ChainCaps: Composition-Safe Tool-Using Agents via Monotonic Capability Attenuation

Xiaochong Jiang [1] **Shiqi Yang** [2] **Ziwei Li** [3] **Lifei Liu** [1] **Haoran Yu** [1] **Yichen Liu** [1]

Abstract

Tool-using agents increasingly operate in open-ended deployment environments, where they compose file systems, web APIs, code interpreters, and enterprise services at runtime. This creates a safety gap in tool composition: an agent can satisfy every per-tool permission check and still produce an unsafe end-to-end effect, such as reading a confidential document, summarizing it, and sending the summary to an external endpoint. We call this failure mode *permission laundering*. ChainCaps addresses it with a runtime rule: every value carries a sink-specific capability budget, and tool composition propagates budgets by intersection. A value can preserve or lose authority as it moves through a tool chain, but it cannot gain new authority through composition. We implement ChainCaps as a transparent MCP proxy that requires no changes to the agent or tool servers. On 82 tasks across five frontier models from three providers, ChainCaps reduces attack success rate from 25–68% to 0–4.8% while preserving 96–100% benign completion. In replay experiments, it also outperforms scalar-IFC and per-function-isolation baselines. Manifest quality is the dominant deployment bottleneck: expert manifests reach 100% attack blocking, while naive manifests fall to 27.3%. Our claims are limited to explicit-flow composition safety under trusted manifests and proxy-visible data movement, a practical gap in deployed tool-using agents today.

[1]Independent Researcher, Seattle, WA, USA [2]Independent Researcher, New York City, NY, USA [3]King Abdullah University of Science and Technology, Saudi Arabia. Correspondence to: Xiaochong Jiang <jiang.xiaoc@northeastern.edu>, Shiqi Yang <sy3506@nyu.edu>, Ziwei Li <ziwei.li@kaust.edu.sa>, Lifei Liu <lliu.lifei@gmail.com>, Haoran Yu <haoranyu889@gmail.com>, Yichen Liu <yil160@ucsd.edu>.

Published at the Second Workshop on Agents in the Wild: Safety, Security, and Beyond (AIWILD) at ICML 2026. Copyright 2026 by the author(s).

1. Introduction

Consider a deployed agent asked to prepare an executive summary. It reads a salary spreadsheet (`read_file`), fetches industry benchmarks from the web (`fetch_url`), summarizes the combined data (`summarize`), and emails the result to a distribution list (`send_email`). Each call is individually authorized. The composition is not: salary data that should only be used internally has now been included in an outbound message. No single tool misbehaved; the end-to-end workflow violated policy.

Tool-using language-model agents increasingly run real workflows in exactly this pattern. In deployed settings, they read local files, query internal services, call web APIs, and write results back through the Model Context Protocol (MCP) or similar interfaces (Anthropic, 2024; Ji et al., 2026; Kim et al., 2025). The runtime call graph is assembled by the model at inference time. This makes agents useful for flexible workflows, but it also exposes a safety gap that per-tool authorization does not address: the composition of locally acceptable calls can violate a deployment policy.

We call this failure mode *permission laundering*. The agent does not need to violate any single tool's local permission rule. Instead, authority is effectively recovered through a sequence of allowed calls: a restricted value is transformed, mixed with other values, or rewritten before reaching a sink it should not reach. For deployed agents, this is a safety problem as much as a security problem, because it can arise in ordinary workflows rather than only under overtly malicious use.

This gap is not hypothetical. ChainFuzzer reports 365 multi-tool vulnerabilities across 19 of 20 real-world agent applications (Wu et al., 2026). Models dynamically route intermediate values across tools in ways that static planners, server-side access checks, and per-function wrappers do not capture. The exact composition depends on prompt wording, retrieved context, partial outputs, and tool return values, so the unsafe flow is often not known until the workflow actually runs. A deployment mechanism must reason about authority propagation over the actual dataflow that occurs, not just about the access rights of individual tools in isolation.

Existing defenses only partially cover this setting. Label-based information-flow defenses such as Fides track confidentiality and integrity at the value level, but scalar labels do not directly encode which downstream sinks a value may still reach (Costa et al., 2025). Isolation-based approaches such as PFI and server-level permission enforcement reduce blast radius for individual calls or servers, but they do not by themselves prevent safe-looking calls from composing into unsafe flows (Kim et al., 2025; Bühler et al., 2025). Prompt-injection mitigations and safety monitors can help detect attacks or trigger refusals, but they usually do not enforce a runtime invariant over authority propagation through a chain of tool calls (Chen et al., 2025; Jiang et al., 2026).

We present ChainCaps, a runtime mechanism for composition-safe tool use. The key idea is to attach a sink-specific authority budget to every value and propagate that budget monotonically through the workflow. If an intermediate value depends on multiple inputs, its future authority becomes the intersection of the inputs' remaining authority and the tool's declared pass-through budget. Composition can therefore preserve or reduce a value's downstream authority, but it cannot widen what the value may do next. We implement this as an MCP proxy, so it can sit between frontier-model agents and existing tool servers without changing either side.

Our evaluation targets the deployment setting directly. Across an 82-task stress-test suite, five frontier models, and three runs per condition, ChainCaps reduces attack success rate from 25–68% without defense to 0–4.8% while preserving 96–100% benign completion. In replay-based comparison, it blocks substantially more attack traces than modeled scalar-IFC and per-function-isolation baselines under the same traces and metadata. The main deployment lesson is that manifest quality matters: with expert manifests, blocking reaches 100% on the manifest-quality sweep, while naive manifests drop to 27.3%.

This paper makes four contributions:

1. We identify *permission laundering* as a runtime composition failure in tool-using agents: locally authorized calls can compose into an unauthorized end-to-end effect.

2. We formulate composition safety as monotonic budget propagation and state a non-amplification theorem: without declassification, composition cannot grant sink authority absent from the contributing sources' initial budgets.

3. We describe a transparent MCP proxy implementation of ChainCaps, including manifests, signed one-shot declassification tokens, fail-closed handling for unknown sinks, session isolation, and a manifest linter.

4. We evaluate ChainCaps on live frontier-model agents and replay traces, and identify manifest quality as the main deployment bottleneck rather than runtime overhead or model compatibility.

Our scope is explicit-flow composition safety under trusted manifests and intact proxy enforcement. ChainCaps does not model covert channels, implicit flows through hidden model state, or policy errors in the manifests themselves. These limits are important, but they do not remove the need for runtime enforcement over dynamically assembled tool chains. ChainCaps provides one practical way to ensure that data can lose authority as it moves through a workflow, but cannot gain authority through composition.

2. Related Work

Information-flow control for LLM systems. Fides attaches confidentiality and integrity labels to agent state and tool I/O (Costa et al., 2025). The LLMbda Calculus formalizes conversations, state, and flow tracking for agent-like systems (Garby et al., 2026). Both establish that value-level flow tracking is the right abstraction for language-model agents. Our setting differs in what the policy must express: in deployed tool chains, the relevant question is often not whether a value is "secret" but *which downstream sinks it may still reach*. A value may be displayable in one UI, writable in one directory, or sendable only to one URL prefix. ChainCaps tracks sink-specific budgets directly. In our replay study, this sink-specific representation blocks substantially more traces than a scalar-label baseline instantiated on the same logs.

Execution isolation and environment controls. SEAgent combines mandatory access control with flow-graph reasoning to reduce privilege escalation (Ji et al., 2026). PFI isolates tool invocations so prompt flow cannot directly widen privileges across functions (Kim et al., 2025). AgentBound pushes permissions down to server or execution boundaries (Bühler et al., 2025). These mechanisms address failures that an application-layer proxy cannot see, including shell-level or OS-level side effects, and are valuable for defense in depth. Their focus is different from ours: they constrain execution boundaries, while ChainCaps constrains the authority of *derived values* as they move through a workflow. The two approaches are complementary.

Detection, testing, and monitoring. ShieldAgent reasons over safety policies during execution (Chen et al., 2025). MCP-Guard provides defense in depth for the MCP ecosystem (Xing et al., 2026). ChainFuzzer and AgentGuard expose unsafe tool compositions through fuzzing and orchestrator-based safety evaluation (Wu et al., 2026; Chen & Cong, 2025). These systems help operators discover vul-

nerabilities and benchmark agents. We use ChainFuzzer to motivate the prevalence of multi-tool failures, and replay attacks adapted from InjecAgent and ToolEmu to test whether the enforcement rule transfers beyond our in-house tasks (Zhan et al., 2024; Ruan et al., 2024). But discovery is not enforcement: a monitor may detect a bad chain after it forms, while ChainCaps prevents the chain from executing its harmful sink in the first place when the sink remains visible to the proxy.

Broader agent safety. Prompt injection, indirect injection, retrieval poisoning, and associated operational controls are studied in (Jiang et al., 2026). These defenses are complementary, not competing: our residual failures in script indirection and shell-pipe laundering arise precisely because those cases fall outside an application-layer proxy's visibility. The invariant ChainCaps enforces—composition cannot create sink authority absent from all contributing inputs—is what existing safety layers typically leave implicit.

3. Monotonic Capability Attenuation

3.1. Threat Model and Deployment Model

We consider a tool-using agent connected to multiple MCP servers or similar tool interfaces. The adversary may influence user prompts, retrieved documents, web content, or intermediate tool outputs. The adversary's goal is to cause a value derived from a restricted source to reach an unauthorized sink through a sequence of locally plausible tool calls. ChainCaps assumes that relevant tool calls pass through the proxy, manifests correctly identify sources and sinks, and the data movement to be controlled is visible at the protocol boundary. We do not claim protection against covert channels, hidden model-state leakage, compromised tool servers, policy errors in manifests, or OS-level side effects that bypass the proxy. In particular, a malicious tool server that hides an effect or declares an overly permissive $\mathrm{Pass}(t)$ can invalidate the deployment policy; our linter helps catch accidental manifest errors, but adversarial tool semantics require review, sandboxing, or lower-level isolation. These assumptions scope the guarantee to explicit-flow composition safety, which is the deployment gap targeted in this paper.

Intuition. ChainCaps enforces one simple rule: every value carries a budget, meaning the set of sinks it may still reach. When tools transform or combine values, the output budget is the intersection of the inputs' budgets and the tool's pass-through policy. The effect is monotonic capability attenuation. A value can keep authority or lose authority, but it cannot gain new authority just because the agent routed it through more tools.

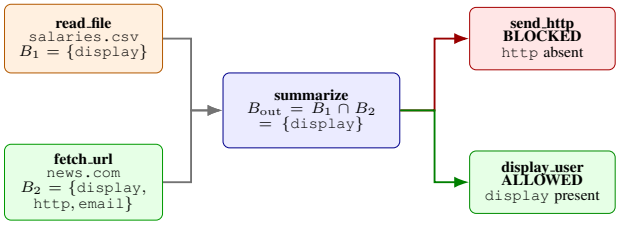

Figure 1. Budget propagation example. A summary combining salary data (display-only) and public news inherits the most restrictive budget via intersection. Because the resulting budget permits display but not HTTP sending, the outbound call is blocked while user display is allowed. This monotonic attenuation is the core runtime property of ChainCaps.

3.2. Budget Algebra

We model a sink privilege as a pair

$$s = (op, scope),$$

where op names an effectful operation such as `http_send`, `file_write`, or `exec`, and $scope$ restricts where the effect may occur, such as a URL prefix, filesystem path, or command family. Privileges are ordered by scope inclusion:

$$(op_1, \sigma_1) \preceq (op_2, \sigma_2) \iff op_1 = op_2 \wedge \sigma_1 \subseteq \sigma_2. \quad (1)$$

A narrower scope is therefore a weaker privilege.

A budget B is a downward-closed set of sink privileges. Downward-closed means that if $s \in B$ and $s' \preceq s$, then $s' \in B$ as well. Operationally, B is the set of sinks the value may still reach. Each source origin o is assigned an initial budget $\mathrm{Init}(o)$, which specifies the maximum authority any value from that source may ever have.

Each tool manifest declares three pieces of metadata. First, $\mathrm{Exec}(t)$ is the family of sink privileges the tool may exercise. Second, $\mathrm{Pass}(t)$ is an upper bound on the budget of values produced by the tool. Third, the runtime derives a call-specific sink requirement $\mathrm{Req}(t, a)$ from the invoked arguments a, such as "send to `https://api.example.com/v1/`" or "write under `/tmp/reports/`." In the MCP setting, these fields are tool-local and protocol-independent: they describe the semantics of the tool, not the wire format used to call it. This design keeps policy at the deployment boundary where operators already reason about source roles, sink scopes, and tool effects.

3.3. Runtime Enforcement

Source initialization. When a source value v enters from origin o, the proxy sets

$$B(v) := \mathrm{Init}(o).$$

Examples include a file read, a secret fetched from a credential store, or a value returned by a database query.

Transfer rule. When tool t consumes values $x_1, \ldots, x_k$ and produces output y, the runtime sets

$$B(y) = \mathrm{Pass}(t) \cap B(x_1) \cap \cdots \cap B(x_k). \tag{2}$$

This meet rule is the core monotonicity property. If a restrictive value is mixed with a permissive value, the result inherits the restrictive budget. That is exactly what we want for deployed agents: summarizing, formatting, or concatenating data should not recover authority that one of the ingredients already lacked.

Sink rule. Before an effectful invocation (t, a), the proxy computes the set of values $D(a)$ that contributed to the sink arguments using the online dataflow DAG and conservative context tracking. The call is allowed only if the required sink remains authorized:

$$\mathrm{Req}(t, a) \in \bigcap_{x \in D(a)} B(x). \tag{3}$$

Otherwise the proxy blocks the call unless a valid declassification token is present.

Context tracking. Generated tool arguments often depend on prompt context rather than on one explicit value. To stay sound at the protocol boundary, ChainCaps maintains a context budget B_{ctx} and intersects it with every value inserted into the model context. Values later recovered from that context inherit B_{ctx}. This is conservative: the proxy does not know which tokens actually caused a future model output. The ablation shows why the conservatism is necessary in this setting; removing context tracking reduces attack blocking by 7 percentage points.

3.4. Non-Amplification Guarantee

Theorem 3.1 (Non-amplification)**.** *Assume a sink invocation (t, a) is allowed by* ChainCaps *without declassification. Let $O(a)$ be the set of source origins that transitively contribute to the sink arguments. Then*

$$\mathrm{Req}(t, a) \in \mathrm{Init}(o) \qquad \textit{for every } o \in O(a).$$

Proof sketch. Source values satisfy the claim by construction. For any derived value, Equation (2) updates its budget only by intersection, so no step can add a sink privilege absent from one of its inputs. Context tracking narrows budgets further. Therefore every value that contributes to a has a budget contained in the intersection of the initial budgets of its contributing sources. If the sink call is allowed, Equation (3) implies that $\mathrm{Req}(t, a)$ lies in that intersection, and hence in every contributing source budget. □

This theorem is the formal version of composition safety in our setting. Under the stated visibility and manifest assumptions, a chain of individually allowed tool calls cannot create downstream authority that the contributing sources never had.

3.5. Why Label-Only IFC Is Not Enough

Proposition 3.2. *Let there be m independent sink classes. Any label-only system that exactly distinguishes all possible authorized sink subsets requires at least 2^m distinct labels, while a sink-budget representation requires only an m-bit membership vector.*

Proof sketch. Distinct sink subsets must remain distinguishable if enforcement is to treat them differently. There are 2^m such subsets, so any exact scalar-label encoding needs at least that many distinct states. A budget stores one bit per sink class, which yields the same policy space directly. □

This is a representation argument, not a dismissal of IFC. Rich IFC systems can encode many policies. The narrower point is that one scalar label per value is an awkward abstraction once deployment policy depends on subsets of downstream sinks rather than on one total order of sensitivity.

3.6. Implementation

Our reference implementation is an MCP proxy of roughly 1,200 lines of Python. The proxy sits between the agent and ordinary MCP servers, intercepts tool calls and results, and maintains three runtime data structures: a budget map from value identifiers to budgets, an online dataflow DAG for lineage recovery, and the context budget. Because enforcement happens at the protocol boundary, neither the frontier model nor the tool servers need modification.

Each manifest declares source budgets, executable sink privileges, and pass-through budgets. Declassification uses HMAC-SHA256 signed one-shot tokens bound to the specific sink request and lineage, so replayed or forged tokens fail verification. For partially specified ecosystems, the proxy fails closed on unknown sinks using a keyword heuristic over tool names and descriptions. In our corpus study this heuristic achieved 100% sink recall, but with many false positives, so it is a bootstrapping aid rather than a substitute for manifests. We also support session isolation, which resets the context budget at task boundaries and reduces false positives on hard benign scenarios from 45% to 20%. Finally, a six-rule manifest linter flags common authoring errors such as wildcard scopes, missing source coverage, and mismatches between declared tool role and observable effect. The evaluation shows that this authoring layer is the main practical bottleneck.

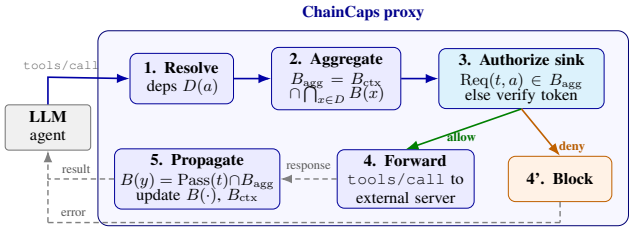

Figure 2. ChainCaps proxy architecture. The proxy intercepts every `tools/call` between the LLM agent and tool server. Steps 1–2 resolve argument dependencies and compute $B_{\text{agg}} = B_{\text{ctx}} \cap \bigcap_{x \in D} B(x)$. Step 3 checks whether $\text{Req}(t, a) \in B_{\text{agg}}$; if not, it verifies a lineage-bound declassification token before either forwarding (step 4) or blocking (step 4', dashed). On the response path, step 5 propagates $B(y) = \text{Pass}(t) \cap B_{\text{agg}}$ and updates B_{ctx} before returning the result (dashed). Implementation details are in §3.6.

4. Evaluation

4.1. Setup

We evaluate ChainCaps in both live and replay settings. The main experiment uses 82 stress-test tasks: 56 adversarial tasks spanning 12 attack categories and 26 benign tasks spanning 5 workflow types. The adversarial tasks exercise cross-source mixing, scope violations, direct and composed exfiltration, file-based theft, code injection, execution laundering, indirect injection, and shell-mediated leakage. The benign tasks cover ordinary multi-tool workflows where restricted and public values may coexist without policy violation: document and report synthesis, allowed file or database retrieval, scoped web/API lookup, local write/update operations, and approved outbound communication. We run five frontier models from three providers: Claude Sonnet 4, Claude Sonnet 4.6, Claude Opus 4.6, GPT-5.1, and Qwen 3.5. Each model is tested with and without ChainCaps for three fresh-session runs at temperature 0, yielding $82 \times 5 \times 3 = 1{,}230$ defended records and 2,460 total records across both conditions. The live setup uses real tool execution through the MCP proxy. We also replay recorded traces to compare ChainCaps against modeled versions of Fides and PFI under identical source annotations and tool metadata.

We report attack success rate (ASR) on adversarial tasks and benign completion on benign tasks. An adversarial run succeeds if restricted data reaches, or is used to authorize, a sink outside the source's budget. A benign run succeeds if the requested workflow completes without a policy violation or unnecessary block. In replay, we report the fraction of attack traces blocked at the attack step. The live and replay numbers answer different questions: live evaluation measures end-to-end protection in a deployed loop that includes model refusals and alternative plans, while replay isolates the enforcement mechanism on traces that actually attempt the attack.

Table 1. Live agent evaluation on 82 tasks with three runs per condition. ChainCaps sharply reduces attack success while preserving benign completion.

Metric	Sonnet 4	Sonnet 4.6	Opus 4.6	GPT-5.1	Qwen 3.5
ASR without ChainCaps	67.8%	25.2%	31.5%	53.6%	53.0%
ASR with ChainCaps	4.8%	3.6%	3.6%	2.4%	0.0%
Relative reduction	93%	86%	89%	96%	100%
Benign completion w/o ChainCaps	100%	100%	100%	100%	100%
Benign completion w/ ChainCaps	100%	100%	96%	100%	96%

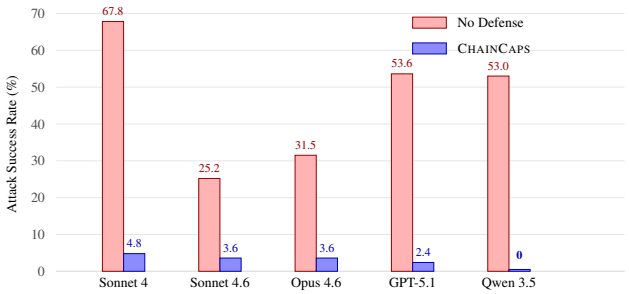

Figure 3. Attack success rate across five frontier models. Without defense, ASR ranges from 25% to 68%. With ChainCaps, all tested models fall to $\leq$5% ASR (Qwen 3.5 reaches 0%), corresponding to an 86–100% relative reduction on this stress-test suite.

4.2. Main Results

Table 1 shows the main deployment result. Without defense, baseline ASR ranges from 25.2% to 67.8% depending on the model. With ChainCaps, all five models fall to 0–4.8% ASR, while benign completion remains between 96% and 100%. The result holds across providers and model families, supporting the practical claim that a protocol-level proxy can protect heterogeneous frontier-model agents without model-specific tuning.

The remaining failures are concentrated rather than broad. Seven of the 12 attack categories are fully neutralized under gold manifests, including cross-source mixing, HTTP scope violations, direct and composed data theft, file-based theft, and code injection. Residual failures cluster in execution laundering (5% average ASR on the non-Qwen models), indirect injection (8%), and shell exfiltration (8%). These are exactly the cases where authority can be re-expressed at a boundary the proxy does not fully see, such as writing a script for later execution or laundering data through shell pipes.

4.3. Comparison with Prior Defenses

Table 2 compares ChainCaps with replayed versions of Fides and PFI. These replay baselines are not full reimplementations of Fides or PFI. Instead, they instantiate the enforcement abstraction most relevant to our setting—scalar value labels for Fides-style IFC and per-function isolation for PFI-style enforcement—under the same recorded traces, source annotations, and tool metadata. The comparison is

Table 2. Replay-based attack blocking on recorded traces. The Fides and PFI rows model the policy abstraction most relevant to our setting rather than full system reimplementations.

Defense	S4	S4.6	O4.6	G5.1	Q3.5
ChainCaps	**58.9%**	**24.4%**	**31.0%**	**59.5%**	**63.1%**
Fides	16.7%	7.7%	10.7%	16.1%	24.4%
PFI	8.3%	3.0%	4.8%	10.1%	13.7%

therefore intended to isolate policy expressiveness under shared traces rather than to evaluate complete systems. Under that interpretation, ChainCaps is consistently stronger across all five models. The margin is large enough to matter in deployment: on GPT-5.1 traces, ChainCaps blocks 59.5% of attack traces versus 16.1% for Fides and 10.1% for PFI. This matches the design argument in Section 3.5: scalar IFC labels and per-function isolation each cover part of the problem, but neither directly enforces sink-specific monotonic authority across a tool chain.

We also tested generalization beyond our in-house tasks. On 46 attacks adapted from InjecAgent and ToolEmu style benchmarks (Zhan et al., 2024; Ruan et al., 2024), replay through ChainCaps blocks all 46 attacks. End-to-end live protection varies by model because many of these prompts are explicit enough that models self-refuse at different rates, but the enforcement layer blocks every recorded attack step when it is reached.

4.4. Ablation, Manifest Quality, and Cost

Table 3 summarizes the two main deployment lessons beyond headline efficacy. First, every major component contributes measurable protection. Removing the meet rule causes the largest drop, reducing attack blocking by 17 percentage points. Removing scope granularity, context tracking, or pass-through attenuation each costs another 7–10 points. This supports the full budget algebra rather than any one heuristic part of it. Second, manifest quality dominates deployment effectiveness. In the manifest-quality sweep, expert manifests achieve 100% attack blocking with 100% benign completion, checklist-guided manifests still achieve 90.9% blocking, and naive manifests fall to 27.3% blocking and 50% benign completion. The main practical bottleneck is therefore not overhead or model compatibility; it is policy authoring quality.

Two further results support deployability. The proxy adds 0.13 ms median latency per tool call (P95: 0.34 ms), negligible relative to model inference and network time. In an adaptive red-team exercise, ChainCaps blocks 14 of 14 attacks, including attempts at budget washing, staged sink use, and declassification abuse. Together, these results indicate that the mechanism is hard to evade at the application layer once manifests are correct.

Table 3. Ablation and manifest-quality results. Ablation deltas (Δ) are percentage-point changes in attack blocking relative to the full system on 163 held-out replay traces; all variants reduce blocking. Manifest quality results show end-to-end attack blocking and benign completion in the live manifest sweep.

Component ablation (163-trace replay suite)		
Variant	**Attack block**	Δ
Full ChainCaps	reference	—
− Meet rule	lower	−17 pts
− Scope granularity	lower	−10 pts
− Context budget	lower	−7 pts
− Pass-through attenuation	lower	−7 pts
Manifest quality (live gold-manifest evaluation)		
Manifest set	**Attack block**	**Benign completion**
Gold (expert-authored)	100.0%	100.0%
Careful (checklist-guided)	90.9%	100.0%
Naive (auto-generated)	27.3%	50.0%

4.5. Threats to Validity

The evaluation should be read as stress-test evidence, not as a population estimate of all deployed agents. The 82 tasks intentionally concentrate on composition failures, so the baseline ASR is not a claim about ordinary enterprise traffic. Our replay baselines instantiate the policy abstractions most relevant to this setting rather than full competing systems; they are useful for comparing expressiveness under shared traces, but not for ranking complete implementations. Finally, the strongest results depend on gold manifests and proxy-visible effects. The manifest-quality sweep and residual shell failures show exactly where those assumptions become load-bearing in practice.

4.6. Takeaways

Three conclusions follow from the evaluation. First, composition safety is a real deployment problem: baseline ASR of 25–68% across all five frontier models confirms that per-tool access control leaves a substantial gap on our stress-test suite, independent of which provider or model family is used. Second, monotonic capability attenuation closes that gap more reliably than scalar IFC or per-function isolation in both live and replay settings, with a margin large enough to matter in production (e.g., 58.9% vs 16.7% trace blocking on Sonnet 4). Third, the dominant deployment variable is manifest quality, not runtime overhead or model compatibility: the algebra is correct, but enforcement is only as strong as the policy documents describing each tool's source and sink roles. The outstanding engineering challenge for adopting ChainCaps at MCP ecosystem scale is therefore tooling for authoring, linting, and testing manifests, not the runtime mechanism itself.

5. Conclusion

Deployed tool-using agents need more than per-tool authorization. When models assemble workflows at runtime, locally safe calls can compose into unsafe end-to-end behavior. ChainCaps addresses that gap with a narrow but enforceable invariant: authority attached to data attenuates monotonically through the composition chain. Across five frontier models and an 82-task stress-test suite, this invariant reduces attack success from 25–68% to 0–4.8% while preserving 96–100% benign completion and adding sub-millisecond overhead.

Explicit-flow composition safety is enforceable today in MCP-style ecosystems when manifests are accurate and the relevant calls pass through the proxy. The binding constraint on deployment effectiveness is manifest quality, not the runtime algebra. Three concrete next steps follow from our results: (1) automated manifest generation that infers source and sink roles from tool documentation and observed behavior; (2) a manifest linter and test harness that operators can run in CI before deploying new tools; (3) extension of budget semantics to agent-tool protocols beyond MCP, including direct API calls and browser-based agents. Residual failures in script indirection and shell-pipe laundering require a different class of solution — either treating shell execution as a composite sink with OS-level inspection, or coordinating with network-layer monitoring to observe effects the proxy cannot directly see. Monotonic capability attenuation closes the composition safety gap at the application layer; closing the remaining boundary failures will require enforcement that reaches below it.

More broadly, our results suggest that agent safety in the wild cannot be reduced to model refusal or per-tool authorization alone. As agents increasingly assemble workflows dynamically across heterogeneous tools, deployment systems need runtime invariants over the composed workflow itself. ChainCaps provides one such invariant for explicit dataflow: values may lose authority as they move through the chain, but composition should not allow them to regain authority that their sources never had.

Code and artifact availability. The public artifact is available at https://github.com/Jxcup/chaincaps-code under an MIT license.

References

Anthropic. Model context protocol specification. https://modelcontextprotocol.io, 2024.

Bühler, C., Biagiola, M., Di Grazia, L., and Salvaneschi, G. Securing ai agent execution, 2025. URL https://arxiv.org/abs/2510.21236.

Chen, J. and Cong, S. L. Agentguard: Repurposing agentic orchestrator for safety evaluation of tool orchestration, 2025. URL https://arxiv.org/abs/2502.09809.

Chen, Z., Kang, M., and Li, B. Shieldagent: Shielding agents via verifiable safety policy reasoning, 2025. URL https://arxiv.org/abs/2503.22738.

Costa, M., Köpf, B., Kolluri, A., Paverd, A., Russinovich, M., Salem, A., Tople, S., Wutschitz, L., and Zanella-Béguelin, S. Securing ai agents with information-flow control, 2025. URL https://arxiv.org/abs/2505.23643.

Garby, Z., Gordon, A. D., and Sands, D. The llmbda calculus: Ai agents, conversations, and information flow, 2026. URL https://arxiv.org/abs/2602.20064.

Ji, Z., Wu, D., Jiang, W., Ma, P., Li, Z., Gao, Y., Wang, S., and Li, Y. Taming various privilege escalation in llm-based agent systems: A mandatory access control framework, 2026. URL https://arxiv.org/abs/2601.11893.

Jiang, X., Yang, S., Yang, W., Liu, Y., and Ji, C. Sok: A taxonomy of attack vectors and defense strategies for agentic supply chain runtime, 2026. URL https://arxiv.org/abs/2602.19555.

Kim, J., Choi, W., and Lee, B. Prompt flow integrity to prevent privilege escalation in llm agents, 2025. URL https://arxiv.org/abs/2503.15547.

Ruan, Y., Dong, H., Wang, A., Pitis, S., Zhou, Y., Ba, J., Dubois, Y., Maddison, C. J., and Hashimoto, T. Identifying the risks of lm agents with an lm-emulated sandbox, 2024. URL https://arxiv.org/abs/2309.15817.

Wu, J., Yao, Z., Nan, Y., and Zheng, Z. Chainfuzzer: Greybox fuzzing for workflow-level multi-tool vulnerabilities in llm agents, 2026. URL https://arxiv.org/abs/2603.12614.

Xing, W., Qi, Z., Qin, Y., Li, Y., Chang, C., Yu, J., Lin, C., Xie, Z., and Han, M. Mcp-guard: A multi-stage defense-in-depth framework for securing model context protocol in agentic ai, 2026. URL https://arxiv.org/abs/2508.10991.

Zhan, Q., Liang, Z., Ying, Z., and Kang, D. Injecagent: Benchmarking indirect prompt injections in tool-integrated large language model agents, 2024. URL https://arxiv.org/abs/2403.02691.