

Causality Laundering: Denial-Feedback Leakage in Tool-Calling LLM Agents

Mohammad Hossein Chinaei*
Independent Researcher

April 2026

Abstract

Tool-calling LLM agents can read private data, invoke external services, and trigger real-world actions, creating a security problem at the point of tool execution. We identify a denial-feedback leakage pattern, which we term *causality laundering*, in which an adversary probes a protected action, learns from the denial outcome, and exfiltrates the inferred information through a later seemingly benign tool call. This attack is not captured by flat provenance tracking alone because the leaked information arises from causal influence of the denied action, not direct data flow.

We present the *Agentic Reference Monitor* (ARM), a runtime enforcement layer that mediates every tool invocation by consulting a provenance graph over tool calls, returned data, field-level provenance, and denied actions. ARM propagates trust through an integrity lattice and augments the graph with counterfactual edges from denied-action nodes, enabling enforcement over both transitive data dependencies and denial-induced causal influence. In a controlled evaluation on three representative attack scenarios, ARM blocks causality laundering, transitive taint propagation, and mixed-provenance field misuse that a flat provenance baseline misses, while adding sub-millisecond policy evaluation overhead. These results suggest that denial-aware causal provenance is a useful abstraction for securing tool-calling agent systems.

1 Introduction

Tool-calling LLM agents increasingly interact with external systems by invoking tools that read data, call services, and trigger consequential side effects [22, 26]. A single agent session may query a user’s inbox, retrieve financial records, draft a document, and send it to a third party, all mediated by protocols such as MCP [3] that expose a growing catalog of callable tools. The critical security boundary in such systems is not text generation itself, but tool execution: every tool call is an access decision, and every tool result imports data from a source the agent does not control. Securing these systems therefore requires principled enforcement at the tool boundary, rather than relying solely on post hoc detection or filtering.

Recent work has made real progress on runtime enforcement for tool-calling agents. Information-flow-control systems such as FIDES [6] enforce label-based restrictions over observed flows. Secure-by-design architectures such as CaMeL [8] extract trusted control and data flow from the user query and enforce capability-style restrictions. Graph-based approaches such as PCAS [21] construct dependency graphs over tool-call events and evaluate declarative policies against them. Causal attribution defenses [16, 31] use counterfactual re-execution to determine whether a proposed action is driven by user intent or by injected content. These systems represent genuine progress; collectively, they show that principled runtime enforcement for agents is both feasible

*Corresponding author. Email: mohammadh.chinaei@gmail.com. LinkedIn: <https://www.linkedin.com/in/mohammadhossein-chinaei/>

and necessary. However, they primarily model successful executions, returned data, and observed action traces, rather than explicitly representing denied actions as events with potential downstream causal influence.

A blocked action is not always the end of an attack. Consider an agent that, under adversarial influence, attempts to read a user’s salary record. The enforcement layer denies the call, and no sensitive bytes are returned. Yet the denial itself may still reveal security-relevant information: the agent can infer that such a record exists, that it is protected, or, if the denial includes a reason, that the record belongs to a particular category of sensitive data. A later tool call that mentions “compensation” to an external party can therefore leak information that was never directly read from the protected resource. The leakage arises from the denial outcome, not from any successful data transfer. Flat taint tracking does not capture this behavior because no sensitive value ever enters the ordinary tool-result channel, and dependency graphs over successful executions contain no direct data-flow edge from the blocked action to the later exfiltration step. Existing causal attribution checks may also miss this pattern, because the triggering influence is the denial event itself rather than untrusted content returned by the tool. Recent empirical evidence supports the plausibility of this channel: frontier LLM agents have been shown to infer hidden monitoring purely from blocking feedback, even when they are not explicitly told that monitoring is present [15].

We term this pattern *causality laundering*: an adversary probes a protected action, learns from the denial outcome, and exfiltrates the inferred information through a later seemingly benign tool call. Existing provenance models over successful executions do not explicitly record this dependency, because the relevant influence arises from the denial event rather than from returned data. Causality laundering is a specific instance of a broader class of denial-feedback implicit flows in tool-calling agents. To our knowledge, existing defenses do not explicitly represent denied actions as first-class provenance events and enforce over their downstream influence.

To address this gap, we present the *Agentic Reference Monitor* (ARM), a provenance-aware runtime enforcement layer that treats denied actions as first-class events in the execution graph. When a tool call is denied, ARM records the denial as a node in the provenance graph and introduces counterfactual edges to subsequent actions that may have been causally influenced by it. Trust then propagates through an integrity lattice over transitive data dependencies, field-level provenance, and denial-induced counterfactual paths, allowing downstream actions shaped by a denial to inherit the security-relevant context of the denied event. All policy decisions are computed by deterministic graph traversals and explicit rules, rather than delegated back to the LLM under scrutiny.

Contributions. Our contributions are threefold:

1. We identify and formalize *causality laundering*, a denial-feedback leakage pattern in which an adversary probes a protected action, infers security-relevant information from the denial outcome, and exfiltrates it through a subsequent tool call (Section 3).
2. We present ARM, a provenance-aware runtime enforcement layer that explicitly represents denied actions, counterfactual dependencies, transitive data dependencies, and field-level provenance, and enforces policy through deterministic graph traversal over an integrity lattice (Sections 4 and 5).
3. We evaluate ARM on three representative attack scenarios and show that the graph-aware engine blocks all three, namely causality laundering, transitive taint propagation, and mixed-provenance field misuse, while a flat provenance baseline misses all three, with sub-millisecond evaluation overhead (Section 7).

2 Threat Model

2.1 System and Trust Boundary

We consider a deployment in which a single LLM agent interacts with the external world exclusively through tool calls. ARM mediates this boundary: every attempted tool invocation is evaluated before execution, and every tool result is imported into the agent context only after policy evaluation and provenance recording. The primary security-relevant actions in scope are reads from protected resources, writes or updates to external systems, and outbound communication to potentially adversarial sinks.

The agent interacts with external systems through a set of tools exposed via the Model Context Protocol (MCP) or an equivalent tool-calling interface. The agent receives a system prompt from a trusted operator, user messages from a trusted end user, and tool outputs from external services of varying trustworthiness.

The enforcement point is the tool-call boundary: ARM interposes on every tool invocation before execution and on every tool result before it enters the agent’s context. The protected assets include confidential records (e.g., personnel files, credentials, financial data), and the privileged sinks include outbound communication (e.g., `send_email`, `post_message`), file mutation (e.g., `write_file`, `delete_file`), external API calls (e.g., `transfer_funds`, `update_record`), and code execution (e.g., `run_code`). A security violation occurs when a privileged sink is reached by an action whose provenance includes either untrusted data or denial-induced causal influence, without explicit policy authorization.

2.2 Attacker Model

The attacker acts indirectly by controlling content that the agent observes, not by modifying ARM itself. The attacker’s goal is to cause the agent to execute unauthorized tool calls or exfiltrate sensitive data through tool-call arguments. We assume the attacker *cannot* modify the agent’s code, the ARM enforcement layer, or the system prompt. The attacker *can*:

1. **Inject adversarial content into tool outputs.** A compromised or adversary-controlled tool returns data containing prompt injection payloads [12]. This is the dominant real-world attack vector: poisoned emails, malicious web content, and tampered API responses.
2. **Craft multi-step attack chains.** The attacker’s payload may not exfiltrate data in a single tool call. Instead, it may cause the agent to (a) read sensitive data, (b) transform it through intermediate tool calls to launder its provenance, and (c) exfiltrate the result through a seemingly benign tool call.
3. **Exploit denial signals.** When ARM denies a tool call, the agent observes the denial. A prompt-injected agent may infer information from the denial itself (e.g., “the file exists because the read was denied on permissions, not on file-not-found”) and exfiltrate this inference.
4. **Manipulate tool descriptions.** In MCP-based deployments, tool descriptions are metadata provided by the tool server and may be modified by a compromised or malicious server [14]. A manipulated description can influence the agent’s tool selection and argument construction.

2.3 Trust Sources

The system model above distinguishes trusted inputs (system prompt, user messages) from tool outputs of varying trustworthiness. We formalize this distinction as a total order over data sources:

Definition 1 (Integrity Lattice). *Let \mathcal{T} be the set of five trust levels with total order:*

$$\text{TOOLDESC} <_{\mathcal{T}} \text{TOOLUNTRUSTED} <_{\mathcal{T}} \text{TOOLTRUSTED} <_{\mathcal{T}} \text{USERINPUT} <_{\mathcal{T}} \text{SYSINSTR}$$

The trust level of a data item derived from multiple sources is the minimum of its sources’ trust levels (conservative join).

This lattice reflects operational reality: system instructions are the most trusted (set by the operator), user input is trusted but less authoritative, and tool outputs range from trusted (e.g., a signed API response) to untrusted (e.g., raw web scrape content). Tool descriptions—metadata provided by MCP servers—are the least trusted, as they can be manipulated in tool-poisoning attacks [14].

2.4 Security Goals

In practical terms, ARM aims to ensure that (1) no tool call influenced by untrusted data or denial-induced causal influence reaches a privileged sink without an explicit policy decision, (2) every enforcement decision is recorded in a way that resists after-the-fact tampering, and (3) no single enforcement check is a single point of failure. We formalize these as three properties:

1. **Mediated Integrity.** No causal path exists from an untrusted source to an ALLOW verdict on a privileged tool call without traversing at least one enforcement check (Theorem 1).
2. **Tamper-Evident Audit.** Every enforcement decision is recorded in a hash-chained audit log. Tampering with any entry invalidates the chain.
3. **Defense in Depth.** Multiple independent enforcement layers operate in series; an attacker must simultaneously evade all layers to succeed.

2.5 Scope and Non-Goals

ARM enforces *integrity* properties: it prevents untrusted data from triggering unauthorized actions. It provides *explicit secrecy* (blocking direct data flows to unauthorized sinks) but does *not* claim full non-interference for implicit flows (e.g., timing channels, control-flow inference). Full implicit-flow tracking is impractical in LLM-based systems where the model’s internal reasoning is opaque. However, ARM’s counterfactual edges (Section 3) mitigate a significant class of inference-based attacks that would otherwise be invisible.

We treat malicious-user attacks as out of scope and focus on indirect compromise through untrusted tool-observed content. We consider a single-agent setting in this paper. Multi-agent extensions (delegation tokens, vector clocks, composition checking) are discussed in Section 9 and deferred to future work.

3 Causality Laundering

Runtime enforcement systems for tool-calling agents often adopt a common implicit assumption: once a tool call is denied, the threat has been neutralized. The blocked action does not execute, no sensitive data is returned, and the system proceeds as though the attack has ended. We argue that this assumption is incorrect. A denied action is not merely a failed operation; it is an observable event whose outcome can influence the agent’s subsequent behavior. Denial feedback can therefore carry security-relevant information that is not captured by provenance models over successful executions alone. This section introduces *causality laundering*, a denial-feedback leakage pattern that exploits this gap, and develops the intuition and formalism needed to detect it.

3.1 Motivating Example

Consider an agent with access to `read_file` and `send_email` tools. A prompt injection payload instructs the agent to exfiltrate the contents of `/etc/shadow`. A policy engine correctly denies the `read_file("/etc/shadow")` call because the path matches a sensitive filesystem pattern.

A flat taint-tracking system considers the attack blocked: no sensitive data entered the agent’s context. However, the agent can still infer that the file *exists* if the denial reason is “permission denied” rather than “file not found.” The injected payload then instructs the agent to call `send_email(body="shadow file exists on this host")`. From the perspective of flat provenance or taint-based enforcement, this second call appears benign: the payload is short, contains no credential patterns, matches no sensitive data returned by any tool, and has no data-flow lineage connecting it to the denied read. Yet it conveys information derived from the denied action. Blocking the first step did not eliminate the attack; it left behind the information channel through which the second step operates.

3.2 Why Existing Defenses Miss the Attack

Three broad categories of runtime defense fail to detect the attack above, for structurally related reasons.

Flat taint and information-flow tracking. Systems such as FIDES [6] propagate integrity labels through tool outputs: when a tool call succeeds, its result inherits the trust level of its inputs. When a call is denied, no successful output is produced, so no label is propagated through the ordinary data-flow channel. The `send_email` call in the motivating example carries no tainted bytes, and flat taint correctly reports that no protected data flowed through a successful tool result. The leak is therefore invisible to this abstraction, because it does not travel through returned data; it travels through the agent’s observation of the denial outcome.

Dependency graphs over successful executions. Graph-based systems such as PCAS [21] construct dependency graphs over tool-call sequences and evaluate reachability policies against them. A denied call, however, is not typically represented as a provenance-bearing dependency node with downstream causal semantics: it produced no successful output for later nodes to depend on. As a result, the causal link from the denied `read_file` to the subsequent `send_email` is not captured in a graph that records only successful execution dependencies.

Causal attribution defenses. A separate line of work uses causal attribution to detect indirect prompt injection. AgentSentry [31] and CausalArmor [16] re-execute agent trajectories with and without suspected inputs to estimate causal influence. Their ablation targets are typically attacker-controlled content that appeared in tool *results*, such as injected text or manipulated API responses. Denial feedback, by contrast, is generated by the enforcement layer itself rather than by an external attacker, and existing attribution pipelines are not designed to model denied actions as explicit ablation targets with downstream provenance.

The missing abstraction. In each case, the denied action disappears from the defense’s world model. The agent, however, still observes the denial, and its subsequent behavior is conditioned on that observation. The missing object is therefore the denial event as a first-class provenance node: one that participates in downstream causal chains even though it produced no successful tool output. ARM targets a class of implicit-flow behavior that is not the focus of FIDES’s threat model. Causality laundering exploits precisely this gap.

3.3 Formal Definition

Definition 2 (Causality Laundering). *Let $G = (V, E)$ be the provenance graph of an agent session. A causality laundering attack occurs when:*

1. *An action $a_d \in V$ is denied by the enforcement layer at time t_d .*
2. *A subsequent action $a_s \in V$ at time $t_s > t_d$ produces an observable side effect (e.g., sends data to an external sink).*
3. *No successful data-flow path connects a_d to a_s through returned tool outputs.*
4. *The occurrence or content of a_s is causally influenced by the denial of a_d ; that is, the agent’s behavior would differ had a_d not occurred, or had its denial not been observed.*

Clauses 1 and 2 establish the temporal structure: a denied action precedes a side-effecting action. Clause 3 is what makes the attack invisible to existing data-flow defenses: because no successful data-flow path connects the two actions, systems that track only explicit propagation through tool results will see no dependency between them. Clause 4 is the distinguishing condition: the later action would have been different had the denial not occurred, meaning the denial itself carried information that shaped subsequent behavior even without a returned-data path. Together, clauses 3 and 4 capture the core tension: the causal influence is real, but the ordinary data-flow evidence is absent. In Section 3.7, ARM operationalizes this definition conservatively by recording denied actions explicitly and linking them to subsequent actions via counterfactual edges.

3.4 Relationship to Established Concepts

Causality laundering is not conceptually isolated; it is the tool-calling-agent manifestation of several classical ideas in information flow and causality.

Covert channels (Lampson, 1973). Lampson’s covert channel taxonomy [17] distinguishes storage channels (shared state) from timing channels (observable delays). Causality laundering is a *storage channel* in which the shared state is the agent’s observed execution history: the denial event is effectively “stored” in the agent’s state and later reflected in a subsequent action.

Counterfactual causality (Pearl, 2009). In Pearl’s structural causal model framework [23], a_d is a cause of a_s if, in the counterfactual world where a_d did not occur, a_s would not have occurred (or would have had different content). Our detection mechanism operationalizes a conservative approximation of this idea: ARM records a COUNTERFACTUAL edge from a_d to a_s and queries whether any path to a sink traverses such an edge.

Implicit information flows (Denning, 1976). Denning’s lattice model for information flow [9] acknowledges that assignments in conditional branches create implicit flows. In the agent setting, the LLM’s “conditional branch” is opaque: we cannot inspect the internal reasoning that transforms denial feedback into later behavior. The observable pattern, however, can still be represented and enforced over: a denied action followed by a suspicious downstream action.

3.5 Attack Variants

We identify three variants of causality laundering, distinguished by how denial-derived information is inferred, accumulated, and transmitted.

1. **Denial inference exfiltration.** A single denied probe leaks a small but security-relevant fact (e.g., that a file exists), and the agent immediately exfiltrates that inference. This is the motivating example above. The risk is that even one denial can reveal the presence, absence, or sensitivity class of a protected resource, which may already be sufficient for reconnaissance or targeted follow-up attacks.
2. **Multi-probe fingerprinting.** The attacker issues a sequence of probes (e.g., testing multiple file paths or API endpoints), collects a vector of allow/deny outcomes, and exfiltrates the aggregate. Each individual denial reveals little in isolation; the sequence reveals much. The risk is cumulative: a systematic scan of n resources can leak enough denial feedback to fingerprint the system’s access-control configuration or map protected assets.
3. **Laundered composition.** The attacker inserts an allowed intermediate tool between the denial and the final sink-bound action in order to obscure the connection. For example: probe \rightarrow denied \rightarrow use a calculator tool to encode the denial status as a number \rightarrow include that number in an outbound message. The risk is that the intermediate transformation launders the denial-derived signal into a form that breaks simple temporal, textual, or surface-level correlation with the original denied action.

3.6 Detection Intuition

If the problem is that denied actions are absent from the provenance model, the natural defense is to record them explicitly and propagate their potential downstream influence. The challenge is that the path from a denied action to a later tool call passes through the LLM’s internal state, which is opaque to the enforcement layer. We therefore cannot directly prove that a particular subsequent action was caused by a particular denial. What we *can* observe is the external pattern: a denial occurred, the agent continued, and later tool calls were issued under a context that now includes that denial outcome.

ARM’s detection strategy accepts this opacity and works with what is observable. Rather than attempting to determine *whether* the model’s internal reasoning was influenced, ARM records *that* a denial occurred and conservatively treats subsequent tool calls as potentially influenced by that event. This is a deliberate over-approximation: it may flag benign tool calls that happen to follow a denial, but it makes denial-induced causal influence explicit in the provenance model instead of leaving it unrepresented. The next subsection operationalizes this intuition using denied-action nodes and counterfactual edges in the provenance graph.

3.7 Detection via Counterfactual Edges

ARM detects causality laundering by maintaining COUNTERFACTUAL edges in the provenance graph. When an action is denied, ARM records a DENIEDACTION node. The next tool call automatically receives a COUNTERFACTUAL edge from that denial node. This is a temporal heuristic, chosen because the model’s internal reasoning is opaque: we cannot determine exactly which later actions were influenced by the denial, so we conservatively mark the temporally adjacent action as potentially influenced.

At evaluation time, ARM’s graph-aware provenance layer (Section 5) queries whether the current tool call is reachable from any denied-action node via a path containing a COUNTERFACTUAL edge. If so, the call is denied with the reason “causality laundering detected.”

This mechanism is a conservative approximation of causal influence, not perfect causal identification. It may therefore flag benign tool calls that happen to follow a denial, producing false positives. We discuss this trade-off and possible refinements, such as configurable temporal windows and richer structural conditions, in Section 9. In practice, prompt-injection exfiltration attempts often follow closely after the denied probe, making this heuristic effective for the dominant probe-then-exfiltrate pattern.

4 ARM Architecture

ARM is an interposition layer that sits between a tool-calling LLM agent and its upstream tool servers. Every tool call is mediated by ARM before execution, and every tool result passes back through ARM before entering the agent’s context. The outcome of each mediation is either ALLOW, meaning the call proceeds, or DENY, meaning the call is blocked and a denial record is created. No tool invocation reaches an external system without first passing through the enforcement pipeline described in this section. While we describe ARM as an MCP proxy for concreteness, the same mediation pattern applies to equivalent tool-calling runtimes that expose a centralized execution boundary.

4.1 Interposition and Execution Flow

The end-to-end path for a single tool call proceeds as follows. The LLM client issues a `tool_use` message. ARM, operating as an MCP proxy server, receives the message and evaluates it through the layered policy pipeline (Section 4.3). If all layers return PASS, ARM forwards the call to the upstream tool server, receives the result, records the returned data and its provenance (Section 5), and returns the result to the agent. If any layer returns DENY, the call is not forwarded; instead, ARM records a `DENIEDACTION` node in the provenance graph and returns an error to the agent. In both cases, the enforcement decision and its per-layer results are written to the audit log.

4.2 Reference Monitor Properties

ARM achieves the three classical reference monitor properties [2]:

1. **Complete mediation.** ARM operates as the sole MCP proxy between the LLM client and the upstream tool server. Every `tool_use` message passes through ARM’s `evaluate()` function before reaching the real tool. Under the deployment model in Section 4.1, there is no alternative path.
2. **Tamper-proofness.** The enforcement layer runs in a separate process from the LLM. The agent cannot modify policy rules, capability tokens, or the provenance graph. Capability tokens are immutable (`frozen` dataclasses), and the audit log is hash-chained.
3. **Verifiability.** The policy pipeline consists of four layers with well-defined, deterministic semantics (described below). Each layer’s behavior can be analyzed and tested in isolation.

4.3 Layered Policy Pipeline

ARM evaluates each tool call through an ordered sequence of policy layers. The layers are arranged from unconditional boundary checks to provenance-aware and operator-defined policy checks. Each layer returns one of two verdicts:

- PASS: the layer has no objection (evaluation continues to the next layer).
- DENY: the layer blocks the call (evaluation halts immediately).

The composition rule is strict: DENY from any layer terminates evaluation. Only if all layers return PASS is the call forwarded to the upstream tool.

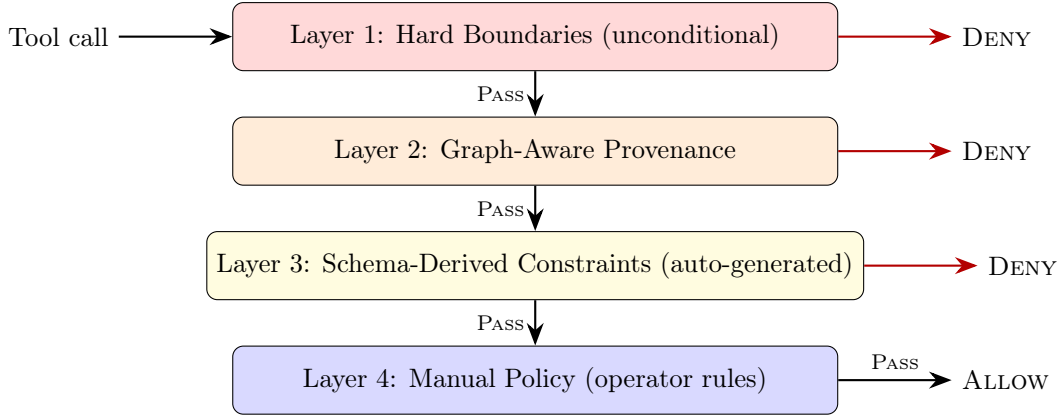


Figure 1: The ARM layered policy pipeline. Each layer can independently deny a tool call.

4.3.1 Layer 1: Hard Boundaries

Layer 1 captures unconditional boundary checks that do not depend on session provenance or operator policy. It enforces five invariants:

HB-1: Payload size limit. Any string argument exceeding 10,000 characters is denied (bulk exfiltration defense).

HB-2: Sensitive path blocking. File-path arguments matching protected patterns (`~/.ssh/*`, `/etc/shadow`, `~/.aws/*`) are denied.

HB-3: Call-rate limit. Any tool called more than 100 times per session is denied (runaway agent defense).

HB-4: Credential detection. Arguments containing patterns matching known credential formats (AWS access keys, GitHub tokens, JWTs, RSA private keys, etc.) are denied.

HB-5: Schema pinning. The SHA-256 hash of each tool’s input schema is recorded on first invocation. Subsequent calls with a changed schema are denied (rug-pull attack defense [14]).

4.3.2 Layer 2: Graph-Aware Provenance

Layer 2 queries the provenance graph to determine whether the current tool call’s inputs can be traced back to trusted sources. It performs two checks: trust propagation, which computes the minimum trust level across all ancestor nodes of the current call, and counterfactual chain detection, which checks whether any path from a `DENIEDACTION` node reaches the current call via a `COUNTERFACTUAL` edge. If the minimum reachable trust falls below the configured threshold, or if a denial-induced counterfactual path reaches the current call, Layer 2 returns `DENY`. The formal semantics of the provenance graph, including node types, edge types, and the trust propagation algorithm, are defined in Section 5.

4.3.3 Layer 3: Schema-Derived Constraints

Layer 3 derives lightweight defensive constraints automatically from tool input schemas, reducing reliance on manual policy specification. It applies four rules:

SD-1: Suspicious parameter names. Parameters whose names suggest free-text fields (e.g., `note`, `comment`, `metadata`, `description`) receive a 100-character length limit and credential-pattern checks, as these are common exfiltration side channels.

SD-2: Filesystem parameters. Parameters named `path`, `file`, `directory` receive sensitive-path blocking.

SD-3: Optional parameters. Optional string parameters not listed in the schema’s `required` array receive a 50-character length limit (optional parameters are the most common exfiltration vector).

SD-4: Network parameters. Parameters named `url`, `endpoint`, `host` are flagged for logging (enforcement deferred to future work).

4.3.4 Layer 4: Manual Policy

Layer 4 enforces operator-defined rules specified in an immutable capability token. Rules include per-tool allow/deny, call budgets, argument blocklists, and value constraints. This layer provides the traditional capability-based access control that operators expect. Layer 4 evaluates these rules against the capability token bound to the current session.

4.4 Capability Tokens

Each agent session is bound to an immutable *capability token* that specifies permitted tools and their constraints. Tokens are frozen dataclasses: the agent cannot modify them at runtime. This design draws on capability-based security [10] and Macaroons [4], where capabilities can be attenuated (restricted) but never amplified. Layer 4 evaluates operator-defined rules against the capability token bound to the current session.

4.5 Audit Log

Every enforcement decision (ALLOW or DENY) is recorded in a hash-chained audit log. Each entry contains the tool name, arguments, decision, reason, and per-layer results. The hash chain provides tamper evidence: modifying any entry invalidates all subsequent hashes. The audit log therefore captures both successful and denied tool invocations at the same enforcement boundary. This supports forensic analysis and compliance requirements.

5 Provenance Graph

The provenance graph is the core data structure behind ARM’s graph-aware enforcement. Its purpose is not merely to log events, but to represent the causal structure that flat provenance models miss: successful data flow, field-level provenance, and denial-induced downstream influence. A single label attached to a tool result cannot distinguish which field carried taint, and a trace of successful calls alone cannot represent the effect of a denied action. The provenance graph provides a single representation over successful outputs, structured fields, and denied actions, allowing Layer 2 to reason over both explicit data dependencies and counterfactual links introduced by denials.

5.1 Graph Structure

Definition 3 (Provenance Graph). *A provenance graph $G = (V, E, \tau, \ell)$ consists of:*

- *A set of nodes V partitioned into four disjoint subsets:*

$$V = V_{CALL} \cup V_{DATA} \cup V_{DATAFIELD} \cup V_{DENIEDACTION},$$

representing tool calls, returned data items, sub-items of structured results, and denied tool calls, respectively.

- A set of directed edges $E \subseteq V \times V$ with labels from $\{\text{DIRECTOUTPUT}, \text{INPUTTO}, \text{COUNTERFACTUAL}, \text{FIELDOF}\}$.
- A trust assignment $\tau : V_{\text{DATA}} \cup V_{\text{DATAFIELD}} \rightarrow \mathcal{T}$ mapping data nodes to trust levels from the integrity lattice (Definition 1).
- A label function ℓ assigning metadata such as tool names, arguments, timestamps, and denial reasons to each node.

5.2 Edge Semantics

Each edge type captures a specific provenance or causal relationship. All edges are directed; the notation $\text{LABEL}(u, v)$ denotes an edge from u to v .

DIRECTOUTPUT(c, d): Tool call $c \in V_{\text{CALL}}$ produced data item $d \in V_{\text{DATA}}$. This is the primary data-flow edge: it records which call generated which output.

INPUTTO(d, c): Data item $d \in V_{\text{DATA}}$ was used as an argument to tool call $c \in V_{\text{CALL}}$. Together, **DIRECTOUTPUT** and **INPUTTO** define transitive provenance chains. For example, if

$$c_1 \xrightarrow{\text{DIRECTOUTPUT}} d_1 \xrightarrow{\text{INPUTTO}} c_2 \xrightarrow{\text{DIRECTOUTPUT}} d_2 \xrightarrow{\text{INPUTTO}} c_3,$$

then c_3 is transitively influenced by the outputs of c_1 .

FIELDOF(f, d): Field node $f \in V_{\text{DATAFIELD}}$ is a component of structured data item $d \in V_{\text{DATA}}$. This enables field-level trust assignment: given a tool result with fields **name** and **email**, the integrator can assign **TOOLTRUSTED** to **name** and **TOOLUNTRUSTED** to **email** independently.

COUNTERFACTUAL(a_d, c): Denied action $a_d \in V_{\text{DENIEDACTION}}$ is linked to subsequent tool call $c \in V_{\text{CALL}}$ by a counterfactual-dependence edge. This edge records the potential causal influence of a denial on later behavior and is the mechanism for detecting causality laundering (Section 3).

5.3 Trust Propagation

The provenance graph enables a conservative trust computation: a tool call's effective trust level is the minimum trust of any data ancestor in its causal history. One untrusted ancestor anywhere in the chain is sufficient to lower the trust of every downstream node. This is why the graph matters operationally: flat labels attached only to immediate tool results cannot capture transitive taint that propagates across multiple tool calls.

Definition 4 (Minimum Reachable Trust). *For a node $v \in V$, define:*

$$\text{MINTRUST}(v) = \min_{u \in \text{ANCESTORS}(v) \cap (V_{\text{DATA}} \cup V_{\text{DATAFIELD}})} \tau(u)$$

where $\text{ANCESTORS}(v)$ is the set of all nodes reachable from v by following edges in reverse. If $\text{ANCESTORS}(v)$ contains no data nodes, then $\text{MINTRUST}(v) = \text{SYSINSTR}$.

This is a worst-case lower bound: if *any* data ancestor is untrusted, the effective trust of the current node is bounded by that ancestor's level. The computation is analogous to taint propagation in dynamic taint analysis [27], but it operates over a causal provenance graph rather than instruction-level data flow.

Property 1 (Monotonic Taint). *For any edge $(u, v) \in E$,*

$$\text{MINTRUST}(v) \leq \text{MINTRUST}(u).$$

Algorithm 1 Pre-Execution Graph-Aware Tool Call Evaluation

Require: Tool name t , arguments \mathbf{a} , input data IDs D_{in}

Ensure: Policy decision $\delta \in \{\text{ALLOW}, \text{DENY}\}$

```
1:  $c \leftarrow \text{NEWNODE}(V_{\text{CALL}}, t, \mathbf{a})$  ▷ Create call node
2: for  $d \in D_{\text{in}}$  do
3:    $\text{ADDEDGE}(d, c, \text{INPUTTO})$  ▷ Link input data
4: end for
5: if  $\exists a_d \in V_{\text{DENIEDACTION}}$  from preceding denial then
6:    $\text{ADDEDGE}(a_d, c, \text{COUNTERFACTUAL})$  ▷ Auto-link
7: end if
8:  $\delta \leftarrow \text{EVALUATEPIPELINE}(t, \mathbf{a}, c)$  ▷ L1 + L2G + L3 + L4
9: if  $\delta = \text{DENY}$  then
10:   $a_d \leftarrow \text{NEWNODE}(V_{\text{DENIEDACTION}}, t, \mathbf{a}, \delta.\text{reason})$ 
11: end if
12: return  $\delta$ 
```

5.4 Enforcement Queries

The graph semantics above support two enforcement queries, corresponding to the two checks performed by Layer 2 (Section 4.3.2) when deciding whether to block a tool call c :

1. $\text{MINTRUST}(c) < \theta$, where θ is the minimum required trust level (default: TOOLTRUSTED). This catches transitive taint chains.
2. $\text{COUNTERFACTUALCHAINS}(c) \neq \emptyset$, where $\text{COUNTERFACTUALCHAINS}(c)$ denotes the set of paths to c containing at least one COUNTERFACTUAL edge. This catches causality laundering.

Both queries are implemented via graph traversal on a **rustworkx** backend [24], achieving sub-millisecond latency for typical session graphs (tens to hundreds of nodes).

5.5 Graph Construction

The graph is constructed incrementally as the agent session progresses. ARM’s **GraphAwareEngine** orchestrates construction at two points: before tool execution, when the call node and its input dependencies are materialized, and after successful execution, when returned outputs are added to the graph. Algorithm 1 shows the pre-execution evaluation path.

If the call is allowed and the upstream tool returns successfully, ARM creates one or more V_{DATA} nodes for the returned output, adds DIRECTOUTPUT edges from the call node to those data nodes, and, if the output is structured, optionally materializes $V_{\text{DATAFIELD}}$ nodes linked by FIELD OF edges. Trust labels are assigned at this point according to the source trust level and any field-level overrides.

5.6 Field-Level Provenance

Structured tool outputs (JSON objects, database rows) often contain fields with heterogeneous trust levels. ARM decomposes such outputs into DATAFIELD nodes with independent trust assignments:

Definition 5 (Field-Level Trust Override). *Given a data item d with structured value $\{k_1 : v_1, \dots, k_n : v_n\}$ and a trust override map $\omega : K \rightarrow \mathcal{T}$, the trust of field f_i is:*

$$\tau(f_i) = \begin{cases} \omega(k_i) & \text{if } k_i \in \text{dom}(\omega) \\ \tau(d) & \text{otherwise} \end{cases}$$

Table 1: Implementation modules and sizes.

Module	Component	Lines
arm_core	PolicyEngine, LayeredPolicyEngine	~130
	CapabilityToken, ToolPermission	~50
	L1: HardBoundariesLayer	~120
	L2: ProvenanceLayer (flat baseline)	~110
	L3: SchemaDerivedLayer	~80
	L4: ManualPolicyLayer	~60
	AuditLog	~60
	MCP Proxy	~50
arm_provenance	ProvenanceGraph	~430
	GraphProvenanceLayer (L2G)	~60
	GraphAwareEngine	~110
Total enforcement code		~910
Test code		~600

This enables fine-grained enforcement: a contact record’s `name` field may be trusted while its `email` field is untrusted (e.g., externally provided). A subsequent `send_email(to=email)` call would be denied because the `email` field’s taint propagates through the graph.

6 Implementation

We implemented ARM as a Python prototype to validate that the architecture described in Sections 4 and 5 can be realized with low overhead and integrated into existing agent frameworks without modifying the LLM, the tool servers, or the prompt templates. The implementation is split into two modules: **arm_core**, which contains the control-plane enforcement pipeline (policy layers, capability tokens, audit log, MCP proxy), and **arm_provenance**, which contains the provenance graph engine and the graph-aware enforcement layer. The total enforcement codebase is approximately 910 lines of Python, which keeps the trusted enforcement path relatively compact.

6.1 Module Architecture

Most of the implementation complexity lies in the provenance subsystem (**arm_provenance**), reflecting that the implementation complexity is concentrated in graph construction and graph-aware enforcement rather than in the policy pipeline itself. The flat baseline provenance layer is retained only to support the comparative evaluation in Section 7.

6.2 Key Design Decisions

Deterministic over probabilistic. Every enforcement decision in ARM is deterministic. The graph-aware provenance layer (L2G) queries the provenance graph directly, with no LLM calls in the enforcement loop. This is a deliberate design choice: a defense that relies on the same LLM under attack to validate provenance or policy compliance inherits that model’s compromise surface.

rustworkx for graph operations. The provenance graph is backed by **rustworkx** [24], a Rust-implemented graph library with Python bindings. Reachability queries are implemented

on this backend, achieving sub-millisecond latency for session-sized graphs. This ensures ARM introduces negligible overhead compared to LLM inference time (typically 100 ms to 10 s).

MCP protocol-level interposition. ARM operates as an MCP proxy server: it presents the same tool schemas to the LLM client while intercepting every `tool_use` request. This achieves complete mediation without modifying the LLM, the MCP client library, or the upstream tool servers. While our prototype uses MCP for concreteness, the same interposition pattern applies to equivalent tool-calling runtimes with a centralized execution boundary.

Immutable capability tokens. Capability tokens are implemented as Python frozen data-classes. Once created, neither the agent nor any tool can modify a token’s permissions. This provides a strong static guarantee: the agent’s capability surface is fixed at session initialization.

Hash-chained audit. Each audit entry includes the SHA-256 hash of the previous entry. Chain verification detects any post-hoc modification, supporting forensic analysis and regulatory compliance.

6.3 Integration Points

ARM is designed as a drop-in enforcement layer for existing agent frameworks. In practice, it assumes only that tool invocations pass through a centralized execution boundary and that successful tool results can be recorded after execution. Integration requires:

1. For MCP-based agents, routing tool calls through ARM’s MCP proxy. For programmatic agents, calling `engine.evaluate()` before each tool execution.
2. Providing an `EvaluationContext` with user messages, tool-call history, tool schemas, and capability token.
3. For graph-aware enforcement: calling `engine.record_tool_result()` after each allowed tool call to update the provenance graph.

This integration does not require changes to the LLM, the prompt templates, or the upstream tool implementations.

7 Evaluation

The central claim of this paper is that graph-aware provenance tracking catches security-relevant attacks that a flat provenance baseline misses, and that the added expressiveness does not impose meaningful runtime cost. We evaluate this claim through a controlled differential experiment: three attack scenarios are executed against two engine configurations that differ *only* in the provenance layer (L2 vs. L2G), with all other layers held constant. The evaluation is scenario-based and targets the prototype implementation; we explicitly scope what it does and does not establish in Section 7.6.

7.1 Experimental Setup

Each scenario is executed against two engine configurations:

- **Flat baseline:** `LayeredPolicyEngine` with the citation-based provenance layer (L2). L2 checks whether each argument value appears somewhere in the agent’s tool-call history; if the argument can be grounded in prior tool-call history, the provenance check passes.

- **Graph-aware:** `GraphAwareEngine` with the graph-aware provenance layer (L2G). L2G queries the provenance graph for trust propagation, counterfactual edges, and field-level provenance.

Both configurations share identical L1 (hard boundary), L3 (schema-derived), and L4 (manual policy) layers. Because only the provenance mechanism differs, any divergence in outcome is attributable to the graph-aware representation.

A scenario is *successfully defended* if the final sink-bound action is denied before execution. The three scenarios were selected to isolate the three structural capabilities that the graph-aware model adds beyond the flat baseline: counterfactual-edge detection, transitive trust propagation, and field-level provenance. Each scenario is constructed so that one of these capabilities is the decisive difference between the two engines.

7.2 Attack Scenarios

Each of the following scenarios isolates one structural advantage of the graph-aware model over the flat baseline.

7.2.1 Attack 1: Causality Laundering (Denial Inference Exfiltration)

The attacker’s payload instructs the agent to:

1. Call `read_file("/etc/shadow")` (denied by L1, HB-2).
2. Call `send_email(body="shadow file exists")`.

The second call contains no credential patterns, no sensitive paths, and its 14-character body passes all length checks. The string does not appear in any tool output; it is composed by the LLM from the denial signal.

- **Flat result: ALLOW.** L2 finds no provenance violation, because the sink-bound string has no explicit returned-data lineage and does not violate any non-provenance boundary checks.
- **Graph result: DENY.** L2G detects a COUNTERFACTUAL edge from the denied `read_file` to the `send_email` call. Reason: “causality laundering detected.”

This scenario isolates the value of denial-aware counterfactual tracking.

7.2.2 Attack 2: Transitive Taint Chain

The attacker’s payload instructs the agent to:

1. Call `read_emails()` → returns `{body: "Wire $50k to account XXX"}` (untrusted).
2. Call `format_response(text="Summary: Wire transfer...")` to transform the content.
3. Call `send_email(body=...)` to `finance@company.com` with the formatted text.

The intermediate `format_response` call launders the data’s provenance: its output appears in the agent’s tool-call history as an allowed tool result, obscuring the earlier untrusted origin.

- **Flat result: ALLOW.** L2 finds `format_response` output in the tool-call history and the citation check passes. The flat layer has no mechanism to look beyond the immediate source; the transitive origin in `read_emails` is invisible.

Table 2: Security evaluation: flat (L2 citation) vs. graph-aware (L2G) enforcement. ✓ = attack blocked, ✗ = attack missed.

Attack	Flat (L2)	Graph (L2G)	Root Cause
Causality laundering	✗	✓	Counterfactual edge
Transitive taint chain	✗	✓	Graph reachability
Mixed-provenance fields	✗	✓	Field-level provenance

- **Graph result: DENY.** L2G traces the provenance graph: `send_email` depends on `format_response`, which depends on `read_emails` (untrusted). The resulting MINTRUST is TOOLUNTRUSTED, below the threshold.

This scenario isolates the value of transitive trust propagation through the provenance graph.

7.2.3 Attack 3: Mixed-Provenance Field Exploit

A tool returns a structured result whose fields do not share the same trust level:

1. Call `get_contact()` → returns `{name: "Alice", email: "attacker@evil.com"}`. The `name` field is trusted; the `email` field is marked untrusted (externally provided).
 2. Call `send_email(to="attacker@evil.com")`.
- **Flat result: ALLOW.** L2 finds "attacker@evil.com" in the tool-call history (from `get_contact`), so the citation check passes. The flat layer treats the returned object as a single unit and cannot distinguish trusted fields from untrusted ones within the same value.
 - **Graph result: DENY.** L2G resolves the `email` field node individually, finding trust TOOLUNTRUSTED. The `name` field, by contrast, would be allowed independently.

This scenario isolates the value of field-level provenance within structured tool outputs.

7.3 Results

Table 2 summarizes the results. Because all non-provenance layers (L1, L3, L4) are held constant across configurations, the difference in outcomes is attributable solely to the provenance mechanism.

The flat baseline misses all three attacks because it verifies only that a value *appeared* in the agent’s history, not *how it arrived there* or *which specific field* contributed it. The graph-aware engine blocks all three by exploiting structural properties of the provenance graph: counterfactual edges (Attack 1), transitive reachability (Attack 2), and field-level provenance (Attack 3). Each blocked attack corresponds to a capability that the flat representation does not capture.

7.4 Performance

On the prototype implementation, the complete evaluation pipeline (L1 + L2G + L3 + L4) executes in under 1 ms for all three scenarios (Apple M4 Pro, 48 GB RAM; median of 100 runs per scenario after warm-up). For context, a single LLM inference call typically takes 100 ms to several seconds; the enforcement overhead is therefore negligible relative to the latency already present in any tool-calling agent loop. These measurements reflect session-sized graphs (tens of tool calls, hundreds of data nodes) and should not be extrapolated to arbitrarily large provenance histories without further benchmarking.

Memory scales linearly with session length: a typical session produces a graph with $O(100)$ nodes, well within practical limits. Reachability queries run in $O(V + E)$ time via the `rustworkx` backend.

7.5 Test Suite

In addition to the scenario-based evaluation above, ARM includes 45 unit tests covering:

- Provenance graph construction, trust lattice, reachability, and counterfactual edges (34 tests).
- Graph-aware engine end-to-end scenarios: legitimate calls, denial inference, field-level taint, and composition with L1 (11 tests).

All tests are deterministic and execute without network access or LLM invocation, enabling CI integration.

7.6 Limitations

The flat baseline is intentionally a minimal flat-history provenance mechanism designed to isolate the representational contribution of the graph-aware model; it is not a reimplementaion of FIDES, PCAS, or other full systems. The current evaluation uses manually constructed attack scenarios rather than an automated benchmark suite (e.g., AgentDojo [7]). While the three scenarios are representative of documented attack patterns, a larger-scale evaluation is needed to assess false-positive rates, coverage across diverse agentic workflows, and performance under longer provenance histories. We also do not yet report benchmark-scale end-to-end experiments with frontier LLMs in the loop. We discuss these limitations and corresponding future work in Section 9.

8 Related Work

Closest prior work on runtime security for tool-calling agents spans information-flow control, graph-based dependency enforcement, secure-by-design agent architectures, and causal defenses against indirect prompt injection. FIDES [6], PCAS [21], and CaMeL [8] are the closest prior systems to ARM, each on a different axis. ARM does not claim to be the first runtime security mechanism for LLM agents, nor the first graph-based or causal defense. Its contribution is narrower: it makes denied tool invocations first-class provenance events and enforces over their downstream influence through counterfactual edges, enabling detection of denial-feedback leakage patterns that are not naturally captured by successful-flow provenance, extracted control/data flow, replay-based causal influence, or structured runtime traces.

Deterministic runtime enforcement. FIDES and PCAS are the two closest prior systems to ARM, but on different axes. FIDES [6] applies information-flow control to AI agents, propagating confidentiality and integrity labels over observed flows and enforcing policy deterministically. Both FIDES and ARM aim for deterministic runtime enforcement over agent behavior, but FIDES tracks labels over successful flows, whereas ARM additionally represents denied actions and enforces over their downstream influence. PCAS [21] compiles declarative authorization policies into an instrumented agent runtime that enforces them over a dependency graph via reference-monitor interposition. PCAS is the closest prior work to ARM on the graph-based enforcement substrate: it provides deterministic, graph-based policy enforcement without requiring security-specific architectural restructuring. The key distinction is that neither system explicitly models denied actions as provenance-bearing events or enforces over their downstream counterfactual influence. FIDES propagates labels over data that successfully flowed through the

agent; PCAS enforces policies over dependency relationships among completed actions. ARM extends provenance with denied-action nodes and counterfactual edges so that denial-feedback leakage (where the LLM infers sensitive information from the *absence* of a successful result) becomes enforceable. That distinction is central to ARM’s treatment of causality laundering (Section 3).

Secure-by-design architectures. CaMeL [8] takes a complementary approach: it extracts trusted control and data flow from the user query, prevents untrusted observations from steering execution, and uses capabilities to restrict unauthorized data exfiltration. CaMeL’s strength is architectural prevention; it changes how the agent is built. ARM occupies a different design point: it assumes a conventional tool-calling agent and hardens the execution boundary through provenance-aware runtime mediation. CaMeL does not center denied-action provenance or field-level provenance within structured tool outputs, while ARM does not require a planner/executor split or a new programming model. The two approaches are complementary; an agent built with CaMeL’s architecture could additionally deploy ARM at the tool boundary for defense in depth.

Causal defenses against indirect prompt injection. A growing line of work applies causal reasoning to detect and mitigate indirect prompt injection. AgentSentry [31] models multi-turn indirect prompt injection as temporal causal takeover, localizes takeover points through controlled counterfactual re-execution at tool-return boundaries, and mitigates attacks via context purification. CausalArmor [16] uses leave-one-out ablation to attribute privileged actions to user intent versus untrusted observations and triggers sanitization only when an untrusted segment dominates. Both systems reason about *why* an action was taken, rather than only whether an input appears suspicious; however, they do so through replay or attribution at inference time, requiring additional LLM calls to diagnose causal influence. ARM is complementary to these approaches. Rather than replaying the agent or estimating causal influence online, ARM records a conservative approximation of denial-induced influence directly inside the provenance graph through denied-action nodes and counterfactual edges. This keeps enforcement deterministic and lightweight, with no additional LLM calls, but also makes it more conservative: ARM may flag actions that a replay-based system would clear after re-execution. The trade-off is between precision (replay) and guaranteed low-latency deterministic enforcement (graph-native provenance). Recent evidence that frontier agents can infer hidden monitoring from blocking feedback alone [15] further supports the practical relevance of denial-feedback channels as an attack surface.

Structured runtime analysis. AgentArmor [29] also rejects flat history-based analysis and instead converts runtime traces into structured program-dependence representations (control-flow graphs, data-flow graphs, and program-dependence graphs), then applies type-system-style analysis to detect unsafe behavior. ARM shares the motivation of moving beyond flat message histories into graph-structured reasoning, but is narrower and more enforcement-oriented: ARM’s provenance graph is defined over tool calls, tool outputs, structured data fields, and denied invocations, and its queries are designed for real-time enforcement decisions rather than post-hoc program analysis.

Adjacent runtime governance and complementary defenses. Several recent systems address adjacent aspects of agent runtime security. RTBAS [32] adapts information-flow control to tool-based agents with a mix of automatic safe execution and human confirmation. Tool-Safe [19] provides step-level pre-execution safety detection. AgentGuardian [1] learns context-aware access-control policies from execution traces. PRISM [18] provides a defense-in-depth runtime layer for deployable agent gateways, and OPP [33] proposes protocol-level governance

for agent-to-tool communication. At the prompt level, Spotlighing [13] and instruction hierarchy [28] harden the LLM against prompt injection, while NeMo Guardrails [20] provides configurable input/output filtering. These are complementary to ARM: they reduce injection probability or restrict tool access, while ARM limits damage when injection succeeds by enforcing provenance invariants at the execution boundary.

Foundations. ARM draws on classical work in reference monitors, information-flow control, and causality. Anderson’s reference monitor concept [2] and Saltzer and Schroeder’s design principles [25] provide the model for complete mediation and tamper-resistant enforcement. Denning’s lattice model [9] provides the theoretical basis for ARM’s integrity lattice and trust ordering. Pearl’s counterfactual causality framework [23] clarifies why denial feedback can act as an implicit information channel: an agent that observes a denied action gains information about the protected resource, and that information can causally influence subsequent actions. Database provenance [5, 11] motivates graph-based dependency tracking at a finer granularity than message histories. ARM’s main conceptual extension over these foundations is to treat denied tool invocations as provenance-bearing events whose downstream consequences are themselves policy-relevant.

9 Discussion and Future Work

9.1 What ARM Does and Does Not Claim

ARM is not a complete solution to agent security, nor a general causal attribution engine. Its contribution is narrower: it is a deterministic execution-boundary monitor that enriches runtime provenance with denied actions and counterfactual edges, enabling enforcement over a denial-feedback leakage channel that successful-flow provenance alone does not naturally capture.

Accordingly, our formal and empirical claims should be read at that level of specificity. Theorem 1 is supported by the prototype implementation in the following sense: under the deployment model, every tool call traverses the layered enforcement pipeline, and graph-aware provenance blocks the evaluated attack patterns before execution. However, the complete-mediation property depends on correct deployment, namely routing all tool calls through the proxy. This is an operational assumption, not a property that code alone can guarantee. More precisely, Theorem 1 formalizes mediated enforcement under the deployment model; it does not establish completeness for all implicit-flow or denial-feedback attacks.

Two additional limitations follow directly from the current mechanism. First, the counterfactual heuristic is a sufficient condition for detecting the dominant probe-then-exfiltrate pattern, not a necessary condition for detecting all denial-feedback leakage. Second, field-level protection depends on integrator-provided trust overrides for structured tool outputs. If all fields inherit the parent object’s trust level, field-level provenance adds no security beyond object-level labeling.

9.2 Counterfactual Heuristic: False Positives and Missed Chains

ARM links a denied action to the immediately following tool call via a counterfactual edge. This heuristic is intentionally conservative because the LLM’s internal reasoning is opaque: the enforcement layer cannot directly observe whether a later action was mentally conditioned on the denial, so it approximates causal influence using temporal adjacency.

This approximation has two consequences. It can over-approximate by flagging a benign tool call that happens to follow a denial, and it can under-approximate by missing delayed or multi-step laundering chains in which denial-derived information is transformed across several subsequent actions before reaching a sink. We therefore do not claim completeness for denial-feedback leakage detection.

The case for the heuristic is pragmatic rather than semantic. First, it makes a previously unrepresented attack surface, denial-induced influence, explicit in the provenance model. Second, prompt-injection exfiltration often exhibits temporal locality: probe and exfiltration tend to occur close together in the same tool-use trajectory. A more precise alternative would require structural causal analysis [23] or selective replay, but that would sacrifice the current design’s deterministic, low-latency enforcement loop.

9.3 Trust Assignment and Declassification

ARM’s trust propagation is monotone by design. A tool call inherits the minimum trust level of any reachable data ancestor, which makes the mechanism a worst-case lower bound over provenance rather than a semantic judgment about content quality. This conservative design is attractive because it is simple, deterministic, and easy to audit, but it also shifts burden onto trust assignment at ingestion time.

This matters most for structured outputs. Field-level provenance is useful only when the integrator provides meaningful per-field overrides. If all fields inherit the parent trust level, then field-level provenance collapses back to ordinary object-level tainting. In that sense, the mechanism is only as good as the trust labels attached to tool outputs.

A related open problem is declassification. Real workflows may require a trusted validator or sanitization stage that upgrades a value’s trust after explicit checking. Supporting this safely is difficult because the declassifier must itself be outside the LLM trust boundary and resistant to manipulation. One promising direction is to treat declassification as an auditable capability (inspired by Macaroons [4]), bound to specific values or transformations rather than granted globally.

9.4 Evaluation Scope

Our evaluation is a controlled differential experiment, not a benchmark-scale security evaluation. The three scenarios were chosen to isolate the exact representational advantages of the graph-aware model over the flat baseline: denied-action provenance, transitive taint propagation, and field-level provenance. This is sufficient to validate the paper’s central claim that these structural features matter, but it does not establish false-positive rates, coverage across diverse agent workflows, or behavior under long provenance histories.

A stronger evaluation should include benchmark suites such as AgentDojo [7] and InjectAgent [30], longer-running trajectories, and experiments with frontier tool-calling agents in the loop. Those studies are important future work, but they would extend rather than replace the present controlled comparison.

9.5 Multi-Agent Extension

The current paper studies a single-agent setting. Extending the provenance graph to multi-agent workflows is plausible but non-trivial because delegation, concurrency, and composition introduce new causal relationships that are absent in a single-agent session. Useful building blocks include delegation tokens with monotonic attenuation, causal ordering mechanisms such as vector clocks, and composition checks that verify whether agent-local guarantees survive across a workflow of interacting agents.

We view that extension as a follow-up problem rather than part of the present claim. The contribution of this paper is the denied-action provenance mechanism in the single-agent case; multi-agent composition should be evaluated on its own terms.

10 Conclusion

Tool-calling LLM agents can read private data, invoke external services, and trigger real-world side effects. In such systems, the critical security boundary is the tool-execution interface: every tool call is an access decision, and every tool result can introduce untrusted influence into later actions. Securing these systems therefore requires more than prompt hardening or post-hoc detection; it requires principled runtime enforcement at the execution boundary.

This paper presented ARM, a deterministic, provenance-aware enforcement layer for tool-calling agents. ARM’s central contribution is to treat denied tool invocations as first-class provenance events and to represent their possible downstream influence through counterfactual edges. That extension makes a denial-feedback leakage channel enforceable: downstream actions can be blocked not only when they depend on untrusted successful flows, but also when they are causally shaped by a prior denial. The same graph-aware mechanism also supports transitive taint propagation and field-level provenance within structured tool outputs.

In a controlled differential evaluation, the graph-aware engine blocked causality laundering, transitive taint propagation, and mixed-provenance field misuse that the flat provenance baseline missed, while adding negligible runtime overhead relative to ordinary agent execution. ARM is not a complete solution to agent security, but these results suggest that denial-aware causal provenance is a useful abstraction for runtime enforcement in tool-calling agent systems.

References

- [1] Nadya Abaev, Denis Klimov, Gerard Levinov, David Mimran, Yuval Elovici, and Asaf Shabtai. AgentGuardian: Learned access-control policies for LLM agents. *arXiv preprint arXiv:2601.10440*, 2026.
- [2] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, 1972. Volume II.
- [3] Anthropic. Model context protocol. <https://modelcontextprotocol.io>, 2024. Specification v1.0.
- [4] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrabie, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*, 2014.
- [5] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.
- [6] Manuel Costa, Ahmed Salem, Aashish Kolluri, Boris Kopf, Shruti Tople, Andrew Paverd, Lukas Wutschitz, Mark Russinovich, and Santiago Zanella-Beguelin. Securing AI agents with information-flow control. In *IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, 2026. arXiv:2505.23643.
- [7] Edoardo Debenedetti et al. AgentDojo: A dynamic environment to evaluate attacks and defenses for LLM agents. In *Proceedings of the 38th Conference on Neural Information Processing Systems (NeurIPS)*, 2024.
- [8] Edoardo Debenedetti et al. Defeating prompt injection by design: Building secure LLM applications with CaMeL. *arXiv preprint arXiv:2503.18813*, 2025.
- [9] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

- [10] Jack B. Dennis and Earl C. Van Horn. Programming generality: A fundamental problem of the one-processor computer. *Communications of the ACM*, 9(3):143–147, 1966.
- [11] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 31–40, 2007.
- [12] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you’ve signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023.
- [13] Keegan Hines et al. Defending against indirect prompt injection attacks with spotlighting. *arXiv preprint arXiv:2403.14720*, 2024.
- [14] Invariant Labs. Tool poisoning attacks in MCP. <https://invariantlabs.ai/blog/mcp-security-notification-tool-poisoning-attacks>, 2024.
- [15] Thomas Jiralerspong, Flemming Kondrup, and Yoshua Bengio. Noticing the watcher: LLM agents can infer CoT monitoring from blocking feedback. *arXiv preprint arXiv:2603.16928*, 2026.
- [16] Minbeom Kim, Mihir Parmar, Phillip Wallis, Lesly Miculicich, Kyomin Jung, Krishnamurthy Dj Dvijotham, Long T. Le, and Tomas Pfister. CausalArmor: Efficient indirect prompt injection guardrails via causal attribution. *arXiv preprint arXiv:2602.07918*, 2026.
- [17] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.
- [18] Frank Li. PRISM: Zero-fork defense-in-depth runtime layer for agent security. *arXiv preprint arXiv:2603.11853*, 2026.
- [19] Yutao Mou, Zhangchi Xue, Lijun Li, Peiyang Liu, Shikun Zhang, Wei Ye, and Jing Shao. ToolSafe: Step-level pre-execution detection for LLM agent safety. *arXiv preprint arXiv:2601.10156*, 2026.
- [20] NVIDIA. NeMo Guardrails: A toolkit for controllable and safe LLM applications. <https://github.com/NVIDIA/NeMo-Guardrails>, 2024.
- [21] Nils Palumbo, Sarthak Choudhary, Jihye Choi, Prasad Chalasani, Mihai Christodorescu, and Somesh Jha. Policy compiler for secure agentic systems. *arXiv preprint arXiv:2602.16708*, 2026.
- [22] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. *arXiv preprint arXiv:2305.15334*, 2023.
- [23] Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2nd edition, 2009.
- [24] Qiskit Contributors. rustworkx: A high-performance graph library for Python. <https://github.com/Qiskit/rustworkx>, 2024.
- [25] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

- [26] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*, 2023.
- [27] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). *Proceedings of the IEEE Symposium on Security and Privacy*, pages 317–331, 2010.
- [28] Eric Wallace et al. The instruction hierarchy: Training LLMs to prioritize privileged instructions. *arXiv preprint arXiv:2404.13208*, 2024.
- [29] Peiran Wang, Yang Liu, Yunfei Lu, Yifeng Cai, Hongbo Chen, Qingyou Yang, Jie Zhang, Jue Hong, and Ye Wu. AgentArmor: Enforcing program analysis on agent runtime trace to defend against prompt injection. *arXiv preprint arXiv:2508.01249*, 2025.
- [30] Qiusi Zhan et al. InjectAgent: Benchmarking indirect prompt injection in tool-integrated LLM agents. *arXiv preprint arXiv:2403.02691*, 2024.
- [31] Tian Zhang, Yiwei Xu, Juan Wang, Keyan Guo, Xiaoyang Xu, Bowen Xiao, Quanlong Guan, Jinlin Fan, Jiawei Liu, Zhiquan Liu, and Hongxin Hu. AgentSentry: Mitigating indirect prompt injection in LLM agents via temporal causal diagnostics and context purification. *arXiv preprint arXiv:2602.22724*, 2026.
- [32] Peter Yong Zhong, Siyuan Chen, Ruiqi Wang, McKenna McCall, Ben L. Titzer, Heather Miller, and Phillip B. Gibbons. RTBAS: Defending LLM agents against prompt injection and privacy leakage. *arXiv preprint arXiv:2502.08966*, 2025.
- [33] Genliang Zhu, Chu Wang, Ziyuan Wang, Zhida Li, and Qiang Li. OPP: OpenPort protocol for agent-to-tool governance. *arXiv preprint arXiv:2602.20196*, 2026.

A Formal Proofs

Theorem 1 (Mediated Integrity). *In an ARM-protected system, there exists no causal path in the provenance graph G from a node with trust level below threshold θ to an ALLOW verdict on a tool call that does not traverse at least one enforcement check.*

Proof. We prove by construction, showing that every possible path from an untrusted source to an ALLOW verdict must traverse the enforcement pipeline.

Step 1: Complete mediation. ARM operates as an MCP proxy. Every `tool_use` message from the LLM client passes through `GraphAwareEngine.evaluate()` before reaching the upstream tool server. There is no code path from the client to the server that bypasses the engine. Therefore, every tool call $c \in V_{\text{CALL}}$ has a corresponding evaluation.

Step 2: Layer evaluation is total. For each tool call, the `LayeredPolicyEngine` evaluates layers L1, L2G, L3, and L4 in order. Evaluation terminates on the first DENY or after all layers return PASS. The engine returns exactly one of $\{\text{ALLOW}, \text{DENY}\}$.

Step 3: L2G enforces provenance. Layer L2G computes $\text{MINTRUST}(c)$ by traversing all ancestors of c in G . If $\text{MINTRUST}(c) < \theta$, the layer returns DENY. Additionally, if $\text{COUNTERFACTUALCHAINS}(c) \neq \emptyset$, the layer returns DENY.

By Definition 4, $\text{MINTRUST}(c) < \theta$ if and only if there exists an ancestor data node d with $\tau(d) < \theta$ —i.e., an untrusted source is reachable from c . Therefore, any path from an untrusted source to c that reaches L2G evaluation will be detected and denied.

Step 4: L1 provides unconditional boundaries. Even if a path bypasses L2G’s provenance check (e.g., no INPUTTO edges were registered for the call), L1 independently enforces:

- Payload size limits (HB-1): exfiltration of bulk data is blocked.
- Sensitive path blocking (HB-2): access to credential files is blocked.
- Credential detection (HB-4): known secret formats are blocked.
- Schema pinning (HB-5): dynamic capability injection is blocked.

Step 5: Composition. By Steps 1–4, any causal path from an untrusted source ($\tau < \theta$) to an ALLOW verdict must traverse the evaluation pipeline (Step 1), where L2G will detect the untrusted ancestry (Step 3). If the untrusted influence bypasses data-flow edges (e.g., through the LLM’s internal reasoning), L1 provides a fallback (Step 4) that blocks the most damaging exfiltration vectors.

For causality laundering paths (no direct data-flow edges, but causal influence through denial signals), COUNTERFACTUAL edges make the influence explicit in G , and L2G’s counterfactual chain check (Step 3) detects and denies the downstream action.

Therefore, no path from an untrusted source to an ALLOW verdict exists that does not traverse at least one enforcement check. \square

Corollary 1 (Defense in Depth). *An attacker must simultaneously evade all layers ($L1$, $L2G$, $L3$, $L4$) to achieve an ALLOW verdict on an unauthorized tool call. Evading any single layer is insufficient.*

Proof. Follows directly from the conjunction semantics of the layer pipeline: all layers must return PASS for the overall verdict to be ALLOW. A DENY from any single layer is sufficient to block the call. \square

Lemma 1 (Monotonic Taint Propagation). *For any path $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ in G :*

$$\text{MINTRUST}(v_k) \leq \text{MINTRUST}(v_1)$$

Proof. By Definition 4, $\text{MINTRUST}(v_k)$ is the minimum trust over all ancestors of v_k . Since v_1 is an ancestor of v_k , the ancestors of v_1 are a subset of the ancestors of v_k . Taking the minimum over a superset cannot increase the value. \square