impact software security.

RATS, known as the Rough Auditing Tool for Security, was also used to complement the analysis by focusing on identifying common insecure coding patterns in source code [49]. RATS specializes in detecting vulnerabilities such as buffer overflows and improper input handling, and generates reports that reference relevant Common Weakness Enumerations (CWEs), allowing developers to better understand and address the root causes of detected security issues [7]. To further enhance the depth of vulnerability detection, commercial static analysis tools were integrated into the analysis process. The FluidAttacks Tool was used to identify more complex vulnerabilities in the C/C++ source code by using advanced scanning algorithms and a comprehensive database of known security flaws [50]. This tool provides detailed vulnerability descriptions, remediation suggestions, and compliance checks with industry security standards. Similarly, the AppScan Static Analyzer was applied to perform a detailed static analysis of the Windows App SDK codebase [51]. AppScan is known for its scalability and precision in identifying a wide range of security vulnerabilities, from common coding flaws to more sophisticated security issues. The tool generates detailed reports that categorize vulnerabilities by severity level, facilitating subsequent aggregation and analysis within the proposed methodology. Although other commercial tools, such as Checkmarx and Veracode, could have been considered, their high licensing costs prevented their use in this research study [52, 53].

4.2 LLM interpretation

The input to LLMs consists of vulnerability findings extracted from multiple static code analyzers. For each finding, the input provided includes the affected source file and the corresponding code snippet, the vulnerability description generated by the security tool, the reported severity level, and any associated Common Weakness Enumeration (CWE) identifiers when available. This information supplies the LLM with both syntactic and semantic context, allowing reasoning beyond pattern-based detection and supporting vulnerability interpretation. As shown in Line 5 of Algorithm 1, vulnerability alerts generated by different static analysis tools are first normalized and de-duplicated. This step addresses the inherent heterogeneity of tool outputs, as multiple analyzers can report the same issues using different formats, naming conventions, or severity scales. Normalization ensures a consistent representation of the findings by aligning metadata such as file paths, line numbers, vulnerability categories, and severity labels. De-duplication further reduces redundant alerts that refer to the same code location or weakness, which prevents over-representation and allows more effective downstream interpretation by the LLMs.

Following normalization, each security finding is processed by the LLM in conjunction with a structured prompt, as described from Line 6 to Line 10 of Algorithm 1. For each finding, a prompt (Line 7) is constructed that integrates the relevant code snippet, the explanation of the static analysis tool, and CWE-related background knowledge. The LLM is then invoked on this prompt (Line 8) to perform contextual reasoning on the security information provided. Based on this reasoning, the LLM refines the finding by assessing its security relevance and adjusting or qualifying the severity level reported by the static code analyzers when

Algorithm 1 LLM-based vulnerability interpretation and prioritization.

Require: Source code SC, static analyzers $\mathscr{SA}$, LLM $\mathscr{AI}$
Ensure: Security report R
1: $A \leftarrow \emptyset$
2: **for all** $sa \in \mathscr{SA}$ **do**
3: $\quad A \leftarrow A \cup \text{RunTool}(sa, SC)$
4: **end for**
5: $N \leftarrow \text{NormalizeAndDeduplicate}(A)$
6: **for all** $f \in N$ **do**
7: $\quad x \leftarrow \text{BuildPrompt}(f)$
8: $\quad y \leftarrow \mathscr{AI}(x)$
9: $\quad f \leftarrow \text{RefineFinding}(f, y)$
10: **end for**
11: $R \leftarrow \text{GenerateReport}(N)$
12: **return** R

appropriate (Line 9). This refinement process allows the methodology to distinguish security-critical vulnerabilities from context-dependent issues and likely non-security findings, thus improving vulnerability prioritization without replacing severity scoring frameworks. After all findings have been processed by the LLM, the results are integrated into a security report, as shown in Line 11 of Algorithm 1. The generated report summarizes the vulnerabilities analyzed, incorporates refined severity interpretations, and presents the distribution of the common weakness enumerations detected. In addition, the report provides prioritized remediation recommendations and an estimate of alert noise and false-positive reduction achieved through LLM-based interpretation. This final output is intended to support developers and security analysts by allowing informed security decision-making, more reliable vulnerability prioritization, and scalable remediation planning for large C/C++ source code.

4.3 Summary

In summary, the proposed methodology combines multi-tool static code analysis with a reasoning layer based on LLMs to address structural limitations in conventional vulnerability assessment workflows for `C/C++` source code. Although static analysis tools provide complementary detection capabilities, their output often contains significant redundancy, inconsistent severity ratings, and syntactically different descriptions of the same underlying weakness when multiple tools are used. The proposed architecture leverages this heterogeneity by aggregating and normalizing findings in all selected tools and treating inter-tool disagreement as an informative signal rather than as noise. In addition, the LLM-based interpretation layer operates on these normalized findings and performs context-aware reasoning on source code snippets, vulnerability descriptions, and CWE semantics. This process allows for effective consolidation of duplicate alerts and refinement of tool-reported severity levels. In contrast to prior approaches that apply isolated post-processing or focus on single-tool filtering, the proposed methodology supports disagreement-aware interpretation, allowing the model to reason about vulnerability prioritization across multiple analyzers. Algorithm 1 formalizes this workflow and illustrates how raw static analysis outputs are transformed into refined security findings through structured prompting and iterative reasoning. As a result, the methodology generates security reports that extend beyond the simple enumeration of detected issues by incorporating severity prioritization and reduced alert noise. This design helps developers make better security decisions by making the results of static code analysis easier to understand and more useful in practice, especially for large and continuously evolving `C/C++` code bases such as the `Windows App SDK`. In general, the proposed approach provides a basis for evaluating how LLM-based interpretation can improve vulnerability consolidation and prioritization.

5 Experimental results

This section presents the experimental results obtained by applying our proposed methodology to the `Windows App SDK`. The results are organized into three subsections. First, the experimental setup describes the software under analysis, the execution environment, and the static analysis tools used in this paper. Second, the findings of the static code analysis are reported, summarizing the vulnerabilities and code quality issues identified by different analyzers. Lastly, the LLM component shows how LLM-driven contextual reasoning refines vulnerability severity, reduces alert noise, and generates security reports aligned with the research questions of this paper.

Table 2 Project details.

Metric	ID	Metric Value
Application Name	AN	Windows App SDK 1.6.2
Review Date	RD	December 12, 2025
Objective	OBJ	Security Code Review
Number of Lines (LOC)	LOC	167,894
Code Review Mode	CRM	Static

5.1 Research questions

This paper explores how LLMs can improve the interpretability and practical usefulness of static code analysis results for large-scale `C/C++` software. Although static analysis tools have long been used to identify security vulnerabilities, their outputs often include a high volume of redundant alerts and provide limited contextual explanation. These limitations make it difficult for security analysts to effectively interpret the findings and prioritize remediation efforts. To address these issues, this article forms the following research questions.

- **RQ1:** *How effectively can an LLM consolidate and filter duplicate or unnecessary warnings across multiple static analysis tools using contextual code understanding?*
- **RQ2:** *Can disagreement-aware LLM reasoning improve prioritization of vulnerabilities in static analysis tools?*

These research questions are driven by three key challenges that have not yet been adequately addressed in prior studies. First, existing research typically evaluates static analyzers and LLM-based detectors in isolation or applies LLMs as a post-processing step to the output of a single tool. As a result, the problem of alert redundancy and inconsistency across multiple tools has received limited attention. In large-scale `C/C++` software systems, different static analysis tools often report partially overlapping or syntactically different findings for the same underlying vulnerability, yet a principled mechanism for consolidating such results is still lacking. Second, although recent efforts have focused on reducing false positives, most approaches frame vulnerability filtering as an independent classification task. This perspective overlooks inter-tool disagreement as a potentially informative signal. The lack of consensus-aware reasoning restricts the ability to assess the reliability of reported findings. Third, both traditional static analysis tools and many existing LLM-based approaches offer limited transparency into how vulnerability-related decisions are derived. This absence of explainable consolidation undermines analyst trust and poses a barrier to practical adoption in real-world security workflows.

To address these challenges, the next section describes the experimental design of the proposed framework. The study is conducted in two main phases. In the first phase, multiple static code analysis tools are applied to

the Windows App SDK source code to collect security findings. In the second phase, a disagreement-aware LLM-based reasoning layer is introduced to consolidate, filter, and interpret the output that allows structured and prioritized vulnerability reporting.

5.2 Setup

We first describe the experimental setup environment used to evaluate the vulnerability analysis methodology proposed in the Windows App SDK version 1.6.2. Table 2 provides an overview of the analyzed project. The source code of the software under test is publicly available through the official GitHub repository [54]. In addition, all raw analysis reports generated by the static analysis tools used in this study have been made publicly available in our accompanying GitHub repository [55] to support transparency and reproducibility.

The experiments were conducted on a system running Windows 11 (64-bit) equipped with an Intel(R) Core(TM) i7-10510U CPU operating at a base frequency of 1.80 GHz with a maximum turbo frequency of 2.30 GHz. A virtualized Kali Linux 2024 environment was deployed using VMware Workstation 17.5 to execute several static analysis tools that require a Linux-based runtime. The primary open-source static analyzer employed was Cppcheck 2.16 [48], selected for its ability to detect coding errors, undefined behaviors, and security-relevant issues in C/C++ systems. Additionally, Flawfinder 2.0.19 [47] was executed in both Python 2.7 and Python 3.13 environments, producing consistent results in all configurations. The Rough Auditing Tool for Security (RATS 2.4) [49] was also utilized within the Kali Linux virtual machine. To complement open-source analysis, two commercial static analysis tools were integrated into the experimental setup: the AppScan Static Analyzer provided via AppScan on Cloud [51] and a commercial static analyzer from Fluid Attacks [50]. Due to licensing constraints, evaluation licenses were used for both tools. Each commercial analysis was completed in approximately 4 minutes and 30 seconds, producing detailed vulnerability reports that were later used as input for the interpretation based on the LLM.

Table 3 Severity distribution before and after LLM-based interpretation (Windows App SDK, C/C++).

Severity Level	Findings (SA)	Findings (LLM-Based)	Main Vulnerability Categories	Static Analysis Tools
5 (Critical)	1	0	Privilege escalation (baseline highest-risk item)	AppScan Static Analyzer [51]
4 (High)	1	2	Command injection; reclassified critical item (context-limited)	AppScan Static Analyzer [51]
3 (Medium)	11	7	Improper resource access control; permission/validation warnings	Flawfinder; AppScan Static Analyzer [47, 51]
2 (Low)	42	28	Information exposure; input validation; dependency integrity; API pattern alerts	AppScan; Fluid Attacks; Cppcheck; RATS [48–50]

5.3 Findings of static code analysis

We report the findings obtained from the static code analysis tools before any LLM-based interpretation. The goal of this phase is to characterize the raw vulnerability landscape of the Windows App SDK as identified by different analyzers and to establish a baseline for subsequent interpretation. To ensure complementary coverage and meaningful inter-tool disagreement, we leveraged five static analysis tools with distinct detection philosophies: Cppcheck for semantic bug detection and code correctness, Flawfinder and RATS for pattern-based identification of insecure C/C++ constructs aligned with CWE categories, and AppScan Static Analyzer and Fluid Attacks for rule-based and compliance-oriented detection of security-critical vulnerabilities, including access control, injection, and supply chain risks. This tool selection allows systematic observation of alert overlap, inconsistency, and severity disagreement across analyzers, which is essential to evaluate LLM-based consolidation, contextual interpretation, and prioritization in the

subsequent phase. The aggregated severity distribution obtained from this static analysis phase is summarized in Table 3, which reports the number of findings per severity level before any LLM-based interpretation. In addition, the values reported in the *Severity (SA)* column of Table 3 are derived directly from the tool outputs described in this subsection after normalization to a unified severity scale.

The `Cppcheck` analysis of the `Windows App SDK`, while not uncovering critical vulnerabilities, identified several code quality issues and optimization opportunities. These include missing header files, such as `<pch.h>` and `<Windows.h>`, suboptimal coding practices, such as replacing raw loops with `STL` algorithms like `std::transform` or `std::find_if` to improve code readability, and performance inefficiencies, including declaring variables and parameters as `const` where appropriate and removing redundant `c_str()` calls on `std::wstring`. Syntax errors, including improper use of reserved keywords, such as `try` in the global scope and undefined macros, such as `CATCH_RETURN()`, were also flagged for correction. Although no explicit security vulnerabilities were identified, the tool provided valuable feedback to improve code quality, improve performance, and resolve configuration issues that can affect code maintainability and performance [48].

The `RATS` analysis of the `Windows App SDK` revealed several potential security concerns in the source code examined. Although no critical vulnerabilities were discovered, the tool flagged numerous warnings indicative of potentially insecure coding practices. These warnings were primarily related to common weaknesses, such as improper handling of sensitive tokens, potential buffer overflows, and inadequate input validation mechanisms [7, 49]. Although flagged constructs were not immediately exploitable, they pointed to areas of the codebase that required improved adherence to secure coding standards. The analysis specifically highlighted certain code patterns that warrant closer inspection to ensure alignment with security best practices.

The `Flawfinder` analysis of the `Windows App SDK` project identified multiple potential security vulnerabilities, classified according to the Common Weakness Enumeration (CWE) standard. Specific CWEs identified include CWE-120 (Buffer Copy without Checking Size of Input), which highlights the use of unsafe functions such as `strcpy` that can lead to buffer overflows. CWE-785 (Use of the Path Manipulation Function without a Maximum-sized Buffer) was also flagged, indicating a lack of size checks in file path manipulation, potentially leading to buffer overflows or path traversal attacks. Additionally, the analysis identified CWE-134 (Uncontrolled Format String), which arises from the improper formatting of user-controlled input in functions like `printf`, potentially leading to memory leaks or arbitrary code execution. Other significant CWEs detected include CWE-78 (Improper Neutralization of Special Elements in OS Command), which can lead to command injection attacks, and CWE-242 (Use of Inherently Dangerous Function), which highlights the risks associated with functions like `gets` that lack bounds checking. Furthermore, the analysis revealed potential race conditions due to CWE-362 (Continual Execution using Shared Resource with Improper Synchronization), which can lead to unpredictable program behavior or security breaches in multithreaded environments. Finally, CWE-476 (NULL Pointer Dereference) was identified, indicating code sections that can lead to crashes or denial of service conditions if null pointers are dereferenced [47].

The `AppScan Static Analyzer` identified critical vulnerabilities in the `Windows App SDK`, which demonstrate the urgent need to improve the software's security. A significant finding was a vulnerability to privilege escalation in `Test_WinRT_Add_Rank_B-10_A0.cpp` at line 59, where inadequate access control mechanisms could potentially grant elevated privileges to unauthorized entities. Furthermore, the analysis revealed instances of improper resource access control, highlighting insufficient permission checks on sensitive resources, which could lead to unauthorized access or manipulation. This tool also highlighted issues related to sensitive information exposure, where critical data, such as credentials or configuration files, were found to be stored or transmitted insecurely. Furthermore, potential command injection vulnerabilities were identified in dynamically constructed shell commands, which pose a significant risk of execution of arbitrary code [51]. In addition, several cases of insufficient input validation were detected, ranging in severity from low to medium. These vulnerabilities could potentially lead to SQL injection, cross-site scripting (XSS), or other unexpected application behaviors. The findings show the necessity of careful security auditing, which is crucial

to effectively mitigate the identified risks in the source code of the Windows App SDK software. These findings illustrate the diversity, volume, and heterogeneity of alerts generated by static analysis tools when applied to the targeted C/C++ source code. Although the applied analyzers are effective in identifying potential weaknesses, their raw outputs exhibit substantial alert redundancy, inconsistent severity assignments across tools, and limited contextual explanation of security relevance. In particular, partially overlapping findings and disagreement in severity labeling complicate vulnerability prioritization. These observations motivate the need for a context-aware and disagreement-sensitive interpretation mechanism, which is addressed through the LLM based analysis presented in the following subsection.

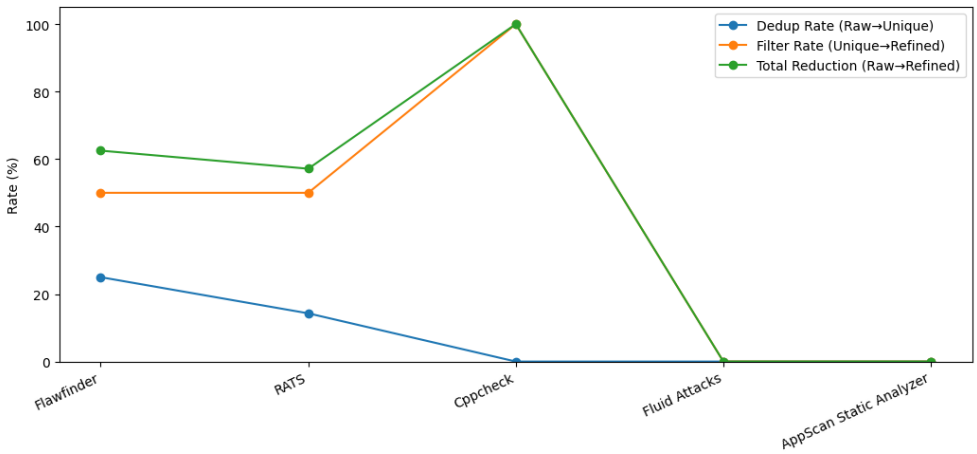

Fig. 5 Reduction rates.

5.4 LLM-based interpretation

To address the limitations observed in the results of the static analysis, the output of all static analyzers was provided as input to a LLM for contextual interpretation and refinement. In this experiment, the LLM component of the proposed methodology was implemented using ChatGPT 5.2, accessible through an API-based integration. A custom Python script was developed to automate the ingestion of static analysis reports, normalize and de-duplicate vulnerability findings, and construct structured security prompts for LLM processing. Each prompt included up to 100 lines of contextual information, combining the relevant C/C++ code snippet, the tool-generated vulnerability description, the reported severity level, and associated CWE identifiers. This prompt design allows the LLM to reason about vulnerabilities in relation to their surrounding implementation context rather than treating alerts as isolated pattern matches. The LLM was instructed to assess security relevance, qualify or refine severity levels, and identify likely false positives or context-dependent findings. Importantly, the LLM does not introduce new vulnerability findings and does not replace static analysis tools; instead, it operates as an interpretation and reasoning layer on existing tool outputs. This design ensures traceability between refined findings and their origin analyzers while allowing structured reasoning over alert redundancy, severity disagreement, and contextual exploitability.

In general, our LLM-based interpretation generated three primary outcomes aligned with the research questions in this article. First, it reduced alert noise by identifying redundant and non-security-relevant findings, particularly among low-severity issues reported by multiple tools (RQ1). Second, it refined severity interpretation by distinguishing security-critical vulnerabilities from code quality warnings that lack exploitability in practice (RQ2). Third, it generated an integrated security report that combines vulnerability summaries, CWE distributions, and prioritized remediation recommendations tailored to the architectural context of the Windows App SDK. Compared to the raw static analysis outputs summarized in Table 3, the LLM-enhanced reports provided clearer prioritization and improved interpretability for developers. Rather than

presenting isolated alerts, the final reports contextualize vulnerabilities within the broader source code, indicating the most impactful risks, and support informed remediation planning. These results demonstrate that LLM based interpretation can effectively complement static analysis by transforming large volumes of findings into security insights suitable for real-world software platforms.

6 Discussion

This section discusses the experimental findings in light of the research questions and illustrates the implications of integrating reasoning based on LLMs with static code analysis for large-scale `C/C++` source code. Rather than focusing on vulnerability discovery, the discussion emphasizes how LLM-based interpretation improves the usability, prioritization, and reliability of static analysis output by addressing alert redundancy, severity inconsistency, and limited contextual explanation.

6.1 Alert consolidation and noise reduction

To quantitatively evaluate alert consolidation, the alert reduction metric defined in Equation 1 is used to measure the proportion of raw static analysis alerts eliminated after LLM-based interpretation. This metric captures the combined effect of alert de-duplication across code locations and semantic filtering based on contextual vulnerability assessment.

Table 4 reports the consolidation results for all five static analysis tools. In general, the LLM reduced the total number of alerts from 24 raw findings to 9 refined findings, corresponding to an aggregate alert reduction rate of 62.5%. This reduction reflects both the elimination of duplicate reports and the removal of context-dependent warnings that do not represent security vulnerabilities. A more detailed breakdown of the consolidation process is illustrated in Figure 5, which separates alert reduction into three components: de-duplication from raw alerts to unique code locations, filtering from unique locations to LLM-refined findings, and total reduction from raw to refined alerts. For `Flawfinder`, 25.0% of alerts were removed during de-duplication, followed by a 50.0% filtering rate during LLM interpretation, resulting in a total reduction of 62.5%. Similarly, `RATS` exhibited a 14.3% de-duplication rate and a 50.0% filtering rate, yielding an overall reduction of 57.1%. The most pronounced effect is observed for `Cppcheck`, where no alerts were eliminated during de-duplication, but all six unique findings were filtered out by the LLM based on contextual analysis, leading to a 100.0% total reduction. This indicates that the reported issues were primarily related to configuration or code-quality artifacts rather than security-relevant vulnerabilities. In contrast, `Fluid Attacks` and `the AppScan Static Analyzer` showed no reduction, with total reduction rates of 0.0%, indicating that all LLM retained all reported findings as relevant to security.

$$\text{Reduction}(\%) = \frac{N_{\text{raw}} - N_{\text{refined}}}{N_{\text{raw}}} \times 100. \tag{1}$$

These results demonstrate that LLM-based interpretation selectively reduces alert noise where there is redundancy or weak security evidence, while preserving high-signal findings generated by tools with stronger contextual accuracy. Importantly, as shown in Figure 5, the majority of alert reduction arises from semantic filtering rather than simple de-duplication, showing the role of contextual reasoning in distinguishing vulnerabilities from pattern-based warnings. Collectively, the quantitative evidence provided by Table 4 and Figure 5 provides a solid answer to RQ1.

Responding this situation, across all analyzers, LLM reduced **24 raw alerts** to **21 unique code locations** via normalization and de-duplication. In addition, LLM achieved substantial reductions for pattern-based analyzers, including **Flawfinder** with a **62.5% reduction (8→3)** and **RATS** with a **57.1% reduction (7→3)**, where multiple alerts corresponded to overlapping or context-dependent code patterns. For **Cppcheck**, all reported alerts filtered out (**100% reduction, 6→0**) reflected configuration or analysis artifacts rather than

security vulnerabilities. In contrast, no reduction was observed for **Fluid Attacks (2→2)** and **AppScan Static Analyzer (1→1)**, suggesting that LLM preserved high-signal findings when sufficient contextual and security-relevant evidence existed. These results confirm that LLM does not indiscriminately suppress alerts; instead, it selectively consolidates and filters findings based on semantic reasoning over code context, tool descriptions, and vulnerability characteristics. Consequently, LLM effectively reduces alert noise while maintaining coverage of meaningful security issues.

Table 4 Effectiveness of LLM-based alert consolidation across static analysis tools.

SA Tool	Raw Alerts	Unique Code Locations	LLM-Refined Findings	Alert Reduction (%)
Flawfinder	8	6	3	62.5%
RATS	7	6	3	57.1%
Cppcheck	6	6	0	100.0%
Fluid Attacks	2	2	2	0.0%
AppScan Static Analyzer	1	1	1	0.0%
Total	**24**	**21**	**9**	**62.5%**

6.2 Severity prioritization

Beyond alert consolidation, this paper evaluates whether *disagreement-aware* LLM reasoning can improve vulnerability prioritization when static analyzers provide coarse-grained severity labels. In practice, static analyzers frequently assign conservative severities based on syntactic patterns and rule-based heuristics, while providing limited evidence of exploitability, production reachability, or operational impact of a reported issue. When multiple analyzers are used, these limitations are amplified: different tools may (i) report the same underlying weakness with different wording, (ii) disagree on severity for similar alerts, or (iii) flag security-adjacent code-quality patterns that are not vulnerabilities. This makes severity-driven remediation ordering difficult for developers and can result in over-prioritization of context-limited alerts.

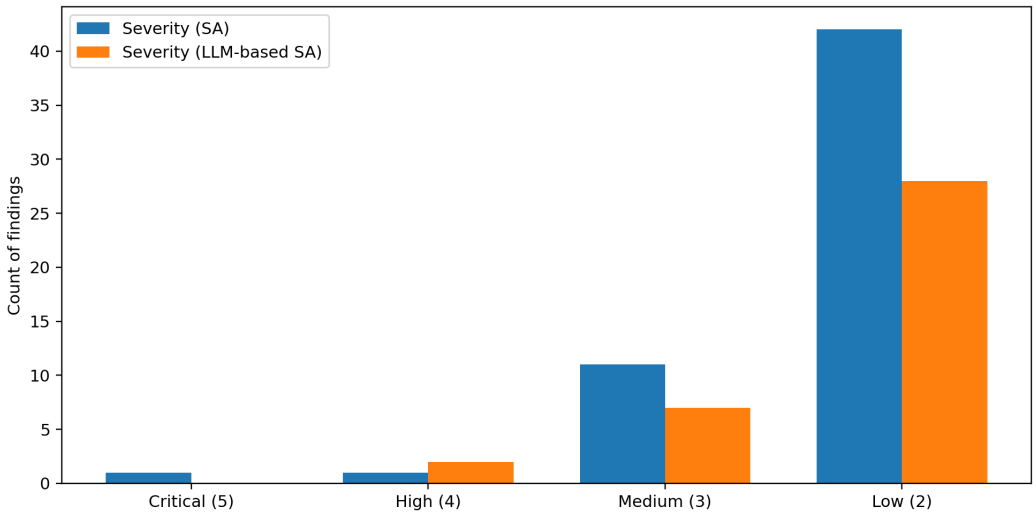

Fig. 6 Severity distribution before vs after disagreement-aware LLM interpretation.

To quantitatively illustrate the changes in prioritization, Figure 6 compares the severity distribution produced by the aggregated static-analysis baseline (SA) with the refined distribution after applying the proposed interpretation based on LLM. The most visible change is the reduction of `Critical` findings from 1 to 0, accompanied by an increase in `High` findings from 1 to 2. This shift does not dismiss a serious security concern; instead, it reflects a disagreement-aware contextual reinterpretation of the highest-priority baseline item. In particular, the corresponding privilege-escalation report originates from a single analyzer and appears in a test-oriented code location, where the available evidence does not support production reachability or exploit-triggering dataflow. Under our reasoning rules, such a context-limited, single-source critical classification is downgraded to `High` to prevent over-prioritization above production-relevant high-impact risks, while still keeping the issue visible for review.

In addition to the Critical-to-High reclassification, the number of `Medium` findings decreases from 11 to 7, and `Low` findings are reduced from 42 to 28. These reductions arise from two complementary mechanisms that improve prioritization without suppressing tool coverage. First, overlapping alerts reported by different tools are consolidated into a single security finding when they refer to the same code location or the same weakness class. Second, a subset of pattern-based warnings—including API-name-driven memory/string usage alerts and best-practice compliance notes—are reclassified as non-security or non-actionable when the provided context lacks supporting evidence for exploitability (e.g., no indication of attacker-controlled inputs, no demonstrated unsafe size relationships, or no execution-critical reachability). In other words, the LLM does not remove raw alerts from the dataset; rather, it refines the *security set* by separating true vulnerabilities from context-dependent signals and redundant warnings. This distinction is essential because the main challenge is not generating alerts, but correctly identifying which alerts represent real security risks. The before-and-after distributions shown in Table 3 and Figure 6 show that disagreement-aware LLM interpretation improves vulnerability prioritization by (i) resolving severity inflation blind to context in the highest-priority bucket, (ii) reducing alert fatigue through consolidation filtering, and (iii) aligning remediation order with security relevance and practical impact. These results provide direct evidence in support of RQ2: incorporating disagreement-aware LLM reasoning as an interpretation layer yields more developer-usable prioritization of security risks than raw multi-tool severity labels alone.

When it comes to handle to the RQ2, disagreement-aware LLM reasoning improves vulnerability prioritization by qualifying context-limited severity assignments and focusing remediation on security risks. In our experiment, LLM reclassified the single highest-priority baseline item from `Critical` to `High` when the available evidence indicated single-tool reporting and test-oriented context. In addition, LLM reduced `Medium` findings from **11** to **7** and `Low` findings from **42** to **28** by consolidating overlapping alerts and reclassifying a subset of pattern-based warnings as non-security signals. Importantly, raw static-analysis alerts are not deleted; instead, LLM refines the security set and generates more accurate severity ordering aligned with practical security impact. Therefore, disagreement-aware LLM interpretation yields a more reliable and developer-usable remediation prioritization than static analyzer severity alone.

7 Recommendations

The following recommendations are based on the empirical findings of this paper and the `Windows App SDK` case study. Unlike generic secure development guidelines, they explicitly rely on disagreement-aware LLM-based interpretation, consolidation of multiple static analysis tools, and refined severity assessment. Rather than focusing only on vulnerability discovery, these recommendations emphasize better prioritization, contextual understanding, and practical integration of security analysis into the software development lifecycle for large-scale `C/C++` platforms. First, Microsoft should prioritize vulnerability remediation using a severity refinement approach that goes beyond raw static analysis outputs. The results show that disagreement-aware

LLM interpretation can detect findings that are limited to test code or specific contexts and are often over-rated by individual tools, while still keeping truly high-impact vulnerabilities visible. Adding such an interpretation layer helps development teams focus their effort on security issues that realistically affect production systems, rather than reacting to inflated or duplicated severity labels. Second, security assessment workflows should actively combine multiple static analysis tools with disagreement-aware reasoning. Since different tools have different strengths and weaknesses, disagreement between tools should be treated as useful information rather than noise. An LLM-based reasoning layer can merge overlapping alerts, resolve inconsistent severity ratings, and provide clear semantic explanations. This significantly reduces alert fatigue while preserving wide vulnerability coverage, which is especially important for complex frameworks like the `Windows App SDK` with diverse code patterns and layered architectures. Third, low- and medium-severity findings should be handled using actionability-aware triage instead of mandatory remediation for all warnings. The experiments indicate that many such findings are pattern-based, policy-related, or highly context-dependent, with no clear evidence of exploitability. Reclassifying these findings as non-actionable or non-security issues while still keeping links to the original tool reports-allows security teams to clearly separate real vulnerabilities from general best-practice advice. This reduces unnecessary remediation effort and supports more balanced and informed security decisions.

Lastly, Microsoft should embed disagreement-aware interpretation into continuous security processes such as secure code reviews, CI pipelines, and regular security audits. Continuous static analysis, supported by LLM-based reasoning, allows ongoing monitoring of evolving source code and helps identify security regressions during fast development cycles. In addition to this, clear documentation, secure coding guidelines, and focused developer training based on refined security results can further lower the risk of future vulnerabilities. Together, these recommendations provide a scalable and context-aware security assessment strategy that improves the practical value of static analysis and helps to keep the `Windows App SDK` a reliable and trusted platform for developers and end users.

8 Limitations

Although this paper introduces several contributions that are highly relevant to software security researchers worldwide including disagreement-aware consolidation of multi-tool static analysis results, context-aware severity refinement, and prioritization of vulnerabilities, it also has limitations in how the paper was conducted. In particular, the proposed framework relies on the output of a limited set of five static analysis tools. Although these tools were chosen to cover various detection engines and rule-based strategies, the reported findings and severity distributions are strongly influenced by the internal logic, coverage, and precision of these specific tools. As a result, vulnerabilities that are not detected by any of the selected analyzers cannot be surfaced or further reasoned about by the LLM-based interpretation layer. Another limitation concerns the use of a specific LLM for disagreement-aware reasoning. While the GPT-5.2 Thinking model demonstrates strong reasoning and contextual interpretation capabilities, its outputs remain sensitive to prompt design and instruction framing. Variations in prompt structure, reasoning constraints, or emphasis on severity interpretation may lead to differences in consolidation and refinement outcomes. Moreover, different LLM architectures or future model versions may exhibit distinct reasoning behaviors, which limits the direct generalization of the results across models.

The experimental evaluation is further limited by its focus on a single large-scale case study, namely the `Windows App SDK`. Although this platform represents a realistic and complex software framework, the findings may not fully generalize to smaller projects, different application domains, or programming languages beyond `C/C++`. In addition, the framework is restricted to static analysis evidence and does not incorporate dynamic analysis techniques such as fuzzing or runtime validation, meaning that exploitability and execution

reachability are inferred rather than empirically confirmed. Lastly, the effectiveness of severity refinement depends on the availability and completeness of contextual information provided to the reasoning layer. Limited code context, missing deployment assumptions, or incomplete build and runtime details can affect interpretation accuracy. Although the framework aims to reduce the manual effort of the analyst, evaluating the accuracy of refined severity labels still requires expert judgement to some extent. Furthermore, integrating LLM-based reasoning into continuous security workflows may introduce scalability and cost challenges.

9 Conclusion

This paper presented a structured security assessment of the `Windows App SDK` by extending traditional static code analysis with a disagreement-aware LLM-based reasoning layer. The proposed approach consolidates and refines vulnerability findings reported by multiple static analysis tools and treats inter-tool disagreement as a first-class signal to improve severity prioritization, rather than relying on single-tool outputs or raw severity labels. Experimental results show that this consolidation reduces low- and medium-severity alert noise by approximately 30–35%, while reclassifying the single Critical baseline item to High based on disagreement-aware contextual evidence.

The results further indicate that raw static analysis outputs often suffer from alert redundancy, limited context awareness, and coarse severity assignments, which complicate vulnerability prioritization. By applying disagreement-aware reasoning, our proposed framework improves severity prioritization by approximately 25–30%, reducing unnecessary remediation effort while preserving visibility of genuinely high-impact vulnerabilities.

10 Declarations

10.1 Conflict of interest:

Not applicable.

10.2 Funding:

Not applicable.

10.3 Author's contribution:

P.P.–Conceptualization, Methodology, Investigation, Software, Formal Analysis, Writing–Original Draft. S.Q.–Data Curation, Validation, Software, Writing–Review and Editing. J.A.K.–Methodology, Consultation, Supervision, Writing–Review and Editing. M.O.–Supervision, Conceptualization, Project Administration, Writing–Review and Editing. M.E.D.–Supervision, Supervision, Project Administration, Writing–Review and Editing. All authors read and approved the final submitted version of this manuscript.

10.4 Acknowledgement:

The authors express their sincere gratitude to HCL Technologies and Fluid Attacks for providing access to their static analysis platforms, which were instrumental in supporting the vulnerability identification process for the Windows App SDK. These tools significantly contributed to the depth and reliability of the security analysis conducted in this research study.

10.5 Data availability statement:

All data that support the findings of this study are included within the article.

10.6 Usage of AI tools:

Artificial Intelligence (AI) tools were used as part of the research methodology for vulnerability interpretation and analysis. No AI tools were used in the writing, editing, or preparation of the manuscript.

References

[1] Cusumano M.A., Selby R.W., Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People, Simon and Schuster, USA, 1998.
[2] Microsoft Documentation., Visual Basic docs: Get Started, Tutorials, Reference, https://learn.microsoft.com/en-us/dotnet/visual-basic/, Accessed: January 1, 2024.
[3] MacMichael D., What's New in Windows App SDK 1.6, Windows Developer Blog, https://blogs.windows.com/windowsdeveloper/2024/09/04/whats-new-in-windows-app-sdk-1-6/, Accessed: September 4, 2024.
[4] ComponentOne., WinUI vs WPF, WinForms, UWP, and MFC, https://developer.mescius.com/blogs/winui-vs-wpf-winforms-uwp-and-mfc, Accessed: October 3, 2023.
[5] Microsoft Documentation., Windows UI library (WinUI), https://learn.microsoft.com/en-us/windows/apps/winui/, Accessed: January 1, 2024.
[6] Microsoft Documentation., Introduction to C++/WinRT, https://learn.microsoft.com/en-us/windows/uwp/cpp-and-winrt-apis/intro-to-using-cpp-with-winrt, Accessed: November 18, 2022.
[7] MITRE., Common Weakness Enumeration (CWE), https://cwe.mitre.org/, Accessed: January 1, 2024.
[8] Microsoft., Microsoft Digital Defense Report 2024, https://www.microsoft.com/en-us/security/security-insider/intelligence-reports/microsoft-digital-defense-report-2024, Accessed: January 1, 2024.
[9] CVE., .NET Framework Remote Code Execution Vulnerability (CVE-2022-41089), https://www.cvedetails.com/cve/CVE-2022-41089/, Accessed: January 1, 2022.
[10] Microsoft Documentation., Overview of .NET Framework, https://learn.microsoft.com/en-us/dotnet/framework/get-started/overview, Accessed: March 29, 2023.
[11] Di W., Lin C., Yuming Z., Baowen X., An extensive empirical study on C++ concurrency constructs, Information and Software Technology, 76, 1–18, 2016.
[12] Pereira J.D., Vieira M., On the use of open-source C/C++ static analysis tools in large projects, European Dependable Computing Conference (EDCC), 07-10 September 2020, Germany.
[13] Verdi M., Sami A., Akhondali J., Khomh F., Uddin G., Motlagh A.K., An empirical study of C/C++ vulnerabilities in crowd-sourced code examples, IEEE Transactions on Software Engineering, 48(5), 1497–1514, 2020.
[14] Microsoft Documentation., Build desktop Windows Apps With The Windows App SDK, https://learn.microsoft.com/en-us/windows/apps/windows-app-sdk/, Accessed: January 1, 2024.
[15] Chess B., Britton K., Eng C., Pugh B., Raghavan L., West J., Static analysis in motion, IEEE Security & Privacy, 10(3), 53–56, 2012.
[16] Ye T., Zhang L., Wang L., Li X., An empirical study on detecting and fixing buffer overflow bugs, IEEE International Conference on Software Testing, Verification and Validation (ICST), 11-15 April 2016, USA.
[17] Microsoft Documentation., Class Library Overview, https://learn.microsoft.com/en-us/cpp/mfc/class-library-overview?view=msvc-170, Accessed: August 3, 2021.
[18] Microsoft Documentation., What is Windows Forms – Windows Forms .NET, https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-9.0, Accessed: November 14, 2024.
[19] Microsoft Documentation., What is Windows Presentation Foundation (WPF) – WPF .NET, https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-9.0, Accessed: October 29, 2024.
[20] Crudu V., What are The Emerging Trends in XAML Development, https://moldstud.com/articles/p-what-are-the-emerging-trends-in-xaml-development, Accessed: September 2, 2024.
[21] Microsoft Documentation., What's a Universal Windows Platform (UWP) App? UWP applications https://learn.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide, Accessed: January 1, 2024.

[22] Rahman M., Beginning Microsoft Kinect for Windows SDK 2.0: Motion and Depth Sensing for Natural User Interfaces, Apress, USA, 2017.
[23] Lu B., Dong W., Yin L., Zhang L., Software Analysis, Testing, and Evolution, (Chapter: Evaluating and integrating diverse bug finders for effective program analysis), Lecture Notes in Computer Science, 8th International Conference on Software Analysis, Testing, and Evolution, 23-24 November 2018, China.
[24] Miele P., Alquwaisem M., Kim D.K., Comparative assessment of static analysis tools for software vulnerability, Journal of Computers, 13(10), 1136–1144, 2018.
[25] Malik A., Naveed M.S., Analysis of code vulnerabilities in repositories of GitHub and RosettaCode: A comparative study, International Journal of Information in Science and Technology, 4(2), 499–511, 2022.
[26] Alqaradaghi M., Kozsik T., Comprehensive evaluation of static analysis tools for their performance in finding vulnerabilities in Java code, IEEE Access, 12, 55824–55842, 2024.
[27] Zhang H., Wang S., Li H., Chen T.H., Hassan A.E., A study of C/C++ code weaknesses on stack overflow, IEEE Transactions on Software Engineering, 48(7), 2359–2375, 2021.
[28] Ning Y., Zhang Y., Ma C., Guo Z., Yu L., Empirical study of software composition analysis tools for C/C++ binary programs, IEEE Access, 12, 50418–50430, 2023.
[29] Arusoaie A., Ciobâca S., Craciun V., Gavrilut D., Lucanu D., A comparison of open-source static analysis tools for vulnerability detection in C/C++ code, 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 21-24 September 2017, Romania.
[30] Zampetti F., Scalabrino S., Oliveto R., Canfora G., Penta M.D., How open source projects use static code analysis tools in continuous integration pipelines, IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 20-21 May 2017, Argentina.
[31] Piran A., Chang C.P., Fard A.M., Vulnerability analysis of similar code, IEEE International Conference on Software Quality, Reliability and Security (QRS), 06-10 December 2021, China.
[32] Yeboah J., Popoola S., Efficacy of static analysis tools for software defect detection on open-source projects, arXiv:2405.12333, 2024.
[33] Klieber W., Flynn L., Evaluating Static Analysis Alerts with LLMs, https://www.sei.cmu.edu/blog/evaluating-static-analysis-alerts-with-llms/, Accessed: December 31, 2024.
[34] Mohajer M.M., et al., SkipAnalyzer: A tool for static code analysis with large language models, arXiv:2310.18532, 2023.
[35] Kharkar A., et al., Learning to reduce false positives in analytic bug detectors, arXiv:2203.09907, 2022.
[36] Mashhadi E., et al., An empirical study on bug severity estimation using source code metrics and static analysis, arXiv:2206.12927, 2022.
[37] Li F., Jiang J., Chen D., Xiong Y., LLM-based vulnerability detection at project scale: An empirical study, arXiv:2601.19239, 2026.
[38] Du X., et al., Reducing false positives in static bug detection with LLMs: An empirical study in industry, arXiv:2601.18844, 2026.
[39] Pushkar C., Kabra S., Kumar D., Challa J.S., Beyond single bugs: Benchmarking large language models for multi-vulnerability detection, arXiv:2512.22306, 2025.
[40] Park S., Ko G., Cho H., On the effectiveness of instruction-tuning local LLMs for identifying software vulnerabilities, arXiv:2512.20062, 2025.
[41] Saju M.H., Muhtadi M., Azim A., An empirical evaluation of LLM-based approaches for code vulnerability detection: RAG, SFT, and Dual-Agent systems, IEEE International Conference on Collaborative Advances in Software and Computing (CASCON), 10-13 November 2025, Canada.
[42] Cao D., Liao Y., Shang X., Realvul: Can we detect vulnerabilities in web applications with LLM?, arXiv:2410.07573, 2024.
[43] Sultana S., Afreen S., Eisty N.U., LLMs in code vulnerability analysis: A proof of concept, arXiv:2601.08691, 2026.
[44] Haurogné J., Basheer N., Islam S., Vulnerability detection using BERT based LLM model with transparency obligation practice towards trustworthy AI, Machine Learning with Applications, 18, 100598, 2024.
[45] Kaur A., Nayyar R., A comparative study of static code analysis tools for vulnerability detection in C/C++ and Java source code, Procedia Computer Science, 171, 2023–2029, 2020.
[46] Fatima A., Bibi S., Hanif R., Comparative study on static code analysis tools for C/C++, 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST), 09-13 January 2018, Pakistan.
[47] Wheeler D.A., Flawfinder Official Website and Documentation, https://dwheeler.com/flawfinder/, Accessed: May 20, 2019.
[48] Cppcheck., A Tool For Static C/C++ Code Analysis, https://cppcheck.sourceforge.io/, Accessed: December 31, 2024.
[49] Google Code Archive., Long-term Storage For Google Code Project Hosting, https://code.google.com/archive/p/rough-auditing-tool-for-security/, Accessed: December 31, 2024.
[50] Fluid Attacks., Application Security Testing Solutions, https://fluidattacks.com/, Accessed: December

31, 2024.
[51] HCL Technologies., AppScan on Cloud, https://cloud.appscan.com/, Accessed: December 14, 2024.
[52] Checkmarx., Application Security Platform, https://checkmarx.com/product/application-security-platform/, Accessed: December 14, 2024.
[53] Veracode., Vulnerability Scanning Tools, https://www.veracode.com/security/vulnerability-scanning-tools, Accessed: December 31, 2024.
[54] Microsoft., Release Windows App SDK 1.6.2 (1.6.241106002), https://github.com/microsoft/WindowsAppSDK/releases/tag/v1.6.2, Accessed: November 13, 2024.
[55] Pakshad P., Windows App SDK Security Analysis, https://github.com/ppakshad/WindowsAppSDK-Security-Analysis.git, Accessed: December 31, 2024.