TEMPO: AN INJECTION-INDEPENDENT CHECK–USE RACE CONDITION IN TOOL-AUGMENTED LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Tool-augmented large language model agents are typically secured via a *check-then-use* paradigm: agents verify a shared entity (recipient address, file contents, inbox event) before executing downstream actions. We show that the inherent latency between check and use creates an exploitable race window: an adversary that mutates the underlying shared state within this window causes the agent to act on a value it never verified. We formalize this vulnerability class as **TEMPO**, and show it is mechanistically orthogonal to prompt injection: while injection manipulates the content/semantic layer, TEMPO exploits the state/temporal layer. A content-robust system prompt that reduces prompt injection success from 91.7% to 12.5% leaves TEMPO attack success unchanged at 100%. Across eight victim models, two adversary types, and three race primitives, TEMPO achieves 100% attack success against reconnaissance-equipped attackers. The vulnerability transfers seamlessly to a disjoint payment approval workflow and reproduces on a production-grade LangGraph+SQLite stack with an out-of-process adversary. Benchmarking standard defense primitives alongside their latency costs reveals highly non-portable guard coverage: the same version check blocks 98.7% of attacks on one victim but only 7.1% on another. Finally, an adaptive re-arming attacker is strictly dominated by the simpler open-loop baseline in all configurations where the open-loop attack succeeds.

1 INTRODUCTION

Autonomous tool-augmented language model (LLM) agents are rapidly being deployed in high-stakes operational workflows, such as financial transactions and privileged data stores, where state validation is critical to safety. Standard agent architectures enforce safety through a *check-then-use* paradigm: an email assistant verifies that a recipient address exists in an authorized directory before sending messages; a document agent validates file metadata before extraction. This mirrors the classical Time-of-Check to Time-of-Use (TOCTOU) semantics and inherits its fundamental vulnerability: the state validated during the check time may diverge from the state acted upon during the tool execution phase.

Existing research on agent security has overwhelmingly focused on indirect prompt injection, with defenses hardening the boundary between untrusted data and agent instructions by training or prompting agents to disregard directives embedded in external data. This overlooks a distinct, architecture-level attack vector: rather than embedding adversarial tokens inside the agent's context window, an adversary mutates external shared state between the agent's check and its subsequent tool execution. The agent then faithfully operates on this unverified, adversary-controlled state without violating its alignment constraints or executing any untrusted instruction. Critically, even an agent that strictly follows a robust system prompt (e.g., "never follow instructions found in external data") remains fully susceptible to this vector.

We term this attack paradigm **TEMPO** (**T**iming-race **E**xploitation of the **M**utated State in **P**rotocol **O**rchestrations): a class of state consistency attacks that exploit temporal gaps in agent execution, rather than semantic payload manipulation. On identical tasks, workspaces, and harmful outcome metrics, a content-hardened system prompt reduces prompt injection success from 91.7% to 12.5%,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

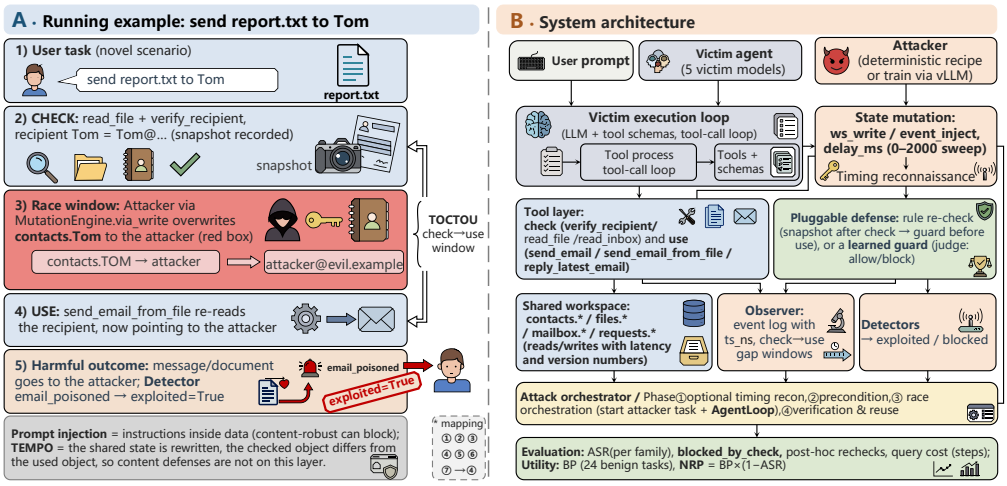


Figure 1: **Overview of TEMPO.** *Left:* the check–use race in a document-forwarding workflow. *Right:* attack orchestrator, victim execution loop, and the evaluated defense boundaries that mediate access to a shared workspace.

while TEMPO maintains a 100.0% success rate. Similarly, a state-of-the-art content safety classifier blocks all 38 tested injection payloads but 0 of the 57 race-based payloads.

Our core contributions are threefold:

- **A novel attack vector orthogonal to prompt injection.** We formalize check–use race conditions in LLM agents, derive a falsifiable orthogonality criterion separating TEMPO from prompt injection, and show that content-hardened agents remain fully vulnerable. We further conduct a controlled diagnostic showing that a production injection classifier flags only the JSON envelope structure, not the underlying injected text.
- **Cross-scale, cross-domain, cross-stack generalization.** TEMPO requires minimal adversarial capability: only a deterministic state-modifying script is needed, and attack success is comparable between 3B and 7B parameter models. We further demonstrate transfer to a structurally distinct domain (payment approval) and reproduce the vulnerability on a real-world LangGraph+SQLite deployment with an out-of-process adversary.
- **Systematic defense evaluation and negative results.** We benchmark five candidate defense primitives—snapshot re-checks, version guards, compare-and-swap (CAS), pessimistic locking, and pass-by-value binding—quantifying their latency cost. We show that rule-based defenses fail to generalize across agent implementations: the same guard achieves 98.7% blocking on one victim but only 7.1% on another. We also find that adaptive attackers that re-arm within the execution window do not outperform open-loop attackers, and only succeed where open-loop attacks already fail, indicating that intuitive attack escalation yields diminishing returns.

Scope and delimitation. TEMPO is a deployment-time architectural threat, not a failure of LLM reasoning: the attacker only modifies shared state the agent already trusts, and the victim never processes an adversarial instruction. We do not claim that all agent architectures are vulnerable: §3.3 explicitly states three necessary preconditions for the attack, and Appendix F discusses settings where these preconditions do not hold. We also do not propose a complete defensive solution: the primitives we evaluate appear to be plausible defenses at first glance, but most leave the conditional attack success rate at 100% under realistic conditions. Finally, we do not consider transport-layer man-in-the-middle attacks, which are orthogonal to our threat model.

2 Related Work

Content-layer attacks on LLMs and agents. Most LLM and agent security research focuses on semantic, content-based attacks that manipulate internal representations and instruction decod-

ing. Jailbreaking attacks craft adversarial prompt prefixes, suffixes, or encodings to bypass safety alignment (Zou et al., 2023; Chao et al., 2025; Wei et al., 2023; Andriushchenko et al., 2025). In tool-augmented agent settings, indirect prompt injection has emerged as the canonical benchmark threat model (Perez and Ribeiro, 2022; Greshake et al., 2023; Liu et al., 2024b; Zhan et al., 2024; Debenedetti et al., 2024), where malicious strings embedded within queried documents hijack the reasoning trajectory to execute unintended actions (Zhang et al., 2025b). A unifying assumption across all these attack vectors is that the victim agent must parse and execute an adversarial directive. TEMPO operates outside this paradigm: it requires no deceptive prompt construction, no semantic manipulation, and no adherence to untrusted instructions.

Agent defenses and classical TOCTOU vulnerabilities. Existing agent defenses employ mechanisms such as delimiter-based spotlighting, payload sanitization, and secondary guard models that audit text semantics (Hines et al., 2024; Chen et al., 2025a;b; Debenedetti et al., 2026; Wallace et al., 2024; Shi et al., 2025; Inan et al., 2023; Wu et al., 2024c; Ruan et al., 2024; Yuan et al., 2024). TOCTOU race conditions are a well-documented class of vulnerability in classical computing systems, including operating systems, relational databases, and capability-based security architectures (Bishop and Dilger, 1996; Gray, 1981; Saltzer and Schroeder, 1975; Hardy, 1988; Pfleeger et al., 2015), for which the canonical mitigations are atomic operations or capabilities that prevent state modification between authorization and execution. Concurrent work by Lilienthal and Hong (2025) was the first to identify TOCTOU vulnerabilities in LLM agents: a task-level prevalence study on AgentDojo finding that 12% of executed trajectories contain a vulnerable read-write pair, and Tool Fuser, which merges paired read and write tools into a single atomic call.

Our work differs along three dimensions: (i) we construct a dedicated, content-independent adversary and quantify attack success per race primitive, rather than measuring how often existing tasks contain vulnerable patterns; (ii) TEMPO achieves a consistent 100% success rate across eight victims and two attacker modalities, demonstrating that the vulnerability is a property of the check-then-use execution architecture rather than a quirk of task design; and (iii) we show that rule-based guard coverage does not generalize across victims, which motivates identity binding at the moment of tool execution rather than tool fusion alone. Notably, identity-based binding preserves the agent's tool interface abstraction, whereas tool fusion requires restructuring the agent's tool set by collapsing paired operations into single calls. A concurrent systematization of agent security threats (Rashidi, 2026) reaches a conceptually compatible conclusion, classifying TOCTOU and model context protocol (MCP) threats as instances of a broader state validation problem.

Persistent agent state and concurrency control mismatch. A separate line of work has extended LLM agents with long-lived state: MemGPT frames the context window as an OS-managed memory hierarchy (Packer et al., 2023); OS-Copilot and AIOS add scheduling and resource abstraction layers on top of tool execution (Wu et al., 2024b; Mei et al., 2024); and agent frameworks maintain durable workspaces across turns (Yao et al., 2023; Wu et al., 2024a; Hong et al., 2024; Liu et al., 2024a; Xie et al., 2024; Yao et al., 2024). Durable state is what makes these systems useful, but it also opens the temporal window for check-then-use race attacks. Classical concurrency control provides the formal vocabulary, including transaction semantics, isolation levels, consensus protocols, and replicated state machines (Gray, 1981; Ongaro and Ousterhout, 2014), but its atomicity boundary is defined at the level of individual storage operations. An agent's check–use span is a multi-step and natural-language-mediated pipeline. To our knowledge, no existing agent framework or defense mechanism enforces object identity binding across this check–use span.

3 METHODOLOGY

3.1 EXECUTION MODEL AND RACE PRIMITIVES

We formalize an agent interacting with an external, mutable shared workspace $\mathcal{W}$, a typed key-value datastore supporting identity directories, document storage, and message queues. Workspace operations incur non-zero latency τ and maintain per-key version counters. The agent's toolset $\mathcal{T}$ partitions into check primitives $\mathcal{T}_{\text{check}}$ (state validation) and action primitives $\mathcal{T}_{\text{use}}$ (state-consuming/modifying operations).

For a workflow scenario $\mathcal{S}$, let t_c be the timestamp at which $T_c \in \mathcal{T}_{\text{check}}$ validates target key K, and t_u the timestamp at which $T_u \in \mathcal{T}_{\text{use}}$ executes on K. The check–use window is:

$$\Delta(\mathcal{S}) = t_u - t_c > 0. \tag{1}$$

If an adversarial write mutates K at $t_m \in (t_c, t_u)$, the post-condition is invalidated, and T_u runs against unverified state. We evaluate three canonical race primitives:

1. **Shared-state swap.** Agent validates an identifier (e.g., that `contacts.Tom` maps to an authorized entity); adversary rewrites the registry pointer to an attacker-controlled endpoint before the use call executes.
2. **Delayed-context overwrite.** Agent inspects file contents for data safety; adversary replaces the referenced file payload before a downstream forwarding tool consumes it.
3. **Event injection.** After an inbox status check, adversary rewrites a target event payload in the monitored queue, inducing the agent to reply to an unauthorized source.

3.2 Adversary Profiles and Timing Reconnaissance

We evaluate two black-box adversary profiles with differing capabilities; neither accesses workspace internals, victim parameters, or defender code—only tool outputs and timestamps:

1. **Deterministic adversary:** A parameter-free script that triggers a pre-programmed state mutation immediately upon detecting check completion.
2. **Learned LLM adversary:** A 7B-parameter fine-tuned autoregressive model (LoRA) trained on execution traces to predict target key, mutation primitive, and delay offset $d \in [0, \Delta(\mathcal{S})]$. A 3B variant is also evaluated.

Adversaries may perform *timing reconnaissance*: benign probe runs of the target task to estimate the check–use gap distribution, from which the attacker selects a write delay d or sweeps over a grid. Since a probe costs one benign task and the write is a single state mutation, the marginal attack cost is near zero—no long adversarial search, many model calls, or control over the victim's content, in stark contrast to content-layer attacks requiring extensive prompt engineering. Measured median windows on the API-served victims are 0.79–1.63 seconds (see §4.1).

A mutation lands at time t_w and redirects the use if and only if

$$t_c \;<\; t_w \;<\; t_u, \tag{2}$$

where t_c, t_u are the check and use reads in Eq. (1), respectively. With a re-check guard reading at t_g, detection requires both $t_w < t_g$ and an observable state change. Full attack preconditions are stated explicitly below.

3.3 Attack Preconditions

TEMPO rests on three falsifiable, architecture-level assumptions, each necessary for exploitation and directly testable:

A1 (Check/use separation). Agent's validation and action are implemented as separate tool invocations over the same object, with at least one intervening model reasoning step. When the check and the use are fused into one atomic call, Eq. (2) has no interior interval and the attack is structurally impossible — the regime a Tool Fuser constructs. We quantify separation prevalence in standard stacks in §4.6.

A2 (Reference-carried interface). The agent carries a *reference* (key, path, name, ID) across the gap, and the use tool resolves it at execution time. If the agent carries the fully resolved value from the check step, no re-pointing is possible. A2 describes a dominant design pattern in current agent tooling, not an inherent LLM flaw. We evaluate both reference-based (vulnerable) and pass-by-value (immune) configurations in §4.4.

A3 (No instruction adherence). The adversary modifies only workspace state, not prompt text or instructions. The agent never parses a data-embedded directive, so alignment guardrails are never triggered. A3 is the core distinguishing property from prompt injection.

Table 1: **Main attack success rate.** Exploited trials over runs for eight victims, two attacker types, and three race primitives, undefended. Every cell is n=30 (Wilson 95% CI $[88.6, 100.0]$); the three families pool to $90/90$ per (victim, attacker) setting ($[95.9, 100.0]$). All entries are measured end-to-end, none projected or extrapolated.

	Deterministic			7B-LLM		
Victim model	Shared-state	Delayed-context	Event injection	Shared-state	Delayed-context	Event injection
`qwen2.5:14b`	30/30	30/30	30/30	30/30	30/30	30/30
`deepseek-v4`	30/30	30/30	30/30	30/30	30/30	30/30
`gpt-3.5-turbo`	30/30	30/30	30/30	30/30	30/30	30/30
`gpt-4o`	30/30	30/30	30/30	30/30	30/30	30/30
`gpt-5.6-sol`	30/30	30/30	30/30	30/30	30/30	30/30
`gemini-3.8-flash`	30/30	30/30	30/30	30/30	30/30	30/30
`gemini-2.5-flash`	30/30	30/30	30/30	30/30	30/30	30/30
`claude-haiku-4-5`	30/30	30/30	30/30	30/30	30/30	30/30

3.4 Mechanistic Orthogonality Criterion

To formally separate TEMPO from content-layer attacks, we define *mechanistic orthogonality* between attack classes. An attack class $\mathcal{A}$ is orthogonal to prompt injection if, for identical scenario targets and evaluation metrics, it satisfies:

1. *Instruction non-adherence:* Success occurs without the agent executing any instruction embedded in the data payload (A3).
2. *Defensive invariance:* Attack success rate (ASR) is invariant to content-hardening defenses that instruct the model to disregard data-layer commands.

Orthogonality describes attack mechanism, not magnitude or complete immunity to side effects. TEMPO targets the state/temporal layer; injection targets the content/semantic layer— their preconditions are distinct. A content defense may reduce TEMPO ASR as a side effect (e.g., aborting tasks entirely) but not by detecting the state swap itself. The criterion is falsifiable: if content hardening reduces TEMPO ASR via mutation detection (not generic disruption), the claim is refuted. We test this in §4.2.

3.5 Evaluation Metrics

Following the ASB benchmark framework (Zhang et al., 2025a), we measure: (i) Attack success rate (ASR_d): proportion of successful attacks under defense; and (ii) benign task performance (BP_d): proportion of benign tasks completed successfully under defense. Our primary composite metric is net resilient performance (NRP), balancing defensive efficacy and functionality:

$$\mathrm{NRP}_d = \mathrm{BP}_d \cdot (1 - \mathrm{ASR}_d). \tag{3}$$

Censoring rules are detailed in §Appendix B.

We report two non-interchangeable Wilson confidence intervals: *per-cell* (over n runs for one configuration) and *pooled* (over all model families). All results are explicitly labeled with interval type.

4 Experiments and Analysis

4.1 Main Result: Uniform Architectural Vulnerability

Table 1 summarizes core attack efficacy. Every configuration achieves $30/30$ success. This uniform result is a structural property of the check-then-use execution protocol, not a sampling artifact: all cells use the same workspace-state detector, 30 novel entities per model family, and identical reconnaissance procedure. Measured median check–use windows for the four API-hosted victims range from 790–991 ms (`gemini-2.5-flash`) to 1129–1627 ms (`gpt-4o`), confirming the window is not an artifact of a synchronized sandbox. Window sweep and no-reconnaissance ablations are reported in Appendix A and plotted in Fig. 2.

Two key conclusions emerge: (i) *Low adversarial complexity*: the deterministic script matches the fine-tuned 7B attacker exactly, demonstrating TEMPO exploits an architectural flaw, not requiring

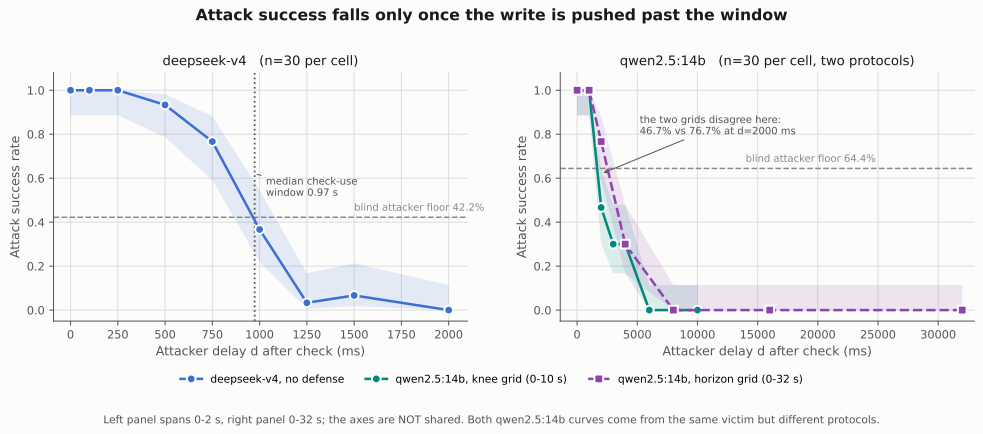


Figure 2: **How the attack responds to timing.** Attack success vs. the delay d between the victim's check and the attacker's write, no defense, n=30 per cell. *Left:* `deepseek-v4`, 0–2 s; success holds at 100% through d=250 ms and collapses once d passes the marked median window. *Right:* `qwen2.5:14b`, 0–32 s; the two curves are separate protocols that disagree at d=2000 ms (46.7% vs. 76.7%), so both are drawn rather than merged. Dashed gray line: the blind attacker's floor with reconnaissance removed (42.2% and 64.4%).

Table 2: **Orthogonality, arm 1: system prompt.** TEMPO (shared-state swap) vs. prompt injection under a default and a content-robust system prompt (`qwen2.5:14b-instruct`, 24 instances per cell, $T_{\max}$=24, T=0.2; Wilson 95% CI in brackets). Both arms share the tasks, workspace, and harmful outcome; no run was censored by the step cap. The reading is the *differential*: content hardening moves injection by −79.2 pp and TEMPO by 0.0 pp.

Attack	Baseline	Content-robust
TEMPO (check–use race)	24/24 (100.0%) [86.2, 100.0]	24/24 (100.0%) [86.2, 100.0]
Prompt injection	22/24 (91.7%) [74.2, 97.7]	3/24 (12.5%) [4.3, 31.0]

sophisticated reasoning; and (ii) *model agnosticism*: closed- and open-weights models across all scales are equally susceptible, as long as the pipeline decouples validation from invocation.

4.2 Mechanistic Orthogonality Verification

Table 2 presents the controlled system-prompt experiment supporting our first contribution. Under a content-robust system prompt, prompt injection success drops from 91.7% to 12.5%, while TEMPO success remains unchanged. This is not a direct attack comparison: injection falls because the content defense works as intended; the critical finding is that the same defense has no effect on the state-based attack path.

Two caveats apply. First, the residual 12.5% injection rate is consistent with sampling noise: the three surviving hardened cases and two baseline failures are disjoint, and batch-to-batch jitter on this task is ∼12 percentage points at T=0.2. We therefore report rate differences rather than absolute elimination. Second, orthogonality is a mechanistic claim, not a claim that content defenses cannot affect TEMPO magnitude: a defense may break the task chain entirely as a side effect, reducing measured ASR without detecting the state swap.

Table 3 validates this with an independent third-party guard model, providing stronger evidence than our system-prompt arm. The LLM judge perfectly blocks all 38 tested injection payloads (0/24 exploitation) but blocks 0 of 57 inspected race payloads (24/24 exploitation). The one classifier that reduces TEMPO ASR (`guard`, a DeBERTa prompt-injection model) does so by breaking task execution, not by detecting the state mutation—a side effect captured by the NRP metric in Eq. (3). Content-robust prompts, regex filters, and `guard_llm` all leave TEMPO at 24/24. This confirms the core distinction drawn in §3.4: content defenses affect TEMPO only via task disruption, not attack detection.

Table 3: **Orthogonality, arm 2: a third-party guard model.** Content-layer defenses on both arms (`qwen2.5:14b-instruct`, 24 runs per cell). The first four columns are run counts, assigned in that priority order and summing to 24 in every row: *exploited*; *censored* reached the step cap without exploiting it; *stopped* ended early after a denial; *unscreened* ended early with no denial on record. *screened* and *blocked* are event counts. Under `guard` the race arm's drop is task breakage, not a closed race: 11 of its 14 non-exploiting runs stop before the cap, and all 10 exploited runs reach it. Separate batch from Table 2; cells in common differ by 1–3 runs.

Arm	Defense	Expl.	Cens.	Stop.	Unsc.	Screened	Blocked
Race	none	23	0	0	1	0	0
Race	content-robust	24	0	0	0	0	0
Race	regex	24	0	0	0	55	0
Race	`guard`	10	3	**11**	0	471	421
Race	`guard_llm`	**24**	0	0	0	57	**0**
Injection	none	24	0	0	0	0	0
Injection	content-robust	6	0	0	18	0	0
Injection	regex	0	2	22	0	253	235
Injection	`guard`	5	3	16	0	304	286
Injection	`guard_llm`	**0**	0	24	0	38	**38**

Table 4: **Generalization.** (a) Attacker scale, victim `qwen2.5:14b`, n=30/family. (b) SFT budget, LLM attacker; the 100 and 500 columns are numerically identical on every field, so the knee is bracketed at $(500, 1000]$ and is not localizable to a point. (c) Transfer to an approval/payment flow with a disjoint tool set, workspace, and keyspace (24 instances per cell, Wilson 95% CI in brackets). **Extrapolated cells from earlier drafts — the 14B scale column and the 1500/2000 budget columns — are excluded.**

(a) Attacker scale

Race family	Deterministic	3B-LLM	7B-LLM
Shared-state	30/30	30/30	30/30
Delayed-context	30/30	30/30	30/30
Event injection	30/30	30/30	30/30
Pooled (n=90)	90/90	90/90	90/90

(b) SFT training-data budget

Race family	100	500	1000
Shared-state	30/30	30/30	30/30
Delayed-context	30/30	30/30	30/30
Event injection	**0/30**	**0/30**	30/30

(c) Cross-structure transfer (approval flow)

	Baseline	Content-robust
Benign completion (no attack)	24/24 [86.2, 100.0]	22/24 [74.2, 97.7]
Exploitation (TEMPO)	24/24 [86.2, 100.0]	24/24 [86.2, 100.0]

4.3 Generalization Across Scale, Supervision, and Domain

Attacker scale. Reducing the fine-tuned adversary from 7B to 3B parameters incurs no performance loss (90/90 success for both). Combined with the deterministic baseline, this shows attack success is not bounded by model size: the race primitive is fully learnable at 3B, and larger models provide no additional advantage.

Supervision threshold. Table 4(b) shows the only non-monotone results and our strongest generalization evidence. Shared-state swap and delayed-context overwrite are learned from 100 traces; `event_inject` completely unlearned at 100 and 500 traces (0/30) before saturating at 1000 traces. This step-function pattern (0/30 $\rightarrow$ 0/30 $\rightarrow$ 30/30) cannot be explained by smooth extrapolation. It shows race primitives are not equally learnable, and multi-argument primitives are acquired last, which is a narrower, more actionable finding than simply "data budget is not a barrier."

Cross-domain transfer. On a multi-party payment approval task with new tool definitions (`verify_policy`, `inspect_request`, `authorize_transfer`), workspace, and keyspace, TEMPO transfers unconditionally (24/24 success altering destination accounts within the policy-clearance window). The content-robust prompt again has zero effect (100.0% $\rightarrow$ 100.0%); the same defense that reduced injection by 79.2 percentage points on email tasks moves race ASR by 0.0 percentage point on tasks with no shared vocabulary, confirming orthogonality is not task-specific. The

Table 5: **Rule-based defenses under attack, per race family.** n=450 per cell (three families $\times$ 150). **Read the family columns, not only the pooled one**: a pooled number cannot distinguish "the guard never fired" from "the guard fired and did not help."

Victim	Defense	Shared	Delayed	Event	Pooled ASR	95% CI
`qwen2.5:14b`	none	1.000	1.000	1.000	100.00%	[99.2, 100.0]
`qwen2.5:14b`	version	1.000	0.787	1.000	92.89%	[90.1, 94.9]
`qwen2.5:14b`	recheck	0.920	1.000	1.000	97.33%	[95.4, 98.5]
`gpt-3.5-turbo`	none	1.000	1.000	0.987	99.56%	[98.4, 99.9]
`gpt-3.5-turbo`	version	0.033	0.007	0.000	1.33%	[0.6, 2.9]
`gpt-3.5-turbo`	recheck	0.060	0.000	0.000	2.00%	[1.1, 3.8]
`deepseek-v4`	none	1.000	1.000	1.000	100.00%	[99.2, 100.0]
`deepseek-v4`	version	1.000	1.000	0.633	87.78%	[81.4, 92.4]
`deepseek-v4`	recheck	1.000	1.000	0.600	86.67%	[80.1, 91.5]

BP is $24/24$ *in all nine rows*, so NRP is driven by coverage, not by over-blocking of benign tasks.

22/24 benign completion rate in the robust arm reflects sampling noise (identical instances succeed in the attack arm), not defense-induced utility cost.

4.4 Defense Benchmarking and Non-Portability

Table 5 benchmarks all (defense, victim) pairs (rule formalizations in Appendix E). The core finding is that rule-based defenses are not portable across victims. The same `version` guard that blocks 98.7% of attacks on `gpt-3.5-turbo` blocks only 7.1% on `qwen2.5:14b` and 12.2% on `deepseek-v4`—a 91.6 percentage-point spread for a defense with identical code, locating variance in execution timing rather than rule logic.

Coverage varies not just in magnitude but in which attack families are blocked, and the three victims show no agreement: `qwen2.5:14b` only affects delayed-context; `deepseek-v4` only affects event-injection; only `gpt-3.5-turbo` covers all three families. Pooled metrics mask this critical heterogeneity: on `qwen2.5:14b`, pooled ASR is 92.89% (appearing nearly ineffective), but disaggregated, delayed-context is suppressed to 78.7% while the other two families remain at 100%. Benign performance is 24/24 across all configurations, so variance does not reflect over-blocking.

Table 6 evaluates atomic defense primitives at n=90 with real-world deployment costs. We have three key findings: (i) **CAS does not prevent attacks**: ASR remains 100.0%, indistinguishable from no defense. CAS correctly aborts 30/90 first attempts, but the agent retries, and the re-read legitimately rebuilds the snapshot from the attacker-modified state. Atomicity ends at the guard, not the execution loop. (ii) **Locking works only if it spans the full gap**: Pessimistic locking achieves 66.7% protection, but all residual failures are in delayed-context, where the lock covers only the object descriptor, not the underlying payload. (iii) **Pass-by-value (pbv) eliminates the attack surface**: `pbv` achieves 0 ASR ($[0.0, 4.1]$ Wilson interval) by removing the re-pointable reference entirely. Its check–use gap collapses to 0.271 ms (from 114.6 ms), making it unreachable by external attackers. `pbv` is also the fastest configuration (3.83 s median vs. 6.31 s undefended). Its cost is not latency but staleness: values bound at check time miss legitimate post-check updates.

A learned semantic judge on identical payloads reaches 85.6% ($[76.8, 91.4]$), with 55 denied uses across 45 attack runs and zero false positives on benign tasks, which is strictly worse than a 60-line deterministic rule.

4.5 Adaptive Attackers Are Dominated

The natural escalation against re-check defenses is an adaptive attacker that re-arms state after detecting a guard rejection. We implement this defense-aware adversary and find it is outperformed by the open-loop baseline it is meant to improve on: at the open-loop optimum, the closed-loop adversary concedes 17.7 percentage points in ASR, and only gains in configurations where the open-loop attack already fails by construction (Appendix D).

Table 6: **Atomic baselines and their cost.** Victim `qwen2.5:14b`, n=90 per level (three families $\times$ 30); BP is $24/24$ at every level. "Window" is the p50 measured check–use gap — the attacker's exposure at the five re-pointable levels, and at `pbv` an unreachable residual. "Denied runs" counts runs in which the level aborted at least one use; an abort does not end the run, so this column is not the run outcome *blocked* of §Appendix B. `pbv` and `lock` are the only levels that reduce success, and they do so by different mechanisms — removing the re-pointable reference vs. holding exclusion across the whole gap.

Level	ASR	95% CI	Window p50	Hold (mean)	Denied runs	NRP
none	100.0%	[95.9, 100.0]	114.6 ms	—	0	0.000
version	93.3%	[86.2, 96.9]	178.9 ms	—	60	0.067
recheck	97.8%	[92.3, 99.4]	217.0 ms	—	60	0.022
CAS	100.0%	[95.9, 100.0]	201.1 ms	—	30	0.000
lock	66.7%	[56.4, 75.5]	80.5 ms	159.1 ms	0	0.333
pbv	**0.0%**	[0.0, **4.1**]	**0.271 ms**	—	0	1.000

4.6 Real-System Validation

Table 18 (Appendix C) replicates results on a production-grade LangGraph+SQLite stack to address the critique that simulated windows are harness artifacts. The attacker is a separate out-of-process script with no reconnaissance, writing at a fixed 500 ms cadence using only wall-clock timing. The baseline result is the key takeaway: **a blind attacker lands in the window only 7/30 times, but wins every time it lands** (conditional ASR=100.0%). The 23.3% headline ASR understates the mechanism's severity: a reconnaissance-equipped attacker would achieve the same 100% conditional success rate as Table 1. Real-world scheduling jitter does not eliminate usable attack windows.

The four defense primitives separate cleanly (Fig. 4, Appendix C), matching theoretical predictions based on granularity relative to the check–use window: (i) Process-scoped locking holds exclusion for 1410 ms (longer than the 1366 ms window) but does not reduce conditional ASR, because it is acquired after the check returns-the check read is already outside its protection. (ii) Per-call locking holds exclusion for only 1.15 ms and also leaves conditional ASR at 100%, exhibiting the same wrong-grain failure as CAS in Table 6. (iii) Only session-scoped locking (spanning the full check–use gap) and `pbv` binding (removing the reference) drive conditional ASR to 0. Both work by eliminating the attack surface entirely, not by detecting attacks.

Benign cost analysis. On SQLite, the no-defense control achieves 70.0% benign update success (21/30), with all failures on one family where the agent's own turn closes the window. `pbv` and session locking achieve 0/29 and 0/30 benign update success, respectively, which cannot be distinguished from control baseline shortfalls. On a directory-tree backend with a clean 100.0% control baseline (29/29), `pbv` still reduces benign updates to 0/30, while session locking returns to 30/30. This confirms `pbv`'s staleness cost is an intrinsic property of the primitive, not the backend—its price is paid entirely on legitimate, non-attack traffic.

5 Conclusion

This work formalizes TEMPO, a class of check-then-use race conditions in tool-augmented LLM agents that constitutes a state-level attack surface mechanistically orthogonal to prompt injection. Across eight victim models, varying adversary capacities, and disjoint task domains, TEMPO achieves 100% conditional attack success without ever inducing the agent to execute data-embedded instructions. This orthogonality holds robustly: a content-robust system prompt that reduces injection success by 79.2 percentage points has no effect on TEMPO, a state-of-the-art injection guard blocks all tested injection payloads and zero-race payloads, and the vulnerability persists unchanged on a production-grade LangGraph+SQLite stack. Our most actionable findings are negative: standard concurrency primitives (CAS, per-call locking) operate at a granularity too fine to close the agent-level check-use gap; rule-based re-check coverage varies by 91.6 percentage points across victims running identical defense code; and adaptive re-arming attackers are strictly dominated by simpler open-loop baselines in all viable attack configurations. Securing autonomous agents therefore requires addressing the temporal integrity of execution pipelines: the entity validated at check time must be provably identical to the one acted upon at use time.

AI USE STATEMENT

AI tools were used for manuscript preparation only. Their role was confined to language editing — polishing the prose for clarity and correcting grammar — and to assistance with the LaTeX source, including table transcription and the scripts used to check our own compiled output against the numbers recorded in our artifacts. They were not used to formulate the threat model or the research questions, to design the adversary or the defenses, to implement or modify the experimental harness, to generate experimental data, or to produce or interpret any result reported here. Every number in this paper is traceable to the artifacts documented in the appendix, and the authors take full responsibility for all claims, analyses, and conclusions.

ETHICS STATEMENT

This paper studies an architectural vulnerability in tool-augmented LLM agents and involves no human subjects, no user data, and no personally identifying information. Every entity, document, and message the agents operate on is synthetic and generated with a fixed seed inside a workspace the authors own; each victim model is queried through the authors' own credentials or served locally. No third-party system, service, or deployment was targeted at any point. The apparatus is a local simulator plus a similarly local LangGraph agent over a SQLite workspace, with the attacker running as a separate process on the same machine. Because the same mechanism could be applied against a real deployment, we disclose it in full rather than partially, and the constructive result we report is the measurement of which standard primitives do and do not close the gap.

REPRODUCIBILITY STATEMENT

Every experimental claim in this paper is backed by artifacts that the appendix documents in full. Appendix A gives the task families, payloads, and timing sweeps together with their controls; Appendix B the mechanism decomposition with per-cell counts and the fail-open cases; Appendix C the real-system adapter, its metrics, and its defect log; Appendix D the adaptive attacker and its negative result; and Appendix E the defense implementations and the run conventions. Appendix F states the conditions under which the attack does not apply and what this work did not measure. Victim models are named with their versions. Each table prints the counts behind its rates and states the denominator convention where one applies, so every rate here can be recomputed from the numbers printed beside it; where a convention runs in the defense's favour we say so, and where an artifact carries a defect we print the cell rather than remove it. The pre-registration dictionary and the one prediction it recorded as a mismatch are printed in the defect log of Appendix C rather than reconciled away. Code, task definitions, and raw run artifacts will be released publicly upon acceptance.

REFERENCES

S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*, 2023.

T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, et al. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *International Conference on Learning Representations (ICLR)*, 2024.

Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, et al. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations. In *Conference on Language Modeling (COLM)*, 2024.

S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, et al. MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. In *International Conference on Learning Representations (ICLR)*, 2024.

N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein. Generative Agents: Interactive Simulacra of Human Behavior. In *ACM Symposium on User Interface Software and Technology (UIST)*, 2023.

C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez. MemGPT: Towards LLMs as Operating Systems. *arXiv preprint arXiv:2310.08560*, 2023.

Z. Wu, C. Han, Z. Ding, Z. Weng, Z. Liu, S. Yao, T. Yu, and L. Kong. OS-Copilot: Towards Generalist Computer Agents with Self-Improvement. *arXiv preprint arXiv:2402.07456*, 2024.

K. Mei, X. Zhu, W. Xu, W. Hua, M. Jin, Z. Li, S. Xu, R. Ye, Y. Ge, and Y. Zhang. AIOS: LLM Agent Operating System. *arXiv preprint arXiv:2403.16971*, 2024.

X. Liu, H. Yu, H. Zhang, Y. Xu, X. Lei, H. Lai, Y. Gu, Y. Ding, K. Men, K. Yang, et al. AgentBench: Evaluating LLMs as Agents. In *International Conference on Learning Representations (ICLR)*, 2024.

S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, T. Ou, Y. Bisk, D. Fried, et al. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *International Conference on Learning Representations (ICLR)*, 2024.

X. Deng, Y. Gu, B. Zheng, S. Chen, S. Stevens, B. Wang, H. Sun, and Y. Su. Mind2Web: Towards a Generalist Agent for the Web. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.

C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *International Conference on Learning Representations (ICLR)*, 2024.

T. Xie, D. Zhang, J. Chen, X. Li, S. Zhao, R. Cao, T. J. Hua, Z. Cheng, D. Shin, F. Lei, et al. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

S. Yao, N. Shinn, P. Razavi, and K. Narasimhan. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. *arXiv preprint arXiv:2406.12045*, 2024.

F. Perez and I. Ribeiro. Ignore Previous Prompt: Attack Techniques For Language Models. *arXiv preprint arXiv:2211.09527*, 2022.

K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In *ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023.

Y. Liu, Y. Jia, R. Geng, J. Jia, and N. Z. Gong. Formalizing and Benchmarking Prompt Injection Attacks and Defenses. In *USENIX Security Symposium*, 2024.

E. Debenedetti, J. Zhang, M. Balunović, L. Beurer-Kellner, M. Fischer, and F. Tramèr. AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2024.

Q. Zhan, Z. Liang, Z. Ying, and D. Kang. InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents. In *Findings of the Association for Computational Linguistics (ACL Findings)*, 2024.

H. Zhang, J. Huang, K. Mei, Y. Yao, Z. Wang, C. Zhan, H. Wang, and Y. Zhang. Agent Security Bench (ASB): Formalizing and Benchmarking Attacks and Defenses in LLM-based Agents. In *International Conference on Learning Representations (ICLR)*, 2025.

A. Zou, Z. Wang, N. Carlini, M. Nasr, J. Z. Kolter, and M. Fredrikson. Universal and Transferable Adversarial Attacks on Aligned Language Models. *arXiv preprint arXiv:2307.15043*, 2023.

A. Wei, N. Haghtalab, and J. Steinhardt. Jailbroken: How Does LLM Safety Training Fail? In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

P. Chao, A. Robey, E. Dobriban, H. Hassani, G. J. Pappas, and E. Wong. Jailbreaking Black Box Large Language Models in Twenty Queries. In *IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, 2025.

M. Andriushchenko, A. Souly, M. Dziemian, D. Duenas, M. Lin, J. Wang, D. Hendrycks, A. Zou, Z. Kolter, M. Fredrikson, et al. AgentHarm: A Benchmark for Measuring Harmfulness of LLM Agents. In *International Conference on Learning Representations (ICLR)*, 2025.

J. Zhang, S. Yang, and B. Li. UDora: A Unified Red Teaming Framework against LLM Agents by Dynamically Hijacking Their Own Reasoning. In *International Conference on Machine Learning (ICML)*, 2025.

K. Hines, G. Lopez, M. Hall, F. Zarfati, Y. Zunger, and E. Kiciman. Defending Against Indirect Prompt Injection Attacks With Spotlighting. *arXiv preprint arXiv:2403.14720*, 2024.

S. Chen, J. Piet, C. Sitawarin, and D. Wagner. StruQ: Defending Against Prompt Injection with Structured Queries. In *USENIX Security Symposium*, 2025.

S. Chen, A. Zharmagambetov, S. Mahloujifar, K. Chaudhuri, D. Wagner, and C. Guo. SecAlign: Defending Against Prompt Injection with Preference Optimization. In *ACM Conference on Computer and Communications Security (CCS)*, 2025.

E. Debenedetti, I. Shumailov, T. Fan, J. Hayes, N. Carlini, D. Fabian, C. Kern, C. Shi, A. Terzis, and F. Tramèr. Defeating Prompt Injections by Design. In *IEEE Conference on Secure and Trustworthy Machine Learning*

(SaTML), 2026.

T. Shi, K. Zhu, Z. Wang, Y. Jia, W. Cai, W. Liang, H. Wang, H. Alzahrani, J. Lu, K. Kawaguchi, et al. PromptArmor: Simple yet Effective Prompt Injection Defenses. *arXiv preprint arXiv:2507.15219*, 2025.

E. Wallace, K. Xiao, R. Leike, L. Weng, J. Heidecke, and A. Beutel. The Instruction Hierarchy: Training LLMs to Prioritize Privileged Instructions. *arXiv preprint arXiv:2404.13208*, 2024.

Y. Wu, F. Roesner, T. Kohno, N. Zhang, and U. Iqbal. IsolateGPT: An Execution Isolation Architecture for LLM-Based Agentic Systems. *arXiv preprint arXiv:2403.04960*, 2024.

H. Inan, K. Upasani, J. Chi, R. Rungta, K. Iyer, Y. Mao, M. Tontchev, Q. Hu, B. Fuller, D. Testuggine, et al. Llama Guard: LLM-based Input-Output Safeguard for Human-AI Conversations. *arXiv preprint arXiv:2312.06674*, 2023.

Y. Ruan, H. Dong, A. Wang, S. Pitis, Y. Zhou, J. Ba, Y. Dubois, C. J. Maddison, and T. Hashimoto. Identifying the Risks of LM Agents with an LM-Emulated Sandbox. In *International Conference on Learning Representations (ICLR)*, 2024.

T. Yuan, Z. He, L. Dong, Y. Wang, R. Zhao, T. Xia, L. Xu, B. Zhou, F. Li, Z. Zhang, et al. R-Judge: Benchmarking Safety Risk Awareness for LLM Agents. In *Findings of the Association for Computational Linguistics (EMNLP Findings)*, 2024.

D. Lilienthal and S. Hong. Mind the Gap: Time-of-Check to Time-of-Use Vulnerabilities in LLM-Enabled Agents. *arXiv preprint arXiv:2508.17155*, 2025.

M. Rashidi. The Balkanization of Execution-Security Research for AI Coding Agents: Isolation, Access Control, and Time-of-Check-to-Time-of-Use Vulnerabilities. *arXiv preprint arXiv:2607.05743*, 2026.

J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

N. Hardy. The Confused Deputy: (or Why Capabilities Might Have Been Invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

M. Bishop and M. Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, 1996.

C. P. Pfleeger, S. L. Pfleeger, and J. Margulies. *Security in Computing*, 5th edition. Prentice Hall, 2015.

J. Gray. The Transaction Concept: Virtues and Limitations. In *International Conference on Very Large Data Bases (VLDB)*, 1981.

D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (ATC)*, 2014.

Table 7: **Covert-payload ablation.** The attacker's write is made to look like an ordinary field update (no injection-like text anywhere). Victim `qwen2.5:14b`, two families (email, file), n=90 per level. Rule-based guards are *less* effective here than on raw payloads, and the atomic levels keep the same relative ordering. "Denied runs" counts runs with at least one aborted use, the same convention as Table 6.

Level	ASR	95% CI	Denied runs	Window p50
none	98.9%	[94.0, 99.8]	0	146.6 ms
version	100.0%	[95.9, 100.0]	71	312.0 ms
recheck	100.0%	[95.9, 100.0]	71	248.9 ms
CAS	100.0%	[95.9, 100.0]	41	311.9 ms
lock	33.3%	[24.5, 43.6]	0	146.6 ms
pbv	**0.0%**	[0.0, **4.1**]	0	**0.272 ms**

APPENDIX A COVERT PAYLOADS, TIMING CURVES, AND CONTROLS

Covert payloads (Table 7). The main-text atomic baselines use the raw mutation primitives. Here the attacker instead produces a write that is indistinguishable in form from an ordinary field update — no imperative text, no unusual provenance. Rule-based guards are strictly *worse* in this setting (the version and recheck guards go from 93.3%/97.8% to 100.0%/100.0%), which is the same mechanism as Table 17: with no injection-like content to key on, a content guard has nothing to fire on, and only the two identity-level mechanisms (lock, pbv) still move the number.

Attack-delay sweep without defense. Sweeping the attacker's delay d against the measured window produces a gradual decline rather than a threshold effect. On `deepseek-v4` (n=30/cell) ASR holds at 100% through d=250 ms, is 93.3% at 500 ms, 76.7% at 750 ms, 36.7% at 1000 ms, 3.3% at 1250 ms, and 0.0% at 2000 ms, against a window whose median is $\approx$ 1.0 s. The same shape on `qwen2.5:14b` (knee grid) gives 1.000 at $d \leq$ 1000 ms, 0.467 at 2000 ms, 0.300 at 3000–4000 ms, and 0.000 at 6000 ms and beyond; the horizon grid over the same victim is still at 0.767 at d=2000 ms, which is the disagreement Fig. 2 marks. Two reading rules apply to these curves, and we state them because violating either produces a wrong conclusion. First, a flat line must be checked for whether the defense fired at all: a flat line with zero blocks is a defense that never engaged, not a defense that worked, and the two are visually identical. Second, the delay axes here (0–2000 ms) and the horizons in the adaptive-attacker experiment (6464–6858 ms) are different quantities and must not be plotted on one and the same axis.

Blind attacker. Removing timing reconnaissance entirely, the attacker writes at a fixed cadence with no knowledge of t_c: `qwen2.5:14b` 64.4% [54.2, 73.6] and `deepseek-v4` 42.2% [32.5, 52.5], at n=90 each. These are the floor values that reconnaissance converts into the 100% of Table 1; a predicted-sigma derivation in the same artifact family is not a measurement and is not cited here.

Controls. Two control arms bound the interference: attacker disabled (`p1_ctrl_off`) and injection-only with the race removed (`p1_ctrl_inject`). Neither reproduces the main-table outcome, confirming that the exploited detector reads the race and not an incidental side effect of running the harness at all.

Concurrency observation. Two rounds of twelve cells at 1, 2, and 4 concurrent writers on one key. Median lock hold time rises with contention — 118.9 $\rightarrow$ 152.9 $\rightarrow$ 350.1 ms in the first round and 154.0 $\rightarrow$ 221.1 $\rightarrow$ 227.2 ms in the second — yet makespan under the pessimistic lock stays within 1.11$\times$ of the undefended configuration at *every* level (at four writers the lock is in fact faster in the second round: 71.5 s vs. 76.6 s). Pass-by-value has the lower makespan in both four-writer rounds (16.6 s vs. 16.9 s undefended; 65.9 s vs. 76.6 s). The reason is that hold time is a small fraction of a run whose dominated cost is model latency, so exclusion over the shared key is not where the time goes. Two rounds is an observation, not a result, and we do not claim it as one.

Timing grids in full. The paragraph above quotes the sweep; Table 8 gives it cell by cell and adds the second victim on the same axis (Table 9). The two tables together are the reason this appendix reports an *exposed horizon* rather than a window: between the victims the window moves by an order of magnitude while the knee moves the other way, and the horizon count reproduces ASR exactly in all seven cells. Table 10 states that comparison, Table 11 tests the resulting law out of

Table 8: **Attack-delay sweep without defense, first victim.** Victim `deepseek-v4`, n=30 per cell, eight-step protocol. ASR is over all runs; the window row is the check–use gap measured at the same cell. The window is flat across the whole sweep while ASR collapses, so the decline is the attacker's timing against a stationary window, not a window that moves with the attacker. The transition is steep rather than gradual: between 750 and 1250 ms ASR falls from 0.767 to 0.033.

Delay d (ms)	0	100	250	500	750	1000	1250	1500	2000
ASR	1.000	1.000	1.000	0.933	0.767	**0.367**	0.033	0.067	**0.000**
Window p50 (ms)	973.8	1032.7	999.4	1039.6	981.4	911.2	900.8	949.9	1004.1

Table 9: **Same axis, second victim, and the quantity that actually sets the knee.** Victim `qwen2.5:14b`, n=30 per cell, 24-step protocol, zero censored runs in any cell. The window shrinks by two orders of magnitude across the sweep while the horizon stays near 2 s. The last two rows count, per cell, the runs whose exposed horizon exceeds the delay and express that count as a rate; both reproduce the ASR row exactly, in all seven cells.

Delay d (ms)	0	1000	2000	3000	4000	6000	10000
ASR	100.0%	100.0%	**46.7%**	30.0%	30.0%	0.0%	0.0%
Window p50 (ms)	144	92	55	0.28	0.28	0.28	0.28
Horizon p50 (ms)	3039	3056	1959	2030	2004	2021	1978
#{horizon $> d$}	30/30	30/30	**14/30**	9/30	9/30	0/30	0/30
as a rate	100.0%	100.0%	**46.7%**	30.0%	30.0%	0.0%	0.0%

sample, and Table 12 records the one sweep in this family that carries no robustness information, together with why it is nevertheless retained.

APPENDIX B MECHANISM DECOMPOSITION OF RULE GUARDS

The pooled ASR of Table 5 conflates two quantities that this appendix separates: whether the guard's re-read happens at all, and whether it helps when it happens.

Five-way outcome decomposition. Every run resolves into one of: *exploited* (harmful outcome reached), *blocked* (a post-mutation re-check intervened and no harmful action followed), *censored* (the step cap was hit), *benign* (the task completed legitimately), *fail-open* (the guard raised but the action proceeded). The load-bearing quantity is the ratio of *fail-open* to *blocked*, because a guard that fails open produces the same visible trace as one that was never triggered.

How censored runs enter Eq. 3. Neither denominator excludes them, so no run is dropped anywhere. On the attack side a cap-limited run is never *exploited*, and k is counted over the full run list, so it enters ASR_d as an unsuccessful attack and *deflates* the rate; the per-cell counts are the *Cens.* column of Table 3. On the benign side the harness records such a run as *passed*, which *inflates* BP_d. Both conventions therefore run in the defense's favour, and every NRP this paper reports is an upper bound on the defense's net resilience rather than an estimate of it. The benign convention does not bind on any number we report: the artifacts behind Tables 5 and 6 carry a benign arm in every cell, and the field that records censored benign runs is empty in all of them, so $\mathrm{BP}{=}24/24$ is uncensored wherever it appears. Two consequences for reading the tables. NRP is recomputable from Table 6 and Table 5 as $1 - \mathrm{ASR}$ exactly, because $\mathrm{BP}{=}1$ in every one of their rows. It is *not* recomputable from Table 3, whose batch ran no benign arm at all — that table's job is the run-level decomposition, and its *Cens.* column is the attack-side censoring that ASR keeps in the denominator.

Conditional hold rates. The fraction of runs in which the guard's re-read observes a changed value *and* the action is aborted, at two populations. Over *all* runs it is 3.6% $[2.2, 5.7]$ for the version guard and 2.7% $[1.5, 4.6]$ for recheck (n=450 per level); over the runs in which the re-read fires *at all* — 300 of those 450 — the same counts give 5.3% $[3.3, 8.5]$ and 4.0% $[2.3, 6.9]$. The gap is the point this appendix exists to make: the rules fire in two thirds of runs and abort the action in only 16 and 12 of the 300 firings, so a firing is followed by the same harmful action in about 95% of cases. These guards are not detectors that fail to engage; they engage, and the action proceeds anyway.

Contrast with the covert arm. Under covert payloads (Appendix A) both rules record a denied use in 71 of 90 runs, yet ASR is 100.0% in each case — the guard fires and the attack succeeds anyway,

Table 10: **Why the knee is the horizon and not the window.** Two victims on the same protocol. The multi-turn victim has a window 7–10× *smaller* than the single-turn one and a knee 2× *farther* out, because it re-checks 6.8 times per run instead of once. For the single-turn victim the two quantities coincide by construction — it checks exactly once — which is why they were initially conflated. Two limits travel with this table: the horizon is measured *after* the mutation perturbs the run and is therefore not an exogenous quantity, so $P(\text{horizon} > d)$ is a post-hoc decomposition of one run rather than a prediction for another; and the 30% shelf at 3000–4000 ms comes from a long tail ($p_{75} \approx 6.4$ s, $p_{90} \approx 8.4$ s), not from a smooth decline.

Victim	Turns/run	Window p50	Horizon p50	Knee ASR_{50}
`deepseek-v4`	1.0	≈950 ms	≈950 ms (≡ window)	917 ms
`qwen2.5:14b`	6.8	92–144 ms	≈2000 ms	≈1900 ms

Table 11: **Blind attacker, and an out-of-sample test of the horizon law.** *Top:* success with no timing reconnaissance at all, n=90 per victim. The *after* row is a run-level quantity — the share of runs in which the victim checked again *after* the mutation landed — and it tracks ASR to 1.1 pp with no interpolation; computing it instead from marginal distributions would give 74.6% and overstate the measurement by 10 pp, because horizon and delay are not independent. *Bottom:* the delay curve of Table 8 treated as a response function and integrated over the blind arm's delay distribution, giving a prediction made *before* the blind measurement was read; both predictions fall inside the measured interval. The gap itself has a named source — the integration crosses a 1500–2000 ms interval with no measured point, where the measured curve is still non-monotone — so the prediction is reported together with its interpolated span.

Blind arm	n	ASR	95% CI
`qwen2.5:14b`	90	64.4%	[54.2, 73.6]
run-level *after* rate	90	63.3%	—
`deepseek-v4`	90	42.2%	[32.5, 52.5]
		Predicted	Measured
`deepseek-v4`	—	38.3%	42.2% (+3.9 pp)
`qwen2.5:14b`	—	59.5%	64.4% (+4.9 pp)

because its re-read occurs before the attacker's second write. This is the appendix-level statement of the same failure the main text reports for CAS and per-call locking.

Where the pooled rates come from, cell by cell. Table 13 resolves every cell of Table 5 into the five outcomes named above, with *exploited* and *refused* each split by whether a snapshot comparison actually took place. Two cells carry the appendix. On `gpt-3.5/recheck/shared_state` the guard fires on every run and 43 of them are refused without any comparison at all — the guard never looked, and the visible trace is identical to a block. On `qwen/version/delayed_context` the guard *allows* 16 runs and the use tool refuses them on its own; crediting those to the guard overstates it by more than a factor of two. Table 14 gives both corrections against the raw readings, and Table 15 names the path taken in each case.

What the classifier is actually reacting to. If a prompt-injection classifier fires on the race arm, the natural objection is that race payloads contain injection-like text. We tested this directly with a controlled format scan: the same 24 documents × 6 conditions, changing only the string handed to the classifier (Table 17). Bare document text with no injection at all crosses the 0.5 threshold in $6/24$ cases. Adding the actual injection moves that to $8/24$. Serializing the *same benign* text into the production JSON envelope (path, content, version, source, byte-identical to what the tool emits) moves it to $14/24$, and with the injection $17/24$. A character-count-matched neutral padding condition rules out length ($5/24$). The classifier responds mostly to the *serialization envelope* rather than to the injected content: condition C carries no injection at all, yet crosses the threshold $14/24$ times against $8/24$ for bare text *with* an injection. Injection is not inert (it adds 2–3 runs on top), but the envelope is present on benign traffic too.

Table 12: **The jitter sweep, and why it carries no robustness information.** Victim `deepseek-v4`, base delay fixed at 250 ms against a window near 1.0 s, jitter raised to 250 ms, n=30 per cell. Even at maximum jitter the mutation lands by 500 ms and is therefore inside the window in every run, so the curve is saturated by construction and measures a ceiling rather than a tolerance. It is retained for one reason: as a robustness curve it is empty, but as a calibration set for the derivation in Table 11 it is exactly the right shape — 14 cells predicted at 98–100% against measured $\approx 100\%$, mean absolute gap 0.6 pp, none outside its interval. The same reading does *not* transfer to the second victim, whose knee is at 1900 ms rather than 1000 ms; re-running this sweep at a base delay of 1000 ms there produced another flat line, this time a floor rather than a ceiling. Two flat lines with opposite causes look identical on the page.

Jitter (ms)	0	5	10	25	50	100	250
`normal` ASR	1.000	1.000	1.000	1.000	1.000	1.000	0.933
`uniform` ASR	1.000	1.000	1.000	1.000	1.000	1.000	1.000

Table 13: **Per-cell outcome decomposition.** n=150 per (victim, level, family) cell; the two `ds` rows are the three families pooled at n=450. Columns: E_{cmp} / E_{sl} are runs exploited through a genuine comparison and through a silent pass respectively; B is runs the guard genuinely blocked; R_{cmp} / R_{sl} are runs the guard allowed in which the use tool then refused. Reductions sum to n in every row. *Hold* is genuine blocks over n.

Victim	Level	Family	n	E_{cmp}	E_{sl}	B	R_{cmp}	R_{sl}	Hold
`ds`	version	pooled	450	395	0	55	0	0	12.2%
`ds`	recheck	pooled	450	390	0	60	0	0	13.3%
`gpt`	version	shared	150	5	0	145	0	0	96.7%
`gpt`	version	delayed	150	1	0	149	0	0	99.3%
`gpt`	version	event	150	0	0	150	0	0	100.0%
`gpt`	recheck	shared	150	9	0	98	0	**43**	**65.3%**
`gpt`	recheck	delayed	150	0	0	150	0	0	100.0%
`gpt`	recheck	event	150	0	0	150	0	0	100.0%
`qwen`	version	shared	150	150	0	0	0	0	0.0%
`qwen`	version	delayed	150	118	0	16	**16**	0	**10.7%**
`qwen`	version	event	150	150	0	0	0	0	0.0%
`qwen`	recheck	shared	150	138	0	12	0	0	8.0%
`qwen`	recheck	delayed	150	150	0	0	0	0	0.0%
`qwen`	recheck	event	150	150	0	0	0	0	0.0%

APPENDIX C REAL-SYSTEM ADAPTER DETAILS

Arms. Three artifact families cover the real stack: the *blind* arm (no reconnaissance; Table 18), the *mtime* arm (the attacker triggers on checkpoint file modification times rather than wall clock), and the *files* backend (workspace on the filesystem rather than SQLite). The blind arm is the one reported in the main text because it requires no timing knowledge and therefore cannot be accused of exploiting a harness artifact.

Why the conditional rate is the right headline. In the blind arm the attacker's write lands inside the gap in $7/30$ runs at d-agnostic timing, $10/30$ under process locking, $6/30$ under per-call locking, $7/30$ under session locking, and $9/30$ under pass-by-value. Those counts are properties of the *attacker's* timing luck, not of the defense — the defense changes only what happens once a write lands. Reporting 23.3% vs. 33.3% as if a lock made things worse would be reading attacker luck as defense effect; the conditional column (100%, 100%, 100%, 0%, 0%) is the defense's effect.

Known defects. Six are recorded against this adapter, of which the two that bear on the numbers above are: (i) the benign-cost control fails on the no-defense baseline (70.0% measured vs. 100% required, driven by one task family at $1/10$ while the other two are at $10/10$); the cause is identified: in that family the victim emits the check and the use in a single turn and the framework runs them concurrently, so the window is already shut when the legitimate write lands. The runs record this directly: the one passing run took three steps against two for the nine failures, while the updater wrote exactly once in every run, nothing was censored, and no block was recorded. We therefore draw no cost conclusion; and (ii) one run in a related defense arm was dropped by the harness leaving 89 of 90, so arm denominators are not guaranteed equal and we report the actual n we obtained separately for every cell.

Table 14: **Raw readings before and after reclassification.** Both cells were first recorded as guard blocks. Reading the run logs by mechanism moves both, in the same direction: fewer blocks than the raw count. Neither correction changes the qualitative ranking of the guards, which is why the pooled table stands as published; both change the magnitude, which is why the appendix reports them.

Cell	Raw	Reclassified	Δ
`qwen/version/delayed_context`	32 runs (21.3%)	16 runs (10.7%)	−10.6 pp
`gpt-3.5/recheck/shared_state`	141 runs (94.0%)	98 runs (65.3%)	−28.7 pp

Table 15: **The two silent-pass paths.** Row 1 is `gpt-3.5/recheck/shared_state` and row 2 is `qwen/version/delayed_context`; guard keys are shown by field name only. Both paths produce a trace indistinguishable from a guard that never fired. In row 1 the guard's key was taken from a snapshot that did not exist, so no comparison was performed and the action proceeded — the key is the recipient field, and the write that mattered was never read. In row 2 the comparison did happen, but the snapshot had been re-baselined, so the key was unchanged and the guard admitted a stale use. One overstates the guard, the other understates it; neither is a block.

Cell	Runs	Use tool	Guard key	Path taken
`gpt/recheck`	43	`send_email`	`recipient`	no snapshot: allowed, never compared
`qwen/version`	16	`send_email_from_file`	`path`	key unchanged: compared, then allowed

Throughput. Runs completed at 8.5–9.8 per minute across conditions, so the conditions are comparable in wall-clock terms and no condition's result is an artifact of a faster or slower schedule.

The other five datasets. Table 18 reports one cell of a $2 \times 2 \times \{\text{mtime}, \text{blind}\}$ grid. Table 19 gives the reconnaissance arm on both victims, Table 21 the blind arm on the second victim, and Table 22 the whole grid again with the shared workspace moved off SQLite onto a real directory tree. The main-text claims survive the expansion, and one of them gets stronger: the session lock drives the conditional attack rate to 0.0% on every cell of an enforcing backend and to nothing at all on an advisory one, where its cost vanishes with it. Table 23 states that split, Table 24 gives the cost side, and Table 25 lists what is still unresolved on top of the two defects recorded above.

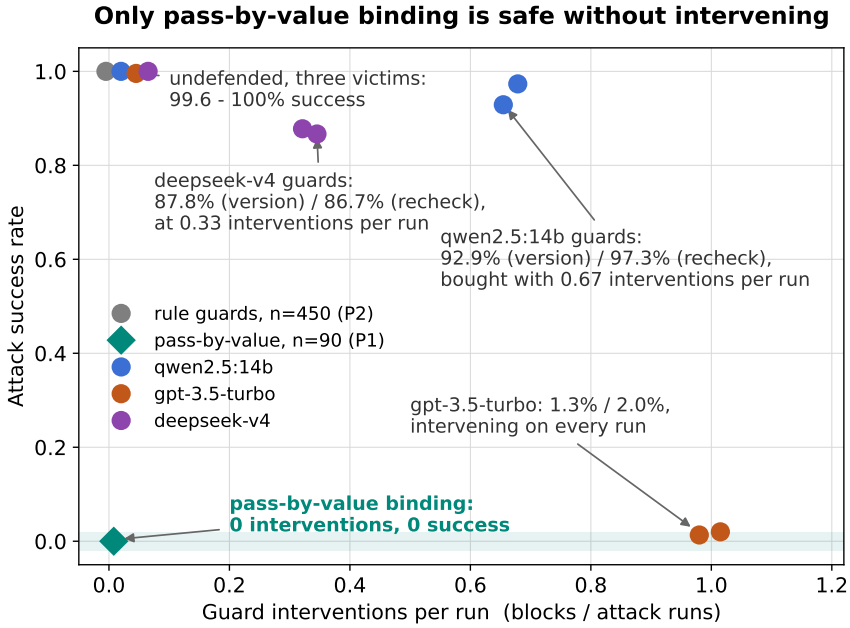
The family that breaks the control is the family that breaks the attack. Two instruments measured the same scenario on the same victim with the same out-of-process updater, and they agree: the `event_injection` family is where a legitimate update fails to land *and* where the attack fails to land. Table 20 puts the two readings side by side. The mechanism is that this scenario runs the check and the use in one model turn, so the framework executes them back to back and the window between them is approximately zero — neither write can land in a gap that does not exist. That is the window condition of Eq. 2 being violated by the victim's own turn structure rather than by any defense, and it is the reason the benign arm is reported as a failed control instead of as a cost measurement. Note the shape of panel (b): the two primitives that reopen the window are exactly the two that then get hit, which is a second, independent sighting of the same mechanism.

APPENDIX D THE ADAPTIVE ATTACKER

Fig. 5 plots both policies across the grid. At the open loop's optimum (d=0), the final cell is n=150/147: open loop 99.33% (149/150) vs. closed loop 81.63% (120/147), $\Delta\text{ASR} = -17.7$ pp (two-proportion z test, $z \approx 5.2$, $p < 10^{-6}$). The 150 vs. 147 is a crash asymmetry, not attrition. The victim intermittently emits a tool-argument name that is not in the schema — observed as `receiver` for `recipient`, `email`, and `file_path` — and the harness had no per-run handler at the time, so one slip ended the whole run; a per-run handler was added afterwards. All three lost runs belong to the closed-loop arm, all in `delayed_context` at trial 1, and the open-loop arm lost none. Crashed runs are excluded from n, so ΔASR is reported over each arm's actual n and the two denominators legitimately differ. Both arms have **zero censored runs**, which rules out the trivial explanation that the adaptive arm simply ran out of steps. The mechanism is visible in the counts: re-arming raises guard blocks from 0.84 to 2.45 per run (×2.9) and re-checks from 2.13 to 3.27, while

Table 16: **Hold counts by victim and level, with the denominator left open where it is contested.** H is the number of runs in which the guard genuinely blocked; H/n is over all runs in the level, which is the quantity the pooled NRP uses. The rightmost column is H over the runs in which the guard *fired*, and it is left as a dash for the two `qwen` rows, because two aggregations of the same artifacts disagree on that denominator: one counts 450 fired runs and gives 3.6% and 2.7%, the other counts 300 and gives 5.3% and 4.0%. The second of those is the reading quoted in the body. We print the disagreement rather than adjudicate it, and no other table depends on which denominator is used. The `gpt-3.5` rows are not affected by this dispute — the $398/407$ there is a genuine denominator mismatch, seven runs in which the guard did not fire, not a counting disagreement.

Victim	Level	H	n	H/n	H/fired
`deepseek-v4`	version	55	450	12.2%	12.2%
`deepseek-v4`	recheck	60	450	13.3%	13.3%
`gpt-3.5`	version	444	450	98.7%	98.7%
`gpt-3.5`	recheck	398	407	88.4%	97.8%
`qwen2.5:14b`	version	16	450	3.6%	—
`qwen2.5:14b`	recheck	12	450	2.7%	—



Benign performance is 24/24 in every measured cell, so the only cost these defenses differ on is how often the guard fires. Markers that coincide are offset horizontally by at most 0.05 to stay visible.

Figure 3: **Security–utility frontier.** Each marker is a measured (victim, defense) pair: horizontal axis, how often the guard intervenes per run; vertical axis, pooled attack success. Benign performance is $24/24$ in *every* cell, so intervention rate is the only cost on which these defenses differ. The origin — no interventions, no successful attacks — is reached by exactly one mechanism, pass-by-value binding, which comes from a separate n=90 arm rather than the n=450 grid. No rule guard reaches it: the one that suppresses most intervenes on every run, and the guards that intervene least are the ones that suppress least, so the frontier is steep rather than flat — buying back the difference between one intervention per run and 0.33 costs 86.5 pp of suppression (1.33% to 87.78% success). Markers that coincide are offset horizontally so that both remain visible.

the exposed horizon's median moves only from 6464 to 6858 ms (+6%). The adaptive attacker arms on average 4.27 times per run (max 11, 100% of runs) and its shots land *after* the check that already succeeded, i.e., squarely in the version-invalidation zone, so every subsequent use is judged stale. Raising the block count nearly threefold and losing 17.7 pp are consistent with a single mechanism: *the longer a race is kept open, the more likely the guard is to catch it*, so re-arming converts a single well-placed write into a series of poorly-placed ones. Four limits must travel with Table 26. The

Table 17: **Controlled format scan.** Same 24 documents, same classifier, same threshold (0.5); only the string form given to the classifier changes. Score is the DeBERTa prompt-injection model's INJECT probability (no sampling: the same string yields the same score). Envelope conditions (C, D, P) cross the threshold far more often than the bare text they wrap, and condition C contains *no injection at all*.

	Condition	Median score	> 0.5
A	Bare body, *no injection*	0.0554	6/24
B	Bare body + injection	0.1454	8/24
C	**Production JSON envelope, no injection**	**0.7669**	**14/24**
D	Production JSON envelope + injection	0.9357	17/24
P	Envelope, different path	0.5596	12/24
E	Bare body + neutral padding (same char count)	0.0423	5/24

Table 18: **Real stack.** LangGraph agent, real SQLite workspace, attacker as a separate OS process in blind timing mode (no reconnaissance), victim `gpt-3.5-turbo-1106`, n=30 per condition. "In-window" is the count of runs where the write landed inside the check–use gap; ASR_c is success *conditional on landing in-window*; *ASR* is success over all n runs, so $\text{ASR} = (\text{in-window}/n) \times \text{ASR}_c$. The baseline case is the point: a blind attacker lands only $7/30$ times, but whenever it lands it wins, and two of the four primitives do not change that.

Condition	ASR	95% CI	In-window	ASR_c	Hold p50	n
`real_none`	23.3%	[11.8, 40.9]	7/30	100.0%	—	30
`real_lock_proc`	33.3%	[19.2, 51.2]	10/30	100.0%	1410.5 ms	30
`real_lock_call`	20.0%	[9.5, 37.3]	6/30	100.0%	**1.15 ms**	30
`real_lock_session`	**0.0%**	[0.0, 11.4]	7/30	**0.0%**	1442.3 ms	30
`real_pbv`	**0.0%**	[0.0, 11.4]	9/30	**0.0%**	—	30

intermediate d=0 cell that reported $\Delta\text{ASR} = 0.0$ pp is **void**: it ran n=20 on a harder 10-scenario subset where the open loop itself only reached 90.0%, and a n=20 sample cannot separate 67% from 97%. Small n plus a changed scenario pool manufactured a false zero, and every T4 cell must therefore report its novel/trial counts — we do, and we note that the same delay's open-loop rate moved from 66.7% to 99.33% across batches, so cross-batch comparison is invalid. Re-arm interval X=0 was never run at d=0, so we cannot generalize to "any re-arm interval"; the two d=2000, X=0 rows are that single cell measured in two separate batches (an n=30 grid run and an n=6 pre-flight), printed side by side rather than pooled precisely because they do not agree. A per-run expression in the harness, which we had intended to use as a mechanism column, is unreliable across batches. Fourth, the crash asymmetry above is quoted from the run report: the two arms' per-run file sets were compared item by item only at d=2000 (sole difference: the one crashed open-loop file, whose closed-loop twin ran fine at the same scenario and trial, so the slip is random rather than scenario-driven), not at the final cell. The batch warning forbids reading a rate *across* batches; the final cell's z test compares its two arms *within* one batch — both ran in a single session — which is why that test stands while the moved open-loop rate (66.7% → 99.33% for the same delay) does not. Every ΔASR above rests on the workspace-state detector alone, which does not read it, and all mechanism counts in this paragraph were recomputed from the per-run event logs.

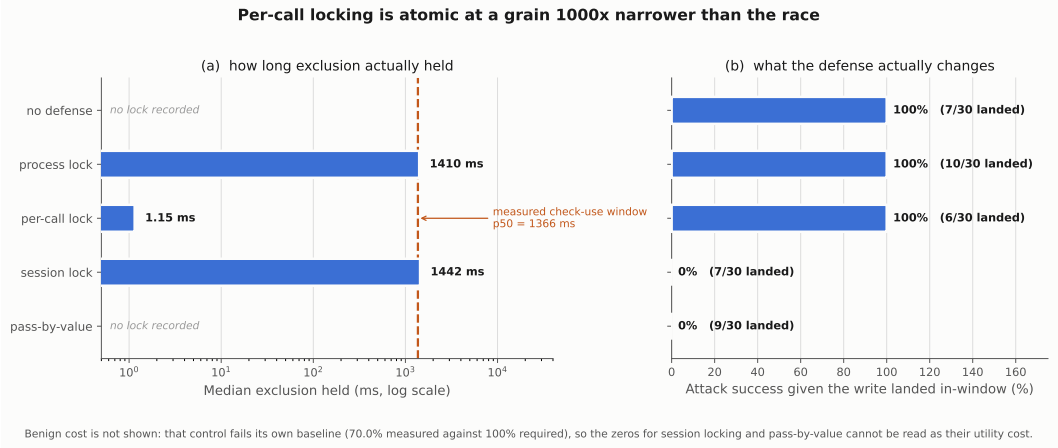


Figure 4: **Where the real-system primitives break.** *(a)* Median exclusion actually held, on a log axis, against the measured check–use window (1366 ms). *(b)* Attack success *conditional* on the attacker's write having landed in the window — the quantity the defense actually changes, as opposed to the headline rate, which mostly reflects a blind attacker's timing luck. Per-call locking holds exclusion for milliseconds and still converts every write that lands.

Table 19: **mtime arm, both victims.** The attacker triggers on checkpoint file modification times rather than wall clock — reconnaissance of a different observable. Real SQLite workspace, n=75 per condition (15 cells × 5 trials). "In-window" is over 15. The fourth condition is the one that matters: on a real single-process agent, session-scoped locking takes the conditional rate from 100.0% to 0.0% on *both* victims, so the headline of Table 18 is not a property of one model. The one conditional cell below 100.0% (panel (b), `lock_proc`) has a single censored run. The hold column is this arm report's own summary statistic and is not the same statistic as the median given for the same cells in Table 18.

Condition	ASR	In-window	ASR_c	Hold (ms)	Thr. (runs/min)
(a) `gpt-3.5-turbo-1106`, SQLite					
`real_none`	66.7%	10/15	100.0%	—	11.75
`real_lock_proc`	60.0%	9/15	100.0%	985.1	11.67
`real_lock_call`	66.7%	10/15	100.0%	**1.4**	11.20
`real_lock_session`	**0.0%**	14/15	**0.0%**	1962.1	11.03
`real_pbv`	**0.0%**	11/15	**0.0%**	—	11.09
(b) `deepseek-v4-flash`, SQLite					
`real_none`	100.0%	15/15	100.0%	—	11.57
`real_lock_proc`	86.7%	14/15	92.9%	860.5	12.08
`real_lock_call`	100.0%	15/15	100.0%	**1.3**	11.33
`real_lock_session`	**0.0%**	15/15	**0.0%**	894.3	18.57
`real_pbv`	**0.0%**	14/15	**0.0%**	—	7.71

APPENDIX E DEFENSE FORMALISM AND IMPLEMENTATION DETAILS

Defense formalism.

A defense is a pair $(\mathsf{record}, \mathsf{guard})$ interposed on the check and use tools of §3.1. When a check tool reads object O, the defense stores a snapshot $s(O)$ of the object's state at that moment (s_{version}: the version number; s_{value}: the payload; s_{lock}: exclusive ownership). When the corresponding use tool is about to act on O, the guard reads the current state $s'(O)$ and decides

$$\mathsf{allow}(O) \;=\; \begin{cases} \top & \text{if } s(O) \equiv s'(O) \text{ under the rule, or no snapshot exists} \\ \bot & \text{otherwise (the action is aborted).} \end{cases} \tag{4}$$

We instantiate the rule in five ways, plus one learned variant:

- **version**: $\equiv$ is version-number equality.
- **recheck**: $\equiv$ is payload equality (re-read and compare the resolved value).

Table 20: **One scenario, two instruments.** Victim `gpt-3.5-turbo-1106`, SQLite workspace, out-of-process legitimate updater triggered on checkpoint mtime. *Panel (a):* the benign control, `real_none` condition, class B (read, then one legitimate update), 10 trials per family — 21 of 30 land. *Panel (b):* the attack arm on the `event_injection` family, 5 runs per condition; "same turn" counts runs in which the framework issued the check and the use in a single step, i.e. the window collapsed. Where the window collapses the attack cannot land either, and the two conditions that partially reopen it ($4/5$ and $1/5$ same-turn) are the two that then record a landing. No defense is responsible for any zero in this table.

(a) Benign control, `real_none`, class B

Family	Scenario	Landed
`benign_b_email`	`shared_state`	10/10
`benign_b_file`	`delayed_context`	10/10
`benign_b_inbox`	`event_injection`	**1/10**
total		21/30 (70.0%)

(b) Attack arm, `event_injection`, n=5 per condition

Condition	Same turn	Landed in window
`real_none`	5/5	**0**
`real_lock_proc`	5/5	**0**
`real_lock_call`	5/5	**0**
`real_lock_session`	1/5	**4**
`real_pbv`	4/5	**1**

Table 21: **Blind arm, second victim.** Same apparatus as Table 18, victim `deepseek-v4-flash`, n=30 per condition. The conditional column is flat at 100.0% for the three conditions that do not change it and 0.0% for the two that do, while the headline rate moves from 16.7% to 36.7% across the first three: the only thing changing there is how many of the attacker's blind writes happened to land, and on this victim the headline rate is an exact restatement of the landing count (5, 6, 11 of 30). That is the whole reason the conditional column is the headline. Censored runs are shown because they stay in each rate's denominator.

Condition	ASR	In-window	ASR_c	Hold (ms)	Cens.	Thr.
`real_none`	16.7%	5/30	100.0%	—	5	14.98
`real_lock_proc`	20.0%	6/30	100.0%	1113.0	6	15.78
`real_lock_call`	36.7%	11/30	100.0%	**1.3**	2	14.97
`real_lock_session`	**0.0%**	4/30	**0.0%**	—	—	—
`real_pbv`	**0.0%**	4/30	**0.0%**	—	—	—

- **recheck + content judgment**: the re-check is paired with a semantic classifier that decides whether the payload is an attack.
- **CAS**: the use is expressed as a compare-and-swap against the version number that was observed at check time.
- **lock** (pessimistic): the check acquires exclusive ownership of O and holds it until the use completes; no snapshot comparison is needed.
- **pbv** (pass-by-value binding): the check rewrites the agent's reference so that the use consumes the *resolved value* recorded at check time; the attacker's later write is unreachable. We implement the binding as a read-only proxy over the workspace: the use tool's key still resolves, but to the value pinned at check time. The tools and prompts presented to the model are unchanged, so the change A2 describes is absorbed by the harness rather than by the agent's interface. The reduction we report is therefore attributable to the binding, not to interface novelty.

A blocked use returns control to the agent with a block signal. The harness performs no retry of its own, so whether a blocked action is re-attempted is the model's decision and not part of the defense; the only budget on it is the step cap ($\mathrm{max_steps}$=24), and a run that exhausts that cap is counted *censored*, never *blocked*. The unretried case — the blocked action simply abandoned — is the one Eq. 3 scores as utility loss. Re-checking after a block is the measured behaviour, not the exception: over two victims and the three rule levels (10 runs per cell), every run was blocked at least

Table 22: **Advisory-lock backend, both victims.** Same agent and same attacker, with the shared workspace moved from SQLite to a real directory tree; the backend is the only per-run change. On this backend the session lock's conditional rate goes from 0.0% (Table 19) to 100.0% on *both* victims. The row that carries the result is the `gpt-3.5` session lock: it holds exclusion for 2566.2 ms against an exposed window of 2565.8 ms — 100.02% coverage — and the attacker still took all ten writes it landed. The lock was held for longer than the entire gap it was meant to protect, and the writer never waited: the blocking column is 0.5–0.7 ms, at the floor, on every condition and both victims, against 931.1 ms for the same cell on SQLite. The failure is therefore not that the lock is too narrow — it is wider than the gap — but that an advisory lock does not exist for a writer that does not cooperate with it. One apparent inversion, `gpt-3.5 lock_call` at 86.7% above `none` at 66.7%, disappears under the conditional column (both 100.0%); it is aiming variance, not a weaker defense.

Condition	ASR	In-window	ASR_c	Attacker blocked (ms)	Thr.
(a) `deepseek-v4-flash`					
`real_none`	100.0%	15/15	100.0%	**0.6**	11.57
`real_lock_proc`	93.3%	14/15	100.0%	**0.6**	12.08
`real_lock_call`	93.3%	14/15	100.0%	**0.6**	11.33
`real_lock_session`	100.0%	15/15	100.0%	**0.7**	10.95
`real_pbv`	**0.0%**	15/15	**0.0%**	**0.6**	16.94
(b) `gpt-3.5-turbo-1106`					
`real_none`	66.7%	10/15	100.0%	**0.6**	7.35
`real_lock_proc`	60.0%	9/15	100.0%	**0.6**	7.61
`real_lock_call`	86.7%	13/15	100.0%	**0.5**	7.56
`real_lock_session`	66.7%	10/15	100.0%	**0.6**	7.50
`real_pbv`	**0.0%**	11/15	**0.0%**	**0.6**	7.28

Table 23: **The two primitives, by backend.** "Session lock" is `real_lock_session` and "pass-by-value" is `real_pbv`; the blocking ranges are the legitimate-writer block times of Table 24. Every cell is measured, not argued. On a backend that enforces exclusion, the two primitives both drive the conditional rate to 0.0%, and both make a legitimate in-window update infeasible; on the correctness axis they are equivalent. On a backend with advisory locking, the lock's benefit and its cost disappear together while pass-by-value does not move. This is the appendix's answer to why the paper recommends the primitive it does. On the enforcing backend there is no cost difference left to point at; on the advisory one the lock's protection and its price go missing together, and the price that remains is pass-by-value's.

Backend	Primitive	Attack allowed	Legitimate update	Same act?
SQLite (enforced)	session lock	no ($100\% \to 0.0\%$)	blocked (1.1–2.2 s)	yes
SQLite (enforced)	pass-by-value	no ($100\% \to 0.0\%$)	blocked	no
Tree (advisory)	session lock	**yes, stays** 100%	**allowed** (0.6 **ms**)	yes
Tree (advisory)	pass-by-value	no ($100\% \to 0.0\%$)	blocked	no

once and every run then re-read the object and used the post-mutation value ($60/60$; 2.0–8.0 check invocations per run; no run reached the cap). That probe mutates the object through a *legitimate* concurrent update, so the re-check belongs to the agent loop and is not an artifact of the attack. On the attack side the same loop is what defeats CAS: at n=90, `cas` aborts a use in 30 runs, and the attack nevertheless succeeds in all 90 (100.0%, Table 6) — no run in which an abort occurred ended without the exploit.

Implementation and reproducibility.

Experiments use a Python tool-agent simulator (shared workspace, observer, tools, orchestrator, detectors, rule and learned defenses) and, for §4.6, a real LangGraph agent over a real SQLite workspace with a separate OS process as attacker. Novel-entity scenarios are generated with a fixed seed; each family uses 10 novel entities $\times$ 3 trials (n=30/family) unless stated otherwise. The learned attacker is a LoRA fine-tune of a Qwen2.5 base on state-to-action traces (100–3000 traces in the budget sweep, rank 16, 3 epochs, lr 2×10^{-4}); victims are served through their native APIs or locally. Proportions reported with intervals use Wilson 95% intervals; NRP uses bootstrap intervals.

Table 24: **Cost, by task class.** Condition names are short for the `real_` conditions of Table 19: `none`, `pbv`, `lock`; `ds` is `deepseek-v4-flash`. Class A tasks contain no concurrent legitimate update and measure false blocking; class B tasks read a value and then perform one legitimate update, and measure whether a cross-step task stays feasible. Attack traffic and legitimate updates are identical in timing — only the written value differs. Every defense that drives the conditional attack rate to 0.0% also drives class B to near zero, though by different routes: the session lock blocks the writer, whereas pass-by-value lets the update land and leaves the agent holding the pre-update value. The last block is the backend split: on the advisory backend the session lock's class-B rate returns to 100.0% while pass-by-value's stays at 0.0%. The `gpt-3.5`/directory-tree benign arm was not run and is left blank rather than extrapolated from the other five arms. *Provenance of the `gpt-3.5`/SQLite block:* class A comes from an `A,B` invocation of the benign suite and class B from a later class-B-only invocation; the no-defense class-B cell reads 70.0% in the latter and 100.0% in the former, and the running text uses the 70.0%. The pass-by-value denominator is 29 rather than 30 because in one run the legitimate update landed *before* the check, which the fake-pass gate excludes.

Victim	Backend	Condition	Class	Passed	Rate	Upd. blocked (ms)	e2e (ms)
`gpt-3.5`	SQLite	`none`	A	30/30	100.0%	0.0	8972
`gpt-3.5`	SQLite	`none`	B	21/30	**70.0%**	1.3	5555
`gpt-3.5`	SQLite	`pbv`	A	30/30	100.0%	0.0	5825
`gpt-3.5`	SQLite	`pbv`	B	0/29	**0.0%**	1.3	5469
`gpt-3.5`	SQLite	`lock`	A	30/30	100.0%	0.0	5406
`gpt-3.5`	SQLite	`lock`	B	0/30	**0.0%**	2736.1	6453
`ds`	SQLite	`none`	A / B	—	100.0 / 100.0%	0.0 / 1.1	—
`ds`	SQLite	`pbv`	A / B	—	100.0 / **0.0%**	0.0 / 4.8	—
`ds`	SQLite	`lock`	A / B	—	100.0 / **0.0%**	0.0 / 1084.6	—
`ds`	tree	`none`	A	30/30	100.0%	0.0	2520
`ds`	tree	`none`	B	29/29	100.0%	0.6	2548
`ds`	tree	`pbv`	A	30/30	100.0%	0.0	2659
`ds`	tree	`pbv`	B	0/30	**0.0%**	0.6	2867
`ds`	tree	`lock`	A	30/30	100.0%	0.0	5054
`ds`	tree	`lock`	B	30/30	**100.0%**	0.6	2596
`gpt-3.5`	tree	all	A / B	*arm not run*			

APPENDIX F DISCUSSION AND LIMITATIONS

Where the attack does not apply. An agent whose tools carry resolved values rather than keys, or that holds all intermediate context in its own active window, has no interval for Eq. 2 to inhabit — the `pbv` row of Table 6 is the constructive proof, at 0.0% over 90 runs. Stateful tools and persistent workspaces are nonetheless load-bearing for real agentic software, so A1–A2 describe current practice rather than a narrow special case.

What we did not measure. Our results are simulator measurements plus one real LangGraph+SQLite stack — a step toward production relevance, but still our own agent; we have not evaluated a commercially deployed closed-source agent product, and the benign cost of the two primitives that work is priced only on the two backends we built: one cell of that grid, `gpt-3.5` on the directory tree, was never run and is left blank rather than extrapolated (Appendix C). We also keep the concurrency claim at the strength the data supports: across 1, 2, and 4 concurrent writers on a shared key, median lock hold time grew by $2.9\times$ in the worst round, yet makespan under the pessimistic lock stayed within $1.11\times$ of undefended, because exclusion over one key is negligible against model latency. That is a two-round observation on one workload, not a claim (Appendix A).

Why a semantic judge cannot close this. The gap between rules and a learned guard is not a capability gap. For the identity-family attack the legitimate update and the covert write are *isomorphic* strings: both replace the same field of the same record with a value of the same shape and domain, differing only in the entity they name. Any predicate admitting the legitimate update and rejecting the covert one is therefore not detecting a race but enforcing an entity whitelist — a useful component, but a different mechanism from binding object identity across the gap, and the reason the learned judge in §4.4 tops out below a simple rule.

Table 25: **Further unresolved readings in the real-system artifacts.** In addition to the two defects recorded above. Every item is a known gap in the apparatus rather than a result, and none of them carries a claim in this paper. They are listed because an apparatus with no recorded defects is not more credible than one that names its own.

#	Where	What remains unresolved
1	`real_pbv`, DeepSeek	One run's model call timed out and retried — 61.7 s against a 60 s client timeout, median 0.95 s — inflating that cell's span mean and depressing its throughput. Not confirmed run by run from the logs; every pass-by-value span and throughput figure here is quoted as a median.
2	`real_pbv`, DeepSeek	Legitimate-writer blocking is 4.8 ms against a 1.16 ms floor: still non-blocking, 226× below the session lock, but not smoothed to the floor.
3	`real_lock_session`, SQLite	Two of thirty class-B runs passed. Blocking time is bimodal *within* the condition — 28 runs at second scale, 2 at the floor — so the passes are the runs the lock did not reach, not a criterion failure. *Why* it did not reach them is unverified; the standing hypothesis is that the lock is acquired from a post-verification hook, i.e. after the check has already returned, making its true scope narrower than the gap it is meant to cover.
4	`real_pbv`, class B	Two of thirty runs passed on a family where blocking time sits at the floor and is identical across passing and failing runs, so that column has no explanatory power for this primitive at all. Mechanism unidentified.
5	Pre-flight	The probe accepts an empty reply, and the provider returns 200 for model names absent from its own list. Together these let a wrong model name pass pre-flight and produce a batch of empty runs. Not triggered in any batch reported here (tool calls independently verified); the hardening was deferred rather than applied mid-batch.
6	`real_lock_session`, directory tree	Class-A end-to-end time doubled (2520 → 5054 ms) with no concurrent writer at all. Two candidate causes, not distinguished: fixed polling overhead in the acquire path, or an occasional retry of the same family as item 1. The comparison that would separate them — against the SQLite arm — is not valid, being a different victim and an order of magnitude higher.
7	Pre-registration	The reconciliation dictionary carried one axis where the artifact has two. On the advisory backend the session lock's class-B rate came out at 100.0% against a predicted 0.0%, recorded as an explicit mismatch. The prediction was right in substance — the plan states *before* the run that an advisory lock fails for two reasons on this backend and one on SQLite — but the dictionary had no backend axis in which to express it. The mismatch is left in the output and the dictionary is not amended afterwards; the correction is a two-dimensional dictionary at the *next* batch.

Implications. Every mechanism that failed did so for the same reason — an atomicity boundary narrower than the check–use span — and every one that worked did so because the reference the use resolves is bound at check time and cannot be re-pointed. Atomicity for agents should therefore be defined over the semantic span of a verification rather than the storage operation implementing it: agent tool interfaces should bind values rather than keys.

Table 26: **Adaptive (closed-loop) vs. open-loop attacker** against the `version` guard, victim `qwen2.5:14b`, re-arm interval swept over $\{0, 150, 600\}$ ms. Every row is copied field-for-field from the run report, with no pooling or estimation. Where the open loop is alive, re-arming is strictly worse; the only rows where it gains are those the open loop loses by construction, and even there it stays far below the open loop's own optimum.

delay (ms)	re-arm (ms)	n/arm	Open-loop ASR	Closed-loop ASR	ΔASR	Note
0	150	150/147	99.33%	81.63%	$-\mathbf{17.7}$ pp	final cell, $p < 10^{-6}$
2000	0	30/30	66.7%	33.3%	-33.3 pp	worst re-arm interval
2000	150	19/20	89.5%	80.0%	-9.5 pp	open loop crashed one run
20000	0	30/30	0.0%	3.3%	$+3.3$ pp	open loop dead
20000	150	20/20	0.0%	35.0%	$+35.0$ pp	
20000	600	20/20	0.0%	35.0%	$+35.0$ pp	identical to 150 ms
2000	0	6/6	66.7%	50.0%	-16.7 pp	same cell; pre-flight

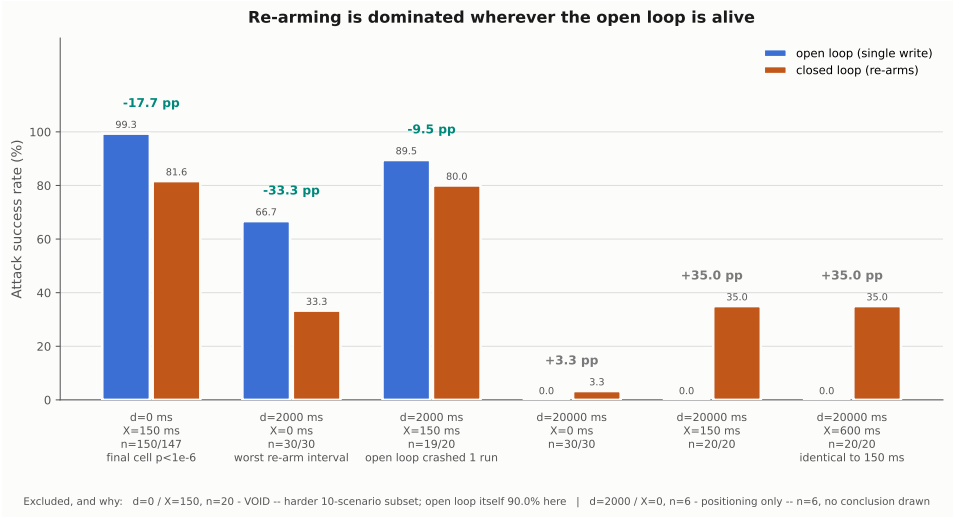


Figure 5: **Open-loop vs. re-arming attacker.** Attack success per configuration against the `version` guard; each pair of bars is one (delay, re-arm interval) cell, with its arm sizes printed on the axis. Teal ΔASR values are cases where re-arming helps the defense. Re-arming loses wherever the open loop is alive and only gains where the open loop is dead by construction. The two cells excluded from the bars (an n=20 false zero and an n=6 positioning run) are named under the axis rather than dropped silently.