Interpretable Vulnerability Detection Reports

***Abstract*—Software security faces a persistent gap: static analysis tools detect vulnerabilities effectively, but their technical outputs remain inaccessible to most developers. This leads to mounting security debt, as organizations must rely on security specialists for remediation, creating bottlenecks that delay fixes. This paper proposes an interpretability convention and a modular workflow that transforms raw static analyzer outputs into clear, actionable vulnerability reports for all developers, not just security experts. Our tool, `SECGen`, automates the workflow by parsing static analyzer outputs and restructuring them into clear, developer-friendly reports based on our convention, and enforcing compliance through automated validation. We validated our approach through a user study with 25 developers, comparing our interpretable reports to other state-of-the-art static analyzer outputs. The results suggest that developers using interpretable reports detect, understand and fix vulnerabilities more effectively, requiring only 67% of the time typically spent with traditional reports while writing more correct fixes. Key reasons for this include participants' preference for structured reports, with clear vulnerability descriptions and actionable fix suggestions.**

***Index Terms*—interpretable vulnerability reports, static analysis**

I. Introduction

Increasing numbers of security vulnerabilities produce mounting security debt, along with pressure on security teams [1]. To manage this growing risk, organizations often rely on static application security testing (SAST) tools to automatically identify source-level vulnerabilities before deployment [1], [2]; such tools are a key component of software security practices [3], [4].

Ideally, organizations would have enough security experts to address all static analysis warnings. In reality, general developers outnumber security specialists by approximately 100 to 1 [5]. So, organizations must either rely on a small and overburdened group of specialists or require general developers to handle security alerts. A key issue is that SAST outputs are highly technical, often complex, and lack actionable guidance [6], [7], [8]. As a result, most general developers are unable to interpret or act on these alerts, avoiding security tasks. This dynamic leaves remediation to a small group of experts and further compounds security debt [5].

Psychology research shows that comprehensible information increases understanding and engagement [9], suggesting that well-structured vulnerability reports could transform how developers respond to security findings. Applied to software security, this suggests that transforming technical vulnerability data into accessible, well-structured reports could fundamentally change how developers interpret and respond to security findings. Our key insight is that the structure and interpretability of vulnerability reports are as important as their technical content. Making security information actionable and understandable is critical to closing the gap between detection and repair, enabling general developers – not just experts – to participate effectively in remediation processes.

Building on this insight, we propose an interpretability convention for vulnerability detection reports. This reporting convention specifies not only *what* information must appear, but also *how* it should be logically organized to guarantee interpretability, actionability, and consistency for general developers. Unlike prior standards that focus on machine readability or patch tracking (e.g., OSV Schema[1]) and existing guidelines that lack operational enforcement [10], [11], our work is the first to make interpretability in security reports practical and enforceable for real-world developer workflows.

We achieve this by introducing a modular workflow – a general, tool-agnostic process for transforming raw outputs of any SAST tool into interpretable vulnerability reports that fully comply with our convention. This workflow consists of: (1) parsing and extracting key elements from static analyzer outputs, (2) restructuring these findings according to our interpretability convention, and (3) automatically checking that each report meets the convention's requirements. To enforce this, we provide explicit validation rules and a linter-based compliance checker. Finally, we propose `SECGen`, a tool that fully automates the entire workflow – from parsing and report generation to compliance checking – making our convention practical and immediately usable in real projects.[2]

We validated the convention through a comprehensive within-subjects user study involving 25 participants with diverse security expertise. Each participant completed structured surveys and hands-on programming tasks, allowing us to directly compare the efficiency of developers using our interpretable reports (generated with `SECGen`) against those using reports from other state-of-the-art tools like `CodeQL` and `AmazonQ`. Specifically, we investigated **(RQ1)** How interpretable vulnerability reports affect general developers' ability to understand and resolve vulnerabilities, compared to traditional reports, and **(RQ2)** How useful the reports are, from the developer perspective. To answer these questions, we employed a mixed-methods approach, analyzing quantitative and qualitative data gathered from the user study. Our results suggest that developers using interpretable reports repair vulnerabilities 33% faster and produce a higher proportion of correct patches compared to those using traditional reports from `CodeQL` and `AmazonQ`. Moreover, participants' qualitative feedback emphasized that the clear, modular structure of our reports – which organizes security information into sections –

[1] https://osv.dev/

[2] `SECGen` is available here [12] and will be made open-source after review.

was critical to their effectiveness, underscoring that structure and interpretability are as important as technical content.

In summary, our contributions are as follows:

1) An **interpretability convention** for vulnerability detection reports, defining a standardized, modular structure to improve interpretability, consistency, and actionable guidance for all developers – not just security experts.
2) A **practical, tool-agnostic workflow** for transforming raw static analyzer outputs into interpretable vulnerability reports. This workflow includes automated steps for parsing SAST reports, restructuring information according to our interpretability convention, and verifying compliance with the standard. We also provide `SECGen` as an implementation of this workflow, automating every stage from parsing to compliance checking and report generation.
3) We provide **empirical evidence** from a user study with 25 developers showing that `SECGen` reports help participants repair vulnerabilities in just 67% of the time required with traditional SAST reports, and write more correct patches. These improvements hold across different levels of security experience. Quantitative and qualitative results further highlight that the clear and layered structure of reports is relevant to their effectiveness.

Replication Package. All study instruments and `SECGen` are available in our repository [12].

II. Illustrative Example

Software security debt (i.e., the accumulation of unresolved security issues) has become a serious concern in modern applications. Nearly 75% of organizations carry some level of security debt, with half of them facing highly severe, long-standing flaws [1]. This persistent gap between vulnerability detection and effective repair suggests that identifying security issues is insufficient: developers must take appropriate action on these findings. Although recognized as an effective method for identifying security vulnerabilities [6], previous work identifies challenges that remain unaddressed in SAST solutions [13], [10], [14]:

C1: ensuring that warning messages are clear and informative;
C2: providing meaningful support to fix detected issues;
C3: minimizing false positives that burden developers;
C4: incorporating user feedback to refine analysis results;
C5: integrating with development workflows;
C6: designing intuitive user interfaces that facilitate interaction without overwhelming users.

Here, we explain how SASTs fall short in these areas with a practical example. In Listing 1, `generate_pwd` returns a pointer to a stack-allocated buffer `pwd` (line 7). Since this array is allocated on the stack (line 3), its memory becomes invalid once the function finishes execution. So, when `printf` dereferences the pointer, the program operates on invalid memory, resulting in undefined behavior. Such a mistake may have severe consequences, such as arbitrary code execution and memory corruption.

```
1  #define PWD_LEN 16
2  char *generate_pwd() {
3      char pwd[PWD_LEN+1];
4      for(int i = 0; i < PWD_LEN; i++)
5          pwd[i] = 'A' + rand() % 26;
6      pwd[PWD_LEN] = '\0';
7      return pwd;
8  }
9  void main(){
10     char *s = gen();
11     printf("Your password is: \%s\n", s);
12     free(s);
13 }
```

Listing 1. C code snippet showing an invalid pointer dereference vulnerability. For brevity, we present a simplified version of Codellama_13b:5773 from FormAI-v2.

`CodeQL`[3] is a widely used code analysis engine capable of detecting such issues. We ran `CodeQL` with its default set of security rules, and for Listing 1, it produced the results shown in Figure 1. Its contextual explanation is minimal (C1); it does

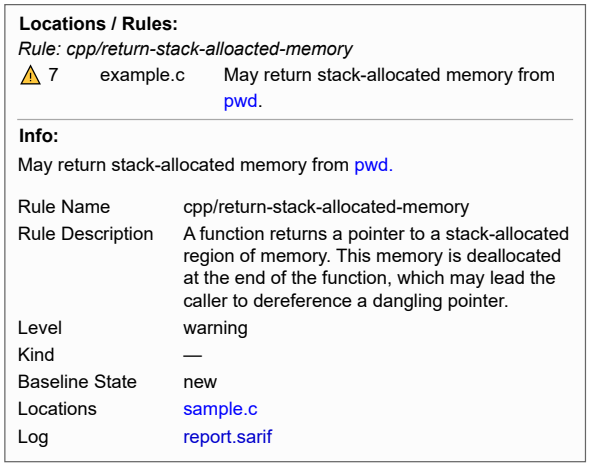

Fig. 1. `CodeQL` output in VS Code's SARIF Viewer for Listing 1.

not elaborate on the risks of returning a dangling pointer, such as memory corruption. It identifies the problematic line, but does not offer fix suggestions (C2). It uses technical language that increases cognitive load (C6). `CodeQL` itself lacks a feedback mechanism (C4); has a high barrier to entry for novice programmers (C5); and tends to a high false positive rate, without opportunity for mitigation (C3).

Section III explains how we address these challenges by focusing on the reporting stage, rather than improving the efficacy of the SAST tool.

III. Vulnerability Detection Report Convention

Based on key findings from static analysis usability studies [13], [11], [15], [16], we design our reporting structure to directly address recurring pain points such as unclear messages and the lack of actionable fixes. We also emphasize consistency and structure, as uniform formats help reduce ambiguity [17] and structured sections allow developers to

[3]https://codeql.github.com

tailor the level of detail to the context [11], [16]. Additionally, we expect that a consistent report layout may help developers locate and interpret relevant information more quickly as they become familiar with where specific details are presented.

To ensure precision and consistency, we define a set of key entities (e.g., vulnerability type, severity, code location, and other security-relevant terms) that must appear in specific sections of the report; they are summarized in Table I. For each section, we specify validation rules indicating which entities are required and how they should be presented. This approach guarantees that every part of the report is complete and standardized. Due to space constraints, we illustrate only one section's validation rule; the complete set of rules and entities is available in our *repository* [12]. Figure 2 shows an example of an interpretable vulnerability detection report.

Entity	Description and Example
CWE ID	Vulnerability type. E.g., CWE-416
SEVERITY	Vulnerability severity. E.g., High
SECWORD	Words or group of words identified as security relevant by prior work [18]. E.g., Use After Free
ACTION	Fixing a vulnerability implies an action. E.g., adding/removing some feature.

TABLE I
LIST OF ENTITIES AND DESCRIPTIONS. OTHER EXAMPLES INCLUDE TOOLING, LOCATION, CODE, IMPACT.

A. Header

Guideline Describe type of vulnerability (CWE ID and/or SECWORD), severity (SEVERITY) and location (LOCATION).
Rationale Unclear prioritization in SAST reports hinders efficient triage and repair [11], [19], [20]. Explicitly including type and severity addresses these issues by allowing developers to immediately assess the relevance and impact of a vulnerability (C1), streamlining navigation and decision-making within reports (C6).
Rule Let H be the set of words that make up the report header and x,y,z words in the header. We evaluate header compliance with the convention as follows: $(\exists\ x \in H : type(x) \in \{\texttt{CWE ID}, \texttt{SECWORD}\}) \wedge (\exists\ y \in H : type(y) = \texttt{SEVERITY}) \wedge (\exists\ z \in H : type(z) = \texttt{LOCATION})$

B. Summary

Guideline Answer four essential security questions: what is the problem (CWE ID and/or SECWORD), where it is located (LOCATION), why it is a problem and what are the consequences of leaving it unresolved (IMPACT and SECWORD), and how would an attacker exploit it (ACTION and SECWORD).
Rationale Prior studies have shown that static analysis reports often lack the context developers need to fully understand and address vulnerabilities [13], [11]. By systematically answering the "what, where, why, and how" of each vulnerability, we address the challenge of understandability (C1) through structured, developer-centered reporting [16]. Moreover, explaining the consequences and exploitation vectors directly supports deeper reasoning about code fixes (C2), as emphasized in recommendations for actionable and context-rich explanations [13], [15]. This section bridges a critical gap identified in the literature, providing developers with the information they need for informed and effective repairs.

C. Program Analysis

Guideline Document how data moves through the program. When sources and sinks are present, illustrate their connection from untrusted origins to critical operations. Otherwise, explain the control flow or state transitions that introduce the vulnerability (e.g. race conditions). In addition, include information about the environment (i.e., configurations and assumptions) if available.
Rationale Prior studies have shown that static analysis warnings are often dismissed or misunderstood by developers, especially when they require awareness of complex states or data flows not anticipated in the original program design [2]. By explicitly documenting data and control flows, our reporting structure improves understandability (C1) and potentially helps developers accurately assess whether a reported vulnerability is relevant or a false positive (C3).

D. Fix Suggestion

Guideline Suggest a patch for the vulnerability (CODE), along with a textual explanation of how the suggested modification addresses the problem (ACTION and SECWORD or CWE ID).
Rationale Developers face security challenges due to time constraints, heavy workloads, and limited security knowledge [6], [13]. While static analysis tools assist in vulnerability detection, they rarely provide actionable repair suggestions; this lack of concrete guidance impedes effective remediation and raises the likelihood of recurring vulnerabilities [13]. At the same time, fully automated patch integration remains a significant challenge, as code fixes must be both correct and compatible with project requirements [21]. By pairing patch suggestions with clear explanations, this section helps developers adapt patches to their codebase, supporting more effective vulnerability repair (C2).

E. Methodology

Guideline Enumerate the tools used to generate the report, including static analyzers and AI mechanisms (TOOLING).
Rationale Security experts are often cautious about integrating tools they do not fully understand – such as large language models – into the security loop [14]. So, full transparency is crucial. By enumerating every tool used to generate our reports, we ensure transparency, reproducibility, and a better understanding of our approach.

IV. GENERATING INTERPRETABLE REPORTS

To contextualize and validate our interpretability standard, we introduce a workflow for automatically generating reports from static analysis tool outputs, as depicted in Figure 3.

Initially, a given code sample is analyzed using a chosen static analyzer, generating a traditional vulnerability report.

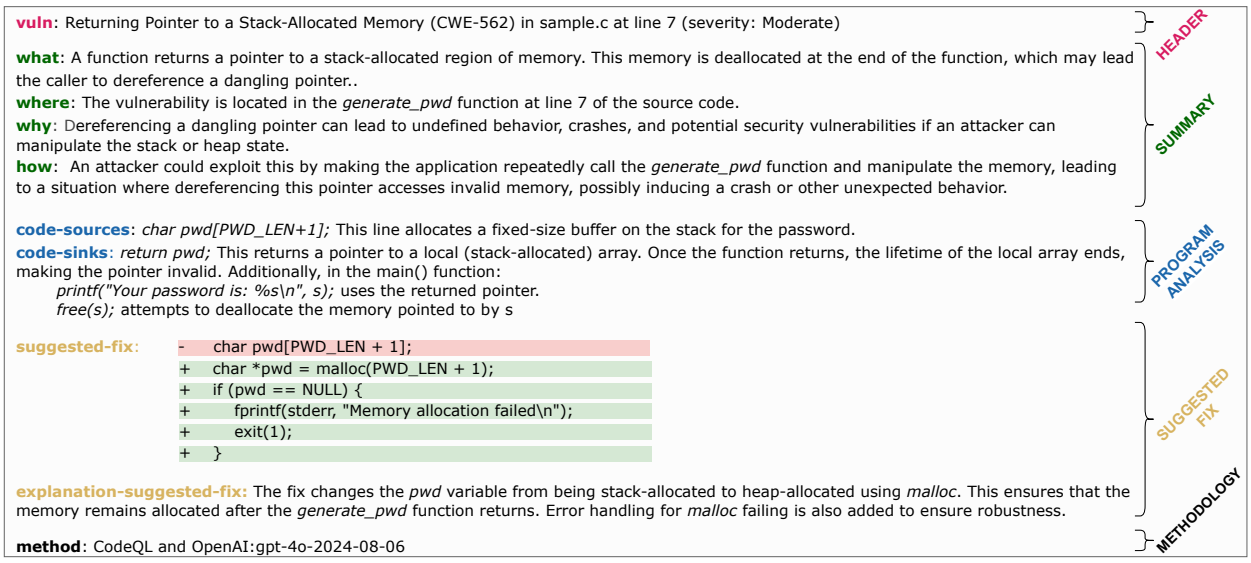

Fig. 2. Interpretable vulnerability detection report for example in Listing 1. Section III details each report section.

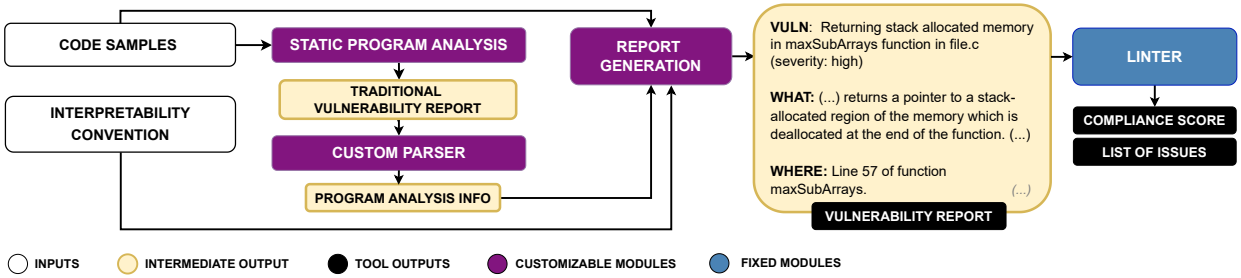

Fig. 3. The SECGen workflow follows a modular design: a static analyzer detects vulnerabilities, a parser extracts key details, a text generator produces an interpretable report, and a linter ensures compliance against our convention.

The effectiveness of this analysis strongly depends on the quality of the underlying security rules of the analyzer. To meet our standard, the static analyzer's rule set ought to be able to associate identified code patterns with standardized security labels (e.g., CWE identifiers) and pinpoint the code location precisely (e.g., start and end line numbers). This mandatory information enables direct mapping of findings to known vulnerabilities. Additionally, while information about vulnerability severity is valuable, its automatic assessment is inherently challenging due to the dependency on the specific code context, program logic, and execution environment. Thus, although severity estimation is not strictly required by our interpretability standard, static analyzers can enhance their reports by providing complementary context-independent metrics, such as the likelihood or ease of exploiting identified vulnerability patterns. This also helps prioritize repair efforts. Finally, note that static analyzers commonly face difficulties when analyzing nonstandard code constructs, such as macros or obfuscated code [22]. Although program transformations can partially alleviate these issues [23], addressing this limitation is outside the scope of the proposed workflow. Once the static analyzer generates a report containing the mandatory vulnerability information described above, a custom parser processes this report, extracting critical vulnerability details and restructuring them into a standardized intermediate format.

The standardized data and the original code sample are then fed into a Report Generation module that builds the report according to our guidelines. LLMs represent a compelling implementation option for this module due to their dual strengths in writing reasonable code fixes [24] and producing clear natural language explanations [25]; this combination directly addresses the convention requirements for both technical patches and contextual descriptions. Moreover, LLMs allow developers to tailor solutions through model selection and parameter optimization, accommodating varying performance requirements, resource limitations, or domain-specific constraints. However, alternative approaches remain viable, such as combining human-written descriptions with traditional program repair techniques [26]. Regardless of the chosen implementation strategy, the only non-negotiable requirement is strict adherence to the established reporting convention.

Finally, a linter module evaluates the quality of the generated report. Using Named Entity Recognition (NER), the linter ensures that all entities are correctly positioned according to

our guidelines, enforcing the convention. The overall compliance score is calculated as the percentage of rules satisfied out of the total number of rules, with scores near 100% indicating high adherence to the guidelines.

`SECGen` CLI Prototype We implemented this workflow in a prototype command-line interface `SECGen`. We used `CodeQL` as the static analyzer due to its robust set of security rules and strong community support; `SECGen` operates specifically on `CodeQL`'s SARIF reports.[4] To generate natural language explanations and patch suggestions, we used OpenAI's `gpt-4o-2024-08-06` model. We also include support for local Ollama[5] models as a cost-free alternative, enabling users to try `SECGen` and generate interpretable reports without relying on commercial APIs. `SECGen` also evaluates report compliance using a linter, adapted from `SecomLint` [18], that automatically detects deviations from the standard. This validation tool provides specific actionable feedback, such as *"Header is missing a severity level.* (`SEVERITY`)".

V. Human Study Design

We evaluated our reporting standard by measuring how it improves developers' ability to understand, detect, and repair security vulnerabilities, using these practical outcomes as proxies for interpretability, which is inherently subjective [25]. We recruited 25 participants (Section V-A) and designed a study that included a pre-study survey (Section V-B) to assess eligibility, followed by a task-based survey that guided participants through hands-on programming tasks – each accompanied by its own set of questions – and a set of final questions (Section V-C). The programming challenges required participants to fix the detected vulnerabilities, providing a practical measure of their repair skills. Each participant interacted with all types of reports, allowing comparisons within the subject and controlling for individual differences. We then analyzed responses using a mixed-methods approach (Section V-D).

A. Participants

We recruited participants with varying technical experience, while ensuring that they had programming experience and sufficient familiarity with the C language.

1) Sampling strategy: We advertised our study in three ways: (1) within the university community through personal contacts, (2) we sent emails to researchers and software developers working in the field of software security, and (3) posted on social media hoping to attract participants with industry experience. To select eligible applicants, we administered an asynchronous pre-test survey to verify the levels of experience with C. We considered eligible those applicants who correctly answered 3 of 5 multiple choice questions; 28 applicants satisfied these criteria. We sent eligible participants an email to schedule their study sessions; 3 participants did not respond. In total, 25 participants took part in the study.

[4]https://docs.github.com/en/code-security/codeql-cli/using-the-advanced-functionality-of-the-codeql-cli/sarif-output\#about-sarif-output

[5]https://ollama.com/

2) Demographics: 14 participants identified as graduate students, 2 as post-doc researchers, 2 as industry developers working in security, 4 as industry developers not working in security, 1 as undergraduate student, and 2 preferred not to answer the question. Regarding programming experience, 5 participants reported having between 1 and 5 years of experience, 17 participants had 6 to 10 years of experience, and 3 participants had more than 10 years of experience. When it comes to vulnerability analysis experience, 12 participants reported no prior experience running or analyzing vulnerability scans, 6 had less than 1 year of experience, 4 had between 1 and 5 years of experience, and 3 had between 5 and 10 years of experience. Regarding tool usage, 4 participants reported familiarity with `CodeQL`; users were not familiar with `AmazonQ`. So, based on self-reported experience with vulnerability analysis processes and tools, we classified participants regarding their vulnerability knowledge into three levels:

Novices ($n = 14$) reported zero experience with security tasks or vulnerability analysis.

Intermediates ($n = 7$) reported prior exposure to security or program analysis tools, typically acquired in school settings (e.g., courses) or personal code experiences.

Experts ($n = 4$) reported being actively involved in security domains, discovering and analyzing vulnerabilities in industry or research settings. They reported a deep understanding of security practices.

B. Pre-study survey

We built a 15-minute pre-study survey to assess security experience and determine eligibility.

1) Design: The survey has an introductory section, followed by a demographics block where participants answer multiple choice questions about their current occupation, years of programming experience, and familiarity with C (including self-assessments of their ability to understand and write C code). Next, the survey features two task-based sections. In Task 1, participants answer two multiple choice questions based on C code snippets that assess their code comprehension, while Task 2 includes three multiple choice items that test understanding of memory management practices and pointer usage in C. Finally, we ask participants for follow-up session date/time options and their email address for contact.

We piloted the pre-study survey with four developers (two security experts) to refine wording and validate code snippets.

2) Protocol: We implemented an asynchronous protocol where participants completed the pre-study survey at their convenience via Qualtrics. All responses were reviewed; eligible participants were emailed to schedule the next phase, while others were notified of exclusion.

C. Study survey and tasks

The study tasks were designed to assess the ability of participants to identify, understand and address security vulnerabilities using different types of vulnerability reports. Each task presented participants with a code snippet and a report which required them to analyze the reported issue, assess the

	SECGen		CodeQL		AmazonQ	
	Detect	Fix	Detect	Fix	Detect	Fix
Sample 1	1/2	1	1	-	0	0
Sample 2	1	1	1	-	1	0
Sample 3	1	1/2	1	-	0	0

Detect column: 1 = correct, 1/2 = partial, 0 incorrect

Patch column : 1 = correct, 1/2 = plausible, 0 = incorrect, - = unprovided

TABLE II
COMPARISON OF THE STUDY TOOLS IN THEIR ABILITY TO DETECT AND FIX VULNERABILITIES IN THE STUDY SAMPLES. DEFINITION OF CORRECT, PLAUSIBLE AND INCORRECT PATCHES ARE IN SECTION V-D.

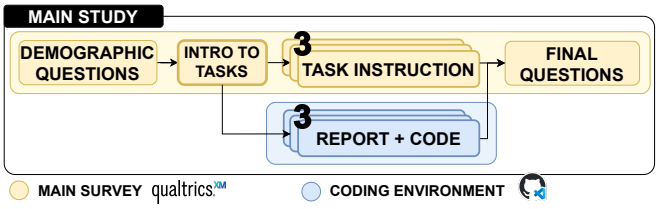

Fig. 4. Participants joined a live Zoom session, completed the survey, and, during coding tasks, used a pre-set coding environment to consult code samples and reports.

interpretability of the report, and attempt a fix. We estimated around 20-30 minutes per task and gave 3 tasks per participant.

1) Code samples: We selected samples based on static analysis and usability requirements. Since SAST tools often require code to be syntactically correct and complete, we selected samples from the FormAI-v2 dataset [27], which has around 331,000 compilable C programs with vulnerability labels. We selected three vulnerabilities to present to the participants, to mitigate fatigue while still collecting sufficient data. Our selection criteria were:

1) **Size.** Code samples have approximately 100 lines, ensuring that the task is doable within 20-30 minutes while still providing meaningful code review challenges [28].
2) **Complexity.** Samples are of moderate complexity, maintaining engagement without causing discouragement.
3) **Relevancy.** All samples feature a subtle yet critical vulnerability: dereferencing pointers to deallocated memory. Unlike more obvious issues like double frees, this vulnerability requires careful analysis to detect, and often precedes exploitable conditions (see Listing 1) which appear in the 2025 Top 25 Most Dangerous Software Weaknesses.[6]

The three samples cover diverse programming scenarios, implementing: string translation (Sample 1), array extraction (Sample 2), and URL sanitization (Sample 3) - each with pointer dereference issues. Sample 1 contains two instances of the same vulnerability; Samples 2 and 3 contain only one.

2) Reports: We based our study on reports from three tools: `SECGen`; `CodeQL`, a widely adopted tool for identifying security vulnerabilities [29], [30]; and `AmazonQ`, a promising AI-assisted code analysis tool.[7] `CodeQL` detects vulnerabilities without recommending fixes, `AmazonQ` goes a step further by suggesting generic non-code-specific patches. We standardized all tool outputs by converting them to Markdown, preserving readability and usability. However, the content and quality of the report varied, as depicted in Table II.

3) Design: The survey guides participants through a series of tasks to evaluate the interpretability of three vulnerability reports; Figure 4 shows the study workflow. First, it asks about the security tools they have used, the contexts in which they performed vulnerability analysis (e.g., bug bounty programs, school courses, testing) and their overall experience with vulnerability scanning and reporting.

The survey then presents the study tasks. Participants answer multiple-choice questions about the reported vulnerability. Next, they evaluate report components using Likert scales (ranging from *Strongly Disagree* to *Strongly Agree*). Finally, participants are asked to create a fix for the vulnerability. All participants work with the same code samples and interact with all three types of report. Although the order of the code samples is fixed, the presentation order of the reports varies, following a counterbalancing scheme [31] to systematically vary report order. This evenly distributes, and thus controls for, order effects. Finally, participants answered questions designed to uncover what makes a report more interpretable, including open responses, and an evaluation of specific properties.

We piloted the main study with six additional developers (two security experts). One identified an unlabeled vulnerability in one code snippet, which we corrected post-pilot.

4) Protocol: The study was conducted in individual 90 minute sessions via Zoom. Each session began with participants consenting, and accessing the Qualtrics survey. We instructed participants to complete all tasks independently without consulting external resources. Participants were given access to a controlled web-based development environment through GitHub Codespaces to ensure a consistent and realistic setup without the need for local installations [32]. We asked participants to share their screens to maintain uniformity. To respect privacy, no video, audio, or screen activity was recorded; instead, we captured insights through open-ended survey responses and interviewer notes. All participants worked sequentially through the three tasks. Finally, we provided a $20 USD Amazon gift card as compensation. The study was approved by our institutions' relevant review boards.

D. Data Collection and Analysis

For each task, we measured:

- **Vulnerability understanding:** (discrete non-negative variable) The number of survey questions about a vulnerability that participants correctly answered.
- **Report utility** (discrete non-negative variable) The number of survey questions about a vulnerability that participants correctly answered based on the information from the report.
- **Patch correctness**: (categorical variable) We categorized user-generated patches as `correct`, `incorrect`, or

[6]https://cwe.mitre.org/top25/

[7]https://docs.aws.amazon.com/codeguru/detector-library/

plausible. We wrote unit tests (with 95% branch coverage) to evaluate functional correctness and method signatures. We manually reviewed patches and used the ESBMC formal verification module[8] to check for flaws. A patch is:
  - Correct if it preserves the program's behavior (i.e., it passes all tests), does not change method signatures, and resolves the vulnerability (without introducing new ones).
  - Plausible if it resolves the vulnerability (without introducing new ones) but changes behavior or signatures
  - Incorrect if it does not resolve the vulnerability and causes the program to fail tests, or changes signatures.
- **Patching time:** (continuous variable) The amount of time elapsed from the participant opening the page containing the vulnerability repair question to advancing to the next page.

At the end of the tasks, we also collected written feedback through closed- and open-ended responses to further assess the perceived utility of the report. To analyze the collected data, we used regression models to answer **RQ1** and **RQ2**, ensuring control for participant differences and task variation. We compared the impact of using SECGen reports versus others by estimating regression models for the outcome variables: *vulnerability detection and understanding*, *patching time*, *patch correctness* and *report utility*. Specifically:

- For *vulnerability understanding* and *report utility* (count variables), we used a Poisson mixed-effects model since there are no signs of overdispersion in the data.
- For *patching time* (continuous variable), we used a linear mixed-effects model.
- For *patch correctness* (categorical variable), we used multinomial logistic regression.

In these models, the main predictors are the type of report and the differences between code samples. To control for individual differences in security experience, we incorporated participants' self-reported data. Given the within-subjects design, where each participant evaluated all conditions, we also included random effects to account for individual variability.

We also conducted a qualitative analysis for **RQ2** to capture richer insights into participant perceptions and the nuanced factors influencing *report utility*. Concretely, we (1) performed a thematic content analysis of the open-ended responses of the participants, using an inductive coding approach [33] to identify emerging themes across the levels of expertise. We compared them to the structured responses., and (2) built a heat map to visualize participants' property relevance preferences, stratified by security expertise levels.

VI. Results

This section summarizes the results of our study.

A. RQ1. Effects of interpretable vulnerability reports on vulnerability detection and repair

1) Effects on vulnerability understanding: To assess whether SECGen reports improve vulnerability understanding by developers, we use a regression model. Consider Table III, column "Vulnerability understanding" shows the results. Developers using SECGen reports correctly identify and locate statistically significantly more vulnerabilities (0.6554 more, $p < 0.01$) than those using AmazonQ reports. There are no statistically significant differences from the CodeQL reports, as expected, since SECGen uses CodeQL as the underlying static analyzer. Similarly, vulnerability analysis experience and sample type do not show statistically significant effects on vulnerability understanding. Interestingly, developers who reviewed both SECGen and CodeQL reports more effectively identified the false positives in AmazonQ's output for Samples 1 and 3 (Table II). Four novice developers identified these issues after examining the SECGen and CodeQL reports, whereas only intermediate or expert participants detected the problem in Sample 1's AmazonQ report. These findings may reflect the benefits of our report structure; more data is needed to confirm causality.

Finding. SECGen reports significantly improve developers' ability to detect and understand vulnerabilities compared to AmazonQ, with CodeQL showing similar performance to SECGen. Moreover, developers who reviewed SECGen and CodeQL reports were better at detecting false positives in AmazonQ's output, with even novice participants successfully identifying errors. Although the improvement seems related to the convention, we need more data to confirm causality.

2) Effects on patching time: Next, we evaluate whether the use of SECGen reports affects the time it takes developers to patch vulnerabilities. Table III (column "Time") presents the coefficients, standard errors (in parentheses), and p-values. We focused only on correct patches, since some participants skipped the task after failing to resolve the vulnerability, and this distorts timing results. For the reference group ("Using SECGen + Sample 1"), average patching time is 211.36 seconds, but this estimate is not statistically significant. In contrast, developers who use CodeQL reports take significantly longer (an additional 104 sec, $p < 0.001$) than those using SECGen reports. This increase is likely due to the lack of patch suggestions in CodeQL reports, which forces developers to spend additional time determining an appropriate fix (as supported by participant feedback discussed in Section VI-B). No user was able to correctly patch vulnerabilities with AmazonQ. Sample 2 is related to a statistically significant reduction in patching time (reduction of 316 sec, $p < 0.001$). Samples 1 and 2 require similar fixes, which suggests that learning effects may influence Sample 2 fix time. On the other hand, Sample 3 significantly increases the patching time (additional 59 sec, $p < 0.001$). Our analysis shows that neither vulnerability experience nor programming experience significantly influences patching time. This suggests that within this group of developers, experience levels do not lead to notable differences in how quickly they patch vulnerabilities.

Finding. Developers fix vulnerabilities faster using SECGen reports rather than CodeQL, requiring only 67% of the time.

3) Effects on patch correctness: To investigate whether SECGen reports contribute to better repair skills, we built

[8]https://github.com/esbmc/esbmc

	Vulnerability Understanding	Patching Time	Patch Correctness		Report Utility
			Incorrect vs. Correct	*Plausible vs. Correct*	
SECGen + Sample 1	0.835 (0.328) *	211.357 (275.38)	-0.90 (1.47)	1.36 (1.34)	1.460 (0.315) ***
AmazonQ	-0.655 (0.245) **	-	17.80 (0.56) ***	15.11 (0.56) ***	-0.322 (0.193) .
CodeQL	0.142 (0.198)	104.991 (0.401) ***	1.64 (0.84) **	-1.44 (0.98)	0.054 (0.171)
Experience	-0.137 (0.092)	-111.406 (68.335)	-0.29 (0.35)	-0.89 (0.43) **	-0.095 (0.065)
Sample 2	-0.091 (0.219)	-316.250 (0.696) ***	0.73 (0.96)	-13.77 (188.37)	-0.080 (0.191)
Sample 3	0.164 (0.214)	59.523 (0.806) ***	-1.05 (1.10)	1.22 (1.02)	0.054 (0.175)

*Shaded row is baseline. Note: . p <0.1; * p <0.05; ** p <0.01; *** p <0.001*

TABLE III

SUMMARY OF REGRESSION RESULTS FOR FOUR OUTCOMES: "VULNERABILITY UNDERSTANDING," "PATCHING TIME", "PATCH CORRECTNESS", AND "REPORT UTILITY". EACH ROW SHOWS A PREDICTOR (E.G., REPORT TYPE, VULNERABILITY ANALYSIS EXPERIENCE), ITS REGRESSION COEFFICIENT, AND STANDARD ERROR IN PARENTHESES. THE BASELINE ROW REPRESENTS THE REFERENCE VALUES WHEN OTHER PREDICTORS ARE AT REFERENCE VALUES (I.E., 0/FALSE). ASTERISKS MARK STATISTICALLY SIGNIFICANT EFFECTS (P-VALUES), SHOWING RELEVANT PREDICTORS.

a multinomial logistic regression model, shown in Table III (column "Patching correctness"). `Correct` patches serves as the reference category, so all coefficients reported below indicate their effect on the log-odds of producing `incorrect` or `plausible` patches relative to `correct` (i.e. columns *Incorrect vs. Correct* and *Plausible vs Correct*, respectively). Once again, Table III shows coefficients, standard errors (in parentheses), and p-values (obtained with Wald tests, as is common practice for multinomial logistic regressions [34]).

Results indicate that `AmazonQ` reports significantly increase the likelihood of producing `incorrect` patches (more 17.80, $p < 0.001$) and `plausible` patches (more 15.11, $p < 0.001$) compared to `correct` patches. `AmazonQ`'s responses lack context-specific guidance; this may be leading the developers to introduce patches that appear reasonable but fail to fully resolve the vulnerability. Furthermore, the high rate of `incorrect` and plausible patches associated with `AmazonQ` suggests that developers may be engaging in trial-and-error, applying fixes that appear plausible without a deep understanding. In contrast, `SECGen` reports offer more structured and context-aware guidance, which may be supportive of the more accurate solutions.

`CodeQL` reports show a weakly significant effect on increasing the likelihood of `plausible` patches (more 1.64, $p < 0.05$) but do not significantly impact the likelihood of `incorrect` patches. This suggests that `CodeQL` reports help developers get closer to the correct solution but do not provide enough actionable information to help programmers reach successful repairs. One reason for this is that `CodeQL` do not explicitly suggest how to fix vulnerabilities, causing developers to implement partial or suboptimal solutions with the knowledge they have. We also find that the vulnerability analysis experience negatively impacts the likelihood of producing `plausible` patches (less 0.89, $p < 0.05$), suggesting that more experienced participants are less likely to submit patches classified as `plausible` instead of `correct`. Programming experience does not have a significant effect.

Finding. `AmazonQ` significantly increases the likelihood of developers writing `incorrect` and `plausible` patches, indicating that its automated guidance may be misleading or incomplete. `CodeQL` reports increase `plausible` patches but do not significantly impact `incorrect` patches, suggesting that they help developers get closer to the right fix, but often leave gaps. In contrast, `SECGen` reports appear to be more effective in guiding developers toward fully `correct` patches, likely due to their structured and context-aware recommendations.

RQ1. Interpretable vulnerability reports generated with `SECGen` significantly enhance developers' ability to detect and understand vulnerabilities. Their structured, context-aware recommendations not only enable faster and more accurate patching but also help developers identify false positives in subsequent reviews.

B. RQ2. Perceived utility of interpretable vulnerability reports

We estimated *report utility* using a Poisson mixed-effects model. To better understand the results, we analyzed open-ended and dichotomous responses (i.e., "improves comprehension"/"does not improve comprehension"). Open-ended responses captured spontaneous opinions, while structured formats encouraged consideration of specific benefits that participants may have not otherwise considered.

Table III (column "Report utility") shows the coefficients, standard errors (in parentheses), and relevant p-values. The baseline – `SECGen` reports and Sample 1 – has an estimated utility score of 1.460 ($p < 0.001$). Compared to this baseline, `AmazonQ` reports are perceived as less useful (less 0.322, $p < 0.1$). `CodeQL` reports do not differ significantly from those of `SECGen` in terms of utility.

Structured responses. To better understand these differences, we presented participants with "it helps"/"it doesn't help" (dichotomous) questions about which report properties helped them understand the vulnerabilities. Figure 5 summarizes the responses; "NL Explanations" refers to vulnerability and patch explanations using natural language. Novices consistently placed the highest value on elements that offered straightforward guidance and actionability – vulnerability location, patch suggestions, and natural language explanations. Their

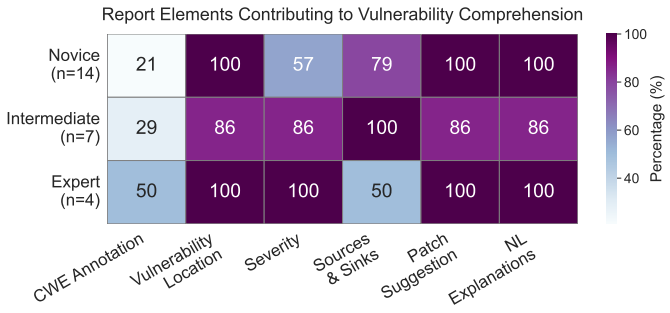

Fig. 5. Participants' preferences for report properties, stratified by security expertise levels.

comparatively low rating for severity and CWE indicates a preference for direct, immediate instructions over risk metrics or abstract classifications. Intermediate developers showed a more balanced pattern, generally rating location, patch suggestions, severity, and explanations highly (all above 80%), but still giving CWEs relatively low importance (29%). Finally, Experts prioritized vulnerability location, patch suggestions, explanations, and severity (all at 100%). Interestingly, they were also more receptive to CWE descriptions (50%) than other participants, possibly because they can integrate this information into their existing knowledge or workflow.

Open-ended responses. By far, the three themes appearing most often are the *explicit location of the vulnerability* in the code, a *clear explanation of why/how it is a problem*, and a *suggested code fix*. Concretely, one participant noted that a good report should "*(...) clearly identify the offending lines of code, why they are vulnerable, and possible fixes.*" In addition, while including patches was generally appreciated, respondents cautioned against blindly applying them. As one participant stated, "Patches are often helpful, but I would not blindly apply them myself." This indicates that developers value understanding the rationale behind a suggested fix, rather than relying solely on automated recommendations.

Several participants also highlighted the importance of report structure. . For instance, P5 wrote reports should "*be easy to navigate,*" P23 appreciated "*the context [being] broken down into bullet points... so that I do not have to dig through information,*" and P24 emphasized the need for reports to be "*divided in clear sections to make it easy to skim (for busy programmers).*" This feedback shows that added information – like explanations or fix suggestions – only benefits developers when it is clearly organized and easy to find. Consistent with previous work [16], [11], [13], [15], our findings suggest that richer content is only useful when paired with clear structure; simply adding information is not enough. This is why our interpretability convention is relevant – it ensures all key properties are not just present but also logically organized.

Regarding technical details, such as severity ratings and CWE indications, they were not universally valued. One comment remarked, "*(...) this type of data is not so important for development phases*" while another observed, "*(...) using technical nomenclature by itself is not transparent at all and puts the onus of searching for the actual explanation on the programmer.*" These responses suggest that developers may not find technical jargon and abstract metrics as useful as immediately actionable insights.

RQ2. Our interpretable reports are perceived as more useful than others because they provide clear vulnerability locations, concrete explanations, and actionable patch guidance – all presented in an organized structure. In contrast, technical jargon, severity metrics, and CWE classifications are valued less, especially by nonexperts.

C. Threats to Validity

With respect to construct validity, first, we measured security experience using self-reported data, which may be prone to over- or under-estimation biases. We mitigated this risk with a pre-study survey validating participants' baseline knowledge of C, including multiple-choice security-related questions. Second, acquiescence bias is the tendency for participants to agree with statements regardless of their true beliefs, which can distort results. To mitigate this, we followed survey best practices: using neutral wording, avoiding check-all-that-apply formats, and including both positive and negative options [35]. All vulnerability reports were also pre-generated and presented in a randomized, anonymized order so users did not know which tool produced each report.

Learning effects represent a threat to internal validity; we mitigated this by using a counterbalancing schema [31] that randomized the order of the conditions.

Regarding external validity, we selected code samples of tractable complexity given our time constraints. Despite this, these samples still reflect real-world vulnerabilities and mirror normal code distribution in real projects [36]. Additionally, we standardized all vulnerability reports in markdown format. While this preserved structure and key information, the reduced interactivity and uniform presentation may have influenced participants' responses. To mitigate these issues, we piloted the markdown format to ensure clarity and randomized and anonymized the report presentation during user studies.

The same considerations apply to the tools used to generate the reports: for our study, `SECGen` reports were generated using `gpt-4o-2024-08` and `CodeQL`, while comparison reports were produced using `CodeQL` and `AmazonQ`. Alternative tools, models, or configurations (such as manual report writing or traditional repair techniques) might yield different results. We partially mitigate this by including a linter to enforce consistency and ensure a standard quality level.

Finally, our study reflects typical development teams, where general developers far outnumber security specialists [5], but included only four experts. Future work should involve more security specialists to identify expert-specific challenges.

VII. Discussion & Opportunities

Integrating Interpretable Vulnerability Reporting into Development Workflows. Developers need varying levels of detail depending on context, preferring concise warnings in some settings and more detailed explanations elsewhere [16]. Our

convention embodies this principle by structuring vulnerability reports into modular, layered sections, allowing developers to access the right information when needed. For example, concise summaries can be shown during code review, with full explanations and fix guidance available as needed.

Our workflow vs intelligent code assistants. Intelligent code assistants, such as AmazonQ and Copilot[37], allow developers to ask for vulnerability explanations or fixes, but their usefulness is often limited by the need for effective prompting – a process that can be time-consuming and particularly challenging for non-experts who may not know which questions to ask. In contrast, our workflow automatically produces comprehensive, structured reports that present all relevant details by default, eliminating the need for manual prompting.

Some vulnerabilities may be harder to explain than others. Some vulnerabilities are inherently more complex than others. For instance, vulnerabilities that span multiple files or involve complex data dependencies may require additional context or specialized expertise to fully understand their impact. Our study did not systematically assess how report effectiveness varies with vulnerability complexity, nor did we analyze which types of vulnerabilities are most challenging for developers to comprehend. Future work should investigate these dimensions to identify where structured reporting conventions are most and least effective, and how to further support developers facing particularly complex cases.

VIII. Related work

Vulnerability Reporting Standards Prioritize Automation Over Developer Guidance. Industry initiatives like Google's "Know, Prevent, and Fix" framework [38] and the OSV Schema focus on structured metadata, automation and risk management for supply chain security. These standards excel at enabling machine readability and future-oriented tracking, but do not directly support developers in understanding and fixing vulnerabilities as they encounter them. In contrast, we transform technical analysis outputs into clear, actionable reports to help developers comprehend and remediate issues right away. SECOM [39] advances vulnerability communication through a structured convention for security commit messages, primarily aimed at improving vulnerability tracking via patch documentation. Our work adapts and extends the principles SECOM to the context of vulnerability detection reports. We also added a compliance layer – inspired by Reis et al. [18] – to ensure reports are consistenta and complete.

Usability Barriers in Static Analysis Limit Developer Adoption. Prior research has widely established that the practical adoption of static analysis tools is hindered by poor usability and lack of actionable communication. Multiple studies [13], [11], [15], [16] consistently highlight recurring barriers (e.g., unclear warning messages, insufficient fix support, workflow disruption, and generic or overwhelming reporting) and recommend guidelines for addressing them (e.g., developers must benefit from context-sensitive information, avoid overwhelming developers with too many warnings at once). However, none of these works proposes solutions directly applicable in development workflows. Other approaches [40], [41], [42] have improved the usability of static analyzers by reducing false positives or refining bug localization. But none of these efforts address how the outputs of static analysis tools are presented. In contrast, instead of improving the tools themselves, we focus on their outputs and introduce a structured reporting convention and offer a practical, ready-to-use solution for report generation that is agnostic to the underlying static analyzer.

LLMs Enhance Detection, Not Actionable Reporting. Recent work explores distinct ways to use LLMs for security. Concretely, some use LLMs to augment static analyzers by improving contextual reasoning, reducing false positives, or helping to understand and repair code [43], [44], [45], [46], [47]. Others focus in making vulnerability detection reports more actionable. Flynn and Klieber [19] used LLMs to automate alert triage by generating structured explanations for security experts, while Mao et al. [20] leveraged LLMs to improve vulnerability detection and generate detailed explanations for developers. Both differ from our approach: the first targets expert triage in large-scale settings, and the latter is tightly coupled to LLM-based detection and prompting. In contrast, our method is tool- and model-agnostic, automatically producing standardized actionable reports from any detection system without human-in-the-loop reviews or custom model integration. Our explicit compliance layer ensures automated validation of completeness and consistency. Most importantly, our reports are designed for immediate use by developers, enabling them to promptly understand and address vulnerabilities as they arise.

IX. Conclusion

Despite significant advances in static analysis for vulnerability detection, persistent usability challenges continue to limit the practical impact of these tools, resulting in heavy reliance on security specialists and growing security debt. Existing solutions focus on automation and machine readability, but lack actionable reporting for general developers. To address these limitations, we proposed: (1) a structured interpretability convention for clear and actionable vulnerability reports; (2) a modular workflow, implemented in `SECGen`, that standardizes the generation of interpretable reports; and (3) an empirical evaluation with 25 participants, assessing the effectiveness and utility of our approach. We found that our interpretable vulnerability reports improved the speed and accuracy of the repair: code fixes were completed in 67% of the time required when using traditional reports, and participants produced more correct patches than incorrect or plausible ones. While further validation is needed, our results provide initial evidence that structured, human-centered reporting can help address persistent usability challenges in vulnerability management.

References

[1] Veracode, "2025 state of software security: A new view of maturity," https://www.veracode.com/resources/analyst-reports/state-of-software-security-2025, February 2025.

[2] D. Baca, B. Carlsson, K. Petersen, and L. Lundberg, "Improving software security with static automated code analysis in an industry setting," *Software: Practice and Experience*, vol. 43, no. 3, pp. 259–279, 2013.

[3] Z. Shen and S. Chen, "A survey of automatic software vulnerability detection, program repair, and defect prediction techniques," *Security and Communication Networks*, vol. 2020, no. 1, p. 8858010, 2020.

[4] G. Dolcetti, V. Arceri, E. Iotti, S. Maffeis, A. Cortesi, and E. Zaffanella, "Helping llms improve code generation using feedback from testing and static analysis," *arXiv preprint arXiv:2412.14841*, 2024.

[5] D. Berman, "Complete guide to application security: Tools & best practice," https://snyk.io/articles/application-security, November 2020.

[6] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[7] M. Alqaradaghi and T. Kozsik, "Comprehensive evaluation of static analysis tools for their performance in finding vulnerabilities in java code," *IEEE Access*, 2024.

[8] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 544–555.

[9] P. J. Silvia and T. B. Kashdan, "Interesting things and curious people: Exploration and engagement as transient states and enduring strengths," *Social and personality psychology compass*, vol. 3, no. 5, pp. 785–797, 2009.

[10] M. Nachtigall, M. Schlichtig, and E. Bodden, "A large-scale study of usability criteria addressed by static analysis tools," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 532–543.

[11] J. Smith, L. N. Q. Do, and E. Murphy-Hill, "Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security," in *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, 2020, pp. 221–238.

[12] Anonymous, "Interpretable vulnerability detection reports – replication package," https://anonymous.4open.science/r/interpretable-reports/README.md, May 2025, retrieved May 30, 2025, from https://anonymous.4open.science/r/interpretable-reports/README.md.

[13] M. Nachtigall, L. N. Q. Do, and E. Bodden, "Explaining static analysis-a perspective," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 2019, pp. 29–32.

[14] B. A. Alahmadi, L. Axon, and I. Martinovic, "99% false positives: A qualitative study of {SOC} analysts' perspectives on security alarms," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2783–2800.

[15] M. Schlichtig, "Building a framework to improve the user experience of static analysis tools," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 165–169.

[16] P. L. Gorski, Y. Acar, L. Lo Iacono, and S. Fahl, "Listen to developers! a participatory design study on security warnings for cryptographic apis," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–13.

[17] D. Whittaker and B. Shelby, "Developing good standards: Criteria, characteristics, and sources," in *A workshop proceedings, defining wilderness quality: The role of standards in wilderness management. Gen. tech. rep. PNW-GTR-305. USDA Forest Service, Pacific Northwest Research Station, Fort Collins, Colorado*, 1992, pp. 6–12.

[18] S. Reis, C. Pasareanu, R. Abreu, and H. Erdogmus, "Secomlint: A linter for security commit messages," *arXiv preprint arXiv:2301.06959*, 2023.

[19] L. Flynn and W. Klieber, "Using llms to automate static-analysis adjudication and rationales," 2024.

[20] Q. Mao, Z. Li, X. Hu, K. Liu, X. Xia, and J. Sun, "Towards effectively detecting and explaining vulnerabilities using large language models," *arXiv preprint arXiv:2406.09701*, 2024.

[21] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software quality journal*, vol. 21, pp. 421–443, 2013.

[22] A. V. Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, "Variability-aware static analysis at scale: An empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 4, pp. 1–33, 2018.

[23] R. van Tonder and C. Le Goues, "Tailoring programs for static analysis via program transformation," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 824–834.

[24] A. Silva, S. Fang, and M. Monperrus, "Repairllama: Efficient representations and fine-tuned adapters for program repair," *arXiv preprint arXiv:2312.15698*, 2023.

[25] Z. C. Lipton, "The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery." *Queue*, vol. 16, no. 3, p. 31–57, Jun. 2018. [Online]. Available: https://doi.org/10.1145/3236386.3241340

[26] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[27] N. Tihanyi, T. Bisztray, M. A. Ferrag, R. Jain, and L. C. Cordeiro, "Do neutral prompts produce insecure code? formai-v2 dataset: Labelling vulnerabilities in code generated by large language models," *arXiv preprint arXiv:2404.18353*, 2024.

[28] [Online]. Available: https://owasp.org/www-project-code-review-guide/

[29] T. Szabó, "Incrementalizing production codeql analyses," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1716–1726.

[30] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi, and Y. Higo, "An empirical study of security warnings from static application security testing tools," *Journal of Systems and Software*, vol. 158, p. 110427, 2019.

[31] J. V. Bradley, "Complete counterbalancing of immediate sequential effects in a latin square design," *Journal of the American Statistical Association*, vol. 53, no. 282, pp. 525–528, 1958.

[32] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[33] V. Clarke and V. Braun, "Thematic analysis," *The journal of positive psychology*, vol. 12, no. 3, pp. 297–298, 2017.

[34] (2025, May) Multinomial logistic regression — r data analysis examples. [Online]. Available: https://stats.oarc.ucla.edu/r/dae/multinomial-logistic-regression/

[35] D. A. Dillman, J. D. Smyth, and L. M. Christian, "Internet, phone, mail, and mixed-mode surveys: The tailored design method," *Indianapolis, Indiana*, 2014.

[36] N. Tihanyi, T. Bisztray, R. Jain, M. A. Ferrag, L. C. Cordeiro, and V. Mavroeidis, "The formai dataset: Generative ai in software security through the lens of formal verification," in *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2023, pp. 33–43.

[37] I. Perez, F. Dedden, and A. Goodloe, "Copilot 3," Tech. Rep., 2020.

[38] E. Brewer, R. Pike, A. Arya, A. Bertucio, and K. Lewandowski, "Know, prevent, fix: A framework for shifting the discussion around vulnerabilities in open source," https://security.googleblog.com/2021/02/know-prevent-fix-framework-for-shifting.html, February 2021.

[39] S. Reis, R. Abreu, H. Erdogmus, and C. Păsăreanu, "Secom: Towards a convention for security commit messages," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 764–765.

[40] T. Tan and Y. Li, "Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1093–1105.

[41] Y. Tymchuk, M. Ghafari, and O. Nierstrasz, "Jit feedback: What experienced developers like about static analysis," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 64–73.

[42] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[43] P. J. Chapman, C. Rubio-González, and A. V. Thakur, "Interleaving static analysis and llm prompting," in *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, 2024, pp. 9–17.

[44] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Assisting static analysis with large language models: A chatgpt experiment," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 2107–2111.

[45] J. Yan, J. Huang, C. Fang, J. Yan, and J. Zhang, "Better debugging: Combining static analysis and llms for explainable crashing fault localization," *arXiv preprint arXiv:2408.12070*, 2024.
[46] Z. Li, S. Dutta, and M. Naik, "Iris: Llm-assisted static analysis for detecting security vulnerabilities," in *The Thirteenth International Conference on Learning Representations*, 2025.
[47] G. Fan, X. Xie, X. Zheng, Y. Liang, and P. Di, "Static code analysis in the ai era: An in-depth exploration of the concept, function, and potential of intelligent code analysis agents," *arXiv preprint arXiv:2310.08837*, 2023.