SPECA: SPECIFICATION-TO-CHECKLIST AGENTIC AUDITING FOR MULTI-IMPLEMENTATION SYSTEMS — A CASE STUDY ON ETHEREUM CLIENTS

Masato Kamba[1], **Akiyoshi Sannai**[2,1,3]
[1]Nyx Foundation, Tokyo, Japan, [2]Kyoto University, Kyoto, Japan,
[3]Research and Development Center for Large Language Models, National Institute of Informatics, Tokyo, Japan
masato.kamba@nyx.foundation, sannai.akiyoshi.7z@kyoto-u.ac.jp

ABSTRACT

Multi-implementation systems (e.g., protocols with independent clients) are increasingly audited against natural-language specifications. Differential testing scales well when implementations disagree, but it provides little signal when all implementations converge on the same incorrect interpretation of an ambiguous requirement. We present SPECA, a **SPE**cification-to-**C**hecklist **A**uditing framework that turns normative requirements into property-based checklists, maps them to implementation locations, and supports cross-implementation reuse.

We instantiate SPECA in an in-the-wild security audit contest for the Ethereum Fusaka upgrade, covering 11 production clients. Across 54 submissions, 17 were judged valid by the contest organizers. Cross-implementation checks account for 76.5% (13/17) of valid findings, suggesting that checklist-derived 1→N reuse is a practical scaling mechanism in multi-implementation audits. To understand false positives, we manually coded the 37 invalid submissions and find that threat model misalignment explains 56.8% (21/37): reports that rely on assumptions about trust boundaries or scope that contradict the audit's rules. We also characterize key gaps: we detected no High/Medium findings in the V1 deployment, with misses concentrated in specification details and implicit assumptions (57.1%), timing and concurrency issues (28.6%), and external library dependencies (14.3%). Our improved agent, evaluated against the ground truth of a competitive audit, achieved a strict recall of 27.3% on high-impact vulnerabilities. This performance places the agent in the top 4% of human auditors, outperforming 49 out of 51 contestants in detecting critical issues. While this evidence comes from a single deployment, it highlights a general systems-security lesson: formalizing threat assumptions early is critical for reducing false positives and focusing agentic auditing effort. Our approach demonstrates significant efficiency gains; the agent-driven process distills the audit, allowing human experts to validate findings and finalize a submission in just 40 minutes on average.

1 INTRODUCTION

Natural-language specifications increasingly define the security-critical behavior of complex systems, from network protocols and distributed ledgers to databases and runtimes. In these ecosystems, independent implementations evolve quickly and are audited under tight time and cost constraints. A persistent bottleneck is translating ambiguous textual requirements into executable checks and then verifying those checks across multiple implementations. Traditional automation (tests, static analysis, fuzzing) can validate particular code paths, but the spec-to-code mapping remains expensive and error-prone.

This bottleneck is especially acute in multi-implementation settings. Differential testing and cross-client comparisons scale well when implementations disagree, but they provide little signal when all implementations converge on the same incorrect interpretation of an ambiguous requirement. We refer to this failure mode as a *semantic blind spot*: when correlated misinterpretations cause implementations to converge on the same (incorrect) reading of an ambiguous requirement, leaving

divergence-based signals weak or absent. Addressing semantic blind spots is a general systems-security problem: audit effort can be consumed by low-value reports unless the threat model (trust boundaries and scope) is made explicit early.

We argue that specification checklisting provides a stable and traceable foundation for security auditing, enabling agents to map normative requirements to concrete code checks and reuse those checks across implementations. Cross-implementation checks are one strategy that benefits from this structure: once a checklist item fails in one implementation, the same item can be evaluated elsewhere, catching shared misinterpretations that differential testing would miss.

We implement this insight in SPECA, a **SPE**cification-to-**C**hecklist **A**uditing framework that decomposes the audit process into reproducible phases with inspectable artifacts. We evaluate SPECA in a real-world multi-implementation audit contest (the Ethereum Fusaka upgrade), and in this deployment Strategy B (cross-implementation checks) represents 76.5% of all valid findings (13 of 17 issues). Our analysis also finds that 56.8% of invalid submissions are explained by threat model misalignment—mismatches between assumed attacker capabilities or scope and the contest rules—suggesting that improvement efforts should focus on formalizing assumptions rather than solely increasing model capability. Our central message is that checklisting the specification and reusing checks across implementations can reduce semantic blind spots caused by correlated misinterpretations—but only when the threat model (trust boundaries and scope) is made explicit.

We re-evaluated the improved V2 agent against the Sherlock "Cant-be-evil" ground truth: on consensus-layer issues rated Low-severity or higher, it achieved 27.3% strict recall (3/11), found two High-severity issues missed by V1, and surpassed 49 of 51 auditors (96

Existing approaches to detecting specification-implementation misalignment fall into two categories: differential testing and formal verification. Differential testing tools like Fluffy (Yang et al., 2021) and LOKI (Ma et al., 2023) have proven effective at finding discrepancies between implementations. However, these techniques fundamentally rely on behavioral divergence as their oracle: when all implementations agree on incorrect behavior, the bug remains invisible. Formal verification, while capable of proving correctness against specifications, requires substantial upfront investment in formalizing natural-language specifications into machine-checkable formats—a cost that few projects can bear, particularly as specifications evolve rapidly. What is needed is an approach that can systematically audit implementations against natural-language specifications at scale, without requiring full formalization.

In summary, this paper makes the following contributions:

- *SPECA: an artifact-centric framework for multi-implementation security audits.* We present a specification-checklisting approach that decomposes auditing into traceable artifacts—normative requirement extraction, specification-to-implementation mapping, property-based checklists, and audit reports with proof traces—and describe how these artifacts support systematic auditing (§3).
- *Cross-implementation checklist reuse as a scalable audit mechanism.* We operationalize "1→N reuse" as a first-class strategy: once a checklist item is validated (or a defect is found) in one implementation, analogous checks can be propagated to other implementations to scale coverage (§3–§4).
- *In-the-wild case study (Ethereum Fusaka audit contest).* We instantiate the framework in a real-world security audit contest for the Ethereum Fusaka upgrade (11 production clients). In this deployment, cross-implementation checks account for 76.5% of valid findings (13/17), illustrating practical scaling benefits under real operational constraints (§4).
- *Failure modes and a systems-security design lesson.* Manual coding of invalid reports shows that threat model misalignment explains 56.8% (21/37) of false positives—assumptions about trust boundaries or scope that contradict the audit's rules—highlighting that formalizing threat assumptions early is critical for reducing noise and focusing agentic audit effort. Missed High/Medium issues cluster in specification details and implicit assumptions (57.1%), timing and concurrency issues (28.6%), and external library dependencies (14.3%), providing concrete targets for future enhancement (§4).
- *Demonstration of Human Effort Reduction.* We quantify the manual effort involved, showing that our agent-driven pipeline significantly reduces the burden on human experts. The

process allows a specialist to validate a core finding in approximately 10 minutes and refine a full PoC and report in another 30, demonstrating a substantial reduction in the manual effort required for complex audits.

2 Related Work

We organize related work as three layers of a systems-security stack. (1) **Audit automation**: classic program-analysis and security-testing techniques (static analysis, fuzzing, symbolic execution, model checking) that provide scalable but often specification-agnostic signals. (2) **Divergence-based oracles in multi-implementation systems**: differential testing and differential fuzzing, which can scale across independent implementations but can become ineffective under *correlated misinterpretations* of ambiguous requirements. (3) **LLM/agentic auditing**: approaches that use LLMs (often as tool-using agents) to align natural-language specifications with code and to generate structured audit artifacts. SPECA sits at the intersection of these layers: it reuses the scalability of multi-implementation settings while reducing reliance on divergence by anchoring checks in specification-derived artifacts, and it is designed to compose with classical analysis tools.

2.1 Multi-Implementation Differential Testing

Differential testing uses behavioral divergence as a bug oracle. For instance, Fluffy (Yang et al., 2021) applied this to Ethereum consensus, surfacing bugs via multi-transaction fuzzing. LOKI (Ma et al., 2023) extended this approach with state-aware fuzzing for consensus protocols. More recently, FORKY (Kim et al., 2025) introduced fork-state-aware differential fuzzing to specifically target fork-handling processes.

While highly effective at finding implementation divergences, these approaches share a fundamental limitation: differential testing cannot detect bugs when *all implementations share the same misinterpretation* of the specification. Knight and Leveson's study (Knight & Leveson, 1986) on N-version programming demonstrated that independently developed implementations often exhibit correlated failures rooted in specification ambiguity. Our work aims to mitigate this *semantic blind spot* by grounding the audit in specification requirements rather than implementation consensus, thereby reducing dependence on divergence-based oracles that fail under correlated misinterpretations. Unlike differential fuzzing approaches that rely on behavioral divergence as an oracle, SPECA anchors correctness in the specification itself, enabling detection even under correlated misinterpretations.

2.2 Specification-to-Implementation Alignment

RFCAudit (Zheng et al., 2025) is most closely related to our approach, proposing an agentic framework with separate indexing and detection agents to audit protocol implementations against RFC specifications. However, RFCAudit focuses on single-implementation audits, whereas SPECA targets multi-implementation environments where cross-implementation checks are evaluated as one of the strategies in our setting.

AUTOSPEC (Liu et al., 2025) tackles a complementary problem: synthesizing precise protocol specifications from natural language. Its emphasis on generating inspectable, traceable intermediate representations aligns with our artifact-centric design philosophy. Recent work on RFC-TCP/IP gap detection (Wu et al., 2025) demonstrates LLM-facilitated differential analysis between specifications and implementations, validating the feasibility of LLM-based specification understanding at scale. Unlike prior work that audits a single implementation or focuses on spec synthesis, SPECA targets multi-implementation security auditing and checklist reuse across clients.

2.3 Autonomous Code Auditing Agents

RepoAudit (Guo et al., 2025) introduces agent memory, path-sensitive data-flow extraction, and validators for satisfiability checking. RepoAudit's validator-centric approach to reducing false positives differs from our emphasis on *threat model formalization*: we find that the majority of false positives stem from mismatches between assumed and actual trust boundaries.

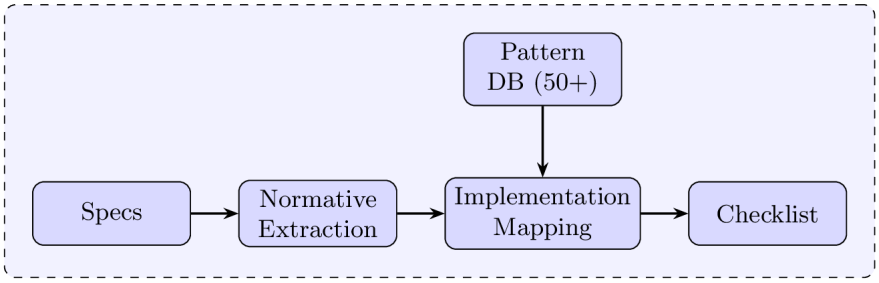

Figure 1: Fig A: Checklist construction. Phase 1 extracts structured knowledge from specifications and produces a reusable checklist.

Multi-agent architectures have shown promise for vulnerability detection. VulAgent (Wang et al., 2025) employs hypothesis-validation with specialized agents. MAVUL (Li et al., 2025a) uses cross-role interaction among analyst, architect, and evaluator agents. While these approaches focus on deepening code understanding through agent collaboration, our work emphasizes *specification as the source of truth* for what constitutes correct behavior. Unlike validator-centric or collaboration-centric agents, SPECA treats specification checklists and explicit threat model formalization as the primary levers for reducing false positives and scaling audits.

Neuro-symbolic and benchmarked approaches offer complementary angles. IRIS (Li et al., 2025b) and QLPro (Hu et al., 2025) couple LLM inference with static analysis (e.g., CodeQL) to synthesize checkers for CWE classes, while Agent Security Bench (Zhang et al., 2025) and AgentDojo (Debenedetti et al., 2024) provide standardized evaluation suites for adversarial robustness. These works start from vulnerability-class taxonomies or benchmark scenarios; SPECA instead anchors on *normative specification requirements* in multi-implementation settings, aiming to catch correlated misinterpretations that evade divergence-based oracles.

3 System Design

This section presents SPECA, our specification-checklist-driven agentic auditing framework. We provide a high-level overview (§3.1), formalize the problem and threat model (§3.2), detail the two main phases (§3.3–§3.4), describe artifact generation (§3.5), present improvements based on failure analysis (§3.6), and cover implementation details (§3.7).

3.1 Overview

SPECA automates the expert security auditor's workflow through a two-phase pipeline: (1) **knowledge structuring**, which extracts and formalizes requirements from natural-language specifications, and (2) **systematic auditing**, which applies three complementary strategies to detect specification violations across multiple implementations. Figures 1 and 2 illustrate the checklist construction and cross-implementation auditing flow.

SPECA works by first building a structured representation of what the code *should* do (from specifications), then systematically checking whether the code *actually* does it (through implementation analysis). This checklist-driven approach helps mitigate semantic blind spots arising from correlated misinterpretations: because the checklist is anchored in the specification, it can flag shared misinterpretations that would not produce behavioral divergence.

Although cross-implementation checks are conceptually simple, our evaluation (§4) shows that they account for the majority of valid findings in practice, justifying their treatment as a first-class strategy.

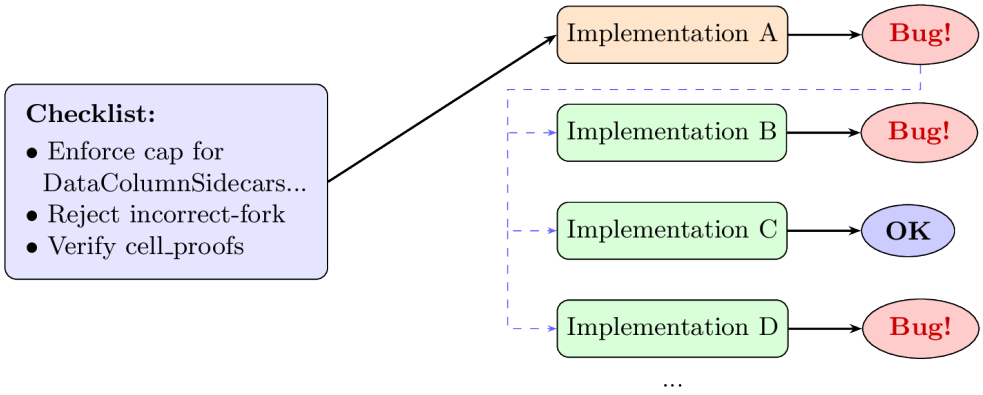

Figure 2: Fig B: Strategy B cross-implementation checks. A checklist item identified on Client A is propagated across other clients, accounting for 76.5% of valid findings in our deployment (§4).

3.2 Problem Formalization and Threat Model

Problem Definition. Given a natural-language specification S consisting of normative requirements $\{r_1, r_2, ..., r_n\}$ and a set of implementations $\mathcal{I} = \{I_1, I_2, ..., I_k\}$ that claim conformance to S, the specification-checklist-based security auditing problem is to identify tuples (r_i, I_j, v) where implementation I_j violates requirement r_i in manner v.

This formulation differs from differential testing, which identifies divergences (I_j, I_l, x) where implementations produce different outputs on input x. Crucially, differential testing cannot detect violations when all implementations exhibit the same incorrect behavior.

Threat Model. We target vulnerabilities in protocol implementations that could lead to consensus failures, denial of service, or information disclosure. The specification documents are assumed correct (we detect implementation bugs, not specification bugs). Human verification is required for all findings. The threat model is treated as an explicit design parameter rather than an implicit assumption, as mismatches in assumed trust boundaries later emerge as the dominant source of false positives (§4.4).

3.3 Phase 1: Knowledge Structuring

Phase 1 transforms unstructured specification documents into a queryable knowledge base.

Specification Extraction. The first component extracts normative requirements from specification documents. Given a specification document (e.g., an EIP markdown file), the extractor identifies statements containing modal verbs (MUST, SHOULD, MAY, SHALL) following RFC 2119 conventions and outputs structured requirements with unique traceability identifiers.

We use LLM-based extraction because specification documents use inconsistent formatting and natural language that resists rigid pattern matching. Each extracted requirement includes a unique ID (e.g., `EIP7594-CUSTODY-001`) and source location, enabling auditors to trace findings back to the original specification text.

Pattern Database Construction. The second component curates a database of known vulnerability patterns (50+ entries) relevant to the target domain, including boundary validation failures, guard condition omissions, state transition errors, cryptographic misuse, and resource management issues.

Implementation Mapping. The third component establishes correspondence between normative requirements and source code locations using a two-phase approach: keyword search to prune the search space, followed by semantic refinement using the LLM to ensure meaningful requirement-code correspondence.

Figure 3: SPECA V2 Agent Overview.

3.4 Phase 2: Systematic Auditing

Phase 2 applies three complementary strategies to detect specification violations.

Strategy A: Specification-Based Static Audit. This strategy directly checks whether each implementation satisfies the mapped normative requirements, examining presence (is there code addressing this requirement?), correctness (does the code correctly implement the requirement?), and completeness (are all conditions and edge cases handled?).

Strategy B: Cross-Implementation Checks. This is one of the strategies in SPECA. Strategies A and B share the same checklist derived from the specification; B evaluates the same checklist item across other implementations once a failure is observed.

The algorithm works as follows: (1) abstract a generalizable pattern from the initial finding, (2) find analogous locations in all other target implementations, (3) check each location for pattern matches, and (4) report cross-implementation findings for human verification.

Why this strategy is effective in multi-implementation settings: Violations often cluster around complex or ambiguous specification requirements. When one implementation team misinterprets a requirement, other teams—facing the same text—may make similar errors. Cross-implementation checks exploit this correlation structure by turning one finding into a systematic sweep across implementations. Representative examples are provided in Appendix Table 6.

Strategy C: Dynamic Test Generation. This strategy generates executable tests for boundary conditions. However, dynamic testing proved least effective in our evaluation (5.9% of valid findings) because setting up test environments requires significant effort, and semantic blind spots can also reduce the usefulness of dynamic testing when the test oracle implicitly inherits the same correlated misinterpretations.

3.5 Artifact Generation

Figure 3 provides a concise overview of the SPECA V2 agent flow from specifications to audit outcomes.

3.6 Improvements Based on Failure Analysis (V2)

Analysis of false positives and missed findings in V1 revealed two primary areas for improvement:

1. **Threat Model Misalignment** (56.8% of false positives): The agent's assumptions about attacker capabilities did not match the actual system context.
2. **Missed High-Severity Issues** (7 issues): Analysis **provides a clear roadmap for enhancement**: specification details and implicit assumptions (57.1%), timing and concurrency issues (28.6%), and external library dependencies (14.3%).

Based on this analysis, V2 introduces **TRUSTMODEL Formalization**: before auditing, V2 explicitly formalizes the threat model as a structured artifact specifying actor trust levels (TRUSTED, SEMI_TRUSTED, UNTRUSTED) and boundary edges. This enables filtering findings that require capabilities outside the defined threat model.

To provide initial empirical evidence for V2, we conduct a post-hoc re-evaluation against the contest's ground-truth issue database. We re-run the V2 pipeline on the six CL clients and measure strict recall—counting only findings that identify the same bug class, code region, and root cause as a contest-validated issue. The results (Appendix B) show that V2 achieves a strict issue recall of 27.3% (3/11) on CL-related Low/Medium/High issues, successfully detecting 2 of 3 High-severity

issues while missing issues that require deep stateful reasoning or client-specific implementation knowledge.

3.7 IMPLEMENTATION

We implement SPECA with a fixed LLM and custom prompts for each pipeline phase. We retain structured artifacts (specifications, maps, checklists, and reports) to support reproducibility and human verification.

4 EVALUATION

We evaluate SPECA as an in-the-wild case study of a broader systems-security problem: auditing multiple independent implementations of the same specification under tight operational constraints. We deploy SPECA in a production security audit contest for the Ethereum Fusaka upgrade (11 clients) and measure (i) how each audit strategy contributes to validated findings, (ii) which factors dominate false positives in practice, and (iii) which vulnerability classes remain challenging for a checklist-driven agentic approach.

4.1 RESEARCH QUESTIONS

RQ1: Which audit strategy contributes most to valid vulnerability discoveries in a multi-implementation environment?

RQ2: What are the primary root causes of false positives in LLM-assisted security auditing?

RQ3: What types of vulnerabilities does specification-checklist-driven agentic auditing fail to detect?

4.2 EXPERIMENTAL SETUP

We evaluate SPECA in the Fusaka audit contest, a real-world multi-implementation security audit for the Ethereum Fusaka upgrade (Osaka EL + Fulu CL).

The contest covered 11 production clients and produced 366 total submissions from all participants, of which 101 were valid (27.6% overall valid rate; full parameters in Appendix 7).

Target Implementations. The contest covered 11 production clients: Nimbus, Lighthouse, Prysm, Teku, Lodestar, Grandine (consensus layer); Geth, Reth, Erigon, Nethermind, Besu (execution layer).

Scope and Severity. The contest scope is limited to Fusaka-specific code; known issues and out-of-scope findings are invalid. Non-informational findings must include an executable PoC at submission time. Severity is defined by network-impact thresholds (Critical $>50\%$, High $>33\%$, Medium $>5\%$, Low $>0.01\%$), and Informational issues are valid only if the client accepts and implements the change. These thresholds make Low/Informational findings operationally meaningful despite their label.

Attribution and Coding. We attribute each valid report to its primary discovery strategy as recorded in the submission: similar-issue-based analysis corresponds to cross-implementation checks, code analysis corresponds to specification-based static audit, and fuzz testing corresponds to dynamic test generation. We manually code invalid submissions into mutually exclusive categories and report the primary cause.

Human Effort. To quantify the manual effort involved in our process, we recorded the time spent on human-in-the-loop verification. On average, each generated report underwent an initial review of approximately 10 minutes to validate the core finding. For findings that proceeded to the next stage, an additional 30 minutes were spent on average to review, refine, and finalize the proof-of-concept (PoC) and the full submission report. This highlights the essential role of human expertise in validating and contextualizing the agent's outputs.

Table 1: Overall submission outcomes. Our 31.5% valid rate exceeds the contest average of 27.6%.

Metric	Count	Percentage
Total submissions	54	100%
Valid submissions	17	31.5%
Invalid submissions	37	68.5%

Table 2: Strategy attribution for valid findings. Cross-implementation checks represent 76.5% of valid discoveries.

Strategy	Findings	%
Cross-Implementation Checks	13	76.5%
Specification-based static audit	3	17.6%
Dynamic test generation	1	5.9%
Total	**17**	**100%**

Table 3: Severity distribution of valid findings.

Severity	High	Med	Low	Info
Count	0	0	1	16

4.3 Results: RQ1 – Strategy Contribution Analysis

While most valid findings are classified as Informational under contest rules, such findings are operationally meaningful because they require client adoption to be considered valid (§4).

Client Coverage: 9 of 11 clients (81.8%)

Our approach achieved a 31.5% valid rate (17/54), exceeding the contest average of 27.6% (101/366). The 81.8% client coverage indicates broad applicability across diverse codebases; the per-client breakdown is in Appendix Table 8. However, all 17 valid findings from the V1 contest deployment were Low or Informational severity. We discovered none of the contest's 5 High-severity or 2 Medium-severity vulnerabilities in V1. This outcome distinguishes between two classes of challenges: issues rooted in implicit specification details, which are addressable by refining the SPECA methodology, and those stemming from complex stateful interactions, which highlight the current limitations of LLM agents. The analysis in Section 4.5 further details this distinction.

Cross-implementation checks contributed 76.5% of valid findings in this deployment. This is consistent with the hypothesis that specification misinterpretations correlate across implementations: when one team misunderstands a requirement, others may make similar errors.

Summary for RQ1. In this deployment, cross-implementation checks represent the highest observed share of valid findings among the three strategies (76.5%).

4.4 Results: RQ2 – False Positive Analysis

We define categories as follows: threat model misalignment (assumptions about trust boundaries or scope that contradict contest rules), already known / duplicate (publicly known or previously reported issues), incorrect analysis (reasoning errors or misread code paths), and out of scope (findings not specific to the Fusaka upgrade).

The largest category (56.8%) was threat model misalignment. Our prompts assumed execution layer (EL) responses were untrusted, while the contest rules treated EL as a trusted component. Under our coding scheme, this mismatch is associated with 21 invalid submissions.

Summary for RQ2. Threat model misalignment is the largest observed false-positive category (56.8%), suggesting that assumption formalization is a high-leverage improvement target.

4.5 Results: RQ3 – Limitation Characterization

Notably, SPECA did not detect any High or Medium severity issues in this deployment. Our analysis of these misses distinguishes between categories that are addressable through methodological improvements and those that represent fundamental limitations of current-generation LLM agents.

Table 4: False positive root cause distribution.

Root Cause	Cnt	%
Threat model misalignment	21	56.8%
Already known / duplicate	8	21.6%
Incorrect analysis	5	13.5%
Out of scope	3	8.1%
Total	**37**	**100%**

Table 5: Miss category summary for High/Medium issues.

Failure Category	Cnt	%
Spec details & implicit assumptions	4	57.1%
Timing and concurrency issues	2	28.6%
External library dependencies	1	14.3%

We found none of the 7 High/Medium issues in the contest (Appendix Table 9). The misses fall into three categories (Appendix Table 10 provides full detail).

Specification details and implicit assumptions (57.1%—Addressable): The majority of missed issues (4/7) were due to subtle details in the specification that were overlooked, such as the exact fields to include in a cache key or nuanced input validation requirements. These gaps can be systematically narrowed by creating more granular checklists.

Timing and concurrency issues (28.6%—Current LLM Limitation): Two issues were related to dynamic behavior that is difficult to capture with static analysis, such as metadata updates during fork transitions. These vulnerabilities depend on complex, state-dependent interactions that are difficult to reason about without a deep, stateful understanding of the system's runtime behavior, representing the frontier of current LLM agent capabilities.

External library dependencies (14.3%—Addressable): One issue stemmed from a vulnerability in the 'c-kzg-4844' library. This category is addressable by integrating automated tools like supply chain scanners into the audit pipeline.

Summary for RQ3. The analysis of missed High/Medium issues provides a critical distinction between improvable aspects of our methodology and the inherent limitations of today's LLM agents. While issues stemming from implicit requirements and external dependencies (71.4%) can be addressed by enhancing checklist granularity and tool integration, complex timing and concurrency vulnerabilities (28.6%) remain a fundamental challenge, highlighting a key area for future research in agentic reasoning and stateful analysis.

4.6 Discussion

The Modern Landscape of Semantic Blind Spots. Our findings offer a contemporary perspective on the nature of semantic blind spots in an era dominated by agentic coding. While classic correlated failures stemmed from ambiguous specification text, we hypothesize that the widespread use of a few powerful LLMs for development assistance introduces a new vector for shared vulnerabilities. If different teams leverage agents with similar underlying reasoning patterns, they may independently introduce homologous bugs, creating a deeper, more uniform blind spot that is even harder to detect with divergence-based methods.

This hypothesis provides a compelling explanation for our primary finding: **cross-implementation checks represent 76.5% of valid findings**. In a world where agent-assisted development is common, a bug pattern found in one implementation is highly likely to exist in others, not just due to spec misinterpretation, but due to the convergent "thinking" of the agents themselves. The remarkable success of our 1→N reuse strategy may therefore be a direct reflection of this modern reality, making it an increasingly critical tool for security auditing in the wild.

Our other findings are also interpreted through this lens. (1) Threat model misalignment remains the largest observed FP category (56.8%); and (2) High/Medium issues were not detected in the V1 contest deployment (0% detection rate), identifying concrete targets for future enhancement.

Preliminary V2 Re-evaluation. To assess whether V2's threat model formalization translates into improved detection capability, we conducted a post-hoc re-evaluation against the contest's ground-truth issue database (details in Appendix B). Filtering to CL-client issues at Low severity or above (11 issues), V2 achieves a strict recall of 27.3% (3/11), detecting 2 of 3 High-severity issues—including the proposer lookahead miscalculation and the c-kzg-4844 Fiat-Shamir weakness—and

1 of 6 Low-severity issues. The 8 undetected issues predominantly require deep stateful reasoning (e.g., fork-transition metadata staleness) or client-specific implementation knowledge (e.g., library-specific point-at-infinity handling), consistent with the limitation categories identified in Section 4.5. While preliminary, these results suggest that V2's structured approach can surface high-impact vulnerabilities that V1 missed entirely, though significant gaps remain for issues requiring dynamic or implementation-specific analysis.

Threats to Validity. *Internal*: Findings may reflect individual auditor skill and the subjective coding of invalid submissions. *External*: Results are specific to Ethereum protocol implementations; generalization requires further study. *Construct*: We rely on contest organizers' severity judgments as ground truth.

5 Conclusion

5.1 Summary of Contributions

We study a general systems-security challenge: scaling audits across multiple independent implementations of the same natural-language specification. In this setting, divergence-based automation (e.g., differential testing) can fail when implementations share the same misinterpretation of ambiguous specification text. We introduced SPECA, an artifact-centric specification-checklisting framework that turns normative requirements into property-based checklists, maps them to code locations, and supports three complementary audit strategies: static checks, cross-implementation checklist reuse, and targeted dynamic tests.

We evaluated the framework in an in-the-wild case study: a production security audit contest for the Ethereum Fusaka upgrade covering 11 clients. In this deployment, cross-implementation checks account for 76.5% of valid findings (13/17), suggesting that checklist-derived "1→N reuse" can be a practical scaling mechanism in multi-implementation audits. We also found that false positives were dominated by threat model misalignment (56.8% of invalid submissions): reports that rely on assumptions about trust boundaries or scope that contradict the audit's rules. Finally, we characterize where the approach fell short in this case study (0% High/Medium detection), identifying concentrated miss categories that motivate concrete improvements to checklist granularity and dynamic analysis integration.

5.2 Limitations

Severity distribution. All 17 valid findings from V1 were Low or Informational severity. A preliminary V2 re-evaluation improves High-severity detection (2/3 on CL clients; Appendix B), but overall strict recall remains modest (3/11), and complex stateful vulnerabilities remain challenging; our analysis (§4.5) highlights checklist granularity, dynamic analysis, and supply-chain signals as concrete directions.

Generalizability. Results come from a single Ethereum contest deployment; replication across other multi-implementation domains is needed to establish robustness.

Replication. We rely on contest organizers' ground truth; additional deployments would strengthen conclusions.

5.3 Future Work

Future work has two priorities: (1) replicate in other multi-implementation domains to test generalizability and threat-model alignment, and (2) integrate complementary techniques (dependency scanning plus fuzzing/model checking/symbolic execution) to address external-library and timing/concurrency misses.

5.4 Closing Remarks

Cross-implementation checks scale human insight across implementations; as agentic coding proliferates, this 1→N transfer becomes increasingly central to security assurance.

REFERENCES

Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL https://openreview.net/forum?id=m1YYAQjO3w.

Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. Repoaudit: An autonomous LLM-agent for repository-level code auditing. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=TXcifVbFpG.

Junze Hu, Xiangyu Jin, Yizhe Zeng, Yuling Liu, Yunpeng Li, Dan Du, Kaiyu Xie, and Hongsong Zhu. Qlpro: Automated code vulnerability discovery via llm and static code analysis integration, 2025. URL https://arxiv.org/abs/2506.23644.

Wonhoi Kim, Hocheol Nam, Muoi Tran, Amin Jalilov, Zhenkai Liang, Sang Kil Cha, and Min Suk Kang. Fork state-aware differential fuzzing for blockchain consensus implementations. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 3048–3059, 2025. doi: 10.1109/ICSE55347.2025.00085.

John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1): 96–109, 1986. doi: 10.1109/TSE.1986.6312924.

Youpeng Li, Kartik Joshi, Xinda Wang, and Eric Wong. Mavul: Multi-agent vulnerability detection via contextual reasoning and interactive refinement, 2025a. URL https://arxiv.org/abs/2510.00317.

Ziyang Li, Saikat Dutta, and Mayur Naik. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=9LdJDU7E91.

Kuangxiangzi Liu, Dhiman Chakraborty, Alexander Liggesmeyer, and Andreas Zeller. Synthesizing precise protocol specs from natural language for effective test generation, 2025. URL https://arxiv.org/abs/2511.17977.

Fuchen Ma, Yuanliang Chen, Meng Ren, Yuanhang Zhou, Yu Jiang, Ting Chen, Huizhong Li, and Jiaguang Sun. LOKI: state-aware fuzzing framework for the implementation of blockchain consensus protocols. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023. URL https://www.ndss-symposium.org/ndss-paper/loki-state-aware-fuzzing-framework-for-the-implementation-of-blockchain-consensus-protocols/.

Ziliang Wang, Ge Li, Jia Li, Hao Zhu, and Zhi Jin. Vulagent: Hypothesis-validation based multi-agent vulnerability detection. 2025. URL https://arxiv.org/abs/2509.11523.

Yifan Wu, Xuewei Feng, Yuxiang Yang, and Ke Xu. Uncovering gaps between rfc updates and tcp/ip implementations: Llm-facilitated differential checks on intermediate representations, 2025. URL https://arxiv.org/abs/2510.24408.

Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 349–365. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/yang.

Hanrong Zhang, Jingyuan Huang, Kai Mei, Yifei Yao, Zhenting Wang, Chenlu Zhan, Hongwei Wang, and Yongfeng Zhang. Agent security bench (ASB): Formalizing and benchmarking attacks and defenses in LLM-based agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=V4y0CpX4hK.

Mingwei Zheng, Chengpeng Wang, Xuwei Liu, Jinyao Guo, Shiwei Feng, and Xiangyu Zhang. Rfcaudit: An llm agent for functional bug detection in network protocols, 2025. URL https://arxiv.org/abs/2506.00714.

.1 Cross-Implementation Check Examples

Table 6: Cross-Implementation Check examples. A single insight yields multiple findings across the implementation ecosystem.

Original Finding	**Pattern**	**Checked In**	**Result**
Client A: missing slot guard	Range guard omission	Clients B, C	Both missing same guard
Client C: batch bounds inconsistency	Array bounds check	Clients A, D	Client A affected
Client B: caps not applied	Upper limit validation	Multiple	3 additional findings

A Evaluation Data

A.1 Contest Parameters

Table 7: Fusaka audit contest parameters.

Parameter	**Value**
Contest duration	4 weeks
Total prize pool	$2,000,000
Target implementations	11 clients
Total participants	Multiple teams
Total submissions (all)	366
Total valid issues (all)	101

A.2 Severity Thresholds

The contest defined severity based on network-wide impact:

- **Critical**: >50% network impact (slashing, consensus failure)
- **High**: >33% network impact
- **Medium**: >5% network impact
- **Low**: >0.01% network impact
- **Informational**: Client-accepted improvements without direct security impact

These thresholds are notably strict; Low and Informational findings still represent operationally meaningful issues.

A.3 Scoring System

Early report multipliers: Week 1 = 2×, Week 2 = 1.5×.

A.4 PER-CLIENT BREAKDOWN (VALID VS. INVALID)

Table 8: Findings by client. SPECA found valid issues in 9 of 11 target clients.

Client	Valid Findings	Invalid Submissions
Nimbus	6	8
Grandine	3	1
Erigon	2	1
Besu	1	5
Lodestar	1	5
Nethermind	1	4
Teku	1	4
Prysm	1	1
Reth	1	1
Lighthouse	0	4
Geth	0	3

Table 9: Miss rate by severity. We found 0% of High/Medium issues.

Severity	Total in Contest	We Found	Missed	Discovery Rate
High	5	0	5	0%
Medium	2	0	2	0%
Low	8	1	7	12.5%

Table 10: Failure category distribution for missed High/Medium issues.

Failure Category	Count	%	Assessment	Improvement Direction
Specification details and implicit assumptions	4	57.1%	Addressable	Finer-grained checklist generation
Timing and concurrency issues	2	28.6%	LLM Limitation	Requires advances in stateful reasoning
External library dependencies	1	14.3%	Addressable	Supply chain scanning

B V2 RE-EVALUATION AGAINST GROUND TRUTH

To empirically assess V2's detection capability, we conducted a post-hoc re-evaluation against the contest's validated issue database. We filtered the 366 contest submissions to the 11 CL-client issues rated Low, Medium, or High severity, and compared them against V2's findings using a strict matching criterion: a match is counted only when the agent's finding identifies the *same bug class, code region, and root cause* as the human-reported issue.

Matching Methodology. Each of the 11 ground-truth issues was compared against all V2 findings from the corresponding client branch. Matching was performed using an LLM-based semantic comparison (GPT-4.1-mini) with the following strict criterion: a finding is considered a direct match only if it describes the same vulnerability mechanism targeting the same code location. Partial overlaps, related-but-distinct findings, and thematically similar reports were explicitly excluded.

Matched Issues. Table 12 lists the three issues that V2 successfully detected.

Unmatched Issues. Table 13 lists the eight issues that V2 did not detect, along with the primary reason for the miss.

The unmatched issues cluster into two categories consistent with the failure analysis in Section 4.5: (1) issues requiring deep stateful reasoning about runtime behavior (fork transitions, caching, sync

Table 11: V2 strict recall on CL-client issues (Low/Medium/High).

Severity	Ground Truth	Matched	Missed	Recall
High	3	2	1	66.7%
Medium	2	0	2	0.0%
Low	6	1	5	16.7%
Total	**11**	**3**	**8**	**27.3%**

Table 12: Issues matched by V2 (strict criterion).

#	Sev.	Issue Title
40	High	Proposer calculation can be incorrect since proposer lookahead is not considered
203	High	Weak Fiat-Shamir in c-kzg-4844 verify_cell_kzg_proof_batch
381	Low	Lodestar accepts and rebroadcasts data column sidecars with invalid signatures

state machines), and (2) issues requiring client-specific implementation knowledge that is not derivable from the specification alone (library internals, client-specific configuration handling). Notably, V2 detected 2 of 3 High-severity issues, suggesting that the specification-checklist approach is effective for vulnerabilities rooted in specification misinterpretation, while remaining limited for implementation-specific bugs.

C Example Program Graph

Figure 4 shows an example prompt used to create program graphs.

```
## Mindset

You are a **Formal Methods Specialist** trained in program graph extraction. Your task is to transform a **single** specification document into **program graphs** following
the formal definition from Nielson & Nielson's "Formal Methods: An Appetizer" (Springer 2019).

> A program graph **PG = (Q, q₀, q₄, Act, E)** consists of:
> - **Q**: a finite set of nodes (program points)
> - **q₀, q₄ ∈ Q**: initial and final nodes
> - **Act**: a set of actions (assignments, tests/guards)
> - **E ⊆ Q × Act × Q**: a finite set of edges

## Scope

This skill processes **one specification URL** per invocation. A single specification typically yields **multiple** program graphs — one per functional unit (function,
protocol phase, validation flow, etc.).

The calling worker is responsible for batching and aggregation.

## Input

The caller provides:
- `url` — the source URL of the specification (always provided)
- `output_dir` — directory where `.mmd` files should be written
- `local_path` *(optional)* — path to a pre-downloaded copy of the specification

## Procedure

1. **Read Specification**: If `local_path` is provided and the file exists, read from it. Otherwise, fetch the content from `url` using `mcp__fetch__fetch`.

2. **Identify Functional Units**: Break down the document into logical units:
   - Function definitions
   - State transition descriptions
   - Protocol phases
   - Validation logic

   Each functional unit becomes one program graph.

3. **Extract Program Graph Components**:

   For each functional unit, identify:

   | Component | What to Extract |
   | :--- | :--- |
   | **Nodes (Q)** | Program points: entry, exit, decision points, intermediate states |
   | **Initial (q₀)** | The starting point of the function/process |
   | **Final (q₄)** | The termination point(s) |
   | **Actions (Act)** | Assignments (`x = expr`), function calls, tests/guards (`x > 0`) |
   | **Edges (E)** | Transitions: `(source_node, action, target_node)` |

4. **Generate Mermaid Diagrams**: For each program graph, write a `.mmd` file to:
   ```
   {output_dir}/{spec_id}/{SG-ID}_{name}.mmd
   ```
   Where `spec_id` is a short identifier derived from the specification (e.g., `EIP-7594`, `fulu-beacon-chain`).

5. **Return Result**: Return the JSON structure described in Output Format below. Do **not** write `index.json` — the calling worker handles aggregation.
```

Figure 4: Program graph creation prompt example.

Table 13: Issues not matched by V2 (strict criterion).

#	Sev.	Issue Title	Miss Category
190	High	Prysm incorrectly caches inclusion proof verification	Stateful caching logic
15	Med	Nimbus: remote DoS via large custody group count	Implementation-specific DoS
216	Med	Nimbus stale metadata after Fulu fork transition	Fork-transition state
48	Low	Lighthouse rust-eth-kzg point-at-infinity handling	External library internals
109	Low	Malicious peer freezes custody rotation	Implementation-specific loop
319	Low	Grandine blob schedule ordering mismatch	Client-specific config
343	Low	No custody peers on partial retry causes sync stall	Sync state machine logic
376	Low	Grandine accepts blocks with forged DA	Return value handling

Figure 5 shows an example program graph (SG-005: `cell_proof_computation`) generated from an EIP specification passage. The graph initializes an extended blob, iterates over `CELLS_PER_EXT_BLOB`, computes a KZG proof per cell, and returns the collected proofs, capturing invariants such as generating exactly one proof per cell index (INV-014/015) including extension cells (INV-016).

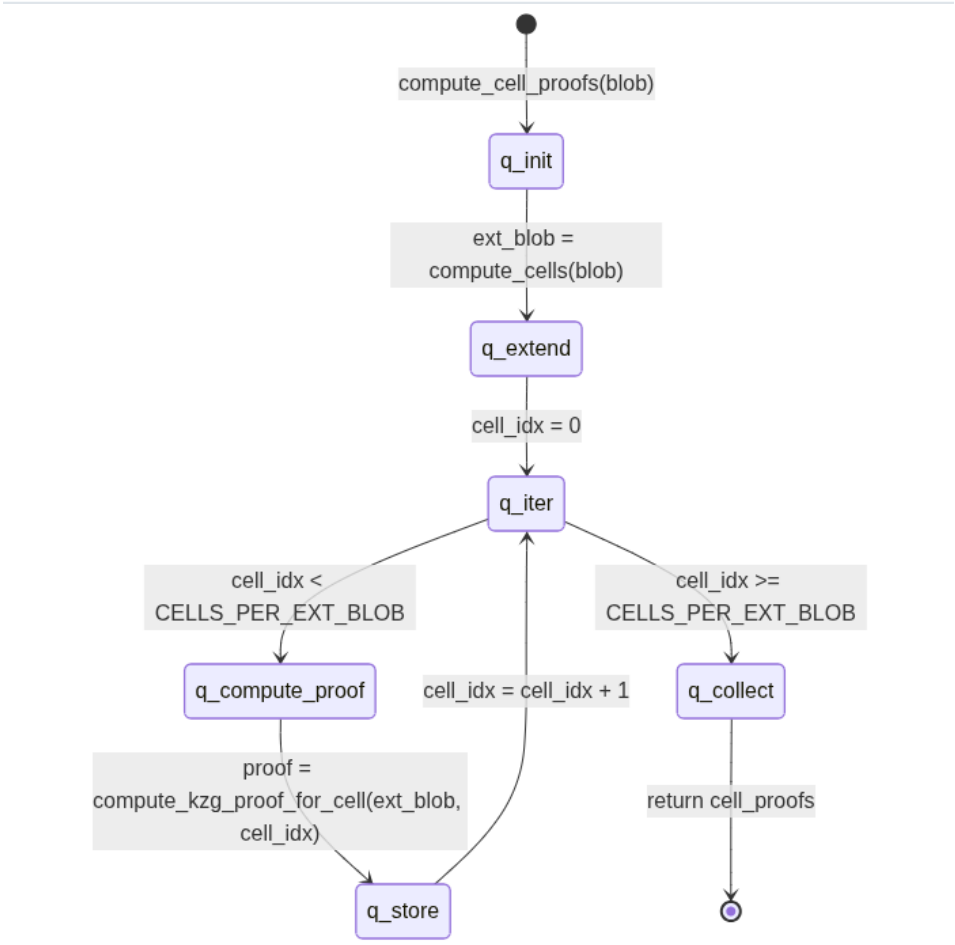

Figure 5: Example program graph generated from an EIP specification (cell proof computation).

D EXAMPLE AUDIT

Figure 6 shows an example prompt used to generate the audit map in V2.

```
<task>
  <goal>For each item in the batch, use MCP tools to resolve the code scope and then invoke the /formal-audit skill to perform a three-stage formal audit.</goal>
  <input type="file" id="queue">{{QUEUE_FILE}}</input>
  <output type="file" id="results">{{OUTPUT_FILE}}</output>

  <critical_requirements>
    **YOU MUST COMPLETE ALL OF THE FOLLOWING:**
    1. Process ALL items in the batch (up to BATCH_SIZE)
    2. After processing ALL items, write a JSON file to <ref id="results"/>
    3. The JSON file MUST be written even if some items are skipped

    **FAILURE TO WRITE THE JSON FILE IS A CRITICAL ERROR.**
  </critical_requirements>

  <instructions>
    1. **Initialize**: Read <ref id="queue"/> and select the first BATCH_SIZE unprocessed items. Create an empty array `results = []`.

    2. **Process Each Item**: For each item in the batch, perform steps 2a-2e:

       2a. **Resolve Code Scope**: Use Tree-sitter MCP tools (`mcp__tree_sitter__get_symbols` or `mcp__tree_sitter__run_query`) to find the relevant file path and line
       numbers based on `item.checklist_item.graph_element_under_test`. Use `mcp__filesystem__read_text_file` with `head`/`tail` parameters to extract only the relevant
       lines efficiently. Extract this code as `code_excerpt`.

       2b. **Check Skip Conditions**: If any of the following conditions are met:
           - `code_scope.file` is `N/A` , `SPECIFICATION-ONLY` , or missing
           - Code resolves to external dependency (`vendor/` , submodules)
           - Component mismatch (EL vs CL)

           Then: Create a result object with `final_classification = "out-of-scope"`, append it to `results`, and **proceed to the next item** (do NOT exit).

       2c. **Run Formal Audit Skills**: If a valid `code_excerpt` is found, run these skills in order:
           - /formal-audit-phase1 (include code_excerpt)
           - /formal-audit-phase2 (include phase1 output)
           - /formal-audit-phase3 (include phase1+phase2 outputs)

       2d. **Merge Outputs**: Merge skill outputs into a single audit result object for this item.

       2e. **Append and Continue**: Append the result object to `results`. **Proceed to the next item.**

    3. **Write Output File**: After ALL items have been processed, write the `results` array to <ref id="results"/>.
       - This step is **MANDATORY**.
       - Even if all items were skipped, write the array containing the skip classifications.

    4. **Confirm Completion**: Print a summary and end with: `Output File: {{OUTPUT_FILE}}`
  </instructions>

  <data_sources>
    - **Checklist Item**: `item.checklist_item` (already resolved)
    - **Property File**: `outputs/01e_PROP_PARTIAL_*.json` (for property assertion)
    - **Subgraph**: `item.subgraph` (pre-extracted relevant subgraph, included in the item)
    - **Tree-sitter MCP**: You **MUST** use `mcp__tree_sitter__get_symbols` and `mcp__tree_sitter__run_query` to find code. Direct file access for code resolution is **NOT
    PERMITTED**.
    - **Filesystem MCP**: Use `mcp__filesystem__read_text_file` with `head`/`tail` for efficient partial reads after code scope is resolved. Use
    `mcp__filesystem__search_files` to find related code files.
  </data_sources>
</task>

<output>
  <format>JSON array</format>
  <stdout>Max 8 lines: batch size, items processed, short status.</stdout>
  <final_line>Output File: {{OUTPUT_FILE}}</final_line>
</output>
```

Figure 6: Audit map prompt example.

Figure 7 shows an example audit map output produced by the V2 pipeline.


```
[
  {
    "check_id": "CHECK-W6-PROP-W6-BOUNDARY-MALICIOUS-VALIDATOR-001-BOUNDARY",
    "title": "Verify Trust Boundary Integrity for EDGE-PROPOSER-SELECTED: reject blocks from non-assigned proposers",
    "code_scope": {
      "file": "beacon-chain/core/helpers/beacon_committee.go",
      "function": "proposer selection edge"
    },
    "severity_hint": "HIGH",
    "bug_class": "Unauthorized Block Injection / Fork Choice Manipulation",
    "final_classification": "in-scope-audited",
    "reachability": {
      "classification": "external-reachable",
      "attacker_controlled": true
    },
    "audit_verdict": "requires_manual_review",
    "audit_priority": "MEDIUM"
  },
  {
    "check_id": "CHECK-W6-PROP-W6-BOUNDARY-MALICIOUS-VALIDATOR-001-NODE",
    "title": "Verify proposer_lookahead consistency between block proposer check and epoch computation",
    "code_scope": {
      "file": "beacon-chain/core/helpers/beacon_committee.go",
      "function": "InitializeProposerLookahead"
    },
    "severity_hint": "HIGH",
    "bug_class": "Lookahead Inconsistency",
    "final_classification": "in-scope-audited",
    "audit_verdict": "requires_manual_review",
    "audit_priority": "MEDIUM"
  }
]
```


Figure 7: Audit map output example.