Democratizing False Positives Filtering Through Learning Assisted Reasoning

Mingjie Shen[1](✉)[0009−0003−4393−6992], Sai Ritvik Tanksalkar[1][0009−0008−6428−0511], Christophe Hauser[2][0000−0001−7698−8041], and Aravind Machiry[1][0000−0001−5124−6818]

[1] Purdue University, West Lafayette, IN, USA
{shen497,stanksal,amachiry}@purdue.edu
[2] Dartmouth College, Hanover, NH, USA
Christophe.Hauser@dartmouth.edu

Abstract. Static Application Security Testing (SAST) tools generate high rates of False Positives (FPs), creating a severe triage burden for developers. Although Large Language Models (LLMs) offer a promising way to automate triage, they struggle on large codebases due to context limitations, and the evidence needed to decide whether an alert is real is hard to determine *a priori*.

We introduce Aithos, an LLM-powered agentic framework that dynamically retrieves precise program context to overcome these limitations. The agent invokes analysis primitives to gather evidence and classify SAST alerts based on natural-language guidelines. A feedback loop further improves accuracy by converting developer corrections into reusable, repository-specific examples. Evaluated on a curated dataset of alerts from four popular tools across real-world C/C++ projects and the Juliet suite, Aithos significantly outperforms a state-of-the-art baseline relying on static context gathering. Aggregated over the real-world datasets, Aithos preserves 97.5% of true positives (vs. 57.9%) while filtering 78.0% of false positives, with both methods powered by Claude Sonnet 4. Furthermore, Aithos's specialized primitives enable safer triage than generic repository-navigation tools (*e.g.,* `Grep`). In addition, Aithos uncovered 90 previously unknown bugs in 28 popular Debian packages, demonstrating that an agentic, tool-augmented approach can substantially reduce the manual SAST triage bottleneck.

Keywords: Static application security testing · False-positive triage · Large language models · Agentic systems · Program analysis

1 Introduction

Static Analyses (StAs) [76] play an important role in various aspects of software security, ranging from vulnerability detection [65] to semantic equivalence checking [6]. Security vulnerability detection [17,56] is one of the most important and popular applications of StA, with numerous State of the Practice (SOTP) tools [27,15], commonly referred to as SAST tools. However, False

Positives (FPs) are a well-known problem with SAST tools and one of the main reasons [58,73,1,63] why developers do not use these tools.

Existing techniques to tackle FPs can be broadly categorized into three categories: ranking, interactive triaging, and domain specialization. *(i) Ranking techniques*, such as z-ranking, rank warnings based on statistical information about the likelihood of the corresponding bug [24] or code features [51]. However, these techniques still require manual identification of relevant features, which depend on the bug and Program Under Test (PUT) – requiring *significant analyst (i.e., tool developer) effort*. On the other hand, ranking does not eliminate FPs; the developer still needs to go through all alerts, resulting in *medium effort*. *(ii) Interactive triaging* techniques try to incorporate user feedback into the triaging process. Specifically, they group alerts with similar features and, based on positive or negative feedback on one alert, promote or penalize other alerts with similar features. By design, these techniques need to be customized for each SAST tool because each interactive triaging technique requires different features. For instance, ARBITRAR [30] requires a symbolic trace, whereas URSA [81] and EUGENE [42] require the results to be expressed as Datalog relations. Although the developer effort is lower, the *analyst effort is high* as the tools should be redesigned to use these triaging techniques. *(iii) Domain specialization* techniques retrofit standard techniques for specific types of programs. For instance, DRCHECKER [41] customizes well-known taint tracking for Linux kernel device drivers and found ∼150 new vulnerabilities. Similarly, BOOTSTOMP [60] customized taint tracking for bootloaders and found six new vulnerabilities. These techniques require *high analyst effort* as they require a comprehensive understanding of the target domain and its mapping to optimizations in the analysis techniques. *We aim to create FP filtering techniques that require minimal developer and analyst effort.*

Effective triaging of SAST alerts relies heavily on project-specific context and domain assumptions (*e.g.,* usage constraints), which are often best known by the developers of the code. Developers can often explain why a flagged pattern is safe within their application's logic, but *codifying* this rationale to systematically suppress FPs is substantially harder. For instance, developers may say that an uninitialized-variable alert for v at line ℓ is an FP if v is initialized along all valid paths reaching ℓ. Yet expressing such reasoning in a form consumable by the SAST tool often requires modifying the analysis using complex, custom Domain-Specific Languages (DSLs), such as the functional query language used in CODEQL [4]. Thus, *effective suppression of FPs requires bridging a gap between developers' project knowledge and analysts' static-analysis techniques* – a challenging task.

Large Language Models (LLMs) show significant potential for identifying software vulnerabilities [20]. However, they struggle with rigorous analysis, often exhibiting unfaithful and non-robust reasoning [82,69]. A key bottleneck in existing LLM-based alert inspection techniques is their reliance on *static, one-shot context construction*. For example, LLM4SA [75] inspects each alert by supplying pre-extracted code slices obtained via warning-guided dependence traversal;

similarly, Mohajer *et al.* [48] use a fixed prompt input assembled *a priori*. While Flynn and Klieber [14] allow the model to request additional function bodies, their approach does not provide a general mechanism to retrieve other relevant evidence on demand (*e.g.,* upstream call sites or global definitions). This creates a critical dilemma: *it is difficult to predict exactly what information is required to verify a specific alert before the analysis begins.* If the pre-computed slice omits a distant variable definition or a complex macro expansion, the LLM lacks the agency to recover, leading to hallucinations or incorrect triaging [19]. Conversely, providing excessive or irrelevant information not only wastes the finite token budget but also degrades performance by introducing noise [69,33].

In this paper, we propose AITHOS, an *agentic* framework [78] that overcomes the limitations of static context selection by enabling an LLM to retrieve evidence *on demand*. We configure the agent with a suite of 10 *analysis primitives* (*e.g.,* `find_var_definitions` identifies all definitions of a variable), enabling it to iteratively explore the PUT. The agent can invoke these primitives and combine their results in different ways to realize diverse verification strategies and gather exactly the program facts needed for each decision. In addition, AITHOS accepts *natural-language guidelines* written by developers to describe their triaging criteria. During triage, AITHOS interprets the guidelines and uses the analysis primitives to retrieve the relevant program facts, determining whether each SAST alert is valid. This design *reduces effort for both analysts and developers*: analysts only implement and expose primitives, without having to anticipate the alert-specific reasoning required to filter diverse FPs, while developers express triaging criteria in natural language instead of modifying the SAST tool's DSL. We also introduce an interactive mode, enabling developers to provide project-specific feedback when AITHOS fails to filter an FP. Our design is generic and applicable to any SAST alert with minimal modification. We curate a dataset of SAST alerts by running CODEQL, Cppcheck, Clang Static Analyzer (CSA), and Infer on a collection of real-world C/C++ projects and the Juliet test suite. Our evaluation shows that AITHOS substantially outperforms a state-of-the-art LLM-based baseline, LLM4SA [75]. On real-world projects, when both approaches use Claude Sonnet 4 as the underlying LLM, AITHOS preserves 97.5% of True Positives (TPs), compared with 57.9% for the baseline, while also filtering 78.0% of FPs. We further find that AITHOS's specialized primitives enable safer triage than generic repository-navigation tools (*e.g.,* `Grep`) used by coding agents such as Claude Code [3], with the largest gains for weaker models. Furthermore, in real-world Debian packages, AITHOS identified 102 bugs, including 90 previously unknown ones, 12 of which have already been confirmed by developers. In summary, the following are our contributions:

- The design and implementation of AITHOS, a novel agentic framework that leverages LLMs to automatically filter FPs from SAST alerts.
- A catalog of guidance prompts for accurately triaging common C/C++ vulnerability classes and a set of analysis primitives that enable the LLM agents to query essential program properties.

- A novel feedback mechanism enabling the agent to learn project-specific coding patterns from developer corrections.
- A new dataset of SAST alerts from real-world C/C++ applications, with ground-truth labels to facilitate future research.
- Discovery of 90 previously unknown bugs in Debian packages, 12 of which have already been confirmed.

2 Background and Motivation

This section provides the necessary background and motivation for our work.

2.1 SAST Tools and False Positives (FPs)

Static Application Security Testing (SAST) tools detect security vulnerabilities early in the lifecycle by analyzing program artifacts (source, bytecode, or binaries) without executing them [12]. These tools often make conservative assumptions about all possible execution paths and data flows to remain scalable [35], resulting in a high volume of FPs: alerts for vulnerabilities on code paths that are infeasible in practice [16,55].

Consequently, the high FP rate is a well-documented barrier to SAST adoption, causing developers to disregard alerts and doubt the tools' efficacy [8,61]. Empirical studies consistently show that developers are hesitant to use SAST tools due to the overwhelming number of false alarms they must triage [22,58,73]. This friction is a key reason why advanced SAST adoption remains low in many ecosystems [7,63]. Even when practitioners tolerate FPs to avoid missing critical vulnerabilities, the time wasted investigating spurious warnings remains a significant drain on developer productivity [1].

2.2 Large Language Models and Prompts

A Large Language Model (LLM) is a large neural sequence model trained on extensive text corpora to predict and generate fluent, human-like text [72]. *Prompting* steers its behavior by supplying input text that may include instructions, examples, and context [34]. Prompts can also establish a *persona* – a profile or character (*e.g.,* software developer or math tutor) – to shape tone and decision-making and improve coherence and engagement [80].

In modern LLM applications, prompts are often structured into roles, most commonly the *system* and *user* prompts [52]. The system prompt provides high-level, persistent guidance that defines the model's overall behavior and persona across the conversation [44]. The user prompt then specifies a concrete request at each turn. Together with the model's prior responses, this role-structured context determines how the LLM interprets the task and produces its next output.

2.3 Motivation

With the rise of powerful LLMs [69,20,82], a natural question is whether they can automatically triage SAST alerts by separating TPs from FPs. To test the standard "static context" approach, we provided several SOTP LLMs with the alert metadata (*i.e.,* bug type and warning message) and a local code snippet around the reported location, and evaluated them on CODEQL [4] alerts from the Juliet test suite [9] and Debian packages [66]. This baseline performs poorly without deeper context or triaging criteria: for example, GPT-4o filters only 20.4% of false positives overall and 12.5% on Debian; similar behavior holds across models (§6.4). These results suggest that alert metadata plus local code is insufficient for reliable automated triage.

Recent techniques such as LLM4SA enrich the prompt with a larger but still *fixed* per-alert context (*e.g.,* via dependence-graph traversal) and then make a one-shot decision. Our evaluation shows this design can be overly aggressive: even with Claude Sonnet 4, LLM4SA preserves only 61.3% of TPs on CODEQL alerts from real-world projects (§6.3), filtering many true vulnerabilities.

To address this "missing context" problem, we develop AITHOS that equips an LLM with (i) *guidance prompts* that state triaging criteria in natural language and (ii) *analysis primitives* that provide deterministic program evidence on demand. This guided, tool-augmented design enables context-adaptive evidence gathering and achieves the accuracy needed for practical triage.

2.4 Notations and Definitions

We use the following notations and definitions to explain our technique:

- **Program Under Test (PUT):** A program analyzed by Static Application Security Testing (SAST).
- **Alert:** A warning or error message emitted by a SAST tool describing a bug, *i.e.,* a violation of a correctness rule (*e.g.,* use of an uninitialized variable).
- **SARIF file:** A SARIF (Static Analysis Results Interchange Format) file uses a standardized, JSON-based format for representing findings from SAST tools [46]. Most tools can emit SARIF natively or via simple converters, making it easy to produce alerts in SARIF.
- **LLM:** A Transformer-based language model, pretrained on massive text corpora for language understanding and generation [47].
- **LLM Agent:** A system that uses an LLM to autonomously achieve goals through iterative reasoning, action, and observation [78].
- **Developers and Analysts:** *Developers* are maintainers of the PUT and primary end users of SAST tools on that codebase; they possess the project-specific knowledge required to triage alerts. *Analysts* are static-analysis experts who design and implement SAST rules and analyses.
- **Analysis Primitives:** Standalone program-analysis tools that the LLM agent can call to fetch concrete information about the PUT. Each analysis primitive is single-purpose, orthogonal, and intended to be composed like building blocks to answer higher-level triage questions.

– **Initial Triaging Information or Guidance Information (GP):** Rule-specific, natural-language criteria that describe what should be kept as a TP and what should be discarded as an FP for any alert emitted by the rule.
– **Correction Guidance or Developer Feedback:** Repository-specific additions to the guidance prompt, supplied by developers after a mis-triage. Feedback can be a simple label or a rationale.

3 Overview of AITHOS

Figure 1 illustrates the high-level workflow of AITHOS. Given SAST results in SARIF format, the SARIF Analyzer (§4.1) parses the file and identifies individual alerts. For each alert, it constructs an initial *Triaging Prompt* (TRP) by combining the alert metadata, relevant source-level context, and the GP for the alert type. For example, for the uninitialized-local alert from `findutils-4.8.0` (Listing 1), the TRP includes the bug type, the corresponding GP, the source location, and the warning message, as shown in Prompt 1.

```
518:   int left_cost, right_cost;
519:   const char *reason = NULL;
...
551:   if (!reason)
552:     {
553:     ✔ left_cost = worst_cost (*pl);
554:       right_cost = worst_cost (*pr);
555:       ...
560:     }
561:   if (!reason)
562:     {
563:       bool want_swap;
564:
565:       ⚠if (left_cost == right_cost)
```

Listing 1. CODEQL reports that `left_cost` may be uninitialized at line 565. This is a false positive: if control reaches line 565, then `left_cost` must have been defined at line 553, since both lines are guarded by the same condition (*i.e.,* `!reason`).

The TRP is then given to an LLM agent equipped with analysis primitives (§4.2) through the Plugin Interface (§4.2). The agent invokes these primitives on demand to retrieve relevant facts about the PUT. In our running example, it first calls `get_path_constraint` and finds that both the initialization at line 553 and the use at line 565 are guarded by the same condition, `!reason`. It then calls `check_always_implying` to confirm that reaching the use implies reaching the initialization.

Based on the gathered evidence, the agent classifies the alert as TP, FP, or Uncertain (UC) and provides a brief rationale. For the alert in Listing 1, it correctly classifies the warning as an FP, since the variable must be initialized whenever execution reaches the reported use. If this reasoning is still incorrect, developers can optionally provide feedback, which AITHOS uses as additional examples to improve later triage.

4 AITHOS Design

There are four main components in AITHOS.

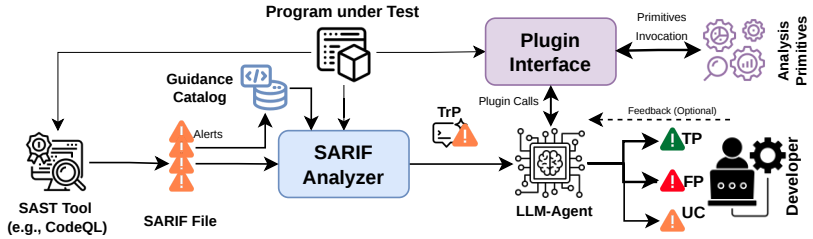

Fig. 1. Architecture of AITHOS. For each alert, the agent receives a TRP, invokes analysis primitives (*e.g.,* path constraints, implication checks) to gather evidence, and outputs a label with rationale. Optional developer feedback becomes dynamic few-shot examples for future alerts.

Prompt 1: TRP for our running example (Listing 1)

Type of bug: cpp/uninitialized-local.
Guidance on triaging this type of bug: The warning at a specific source line is a false positive if the variable is always initialized along all the paths that reach that line.
Source location: find/tree.c:565
Message: The variable left_cost may not be initialized at this access.
Code context: [omitted]
Task: Please classify this alert as TP, FP, or UNCERTAIN, and provide your reasoning.

4.1 SARIF Analyzer

This component takes in a SARIF file and the source location of the PUT. It parses the file and identifies all the alerts. Each alert a is a tuple (m, ℓ, t), where m, ℓ, and t are the violation message, line number, and type of the alert, respectively. It creates a TRP using (m, ℓ, t), initial code context c_ℓ (20 lines around ℓ, *i.e.,* source lines from $\ell - 10$ to $\ell + 9$), and the GP based on t. The TRP is generated from the following template (illustrated in Prompt 1): "*Bug type: t. Guidance:* GP. *Source location: ℓ. Message: m. Code context: c_ℓ. Please classify the alert as TP/FP/UC and provide your reasoning.*"

Guidance Catalog We created a catalog of GPs for common types of alerts in C/C++ codebases [31,32]. We index each GP using identifiers from widely used SAST tools, *i.e.,* CODEQL [4] and CSA [67]. However, the catalog itself is tool-agnostic: each GP describes decision criteria for triaging the underlying bug type, independent of which SAST tool reports it. Table 1 shows the GPs; the right column shows simple decision criteria for triaging an alert. We also maintain one good and one bad example of each alert type. Listing 2 shows examples we maintain for a couple of our alerts. The *GOOD* examples are valid, and the

corresponding alerts should be considered FPs, whereas the *BAD* examples are invalid and indicate true bugs (*i.e.,* TPs).

4.2 Plugin Interface

This component exposes a common interface to access various analysis information about PUT using corresponding analysis primitives. The design of the plugin interface is primarily focused on enabling LLMs to invoke various analysis primitives easily. The plugin interface exposes simple text-based plugin calls and handles the invocation of the corresponding analysis primitives to retrieve the information. Specifically, the plugin interface prepares the necessary information for the analysis primitives (*e.g.,* LLVM bitcode [36], ctags index [70]), invokes them, and provides the results in JSON format. For instance, a call to `find_var_decl` may return `{"file": "dmidecode.c", "line": 225}`. Any invalid or incorrect invocations of analysis primitives (*e.g.,* requesting definitions of a nonexistent variable) will be reported back in an instructive manner, enabling correct invocations. For instance, if the agent incorrectly calls `find_var_decl` on a function (`read_file`) instead of a variable, it receives an instructive error guiding it toward the correct primitive for its next attempt, *i.e.,* "*Unable to locate a declaration for 'read_file'. Please verify the name refers to a variable. If it is a function, use find_callee.*"

```
// BAD: unbounded allocation; factor
// may be very large (overflow risk)
int f = atoi(getenv("FACTOR"));
char **root_node =
  (char**)malloc(f * sizeof(char*));

// GOOD: bound allocation size
int f = atoi(getenv("FACTOR"));
if (f > 0 && f <= 1000) {
  char **root_node =
  (char**)malloc(f * sizeof(char*));
}
```

(a) `cpp/uncontrolled-alloc-size`

```
// BAD: unbounded write may overflow
↪  buffer
char buf[80];
sprintf(buf, "Hi %s!", userName);

// GOOD: enforce destination bound
char buf[80];
snprintf(buf, sizeof(buf), "Hi %s!",
↪  userName);

// GOOD: allocate enough space for
// the string and null terminator
char *d = (char
↪  *)malloc(strlen(userName)+1);
if (!d) return;
strcpy(d, userName);
```

(b) `cpp/unbounded-write`

Listing 2. Few-shot examples from the guidance catalog. Each subfigure shows a *bad* snippet (violates guidance, TP) and a *good* snippet (satisfies guidance, FP).

Analysis Primitives These are deterministic, atomic static analysis utilities designed to compute specific semantic information about the PUT.

Derivation Methodology. To enable the LLM agent to mimic how security experts triage alerts, we systematically analyzed our guidance catalog for common C/C++ vulnerability classes (see §4.1). We identified the types of program facts a human expert would consult to apply each guideline during manual verification. We observed that these facts (*i.e.,* information about PUT) consistently fall into three categories: (i) *Structural Information (SI)*: definitions, references, and call relations; (ii) *Path Analysis (PA)*: reachability and control-flow constraints;

Table 1. Rule-specific guidance for common C/C++ alert types.

Rule ID	Guidance (decision criterion)
`uncontrolled-alloc-size`	*FP* if the memory allocation size has upper bounds.
`invalid-pointer-deref`	*FP* if the pointer cannot be used to access past the end of the allocation.
`missing-check-scanf`	*FP* if `scanf`'s return value is compared with the expected item count before use.
`offset-use-before-check`	*FP* if a bounds check keeps the index within limits before the access.
`unbounded-write`	*FP* only if destination buffer size $\geq$ source length.
`overrun-write`	*FP* only if destination buffer size $\geq$ source length.
`overrunning-write`	*FP* only if destination buffer size $\geq$ source length.
`overrun-write-float`	*FP* only if destination buffer size $\geq$ formatted output length.
`bad-strncpy-size`	*FP* only if the limit passed to `strncpy` is the destination's capacity.
`uninitialized-local`	*FP* if the variable is always initialized on all paths reaching the use.
`unsafe-strcat`	*FP* if source size $\leq$ the remaining space in the destination buffer.
`memory-may-not-free`	*FP* if along feasible paths, the allocation is either freed or its ownership is transferred.
`use-after-free`	*FP* if no feasible path frees and then uses the same allocation without an intervening reallocation.
`double-free`	*FP* if no feasible path frees the same allocation twice without an intervening reallocation.
`null-pointer-deref`	*FP* if the pointer cannot be `NULL` at the dereference site.
`divide-zero`	*FP* if the divisor cannot be zero at the division site.

and (iii) *Bounds Information and Expression Comparison (BIC)*: value-range reasoning and buffer-size inference. Based on this taxonomy, we implemented a minimal set of primitives to cover these recurring reasoning patterns. We froze the primitive set before quantitative evaluation to avoid post-hoc tuning to our benchmarks.

Usage Example. Consider a potential buffer overflow alert in the following code snippet: `void f(char *s) {char d[10]; strcpy(d, s);}`. To verify this, an expert would first need to establish the size of the destination buffer `d` (SI) and then determine if the length of the source string `s` can exceed this limit (BIC). In our framework, the agent mimics this workflow by invoking the corresponding primitives: first, it calls an SI primitive to locate the definition of `d`; second, it employs a BIC primitive (`infer_bounds`) to compare the sizes. If the inferred bound for `s` exceeds 10, the agent concludes this is a TP.

Table 2 lists all analysis primitives with their inputs, outputs, and implementations.

4.3 LLM Agent

The agent is an LLM that triages a given alert as TP, FP, or UC using the corresponding TRP. Rather than providing the LLM with the entire PUT, we expose the codebase through a plugin interface that offers a set of *analysis primitives*. These primitives serve as the agent's sole mechanism for retrieving program facts about the PUT. As discussed in §4.2, the primitive set is designed to cover the program facts typically needed for manual triage, and thus provides the evidence on which the agent bases its decisions. Each request to the LLM includes the schemas of the available primitives [53], informing the agent which tools it may invoke and how to invoke them.

Table 2. Analysis primitives with description, signature (inputs → output), procedural scope (Proc.), and implementation strategy (Impl.). Notation: *fp* file path, ℓ line number, *var* variable name, *fun* function name, *name* variable or function name, *idx* parameter index, *expr* symbolic expression. Subscripts distinguish multiple arguments of the same kind (*e.g.,* $expr_1$, $expr_2$). In the Impl. column, FS denotes file-system access and SrcPass a source-level analysis pass. See §4.2.

	Name	Description	Signature	Proc.	Impl.
Structural Information (SI)					
P1	`dump_source_file`	Slice of a source file, with line numbers.	$(fp, \ell_1, \ell_2) \to$ string	–	FS
P2	`find_var_decl`	Declaration site of a local or global variable.	$(fp, \ell, var) \to (fp, \ell)$	–	LLVM
P3	`find_callers`	Callers of a target function.	$(fp, \ell) \to \text{list}\langle(fun, fp, \ell)\rangle$	Inter	LLVM
P4	`find_callee`	Function called at a source line.	$(fp, \ell) \to (fun, fp, \ell)$	Inter	LLVM
P5	`ctags_readtags`	Definition sites of an identifier, via ctags.	$(name, fp) \to \text{list}\langle(fp, \ell)\rangle$	–	Ctags
P6	`find_var_definitions`	All definitions that reach a variable use.	$(fp, \ell, var) \to \text{list}\langle(fp, \ell)\rangle$	Intra	LLVM
Path Analysis (PA)					
P7	`get_path_constraint`	Path condition required to reach a source line.	$(fp, \ell) \to expr$	Intra	LLVM
P8	`check_always_implying`	Whether one constraint always implies another.	$(expr_1, expr_2) \to$ bool	–	Z3
Bounds Information and Expression Comparison (BIC)					
P9	`infer_bounds`	Bounds of a function's pointer parameter.	$(fp, fun, idx) \to expr$	Inter	SrcPass
P10	`check_le`	Whether $expr_1 \leq expr_2$ always holds.	$(expr_1, expr_2) \to$ bool	–	Z3

As mentioned in §2.2, the LLM receives a system prompt and a user prompt. We perform a one-time configuration via the system prompt that instructs the agent to adopt a security-researcher persona (Prompt 2). For each alert, the corresponding TrP (described in §4.1) is provided as the user prompt.

4.4 Developer Interaction

In the default setting, our LLM agent returns a triage decision (TP/FP/UC) along with a brief justification. However, some alerts are based on *repository-specific assumptions*, such as usage constraints, API contracts, or trust boundaries, which are valid for the PUT but difficult to infer automatically. In these cases, general security reasoning may be sound in isolation yet still produce the wrong label for a particular project.

For example, Figure 2a shows a warning where CodeQL flags the use of `sscanf` with `%s` as potentially unsafe because it may overflow `m.name`. In this PUT, the input is read from `/proc/net/dev_mcast` via `buf`. The project's threat model

Prompt 2: System prompt used by AITHOS

You are a software security researcher tasked with classifying SAST alerts on C/C++ code. Each alert must be classified as one of: TP (true positive): the code violates the guidance provided by the user; FP (false positive): the code follows the guidance; UNCERTAIN: insufficient information to decide. Each user input will include: the bug type, source file name and line number of the potential bug, the alert message, code context around the potential bug, and examples showing similar cases and their correct classifications.
Guidelines: Focus only on the specified bug type and location. Don't speculate about future code changes. Think step by step; use provided tools to obtain necessary information.

treats this source as trusted and assumes the kernel-provided format constrains the interface name length (*e.g.,* to at most `IFNAMSIZ`), making the reported overflow infeasible and the alert an FP. Such assumptions are defined by developers and can vary across projects; without such domain knowledge, they cannot be reliably derived from the code or from the general guidance prompts (Table 1) alone.

To handle such cases, AITHOS provides an optional feedback mode in which developers override an incorrect triage by supplying brief domain knowledge. Upon receiving feedback, the agent performs a self-reflection step based on (i) its initial triage and supporting evidence and (ii) the developer's corrected label and rationale. We prompt the agent to identify the misconception or missing evidence in its earlier reasoning and to conclude with a concrete improvement. This produces a concise reflection [64] that explains the mistake in light of the developer feedback, as illustrated in Figure 2b.

To avoid unintended cross-project bias, we scope feedback *to the current PUT*. We store each interaction as a tuple *(rule ID, code context, corrected label, reflection)* and treat it as a repository-specific exemplar. During subsequent triage *within the same PUT*, the agent retrieves exemplars that match the current alert's rule ID and injects them into the TRP (§4.1). We do not claim these exemplars generalize across projects; instead, they provide lightweight, *repository-specific* customization that reduces repetitive manual corrections.

5 Implementation

Most of AITHOS, except for analysis primitives, is implemented using Python. The SARIF Analyzer is a simple Python script, and interaction with the LLM agent is implemented using the LangChain library [25], which enables us to use different LLMs in a plug-and-play manner. Communication between the Python-based LangChain framework and the Plugin Interface is handled via JSON-RPC [23]. To enable program-wide analysis, the whole-program bitcode for each repository was generated using WLLVM [71].

```
// iproute2-5.15.0/ip/ipmaddr.c
struct ma_info { ... char name[IFNAMSIZ];
/* IFNAMSIZ is typically 16 */ ... };

static void read_dev_mcast(...) {
    FILE *fp = fopen("/proc/net/dev_mcast",
    ↪  "r");
    ...
    while (fgets(buf, sizeof(buf), fp)) {
        struct ma_info m = ...;
        ...
        sscanf(buf, "%d%s%d%d%s", &m.index,
               m.name⚠, &m.users, ...);
    }
}
```

(a) CodeQL flags a potential buffer overflow in the `sscanf` call, targeting `m.name`.

```
Rule: cpp/unbounded-write @ ip/ipmaddr.c:101
Initial label: TP
Rationale: "%s" in sscanf lacks bounds;
↪  m.name is an array of size IFNAMSIZ.

Developer> FP. /proc/net/dev_mcast is
↪  kernel-generated and guarantees
interface name is no longer than IFNAMSIZ.

Reflection:
- Misconception: Overgeneralized ‘‘%s is
↪  unsafe’’ without source constraints.
- Missing evidence: Ignored kernel format
↪  contract and field bounds.
- Next time: Check data-source specs
↪  (kernel/procfs formats) before labeling.
```

(b) Developer correction and the agent's reflection.

Fig. 2. End-to-end interactive feedback: (a) a SAST alert; (b) a domain-specific correction and the LLM's reflection.

As summarized in the **Impl.** column of Table 2, our analysis primitives are implemented using LLVM [37] and Clang [68] passes, file-system access, Z3 [49], and Ctags [70]. The LLVM-based primitives are implemented as LLVM 17 module passes.

Most of the analysis primitives are implemented using standard techniques. For instance, `find_callers` and `find_callee` are implemented using call-graph analysis. Similarly, `get_path_constraint` is implemented using a control-dependence graph, *i.e.,* by identifying all conditional statements along the path from the function entry point to the statement on line ℓ.

We use correlation analysis [57] based on 3C [39] to implement our `infer_bounds` primitive. We start with the seed bounds, *i.e.,* bounds available from the program source, such as allocators (*e.g.,* `p = malloc(n);`, which indicates that the length of the buffer pointed to by `p` is `n`) and static declarations (*e.g.,* `int arr[10]`, where the bound of `arr` is 10). We propagate these bounds through assignments and in-scope variables. For instance, if we have a call `foo(p, n)`, then the above seed bounds imply that the size of the buffer pointed to by the first parameter `p` is `n`.

For implication checking (*i.e.,* `check_always_implying`), we follow the approach of SPIDER [40] and convert the C expressions in the two path constraints ($expr_1$ and $expr_2$) into symbolic expressions. Then we ask Z3 for the satisfiability of the inverse, *i.e.,* whether $expr_1 \wedge \neg expr_2$ is satisfiable. If it is unsatisfiable, this means the implication, *i.e.,* $expr_1 \implies expr_2$, always holds, and we return `True`; otherwise, we return `False`. We follow a similar approach for `check_le`.

Our implementation comprises 4.1k SLoC of Python and 2.1k SLoC of C++.

6 Evaluation

We aim to evaluate the system's effectiveness in triaging static analysis warnings and in reducing effort in real-world bug finding. Specifically, we seek to answer the following research questions:

- RQ1: How effective is AITHOS in reducing FPs while preserving TPs, and how does it compare with existing state-of-the-art approaches?
- RQ2: What is the contribution of guidance and primitives to effectiveness?
- RQ3: How effective is human-in-the-loop feedback in boosting effectiveness?
- RQ4: Does AITHOS reduce effort in real-world bug finding?
- RQ5: How well does AITHOS generalize across LLMs? (in Appendix)
- RQ6: How easily can AITHOS be extended to new SAST tools? (in Appendix)

To answer these questions, we built a comprehensive evaluation dataset and constructed reliable ground-truth labels.

6.1 Dataset and Ground Truth Construction

A rigorous evaluation requires a dataset of alerts with accurate ground-truth labels. During our preliminary investigation, we discovered that existing datasets of SAST alerts on real-world repositories, specifically DEFP [51], cover limited bug types and contain labeling errors. Therefore, we constructed a high-quality dataset that (i) targets the common C/C++ vulnerability classes in our guidance catalog (§4.1) and (ii) establishes reliable ground truth.

Curating a CODEQL Dataset with Ground Truth We used CODEQL (v2.15.1) to scan three software collections. The first is the Juliet test suite [9], a synthetic benchmark of 64,099 test cases with built-in labels. The second is a set of 26 widely used Debian packages [66], representing complex real-world systems software (median 36K, max 378K SLoC). The third is a set of 10 open-source embedded software repositories [63] (median 34K, max 7,178K SLoC), included to capture the distinct challenges of SAST on embedded codebases.
Manual Labeling Protocol. To establish reliable ground truth for the real-world repositories, two graduate students with extensive experience in C/C++ security auditing independently labeled each alert. This initial phase yielded a Cohen's κ of 0.82, indicating *almost perfect agreement*. Following this, all disagreements were reviewed in a consensus meeting with a third senior researcher to adjudicate and finalize the labels. Table 3 reports the resulting numbers of TPs and FPs.

Reusing the LLM4SA Real-World Dataset To benchmark against prior work, we additionally evaluate our approach on the dataset released by LLM4SA [75], which comprises 11 Debian packages and 3 embedded OS projects analyzed by Cppcheck [43], CSA [67], and Infer [13]. This dataset targets six vulnerability classes (null pointer dereference, uninitialized variable, use-after-free, divide by

zero, memory leak, and buffer overflow), all of which are covered by our guidance catalog (Table 1). We used a conservative reuse protocol. The LLM4SA artifact release provides raw tool outputs and confirmed true positives, but it does not explicitly enumerate false positives and only partially documents the alert preprocessing pipeline used in the paper. To ensure faithful reuse, we conservatively re-implemented the described preprocessing and retained only tool–project subsets whose reconstructed alert counts exactly matched the published figures, reproducing over 85% of the CSA and Infer subsets and approximately 40% of the Cppcheck subsets; subsets with disparate counts were excluded to avoid introducing unverified or irrelevant alerts. During verification, we also identified and corrected 31 labeling errors in the LLM4SA ground truth, including misclassified instances of standard bug patterns (*e.g.,* misuse of `p = realloc(p, ...)`); every amendment is explicitly documented and justified in our released artifact (see the Data Availability statement).

6.2 Evaluation Metrics

To evaluate whether AITHOS is effective in filtering out FPs, we measure the FP Filtered Rate = $\frac{FP_{\text{filt}}}{FP}$, where FP denotes the number of FPs in the ground truth and FP_{filt} the number of FPs filtered. A higher value indicates that AITHOS successfully removes noisy alerts.

To evaluate TP preservation, we use two metrics. Let TP count ground-truth TPs; TP_{keep} and TP_{unc} count those labeled by AITHOS as TP and UC, respectively. The strict rate is TP Preserved Rate = $\frac{TP_{\text{keep}}}{TP}$. The conservative rate is TP Pres. (Cons.) = $\frac{TP_{\text{keep}}+TP_{\text{unc}}}{TP}$. The strict metric reflects how often AITHOS preserves TPs *confidently*, while the conservative metric treats UC as *not filtered* (*i.e.,* still available for manual inspection), providing an upper bound on TP preservation.

6.3 RQ1: How Effective Is AITHOS in Triaging SAST Alerts Compared with LLM4SA?

Methodology We first evaluate whether AITHOS improves practical triage quality over the state-of-the-art LLM-based baseline, LLM4SA. To ensure comparability, we power both AITHOS and LLM4SA with the same underlying LLM, Claude Sonnet 4, which demonstrates strong coding performance [2], and use its default temperature. For LLM4SA, we follow its original setup, including its prompt engineering and its code-snippet extraction algorithm. We run both methods on all alerts in our datasets (§6.1) and evaluate their predictions against ground truth using the metrics in §6.2.

Results Table 3 compares AITHOS against LLM4SA across the datasets. Aggregated over the real-world datasets, AITHOS achieves *substantially higher TP preservation* than LLM4SA (97.5% vs. 57.9%) while maintaining comparable FP filtering (78.0% vs. 83.6%). Breaking this down by source, AITHOS outperforms

Table 3. Performance comparison between AITHOS and LLM4SA. Both methods are powered by Claude Sonnet 4. Within each dataset block, **bold** marks the best method for each metric (higher is better for TP Preserved, TP Pres. (Cons.), and FP Filtered; lower is better for FP Uncertain). Ties are bolded.

	TP Preserved		TP Pres. (Cons.)		FP Filtered		FP Uncertain	
Method	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**
CODEQL **on Juliet (TP=1,028; FP=752; Total=1,780)**								
AITHOS	**1,028**	**100.0**	**1,028**	**100.0**	**634**	**84.3**	**1**	**0.1**
LLM4SA	985	95.8	993	96.6	393	52.3	7	0.9
CODEQL **on real-world (RW) repos (TP=62; FP=237; Total=299)**								
AITHOS	**60**	**96.8**	**60**	**96.8**	**207**	**87.3**	3	1.3
LLM4SA	38	61.3	38	61.3	183	77.2	**1**	**0.4**
LLM4SA **Dataset (All 3 tools) (TP=59; FP=1,055; Total=1,114)**								
AITHOS	**58**	**98.3**	**58**	**98.3**	801	75.9	**2**	**0.2**
LLM4SA	32	54.2	32	54.2	**897**	**85.0**	3	0.3
All RW (CodeQL RW + LLM4SA**) (TP=121; FP=1,292; Total=1,413)**								
AITHOS	**118**	**97.5**	**118**	**97.5**	1,008	78.0	5	0.4
LLM4SA	70	57.9	70	57.9	**1,080**	**83.6**	**4**	**0.3**
All Datasets (incl. Juliet) (TP=1,149; FP=2,044; Total=3,193)								
AITHOS	**1,146**	**99.7**	**1,146**	**99.7**	**1,642**	**80.3**	**6**	**0.3**
LLM4SA	1,055	91.8	1,063	92.5	1,473	72.1	11	0.5

LLM4SA on both metrics for the CODEQL dataset. On the LLM4SA dataset, AITHOS preserves 98.3% of TPs (vs. 54.2%); here, LLM4SA is more aggressive at filtering FPs, illustrating a trade-off between FP removal and TP safety. However, on the synthetic Juliet benchmark, AITHOS dominates on both dimensions. When combining all datasets (including Juliet), AITHOS surpasses LLM4SA on all reported metrics, achieving higher TP preservation, higher FP filtering, and lower FP uncertainty.

Illustrative Example: Use-After-Free in vasnprintf Listing 3 illustrates a challenging UAF warning in `diffutils-3.3/lib/vasnprintf.c`. The pointer `dp` iterates over the array `d.dir`. On an error path, the `CLEANUP()` macro frees `d.dir`, yet the code subsequently reads `dp->conversion`. Validating this report requires connecting two dispersed facts: (1) `dp` aliases the freed array `d.dir`, and (2) `CLEANUP()` hides the `free` call. AITHOS confirms this alert by employing targeted primitives to *actively* retrieve the definition of `CLEANUP()` (via `ctags_readtags`) and the initialization of `dp` (via `find_var_definitions`). In contrast, LLM4SA misses the bug, as the crucial aliasing relationship

```
#define CLEANUP()                    \
  if (d.dir != d.direct_alloc_dir)   \
    free (d.dir)⚠;
1870: for (..., dp = &d.dir[0]; ; ...,
↪  dp++) {
      ...
5188:   if (count < 0) {
          int saved_errno = errno;
          ...
          CLEANUP();  // may free d.dir
          errno = (saved_errno != 0 ?
          ↪  saved_errno
            : (💣dp->conversion == 'c'
               ? EILSEQ : EINVAL));
          return NULL;
5206:   }
5544: }
```

Listing 3. Use-after-free identified by AITHOS in `diffutils-3.3` (line numbers on left).

and the macro definition are lost amidst the large code context (> 4,600 SLoC) supplied to the LLM.

- **Safer triage.** AITHOS preserves substantially more TPs than LLM4SA on real-world datasets, reducing the risk of filtering real bugs.
- **Strong FP reduction.** AITHOS filters a large fraction of false positives, remaining competitive with LLM4SA while prioritizing TP safety.
- **Generalizability across diverse SAST tools.** AITHOS shows effective triage across all four tools we evaluate.

6.4 RQ2: What Is the Contribution of AITHOS's Guidance and Analysis Primitives to Triage Effectiveness?

Methodology Having established that AITHOS outperforms LLM4SA, we next isolate which design components drive this improvement. We perform three complementary analyses on our labeled dataset (§6.1).

First, we conduct an ablation study with two reduced variants. Comparing these variants with the full system isolates the incremental benefit of guidance and of primitive-based evidence gathering:
(A1) *SARIF-only*, which provides only SARIF fields (bug type, source location, message) and a 40-line local code snippet;
(A2) *Guidance with wide local context*, which adds rule-specific guidance and expands the snippet to 200 lines, but still disables analysis primitives.

Second, to understand the role of individual primitives, we log all primitive invocations during triage and compute the average number of calls per alert for each bug type. We use invocation frequency as a proxy for how useful each primitive is to the agent's reasoning process for different vulnerability classes.

Third, we assess whether AITHOS's specialized analysis primitives provide benefits beyond the repository-navigation tools used by general-purpose coding agents such as OpenAI Codex [54] and Claude Code [3]. In contrast to AITHOS, these agents typically rely on tools such as `Read`, `Glob`, `Grep`, and `LSP` [45] to inspect and navigate code. To approximate this setting, we replace AITHOS's analysis primitives with this basic toolset while keeping the same guidance prompts and underlying LLM. Both configurations are evaluated on the CODEQL real-world dataset.

Results Table 4 summarizes the results of our ablation study with Claude Sonnet 4 and GLM-4.7 on the full dataset. Adding rule guidance and a wide context (A2) over the SARIF-only baseline (A1) moderately improves FP filtering; for instance, Claude Sonnet 4's FP Filtered rate increases from 37.8% to 48.8%. Moreover, the introduction of analysis primitives in the full system provides the most dramatic performance gain. This addition elevates Claude's FP Filtered rate from 48.8% to 80.3%, while nearly eliminating uncertain FPs. We report additional ablation study results for other LLMs on a fixed subset of the dataset in Appendix A.1; these results exhibit the same trend. This demonstrates that

Table 4. Ablation study comparing three configurations in order of increasing complexity: Ablation (A1) *(SARIF-only)*, Ablation (A2) *(Guidance with wide local context)*, and the Full System (Guidance and analysis primitives). Within each *model* row, bold indicates the best-performing configuration for that metric (higher is better for TP Preserved, TP Pres. (Cons.), and FP Filtered; lower is better for FP Uncertain). Ties are bolded for all tied configurations. The total dataset size is TP=1,149; FP=2,044; Total=3,193.

	TP Preserved						**TP Pres. (Cons.)**					
	Ablation (A1)		**Ablation (A2)**		**Full System**		**Ablation (A1)**		**Ablation (A2)**		**Full System**	
Model	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**
Claude Sonnet 4	1,116	97.1	1,141	99.3	**1,146**	**99.7**	1,119	97.4	1,143	99.5	**1,146**	**99.7**
GLM-4.7	1,112	96.8	**1,134**	**98.7**	1,119	97.4	1,128	98.2	**1,144**	**99.6**	1,126	98.0
	FP Filtered						**FP Uncertain**					
	Ablation (A1)		**Ablation (A2)**		**Full System**		**Ablation (A1)**		**Ablation (A2)**		**Full System**	
Model	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**
Claude Sonnet 4	772	37.8	998	48.8	**1,642**	**80.3**	272	13.3	158	7.7	**6**	**0.3**
GLM-4.7	1,098	53.7	1,308	64.0	**1,563**	**76.5**	255	12.5	250	12.2	**67**	**3.3**

the analysis primitives are the key drivers of AITHOS's triage effectiveness, enabling models to resolve nuanced cases beyond the reach of simple analysis in limited contexts.

Table 5 shows the average invocation frequency for each analysis primitive. Structural-information primitives dominate overall usage. The agent invokes `dump_source_file` (P1) most often, as it uses this primitive to inspect source locations returned by other primitives. Identifier resolution through `ctags_readtags` (P5) is the next most frequent, while `find_var_decl` (P2), `find_callee` (P4), and `find_var_definitions` (P6) provide supporting declaration, call, and definition information. Path-analysis primitives exhibit more bug-type-specific usage. The agent invokes `get_path_constraint` (P7) most frequently for `uninitialized-local` alerts and uses `check_always_implying` (P8) almost exclusively for this bug type. This concentration reflects the need to reason about whether an execution path can reach a use before initialization and shows that the agent adapts its primitive selection to the vulnerability class.

Table 6 compares AITHOS's specialized analysis primitives against a generic coding-agent toolset. For Claude Sonnet 4, both configurations achieve nearly identical performance. For GLM-4.7 and GPT-5.4-mini (high), however, AITHOS consistently improves TP preservation: from 82.3% to 90.3% for GLM-4.7 and from 90.3% to 95.2% for GPT-5.4-mini. In both cases, the basic toolset is slightly more aggressive in filtering FPs, but at the cost of discarding more TPs. These results indicate that AITHOS's specialized primitives are particularly valuable for safer triage, especially for models that cannot reliably reconstruct vulnerability-relevant evidence using only generic coding-agent tools. Although this experiment shows comparable performance for Claude Sonnet 4, AITHOS remains useful because it provides a *model-agnostic* mechanism for effective FP filtering, rather than relying on the strengths of a particular coding-optimized model.

Table 5. Mean analysis-primitive invocations per alert by bug type for GLM-4.7 on the CodeQL real-world dataset (299 alerts). Primitive labels P1–P10 follow Table 2. Values are rounded to two decimal places, so 0.00 may represent a nonzero value below 0.005.

	SI						PA		BIC	
Bug type	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**
`unbounded-write`	5.51	0.36	0.07	0.32	3.51	0.34	0.08	0.00	0.01	0.01
`uninitialized-local`	2.69	0.19	0.03	0.15	0.44	0.60	2.14	0.42	0.00	0.00
`uncontrolled-alloc-size`	11.90	0.70	0.60	0.43	2.37	0.37	0.23	0.00	0.00	0.07
`missing-check-scanf`	2.35	0.23	0.00	0.00	0.12	0.15	0.19	0.00	0.00	0.00
`offset-use-before-check`	2.20	0.30	0.00	0.10	0.10	0.40	1.00	0.00	0.00	0.10
`unsafe-strcat`	2.88	0.25	0.13	0.13	0.25	0.75	0.25	0.00	0.00	0.00
`overrunning-write`	0.86	0.29	0.14	0.00	0.00	0.00	0.57	0.00	0.00	0.00
`invalid-pointer-deref`	14.17	0.50	0.17	1.00	2.33	0.00	0.50	0.00	0.00	0.00
`overrun-write`	11.00	0.00	1.00	0.00	3.00	0.00	0.00	0.00	0.00	0.00
Overall	**4.78**	**0.32**	**0.11**	**0.23**	**1.72**	**0.41**	**0.86**	**0.14**	**0.00**	**0.01**

Table 6. Comparison between Aithos Tools (specialized analysis primitives) and Basic Tools (`Read`, `Glob`, `Grep`, `LSP`) on the CodeQL real-world dataset. Within each *model* row, bold indicates the better-performing configuration for that metric (higher is better for TP Preserved, TP Pres. (Cons.), and FP Filtered; lower is better for FP Uncertain). Ties are bolded. Dataset size: TP=62; FP=237; Total=299.

	TP Preserved		TP Pres. (Cons.)		FP Filtered		FP Uncertain	
Model	**Basic**	**Aithos**	**Basic**	**Aithos**	**Basic**	**Aithos**	**Basic**	**Aithos**
Claude Sonnet 4	**60 (96.8%)**	**60 (96.8%)**	**60 (96.8%)**	**60 (96.8%)**	**207 (87.3%)**	**207 (87.3%)**	**0 (0.0%)**	3 (1.3%)
GLM-4.7	51 (82.3%)	**56 (90.3%)**	55 (88.7%)	**57 (91.9%)**	**204 (86.1%)**	199 (84.0%)	**10 (4.2%)**	14 (5.9%)
GPT-5.4-mini	56 (90.3%)	**59 (95.2%)**	56 (90.3%)	**59 (95.2%)**	**228 (96.2%)**	225 (94.9%)	**0 (0.0%)**	**0 (0.0%)**

All components contribute to the effectiveness of Aithos. Analysis primitives are the most critical, delivering a dramatic reduction in FPs. They also provide safer triage than a generic coding-agent toolset.

6.5 RQ3: Impact of Interactive Feedback

Methodology We next evaluate whether lightweight developer feedback further improves triage. We study repositories where baseline triage leaves at least five FPs for a single CodeQL rule, yielding the Juliet test suite and four real-world projects (`openssh`, `readline`, `wget`, and `findutils`). In each round, we add one corrected example for a misclassified FP and rerun triage using GPT-4o. We omit the Llm4sa dataset due to its large scale: the time and cost of this human-in-the-loop, multi-round study would be prohibitively high.

Results Feedback substantially improves FP filtering, often with the largest gain after the first example: for Juliet's `unsafe-strcat` alerts, FP filtering improves from 25.0% to 76.6%, and for `wget`'s `overrun-write` alerts, from 0% to 80.0% after one round. For the four real-world projects, the evaluated subsets

contain no ground-truth TPs, so TP-preservation metrics are not applicable there. On Juliet, where both TPs and FPs are present, conservative TP preservation remains 100% across rounds. Further rounds yield smaller and occasionally unstable gains, suggesting diminishing returns. Appendix A.2 reports the full per-round breakdown.

A small amount of developer feedback yields immediate, substantial gains, while additional feedback offers diminishing returns.

6.6 RQ4: Real-World Bug Finding

Methodology Finally, we evaluate whether the triage gains observed above translate into lower manual review effort during real-world bug finding. For this bug-finding study, we used a separate set of Debian packages, distinct from the labeled dataset in §6.1: we randomly selected 28 packages from the top 50 most-installed set and analyzed them with CODEQL, then applied AITHOS using Claude Sonnet 4. We first ran the standard pipeline. If more than five FPs of the same rule type remained in a given package, we enabled the interactive triage step to further filter out FPs. For all remaining alerts, we performed manual validation and followed responsible disclosure: submitting patches or filing issues with maintainers when appropriate.

```c
/* xauth/process.c */
static const char *xauth_filename = NULL;
int auth_finalize(void) {
  char temp_name[1025];
  temp_name[0] = '\0';
  if (write_auth_file (temp_name) == -1) {
    /* ... */
  }
}
static int write_auth_file(char *tmp_nam)
{
  strcpy (tmp_nam, xauth_filename);
  strcat (tmp_nam, "-n");
}
int auth_initialize(const char
↪  *authfilename) {
  xauth_filename = authfilename;
}

/* xauth/xauth.c */
int main(int argc, char **argv) {
  auth_initialize (argv[1]);
  /* ... */
  auth_finalize ();
}
```

Listing 4. Buffer overflow in `xauth`: unbounded copy from user input into a fixed-size stack buffer.

Results AITHOS drastically reduced the manual review effort. Of the initial 690 alerts, AITHOS automatically filtered out 530, leaving only 160 for manual inspection – a **76.8%** reduction in developers' triage workload. Manual validation of the 160 remaining alerts found 102 true bugs (63.8% post-filter precision). Of these 102, 12 were already fixed upstream at the time of our review; for the remaining cases, we prioritized higher-impact alerts and reported 48 bugs to maintainers via patches or issues, 12 of which have been confirmed so far. We are in the process of reporting the rest of the bugs. Listing 4 shows a buffer overflow we found in `xauth`.

AITHOS saves developers substantial review time on SAST alerts, allowing them to focus on a smaller set of high-value candidates.

7 Related Work

We discussed the closest prior work in §1. Here we summarize additional research.

A common strategy for reducing FPs is to run auxiliary analyses. For example, SYS [10] uses limited symbolic execution to filter warnings, while FUZZSLICE [50] applies directed fuzzing to validate alerts. These approaches can be effective, but they require custom design of auxiliary analyses, specialized infrastructure, or the ability to execute the target program, which is not always practical for low-level software such as bootloaders [60]. Conversely, AITHOS combines natural-language guidance with reusable analysis primitives, making FP filtering easier to adapt to new projects and StAs without extensive customization or expertise in symbolic or dynamic analysis.

7.1 LLMs for Program Analysis

Direct use of LLMs for program analysis remains challenging due to non-determinism [83,21], hallucinations [77,59], and limited context windows. Prompt-engineering techniques [28,5] attempt to narrow the search space of LLMs, for example by directing the model to inspect only variable definitions when analyzing uninitialized-variable warnings. However, such prompts are often brittle and do not generalize well across models. Other approaches mitigate context limits through compact program representations, such as graphs [38] or DSLs. While these representations improve scalability, they still provide largely fixed, program-wide context that may be unnecessary for a given analysis task and can still induce hallucinations.

Several works instead integrate LLMs into classic program analyses. For example, LLift [26] builds on UBITect [79] for use-before-initialization detection in Linux, and IRIS [29] uses LLMs to synthesize CODEQL taint specifications for external functions. Chapman *et al.* [11] interleave static analysis with LLM queries to infer error specifications. These approaches are effective, but are tightly coupled to particular analyses. Other systems, such as LLMDFA [74], go further and use LLMs as the primary engine for dataflow analysis and bug detection. In contrast, AITHOS targets post-analysis triage, acting as a general-purpose filter for SAST alerts. By combining lightweight prompts with reusable analysis primitives, AITHOS can adapt to different SAST tools with minimal effort.

8 Discussion and Limitations

Deterministic Rules vs. LLMs. One might ask whether AITHOS could replace the LLM with a deterministic engine that applies the triaging guidance using the

analysis primitives. In principle, this is possible. In practice, however, it would require translating natural-language guidance into rigid machine-readable rules and coupling them tightly to tool-specific alert formats, increasing engineering effort and reducing flexibility.

Cost. AITHOS has low inference cost. On the CodeQL real-world dataset, when powered by GPT-5.4-mini, it costs $0.016 per alert on average, with an average triage time of 47.6 s and a median of 24.5 s, while maintaining strong accuracy.

Generality. Our evaluation focuses on memory-safety alerts across four SAST tools. Results may not transfer unchanged to other bug classes, programming languages, or analyzers. Extending AITHOS to such settings requires alert-specific triaging guidance and/or additional analysis primitives. In our setting, however, this extension effort is modest, since the guidance is a short natural-language instruction and the Plugin Interface makes primitives easy to add.

Limits of Primitives. AITHOS's analysis primitives provide evidence of varying strength. Some primitives return compiler-backed structural facts, such as declaration locations, definition sites, call relations, and intra-procedural control dependencies. Others, especially symbolic checks, can be incomplete when reasoning requires recovering correlations between conditions and program state. Such approximations are often effective in practice. SPIDER [40] shows the value of lightweight symbolic reasoning for safe-patch identification, while recent LLM-assisted analyses [74,18] show that imperfect analyses can still be useful for path pruning and bug detection.

We mitigate this risk in three ways. First, we expose each primitive's scope (*e.g.,* intra-procedural for `get_path_constraint`, and limited implication checks for `check_always_implying`) to the agent so it does not over-generalize from partial evidence. Second, when broader context is needed, the agent can expand its evidence by traversing callers and callees (*e.g.,* via `find_callers`) and re-invoking primitives. Third, when the evidence is insufficient to justify suppression, the prompt instructs the agent to retain the alert or label it UC. This conservative design is consistent with our results: AITHOS retains 97.5% of TPs on the real-world datasets (§6.3).

9 Conclusion

We introduced AITHOS, a framework that equips LLMs with program analysis primitives to dynamically gather context and filter false positives in SAST. Aggregated over the real-world datasets, AITHOS reduced false positives by 78.0% while retaining 97.5% of true positives and uncovering 90 new bugs, showing the effectiveness of tool-augmented agents in easing the triage burden.

Data Availability

AITHOS, our dataset of SAST alerts with ground-truth labels, and per-alert LLM outputs are available at https://github.com/purs3lab/RAID-2026-AI-Assisted-FP-Filtering-Artifact.

Acknowledgments. This research was supported by the National Science Foundation (NSF) under Grants CNS-2340548 and TIP-2534021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the NSF, or the U.S. Government.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

A Appendix

A.1 Additional Ablation Study Results (RQ2)

Table 7 shows ablation results for Gemini 2.5 Pro and GPT-4o on CodeQL.

Table 7. Ablation study comparing three configurations in order of increasing complexity: Ablation (A1) *(SARIF-only)*, Ablation (A2) *(Guidance with wide local context)*, and the Full System (Guidance and analysis primitives). Within each *model* row, bold indicates the best-performing configuration for that metric (higher is better for TP Preserved, TP Pres. (Cons.), and FP Filtered; lower is better for FP Uncertain). Ties are bolded for all tied configurations. The total dataset size is TP=1,090; FP=989; Total=2,079.

	TP Preserved						**TP Pres. (Cons.)**					
	Ablation (A1)		**Ablation (A2)**		**Full System**		**Ablation (A1)**		**Ablation (A2)**		**Full System**	
Model	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**
GPT-4o	998	91.6	1,001	91.8	**1,007**	**92.4**	1,087	99.7	1,084	99.4	**1,088**	**99.8**
Gemini 2.5 Pro	1,081	99.2	**1,083**	**99.4**	1,033	94.8	1,082	99.3	1,084	99.4	**1,087**	**99.7**
	FP Filtered						**FP Uncertain**					
	Ablation (A1)		**Ablation (A2)**		**Full System**		**Ablation (A1)**		**Ablation (A2)**		**Full System**	
Model	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**
GPT-4o	202	20.4	260	26.3	**381**	**38.5**	231	23.4	**165**	**16.7**	194	19.6
Gemini 2.5 Pro	408	41.3	568	57.4	**743**	**75.1**	10	1.0	**9**	**0.9**	94	9.5

A.2 Impact of Interactive Feedback (RQ3)

Table 8 shows the impact of interactive feedback on FP filtering rates across multiple rounds.

A.3 RQ5: How Well Does Aithos Generalize across Different LLMs?

Methodology To evaluate whether Aithos is LLM-backend-agnostic, we run it with five LLMs: GPT (GPT-4o, GPT-5), Claude Sonnet 4, Gemini 2.5 Pro,

Table 8. Effectiveness of interactive feedback across multiple rounds (R0–R3), using GPT-4o. The experiment includes repositories with ≥ 5 FPs of a given type after the initial triage (R0). Each subsequent round adds one new few-shot example based on feedback for a single misclassified alert. Boldface highlights the best-performing round for each repository and bug-type combination.

Juliet (Conservative TP Pres. remains 100% across all rounds and bug types.)

Bug Type	GT		TP Preserved (N, %)		FP Filtered (N, %)		FP Uncertain (N, %)	
	TP	FP	R0	R1	R0	R1	R0	R1
Overall	752	752	**685 (91.1)**	668 (88.8)	321 (42.7)	**519 (69.0)**	**89 (11.8)**	105 (14.0)
`uninit-local`	160	16	**157 (98.1)**	**157 (98.1)**	2 (12.5)	**11 (68.8)**	4 (25.0)	**1 (6.2)**
`bad-strncpy`	400	484	**346 (86.5)**	343 (85.8)	256 (52.9)	**315 (65.1)**	**60 (12.4)**	72 (14.9)
`unsafe-strcat`	192	252	**182 (94.8)**	168 (87.5)	63 (25.0)	**193 (76.6)**	**25 (9.9)**	32 (12.7)

Real-world repos (TP=0 in ground truth; show only FP outcomes)

Repo	Bug Type	GT	FP Filtered (N, %)				FP Uncertain (N, %)			
		FP	R0	R1	R2	R3	R0	R1	R2	R3
openssh	`uninit-local`	11	4 (36.4)	7 (63.6)	**8 (72.7)**	5 (45.5)	**0 (0.0)**	**0 (0.0)**	1 (9.1)	**0 (0.0)**
wget	`overrun-write`	5	0 (0.0)	4 (80.0)	**5 (100.0)**	–	1 (20.0)	**0 (0.0)**	**0 (0.0)**	–
readline	`unbounded-write`	62	26 (41.9)	**32 (51.6)**	31 (50.0)	–	24 (38.7)	**23 (37.1)**	26 (41.9)	–
readline	`uninit-local`	18	4 (22.2)	**7 (38.9)**	5 (27.8)	–	4 (22.2)	**3 (16.7)**	4 (22.2)	–
findutils	`uninit-local`	11	1 (9.1)	**6 (54.5)**	**6 (54.5)**	–	4 (36.4)	**1 (9.1)**	2 (18.2)	–

and the open-weight GLM-4.7. We chose models that support the tool-calling capabilities AITHOS requires, cost at most \$3/M input and \$15/M output tokens (keeping large-scale runs affordable), and span both commercial and open ecosystems.

We run AITHOS on alerts in our labeled datasets and evaluate its triage decisions against ground truth using the metrics in §6.2. For comparability, we use the same guidance prompts and analysis primitives for every LLM, with no per-model prompt tuning, and we use each model's default temperature. Due to budget constraints, some models are evaluated on subsets of datasets: GPT-5 is only run on CODEQL alerts from real-world repositories, and the LLM4SA dataset is evaluated only with Claude Sonnet 4 and GLM-4.7.

Results Table 9 demonstrates the performance of AITHOS on our dataset across different LLMs. Without per-model tuning, AITHOS substantially reduces FPs while preserving TPs on both the real-world and synthetic datasets.

On CODEQL alerts for real-world applications, the median TP preservation rate is 95.2% and the median FP filtering rate is 84.0%. Stronger models push this to **95.2–98.4% TP preservation** and **79.3–94.9% FP filtering**, with low FP uncertainty (1.3–4.2%). On Juliet, AITHOS keeps a median **97.1%** of TPs while filtering a median **73.4%** of FPs. Top models achieve **94.7–100.0%** TP preservation and **72.9–84.3%** FP filtering. All listed models achieve **99.7–100.0%** *conservative* TP preservation, indicating that nearly all true alerts are either preserved or deferred as uncertain.

Aggregated across all datasets, AITHOS performs strongly with both Claude Sonnet 4 and GLM-4.7, preserving nearly all TPs while filtering the majority of FPs with low uncertainty. Overall, the framework delivers strong triage quality

Table 9. Performance of AITHOS across LLMs and datasets. Within each dataset block, **bold** marks the best model per metric (higher is better except FP Uncertain); ties are bolded.

	TP Preserved		TP Pres. (Cons.)		FP Filtered		FP Uncertain	
Model	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**
CODEQL **on real-world applications (TP=62; FP=237; Total=299)**								
GPT-4o	46	74.2	60	96.8	60	25.3	105	44.3
GPT-5	**61**	**98.4**	**61**	**98.4**	**225**	**94.9**	**3**	**1.3**
Claude Sonnet 4	60	96.8	60	96.8	207	87.3	**3**	**1.3**
Gemini 2.5 Pro	59	95.2	59	95.2	188	79.3	10	4.2
GLM-4.7	56	90.3	57	91.9	199	84.0	14	5.9
CODEQL **on Juliet (TP=1,028; FP=752; Total=1,780)**								
GPT-4o	961	93.5	**1,028**	**100.0**	321	42.7	89	11.8
Claude Sonnet 4	**1,028**	**100.0**	**1,028**	**100.0**	**634**	**84.3**	**1**	**0.1**
Gemini 2.5 Pro	974	94.7	**1,028**	**100.0**	555	73.8	84	11.2
GLM-4.7	1,022	99.4	1,025	99.7	548	72.9	11	1.5
LLM4SA **dataset (TP=59; FP=1,055; Total=1,114)**								
Claude Sonnet 4	**58**	**98.3**	**58**	**98.3**	801	75.9	**2**	**0.2**
GLM-4.7	41	69.5	44	74.6	**816**	**77.3**	42	4.0
All Datasets (TP=1,149; FP=2,044; Total=3,193)								
Claude Sonnet 4	**1,146**	**99.7**	**1,146**	**99.7**	**1,642**	**80.3**	**6**	**0.3**
GLM-4.7	1,119	97.4	1,126	98.0	1,563	76.5	67	3.3

Table 10. Performance of AITHOS (powered by Claude Sonnet 4) on Semgrep alerts across bug types in real-world repositories.

	Ground Truth			TP Preserved		TP Pres. (Cons.)		FP Filtered		FP Uncertain	
Bug Type	**TP**	**FP**	**Total**	**Count**	**%**	**Count**	**%**	**Count**	**%**	**Count**	**%**
Semgrep (Overall)	54	49	103	52	96.3	52	96.3	48	98.0	0	0.0
`path-manipulation`	46	0	46	44	95.7	44	95.7	0	–	0	–
`double-free`	0	21	21	0	–	0	–	21	100.0	0	0.0
`use-after-free`	1	28	29	1	100.0	1	100.0	27	96.4	0	0.0
`command-injection-path`	7	0	7	7	100.0	7	100.0	0	–	0	–

across a range of LLM backends; performance scales with model capability but remains effective even with weaker models.

A.4 RQ6: How Easily Can AITHOS Be Extended to New SAST Tools?

Methodology AITHOS uses natural-language guidance and reusable analysis primitives, making its core design independent of any particular SAST tool. To measure the effort required to support a new tool, we adapted AITHOS to *Semgrep* [62], a widely used syntactic and dataflow analyzer. We ran Semgrep on our real-world corpus and selected the six projects that triggered the most distinct Semgrep rules. We converted the alerts to SARIF, established ground truth as described in §6.1, and evaluated with Claude Sonnet 4.

Results Supporting Semgrep required only two new guidance prompts and two new analysis primitives; no other component changed.

`path-manipulation` marks an alert as TP when untrusted input controls an unconfined file path; `command-injection-path` does so when attacker-controlled input reaches an OS command without a strict allowlist. The new primitives are `taint_check` for LLVM-based source-to-sink analysis and `mutually_exclusive`, which uses Z3 to check whether the conjunction of two path conditions is satisfiable. Mutually exclusive free-and-use paths indicate an FP `use-after-free` alert, while mutually exclusive free paths indicate an FP `double-free` alert. The `use-after-free` and `double-free` rules reuse the guidance in Table 1.

Overall, AITHOS filters 98.0% of FPs and preserves 96.3% of TPs, without uncertain classifications (Table 10). Together with the strong results for `double-free` and `use-after-free`, which reuse existing guidance, this shows that AITHOS supports a new SAST tool with limited changes and high accuracy.

References

1. Ami, A.S., Moran, K., Poshyvanyk, D., Nadkarni, A.: "False negative - that one is going to kill you": Understanding industry perspectives of static analysis based security testing. In: IEEE Symposium on Security and Privacy, SP 2024. pp. 3979–3997. IEEE (2024). https://doi.org/10.1109/SP54263.2024.00019
2. Anthropic PBC: Introducing Claude 4. https://www.anthropic.com/news/claude-4 (2025), accessed: January 21, 2026
3. Anthropic PBC: Claude Code by Anthropic | AI Coding Agent, Terminal, IDE. https://claude.com/product/claude-code (2026), accessed: April 9, 2026
4. Avgustinov, P., de Moor, O., Jones, M.P., Schäfer, M.: QL: object-oriented queries on relational data. In: Krishnamurthi, S., Lerner, B.S. (eds.) 30th European Conference on Object-Oriented Programming, ECOOP 2016. LIPIcs, vol. 56, pp. 2:1–2:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPICS.ECOOP.2016.2
5. Bae, J., Kwon, S., Myeong, S.: Enhancing software code vulnerability detection using GPT-4o and Claude-3.5 Sonnet: A study on prompt engineering techniques. Electronics **13**(13), 2657 (2024). https://doi.org/10.3390/electronics13132657
6. Barbuti, R., Martelli, A.: A structured approach to static semantics correctness. Sci. Comput. Program. **3**(3), 279–311 (1983). https://doi.org/10.1016/0167-6423(83)90022-9
7. Bennett, G., Hall, T., Counsell, S., Winter, E., Shippey, T.: Do developers use static application security testing (SAST) tools straight out of the box? A large-scale empirical study. In: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2024. pp. 454–460. ACM (2024). https://doi.org/10.1145/3674805.3690750
8. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Gros, C., Kamsky, A., McPeak, S., Engler, D.R.: A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM **53**(2), 66–75 (2010). https://doi.org/10.1145/1646353.1646374
9. Black, P.: Juliet 1.3 test suite: Changes from 1.2. Technical Note (NIST TN) 1995, National Institute of Standards and Technology, Gaithersburg, MD (2018). https://doi.org/10.6028/NIST.TN.1995
10. Brown, F., Stefan, D., Engler, D.R.: Sys: A static/symbolic tool for finding good bugs in good (browser) code. In: 29th USENIX Security Symposium, USENIX

Security 2020. pp. 199–216. USENIX Association (2020), https://www.usenix.org/conference/usenixsecurity20/presentation/brown
11. Chapman, P.J., Rubio-González, C., Thakur, A.V.: Interleaving static analysis and LLM prompting with applications to error specification inference. Int. J. Softw. Tools Technol. Transf. **27**(2), 239–254 (2025). https://doi.org/10.1007/S10009-025-00780-7
12. Chess, B., McGraw, G.: Static analysis for security. IEEE Secur. Priv. **2**(6), 76–79 (2004). https://doi.org/10.1109/MSP.2004.111
13. Facebook, Inc.: Infer Static Analyzer. https://fbinfer.com/ (2025), accessed: January 21, 2026
14. Flynn, L., Klieber, W.: Using LLMs to adjudicate static-analysis alerts. In: Proceedings of the 58th Hawaii International Conference on System Sciences, HICSS 2025. Hawaii International Conference on System Sciences (2025). https://doi.org/10.24251/hicss.2025.903
15. Gentsch, C.: Evaluation of open source static analysis security testing (SAST) tools for C. Tech. rep., DLR DW (2020), https://elib.dlr.de/133945/
16. Gosain, A., Sharma, G.: Static Analysis: A Survey of Techniques and Tools, pp. 581–591. Springer India (2015). https://doi.org/10.1007/978-81-322-2268-2_59
17. Goseva-Popstojanova, K., Perhinschi, A.: On the capability of static code analysis to detect security vulnerabilities. Inf. Softw. Technol. **68**, 18–33 (2015). https://doi.org/10.1016/J.INFSOF.2015.08.002
18. Guo, J., Wang, C., Xu, X., Su, Z., Zhang, X.: RepoAudit: An autonomous LLM-agent for repository-level code auditing. In: Singh, A., Fazel, M., Hsu, D., Lacoste-Julien, S., Berkenkamp, F., Maharaj, T., Wagstaff, K., Zhu, J. (eds.) Forty-second International Conference on Machine Learning, ICML 2025. Proceedings of Machine Learning Research, vol. 267. PMLR / OpenReview.net (2025), https://proceedings.mlr.press/v267/guo25n.html
19. Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., Chen, Q., Peng, W., Feng, X., Qin, B., Liu, T.: A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. ACM Trans. Inf. Syst. **43**(2), 42:1–42:55 (2025). https://doi.org/10.1145/3703155
20. Huynh, L., Zhang, Y., Jayasundera, D., Jeon, W., Kim, H., Bi, T., Hong, J.B.: Detecting code vulnerabilities using LLMs. In: 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2025. pp. 401–414. IEEE (2025). https://doi.org/10.1109/DSN64029.2025.00047
21. Jahic, J., Sami, A.: State of practice: LLMs in software engineering and software architecture. In: 21st IEEE International Conference on Software Architecture, ICSA 2024 - Companion. pp. 311–318. IEEE (2024). https://doi.org/10.1109/ICSA-C63560.2024.00059
22. Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: 35th International Conference on Software Engineering, ICSE 2013. pp. 672–681. IEEE (2013). https://doi.org/10.1109/icse.2013.6606613
23. JSON-RPC Working Group: JSON-RPC. https://www.jsonrpc.org/ (2010), accessed: September 10, 2025
24. Kremenek, T., Engler, D.R.: Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In: Cousot, R. (ed.) Static Analysis, 10th International Symposium, SAS 2003. Lecture Notes in Computer Science, vol. 2694, pp. 295–315. Springer (2003). https://doi.org/10.1007/3-540-44898-5_16
25. LangChain, Inc.: LangChain. https://www.langchain.com (2025), accessed: September 8, 2025

26. Li, H., Hao, Y., Zhai, Y., Qian, Z.: Enhancing static analysis for practical bug detection: An LLM-integrated approach. Proc. ACM Program. Lang. **8**(OOPSLA1), 474–499 (2024). https://doi.org/10.1145/3649828
27. Li, K., Chen, S., Fan, L., Feng, R., Liu, H., Liu, C., Liu, Y., Chen, Y.: Comparison and evaluation on static application security testing (SAST) tools for Java. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023. pp. 921–933. ACM (2023). https://doi.org/10.1145/3611643.3616262
28. Li, Y., Li, X., Wu, H., Zhang, Y., Cheng, X., Zhong, S., Xu, F.: Attention is all you need for LLM-based code vulnerability localization. CoRR **abs/2410.15288** (2024). https://doi.org/10.48550/ARXIV.2410.15288
29. Li, Z., Dutta, S., Naik, M.: IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In: The Thirteenth International Conference on Learning Representations, ICLR 2025. OpenReview.net (2025), https://openreview.net/forum?id=9LdJDU7E91
30. Li, Z., Machiry, A., Chen, B., Naik, M., Wang, K., Song, L.: ARBITRAR: user-guided API misuse detection. In: 42nd IEEE Symposium on Security and Privacy, SP 2021. pp. 1400–1415. IEEE (2021). https://doi.org/10.1109/SP40001.2021.00090
31. Lipp, S., Banescu, S., Pretschner, A.: An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022. pp. 544–555. ACM (2022). https://doi.org/10.1145/3533767.3534380
32. Liu, B., Meng, G., Zou, W., Gong, Q., Li, F., Lin, M., Sun, D., Huo, W., Zhang, C.: A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE 2020. pp. 1547–1559. ACM (2020). https://doi.org/10.1145/3377811.3380923
33. Liu, N.F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., Liang, P.: Lost in the middle: How language models use long contexts. Trans. Assoc. Comput. Linguistics **12**, 157–173 (2024). https://doi.org/10.1162/TACL_A_00638
34. Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., Neubig, G.: Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Comput. Surv. **55**(9), 195:1–195:35 (2023). https://doi.org/10.1145/3560815
35. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: a manifesto. Commun. ACM **58**(2), 44–46 (2015). https://doi.org/10.1145/2644805
36. LLVM Project: LLVM Bitcode File Format. https://llvm.org/docs/BitCodeFormat.html (2025), accessed: September 9, 2025
37. LLVM Project: Writing an LLVM Pass. https://llvm.org/docs/WritingAnLLVMNewPMPass.html (2025), accessed: September 10, 2025
38. Lu, G., Ju, X., Chen, X., Pei, W., Cai, Z.: GRACE: empowering LLM-based software vulnerability detection with graph structure and in-context learning. J. Syst. Softw. **212**, 112031 (2024). https://doi.org/10.1016/J.JSS.2024.112031
39. Machiry, A., Kastner, J.H., McCutchen, M., Eline, A., Headley, K., Hicks, M.: C to Checked C by 3C. Proc. ACM Program. Lang. **6**(OOPSLA1), 1–29 (2022). https://doi.org/10.1145/3527322
40. Machiry, A., Redini, N., Camellini, E., Kruegel, C., Vigna, G.: SPIDER: enabling fast patch propagation in related software repositories. In: 2020 IEEE Symposium

on Security and Privacy, SP 2020. pp. 1562–1579. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00038
41. Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., Vigna, G.: DR. CHECKER: A soundy analysis for Linux kernel drivers. In: 26th USENIX Security Symposium, USENIX Security 2017. pp. 1007–1024. USENIX Association (2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry
42. Mangal, R., Zhang, X., Nori, A.V., Naik, M.: A user-guided approach to program analysis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015. pp. 462–473. ACM (2015). https://doi.org/10.1145/2786805.2786851
43. Marjamäki, D.: Cppcheck: A tool for static C/C++ code analysis. https://cppcheck.sourceforge.io/ (2013), accessed: August 4, 2026
44. Microsoft: System message design. https://learn.microsoft.com/en-us/azure/ai-foundry/openai/concepts/advanced-prompt-engineering (2025), accessed: September 10, 2025
45. Microsoft: Official page for Language Server Protocol. https://microsoft.github.io/language-server-protocol/ (2026), accessed: April 9, 2026
46. Microsoft Corporation: SARIF Home. https://sarifweb.azurewebsites.net/ (2025), accessed: September 9, 2025
47. Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., Gao, J.: Large language models: A survey. CoRR **abs/2402.06196** (2024). https://doi.org/10.48550/ARXIV.2402.06196
48. Mohajer, M.M., Aleithan, R., Harzevili, N.S., Wei, M., Belle, A.B., Pham, H.V., Wang, S.: Effectiveness of ChatGPT for static analysis: How far are we? In: Proceedings of the 1st ACM International Conference on AI-Powered Software, AIware 2024. ACM (2024). https://doi.org/10.1145/3664646.3664777
49. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
50. Murali, A., Mathews, N., Alfadel, M., Nagappan, M., Xu, M.: FuzzSlice: Pruning false positives in static analysis warnings through function-level fuzzing. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE 2024. pp. 1–13. ACM (2024). https://doi.org/10.1145/3597503.3623321
51. Ngo, K., Do, D., Nguyen, T., Vo, H.D.: Ranking warnings of static analysis tools using representation learning. In: 28th Asia-Pacific Software Engineering Conference, APSEC 2021. pp. 327–337. IEEE (2021). https://doi.org/10.1109/APSEC53868.2021.00040
52. OpenAI: API reference. https://platform.openai.com/docs/api-reference/chat (2025), accessed: September 10, 2025
53. OpenAI: Function calling - OpenAI API. https://platform.openai.com/docs/guides/function-calling (2025), accessed: September 11, 2025
54. OpenAI: Codex | AI Coding Agent. https://chatgpt.com/codex/ (2026), accessed: April 9, 2026
55. Park, J., Lee, H., Ryu, S.: A survey of parametric static analysis. ACM Comput. Surv. **54**(7), 149:1–149:37 (2022). https://doi.org/10.1145/3464457
56. Pistoia, M., Chandra, S., Fink, S.J., Yahav, E.: A survey of static analysis methods for identifying security vulnerabilities in software systems. IBM Syst. J. **46**(2), 265–288 (2007). https://doi.org/10.1147/SJ.462.0265

57. Pratikakis, P., Foster, J.S., Hicks, M.: Existential label flow inference via CFL reachability. In: Yi, K. (ed.) Static Analysis, 13th International Symposium, SAS 2006. Lecture Notes in Computer Science, vol. 4134, pp. 88–106. Springer (2006). https://doi.org/10.1007/11823230_7
58. Rajapakse, R.N., Zahedi, M., Babar, M.A.: An empirical analysis of practitioners' perspectives on security tool integration into DevOps. In: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2021. pp. 6:1–6:12. ACM (2021). https://doi.org/10.1145/3475716.3475776
59. Reddy, G.P., Pavan Kumar, Y.V., Prakash, K.P.: Hallucinations in large language models (LLMs). In: 2024 IEEE Open Conference of Electrical, Electronic and Information Sciences, eStream 2024. pp. 1–6. IEEE (2024). https://doi.org/10.1109/estream61684.2024.10542617
60. Redini, N., Machiry, A., Das, D., Fratantonio, Y., Bianchi, A., Gustafson, E., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: BootStomp: On the security of bootloaders in mobile devices. In: 26th USENIX Security Symposium, USENIX Security 2017. pp. 781–798. USENIX Association (2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/redini
61. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C.: Lessons from building static analysis tools at Google. Commun. ACM **61**(4), 58–66 (2018). https://doi.org/10.1145/3188720
62. Semgrep, Inc.: Semgrep App Security Platform | AI-assisted SAST, SCA and Secrets Detection. https://semgrep.dev (2025), accessed: September 10, 2025
63. Shen, M., Pillai, A.A., Yuan, B.A., Davis, J.C., Machiry, A.: Finding 709 defects in 258 projects: An experience report on applying CodeQL to open-source embedded software (experience paper). Proc. ACM Softw. Eng. **2**(ISSTA), 1077–1100 (2025). https://doi.org/10.1145/3728923
64. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: language agents with verbal reinforcement learning. In: Advances in Neural Information Processing Systems 36, NeurIPS 2023. pp. 8634–8652 (2023). https://doi.org/10.52202/075280-0377
65. Sotirov, A.I.: Automatic Vulnerability Detection Using Static Source Code Analysis. Master's thesis, The University of Alabama, Tuscaloosa, Alabama (2005), https://www.ida.liu.se/~TDDC90/literature/papers/sotirov.pdf
66. SPI and others: Debian – Packages. https://www.debian.org/distrib/packages.en.html (2023), accessed: September 10, 2025
67. The Clang Team: Clang Static Analyzer. https://clang-analyzer.llvm.org/ (2025), accessed: September 10, 2025
68. The Clang Team: Clang Tools. https://clang.llvm.org/docs/ClangTools.html (2025), accessed: September 10, 2025
69. Ullah, S., Han, M., Pujar, S., Pearce, H., Coskun, A.K., Stringhini, G.: LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In: IEEE Symposium on Security and Privacy, SP 2024. pp. 862–880. IEEE (2024). https://doi.org/10.1109/SP54263.2024.00210
70. Universal Ctags Project: Universal ctags. https://ctags.io/ (2021), accessed: September 11, 2025
71. Vachon, T.: Whole program LLVM. https://github.com/travitch/whole-program-llvm (2019), accessed: August 3, 2026
72. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Guyon, I., von Luxburg, U., Bengio,

S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017. pp. 5998–6008 (2017), https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html
73. Wadhams, Z.D., Izurieta, C., Reinhold, A.M.: Barriers to using static application security testing (SAST) tools: A literature review. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASEW 2024. pp. 161–166. ACM (2024). https://doi.org/10.1145/3691621.3694947
74. Wang, C., Zhang, W., Su, Z., Xu, X., Xie, X., Zhang, X.: LLMDFA: Analyzing dataflow in code with large language models. In: Advances in Neural Information Processing Systems 37, NeurIPS 2024. pp. 131545–131574 (2024). https://doi.org/10.52202/079017-4181
75. Wen, C., Cai, Y., Zhang, B., Su, J., Xu, Z., Liu, D., Qin, S., Ming, Z., Tian, C.: Automatically inspecting thousands of static bug warnings with large language model: How far are we? ACM Trans. Knowl. Discov. Data **18**(7), 168 (2024). https://doi.org/10.1145/3653718
76. Wögerer, W.: A survey of static program analysis techniques. Tech. rep., Technische Universität Wien (2005)
77. Xu, Z., Jain, S., Kankanhalli, M.S.: Hallucination is inevitable: An innate limitation of large language models. CoRR **abs/2401.11817** (2024). https://doi.org/10.48550/ARXIV.2401.11817
78. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K.R., Cao, Y.: ReAct: Synergizing reasoning and acting in language models. In: The Eleventh International Conference on Learning Representations, ICLR 2023. OpenReview.net (2023), https://openreview.net/forum?id=WE_vluYUL-X
79. Zhai, Y., Hao, Y., Zhang, H., Wang, D., Song, C., Qian, Z., Lesani, M., Krishnamurthy, S.V., Yu, P.L.: UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel. In: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020. pp. 221–232. ACM (2020). https://doi.org/10.1145/3368089.3409686
80. Zhang, S., Dinan, E., Urbanek, J., Szlam, A., Kiela, D., Weston, J.: Personalizing dialogue agents: I have a dog, do you have pets too? In: Gurevych, I., Miyao, Y. (eds.) Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Volume 1: Long Papers. pp. 2204–2213. Association for Computational Linguistics (2018). https://doi.org/10.18653/V1/P18-1205, https://aclanthology.org/P18-1205/
81. Zhang, X., Grigore, R., Si, X., Naik, M.: Effective interactive resolution of static analysis alarms. Proc. ACM Program. Lang. **1**(OOPSLA), 57:1–57:30 (2017). https://doi.org/10.1145/3133881
82. Zheng, Z., Ning, K., Zhong, Q., Chen, J., Chen, W., Guo, L., Wang, W., Wang, Y.: Towards an understanding of large language models in software engineering tasks. Empir. Softw. Eng. **30**(2), 50 (2025). https://doi.org/10.1007/S10664-024-10602-0
83. Zhou, X., Zhang, T., Lo, D.: Large language model for vulnerability detection: Emerging results and future directions. In: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER 2024. pp. 47–51. ACM (2024). https://doi.org/10.1145/3639476.3639762