

SPECTRACE: Automatically Auditing the MCP Tool Servers Behind AI Agents for Protocol Compliance

Abstract—AI agents are increasingly integrated into real software systems through the Model Context Protocol (MCP), an emerging standard that lets an agent call external tools and read data sources. Each tool is supplied by a separate MCP server. Because anyone can publish one, these servers are largely third-party code that runs with direct access to a user’s files, databases, and shell. They are becoming critical infrastructure for agent-based software, yet no one checks whether they actually obey the protocol’s security rules. The negative impact of this gap is already clear. A command-injection flaw in the official AWS MCP server allowed remote code execution on the user’s host. The irony is that the rules that would have caught these bugs are written, in plain English, in the MCP specification, but plain English cannot be run against code. We present SpecTrace, which makes the specification automatically checkable. SpecTrace turns each of MCP’s 30 server-side security rules into a machine-checkable logic rule, and uses an LLM only for the narrow, local job of reading a server’s source code into structured facts; a logic engine then decides compliance and reports, for every violation, the exact rule broken and the line of code that broke it. We build a benchmark of 162 real MCP violations from bug-fixing commits across 48 open-source projects. In our evaluation, SpecTrace detects 55.6% of violations, against 29.0% for an LLM given the same rules as context and 19.8% for an LLM with none. Because compliance is decided by logic rather than prose, the result is deterministic, explainable, and cheap enough to run on every commit. We release the benchmark, the first of its kind, to support further work on securing the AI-agent ecosystem.

I. INTRODUCTION

Software verification relies on machine-readable specifications that act as correctness oracles [1]. A compiler is checked against a formal operational semantics [2]; a protocol implementation against a state-machine and packet-format definition [3], [4]. Without an executable specification, a verification framework cannot mechanically separate a correct execution trace from an erroneous one.

AI agents are increasingly integrated into software systems by executing external tools through the Model Context Protocol (MCP) [5]. Under this protocol, third-party developers write MCP servers that expose local file systems, shell environments, and database interfaces directly to client agents. These servers act with host system privileges, so they have become critical infrastructure for agent-based software. Yet the foundational verification principle of checking implementations against machine-readable specifications does not hold here. The MCP security rules exist only as natural-language prose, so the compliance of these privileged servers goes unchecked in practice.

A recent vulnerability in the official AWS MCP server (CVE-2025-5277) shows the consequence of unverified com-

pliance. The server exposed a tool to run command-line interface tasks on behalf of the user. It did not validate the command arguments. An agent processing untrusted data could propagate shell metacharacters to the host shell. The result was remote code execution. The MCP specification dictates that servers must validate all tool inputs. The vulnerability occurred because developers had no mechanical way to check their code against that requirement.

Existing testing techniques fail to verify compliance within the MCP ecosystem. Dynamic debugging tools, such as the official MCP Inspector [6], exercise tools against a running server instance. They evaluate only the execution paths that concrete inputs trigger, leaving unexercised handlers unchecked. Static analysis built on abstract syntax trees is limited by syntactic heterogeneity. Input validation takes many forms. For example, custom regular expressions, Pydantic schemas, or manual conditional checks. Syntax-based analysis cannot recognize the semantic property of input validation across these forms. An LLM that audits code directly produces unstructured, nondeterministic verdicts that cannot yield formal compliance proofs. None of these techniques checks a server’s source code against the specification’s own rules.

Our insight is to separate semantic program abstraction from symbolic compliance reasoning. By decoupling these tasks, we can apply formal logic programming to check unconstrained code against formalized protocol rules. We define an abstract semantic domain. Specifically, a controlled vocabulary of logic predicates. An LLM agent acts as a semantic interpreter that translates diverse source code patterns into structured facts in this vocabulary. A logic programming engine then evaluates the formalized protocol rules over these facts. This separation of concerns allows us to resolve syntactic variance while preserving the rigor of symbolic verification.

We present SPECTRACE, an automated framework to audit MCP server implementations for protocol compliance. To our knowledge, SPECTRACE is the first system to check MCP server source code against the specification’s normative rules. The framework operates in three phases. First, SPECTRACE autoformalizes the natural-language specification rules into executable Prolog clauses. Second, a semantic interpreter, implemented as an LLM-agent, analyzes the target server’s source code to extract concrete ground facts into a fact base conforming to a controlled vocabulary of abstract predicates. Third, a logic engine evaluates the formalized rules against those facts. When it derives a violation, it outputs a structured proof certificate. The certificate identifies the violated rule, the non-compliant code location, and the logical deduction steps.

We evaluate SPECTRACE on a curated benchmark of 162 confirmed MCP violations across 48 open-source repositories. In our evaluation, SPECTRACE detects 55.6% of evaluated violations, against 29.0% for a specification-augmented LLM agent that receives the rules as prompt context. This is a gap of 26.6 percentage points on identical specification knowledge. A vanilla LLM agent, given no rules, detects 19.8%. SPECTRACE runs in 15 seconds per project on average, at an API cost of about \$0.02. This latency and cost are low enough for continuous-integration use.

We make the following contributions.

- **Specification autoformalization.** We introduce an automated methodology to translate natural-language specification requirements into executable Prolog rules, which we use to formalize the 30 server-side rules of the MCP standard.
- **Framework design.** We design and implement SPECTRACE, a compliance auditing framework that combines language-model fact extraction with logic-based evaluation.
- **Violation benchmark.** We construct and release a benchmark of 162 real-world MCP specification violations collected from 48 open-source software repositories, to our knowledge the first such dataset.
- **Empirical evaluation.** We demonstrate that SPECTRACE detects 55.6% of evaluated violations, outperforming a specification-augmented LLM agent (29.0%) and a vanilla LLM agent (19.8%), while maintaining low average analysis overhead (15 seconds and \$0.02 per project).

The remainder of this paper is organized as follows. Section III presents background and a motivating example of an MCP specification violation. Section IV describes the three phases of the SPECTRACE framework. Section V outlines the categories of protocol violations supported by our tool. Section VI details the implementation of our framework, and Section VII evaluates its effectiveness and efficiency. Section VIII discusses threats to validity. Section IX surveys related work, and Section X concludes the paper.

II. BACKGROUND

This section provides background on the Model Context Protocol (MCP) and logic programming.

A. Model Context Protocol (MCP)

The Model Context Protocol (MCP) is an open standard that defines how applications powered by large language models (LLMs) exchange context with external data sources and tools [7]. Figure 1 shows the MCP architecture. The protocol follows a client-server design with three distinct roles, which we describe below.

a) MCP Host: The host is the user-facing application that orchestrates interactions. Examples of hosts include IDEs (e.g., Cursor, VS Code), chat interfaces (e.g., Claude Desktop), or command-line developer tools. A single host process creates and manages one or more MCP clients. Each client handles communication with a specific server connection.

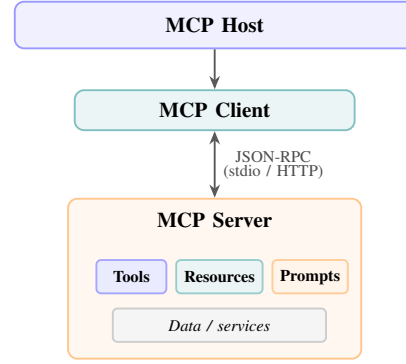


Fig. 1: MCP architecture. An MCP host (e.g., Claude Desktop, Cursor) contains an MCP client that maintains a one-to-one JSON-RPC connection (over `stdio` or Streamable HTTP) with an MCP server. Each server exposes three capability types: tools, resources, and prompts.

b) MCP Client: Each client maintains a stateful, one-to-one session with an MCP server. Under the protocol, the host process does not communicate with servers directly. Instead, it delegates connection management to the client. The client sends JSON-RPC 2.0 requests over one of two standard transports. Under the `stdio` transport, the host spawns the server as a child process and exchanges newline-delimited JSON on standard input and output. Under the `Streamable HTTP` transport, the server exposes a single HTTP endpoint that accepts POST requests and upgrades to Server-Sent Events for streaming responses.

c) MCP Server: Each server exposes up to three categories of capabilities to its client. For example, a filesystem server might expose a tool to write files, while a database server might expose resources containing database schema definitions. Specifically, these capabilities include tools, resources, and prompts. *Tools* are model-callable functions that let the LLM perform actions such as querying a database or calling an API. *Resources* are read-only data endpoints (e.g., files, database records, live system logs) that the host can read on behalf of the user. *Prompts* are reusable, templated interaction patterns that guide the LLM with pre-authored instructions and context.

d) Normative Specification Rules: The specification uses RFC 2119 keywords (MUST, SHOULD, MAY) to define normative requirements for both client and server implementations. Client-side rules govern behaviors such as user consent dialogs and credential handling. These rules are implemented by the developers of the host applications and official client SDKs. In contrast, server-side rules must be satisfied by third-party developers writing custom servers. As anyone can write an MCP server, and because these servers execute with access to local system resources, server-side compliance is critical. For example, a server that supports tools must declare the `tools` capability during initialization, and tool input schemas must be valid JSON Schema objects. SDK frameworks such as FastMCP [8] automate some of these requirements. However,

many normative rules remain the developers’ responsibility.

B. Logic Programming and Prolog

SPECTRACE encodes specification rules as a logic program, so we briefly review it here. Logic programming is a declarative paradigm where computation is expressed as relations between entities. A logic program consists of a set of facts and rules from which a query engine deduces new logical conclusions. Prolog is a widely used logic programming language built on this model [9]. A fact asserts a relationship or property, such as stating that a specific function is a tool handler. A rule defines a conditional relationship in the form `Head :- Body`, meaning the head relation holds if all conditions in the body hold. For example, a rule can specify that a violation occurs if a capability is required but not declared. Finally, a query asks whether a specific relation holds, which the Prolog engine resolves by searching the facts and rules to find satisfying variable assignments.

Logic programming is well-suited for verifying protocol compliance. First, logic rules are declarative, which allows normative specifications to be represented directly in code. Second, logic execution is deterministic, ensuring that the same facts always yield the identical set of findings. Third, the deduction process generates an explicit proof path that links derived conclusions back to the initial facts.

III. MOTIVATION

A. A Real-World Failure of MCP Specification Compliance

LLM-based coding agents increasingly interact with production systems by executing external tools. The Model Context Protocol (MCP) has emerged as the open standard for exposing local capabilities, such as filesystems, shell execution, or cloud APIs, to these agents via standard JSON-RPC handlers [5]. An MCP *server* is a third-party process that exposes three kinds of capability: *tools* (model-callable functions, such as running a shell command), *resources* (read-only data endpoints), and *prompts* (templated interactions). Because anyone can publish a server, and because servers run with direct access to local files, shells, and databases, each server is a security boundary. The specification states server obligations using RFC 2119 keywords (*MUST*, *SHOULD*, *MAY*) [10]: a server that exposes tools, for instance, *MUST* declare the `tools` capability and validate all tool inputs. These server-side rules are the third-party developer’s responsibility, and SDKs such as FastMCP [8] automate only some of them. While this protocol enables agents to take real-world actions, it introduces severe security risks when a server implementation fails to follow MCP specifications.

B. A Real-World Failure

A real-world vulnerability in `aws-mcp-server` (CVE-2025-5277) illustrates this risk [11], [12]. This server exposes the tool `aws_cli_pipeline`, allowing an agent to run AWS commands on behalf of a user. The tool accepts a command string from the agent and pipes it to system utilities (e.g., `jq` or `grep`). Under the hood, the server passed this command

string directly to a system shell. Because the command was not validated, it allowed an attacker to inject shell metacharacters.

This flaw exposes a threat model in which the victim is the end-user running the agent. The agent is tasked with inspecting an untrusted source, such as a poisoned S3 bucket or a public repository containing malicious files. It reads the untrusted data and translates it directly into arguments for a tool call. The input contains shell metacharacters and command separators (e.g., `;`, `|`, or `&&`) appended to a standard command parameter. The agent processes this input and propagates the payload into the MCP request parameters. The server, relying on the inadequate validation logic in `cli_executor.py`, fails to detect these separators and executes the entire string in the host system shell. The attack bypasses the agent’s intent, resulting in arbitrary Remote Code Execution (RCE) with the privileges of the server process and exposing sensitive cloud credentials.

The MCP specification explicitly mandates that servers *MUST* validate all tool inputs [5], [10]. Yet, CVE-2025-5277 arose because this requirement went unverified. Furthermore, an automated tool to detect such issues does not exist. Figure 2 contrasts the unverified execution that led to this vulnerability with the verification pipeline provided by SPECTRACE.

As shown in Figure 2 (left), the prose specification (“validate all inputs”) has no operational connection to the Python implementation. The developer implemented the handler in `handler.py` but omitted validation, and the untrusted payload reached the subprocess shell directly. On the right, SPECTRACE bridges this gap. First, it autoformalizes the specification rule into executable Prolog clauses. Second, it abstracts the Python code into a high-level fact base, and runs a symbolic solver to deduce the violation and produce a verifiable proof certificate.

C. Observations

We draw three observations that motivate the design of SPECTRACE.

a) *Observation 1 [Prose specifications lack an executable form]*: The obligation to validate tool inputs exists only as prose in the specification. RFC 2119 keywords state requirements but carry no machine-executable representation, so no framework can mechanically check them. Compliance is left to manual review, which is error-prone and does not scale.

b) *Observation 2 [Semantic abstractions are hard to build]*: Compliance constraints govern semantic behaviors, such as input validation, authorization bounds, and transport discipline, rather than syntactic patterns. Traditional tools, such as AST pattern matchers and CFG taint analyzers, reason over syntactic signatures. They miss these properties when developers implement them through custom conditionals, validation libraries (e.g., Pydantic), or third-party wrappers, leaving a semantic gap.

c) *Observation 3 [Audits require verifiable proof certificates]*: A trustworthy audit must yield deterministic, auditable verdicts. A pure LLM agent can spot missing checks, but it reasons heuristically. A verdict cannot be trusted unless a

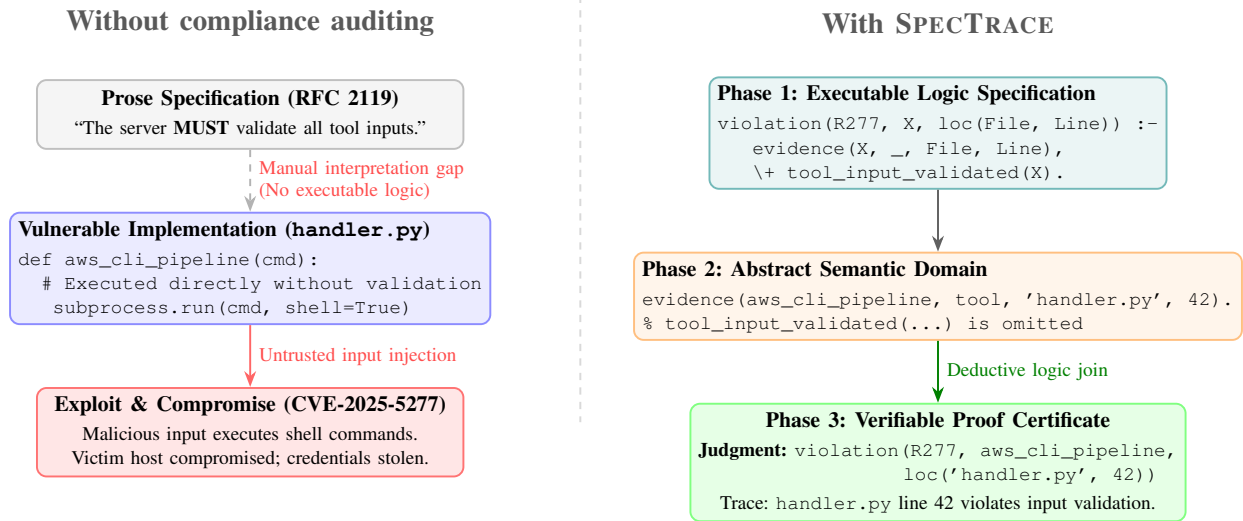


Fig. 2: Contrasting manual compliance checking with SPECTRACE. **Left:** The prose specification is disconnected from the implementation; without automated verification, manual interpretation errors lead to command injection (CVE-2025-5277), letting untrusted input compromise the host running the agent. **Right:** SPECTRACE compiles the spec into executable logic (Phase 1), abstracts the Python codebase into high-level protocol facts via an LLM semantic parser (Phase 2), and runs a logic engine to deduce compliance and generate a traceable proof certificate (Phase 3).

formal proof certificate links each rule to the concrete code evidence.

D. Key Ideas

From these observations, we derive three key ideas that guide the design of SPECTRACE.

a) Key Idea 1 [Executable Declarative Specifications]:

We compile the natural-language specification rules into executable Prolog clauses. Natural-language specification prose lacks a formal representation, which prevents automated compliance checking. By translating each RFC 2119 requirement into a Prolog clause, we obtain an executable logical representation of the specification. These logic rules define the formal relations that must hold for compliance, enabling a reasoning engine to evaluate specification requirements directly against program facts.

b) *Key Idea 2 [Semantic Abstraction with Fact Contracts]:* We bridge the semantic gap by defining a controlled vocabulary of abstract predicates (e.g., a location provenance predicate and behavioral attributes like tool input validation). The vocabulary is constructed incrementally during rule compilation, with a namespace gate that prevents open-vocabulary drift. To audit a target codebase, we mechanically derive a fact contract that defines the exact set of predicates the fact extractor is authorized to query. We then employ an LLM agent as a semantic interpreter. Constrained by the fact contract to prevent open-vocabulary hallucination, the agent evaluates the program’s source code and emits concrete ground facts into a fact base.

c) *Key Idea 3 [Deductive Logic Reasoning and Proof Certificates]:* We decouple semantic perception from logical compliance checking. We feed the agent-extracted facts and

the autoformalized Prolog clauses into a deterministic logic engine. Instead of relying on an LLM to perform logical reasoning, which is fragile and non-reproducible, the Prolog engine evaluates the clauses over the facts to deduce compliance verdicts. When a violation is derived, the solver generates a formal proof certificate, which is a structured trace linking the specification rule directly to the concrete file paths and line numbers of the code. This ensures all judgments are reproducible, auditable, and mechanically verifiable.

IV. SPECTRACE APPROACH

This section presents SPECTRACE, an auditor that decides whether an MCP server complies with the protocol’s normative rules. Figure 3 illustrates the overall architecture of SPECTRACE. It formalizes the specification’s rules into executable logic (①), abstracts a server’s source code into typed behavioral facts (②), and joins the two with a deterministic reasoner that generates traceable findings (③). The first step runs once for the specification; the other two run per server under audit.

A. Preliminaries: Logic Programming

SPECTRACE encodes specification rules as a logic program, which we briefly review. In Prolog, a *fact* asserts a property (e.g., that a function is a tool handler), a *rule* of the form `Head :- Body` holds when its body holds, and a *query* asks whether a relation is derivable [9]. We adopt this model for three reasons. It is declarative, so each normative rule maps directly to a clause. It is deterministic, so the same facts always yield the same findings. Its deduction produces an explicit proof path from every verdict back to the facts that caused it.

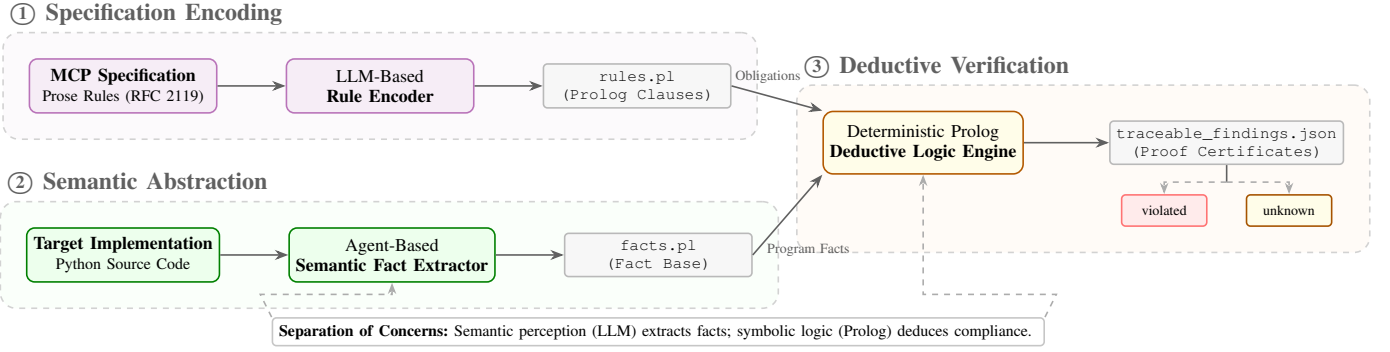


Fig. 3: Overview of SPECTRACE.

B. Problem Formulation

We define the formal concepts and notations used to model the compliance auditing process.

Definition 1 (MCP Server). An *Model Context Protocol* (MCP) server S is a software exposing a set of handler functions (subjects) $Sub(S) = \{s_1, s_2, \dots, s_k\}$, where each subject $s \in Sub(S)$ implements a specific capability endpoint (e.g., a prompt, resource, or tool).

Definition 2 (Normative Rule). A normative rule $r \in R$ is a requirement in the MCP specification that governs a capability using standard deontic operators (e.g., MUST, MUST NOT, SHOULD).

Definition 3 (Subject Governance). A subject $s \in Sub(S)$ is governed by a normative rule $r \in R$ if the capability category of s matches the primitive type of r . We denote this governance relation as $gov(r, s)$.

Definition 4 (Judgment). A judgment $J(r, s) \in \mathcal{J}$ is a compliance verdict assigned to a governed pair (r, s) , where the set of possible verdicts is: $\mathcal{J} = \{\text{violated}, \text{unknown}\}$.

Definition 5 (Proof Trace). A proof trace $P(J(r, s))$ is a structured certificate that maps a judgment $J(r, s) = \text{violated}$ to the rule r evaluated, the subject s under audit, and the logical derivation path over ground facts Φ containing concrete source-code locations.

To anchor abstract logical concepts to physical locations in the target codebase, we introduce a special provenance relation: $evidence(s, K, F, L) \in \Phi$, where $s \in Sub(S)$ is the subject under audit, K is its capability kind (e.g., tool or prompt), and F and L represent the file path and line number of the subject's registration or definition.

Definition 6 (MCP Compliance Audit Problem). Given a target MCP server S , a set of natural-language normative rules R , an autoformalization compiler $\mathcal{F}$, and a semantic fact extractor $\mathcal{E}$:

- 1) Let $\mathcal{M}(R) = \{\mathcal{F}(r) \mid r \in R\}$ be a logic program of executable Prolog clauses formalizing the rules over a controlled vocabulary $\mathcal{V}$.

- 2) Let $\Phi = \mathcal{E}(S)$ be the fact base of ground atoms over $\mathcal{V}$ extracted from the codebase of S , including the provenance facts.
- 3) A subject $s \in Sub(S)$ is non-compliant with respect to rule $r \in R$ (where $gov(r, s)$ holds) if and only if:

$$\Phi \cup \mathcal{M}(R) \vdash \text{violation}(r, s, \text{loc}(F, L))$$

corresponding to the judgment $J(r, s) = \text{violated}$.

The MCP compliance audit problem is to compute the compliance judgment $J(r, s) \in \mathcal{J}$ for all (r, s) satisfying $gov(r, s)$, and to construct a proof trace $P(J(r, s))$ for each violation.

We identify two core technical challenges in auditing protocol compliance. First, natural-language specifications must be compiled into executable rules without introducing semantic drift or disjoint, open-vocabulary predicates. Second, while verifying concrete implementations requires projecting the infinite program state space into a finite abstract domain, constructing this abstraction is inherently difficult. Traditional static analysis techniques rely on structural or control-flow signatures, which cannot reliably identify high-level semantic properties (e.g., input validation safety or authorization bounds) that developers implement through diverse control paths, custom wrappers, or third-party libraries. Auditing must therefore bridge this semantic gap, mapping the concrete execution patterns to an abstract predicate domain.

C. Specification to Logic Program

In this phase, we translate the natural-language normative rules of the protocol specification into executable logic constraints. Manual formalization of protocol specifications is labor-intensive and requires deep formal methods expertise, which restricts the scalability of verification tools. To address this limitation, we design an automated rule compiler that translates prose rules into declarative Horn clauses. However, for the generated constraints to be evaluable, they must interface with program observations over a unified boundary. If the compiler formalizes each rule in isolation, it generates disjoint, open-vocabulary predicates that prevent downstream logical deduction.

To resolve this challenge, the compiler translates rules over a shared controlled vocabulary $\mathcal{V}$ that is constructed incrementally during compilation. We seed $\mathcal{V}$ with a preliminary set of unary predicates representing protocol-specific boolean properties of capability handlers. When a rule $r \in R$ requires a predicate not yet in $\mathcal{V}$, the translator proposes new unary predicate signatures $\Delta\mathcal{V}$. We enforce an automated namespace gate $\mathcal{G}$ on each proposal:

$$\mathcal{G}(\Delta\mathcal{V}) = \{p/1 \in \Delta\mathcal{V} \mid \text{ns}(p) = \text{kind}(r) \wedge p \notin \mathcal{V}\}$$

where $\text{ns}(p)$ is the capability namespace prefix of the predicate (e.g., ‘tool_’ for tool rules), and $\text{kind}(r)$ is the rule’s primitive capability type. Accepted proposals are merged into $\mathcal{V}$, ensuring that every predicate is namespace-consistent and non-redundant. This gate prevents the compiler from introducing disjoint, open-vocabulary predicates that would fragment downstream deduction.

Once the final logic program $\mathcal{M}(R)$ is successfully validated over $\mathcal{V}$, the compiler mechanically derives a Fact Contract $\mathcal{C} \subseteq \mathcal{V}$ containing the exact predicates referenced in the bodies of the clauses. The corresponding dynamic predicate directives (‘:- dynamic p/l.’) are automatically generated and prepended to the rule base to form `rules.pl`.

a) *Running Example:* Consider Rule R_{277} (“Servers MUST: Validate all tool inputs”). The vocabulary contains the predicate `tool_input_validated/1` $\in \mathcal{V}$. Because R_{277} is an obligation, the compiler formalizes it as a violation clause using negation-as-failure:

$$\text{evidence}(X, _, F, L) \wedge \neg \text{tool_input_validated}(X)$$

In Prolog, the rule translator outputs this as the following declarative clause:

```
1 violation(R277, X, loc(File, Line)) :-
2     evidence(X, _, File, Line),
3     \+ tool_input_validated(X).
```

D. Extracting Abstract Behavioral Facts

In this phase, we abstract the raw target server source code into a knowledge base of grounded behavioral facts. Whole-program verification of concrete implementations is undecidable due to infinite state spaces and complex environment interactions. To make verification tractable, static analysis must project the concrete program state space into a finite set of abstract predicates. However, automatically extracting these abstract predicates is difficult because they represent high-level semantic behaviors rather than rigid syntactic patterns. Conventional static checkers struggle to verify protocol requirements, such as input validation or access control, because these properties can be implemented through diverse syntactic constructs, wrapper functions, or third-party libraries.

To overcome this semantic gap, we decouple program observation from syntactic structure. We frame fact extraction as a semantic observation process, using an LLM agent as a semantic interpreter that maps raw source code to the abstract predicate domain. Crucially, the extraction process $\mathcal{E}$ is pull-driven and strictly bounded by the Fact Contract $\mathcal{C}$ generated

in Phase 1. This boundary prevents the extractor agent from emitting open-vocabulary facts or hallucinated predicates.

For each subject $s \in \text{Sub}(S)$ discovered in the server code, the extractor performs semantic code inspection to check whether the semantic property represented by a predicate $p \in \mathcal{C}$ holds. If the agent verifies the property, it emits a positive ground fact: $p(s) \in \Phi$. Additionally, the extractor registers the location of the subject by emitting a provenance fact: $\text{evidence}(s, K, F, L) \in \Phi$, where K is the capability kind, F is the file path, and L is the line number. If the property $p(s)$ is not satisfied, the fact is omitted from Φ , adhering to negation-by-omission.

a) *Running Example:* Consider an MCP server implementing a tool `calculate_bmi` in `server.py` at line 9:

```
1 @server.tool()
2 async def calculate_bmi(weight_kg: float,
3     height_m: float) -> str:
4     bmi = weight_kg / (height_m ** 2)
5     return str(bmi)
```

During repository traversal, the extractor agent identifies `calculate_bmi` as a tool subject. It registers its location by emitting:

```
1 evidence(calculate_bmi, tool, 'server.py', 9).
```

The agent then evaluates whether the input parameters `weight_kg` and `height_m` are validated before execution. Since the division uses `height_m` directly without checking if it is positive or non-zero, the validation property is unsatisfied. As a result, the extractor does *not* emit `tool_input_validated(calculate_bmi)` to the fact base Φ .

E. Deductive Compliance Auditing with Proof Traces

In this phase, we evaluate the compliance of the target server by querying a logic engine over the formalized rules and extracted facts. We formalize the compliance check as a query over a deductive knowledge base. This design separates semantic program abstraction from the evaluation of logical specifications.

Let $\mathcal{M}(R)$ be the logic program representing the specification rules, and let Φ be the extracted fact base. To identify protocol non-compliance, we query the logic engine for the violation relation: $\Phi \cup \mathcal{M}(R) \vdash \text{violation}(r, s, \text{loc}(f, l))$, where $r \in R$ is a rule identifier, $s \in \text{Sub}(S)$ is the non-compliant subject, and f and l represent the file name and line number of the violation.

The engine resolves the query deterministically using backward chaining in a Prolog-style environment. If the query succeeds with a satisfying assignment, the engine outputs the judgment $J(r, s) = \text{violated}$. For each violation, the engine constructs a structured proof trace $P(J(r, s))$ corresponding to the logical derivation path that satisfies the query. The proof trace compiles a derivation tree:

$$\mathcal{D} = \{\phi_1, \phi_2, \dots\} \subseteq \Phi$$

consisting of the ground facts that triggered the violation clause. The engine maps the logical derivation directly to the

source code evidence using the provenance relation. By using a deterministic logic engine, we ensure that the audit process is reproducible and that every violation verdict is accompanied by a checkable proof chain.

a) *Running Example:* To audit the tool input validation rule, the reasoning engine queries the violation relation:

```
1 ?- violation(R277, X, loc(File, Line)).
```

The Prolog engine resolves this query using backward chaining:

- 1) It evaluates the first subgoal of the rule, matching evidence(`calculate_bmi`, `tool`, `'server.py'`, 9) in the fact base Φ . This binds the variables: $X = \text{calculate_bmi}$, `File` = `'server.py'`, and `Line` = 9.
- 2) It then evaluates the negative subgoal $\neg \text{tool_input_validated}(\text{calculate_bmi})$.
- 3) Because the fact `tool\_input\_validated(calculate\_bmi)` is absent from Φ , the negation-as-failure operator succeeds.
- 4) The query succeeds, yielding the judgment $J(R_{277}, \text{calculate_bmi}) = \text{violated}$.

The engine outputs a structured proof trace certificate mapping the violation of R_{277} on subject `calculate_bmi` directly to line 9 of `server.py`, providing the developer with clear, traceable compliance diagnostics.

V. SUPPORTED MCP VIOLATIONS

We focus our analysis on server-side rules because they represent the primary security boundary of the protocol. Host applications and client SDKs are developed centrally by platform vendors (such as Claude Code, Cursor, or OpenAI). In contrast, MCP servers are written by a decentralized community of third-party developers. These servers execute with direct access to local system resources, such as filesystems and databases. Thus, server-side compliance is critical to prevent security vulnerabilities and connection failures.

To that end, we collected the 30 server-side compliance rules, including security-relevant rules, defined in the MCP specification. We organize the supported rules along two dimensions. The first is the MCP protocol area to which a rule belongs: transport, tools, resources, or prompts. This dimension follows the structure of the MCP specification and is used to report coverage across protocol components. These rules cover standard server capabilities (tools, resources, and prompts) along with the underlying transport protocol. We group these rules into four protocol groups, represented with identifiers G1 through G4. We describe each group below and summarize the full set of rules in our replication package [13].

- **G1: Transport.** Transport rules govern how servers exchange JSON-RPC messages with clients over `stdio` and `Streamable HTTP` channels. For example, the specification prohibits servers from writing non-MCP content to `stdout`, requires validation of the `Origin` header on HTTP connections to prevent DNS rebinding, and restricts session identifiers to visible ASCII characters.

- **G2: Tools.** Tool rules govern how servers register and expose callable functions. For example, servers must declare the `tools` capability, provide a valid JSON Schema for every tool’s input, validate tool inputs, and sanitize tool outputs.
- **G3: Resources.** Resource rules govern how servers expose readable data endpoints. For example, servers must declare the `resources` capability, validate all resource URIs, and properly encode binary data.
- **G4: Prompts.** Prompt rules govern how servers define templated interaction patterns. For example, servers must declare the `prompts` capability, base64-encode image and audio content with a valid MIME type, and validate prompt inputs and outputs to prevent injection attacks.

VI. IMPLEMENTATION

SPECTRACE is implemented in Python. The baselines and parts of the SPECTRACE agent are built on top of `swe-agent` [14]. An off-the-shelf Prolog engine [15] serves as the logic backend, evaluating the formalized rules against the extracted fact base. We used `GPT-5.4` for both rule formalization and fact extraction.

VII. EMPIRICAL EVALUATION

We address the following research questions to evaluate SPECTRACE.

- **RQ1 (Effectiveness):** How effective is SPECTRACE in detecting protocol violations in agent tool (MCP) implementations compared to baselines?
- **RQ2 (Stratification Analysis):** Does detection effectiveness vary across MCP protocol groups, and what does that variation reveal about SPECTRACE and the baselines?
- **RQ3 (Efficiency):** How does SPECTRACE compare to the baseline in terms of analysis time?

A. Dataset of MCP Violations

To evaluate SPECTRACE and compare it to baseline approaches, we collected MCP-related bug-fixing commits from open-source GitHub repositories. We followed methodologies established by prior work on deep learning bug studies [16], [17] and adapted them to the MCP ecosystem. Our collection process consists of the following:

Step 1: Repository discovery. We searched GitHub using the query `topic:mcp language:python` to capture MCP implementations.

Step 2: Filtering. We removed repositories that (a) had no open issues (inactive), (b) contained fewer than 100 commits (trivial history), or (c) had fewer than 10 stars and fewer than 10 forks (insufficient community adoption). This step is adopted from prior work [17] because it has been shown to retain only mature, actively maintained projects. Furthermore, it also reduces the risk of including toy or abandoned implementations. After filtering, 79 repositories remained.

Step 3: Bug-fixing commit mining. We mined the commit history of each remaining repository for commits whose messages match bug-fixing keywords (i.e., “fix” and “bug”). This step yielded 6,906 candidate commits across 79 repositories.

To identify violations related to the MCP rule groups defined in section V (transport, tools, resources, and prompts), we first employed an LLM to filter out candidate commits that did not contain protocol violations. This step yielded 202 candidate commits. This pre-filtering step is justified by prior empirical evidence validating the use of LLMs to reduce manual software engineering artifact annotation overhead [18]. Two authors then manually audited the remaining candidate commits. We retained only commits that modify Python source files using the `python-sdk` or `fastmcp` library. During the manual review, we discarded 40 commits because they either belonged to repositories without a working URL required to clone and run the codebase, or bug-fixing commits not supported by our approach. This process produced a benchmark of 162 confirmed MCP violations across 48 repositories.

To evaluate the reliability of our manual annotation, the two authors independently labeled a random subset of 20% of the candidate commits. This validation process was conducted over three iterations, yielding Cohen’s kappa coefficients of 0.76, 0.60, and 0.67, indicating moderate to substantial agreement. Disagreements were resolved through consensus discussions after each iteration. Finally, the remaining 80% of the dataset was partitioned and annotated individually. For each confirmed violation, we recorded the violated rule and its rule group. To the best of our knowledge, this is the first curated dataset of real-world violations of MCP specifications.

B. Experimental Setup

a) Baseline: We compare SPECTRACE against two LLM-based baselines that represent the naive and best-effort points in the design space of specification-aware code auditing.

Vanilla Agent is an end-to-end LLM auditor that receives only the target server’s source code and a task prompt asking it to identify MCP protocol violations. It is given no specification text, no predefined rule set, and no vocabulary contract. This baseline isolates the contribution of explicit specification grounding: any violation the vanilla agent detects must be recalled from the model’s parametric knowledge of MCP rather than derived from the authoritative rule text. We use the same backbone model as that used in SPECTRACE (GPT-5.4) to hold model capacity constant.

Spec-Augmented Agent is an end-to-end LLM auditor that receives the full MCP specification document prepended to its context window, together with the target server’s source code. It has access to the same normative rules as SPECTRACE but lacks the structured decomposition into a shared vocabulary, formalized logic clauses, and a deterministic reasoning backend. The agent is prompted to read the specification, identify applicable rules, inspect the source code, and report violations along with the relevant file and line number. This baseline measures the contribution of SPECTRACE’s symbolic architecture. In particular, it tests the hypothesis that the separation of semantic fact extraction from deterministic deductive reasoning over the shared observation vocabulary $\mathcal{V}$ improves effectiveness and efficiency.

b) Agent Configurations: To ensure a fair evaluation and isolate the effects of our proposed deductive compliance architecture, we maintain strict experimental controls across all evaluated configurations (Vanilla Agent, Spec-Augmented Agent, and SPECTRACE’s Phase ① and ② agents):

- **Model and Hyperparameters:** All configurations utilize the same underlying foundation model (GPT-5.4) configured with a temperature of 0.0.
- **Execution Constraints:** The agents share the same maximum step limit of 50 steps and a cost budget of \$0.50 per invocation, executing under identical environment timeout parameters. These thresholds represent conservative safety ceilings established through empirical validation during pilot runs. They thus serve as a conservative upper bound, ensuring that no evaluation is prematurely truncated due to resource limits.
- **Repository Access:** Each agent is granted the same read-only access to the target repository workspace, allowing traversal of the source directory tree.
- **Output Schema and Task Alignment:** Although the prompts differ in operational instructions (since the baselines detect compliance violations end-to-end, while SPECTRACE’s agent focuses on formalization and local semantic fact extraction), all configurations are aligned to the common task of auditing the codebase for protocol violations. To ensure a fair comparison, all evaluations are constrained to the same structured output schema, mapping compliance issues to rule identifiers, target files, line numbers, and reasoning explanation strings.

c) Metrics: We measure three quantities. *Recall* is the fraction of benchmark violations that the tool detects: $\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$, where a prediction counts as a true positive only when the agent both flags a violation *and* localizes it to the correct file.

Rule EM measures whether the predicted rule exactly matches the ground-truth rule violated at the flagged file:

$$\text{RuleEM} = \frac{|\{i \in \text{TP} : \hat{r}_i = r_i\}|}{|\text{TP}|}$$

Group EM measures whether the predicted MCP protocol group (i.e., transport, tools, resources or prompts) exactly matches the ground-truth group:

$$\text{GroupEM} = \frac{|\{i \in \text{TP} : \hat{g}_i = g_i\}|}{|\text{TP}|}$$

C. RQ1: Effectiveness

Experimental Setup. We compare the three approaches: the Vanilla Agent, the Spec-Augmented Agent, and SPECTRACE.

TABLE I: Effectiveness of MCP violation detection.

Metric	Vanilla Agent	Spec-Augmented Agent	SpecTrace
Recall	19.8%	29.0%	55.6%
Rule EM	—	42.5%	62.2%
Group EM	62.5%	68.1%	100.0%

Results. Table I reports violation detection effectiveness across the evaluated approaches. The three-way comparison isolates the contribution of *formalization* from that of *specification knowledge*. Both the Spec-Augmented Agent and SPECTRACE have access to the same normative rules. They differ only in how they reason over them. The Spec-Augmented Agent injects the rules as prompt context and reasons in natural language, whereas SPECTRACE compiles them into executable Prolog clauses and evaluates them deterministically over structured facts.

The Vanilla Agent, operating solely on parametric knowledge without any specification text, achieves 19.8% recall. The Spec-Augmented Agent raises recall to 29.0%, a 9.2-percentage-point gain that quantifies the effect of explicit specification grounding. SPECTRACE achieves 55.6% recall, a further 26.6-percentage-point gain over the Spec-Augmented Agent. Because both the baseline and our approach operate on identical normative rules, this gap is attributable entirely to the reasoning architecture. In particular, autoformalization into executable Horn clauses and deductive evaluation over a shared vocabulary $\mathcal{V}$, as described in our approach. By constraining the LLM to the narrow task of emitting structured facts under a fact contract, and delegating compliance evaluation to the Prolog engine, SPECTRACE avoids the attention-window saturation and heuristic failures that cause monolithic neural reasoning to miss violations in long handlers.

The difference in reasoning architecture is also visible in attribution accuracy. For group-level attribution, SPECTRACE achieves 100% Group EM, compared to 68.1% for the Spec-Augmented Agent and 62.5% for the Vanilla Agent. This result occurs because each formalized Prolog rule is mapped to its respective protocol group during rule compilation, allowing group membership to be determined by construction. Similarly, Rule EM rises from 42.5% for the Spec-Augmented Agent to 62.2% for SPECTRACE. The 19.7-percentage-point difference within true positives indicates that when the Spec-Augmented Agent does detect a violation, it frequently misattributes it to an unrelated specification clause, whereas SPECTRACE’s symbolic engine traces each violation back to the exact clause that triggered it.

Finding 1: Decoupling semantic fact extraction from deterministic reasoning improves violation detection recall to 55.6% and enhances attribution accuracy, demonstrating the effectiveness of hybrid neuro-symbolic compliance verification over purely neural baselines.

D. RQ2: Stratification Analysis

Experimental Setup. To answer this research question, we stratify the benchmark violations by MCP protocol group and measure per-group recall across the three approaches from RQ1. The benchmark spans four groups: Transport Protocol (62 instances), Tools Protocol (50 instances), Resources Protocol (41 instances), and Prompts Protocol (9 instances). Stratification reveals whether the aggregate recall gains reported in

RQ1 are uniform or concentrated, and whether certain protocol concerns systematically resist neural detection.

TABLE II: Per-group recall across the evaluated approaches.

Protocol Group	n	Vanilla	Spec-Augmented	SpecTrace
Transport Protocol	62	27.4%	45.9%	71.0%
Tools Protocol	50	22.0%	18.8%	52.0%
Resources Protocol	41	12.2%	19.5%	31.7%
Prompts Protocol	9	0.0%	22.2%	55.6%

Results. Table II shows that SPECTRACE consistently outperforms both baselines across all protocol groups, with particularly pronounced gains in Transport and Tools. The comparison across layers highlights a key weakness in pure neural auditing. In Tools, injecting specification rules as text in the Spec-Augmented Agent actually degrades recall compared to the Vanilla Agent. This degradation indicates a key challenge in end-to-end neural verification. In particular, when rules require checking multiple distributed conditions, requiring a single-pass model to locate relevant program elements and evaluate those conditions simultaneously can hinder detection performance. SPECTRACE resolves this by decoupling semantic fact extraction from logical verification.

The core intuition behind SPECTRACE’s superior performance is its separation of concerns, which shifts multi-conditional checks from a probabilistic language model to a deterministic logic engine. The model is only tasked with extracting local program observations as ground facts, while Prolog handles composition and verification. However, this logic-based auditing faces two distinct challenges. First, a *deductive mismatch* occurs when developer implementation styles diverge from the formal protocol specification $\mathcal{M}(R)$. In the MCP specification, resources and tools are distinct capabilities with separate predicates in the controlled vocabulary $\mathcal{V}$ (e.g., *resource_uri_validated(X)* vs. *tool_input_validated(X)*). In practice, we observe that developers routinely implement file-access and path resolution as standard MCP tools (using the `@mcp.tool()` decorator) rather than MCP resources. Consequently, the extractor $\mathcal{E}$ identifies the subject $X \in \text{Sub}(S)$ as a tool and asserts *tool_input_validated(X)*, omitting *resource_uri_validated(X)*. Under the negation-as-failure (NAF) rule for resource URI validation:

$$\text{violation}(R203, X, \text{loc}(F, L)) \leftarrow$$

$$\text{evidence}(X, G, F, L) \wedge \neg \text{resource_uri_validated}(X)$$

the logic engine derives a violation even when the tool validates the resource path, because the Prolog database lacks the cross-protocol rules to unify these namespaces.

Second, an *observational gap* arises from modular indirection. In many servers, subject registrations (representing tools, resources, or prompts) occur at entry points (e.g., `server.py`), while their actual implementations reside in separate handler modules (e.g., `utils/loader.py`). Because the fact extraction interpreter $\mathcal{E}$ processes files in isolation, it fails to resolve inter-procedural bindings across files.

Consequently, the extractor cannot link the verification facts in handler files to the subject registration in the entry file, preventing the logic engine from proving compliance at the correct provenance file path. Closing this observational gap requires extending fact extraction to trace inter-procedural bindings across modular boundaries.

Finding 2: SPECTRACE outperforms both neural baselines across all four protocol groups. Notably, injecting specification text into the LLM can *degrade* recall on multi-conditional rules, whereas symbolic evaluation over extracted facts performs better compared to baselines.

E. RQ3: Efficiency

Experimental Setup. We evaluate the execution efficiency and resource consumption of the three approaches by measuring: (1) wall-clock execution time (averaged over five runs to mitigate network and API latency variance), (2) API token usage (both input/prompt and total tokens), and (3) average financial costs based on backbone model pricing.

TABLE III: Cost and resource consumption per approach.

Metric	Vanilla Agent	Spec-Augmented Agent	SpecTrace
Avg Cost (USD)	\$0.035	\$0.05	\$0.02
Avg Time (s)	18	25	15
Avg Steps	11	11	8
Avg Prompt Tokens	20.9K	27.2K	17.8K
Avg Total Tokens	22K	29.1K	19.7K

Results. Table III summarizes resource consumption across the evaluated approaches. SPECTRACE executes faster (15 s average) than both the Vanilla Agent (18 s) and the Spec-Augmented Agent (25 s). This reduction in wall-clock time corresponds directly to a reduction in interaction steps (8 steps versus 11). By constraining the LLM to extract specific ground facts rather than conducting open-ended heuristic reviews, SPECTRACE reduces the search space and avoids unguided exploration. The Prolog reasoning engine, which performs the actual compliance verification over these facts, evaluates the specifications in sub-second time, adding negligible latency.

The Spec-Augmented Agent incurs the highest computational cost. It consumes 27.2K prompt tokens on average (a 52.8% increase over SPECTRACE) and yields the highest execution cost (\$0.05). This overhead is structural: injecting raw specification text into the context window forces the neural model to perform both retrieval and reasoning over the compliance rules at runtime. Conversely, SPECTRACE minimizes context size (17.8K prompt tokens) by offloading the normative verification logic to the external Prolog engine.

This architectural factoring makes SPECTRACE the most cost-effective approach. Averaging \$0.02 per execution, SPECTRACE is 42.9% cheaper than the Vanilla Agent (\$0.035) and 60.0% cheaper than the Spec-Augmented Agent (\$0.05). These cost reductions occur despite SPECTRACE achieving significantly higher violation detection recall (55.6% versus

19.8% and 29.0%, respectively, as shown in RQ1). The combination of lower cost and higher effectiveness demonstrates that decomposing the audit task. Specifically, assigning structure extraction to the LLM and deterministic reasoning to the formal logic engine is strictly more efficient than monolithic neural reasoning.

Finally, SPECTRACE generates a structured output consisting of Prolog predicates, which accounts for its completion tokens (1.9K on average). This generation represents the necessary cost of translating neural observations into machine-checkable facts. At \$0.02 per execution and a 15-second runtime, SPECTRACE demonstrates that formal verification over LLM-extracted facts decreases both latency and monetary cost compared to purely neural baselines, establishing its practicality for continuous integration environments. Importantly, the one-time cost of formalizing the specification (vocabulary construction and rule compilation) is \$0.02 across 7 API calls. Because this step runs once per specification version and produces reusable Prolog artifacts, it is amortized over all subsequent audits.

Finding 3: SPECTRACE reduces execution latency to 15 s and financial cost to \$0.02 per run, outperforming both neural baselines. Decomposing compliance auditing into local fact extraction and symbolic verification reduces input context size and avoids unguided LLM exploration, demonstrating that hybrid neuro-symbolic reasoning is more efficient than monolithic neural reasoning.

VIII. THREATS TO VALIDITY

We identify threats to validity organized by category.

Internal Validity. The manual classification of ground-truth violations from bug-fixing commits is susceptible to human annotator bias. To mitigate this threat, two authors independently labeled a randomly selected calibration subset of 20% of our benchmark. We measured inter-annotator agreement on this subset, yielding an overall Cohen’s Kappa of moderate agreement to substantial agreement. The annotators reconciled all disagreements through discussion to establish a unified labeling guideline. The remaining commits were then partitioned and annotated individually, followed by a final joint review. Second, the non-determinism of the foundation model during fact extraction could lead to inconsistent compliance results. We addressed this by setting the model temperature to 0.0.

External Validity. The evaluation dataset is limited to Python implementations of MCP servers. Specifically, all benchmark projects are written in Python and rely on the `python-sdk` or `fastmcp` libraries. While Python is the dominant language for language-model integrations, developer mistakes and protocol compliance issues might manifest differently in other language ecosystems. The size and diversity of the benchmark dataset may restrict generalizability. To address this threat, we gathered projects from 48 different GitHub repositories with diverse star ratings and adoption levels.

IX. RELATED WORK

SPECTRACE sits at the intersection of protocol conformance checking, MCP security analysis, and automated specification formalization. We now discuss related ideas.

Protocol Conformance Checking. Conformance checking verifies that a software implementation complies with its governing protocol specification. Traditional approaches rely on model checking [3] or static differential analysis [19] to identify non-compliance. In the context of API usage, NeuralState [20] applies a tpestate-based technique to detect usage protocol violations in deep learning programs. Recent techniques focus on validating network protocol parsers by extracting specifications (e.g., from RFC documents). For instance, Mingwei *et al.* [4] introduce an LLM-based technique to convert an RFC document into a graph. The graph is later used to detect incorrect parser implementations. Similarly, ProtocolGuard [21] combines LLM-guided static slicing with dynamic verification. Furthermore, LLM-based agents have also been proposed to audit protocols. For example, RFCAudit [22] uses an LLM agent for functional bug detection.

Unlike these techniques, which are tailored to structural parser code or sequential tpestate properties, SPECTRACE addresses the compliance of MCP servers. Checking MCP compliance requires validating semantic constraints (such as whether a server validates tool inputs) that are implemented via diverse syntactic patterns. Furthermore, static slicing techniques (such as those in ProtocolGuard [21]) are unsuitable for MCP servers. In particular, slicing abstracts away statements that do not influence specific execution targets, whereas an MCP specification violation can be triggered by any unvalidated input or missing capability declaration within a handler, meaning slicing would increase false negatives.

MCP Security Analysis. As the Model Context Protocol is increasingly adopted to connect LLMs to external systems, security auditing of MCP implementations has become critical. Initial studies analyze the security threats and maintainability of the MCP ecosystem, mapping potential attack vectors and vulnerabilities [23], [24]. To secure interactions at runtime, MCIP [25] introduces a contextual integrity protocol that monitors information flow and applies runtime safety guardrails. In contrast, DeltaMCP [26] focuses on the maintenance of MCP servers, using specification-aware code transformations to incrementally regenerate servers on the APIs change.

While existing work analyzes the security landscape, enforces runtime safety policies, or regenerates code, SPECTRACE is the first to propose an approach that verifies the conformance of third-party MCP server implementations against the normative rules of the official MCP specification.

Automated Specification Formalization. Formal verification requires machine-readable specifications, prompting research into automating the translation of natural-language requirements into formal models. Early work translates code comments into procedure specifications [27]. With the advent of LLMs, approaches have been proposed to synthesize Signal Temporal Logic (STL) from requirements [28], generate post-

conditions from intent [29], or translate requirements to Linear Temporal Logic (LTL) using semantic decomposition [30], [31]. Expecto [1] extracts executable oracles from natural-language descriptions, while HoarePrompt [32] uses structural prompting for program correctness. LTL specifications have also been utilized to guide greybox fuzzing [33].

While prior works synthesize program assertions, temporal logic formulas, or postconditions for individual functions, SPECTRACE translates the natural-language normative rules of an entire protocol specification into a logic program of Prolog clauses over a shared controlled vocabulary. A deterministic reasoner then utilizes the executable logic representation and a program fact base to query for violations.

X. CONCLUSION

The Model Context Protocol (MCP) enables AI agents to interact with external environments through a decentralized ecosystem of third-party tool servers. Because these servers execute with host-level privileges, verifying their compliance with protocol specifications is critical to preventing security vulnerabilities and operational faults. We present SPECTRACE, a conformance-auditing framework that makes natural-language specifications executable. The core contribution of SPECTRACE is the separation of semantic program observation from symbolic compliance verification. By employing a large language model as a local semantic interpreter constrained by a Fact Contract, SPECTRACE projects source code into a uniform abstract fact base. A deterministic Prolog engine then evaluates formalized specifications over these facts, yielding reproducible compliance verdicts and machine-checkable proof certificates.

Our evaluation on a benchmark of 162 real-world violations across 48 repositories demonstrates that this neuro-symbolic separation outperforms purely neural auditing. SPECTRACE detects 55.6% of violations, nearly doubling the performance of an end-to-end specification-augmented LLM auditor. More importantly, the results show that the primary driver of auditing effectiveness is not specification knowledge, but specification formalization. In particular, grounding reasoning in symbolic logic avoids the context-window saturation and heuristic failures common in LLMs.

XI. DATA AVAILABILITY

Our replication package (data, benchmark, and source code) is available on Zenodo [13].

XII. ACKNOWLEDGEMENTS

We used AI-assisted tools to support software development and improve the clarity of the manuscript.

REFERENCES

- [1] D. Lee and K. Heo, “Expecto: Extracting formal specifications from natural language description for trustworthy oracles,” *Proceedings of the ACM on Programming Languages*, 2026.
- [2] X. Leroy, “Formal verification of a realistic compiler,” *Communications of the ACM*, 2009.
- [3] M. Musuvathi, D. R. Engler *et al.*, “Model checking large network protocol implementations,” in *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, 2004.
- [4] M. Zheng, D. Xie, Q. Shi, C. Wang, and X. Zhang, “Validating network protocol parsers with traceable RFC document interpretation,” *Proceedings of the ACM on Software Engineering*, 2025.
- [5] Anthropic, “Model context protocol specification (2025-11-25),” <https://modelcontextprotocol.io/specification/2025-11-25>, 2025, accessed: June 24, 2026.
- [6] Anthropic, “MCP Inspector,” <https://modelcontextprotocol.io/docs/tools/inspector>, 2025, accessed: 2026.
- [7] Anthropic, “Model context protocol (MCP),” <https://modelcontextprotocol.io/docs/getting-started/intro>, accessed: 2026-06-27.
- [8] PrefectHQ, “FastMCP: The fast, Pythonic way to build MCP servers and clients,” <https://github.com/PrefectHQ/fastmcp>, accessed: 2026-06-27.
- [9] L. Sterling and E. Y. Shapiro, *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1994.
- [10] S. Bradner, “Key words for use in RFCs to indicate requirement levels,” RFC 2119, Internet Engineering Task Force, 1997, <https://www.rfc-editor.org/rfc/rfc2119>.
- [11] MITRE, “CVE-2025-5277: OS command injection in aws-mcp-server,” National Vulnerability Database, 2025, <https://nvd.nist.gov/vuln/detail/CVE-2025-5277>.
- [12] SentinelOne, “CVE-2025-5277: aws-mcp-server command injection RCE flaw,” <https://www.sentinelone.com/vulnerability-database/cve-2025-5277/>, 2025, accessed: June 2026.
- [13] anonymous, “SpecTrace replication package,” 2026, <https://zenodo.org/records/21071225>.
- [14] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press, “SWE-agent: Agent-computer interfaces enable automated software engineering,” in *Proceedings of the 38th Annual Conference on Neural Information Processing Systems*, 2024.
- [15] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, “SWI-Prolog,” *Theory and Practice of Logic Programming*, 2012.
- [16] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, “A comprehensive study on deep learning bug characteristics,” in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.
- [17] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, “Taxonomy of real faults in deep learning systems,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.
- [18] T. Ahmed, P. Devanbu, C. Treude, and M. Pradel, “Can llms replace manual annotation of software engineering artifacts?” in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories*, 2025.
- [19] M. Zheng, Q. Shi, X. Liu, X. Xu, L. Yu, C. Liu, G. Wei, and X. Zhang, “ParDiff: Practical static differential analysis of network protocol parsers,” *Proceedings of the ACM on Programming Languages*, 2024.
- [20] F. Batole, R. Manke, R. Dyer, T. N. Nguyen, and H. Rajan, “Typestate-based fault localization of API usage violations in a deep learning program,” *IEEE Transactions on Software Engineering*, 2026.
- [21] X. Song, L. Pei, J. Wu, Y. Zeng, G. He, C. Zuo, X. Liu, Q. Zhao, and S. Guo, “ProtocolGuard: Detecting protocol non-compliance bugs via LLM-guided static analysis and dynamic verification,” in *Proceedings of the Network and Distributed System Security Symposium*, 2026.
- [22] M. Zheng, C. Wang, X. Liu, J. Guo, S. Feng, and X. Zhang, “RFCAudit: AI agent for auditing protocol implementations against RFC specifications,” in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, 2025.
- [23] X. Hou, Y. Zhao, S. Wang, and H. Wang, “Model context protocol (MCP): Landscape, security threats, and future research directions,” *ACM Transactions on Software Engineering and Methodology*, 2025.
- [24] M. M. Hasan, H. Li, E. Fallahzadeh, G. K. Rajbahadur, B. Adams, and A. E. Hassan, “Model context protocol (MCP) at first glance: Studying the security and maintainability of MCP servers,” *ACM Transactions on Software Engineering and Methodology*, 2025.
- [25] H. Jing, H. Li, W. Hu, Q. Hu, X. Heli, T. Chu, P. Hu, and Y. Song, “MCIP: Protecting MCP safety via model contextual integrity protocol,” in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025.
- [26] A. Pujara, X. Zhu, and H.-T. Chen, “DeltaMCP: Incremental regeneration via spec-aware transformation for MCP servers,” *arXiv preprint arXiv:2605.28148*, 2026.
- [27] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, “Translating code comments to procedure specifications,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.
- [28] J. He, E. Bartocci, D. Ničković, H. Isakovic, and R. Grosu, “DeepSTL: From english requirements to signal temporal logic,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022.
- [29] 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*, 2024.
- [30] Z. Ma, C. Wen, Z. Su, X. Liang, C. Tian, S. Qin, and M. Yang, “Bridging natural language and formal specification—automated translation of software requirements to ltl via hierarchical semantics decomposition using llms,” in *2025 40th IEEE/ACM International Conference on Automated Software Engineering*, 2025.
- [31] D. Mendoza, A. Mavridou, A. Katis, and C. Trippel, “Automating requirements formalization: Using LLMs and low-complexity distinguishing traces for semantic validation,” in *Proceedings of the 48th International Conference on Software Engineering*, 2026.
- [32] D. S. Bouras, Y. Dai, T. Wang, Y. Xiong, and S. Mehtaev, “Hoare-Prompt: Structural reasoning about program correctness in natural language,” in *Proceedings of the 48th International Conference on Software Engineering*, 2026.
- [33] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury, “Linear-time temporal logic guided greybox fuzzing,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022.