CODE-AUGUR: Agentic Vulnerability Detection via Specification Inference

Zhengxiong Luo†, Mehtab Zafar†, Dylan Wolff, Abhik Roychoudhury
National University of Singapore
{luozx, mehtab, abhik}@nus.edu.sg, wolffdy0@gmail.com

***Abstract*—The advent of agentic vulnerability detection is already becoming a watershed moment for software security. Audits conducted entirely by autonomous LLM agents are uncovering critical vulnerabilities in fundamental software that forms the basis of digital society. Many of these vulnerabilities have remained masked for years and are being uncovered only now with the help of AI agents. Yet the *reasoning* behind these discoveries remains alarmingly opaque and unvalidated. What assumptions did the agent make about a function's inputs when it deemed that function to be secure? Failures in reasoning and incorrect assumptions can lead to missed vulnerabilities and reduce trust in agentic analysis.**

In this work, we propose a novel security-specification-first paradigm that (1) exposes the agent's tacit assumptions explicitly as security specifications and (2) continuously refines those specifications via runtime falsification. We realize our approach in CODE-AUGUR, a novel harness for *agentic vulnerability detection*. Given a codebase, CODE-AUGUR analyzes each component of the system for vulnerable code. When it deems a component to be secure, it commits the local invariants behind that judgment as in-source assertions. In parallel, CODE-AUGUR leverages a guided fuzzer to attempt to falsify those assumptions. When the fuzzer triggers an assertion, this either reveals a genuine vulnerability or a flawed specification to refine. In both cases, this process grounds the agent's understanding, aligning its view of code intent with how the code actually behaves. On real-world subjects, we find that CODE-AUGUR effectively leverages security specifications to detect more vulnerabilities than other state-of-the-art agents. Additionally, CODE-AUGUR found 22 *new* vulnerabilities in key open-source projects, 16 of which have already been fixed or confirmed by developers. Compared to curated specialized models like Claude Mythos, our approach presents an effective agentic vulnerability detection approach that can be built on top of widely available LLMs like Sonnet and DeepSeek.

1. Introduction

While experts have long decried the inadequacy of security measures in the software industry, recent events have made these warnings impossible to ignore. Agentic vulnerability detection systems, such as Claude Mythos [1], are finding more vulnerabilities than ever before [2]. Meanwhile, many high-impact vulnerabilities being discovered, such as the Copy-Fail bug in the Linux kernel, have lain dormant for years in high-profile, open-source software.[1] As such, it is critical that we understand and harness autonomous security agents to reduce the dangerous security debt accrued in our collective software infrastructure.

Security audits build far more knowledge than the bug reports they deliver. To find a vulnerability, an analyst must understand the assumptions of the codebase, typically implicit and scattered across disparate contexts, and spot where they might be violated. Yet the CVE reports that result from this analysis capture only *one instance of a vulnerability*, not this hard-won understanding and analysis. This loss is consequential: as the discovery of the Copy-Fail bug, which quickly led to the similar Dirty-Frag vulnerability[2], shows, patterns of invalidated assumptions often lead to repeated vulnerabilities. By focusing only on bug reports, analysts (autonomous agents or otherwise) fail to generalize this knowledge to similar code elsewhere or persist it for future audits, leading to regressions, incomplete fixes, and duplicated analysis.

More broadly, the knowledge built during an audit represents *security specifications* for that project. While incomplete relative to functional correctness specifications, security specifications in the form of local invariants—predicates that should hold at strategic program points—can encode the critical conditions for maintaining the system's global security. Unfortunately, in practice, most software comes *without* explicit security specifications, so analysts must *infer* them.

Thus, an ideal agentic analyst must also be able to infer security specifications effectively, not only to find today's vulnerabilities but also to lay the foundation for a secure codebase in the future. Yet how can we obtain security specifications *autonomously* from *existing* codebases? How do we gain *confidence* in these specifications? How do we effectively and automatically *leverage* specifications to aid in finding new bugs?

Large Language Models (LLMs), and in particular agentic systems, have proven effective at inferring functional correctness specifications [3], [4]. It thus stands to reason that

†*Both authors contributed equally.*

1. https://nvd.nist.gov/vuln/detail/CVE-2026-31431
2. https://nvd.nist.gov/vuln/detail/CVE-2026-43284

they can also infer *security specifications*: indeed, Claude Mythos and other agentic systems [1], [5] are already finding many new vulnerabilities in open-source software, implying some capability for security specification inference. However, existing agentic analysts infer these specifications only *implicitly*, leaving them both (1) **opaque**: the reasoning unfolds in a semantically ambiguous natural-language trajectory that is difficult to understand, let alone reuse in later audits; and (2) **fallible yet largely unvalidated**: an agent's reasoning, like a human expert's, is intuitive rather than systematic, following a few plausible paths and prone to missing corner cases. Worse still, agents possess limited capability to validate the specifications they infer: even when coupled with dynamic execution [6], an agent validates only the suspected *bugs* these specifications lead to, not the *specifications* themselves. These limitations make a clean audit hard to trust: when an agent reports no bugs, one cannot tell whether the code withstood scrutiny or the implicit specifications behind that verdict were simply incorrect.

Our Approach. We address these limitations with a security specification-first approach, in which the agent's tacit understanding of the program is made explicit so that it can be continually falsified at runtime and ultimately leveraged to find new bugs. Our agentic vulnerability detection system, CODE-AUGUR, combines the intuitive reasoning of an agent with the more comprehensive exploration of a fuzzing engine, using explicit security specifications to bridge the gap between these two paradigms and provide a comprehensible and reusable artifact from each audit beyond individual vulnerability reports.

Guided by a threat model indicating the system's security boundaries and objectives, CODE-AUGUR begins its audit by analyzing each component, either flagging a vulnerability directly or, when it deems the code secure, determining local invariants at program points in support of that judgment. CODE-AUGUR durably commits these invariants as executable security specifications to the project's source repository in the form of assertions. Each local invariant encodes assumptions the agent believes must hold for the system to remain secure and serves a dual role: a *reasoning anchor* at which the agent records its semantic understanding of the program, and a *directional landmark* that comprehensive exploration approaches such as fuzzing can steer toward.

These executable security specifications are directly amenable to validation, for which CODE-AUGUR leverages a complementary fuzzer to falsify them, a process less susceptible to omissions or other failures in LLM reasoning itself. In turn, the invariants make fuzzing more tractable: an invariant typically lies at a shallower semantic layer than the crash it forestalls, giving the fuzzer an accessible target rather than a deep symptom to stumble upon. When the fuzzer uncovers a violation of an invariant, this results in one of two outcomes: (i) a *security-relevant violation* indicates an actual defect in the program and surfaces a vulnerability, while (ii) a *benign violation* indicates that the agent's security specification needs to be refined, exposing a divergence between the agent's assumptions about the program and the program's actual behavior. The latter case drives the agent to reassess the corresponding code region and iteratively refine its security specifications before they are checked again by the fuzzer. This closes the loop between agentic semantic analysis and fuzz-driven edge-case exploration, harnessing each paradigm's strengths to cover the other's weaknesses.

We evaluate CODE-AUGUR on two state-of-the-art benchmarks drawn from DARPA's AI Cyber Challenge (AIxCC) [7] and the OSV database [8]. Across both benchmarks and under two different frontier LLMs, CODE-AUGUR finds 34%–370% more bugs than the state-of-the-art agentic bug detection systems CLAUDE CODE [6] and the AIxCC-winning ATLANTIS [9], with the inferred specifications playing a central role in these discoveries. Beyond the benchmarks, CODE-AUGUR discovered 22 new vulnerabilities in widely-used open-source projects, 16 of which have already been fixed or confirmed by developers, with several earning bug bounty rewards. Finally, we showcase that the inferred invariants are valuable, durable artifacts that outlast a single audit, e.g., pinpointing the incomplete fix and bug family behind a four-month series of fixes in the actively developed project `gpsd` [10].

In summary, we make the following contributions:

- We formulate a novel security-specification-first paradigm for agentic vulnerability detection, which makes the agent's tacit security judgments explicit, falsifiable, and continually aligned with the program's actual behavior.
- We realize this paradigm in CODE-AUGUR: the agent commits its assumptions as in-source invariants, and a guided fuzzer continuously falsifies them to expose flawed assumptions, forming a reason-falsify-refine loop.
- We evaluate CODE-AUGUR and demonstrate its substantial improvement in bug finding over state-of-the-art agentic systems, uncovering 22 previously unknown vulnerabilities in widely-used projects (16 fixed or confirmed, several bounties awarded).

2. Motivation

To motivate the design of CODE-AUGUR, we examine an example vulnerability from the recent AIxCC challenge [7] and analyze the limitations of existing work.

2.1. Motivating Example

Figure 1 walks through the challenge in `Little CMS`, a widely used color-management library. `Little CMS` converts *pixels* between `ColorSpaces` (e.g., RGB, GRAY). A conversion pairs (1) a `ColorSpace` with (2) a `PixelFormat` recording how the input bytes are packed (Figure 1a). Since `PixelFormat` and `ColorSpace` are two views of the same *pixels*, they must agree on the *channel* count: the count $\mathbb{C}_{\texttt{PixelFormat}}$ the `PixelFormat` declares must equal the count $\mathbb{C}_{\texttt{ColorSpace}}$ the `ColorSpace` implies, i.e., $\mathbb{C}_{\texttt{PixelFormat}} = \mathbb{C}_{\texttt{ColorSpace}}$. Otherwise, the code that later unpacks a *pixel* would read the wrong bytes.

At a high level, the `TestHarness` (Figure 1b) reads an input from a user-supplied color profile (lines 1-4) and passes

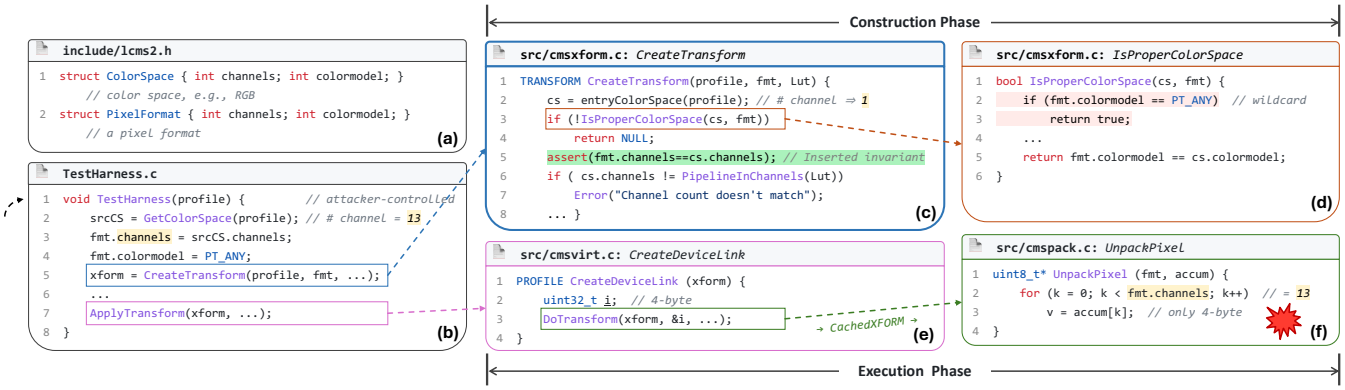

Figure 1. The simplified code snippets of the bug example in `Little CMS`. In the *construction phase* (above), an attacker-controlled profile builds a pixel format with 13 channels and the `PT_ANY` wildcard, while the library fixes the transform's entry color space to a single-channel space. `IsProperColorSpace` accepts the format through its `PT_ANY` branch without comparing channel counts, so the agent regards the construction as secure; CODE-AUGUR can commit this assumption as the local invariant ϕ, `fmt.channels` = `cs.channels`, which the fuzzer may later falsify when the transform is created, at which point no memory error is visible. In the *execution phase* (below), the same input carries the inconsistent channel count through a separate, later call into `UnpackPixel`, which can read past the 4-byte buffer and trigger an out-of-bounds read. Dashed arrows are call-graph edges; the inconsistent channel count (13) is highlighted along its path from source to sink.

it to the *construction phase* via `CreateTransform` (line 5), which checks the input and assembles a `Transform`; only once these checks pass does the *execution phase* execute such a `Transform` by calling `ApplyTransform` (line 7).

Figure 1c shows the construction phase: it validates the `PixelFormat fmt` against the `ColorSpace cs` using the guard `IsProperColorSpace` (line 3, Figure 1d). This function compares the `colormodel` of `fmt` and `cs`: a matching `colormodel` typically implies a matching *channel* count. After that, `CreateTransform` also compares $\mathbb{C}_{\texttt{ColorSpace}}$ against a different source—the conversion's internal pipeline, and rejects any mismatch with the error "`Channel count doesn't match`" (lines 6-7). The validation therefore looks sound, and the LLM agent, reading the same code, concludes likewise and regards it as secure.

However, this reasoning is *incorrect* due to a branch within the `IsProperColorSpace` guard: as shown in lines 2–3 in Figure 1d, for a special `colormodel PT_ANY`, the guard returns `true` directly, since `PT_ANY` is a wildcard that matches any `colormodel`. Therefore, for a `TestHarness` carrying exactly this `colormodel PT_ANY` (Figure 1b, line 4), the intended agreement $\mathbb{C}_{\texttt{PixelFormat}} = \mathbb{C}_{\texttt{ColorSpace}}$ is never enforced. In this scenario, a carefully crafted profile input could push the two counts apart: (i) its declared color space sets `fmt.channels` to 13 (Figure 1b, lines 2–3), so $\mathbb{C}_{\texttt{PixelFormat}} = 13$; and (ii) other fields of the input profile make `entryColorSpace` (Figure 1c, line 2) return a single-channel `ColorSpace`, so $\mathbb{C}_{\texttt{ColorSpace}} = 1$.

The divergence between $\mathbb{C}_{\texttt{PixelFormat}}$ and $\mathbb{C}_{\texttt{ColorSpace}}$ has no immediate ill effects: the mismatched counts are stored as ordinary integers, and nothing in the construction phase signals an error. Whether the divergence manifests later as a visible bug depends on how the stored `PixelFormat` is used. In the execution phase, `ApplyTransform` (Figure 1b) routes the inflated `PixelFormat` onto a path provisioned for unpacking a small pixel (Figure 1e), where the unpack loop (Figure 1f) iterates the declared $\mathbb{C}_{\texttt{PixelFormat}} = 13$ times over a 4-byte buffer (`uint32_t i`), causing an out-of-bounds read that AddressSanitizer [11] reports.

Identifying this vulnerability is extremely difficult: the causal chain from the bypassed validation check to the crash-site spans two phases of execution, multiple call layers, and separate source files.

2.2. Limitations of Existing Work

Whether an analysis tool can detect a bug like the one in Figure 1 comes down to how faithfully its understanding of the program matches the program's actual semantics. For a target program P, let I be the in-scope inputs an attacker can supply, and $\Pi_P(I)$ be the executions P admits on them. A real vulnerability is an execution that reaches an error state, collected in ERR; the ground-truth bugs are thus:

$$\Pi_P(I) \cap \text{ERR}$$

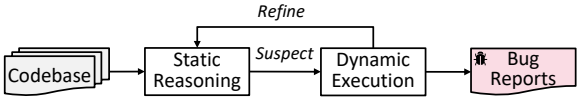

Figure 2. Existing paradigm of agentic bug detection: the agent reasons and flags suspects for dynamic validation. However, the underlying *implicit* security specification itself remains *largely unvalidated*.

Agentic Approach. An LLM-based analyst reasons not over the real behaviors $\Pi_P(I)$ but over its own *view* of program P, written $\hat{\Pi}_P(I)$: the executions its reasoning reconstructs, over the input space I. The ideal case is $\hat{\Pi}_P(I) = \Pi_P(I)$, but this is hard to achieve especially for complex programs with intricate dynamic behaviors.

The prevailing approach to closing this gap is to selectively ground the agent via simple tools such as code execution [6]. As Figure 2 shows, the agent first reasons statically

and flags each *suspect*, an input $i^* \in I$ it believes can reach an error state, then validates it by dynamic execution. This workflow leads to a dual view for the agent:

$$\hat{\Pi}_P(I) \xrightarrow{\substack{\textit{agentic} \\ \textit{capability}}} \begin{cases} \Pi_P(i^*) & \text{for } \textit{suspect } i^* \in I, \quad (1) \\ \hat{\Pi}_P(i) & \text{for } \textit{non-suspect } i \in I. \quad (2) \end{cases}$$

It reports a vulnerability whenever a suspect i^*'s run genuinely reaches an error state, i.e., $\Pi_P(i^*) \cap \text{ERR} \neq \emptyset$. Meanwhile, the agent's view $\hat{\Pi}_P$ is continuously refined by what concrete executions reveal during this process.

Although this agentic approach is powerful and has uncovered many vulnerabilities, it still faces major limitations. In case (2), the agent deems a site secure and never raises it as a suspect, so the dynamic stage is never invoked to test it. Put differently, the agent's view $\hat{\Pi}_P$ acts as an *implicit security specification*: dynamic execution validates only the suspected bugs flagged by this specification, i.e., case (1), not the specification itself, which harbors the seemingly secure, non-suspect verdicts, i.e., case (2). In our example (Figure 1), the agent follows the guard `IsProperColorSpace` along plausible paths and (incorrectly) infers that this guard *implies* a matching channel count, deeming `CreateTransform` secure. Fundamentally, the security of a program point is not a property of any single path, but must hold over *all* paths that reach it, under *diverse* inputs. An agent may reason strongly along one or a few individual paths, yet a sound security judgment must account for that entire combinatorial space, which is far harder.

Brute-Force Approach. A brute-force approach, such as fuzzing [12], instead randomly samples $\Pi_P(I)$ directly and reports only bugs it can trigger in $\Pi_P(I) \cap \text{ERR}$. By virtue of this randomized sampling process, fuzzing often surfaces counter-intuitive inputs that developers and security auditors overlook. However, lacking semantic reasoning, blind randomization struggles to reach deep program states. In our example (Figure 1), producing a proof-of-vulnerability means satisfying several narrow constraints simultaneously: building a semi-valid `profile` that passes validation, declaring more channels than the data carries, and carrying that `profile` through the execution phase into `UnpackPixel` to trigger the out-of-bounds read. This semantic blindness also extends to the channel-count mismatch when it occurs at construction (Figure 1c). A fuzzer does not know that this mismatch constitutes an interesting state, and thus even grey-box fuzzers [13] cannot bias sampling that state to alleviate the difficulty of satisfying all constraints necessary to manifest the bug.

3. Our approach

Our **goal** is to make the LLM agent's view of the program *more* faithful to reality, i.e., in the notation of §2.2:

$$\hat{\Pi}_P(I) \rightsquigarrow \Pi_P(I), \quad (3)$$

To fill this gap, the agent that pronounces a site secure must itself be audited: we must check whether its security judgments actually hold, i.e., case (2), the non-suspect region where false negatives hide. This task is challenging at two levels: *understanding* the reasoning behind the judgment and *validating* the understanding.

Challenge 1: *Understanding* an agent's judgment is difficult because the reasoning behind it is *implicit*, reached through a semantically ambiguous natural-language reasoning trajectory. Moreover, this reasoning is couched in the program's own intent (e.g., that a color space and a pixel format must agree on channel count, as shown in §2.1) and states (e.g., `fmt.channels` and `cs.channels`), so interpreting it correctly requires grasping these program-specific concepts, which any *separate* formalism (e.g., a bespoke specification language) would have to re-encode before the claim could even be stated, a step that is both prone to information loss and inefficient.

Challenge 2: *Validating* is often equally difficult, as it typically involves reaching the precise conditions under which an incorrect judgment will break in a real execution. Neither of the two natural validators suffices by itself: (a) An LLM-based validator shares the same blind spots that produced the judgment in the first place, and tends to overlook the same corner cases a second time. (b) A brute-force approach, such as a fuzzer, is unaware of program semantics, rarely reaching deep states and making it difficult to distinguish between semantically relevant and irrelevant states among the many reachable states.

Basic Idea. CODE-AUGUR pursues the goal (3) with a security-specification-first approach to these two challenges.

(i) For *understanding*, rather than recovering the agent's implicit, program-specific reasoning trajectory after the fact, CODE-AUGUR lets the agent write its understanding down *explicitly*: whenever the agent deems a site secure, it states the assumption behind that verdict as an *invariant* ϕ. ϕ is a falsifiable assertion placed at the site in the program's own source (e.g., Figure 1c, line 5). By encoding assumptions as local invariants, CODE-AUGUR ensures that the assumption becomes an obligation over *any* path that reaches the site, instead of a statement about one particular path (§2.2). Writing ϕ as an assertion in the program's source also spares CODE-AUGUR from re-encoding each project's concepts into a separate formalism. However, the most important advantage of an *executable specification* is for *validation*.

(ii) For *validation*, CODE-AUGUR grounds ϕ via executing the program under a large number of automatically generated test cases. Rather than using the agent itself to generate test cases, which would be susceptible to the same blind spots that might lead to invalid invariants in the first place, CODE-AUGUR delegates test generation to a grey-box fuzzer. CODE-AUGUR makes ϕ the fuzzer's target to *falsify*, searching for a witness input $i_{\neg\phi}$ whose execution trace satisfies $\neg\phi$. Because ϕ pins down only *what* must hold and not *how* to reach the site, the fuzzer is free to violate it by *any* path. Steering the fuzzer by ϕ rather than by blind coverage is more effective in two ways. First, the search aims squarely at falsifying a given property ϕ instead of widening coverage indiscriminately. Second, ϕ typically sits at a higher semantic layer than the eventual crash, so

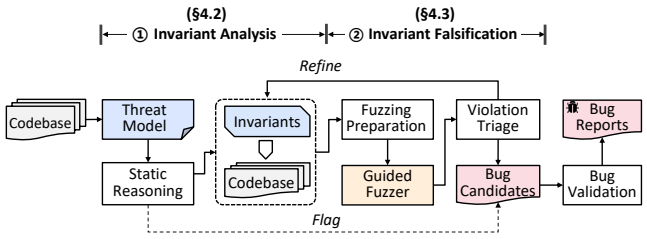

Figure 3. Overview of CODE-AUGUR, which turns the agent's *implicit* reasoning into *explicit*, *falsifiable* security specifications. ① Given a codebase, CODE-AUGUR distills a threat model capturing the project's high-level security intent and context. Guided by it, CODE-AUGUR inspects the code and either *flags* bug candidates directly or, deeming a site secure, *commits* the supporting invariants as in-source assertions. ② These beliefs the agent cannot guarantee become checks a `Guided Fuzzer` can run and falsify, and each violation, once triaged, yields either bug candidates or flawed assumptions to refine. Finally, CODE-AUGUR validates each candidate before reporting.

the fuzzer pursues a shallow, semantic goal (reaching the inconsistent state that breaks ϕ in Figure 1c) rather than a deep symptom to stumble upon (the out-of-bounds read in Figure 1f). A resulting violation of ϕ surfaces the divergence between the agent's model $\hat{\Pi}_P(I)$ and the program's actual behavior $\Pi_P(I)$ and serves as a *lead* that guides the agent to refine its understanding of the code. In our example in §2.1, the agent's belief that `fmt.channels` = `cs.channels` (Figure 1c) can become the invariant ϕ, which a fuzzer-generated 13-versus-1 `profile` input $i_{\neg\phi}$ later falsifies, surfacing the `PT_ANY` bypass missed by purely static reasoning.

4. System Design

Figure 3 shows CODE-AUGUR's workflow. Given a codebase P, CODE-AUGUR first distills a threat model from the code, its documentation, and its build context. This threat model captures the project's security boundary (e.g., the attack surface and trust boundaries) and the security-relevant properties that code within this boundary must uphold, serving as the global context about the project for the subsequent audit.

Guided by the threat model, CODE-AUGUR enters the **Invariant Analysis** stage (①), where `Static Reasoning` inspects the code, asking what each site should guarantee under this security boundary. When a site appears insecure, CODE-AUGUR flags it directly as a *bug candidate* (along the dashed "*Flag*" path), as a conventional agentic audit would. When a site is deemed secure, CODE-AUGUR instead records the assumptions behind this verdict as invariants, each committed into the source as an assertion, yielding the instrumented program P'.

In **Invariant Falsification** (②), CODE-AUGUR leverages grey-box fuzzing to run P', seeking to falsify any committed invariant ϕ. `Fuzzing Preparation` builds P' into a fuzzing target whose feedback rewards progress toward a violation $\neg\phi$. Guided by this feedback, the `Guided Fuzzer` searches for a violating input. `Violation Triage` then inspects each violation. A violation that reflects a genuine security issue becomes a bug candidate. Such a violation is not always the vulnerability itself: it is often an early inconsistency that leads the agent to a downstream defect, as in our motivating example (§2.1). A benign divergence, by contrast, reveals a flaw in the agent's understanding of the code rather than a bug. CODE-AUGUR then updates that understanding, refining this invariant as necessary. This reason-falsify-refine loop progressively aligns the agent's view of the program with the program's actual behavior.

Finally, every bug candidate is validated before being reported. The audit's final artifacts are the threat model, the surviving invariants, and the validated bug reports.

4.1. Threat Model Construction

CODE-AUGUR begins by distilling the global context for the audit, including the threat model. Reading the codebase, its documentation, and its build configuration, CODE-AUGUR reasons at a high level, aiming to capture the security properties the system is meant to preserve rather than the details of any single function. It records the threat model as a structured document. We show an example snippet of the threat model generated for the `Little CMS` running example in Listing 1. Notably, we can see that in this case, CODE-AUGUR identifies that the pixel format and channel count are *important* variables within the global context, helping it later deduce the critical *local* invariant that $\mathbb{C}_{\texttt{PixelFormat}} = \mathbb{C}_{\texttt{ColorSpace}}$.

```
attacker_control: input ICC profile bytes
in_scope: transform construction, pixel unpacking
trust_boundary: profile fields -> transform state
security_relevant_state:
  - pixel format channel count
  - color space channel count
out_of_scope:
  ...
```

Listing 1. Threat-model fragment for the motivating example.

A threat model contains many different fields, each with different purposes throughout the audit. For example, the `security_relevant_state` field lists variables that relate to possible global security properties. The `attacker_control` and `in_scope` fields tell CODE-AUGUR which inputs and execution paths matter, `trust_boundary` marks where invariants belong, and `out_of_scope` bounds which failures bug validation may report. Together the threat model is used throughout the overall workflow, guiding the static reasoner towards in-scope and relevant source locations, giving relevant state variables from which to build invariants during invariant analysis, and giving context to the validation and triage steps for what constitutes a valid report for a given project.

4.2. Invariant Analysis

Algorithm 1 details CODE-AUGUR's *iterative* invariant analysis, whose LLM-driven steps are GENERATEHYPOTHESIS, EVALUATEHYPOTHESIS, and REFINEINVARIANT. Given the codebase P and the threat model T, CODE-AUGUR repeats a reason-falsify-refine cycle, flagging bug

Algorithm 1: Iterative Invariant Analysis

```
   Input  : P – program under audit
            T – threat model
   Output : B – bug candidates
            Φ – surviving invariants
1  Φ ← ∅,   B ← ∅,   C ← ∅
2  repeat
       /* Static Reasoning: reason about a site against T  */
3      (h, s) ← GenerateHypothesis(P, T, Φ)
4      (b, φ) ← EvaluateHypothesis(P, T, h, s)
5      if b ≠ ⊥ then
6          B ← B ∪ {b}                          // insecure site: flag
7      else if φ ≠ ⊥ then
8          repeat
9              P′ ← Instrument(P, Φ ∪ {φ})
10             (ok, e) ← Check(P′, φ)
11             if ¬ok then φ ← RepairAssertion(φ, e)
12         until ok
13         (b, fb, C) ← Falsify(P, P′, φ, C)
14         if b ≠ ⊥ then
15             B ← B ∪ {b}                      // a real bug
16         else if fb ≠ ⊥ then
17             φ ← RefineInvariant(T, φ, fb)
18         else
19             Φ ← Φ ∪ {φ}                      // φ survives
20 until audit budget is exhausted
21 return B, Φ
```

candidates B and accumulating the invariants Φ that survive to later iterations.

Static Reasoning. In `Static Reasoning`, CODE-AUGUR forms hypotheses regarding the source code against the threat model (GENERATEHYPOTHESIS, line 3). Each hypothesis h indicates a developer assumption about a location s (e.g., a function) that it proposes can be broken. These hypotheses are grounded on one side by the source code s and on the other by an attacker capability declared by the threat model. After forming h, CODE-AUGUR first evaluates it statically, using iterative LLM-based reasoning to either flag a hypothesis as a true candidate or as unreachable within the context of the project (EVALUATEHYPOTHESIS, line 4).

For example, in `Little CMS`, the threat model states that the attacker controls the ICC-profile bytes that flow into transform construction (Listing 1). Based on this information, CODE-AUGUR might hypothesize when analyzing `CreateTransform` (Figure 1c) that a named-color profile passes the color-space guard while storing a pixel format whose channel count differs from the transform's color space. The distinction between local behavior and intended property here is critical; seeing the call to `IsProperColorSpace` in isolation might lead the model to believe that *the color models match*. But as we saw in §2.1, this guard statement is *insufficient* to uphold the intent recorded in the threat model.

If the hypothesis is deemed plausible by this evaluation, CODE-AUGUR *does not* directly report it. This is because hallucinations and other errors in LLM-reasoning at this stage could lead to false positive bug reports. Instead, CODE-AUGUR emits a bug candidate b (line 6) to be *validated* by a concrete execution (§4.4). If CODE-AUGUR instead deems the hypothesis to be invalid, it commits a falsifiable invariant ϕ that records the assumption on which that verdict rests, described below.

Committing Invariants. Each committed ϕ is tied to the threat model through a trust boundary or attack surface. Additionally, the assertions are not arbitrary functional properties but the conditions the threat model identifies as *security-relevant*. CODE-AUGUR inserts each ϕ into the project source as a tagged assertion and rebuilds the project as P' without any further change to its build configuration (line 9). The assertion takes whatever form is native to the project's source language: sanitizer-style traps for C and C++, standard assertions for Go, Jazzer [14]-compatible feedback channels for the JVM, and so on. The only requirement is that an invariant violation at runtime is structurally distinguishable from other program failures, so it can be identified and triaged later. Each invariant ϕ therefore carries a unique identifier. After instrumentation, CODE-AUGUR checks that P' builds successfully and that the inserted assertion is syntactically valid (line 10), repairing the assertion and rebuilding until it is valid. This iterative build-time repair only fixes a malformed assertion, and is distinct from the semantic refinement in §4.3.

Notably, these invariants Φ can generalize far beyond an individual bug report, which we examine further in §6.5. For example, due to its placement in the source code, the inferred `Little CMS` invariant guards other, similar conversions present in the library but not explored via `TestHarness.c`. Indeed, the same predicate can be re-checked on the symmetric output-format path, on other entry points, and on later revisions of the library, flagging code that re-introduces the wildcard bypass.

Algorithm 2: Invariant Falsification

```
1  Procedure Falsify(P, P′, φ, C)
       /* P: original program; P′: instrumented program;
          φ: target invariant; C: seed corpus              */
       /* Fuzzing Preparation: build target, test harness,
          extend corpus                                    */
2      (H, C) ← FuzzingPreparation(P′, C)
       /* Specification Guided Fuzzer: seek a witness that
          violates φ                                       */
3      (i_¬φ, C) ← SpecGuidedFuzz(P′, H, φ, C)
4      if i_¬φ = ⊥ then return (⊥, ⊥, C)        // φ holds
       /* Violation Triage: a real bug, or φ too strong?   */
5      (b, fb) ← Triage(i_¬φ, φ, P, P′)
6      return (b, fb, C)
```

4.3. Invariant Falsification

FALSIFY (Algorithm 2) is the *asynchronous* subroutine the main loop invokes on each committed invariant (Algorithm 1, line 13). It takes the original program P, the instrumented program P', the target invariant ϕ, and the running seed corpus C, and returns whether the fuzzer found

a real defect, a too-strong invariant, or no violation at all with a justification feedback fb. Recall from §2.2 that, for an input $i \in I$, $\Pi_P(i)$ denotes the executions P admits on i. If ϕ holds, the corresponding assertion instrumentation should not change the program's behavior on any input, i.e., $\Pi_P(i) = \Pi_{P'}(i)$. The `Guided Fuzzer` aims to validate this by searching P' for a witness $i_{\neg\phi}$ that violates ϕ, i.e., $\Pi_P(i_{\neg\phi}) \neq \Pi_{P'}(i_{\neg\phi})$.

Falsifying and Refining. Each committed invariant is handed straight to the *asynchronous* FALSIFY subroutine (line 13), which fuzzes P' for an input that breaks ϕ and returns one of three outcomes. If no input breaks ϕ, the invariant holds and joins the surviving set Φ (line 19). If the breaking input is a genuine defect, it becomes a bug candidate in B. If ϕ was merely too strong, i.e., the behavior that violates it is in fact secure, the gap lies in CODE-AUGUR's understanding rather than in the code, so CODE-AUGUR refines the invariant ϕ (REFINEINVARIANT, line 17). This reason-falsify-refine loop runs until the audit budget is exhausted.

Grey-Box Fuzzing. CODE-AUGUR achieves falsification via *fuzzing* [12]. Fuzzing is a randomized search over the input domain of a program (the search space) with the goal of finding inputs that trigger vulnerabilities. The search process for *grey-box* fuzzers is adaptive, biasing the search towards interesting inputs using a feedback function. The feedback for a given input involves an assessment of whether novel program behavior was exposed by that input's execution. Typically, novelty is determined by a measure of code coverage, checking whether new program locations instrumented at compile time were executed at runtime.

Fuzzing Preparation. Before any campaign within the broader audit, CODE-AUGUR builds the instrumented source code P' into a runnable grey-box fuzzing target (line 2). As part of the build process, CODE-AUGUR uses an existing fuzz test harness $\mathcal{H}$ when one is available, refining it as needed, or constructs a new test harness otherwise. It also wires each committed invariant into the fuzzer's feedback mechanism such that *approaching* an invariant results in gradual, observable progress for the fuzzer. CODE-AUGUR persists a corpus of interesting fuzzing inputs C across calls to FALSIFY, so exploration is cumulative as invariants are committed and refined.

As part of the transformation from P to P', P' aborts whenever a committed invariant is *first* violated by a fuzzer input, triggering triage (discussed at the end of this subsection). Subsequent triggers of the invariant do not immediately abort, allowing the fuzzer to progress and find crashes beyond the initial violating state. Moreover, each invariant exports a feedback channel in P' for when the invariant itself has been tripped, whether it has been reached, and how close the invariant is to tripping (via a bucketed distance function when available, described in §5). As a result, rather than being purely guided by code coverage, as is typical for grey-box fuzzers [13], the engine treats progress toward an inconsistent state as interesting behavior and steers toward it. These invariants also represent an easier-to-reach goal than directly finding a crash.

Specification Guided Fuzzing. The `Specification Guided Fuzzer` takes the instrumented target P' and the invariant ϕ, and searches for a witness $i_{\neg\phi}$ that violates it (line 3). It exercises P' with an off-the-shelf engine such as libFuzzer [13] or Jazzer [14]. Rather than adapting the fuzzer *implementation*, CODE-AUGUR instead adapts the target's failure surface and feedback. This design choice allows for CODE-AUGUR to adopt new, more advanced fuzzing algorithms and implementations in the future for more effective falsification. Both sanitizer crashes and invariant violations found by the fuzzer are handed to the triage component.

Violation Triage. `Violation Triage` takes a witness $i_{\neg\phi}$ and its invariant tag and uses LLM-based reasoning to decide which of two interpretations applies (line 5). Either $i_{\neg\phi}$ exposes a real bug in P, or ϕ was too strong and the violation is benign. To decide, CODE-AUGUR reproduces the violation and inspects the program state at the assertion site. When the underlying behavior is genuinely insecure, it returns a bug candidate b, carrying the violating input and the invariant tag. Otherwise it returns feedback fb that explains why ϕ was too strong, which the main loop uses to refine the invariant (REFINEINVARIANT, line 17).

4.4. Bug Validation

The final stage turns the bug candidates B into reported vulnerabilities R by validating them.

Proof-of-Vulnerability Construction. A candidate surfaced by the fuzzer already carries a witness input, but a statically flagged candidate does not. For the latter, CODE-AUGUR first constructs a Proof of Vulnerability (PoV): an input that drives the defect from a program-boundary entry point rather than by invoking an internal function directly. Anchoring the PoV at the boundary establishes that the defect is reachable the way a real attacker would reach it, not through an interface no adversary controls. If CODE-AUGUR cannot produce a boundary PoV for a given bug candidate, it does not report the bug.

Confirming the Candidates. With a reproducing input in hand, CODE-AUGUR decides whether the candidate is a genuine vulnerability. CODE-AUGUR confirms the bug *only* if it deems it to be reachable under the attacker capabilities as declared by the threat model. Many reports, even those with a PoV, may fall outside the attacker's capabilities due to following a code path outside the in-scope attack surface. In these cases, CODE-AUGUR does not surface them as vulnerabilities.

5. Implementation

CODE-AUGUR is implemented on top of `pi v0.77.0`, an AI agent toolkit [15], in 9.1K lines of TypeScript for the runtime and subagent orchestration. The subagents have access to standard coding tools, including `read_file`, `edit_file`, Language Server Protocol queries, `bash`, and `grep`, as well as workflow-specific tools such as

`spawn_subagent` and `steer_subagent` for controlling subagents and `complete_stage` for reporting stage completion. We run all CODE-AUGUR agents and subagents with the model's thinking disabled and temperature set to 0.

Fuzzing and Instrumentation We use off-the-shelf grey-box fuzzing tools for each language in our evaluation: libFuzzer [13] for C/C++ programs, Jazzer [14] for the JVM, and the native fuzzing drivers for Go and Rust. CODE-AUGUR adds invariants as *source annotations* in the target language. For a subset of binary-operator invariants with numeric arguments, CODE-AUGUR also emits fuzzer-specific annotations, when available, to indicate how close the current execution is to violating the invariant. For example, if an invariant consists of `len` $\leq$ `max`, the distance is the absolute value of the difference between `len` and `max`. In the C/C++ backend, each invariant uses 16 libFuzzer extra-counter slots: one for reachability, one for tripping, and 14 logarithmically sized range buckets for numeric distances. In the JVM backend, CODE-AUGUR uses Jazzer's native `exploreState` for reachability and `minimize` for the distance-to-violation value [14]. In both cases, these signals *do not* modify the fuzzing algorithm; they only make the inferred safety boundary *visible* to the existing coverage-guided search. For other backends such as Go and Rust, CODE-AUGUR currently does not support distance-guided feedback, though it can be easily extended to do so.

6. Evaluation

We evaluate CODE-AUGUR to answer the following research questions:

RQ1 (Effectiveness) How effective is CODE-AUGUR compared to other agentic baselines in bug discovery?

RQ2 (Component Contribution) How does each component of CODE-AUGUR contribute to its effectiveness?

RQ3 (Security Impact) Is CODE-AUGUR effective in exposing unknown vulnerabilities in real-world software?

RQ4 (Usefulness of Artifacts) How useful are the inferred security specifications across the software's lifecycle?

6.1. Evaluation Setup

Benchmarks. We evaluate CODE-AUGUR on two complementary benchmarks of known vulnerabilities in real-world software. AIxCC is a *controlled* benchmark of curated vulnerabilities seeded into mature open-source projects, whereas OSV is an *in-the-wild* benchmark of vulnerabilities independently discovered in deployed software. In both, every vulnerability comes with a reproducible PoV, so that each finding can be verified against a known ground truth.

1) **The AIxCC benchmark** (Table 1) is drawn from DARPA's AI Cyber Challenge [7], a competition for developing autonomous cyber reasoning systems (CRSs) that detect and repair vulnerabilities in software. Its challenges are carefully constructed by the organizers from mature, widely-used open-source projects: each project is seeded with a set of known vulnerabilities. From these, we use all full-scan challenges (whole-repo scan tasks that match our setting) and retain the vulnerabilities with a reliably reproducing PoV, yielding 39 vulnerabilities across 9 projects spanning C, C++, and Java. Except for `nginx`, which served as a demonstration challenge, all challenges were publicly released in May 2026.
2) **The OSV benchmark** (Table 2) is a living, rotating set of recently disclosed vulnerabilities in widely-used open-source projects that we assemble from the OSV database [8]. We use OSV rather than the more widely used CVE database because each OSV record ties a vulnerability to its introducing and fixing commits and to a public PoV with the disclosed crash information—the per-bug provenance a reproducible benchmark needs but CVE rarely provides. To mitigate data-leakage risk, we restrict the snapshot to the most recently disclosed records, covering February–April 2026. Of the 45 vulnerabilities disclosed in this window, only 29 provided a public PoV at the time of collection, of which 24 reproduce reliably—yielding 24 confirmed bugs across 9 projects spanning C, C++, Java, and Rust.

Comparison Tools. We compare CODE-AUGUR against agent baselines that span the design space of our approach:

1) **CLAUDE CODE** [6], Anthropic's frontier general-purpose coding agent, represents the minimally-structured harness in the design space. It audits a codebase by reasoning over it directly. It can also invoke tools, e.g., code execution, to ground its analysis, but does so in a completely *ad-hoc* manner. Run under the same model as CODE-AUGUR, CLAUDE CODE demonstrates the utility of CODE-AUGUR's more structured, specification-first approach over ad-hoc agentic reasoning alone.
2) **ATLANTIS** [9], [16], [17], the cyber reasoning system from Team Atlanta that won the AIxCC final, combines the same two ingredients (an LLM agent and a fuzzer) as CODE-AUGUR. However, in contrast to CODE-AUGUR, ATLANTIS represents a fuzzing-centric approach: an ensemble of fuzzers and concolic executors forms its base infrastructure, while LLM agents support the fuzzing campaign. These LLM-based components serve as a semantic front-end that feeds and steers fuzzing, including generating format-aware seeds, selecting fuzzing targets, and synthesizing PoVs for sinks the fuzzer reaches but cannot trigger. Comparing against ATLANTIS thus isolates the contribution of our infer-falsify-refine loop over supplementing a fuzzer with agentic capabilities.

Configurations. We run every system through OSS-CRS [18], the open-source orchestration framework for evaluating Cyber Reasoning Systems (CRSs) maintained under the OpenSSF. A CRS performs both bug detection and repair. We evaluate only the detection task that CODE-AUGUR targets. OSS-CRS runs any tool against an OSS-Fuzz-format target via a designated test harness, as in the AIxCC competition [7], which omits the false positive issues since all submitted PoVs are through a trusted entry point.

Resources and Models. Following the AIxCC semifinal budget [19], each system runs for 4 h per challenge with up

TABLE 1. BUG-FINDING RESULTS ON THE AIXCC BENCHMARK FOR DIFFERENT TOOLS WITH DIFFERENT BACKING MODELS.

Project	Challenges	CODE-AUGUR				ATLANTIS				CLAUDE CODE			
		Claude		DeepSeek		Claude		DeepSeek		Claude		DeepSeek	
		Existing	New	Existing	New	Existing	New	Existing	New	Existing	New	Existing	New
nginx	challenge-04_1	8/11	4	3/11	1	6/11	0	4/11	1	6/11	1	4/11	2
nginx	challenge-04_2	1/2	0	2/2	1	2/2	0	2/2	0	2/2	0	2/2	0
nginx	challenge-04_3	1/1	0	1/1	2	1/1	0	1/1	0	1/1	0	1/1	0
dav1d	dav1d-001	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	1
little-cms	lcms-001	1/1	0	1/1	0	0/1	0	0/1	0	1/1	0	0/1	0
little-cms	lcms-002	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0
mongoose	mongoose_0	1/1	1	1/1	0	1/1	0	1/1	0	1/1	0	1/1	1
apache-poi	vuln_0,vuln_1	2/2	1	2/2	1	2/2	0	2/2	1	2/2	6	2/2	1
apache-poi	vuln_2	1/1	8	1/1	5	1/1	9	0/1	10	1/1	1	0/1	0
apache-poi	vuln_3	1/1	0	1/1	1	0/1	0	0/1	0	1/1	1	1/1	0
apache-poi	vuln_4	1/1	1	0/1	0	0/1	1	0/1	0	1/1	0	1/1	0
shadowsocks	libev_0-4	5/5	1	5/5	2	1/5	0	1/5	0	5/5	0	5/5	0
systemd	systemd-001	1/1	0	1/1	1	1/1	0	1/1	0	1/1	0	1/1	0
systemd	systemd-003	1/1	0	1/1	1	1/1	0	1/1	0	1/1	0	1/1	0
systemd	systemd-004	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
systemd	systemd-005	1/1	2	1/1	2	0/1	0	0/1	0	0/1	0	1/1	0
wireshark	vuln_001	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
wireshark	vuln_002	1/1	5	1/1	2	1/1	1	0/1	2	1/1	1	1/1	1
wireshark	vuln_005	1/1	1	1/1	1	1/1	0	1/1	0	1/1	2	1/1	0
wireshark	vuln_010	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
wireshark	vuln_011	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
wireshark	vuln_012	1/1	1	1/1	1	1/1	0	1/1	0	1/1	0	1/1	1
xz	xz-001	1/1	1	1/1	1	1/1	0	1/1	0	1/1	0	1/1	0
Subtotal		33/39	26	29/39	22	25/39	11	21/39	14	32/39	12	29/39	7
Total (Existing+New)		**59**		**51**		**36 (+63%)**		**35 (+45%)**		**44 (+34%)**		**36 (+41%)**	

- **Existing**: known bugs reproduced ("*x/total*"). **New**: newly discovered bugs at the benchmarked version. **Total (Existing+New)**: the two combined.
- **Parenthesized %**: CODE-AUGUR's relative improvement over that baseline under the same backing model, computed over the subjects where both ran successfully.

to $100 in LLM API credits, in a Docker container capped at 12 CPU cores and 32 GB of RAM. All experiments run on an Intel Xeon Platinum 8468V server (96 cores, 512 GB of RAM). To separate the effect of the underlying LLM from that of the tool built around it, we run each agentic system under two frontier LLMs: the *proprietary* Claude Sonnet 4.6 [20] and the *open-weight* DeepSeek V4 Pro [21] (we use "Claude" and "DeepSeek" for short hereafter).

Prompt for Baselines. We run both baselines exactly as packaged in OSS-CRS, rather than tuning their prompts ourselves, so that neither is advantaged or disadvantaged by our setup. For CLAUDE CODE, we use the framework's bug-finding Claude Code agent [22], prompted to audit the target and produce proof-of-vulnerability inputs for the given test harness. For ATLANTIS, we reuse the competition configuration its authors contributed to the framework.

6.2. RQ1: Reproducing Known Vulnerabilities

We compare CODE-AUGUR with the two baselines on both benchmarks. For each tool, we measure the two quantities reported in Tables 1 and 2: (1) "*Existing*", the number of known, ground-truth vulnerabilities it reproduces, i.e., its recall against a reproducible PoV; and (2) "*New*", the number of unique additional bugs found on those subjects at their benchmarked versions, de-duplicated by top-3 stack trace locations. We report the results of each agentic system under both Claude and DeepSeek. Because DeepSeek V4 Pro is substantially cheaper to run, we repeat each of its runs four times and report the average. All tools see the same projects, versions, entry points, and per-subject budget (§6.1), and every finding is confirmed by the same reproduction-and-deduplication procedure.

AIxCC Benchmark Results. Table 1 presents the results on the AIxCC benchmark. On average, CODE-AUGUR finds 34–63% more bugs than either ATLANTIS or CLAUDE CODE, and this advantage holds under both Claude and DeepSeek. On the existing bugs already known to the competition, CODE-AUGUR is comparable with CLAUDE CODE and better than ATLANTIS. But for new bugs, CODE-AUGUR finds more than *twice* as many vulnerabilities as either baseline. Moreover, CODE-AUGUR finds these new bugs across more subjects than the baselines: combining both models, CODE-AUGUR uncovers new bugs on 16 of the 23 challenge groups, against 9 for CLAUDE CODE and 5 for ATLANTIS. Its advantage therefore reflects broad effectiveness rather than a few favorable targets.

OSV Benchmark Results. Table 2 reports the results on the OSV benchmark. ATLANTIS fails to run on `mongoose` and `wasmtime` due to configuration issues on that version and lack of Rust support, respectively. Overall, CODE-AUGUR finds 61–370% more bugs than the baselines. As on the AIxCC benchmark, the vast majority of these additional bugs were *not previously known* in the benchmark, and CODE-AUGUR again finds them more broadly than the baselines, uncovering new bugs in 12 of the 13 commits,

TABLE 2. BUG-FINDING RESULTS ON THE OSV BENCHMARK FOR DIFFERENT TOOLS WITH DIFFERENT BACKING MODELS ("–" INDICATES THE TOOL FAILED TO RUN ON THE GIVEN PROJECT)

Project	Commit	CODE-AUGUR				ATLANTIS				CLAUDE CODE			
		Claude		DeepSeek		Claude		DeepSeek		Claude		DeepSeek	
		Existing	New	Existing	New	Existing	New	Existing	New	Existing	New	Existing	New
apache-poi	eafd6c0	0/2	1	0/2	0	0/2	0	0/2	0	0/2	0	0/2	0
assimp	21607df	0/1	4	0/1	11	1/1	19	0/1	2	0/1	24	0/1	7
gpsd	4f56109	1/1	3	1/1	7	1/1	0	1/1	0	1/1	0	1/1	0
gpsd	d700650	0/3	11	1/3	7	0/3	0	0/3	0	0/3	0	0/3	0
gpsd	50153e3	0/4	11	0/4	3	0/4	0	0/4	0	0/4	0	0/4	1
gpsd	1c9dd87	1/3	12	1/3	5	1/3	0	1/3	0	2/3	0	1/3	0
pjsip	2b7c8b5	1/1	0	1/1	1	0/1	0	0/1	0	1/1	0	1/1	0
libhevc	6763519	0/1	0	0/1	1	1/1	0	1/1	0	0/1	0	0/1	0
libical	d6f6b1c	2/3	1	2/3	1	3/3	0	3/3	0	3/3	1	1/3	1
libical	fe0de01	1/2	2	1/2	2	1/2	1	1/2	0	2/2	0	2/2	2
tinyobjloader	7dbc543	1/1	2	1/1	1	1/1	0	1/1	0	1/1	0	1/1	0
wasmtime	d54924a	0/1	0	0/1	0	–	–	–	–	0/1	0	1/1	0
mongoose	ba76869	1/1	3	1/1	1	–	–	–	–	0/1	1	0/1	0
Subtotal		8/24	50	9/24	40	9/22	20	8/22	2	10/24	26	8/24	11
Total (Existing+New)		**58**		**49**		**29 (+86%)**		**10 (+370%)**		**36 (+61%)**		**19 (+157%)**	

- **Existing**: known bugs reproduced ("*x/total*"). **New**: newly discovered bugs at the benchmarked version. **Total (Existing+New)**: the two combined.
- **Parenthesized %**: CODE-AUGUR's relative improvement over that baseline under the same backing model, computed over the subjects where both ran successfully.

against 5 for CLAUDE CODE and 3 for ATLANTIS. Notably, all of these programs are kept under *continuous fuzzing* via OSS-Fuzz [23], so the bugs are *non-trivial*, having evaded detection at the time of benchmark construction.

Cross-Model / Agent Comparison. Looking at both Table 1 and Table 2, we can see that in 10/12 subtotal configurations, agents backed by Claude outperform or are at parity with the same agent backed by DeepSeek, with the lone exception being that ATLANTIS backed by DeepSeek finds 3 more *new* bugs on the AIxCC benchmark. Importantly, CODE-AUGUR using DeepSeek finds more total vulnerabilities on average than either ATLANTIS or CLAUDE CODE using Claude across both benchmarks. In other words, CODE-AUGUR's specification-first approach can leverage an *open-weight model* more effectively than other agents can leverage a *frontier* model. We believe this result is of increased importance with recent public attention on expensive or limited availability specialized frontier models, such as Claude Mythos [1].

6.3. RQ2: Component Contribution

To better understand CODE-AUGUR's effectiveness, we measure each component's contribution by attributing every confirmed vulnerability CODE-AUGUR finds on both benchmarks (§6.2) to the component that *directly* surfaced it. Each finding falls into one of three discovery lanes: (1) *invariant falsification*, where the fuzzer breaks an agent-committed invariant, guiding the agent to analyze the resulting violation and trace it to a bug; (2) *fuzzing*, where the fuzzer triggers an ordinary sanitizer crash without violating an invariant; and (3) *code review*, where the agent flags a suspect while reviewing the code and reports it once validated, the mechanism that conventional agentic bug scanners rely on.

TABLE 3. CODE-AUGUR'S EXPOSED VULNERABILITIES, BROKEN DOWN BY DISCOVERY LANE FOR EACH BENCHMARK AND BACKING LLM. THE THREE LANE COUNTS IN EACH ROW SUM TO *Total*.

Benchmark	LLM	Total	Invariant	Fuzzing	Code Review
AIxCC	Claude	59	21	9	29
	DeepSeek	51	17	10	24
OSV	Claude	58	24	9	25
	DeepSeek	49	13	13	23

Results. Table 3 shows that all three discovery lanes contributed to CODE-AUGUR's success on both benchmarks. Fuzzing alone was able to surface a small number of relatively shallow bugs on both benchmarks. However, across backing LLMs, a *large* number of vulnerabilities can be detected using only *static reasoning* via code review. This finding supports recent anecdotal evidence that scanning code with AI agents can already identify many new vulnerabilities [2].

Despite the effectiveness of code review, we also see that inferred invariants directly lead to a large number of vulnerability discoveries. Indeed, nearly half (36-41%) of all vulnerabilities discovered by CODE-AUGUR with Claude were found via invariant trips. However, with DeepSeek, this number decreases substantially. Indeed, *nearly all* of the drop in effectiveness between CODE-AUGUR using Claude and DeepSeek as backing models on the OSV benchmark can be attributed to the fewer bugs found via invariant trips. On manual inspection, we found that DeepSeek struggles to maintain focus on the longer horizon task of setting and refining invariants. A possible future line of research could be into more sophisticated context management to alleviate this issue for open-weight models.

We also examine the auditing cost. Averaged over both

benchmarks, an audit costs ($45.41, $2.19) for CODE-AUGUR, compared to ($73.90, $3.20) for ATLANTIS and ($16.04, $0.30) for CLAUDE CODE, under the form of "(Claude Sonnet 4.6 [20], DeepSeek V4 Pro [21])". Unlike CODE-AUGUR and ATLANTIS, the default CLAUDE CODE harness does not record exact pricing, so we compute its cost from token usage in the session transcripts, de-duplicated by message ID and priced at published provider rates. We see that CODE-AUGUR is similar in cost to alternatives *and* that it gives comparable results on the cheaper DeepSeek to those on the more expensive Claude across our two benchmarks. Moreover, CODE-AUGUR with DeepSeek already exceeds the capabilities of baselines backed by Claude at a fraction of the cost! Moreover, as the inferred invariants are committed into the source as durable instrumentation, much of this cost is a one-time investment that can be amortized over later audits.

6.4. RQ3: Detecting Unknown Vulnerabilities

We next assess whether CODE-AUGUR finds previously unknown vulnerabilities in the wild, beyond the two benchmarks above. We ran CODE-AUGUR on a set of widely-used and well-tested open-source projects that together span multiple programming languages and application domains, from network protocols and daemons to web applications and libraries. Here we follow the same setup as our benchmark experiments (§6.1), with two relaxations suited to a real-world audit rather than a controlled benchmark comparison: CODE-AUGUR is given no designated test harness or entry point, and a finding need not trigger a crash but may instead be a logic flaw. CODE-AUGUR itself attempts to synthesize a working PoV for each candidate. For memory-safety bugs this PoV is a crashing input, whereas for logic and access-control flaws it is an input that violates a high-level security property outlined in the threat model, but may not necessarily crash the program. Because such findings may include logic flaws that no sanitizer can flag, we validated each candidate manually, reading CODE-AUGUR's report and any PoV it produced, tracing the issue through the source code, and judging whether the behavior genuinely violates the developers' intent. We count a finding only if it is previously unknown, has no public report at audit time, and passes this manual validation. We disclosed every confirmed finding to its maintainers under the responsible disclosure policy.

Results. Table 4 lists the 22 previously unknown vulnerabilities CODE-AUGUR discovered across seven widely-used open-source projects. Their maintainers have already fixed or confirmed 16 of them, and two have been assigned CVEs (CVE-2026-48113 for Bug #2 in `chisel` and CVE-2026-34830 for Bug #20 in `rack`). Remaining reports are undergoing CVE assignment. Due to their severity, the findings on `lightway` have also earned bug bounty rewards totaling $1,400. These are all mature, widely-used projects. Some of them, such as `zlib` and `gpsd`, are even kept under continuous fuzzing in OSS-Fuzz [23], so surfacing new bugs in them is non-trivial. Meanwhile, the findings span diverse vulnerability classes, from classic memory-safety bugs (e.g., the heap-buffer-overflow in `gpsd`'s `ais_binary_decode()`) to logic and access-control flaws such as `chisel`'s post-handshake ACL bypass and `ntpd-rs`'s IPv4-mapped filter bypass. Through manual inspection, we further find that many of these findings hinge on counter-intuitive inputs that an agent reasoning over the code alone is unlikely to anticipate. For example, Bug #7 requires a malformed packet that slips past a length check which only warns instead of returning, while Bug #21 requires a ZIP64 extra field whose 20-byte block a 16-bit size guard silently omits. Such inputs are hard for either paradigm alone: a purely analytical agent is likely to overlook such counter-intuitive inputs, while plain fuzzing struggles to reach the deep states behind them. CODE-AUGUR makes them far more reachable by turning the agent's invariant into a *shallow, semantic* target the fuzzer can reach and falsify, rather than a deep crash it must stumble upon.

```
1  #define MAXCHANNELS 184 // a receiver's channel capacity
2  int satellites_visible; // number of satellites currently in view
3  // Code-Augur inferred invariant φ: satellites_visible <= MAXCHANNELS
4  // Producers: 10+ drivers set the satellites_visible from raw packet:
5  geostar(): satellites_visible = (int)getleu32(buf, ...);
6  sirf()   : satellites_visible = num_of_sats;
7  // Consumers: 10+ readers walk skyview using satellites_visible:
8  struct satellite_t skyview[MAXCHANNELS];
9  fill_dop()      : for (k=0;k<satellites_visible;k++) ... skyview[k]
10 json_sky_dump() : for (i=0;i<satellites_visible;i++) ... skyview[i]
```

Listing 2. The `satellites_visible` bug family in `gpsd`, shaded by role: the invariant ϕ over variables, the producers, and the consumers.

6.5. RQ4: Usefulness of Specifications

Invariants inferred by CODE-AUGUR are not only a one-time effort but durable artifacts that remain useful long after the audit that produced them.

We illustrate this with a case study (Listing 2 and 3) on `gpsd`, a deployed GPS daemon and a subject from the OSV benchmark. In Listing 2, `satellites_visible` is the number of satellites currently in view, and `MAXCHANNELS` is the receiver's channel capacity (lines 1–2). A receiver cannot report more satellites than it has channels, so during its audit CODE-AUGUR infers and records this semantic bound as the in-source invariant ϕ: `satellites_visible` $\leq$ `MAXCHANNELS` (line 3). On that basis it deems the downstream operations secure, including the 10+ *consumer* functions that walk `skyview` indexed by `satellites_visible` (lines 8–10). However, this verdict is fragile: `satellites_visible` also has 10+ *producer* drivers that assign it by directly copying raw packet fields (lines 5–6), so a crafted packet can drive it to a value such as 256 and falsify ϕ, which is how CODE-AUGUR surfaces the bug based on fuzzing's falsification.

Tracing `gpsd`'s subsequent development history shows why ϕ is *durable*: inferred *once*, it keeps flagging the same root cause across a four-month chain of remediations (Listing 3), first exposing an **incomplete fix**, then tying a whole **bug family** to that one cause, and finally revealing that even the bound ϕ assumes was itself too low.

(1) Exposing an incomplete fix. The family first surfaced as the disclosed report OSV-2026-189 [24], an out-

TABLE 4. SUMMARY OF PREVIOUSLY UNKNOWN VULNERABILITIES FOUND BY CODE-AUGUR

#	Project	Bug Description	Status
1	chisel	ACL bypass in User.HasAccess() when regexp.MatchString() evaluates unanchored ACL patterns	Confirmed
2	chisel	Access-control bypass in tunnel_out_ssh.go due to the unchecked host:port in a post-handshake channel's ExtraData()	Fixed
3	Ghost	LocalStorageBase.save() ignores type parameter, serving uploaded .html as text/html	Reported
4	Ghost	SSRF in Webhook.trigger() and Slack via @tryghost/request skipping request-external.js IP checks	Reported
5	gpsd	RTK baseline data in processPSTI030()/processPSTI032() silently discarded when PSTI opens a new cycle	Fixed
6	gpsd	Heap buffer overflow in ais_binary_decode() when a Type 21 message appends a 16-char name extension past name[35]	Fixed
7	gpsd	Out-of-bounds read in decode_itk_pseudo() on a malformed packet whose length check only warns instead of returning	Fixed
8	gpsd	One-byte buffer overflow in ais_binary_decode() when an un-padded 1008-bit Type 14 text writes past text[161]	Fixed
9	gpsd	Pointer write past field[] in NMEA field splitting when a sentence has more comma-separated fields than NMEA_MAX_FLD	Fixed
10	gpsd	Signed left-shift undefined behavior in the driver-identification bitmask (1 « driver_index) in libgpsd_core.c	Fixed
11	gpsd	Out-of-bounds write past sats_used[] in TSIP decode_x6c() when the 8-bit count field exceeds MAXCHANNELS	Fixed
12	gpsd	Out-of-bounds read in the RTCM3 CRC-failure path when logging indexes inbufptr instead of inbuffer in packet.c	Reported
13	gpsd	Out-of-bounds write past skyview[] in the Skytraq 0xDE decoder when records exceed MAXCHANNELS before the post-loop clamp	Fixed
14	gpsd	Source-side out-of-bounds read in the RTCM3 1008/1033 string decoders when later length fields skip cumulative bounds checks	Fixed
15	gpsd	Out-of-bounds read past skyview[] in the Skytraq 0xDE decoder when skyview[st].used is read after st is incremented	Fixed
16	lightway	try_from_wire() advances the anti-replay window by plaintext wire_counter before AES-GCM auth, dropping valid packets	Fixed
17	lightway	flags field excluded from the AES-GCM AAD, so flipping its encoded bit reroutes the packet into a fatal decoder error	Fixed
18	lightway	Plaintext version mismatch in outside_data_received() is treated as fatal, so one packet disconnects the client	Fixed
19	ntpd-rs	IPv4-mapped IPv6 filter bypass in IpFilter::is_in() on dual-stack sockets, skipping IPv4 allowlists rules	Reported
20	rack	Regex injection in Rack::Sendfile caused by interpolating the unescaped X-Accel-Mapping header into a path-rewrite regex	Fixed
21	zlib	Integer overflow in zipOpenNewFileInZip4_64() when the 16-bit extra-field guard omits the appended 20-byte ZIP64 block	Reported
22	zlib	NULL dereference in zipOpenNewFileInZip4_64() as ALLOC'd central_header is written for 40+ lines before NULL check	Reported

```
1  // (1). Patch (+1day): clamp the loop at ONE consumer (incomplete fix)
2  - fill_dop(): for (k=0; k<satellites_visible; k++)
3  + fill_dop(): for (k=0; k<min(satellites_visible, MAXCHANNELS); k++)
4
5  // (2). Patches: bound the count at the producer drivers, enforcing φ
6  // batch-1 (+5days): clamp the VALUE at ~10 producer drivers
7  - driver(): satellites_visible = <count from packet>;
8  + driver(): satellites_visible = min(<count from packet>, MAXCHANNELS);
9  // batch-2 (+8days..+4months): the sweep missed TSIP's own skyview[]
       fills
10 - tsip(): ... skyview[i] = ... // i ran past MAXCHANNELS
11 + tsip(): if (count < TSIP_CHANNELS) { ... skyview[i] = ... }
12
13 // (3). closure (+4months): a driver outgrew 184 (violating φ) but it
       is legit, indicating the array's capacity was undersized
14 - #define MAXCHANNELS 184
15 + #define MAXCHANNELS 230
16 + #if MAXCHANNELS < SKY_CHANNELS // SKY_CHANNELS = 230, SkyTraq's max
17 + #error // the array must cover SkyTraq
18 + #endif
```

Listing 3. The four-month remediation history of a bug family in gpsd: (1) clamping one crash site, an *incomplete fix*; (2) bounding the value at its multiple sources, fixing the whole *bug family* by enforcing ϕ; (3) enlarging the capacity, repairing the undersized bound. Across 4 months and 3 batches of fixes, each restores the one invariant ϕ CODE-AUGUR inferred once.

of-bounds read in the fill_dop consumer (line 9) that violates ϕ. The developers responded within a day, but their patch clamped only the loop in that single consumer fill_dop (Listing 3, lines 1–3). Such a loop-local fix could silence a sanitizer: the lone out-of-bounds access it would catch, in fill_dop, is now bounded, and the other consumers cannot be reached by this input, so the defect appears resolved. However, re-checking ϕ still flags a violation: the producers (e.g., geostar, line 5) keep storing the raw count, and the consumers the patch missed (e.g., json_sky_dump, line 10) remain unbounded. Because ϕ constrains the program state rather than a single access site, it catches this latent corruption even when no out-of-bounds access executes. This is exactly what a sanitizer-guided fix overlooks.

(2) Tying together a bug family. A recurrence soon makes the root cause more clear. A second report, a crash reached through the tsip driver, surfaced at a different site. Despite the different symptom, it violates the *same* invariant ϕ, so the two are not isolated bugs but one root cause shared across a family of producers and consumers. Recognizing this, the developers moved the fix to the *producers*: as shown in Listing 3, they bounded the count at its source across roughly ten drivers (lines 5–8) and extended the sweep to the cases the first pass missed, such as tsip's own skyview fills (lines 9–11). This remediation for such a bug family runs from five days to four months after the first report, and each of them enforces exactly what ϕ asserts.

(3) Revealing an undersized bound. A later input pushed satellites_visible past the upper bound MAXCHANNELS, yet triage showed the value to be *legitimate*: a SkyTraq receiver genuinely reports up to 230 satellites, beyond the maximum capacity MAXCHANNELS assumes (i.e., 184). The violation thus pointed not to a producer but to the bound itself. Upstream consequently enlarged the array to 230, under a compile-time guard ensuring it never falls below SkyTraq's channel count (Listing 3, lines 13–18).

Across all the three episodes, ϕ pins down the single property ϕ every complete fix had to restore, whether by bounding the value or enlarging the array's capacity, and outlives both the original report and each point fix. We are currently engaged in active discussions with developers of gpsd to merge this invariant as a defensive check against future regressions.

7. Related Work

Agentic and LLM Assisted Fuzzing. Many works have proposed using LLMs to assist fuzzing in harness generation [25], [26], [27], seed creation [28], and taint analysis [29]. This was also the strategy used by many teams

in the recent DARPA AIxCC [17], [30], [31]. Recent works have even begun to replace fuzzers altogether with LLM or agentic approaches [32], [33]. We have compared CODE-AUGUR with the winning AIxCC system. Additionally, while these approaches are all fuzzing-centric, we take an inverted approach in this work: fuzzing is a tool used by human security analysts and CODE-AUGUR alike to find edge cases and check assumptions. Indeed, advances in fuzzing can be leveraged by CODE-AUGUR's fuzzing component to increase its effectiveness in invariant falsification.

Agentic Vulnerability Scanning. Recently, agentic vulnerability scanning systems, such as Claude Mythos [1] and Google's Big Sleep [5] have gained widespread attention. In addition to custom models [1], several works have explored approaches such as Retrieval Augmented Generation (RAG) [34], deep-agents [35], and role-based multi-agent systems [36], [37], [38]. These systems, however, keep their security reasoning *implicit* within the agent, as discussed in §2.2. CODE-AUGUR instead externalizes it as explicit, executable security specifications and continuously refines them via dynamic falsification.

Specification Inference. Specification inference [39], [40] constitutes deduction of the *intended* behavior of code without explicit formalized descriptions of this behavior. LLMs have demonstrated impressive capacity to infer intended specifications related to *repairing* buggy code [3], functional requirements [41], and formal specifications [4], [42], [43]. A key contribution of this work is to likewise examine LLMs' capacity for inferring security specifications for vulnerability *discovery*. In the context of fuzzing, ECG [44] and Halo [45] infer *input specifications* to assist in embedded systems fuzzing and to constrain the input space in directed fuzzing, respectively. Similarly, Locus [46] infers specifications to *prune* infeasible code paths in directed fuzzing. These approaches contrast with the *local invariants* and *guidance* leveraged by CODE-AUGUR. Fioraldi et al. [47] also infer invariants to guide grey-box fuzzing, but do so from the behavior of *observed execution traces*, rather than based on the necessary conditions for code to be secure, as in CODE-AUGUR. FM-Agent [48] infers pre- and post-conditions for functions in large software projects and uses ad-hoc test-case generation to identify bugs. IRIS [49] and LLift [50] infer some specifications via LLM to assist in static analysis. However, rather than inferring specifications alone, CODE-AUGUR leverages comprehensive falsification via fuzzing to *validate and refine* its inferred specifications.

8. Limitations

Reliability of LLMs. Agentic vulnerability detection is limited by the inherent unsoundness of LLMs (i.e., the agent may misunderstand the code or even hallucinate). Our design combats such unsoundness in two ways: (1) *reasoning externalization and validation*: we turn the agent's implicit reasoning into explicit, *executable* invariants, an interface that (i) grounded tools can check, a process less susceptible to omissions or other failures in LLM reasoning itself, and that (ii) developers can also directly read; (2) *output validation*: CODE-AUGUR reports a vulnerability only with a generated proof-of-vulnerability input that reproduces it on the original program. This reason-falsify-refine loop makes CODE-AUGUR more resilient to its own reasoning flaws, helping uncover more bugs as shown in our evaluation.

We did not evaluate CODE-AUGUR in adversarial settings, where an attacker already has sufficient access to modify a project's source code and plant a back-door vulnerability. This work instead focuses on *inadvertent* vulnerabilities, which are far more common than malicious back-doors. We view detecting adversarially hidden vulnerabilities as an interesting direction for future research.

Data Leakage. LLMs are trained on enormous corpora that include code from many open-source projects, so data leakage [51], [52], where evaluation subjects overlap with the training data, threatens the validity of our results. We mitigate this risk along several fronts. First, we created a *new* benchmark, consisting only of bugs reported *after* the training cutoff for Claude Sonnet 4.6. Unfortunately, for the OSV-benchmark, at the time of writing, no reproducible vulnerabilities had yet been recorded in the OSV database [8] fully after the training cutoff for DeepSeek V4. However, we see a strong correlation between the results obtained by both CODE-AUGUR and other agents with DeepSeek and with Claude (§6), and thus we believe that the impact of the possible leakage is not significant. Second, all AIxCC subjects except `nginx` were publicly released *after* the training cutoffs of both models, so results on this benchmark are uncontaminated. Finally, CODE-AUGUR found many previously unknown bugs that we reported to developers, which by definition cannot be attributed to data leakage.

Model Context Window. Both backed LLMs in our evaluation have a context window limited to 1M tokens, which could be exhausted on very large programs. We mitigate this by designing CODE-AUGUR as a multi-agent system, where sub-agents are passed only the context necessary for their more granular tasks. We also set an automatic compaction of context to occur at 800k tokens, but we did not observe CODE-AUGUR reaching this limit in any of our experiments, including on `wireshark` with 2.1 million source lines of code.

9. Perspectives

Software security has always been a fragile balance of power between attackers and defenders. As new tactics and technologies emerge on the offensive side, they must be mitigated by new defenses. LLM agents have drastically shifted the balance in this equation; attackers with little to no expertise can point agents at open-source repositories and rapidly discover new vulnerabilities. We believe that merely replicating this workflow as a defensive measure is drastically insufficient: each vulnerability discovered represents only a single exploitable weakness rather than an underlying security principle. To reach parity, we formulate an approach in this work that extracts generalized knowledge from each security audit in the form of security specifications. By

making these specifications explicit and checkable, CODE-AUGUR achieves high efficacy in finding vulnerabilities and creates reusable artifacts that help future human and automated analysts reason about the security of their systems. As an increasing share of code is being written and developed by AI, deepened understanding provided by these specifications will be more valuable than ever. We believe that our specification-first approach can serve as the basis and inspiration for increasing the use of LLM-based security analysis.

9.1. Commentary about Claude Mythos

In this work, we have presented a rather complex agent harness for detecting security vulnerabilities. Our approach differs significantly, in both methodology and philosophy, from curated Large Language Models like Claude Mythos [1], which have gained recent attention.

- First and foremost, models like Claude Mythos currently have restricted access. Fable has been released, but its guardrails prevent it from being used for any security research and development. Thus our agent CODE-AUGUR, built on top of regular models like Claude Sonnet 4.6 and the open-weight DeepSeek V4 Pro, provides an alternative when curated models like Claude Mythos are not available. In an experiment on a `wolfSSL` bug [53] originally reported by Mythos, CODE-AUGUR with the DeepSeek V4 Pro backend also uncovered it, taking 2.25 hours and $1.39.
- Secondly, as mentioned earlier, we believe that instead of finding many bugs using LLMs, it is more important to generalize them in the form of specifications. These specifications, when deposited with a software project, can also prevent vulnerabilities from occurring in the future. Thus, they provide an additional form of regression checking, as we show in the case study in §6.5, which we cannot achieve using bug-finding models like Mythos alone.
- Last but not least, we strongly believe that CODE-AUGUR's invariant specifications—once deposited and refined—have widespread usages beyond vulnerability detection! Such usages would not be possible with models alone. The specifications are an important ingredient in systematizing and documenting the informal reasoning in an AI agent. A core problem in software engineering has always been understanding the developer's intent. Indeed, program analysis techniques have been used in automated program repair to extract a formal description of developer intent—both in the pre-LLM and post-LLM era [3], [54]. In the future, if agents are part of software teams, we will similarly need techniques for understanding agent intent, to effectively manage and maintain a software system. Our work envisions and supports this forward-looking perspective of future software engineering practice.

Acknowledgments

This research is supported by the National Research Foundation, Singapore, under its Artificial Intelligence (AI)-for-Science (AI4S) Challenge Grant (Award No. NRF-AI4SCH-2025-0003), called "AI for Program Reasoning". Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation.

References

[1] N. Carlini, N. Cheng, K. Lucas, M. Moore, M. Nasr, V. Prabhushankar, W. Xiao Hakeem Angulu, E. Ben Asher, J. Bow, K. Bradwell, B. Buchanan, D. Forsythe, D. Freeman, A. Gaynor, X. Ge, L. Graham, K. Guru, H. Lakhani, M. McNiece, M. Mehrara, R. Nichol, A. Pirzada, S. Porter, A. Terzis, and K. Troy, "Claude Mythos preview," 2026. [Online]. Available: https://red.anthropic.com/2026/mythos-preview/

[2] B. Grinstead, C. Holler, and F. Braun, "Behind the scenes hardening firefox with claude mythos preview," May 2026. [Online]. Available: https://hacks.mozilla.org/2026/05/behind-the-scenes-hardening-firefox/

[3] H. Ruan, Y. Zhang, and A. Roychoudhury, "SpecRover: Code intent extraction via LLMs," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025.

[4] L. Ma, S. Liu, Y. Li, X. Xie, and L. Bu, "Specgen: Automated generation of formal program specifications via large language models," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 16–28.

[5] B. S. Team, "From Naptime to Big Sleep: Using large language models to catch vulnerabilities in real-world code," 2024. [Online]. Available: https://projectzero.google/2024/10/from-naptime-to-big-sleep.html

[6] Anthropic. (2025) Claude Code: An agentic coding tool. https://claude.com/product/claude-code.

[7] DARPA. (2025) AI Cyber Challenge (AIxCC). https://aicyberchallenge.com/.

[8] Google. (2021) OSV: Open source vulnerabilities database and triage service. https://github.com/google/osv.dev.

[9] Team Atlanta. (2025) Atlantis: Team atlanta's cyber reasoning system for the DARPA AIxCC final competition. https://github.com/Team-Atlanta/aixcc-afc-atlantis.

[10] The GPSd Project, "GPSd: Put your GPS on the net!" 2026. [Online]. Available: https://gpsd.io/

[11] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," *2012 USENIX Annual Technical Conference*, 2012.

[12] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[13] LLVM Project, "libFuzzer: A library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html.

[14] Code Intelligence, "Jazzer: Coverage-guided, in-process fuzzing for the JVM," https://github.com/CodeIntelligenceTesting/jazzer.

[15] earendil-works. (2026) Pi: An AI agent toolkit. https://github.com/earendil-works/pi/releases/tag/v0.77.0.

[16] T. Kim, H. Han, S. Park, D. R. Jeong, D. Kim, D. Kim, E. Kim, J. Kim, J. Wang, K. Kim *et al.*, "ATLANTIS: AI-driven threat localization, analysis, and triage intelligence system," *arXiv preprint arXiv:2509.14589*, 2025.

[17] C. Zhang, Y. Park, F. Fleischer, Y.-F. Fu, J. Kim, D. Kim, Y. Kim, Q. Xu, A. Chin, Z. Sheng *et al.*, "Sok: Darpa's ai cyber challenge (aixcc): Competition design, architectures, and lessons learned," *Usenix Security*, 2026.

[18] A. Chin, D. Kim, Y.-F. Fu, F. Fleischer, Y. Kim, H. Han, C. Zhang, B. J. Lee, H. Zhao, and T. Kim, "OSS-CRS: Liberating AIxCC cyber reasoning systems for real-world open-source security," *arXiv preprint arXiv:2603.08566*, 2026.

[19] DARPA. (2024) AIxCC competition: Procedures and scoring guide. https://aicyberchallenge.com/wp-content/uploads/2024/06/ASC-Procedures-and-Scoring-Guide-v4.pdf.

[20] Anthropic. (2026) Claude Sonnet 4.6. https://docs.anthropic.com/en/docs/about-claude/models/overview.

[21] DeepSeek-AI. (2026) DeepSeek V4 Pro. https://api-docs.deepseek.com.

[22] Team Atlanta. (2026) Claude Code bug-finding agent (`crs-bug-finding-claude-code`). https://github.com/Team-Atlanta/crs-bug-finding-claude-code.

[23] K. Serebryany, "OSS-Fuzz: Google's continuous fuzzing for open-source software." Vancouver, BC: USENIX Association, Aug 2017.

[24] OSV. (2026) OSV-2026-189: Out-of-bounds read in gpsd. https://osv.dev/vulnerability/OSV-2026-189.

[25] Google, "OSS-Fuzz-Gen: LLM powered fuzzing via OSS-Fuzz," 2024. [Online]. Available: https://github.com/google/oss-fuzz-gen

[26] Y. Lyu, Y. Xie, P. Chen, and H. Chen, "Prompt fuzzing for fuzz driver generation," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3793–3807.

[27] Y. Liu, J. Deng, X. Jia, Y. Wang, M. Wang, L. Huang, T. Wei, and P. Su, "Promefuzz: A knowledge-driven approach to fuzzing harness generation with large language models," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 1559–1573.

[28] Z. Luo, Q. Du, Y. Wang, A. Roychoudhury, and Y. Jiang, "Enhancing protocol fuzzing via diverse seed corpus generation," *IEEE Transactions on Software Engineering*, 2025.

[29] J. Ji, C. Zhang, S. Gan, L. Jian, H. Liu, T. Liu, L. Zheng, and Z. Jia, "Firmagent: Leveraging fuzzing to assist llm agents with iot firmware vulnerability discovery." in *NDSS*, 2026.

[30] Z. Sheng, Q. Xu, J. Huang, M. Woodcock, H. Huang, A. F. Donaldson, G. Gu, and J. Huang, "All you need is a Fuzzing Brain: An LLM-powered system for automated vulnerability detection and patching," *arXiv preprint arXiv:2509.07225*, 2025.

[31] D. Wolff, M. Mirchev, and A. Roychoudhury, "Large language models in software security analysis," *Communications of the ACM*, vol. 69, no. 6, pp. 60–67, 2026.

[32] Z. Luo, H. Zhao, D. Wolff, C. Cadar, and A. Roychoudhury, "Agentic concolic execution," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2026, pp. 1–19.

[33] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4All: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[34] X. Du, G. Zheng, K. Wang, Y. Zou, Y. Wang, W. Deng, J. Feng, M. Liu, B. Chen, X. Peng *et al.*, "Vul-RAG: Enhancing LLM-based vulnerability detection via knowledge-level RAG," *ACM Transactions on Software Engineering and Methodology*, 2024.

[35] J. Park and I. Yun, "Agentic fuzzing: Opportunities and challenges," *arXiv preprint arXiv:2605.10074*, 2026.

[36] Z. Wang, G. Li, J. Li, H. Zhu, and Z. Jin, "VulAgent: Hypothesis-validation based multi-agent vulnerability detection," *arXiv preprint arXiv:2509.11523*, 2025.

[37] Z. Wei, J. Sun, Y. Sun, Y. Liu, D. Wu, Z. Zhang, X. Zhang, M. Li, Y. Liu, C. Li, M. Wan, J. Dong, and L. Zhu, "Advanced smart contract vulnerability detection via LLM-powered multi-agent systems," *IEEE Transactions on Software Engineering*, vol. 51, no. 10, pp. 2830–2846, 2025.

[38] S. Hu, T. Huang, F. İlhan, S. F. Tekin, and L. Liu, "Large language model-powered smart contract vulnerability detection: New perspectives," in *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 2023, pp. 297–306.

[39] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[40] C. Lemieux, D. Park, and I. Beschastnikh, "General ltl specification mining (t)," in *2015 30th IEEE/ACM international conference on automated software engineering (ASE)*. IEEE, 2015, pp. 81–92.

[41] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2332–2354, 2024.

[42] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri, "Can large language models transform natural language intent into formal method postconditions?" *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1889–1912, 2024.

[43] S. K. Lahirie, "Evaluating llm-driven user-intent formalization for verification-aware languages," in *2024 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2024, pp. 142–147.

[44] Q. Zhang, Y. Shen, J. Liu, Y. Xu, H. Shi, Y. Jiang, and W. Chang, "ECG: Augmenting embedded operating system fuzzing via LLM-based corpus generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 4238–4249, 2024.

[45] H. Huang, A. Zhou, M. Payer, and C. Zhang, "Everything is good for something: Counterexample-guided directed fuzzing via likely invariant inference," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1956–1973.

[46] J. Zhu, C. Shen, Z. Li, J. Yu, Y. Chen, and K. Pei, "Locus: Agentic predicate synthesis for directed fuzzing," in *Proceedings of the ACM/IEEE 48th International Conference on Software Engineering*, ser. ICSE '26. New York, NY, USA: Association for Computing Machinery, 2026. [Online]. Available: https://doi.org/10.1145/3744916.3773102

[47] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2829–2846.

[48] H. Ding, Z. Wang, and H. Chen, "FM-Agent: Scaling formal methods to large systems via LLM-based Hoare-style reasoning," *arXiv preprint arXiv:2604.11556*, 2026.

[49] Z. Li, S. Dutta, and M. Naik, "Llm-assisted static analysis for detecting security vulnerabilities," *arXiv preprint arXiv:2405.17238*, 2024.

[50] 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.

[51] M. Roberts, H. Thakur, C. Herlihy, C. White, and S. Dooley, "To the cutoff... and beyond? a longitudinal perspective on llm data contamination," in *The Twelfth International Conference on Learning Representations*, 2023.

[52] C. Deng, Y. Zhao, X. Tang, M. Gerstein, and A. Cohan, "Benchmark probing: Investigating data leakage in large language models," in *NeurIPS 2023 Workshop on Backdoors in Deep Learning - The Good, the Bad, and the Ugly*, 2024. [Online]. Available: https://openreview.net/forum?id=a34bgvner1

[53] Anthropic, "ANT-2026-6615Y595: wolfSSL vulnerability finding," 2026. [Online]. Available: https://red.anthropic.com/2026/cvd/findings/ANT-2026-6615Y595

[54] H. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013.