

Safe to Resume? Breaking Execution Continuity of Agent Execution via Rollback

Guanlong Wu*, Dahui Li*, Ke Jiang*, Jianyu Niu[†], Cong Wang[†], Yinqian Zhang*,

*Southern University of Science and Technology, [†]City University of Hong Kong

Abstract—AI agents are moving toward persistent, stateful execution across various applications, accumulating execution state and external effects that are costly to reconstruct after failures. Checkpoint and rollback (C/R) are becoming essential for recovery, yet their security implications remain largely unexplored. Correct rollback does not imply secure recovery: a faithfully restored checkpoint may resume an execution whose states, assumptions, and external effects never coexisted in any valid history. In this paper, we present the first systematic security study of checkpoint and rollback in existing agent systems. By examining representative agent C/R systems, we characterize the design space of existing C/R mechanisms and develop a general execution model that captures their recovery boundaries and state dependencies. From this model, we identify five fundamental failure modes spanning incomplete or inconsistent internal state, stale external dependencies, nondeterministic replay, and unrecorded external effects. We further demonstrate their security impact through three end-to-end attacks on Hermes, Cline, and LangGraph, enabling malware-verification bypass, unauthorized mail forwarding, and double payment. To systematically study these failures in practice, we develop a multi-agent analysis pipeline that reconstructs execution semantics, identifies violations of the five failure conditions, and validates them through actual rollback. Across five representative frameworks, our evaluation shows that these failures recur across heterogeneous C/R designs and stem from a common gap between the state restored by a checkpoint and the dependencies required for secure continuation.

I. INTRODUCTION

AI agents are moving from short-lived interactions toward persistent, stateful execution across autonomous research [1], coding [2], and enterprise services [3]. This shift is already visible in production: Tata Steel reports deploying over 300 specialized agents, while Cognizant operates more than 200 agents serving 350,000 employees [4], [5]. Yet reliability remains a major concern: a 2026 enterprise survey found that 79% of respondents had reversed an agent action and 42% reported revenue loss from agent failures [6]. As long-running executions accumulate state and external effects, restarting from scratch becomes increasingly costly. Checkpoint and rollback (C/R) are therefore becoming a basic recovery mechanism, with modern systems checkpointing framework state [7], workspaces [8], and execution environments [9].

C/R periodically records selected execution state and restores a prior checkpoint after interruption, which has long been studied for reliable recovery and state continuity maintenance [10], [11], [12], [13]. Recent systems have begun optimizing C/R specifically for agent execution: CRAB improves

recovery through semantics-aware checkpointing [14], while DeltaBox provides low-latency sandbox checkpoint and rollback [15]. Concurrently, recent studies have begun examining the semantics and security of agent recovery, including valid resume behavior, recovery consistency, and semantic rollback attacks [16], [17]. These advances establish both the practical importance and security relevance of agent recovery, but leave a broader question open: *when rollback carries forward a security-relevant fact, what dependencies must remain valid for that fact to be safely reused?*

This question is fundamental because recovery carries forward security-relevant facts established earlier in execution: an artifact has been validated, an operation authorized, a resource binding remains valid, or an external effect has not yet occurred. Yet the validity of these facts may depend on state distributed across framework components, files and runtimes, nondeterministic decisions, and external services, which do not necessarily share the same recovery scope. Thus, a checkpoint’s recovery boundary may not cover the full set of security dependencies on which the restored state relies. *Correct rollback does not imply secure recovery*: every recorded state may be restored faithfully while the relationships that made those states security-valid are broken.

Consider a coding agent tasked with removing malware from a repository before release. The agent removes the malicious payload, scans the cleaned repository, and records the passing result. If recovery restores the pre-cleanup workspace while retaining the post-scan agent state, the agent may release the malicious repository as verified. Neither the checkpoint nor the scan result is forged; rollback has rebound a genuine security judgment to an artifact it never validated. We call the underlying requirement *execution continuity*: security-relevant states, decisions, assumptions, and effects carried across recovery must remain jointly consistent with the valid execution history that justifies them.

In this paper, we conduct a systematic cross-system security study of checkpoint and rollback in agent systems. We first examine representative agent C/R systems and characterize their designs by three recurring recovery boundaries: framework state, workspace state, and OS/VM state. This study motivates a general execution model that captures independently evolving internal states, nondeterministic execution, checkpoints with partial recovery boundaries, and an outside world that does not roll back with the agent.

Using this model, we identify five recurring ways in which

rollback can break execution continuity: *incomplete internal state coverage*, *inconsistent checkpoint state*, *external state mismatch*, *unbound nondeterministic replay*, and *unrecorded external effects*. These failures capture how a legitimately restored checkpoint can invalidate the state, context, or effects that justify security-relevant decisions carried across recovery. Importantly, they do not require checkpoint corruption or a compromised recovery mechanism.

We demonstrate the security impact of these failures through three end-to-end attacks on Hermes, Cline, and LangGraph, causing malware-verification bypass, unauthorized mail forwarding, and duplicate payment, respectively. We further develop a trace-based framework that reconstructs execution dependencies, detects candidate recovery failures, and validates them through native rollback and re-execution. Across 347 benchmark traces from TerminalBench and AgentBench, yielding 1,735 framework-task executions over five representative C/R systems, our evaluation shows that recovery failures are common but exhibit distinct profiles across different recovery boundaries, and reveals recurring factors involving external dependencies, nondeterministic replay, and persistent effects.

Contributions. Main contributions are listed as follows:

- **Agent recovery abstraction.** We characterize representative agent C/R systems and develop an execution model that captures heterogeneous recovery boundaries, nondeterministic execution, and an independently evolving outside world.
- **Security characterization and attacks.** We introduce execution continuity and identify five recurring ways in which legitimate rollback can invalidate security-relevant state or transitions. We demonstrate their practical impact through three end-to-end attacks on real agent frameworks.
- **Systematic detection and validation.** We develop and evaluate a trace-based methodology for detecting recovery failures and validating them through actual rollback, providing an empirical characterization of how these failures manifest across representative agent C/R designs.

II. CHARACTERIZING EXISTING AGENT CHECKPOINTING

We examine 12 representative agent C/R systems and classify their checkpointing mechanisms by *recovery boundary*, i.e., the primary state that is restored upon rollback. As summarized in Table I for five representative systems, existing designs fall into three recurring categories: *framework-state*, *workspace-state*, and *OS/VM-state* checkpoints. For each category, we characterize *what state is captured*, *when checkpoints are created*, and *how state is restored*.

A. Framework-State Checkpoints

Framework-state checkpoints preserve framework-managed execution state, such as conversation history, workflow progress, task outputs, agent memory, and pending steps, while leaving workspace and runtime state outside the recovery boundary. Representative systems include LangGraph [7], CrewAI [18], LlamaIndex Workflows [19], Google ADK [20], and

Microsoft Agent Framework [21]. In this paper, we focus on LangGraph and CrewAI.

LangGraph. LangGraph represents an agent workflow as a graph whose nodes perform operations such as LLM calls or tool invocations. Execution proceeds in *supersteps*: each superstep executes the current scheduled nodes before advancing to the next graph state, thereby providing a checkpoint boundary.

- **Checkpointing.** When persistence is enabled, LangGraph associates each execution history with a `thread_id` and uses a configured checkpointer to save a `StateSnapshot` at every superstep boundary. A snapshot records the accumulated graph state, such as conversation messages and intermediate results, together with metadata describing the nodes and pending tasks scheduled for subsequent execution. Snapshots can be persisted in backends such as SQLite or PostgreSQL for durable recovery, and LangGraph does not impose a default retention limit on the checkpoint history [7].
- **Restoration.** An application may resume from the latest checkpoint using the same `thread_id`, or select an earlier checkpoint from the state history for replay or branching. Execution resumes from the corresponding superstep boundary: state represented before that checkpoint is retained, whereas subsequent nodes, including their LLM calls and tool invocations, are re-executed [7], [22].

CrewAI. CrewAI organizes applications around two execution abstractions: a *Crew* coordinates role-specific agents and tasks, whereas a *Flow* maintains shared state and orchestrates event-driven execution involving code, LLM calls, and Crews [18].

- **Checkpointing.** CrewAI checkpoints framework-managed execution state. For Crews, this includes configuration and inputs, task progress and outputs, and participating agents' memory and runtime attributes; for Flows, it includes shared state, completed methods, intermediate outputs, and execution position. Checkpoint creation is event-driven: task completion (`task_completed`) is the default trigger, while finer-grained events such as completed LLM calls or Flow methods can also be configured; developers may additionally create checkpoints explicitly. Checkpoints can be persisted using JSON or SQLite backends, with `max_checkpoints` optionally bounding the retained checkpoint history [23].
- **Restoration.** A stored checkpoint can be selected to restore the corresponding framework state and resume execution from that point, without re-executing tasks recorded as completed in the checkpoint [23].

B. Workspace-State Checkpoints

Workspace-state checkpoints extend recovery to artifacts manipulated by the agent, most commonly project files. They are particularly common in coding agents and are typically implemented using file snapshots or a *shadow Git repository*—a separate repository used to version the agent's workspace without modifying the project's own Git history. Their recovery boundary, however, generally excludes runtime state such as process memory, installed system state, active services, and files outside the selected workspace. Representative systems

TABLE I: Checkpoint and rollback mechanisms in representative agent systems.

System	State Scope	Captured State	Checkpoint Creation	Restoration
LangGraph	Framework	Graph state, including messages, intermediate results, pending execution state	Automatic at each superstep boundary once persistence is enabled	Resume the latest checkpoint or select an earlier checkpoint for replay/branching
CrewAI	Framework	Crew/Flow/Agent execution state, including task progress, outputs, memory, and intermediate state	Event-driven once enabled; task completion is the default trigger	Restore a selected checkpoint and resume without re-executing completed tasks
Hermes	Workspace	Project/workspace files; rollback also adjusts the most recent conversation turn	Automatic before file mutations or recognized destructive commands once checkpointing is enabled	Restore a selected workspace checkpoint (or individual file) via <code>/rollback</code>
Cline	Workspace	Project files associated with the corresponding task-history position	Automatic after file modifications and terminal commands; enabled by default	Restore files, task history, or both from a selected checkpoint
E2B	OS/VM	Sandbox filesystem and memory state, including running processes	Explicit, at developer-selected points via <code>create_snapshot()</code>	Create a new sandbox from a selected snapshot

include Hermes [24], Cline [25], Gemini CLI [26], and Claude Code [27]; we focus on Hermes and Cline.

Hermes. Hermes is a general-purpose tool-using agent that follows an iterative tool-calling loop, allowing it to manipulate files and execute terminal commands while maintaining the surrounding conversation state [28].

- *Checkpointing.* When checkpointing is enabled, Hermes creates a workspace checkpoint immediately before file-writing operations such as `write_file` and `patch`, as well as recognized destructive terminal commands. It resolves the relevant project root, creates at most one checkpoint per directory within each conversation turn, and stores workspace snapshots as project-specific references in a shared shadow Git store. By default, Hermes retains at most 20 checkpoints per project and 500MB across the shared store, while excluding individual files larger than 10 MB [24].

- *Restoration.* Users invoke `/rollback` to select a previous checkpoint. Hermes restores the files represented by the selected Git tree and rolls back the most recent conversation turn. Its recovery restores workspace state together with only a limited portion of the surrounding conversation state [24].

Cline. Cline is an IDE-integrated coding agent in which each task maintains a user goal, conversation, and task history, and a sequence of tool interactions over a project workspace. During execution, Cline can inspect and modify files, execute terminal commands, and invoke browser or Model Context Protocol (MCP) tools, subject to explicit user approval or configured auto-approval policies [25], [29].

- *Checkpointing.* Cline enables checkpoints by default and maintains a shadow Git repository independent of the project’s Git history. It snapshots the complete file state of the current workspace, including files not tracked by the user’s repository, and associates each snapshot with the corresponding position in the task history. Checkpoints are created automatically after relevant tool use, including file modifications and terminal commands, and persist across editor sessions; Cline has no default retention bound [30].

- *Restoration.* Cline allows the user to restore the workspace while retaining the current task history, restore the task history while retaining the current workspace, or restore both to the selected checkpoint. Thus, workspace state and

framework-managed state can be handled independently [30].

C. OS/VM-State Checkpoints

OS/VM-state checkpoints extend the recovery boundary to the execution environment, capturing filesystems, process memory, and running services at higher checkpointing and storage costs. Representative systems include E2B [31], CRAB [14], and DeltaBox [15]. While CRAB and DeltaBox reduce these costs through semantics-aware or incremental checkpointing, we use E2B as a representative system.

E2B. E2B provides isolated cloud *sandboxes*—self-contained execution environments in which agents can manipulate files, execute programs, and run services. Each sandbox maintains its own filesystem, processes, and memory state and is accessed through the E2B SDK.

- *Checkpointing.* An application explicitly invokes `create_snapshot()` to create a point-in-time snapshot of a running sandbox. E2B briefly pauses the sandbox while capturing its filesystem and memory state, including running processes and loaded data, after which the original sandbox continues execution. Snapshots persist until deleted, with no default retention period or history bound [31].

- *Restoration.* A snapshot identifier can be supplied to `Sandbox.create()` to instantiate a new sandbox from the captured filesystem and memory state. Snapshots are reusable, allowing the same captured state to initialize multiple sandboxes [32], [31].

Takeaway. Agent C/R mechanisms span framework, workspace, and runtime state, but broader coverage does not guarantee safe recovery. External state and persistent effects remain outside the recovery boundary, while some designs restore state components independently. Thus, recovery may break the dependencies required for security-valid resumption, motivating the general model in Sec. III-A.

III. PROBLEM DEFINITION

We develop a general model for agent execution and rollback based on the recovery boundaries characterized in Sec. II. Our model draws on the classic distributed-state abstraction of Chandy and Lamport [10], while capturing three characteristics central to agent recovery: heterogeneous internal state,

nondeterministic execution, and interaction with an external world that does not roll back with the agent.

A. System Model

Components. We model an agent system using the following components.

- **Process.** The agent system consists of a set of logical processes $\mathcal{P} = \{P_1, P_2, \dots, P_n\}$. Each P_i represents an independently evolving stateful component rather than necessarily an operating-system process [10]. For example, the main agent, a subagent, framework-managed conversation state, a tool runtime, or the workspace filesystem may each be modeled as a process.
- **Process state.** Each process P_i evolves through a sequence of local states $s_i^0, s_i^1, \dots$, where $s_i^j \in S_i$ denotes the state after its j -th local transition and S_i is its state space. Depending on the process, this state may include conversation history, workflow progress, intermediate results, permissions, in-memory variables, or filesystem contents. A global internal state combines one local state from each process:

$$S = (s_1^{k_1}, s_2^{k_2}, \dots, s_n^{k_n}).$$

The indices k_i need not be equal because different processes evolve independently.

- **Event.** An event $e = (a, \xi)$ represents an operation a together with the nondeterministic factors ξ that determine its realized outcome, inducing a transition

$$s_i^j \xrightarrow{e} s_i^{j+1}.$$

Events include internal computation, agent decisions, and tool invocations, while ξ may capture LLM sampling, randomness, or responses from external services.

- **Message.** A message m carries information to a process from another process or from the outside world. Examples include user inputs, inter-agent messages, and tool results. A delivered message may trigger an event and thereby change the receiving process state. We write $e_a \prec e_b$ when e_a causally precedes e_b , either through local execution order or through such an interaction.
- **Checkpoint.** A checkpoint records selected process states as a recovery point. Let $K \subseteq \mathcal{P}$ denote the processes covered by a checkpoint. We write

$$C_K = (s_i^{k_i})_{P_i \in K}.$$

The set K therefore defines the checkpoint’s *recovery boundary*. A full-state checkpoint is the special case $K = \mathcal{P}$. During execution, the system may retain multiple checkpoints, $\mathcal{C} = \{C^1, \dots, C^q\}$. We do not model in-flight messages as a checkpoint component. When a framework persists pending work or communication state, such information is included in the corresponding process state. This abstraction captures the coarse execution boundaries used by the agent systems characterized in Sec. II.

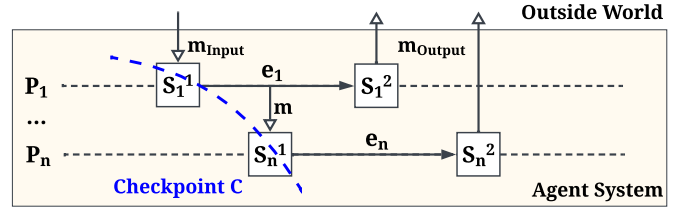


Fig. 1: System model for agent checkpoint and rollback.

- **Outside world.** The outside world W contains state and entities beyond the agent system that are not directly restored by agent rollback. Examples include remote services, payment systems, mailboxes, databases, user or organizational state, and persistent effects of previous agent actions. Processes may read from or modify W , but restoring an agent checkpoint does not in general restore the corresponding external state. Thus, even a checkpoint with $K = \mathcal{P}$ need not restore the complete state on which the execution depends.

Execution and recovery. Execution begins when an internal process receives an input, for example a user instruction delivered to the main agent. The resulting events update local states and may exchange messages or interact with W , producing further state transitions and external effects. During execution, the system records selected process states as checkpoints $C \in \mathcal{C}$.

Consider rollback at time t_r to a checkpoint C_K created earlier. For every $P_i \in K$, recovery restores the state recorded in C_K . Processes outside K are not controlled by this checkpoint and may retain their current state, restart from an earlier state, or be reconstructed from another source; we denote the recovered state of such a process P_j by ρ_j . Meanwhile, the outside world remains at its current state W^{t_r} . Rollback therefore reconstructs the configuration

$$\hat{X} = \left((s_i^{k_i})_{P_i \in K} \oplus (\rho_j)_{P_j \notin K}, W^{t_r} \right),$$

from which execution resumes.

This formulation makes explicit a key property of C/R: a recovered configuration may combine checkpointed state, uncovered internal state, and external state originating from different points in the execution.

B. Execution Continuity

Correctly restoring the contents of a checkpoint does not by itself imply that the resulting execution is security-valid. We capture this distinction through *execution continuity*: security-relevant states, decisions, assumptions, and effects carried across recovery must remain jointly consistent with a valid continuous execution history.

We use a *valid continuous execution* to denote an execution that follows the agent workflow and its security constraints without rollback, under the corresponding inputs, authorizations, and external interactions. Recovery can violate execution continuity in two fundamental ways [12]:

- **Invalid recovered state.** Rollback reconstructs a combination of states or assumptions that could not jointly occur in a valid continuous execution. For example, an agent may retain a successful validation result while restoring the validated artifact to an earlier, unverified version.
- **Invalid recovery transition.** The recovered state may itself be individually valid, but rollback enables a subsequent transition that no valid continuous execution would permit. For example, restoring a pre-payment state may allow a one-time approval to authorize a payment whose external effect has already occurred.

The security failures in Sec. IV characterize different ways in which C/R breaks these two forms of execution continuity.

C. Threat Model

Adversary goal. The adversary aims to exploit C/R to violate execution continuity, causing the agent either to resume from an invalid recovered state or to perform an invalid recovery transition. Importantly, the adversary need not corrupt a checkpoint or compromise the recovery implementation; the attack instead exploits how legitimate rollback behavior interacts with agent state and the outside world.

Adversary capabilities. We assume the adversary knows the task or workflow and the system’s publicly documented C/R behavior, consistent with malicious skills, MCP servers, tools, and external services considered in prior agent-security work [33], [34], [35], [36]. Depending on the concrete setting, the adversary may influence agent inputs, workflow instructions, or tool and service responses. Such influence can steer execution toward operations that trigger checkpoint creation or place the system in an advantageous state around a checkpoint.

The adversary may also induce a recoverable failure or interruption, for example by causing a malicious tool, MCP server, or external service to return an error or malformed response. Recovery may then be initiated through the framework’s normal mechanism by the user, framework, or deployment environment. When a rollback interface is directly exposed to an adversarial component, the adversary may invoke it within the permissions provided by that interface.

Trusted components. The adversary cannot directly modify checkpoint contents or storage, alter the semantics of the C/R implementation, or forge security decisions provided by otherwise trusted parties. We also do not assume deterministic replay: LLM outputs, tool responses, and other nondeterministic factors may legitimately differ after rollback. Each attack in Sec. V instantiates this general threat model with a concrete adversary and states its additional capabilities.

IV. SECURITY FAILURES IN AGENT C/R

Under the model in Sec. III, we identify five recurring ways in which checkpoint and rollback can violate *execution continuity*. The first two concern the internal state reconstructed by recovery: a checkpoint may omit recovery-relevant state (SF1) or combine captured states that are mutually inconsistent (SF2). The remaining failures arise from relations that cross

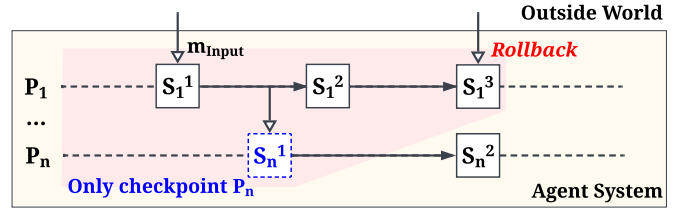


Fig. 2: SF1: Incomplete internal state coverage.

the checkpoint boundary: restored state may become incompatible with the current outside world (SF3), nondeterministic replay may follow a security-relevant execution path different from the original one (SF4), or rollback may forget effects that have already persisted externally (SF5). These conditions can yield either an invalid recovered state or an invalid recovery transition, as defined in Sec. III-B.

A. SF1: Incomplete Internal State Coverage

Definition (Fig. 2). Incomplete internal state coverage occurs when a checkpoint omits internal state required to safely resume the execution. For a task T , let $\mathcal{P}_T \subseteq \mathcal{P}$ denote the internal processes whose states are relevant to the resumed execution, and let

$$S_T = (s_i^{k_i})_{P_i \in \mathcal{P}_T}$$

denote their states when checkpoint C_K is created. The checkpoint has incomplete coverage if $\mathcal{P}_T \not\subseteq K$. Upon rollback, states in K are restored from C_K , whereas an omitted process $P_j \notin K$ recovers to some state ρ_j , as modeled in Sec. III-A. If $\rho_j \neq s_j^{k_j}$, recovery may combine internal states that never coexisted at the checkpointed execution point.

Consequences. The omitted state can diverge from the checkpoint in two directions:

- **Newer-state carryover.** An uncovered process may remain live across rollback and retain a state reached after the checkpoint. Specifically, if P_j evolves from $s_j^{k_j}$ at checkpoint creation to $s_j^{k'_j}$, where $k'_j > k_j$, rollback leaves $\rho_j = s_j^{k'_j}$. The recovered configuration therefore combines checkpointed state with a later state of P_j .
- **Older-state fallback.** An uncovered process may instead restart after a failure without access to its checkpoint-time state. It may therefore recover from $s_j^{k'_j}$, where $k'_j < k_j$, or from its initial state. The recovered configuration then combines checkpointed state with an earlier state of P_j .

For example, an agent validates a deployment manifest and records `validated=true` in framework state, while the manifest resides in an ephemeral workspace outside the checkpoint. After a crash, the workspace is reconstructed from its original template while the framework restores the post-validation state. The resumed agent may therefore deploy a manifest that was never validated.

In existing designs. Incomplete coverage can arise both from the intended recovery boundary and from

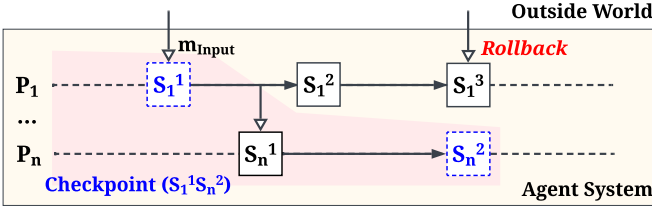


Fig. 3: SF2: Inconsistent checkpoint state.

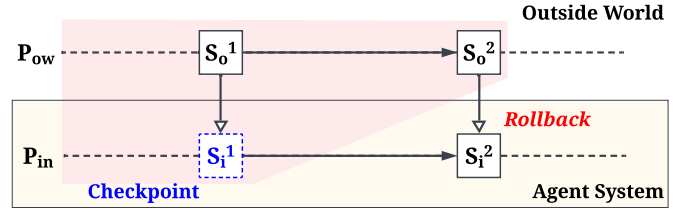


Fig. 4: SF3: External state mismatch.

implementation-specific exclusions. Framework-state checkpoints omit workspace and runtime state; workspace-state checkpoints additionally preserve workspace artifacts but generally omit process and environment state. Even OS/VM-state checkpoints need not cover framework-managed state stored outside the snapshotted environment or state maintained by external services. Concrete implementations can narrow coverage further: Git-based mechanisms typically snapshot only a selected project scope, while Hermes, for example, excludes individual files larger than 10 MB and applies additional scope and storage limits [24].

B. SF2: Inconsistent Checkpoint State

Definition (Fig. 3). SF1 concerns state that is *missing*; SF2 concerns captured states that are individually present but *mutually inconsistent*. Let

$$C_K = (s_i^{k_i})_{P_i \in K}$$

be a checkpoint, and let $H_i(s_i^{k_i})$ denote the local event history reflected by state $s_i^{k_i}$. Define

$$H(C_K) = \bigcup_{P_i \in K} H_i(s_i^{k_i}).$$

Using the causal relation $\prec$ from Sec. III-A, C_K is causally consistent if, for every $e_b \in H(C_K)$, each causal predecessor $e_a \prec e_b$ belonging to a captured process also appears in $H(C_K)$. Otherwise, the checkpoint contains states that cannot correspond to a single causally valid execution cut.

For example, suppose captured state of P_n reflects consumption of an event produced by P_1 , while the captured state of P_1 precedes the production of that event. Both local states may be valid individually, yet they cannot belong to the same execution history.

Consequences. Rollback to an inconsistent checkpoint reconstructs an invalid internal state. A result, approval, or validation may survive without the state that produced or justified it; control state may indicate that an operation has completed while associated data predates the operation; or different components may disagree on execution progress, causing later actions to be skipped, repeated, or reordered.

For example, an agent patches a vulnerable file, tests the patched version, and records `tests_passed=true` in its conversation state. If recovery restores the conversation state from after testing but the workspace from before the patch, both restored states are individually valid but do not

form a consistent causal cut. The agent may then submit the vulnerable file without repeating the patch or tests.

In existing designs. Framework-defined checkpoints can preserve consistency within the framework-managed state when that state is captured at a common execution boundary. Workspace-state mechanisms introduce an additional state domain whose checkpoint and restoration may not be coupled to framework state. Cline provides a concrete example: it allows the workspace, task history, or both to be restored from a checkpoint, so the two components can intentionally be rolled back independently [30]. OS/VM-state mechanisms can atomically capture a broader execution environment, preserving consistency within the recovery boundary.

C. SF3: External State Mismatch

Definition (Fig. 4). External state mismatch occurs when checkpointed internal state is restored against an outside world under which the assumptions encoded by that state are no longer valid. Let $s_i^{k_i}$ be an internal state captured in checkpoint C_K , and let W^{t_c} denote the outside-world state on which it depends when the checkpoint is created at time t_c . We define

$$\Gamma(s_i, W) \in \{0, 1\}$$

as a compatibility predicate indicating whether the security-relevant judgments, or assumptions represented by s_i remain valid under outside-world state W . At checkpoint creation,

$$\Gamma(s_i^{k_i}, W^{t_c}) = 1.$$

Because rollback does not restore the outside world, recovery at $t_r > t_c$ combines $s_i^{k_i}$ with W^{t_r} . An external state mismatch occurs when

$$\Gamma(s_i^{k_i}, W^{t_r}) = 0.$$

Consequences. A decision or transition that was valid under W^{t_c} may no longer be permitted under W^{t_r} . The agent may reuse a validation, approval, or trust decision for a changed external object; continue after policy or ownership status has changed; or act on an external binding that no longer holds.

For example, suppose an agent checkpoints a recipient list used to send sensitive documents to current employees. If an employee leaves after the checkpoint is created, rollback may restore the obsolete recipient list while the organization's membership state remains current. The resumed agent can then disclose subsequent documents to a former employee.

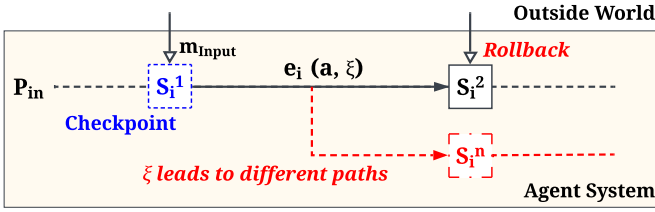


Fig. 5: SF4: Unbound nondeterministic replay.

In existing designs. Among the systems reviewed in Sec. II, we find no general mechanism that couples checkpoint restoration with revalidation of all external dependencies. This limitation is structural: the relevant outside-world state may be owned by remote services and remain beyond the recovery boundary even when the agent’s local execution environment is fully snapshotted. We discuss its relation to traditional stale-state recovery in Sec. VII.

D. SF4: Unbound Nondeterministic Replay

Definition (Fig. 5). Rollback commonly re-executes operations that occurred after the selected checkpoint. Such replay becomes security-relevant when, under the same restored state and relevant external conditions, a nondeterministic operation produces a different outcome that is not bound to the state or security decisions carried across recovery.

Suppose operation a , executed after checkpoint state s_i^j , realizes event

$$e_1 = (a, \xi_1)$$

and transitions to s_i^{j+1} . After rollback to s_i^j , the same operation may execute as

$$e_2 = (a, \xi_2), \quad \xi_2 \neq \xi_1,$$

producing a different security-relevant outcome $\hat{s}_i^{j+1} \neq s_i^{j+1}$. An *unbound nondeterministic replay* occurs when recovery permits this divergent outcome without preserving the security relationships established for the original execution.

Consequences. Two forms are particularly security-relevant.

- *Cross-path reuse.* Security-relevant state established for one continuation—such as an approval, validation result, permission, or object binding—remains usable after replay follows a different continuation for which that state is no longer valid.
- *Retry amplification.* Repeated rollback turns a security-sensitive nondeterministic decision into repeated trials until a favorable outcome appears.

For example, consider a release agent that checkpoints immediately before a randomized regression test that samples a subset of the test suite. The original execution samples a test that detects a defect but fails before the result is durably recorded. After rollback, replay draws a different sample that omits the detecting test, allowing the build to pass and be released. Neither test execution is individually invalid; the violation arises because rollback substitutes one security-relevant outcome for another.

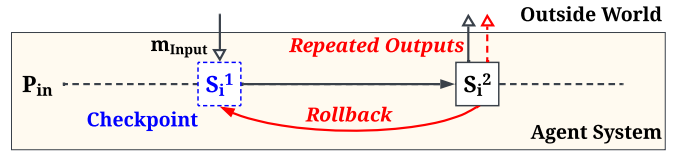


Fig. 6: SF5: Unrecorded external effects.

In existing designs. The frameworks reviewed in Sec. II generally permit post-checkpoint LLM calls and tool operations to be executed again, but we find no general mechanism that binds security-relevant nondeterministic outcomes across replay or constrains repeated rollback. The issue is particularly salient for agents because model sampling and external tool responses make nondeterminism a routine part of execution, rather than an exceptional source of divergence.

E. SF5: Unrecorded External Effects

Definition (Fig. 6). An unrecorded external effect occurs when an action executed after a checkpoint produces an effect outside the recovery boundary, but rollback to that checkpoint removes the internal record that the action has occurred. Specifically, let checkpoint C capture internal state s_i^j . After C , an action a transitions the internal state and produces an external effect:

$$s_i^j \xrightarrow{a} s_i^{j+1}, \quad W \xrightarrow{a} W'.$$

If the system later rolls back to C , it restores s_i^j , while the external effect persists in W' . The recovered execution therefore reaches a again from a state in which the action appears unexecuted, even though its previous effect remains. Re-executing a may consequently produce the external effect.

Consequences. This mismatch can manifest in two ways:

- *Orphaned effect.* Rollback removes the internal result, identifier, or metadata required to track an already committed effect, preventing later confirmation or revocation.
- *Effect replay.* The restored agent treats the action as unexecuted and issues it again, potentially duplicating a non-idempotent external effect.

For example, an agent requests a new cloud credential and intends to record its identifier before revoking the old one. The service successfully creates credential k_1 , but the agent fails before recording it. Rollback restores the pre-request state, so the agent repeats the request and receives k_2 . It subsequently manages k_2 , while k_1 remains active and untracked.

In existing designs. Among the systems reviewed in Sec. II, we find no general mechanism that atomically couples arbitrary externally committed effects with agent checkpoints. External services may independently commit payments, messages, credentials, or other state changes before the corresponding result becomes durable in the agent’s recovery state. Traditional systems provide mechanisms such as idempotence, transactions, and compensation for controlled external effects, but agent workflows interact dynamically with heterogeneous

services that may not expose such coordination interfaces. We discuss these distinctions further in Sec. VII.

Takeaway. The five SF expose a common gap: C/R restores execution according to a recovery boundary, whereas security validity can depend on relationships among internal state, external context, nondeterministic outcomes, and persistent effects that cross boundary. Consequently, even specification-conformant rollback can violate execution continuity without corrupting checkpoint contents or the recovery mechanism. In Sec. V, we show how adversaries with limited influence over execution and C/R can deliberately trigger these conditions to realize concrete attacks.

V. ATTACK CASE STUDIES

We demonstrate that the recovery failures identified in Sec. IV can be deliberately exploited to cause security-critical actions. We construct three end-to-end attacks on different agent frameworks and under different adversarial settings, instantiating incomplete state coverage (SF1), external state mismatch (SF3), and unrecorded external effects (SF5). None of the attacks requires modifying checkpoint contents or compromising the C/R implementation.

A. Case 1: Malware Verification Bypass

Task and framework. We consider a repository-inspection task executed by Hermes. The user asks the agent to identify and remove malicious code, verify the cleaned repository, and prepare it for release. We adapt the real execution flow from a public SWE-agent trajectory for the SWE-bench task `marshmallow-code_marshmallow-1867` [37], where the agent modifies the repository, verifies the resulting behavior, and submits the change only after verification succeeds. Hermes checkpoints workspace files using shadow-Git snapshots before file modifications, whereas security judgments produced during execution may reside in agent-side state outside the workspace checkpoint. In a normal execution, the agent removes the malicious payload, verifies the cleaned repository, and releases it only after verification succeeds.

Adversarial setting. Following the malicious-skill setting of prior work [34], we assume that the task incorporates an attacker-controlled skill whose instructions influence the agent workflow. The adversary cannot modify Hermes’s checkpoints or security checker. Its goal is to make a repository containing the malicious payload appear verified and eligible for release.

Attack workflow. The malicious skill steers execution through the following sequence.

- *Step 1: Checkpoint the malicious repository.* The skill instructs the agent to remove the malicious payload, requiring a file modification. Before the modification, Hermes creates checkpoint C_0 , which preserves the malicious workspace version X_M . The edit then produces the cleaned version X_C .
- *Step 2: Verify the cleaned repository.* The agent runs the trusted security check on X_C , producing a passing verification result V_C . The result is retained in agent-side execution

state, whereas the workspace checkpoint remains C_0 , which represents X_M .

- *Step 3: Induce recovery.* The skill subsequently directs the agent to access a required release artifact from an invalid location, causing a recoverable failure. The user then invokes Hermes’s normal rollback mechanism and selects C_0 . Rollback restores the workspace to X_M , while the relevant verification state V_C remains available to the resumed execution, yielding the combination (X_M, V_C) .
- *Step 4: Release under stale verification.* The workflow treats the completed verification as sufficient for release. The resumed agent therefore bypasses the security check and releases X_M , even though V_C was produced only for X_C .

Security violation. The attack instantiates SF1: rollback restores the workspace without restoring all security-relevant agent state to the same execution point. As a result, a genuine verification judgment V_C becomes rebound from the artifact it actually validated, X_C , to the malicious artifact X_M . The attack therefore bypasses verification without forging either the checkpoint or the security result. We provide the complete malicious skill in Appendix A. In our experiments, a recent malicious-skill detector [38] does not flag the constructed skill.

B. Case 2: Unauthorized Mail Forwarding

Task and framework. We consider a mailbox-migration task executed by Cline. The task is adapted from the OSWorld-Human Thunderbird [39], [40] forwarding task, while we implement the mailbox operations using a controlled test service. We extend this workflow with a disposable mailbox M_T for testing and a real mailbox M_R for read-only compatibility checks. The mailbox service maintains a session-level *active mailbox*, and `create_forwarding_rule` applies to whichever mailbox is currently active. A normal migration repeatedly tests changes on M_T , inspects M_R , and then returns to M_T for testing.

Adversarial setting. The attacker controls a MCP service used for migration compatibility checks. The MCP service may manipulate the information returned to Cline and induce recoverable failures, but exposes no mailbox-mutation primitive and cannot issue forwarding requests. Mailbox selection and forwarding are performed through a separate trusted mailbox interface under the user’s authorized session. The adversary cannot directly modify M_R , forge user authorization, or alter Cline’s C/R implementation. Its goal is to cause a forwarding operation authorized only for M_T to execute on M_R .

Attack workflow. The attack exploits the fact that Cline can restore its task state without restoring the mailbox service’s session state.

- *Step 1: Checkpoint the pending test operation.* The agent selects M_T and prepares a forwarding rule for the disposable mailbox. Before issuing the forwarding request, it records the migration configuration in the workspace, causing Cline to create checkpoint C_0 . At C_0 , the task state identifies M_T as the intended mailbox and the forwarding operation remains

pending. The agent then executes the operation, which is correctly applied to M_T .

- *Step 2: Switch the external binding.* The workflow next selects M_R for a read-only compatibility check. Although the check does not modify M_R , selecting it changes the mailbox service’s session-level active mailbox from M_T to M_R . This external binding is outside Cline’s recovery boundary.
- *Step 3: Roll back the task state.* During the compatibility check, the malicious MCP server induces a recoverable failure. The user restores C_0 using Cline’s task-history-only recovery, which returns the workflow to the point where the forwarding operation is still pending. The mailbox service is not rolled back and therefore remains bound to M_R .
- *Step 4: Re-execute under the changed binding.* The resumed workflow re-executes the pending forwarding operation. The restored task state still identifies M_T as the intended target, but the mailbox service resolves the request against its current active mailbox, M_R . The forwarding rule is therefore installed on the real mailbox.

Security violation. The attack instantiates SF3. Rollback restores an operation whose authorization context refers to M_T , while the external resource binding has advanced to M_R . Recovery therefore rebinds an authorized test action to an unintended live resource, causing real emails to be forwarded. The attack illustrates a broader risk for workflows that alternate between test and production resources whose bindings are maintained outside the agent’s recovery boundary.

C. Case 3: Duplicate Payment under One Approval

Task and framework. We consider an invoice-payment workflow implemented in LangGraph. We follow the payment flow from AgentDojo’s Banking benchmark [33], [41]. The workflow verifies an invoice, obtains user approval, submits the payment, verifies the receipt, and records the resulting transaction. Payment submission and receipt verification execute within the same graph node, whose state is checkpointed only after the node completes. In a normal execution, one user approval authorizes one payment, after which the transaction and receipt become durable before execution continues.

Adversarial setting. The payment service itself is trusted, but the receipt is retrieved from an attacker-controlled remote service after payment submission. The adversary can observe that payment has completed and control the receipt response, but cannot modify the payment service, forge user approval, or alter LangGraph’s C/R mechanism. Its goal is to cause two payments to be issued under a single approval.

Attack workflow. The attack exploits the interval between an externally committed effect and the checkpoint that would record its completion.

- *Step 1: Checkpoint the approved state.* The user approves the invoice and the approval node completes. LangGraph creates checkpoint C_0 , recording that the payment is authorized but not yet completed.

- *Step 2: Commit the first payment.* The payment node submits the transaction, and the trusted payment service commits payment T_1 . The node then requests the corresponding receipt from the attacker-controlled service, which deliberately returns a failure. Because the node does not complete, no subsequent checkpoint records T_1 ; the durable graph state therefore remains at C_0 , while T_1 persists externally.
- *Step 3: Replay after recovery.* Recovery resumes from C_0 , where the payment remains authorized and appears pending. LangGraph consequently re-executes the payment node. Because the external payment T_1 survives rollback and the repeated submission is not bound to a stable idempotency identifier, the payment service commits a second payment.

Security violation. The attack instantiates SF5. Rollback restores a pre-payment internal state over a post-payment outside world, erasing the agent’s durable knowledge of T_1 while leaving the effect itself intact. The same approval can therefore authorize a second externally committed effect. Moreover, any non-idempotent external action can be replayed if its effect becomes durable before the agent records its completion and an adversary can induce failure within this interval.

Takeaway. Across all three attacks, rollback rebinds a valid security decision to an invalid context, *i.e.*, validation to an unverified artifact, authorization to an unintended resource, or approval to a duplicated effect. Thus, correct C/R can still break execution continuity and become exploitable.

VI. EMPIRICAL STUDY

In this section, we measure checkpoint security failures in realistic agent executions, addressing two main questions:

- **[RQ1] Prevalence:** How prevalent are the five security failures in real-world agent tasks?
- **[RQ2] Root causes and lessons:** What broader insights do validated failures reveal, and what lessons can we learn for better checkpoint design?

A. Methodology

1) *Overview:* As shown in Fig. 7, we build a multi-agent pipeline to identify and validate checkpoint security failures. The *collector agent* executes the target task and collects its runtime trace and program context. The *analyzer agent* reconstructs execution semantics into a structured event trace by extracting events, versioned internal and external states, and their dependencies. The *checker agent* applies the five SF-specific rules to identify candidate failures, while the *validator agent* restores selected checkpoints using the system’s native C/R mechanism and determines whether the candidates manifest after recovery.

2) *Analysis workflow:* We next describe the four agents and their analysis procedures.

Collector agent. Given a task and a target C/R system, the collector agent executes the task and collects two inputs:

- *Runtime trace.* The observed execution trace, including model and framework operations, checkpoint creation and

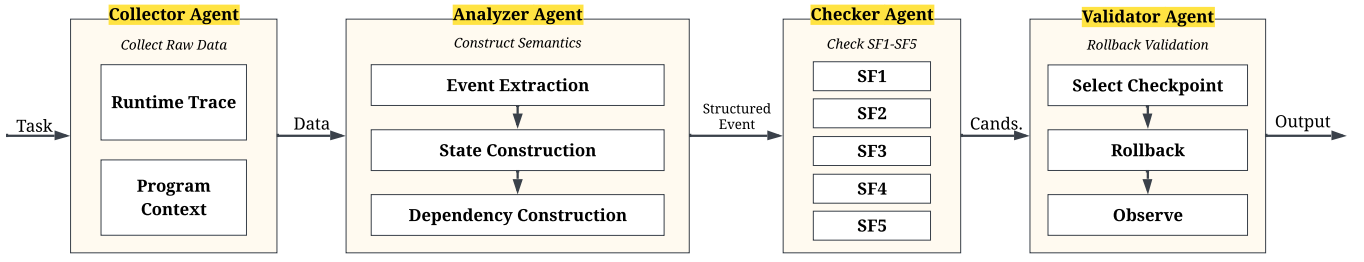


Fig. 7: Methodology overview.

contents, observed state changes, tool inputs and outputs, and interactions with internal and external components.

- **Program context.** The static information needed to interpret the observed execution, including task-specific code and artifacts, workflow implementation, tool definitions, and relevant framework and C/R configuration.

Analyzer agent. The analyzer agent transforms the runtime trace and execution context into a structured event trace through three steps. First, during *event extraction*, it normalizes framework-specific records into semantic events, including model decisions, framework operations, tool invocations, and external interactions, and places each event relative to the observed execution order and checkpoint boundaries. Second, during *state construction*, it reconstructs the states consumed and produced by each event. For each state, the analyzer records its object, observed version, whether it is internal or external, and its coverage by each relevant checkpoint. Persistent changes to external state are recorded as external effects. We represent each event e_i as

$$e_i = \langle \text{op}_i, \text{In}_i, \text{Out}_i, \text{ND}_i \rangle,$$

where op_i is the performed operation, In_i and Out_i are the versioned states consumed and produced by the event, respectively, and ND_i records its source of nondeterminism. Third, during *dependency construction*, the analyzer reconstructs semantic dependencies among events and states. These include event-state dependencies between operations and their inputs or outputs, causal dependencies between events, and state-state dependencies such as reference, derivation, or containment relations. We record $e_i \prec e_j$ when e_j causally depends on e_i . The analyzer further identifies security-relevant states, decisions, bindings, and effects from the task specification, program context, and observed execution. The analyzer further annotates security-relevant states and outputs using the task specification, program context, and reconstructed dependencies. A state or output is marked security-relevant if changing it can cause task failure or alter the task outcome. Persistent state-changing external outputs are additionally marked as security-relevant external effects, while read-only interactions are excluded. All annotations are fixed before rollback validation. The resulting structured trace provides the semantic dependencies required by the subsequent SF-specific checkers.

Checker agent. Given the structured event trace, the checker agent evaluates the checkpoint using five SF-specific rules.

- **SF1.** For each event e_i after checkpoint C , we examine the internal states in In_i . We report a candidate when e_i depends on a state version established before C that is not covered by C and is not reconstructed between C and e_i .
- **SF2.** For each state captured by C , we trace its event and state dependencies. We report a candidate when a captured state reflects an event or state transition whose required predecessor in another captured component is not reflected by the version stored in C . The captured states therefore cannot form a causally consistent execution cut.
- **SF3.** For each internal state captured by C , we trace its semantic dependencies to external states. We report a candidate when the restored internal state carries a judgment, authorization, binding, or assumption derived from external state outside the recovery boundary.
- **SF4.** For each event e_i after checkpoint C , we examine ND_i . We report a candidate when e_i involves nondeterminism and will be re-executed after rollback to C .
- **SF5.** For each event e_i after C , we examine the external effects in Out_i . We report a candidate when e_i produces a persistent effect outside the recovery boundary while rollback to C restores the agent to a state preceding that effect.

Validator agent. For each execution trace, the validator restores the checkpoint selected for validation using the target C/R system's native recovery mechanism. It then observes the recovered state and, when needed, re-executes the relevant post-checkpoint operations. The validator evaluates the security-relevant states, decisions, bindings, and effects identified by the analyzer before rollback. A candidate is reported as a validated finding only when rollback invalidates the corresponding security-relevant relation:

- **SF1.** We confirm SF1 when an internal state outside the recovery boundary recovers to a version different from its checkpoint-time version, and the mismatched state is marked as security-relevant by the analyzer.
- **SF2.** We confirm SF2 when rollback materializes the causal inconsistency identified by the checker, and involves internal state marked as security-relevant by the analyzer.
- **SF3.** We confirm SF3 when an external state on which the checkpoint depends has changed after checkpoint creation,

and the changed external state is marked as security-relevant.

- *SF4*. We re-execute the relevant post-checkpoint execution while keeping the restored state and relevant external conditions unchanged. We confirm SF4 when nondeterministic replay produces an output different from the original execution, and the changed output is marked as security-relevant.
- *SF5*. We confirm SF5 when a security-relevant persistent external effect produced after checkpoint C remains after rollback, while the recovered agent state no longer records that effect, allowing the effect to be issued again or leaving the existing effect untracked.

B. Experimental Setup

Tasks and datasets. We evaluate our framework on two agent benchmarks, yielding 347 execution traces and 1,735 framework-task executions across five frameworks. We use 241 traces from Terminal Bench [42], covering terminal-based tasks in software engineering, system configuration, machine learning, scientific computing, and security. We further use 106 traces from three AgentBench [43] environments: *OS* for Bash-based system interaction, *DB* for SQL-based database operations, and *ALFWorld* for multi-step tasks in a simulated household environment. Together, they cover diverse interactions with files, databases, and external environments.

Checkpointing frameworks. We evaluate the five representative frameworks studied in Sec. II: LangGraph, CrewAI, Hermes, Cline, and E2B. For LangGraph and CrewAI, we use a fixed agent–tool workflow shared across all tasks, rather than constructing a task-specific workflow. Within this workflow, the LLM autonomously selects tools and actions based on the task and execution state until completion. Checkpoints are created according to each framework’s checkpoint policy, *i.e.*, after each superstep in LangGraph and each completed task in CrewAI. Hermes and Cline execute each task directly, with checkpoints generated by their native automatic triggers. E2B does not define an automatic checkpoint policy, so we create a snapshot after each completed tool call during execution.

Execution and validation. For each task, we first execute it to completion while collecting the execution trace and all checkpoints created during the run. For validation, we use a random checkpoint in each trace as a fixed recovery point across executions. We restore this checkpoint using the framework’s native recovery mechanism and re-execute the relevant post-checkpoint execution to validate the candidates identified at this recovery point. For nondeterministic-replay candidates, we independently re-execute the relevant post-checkpoint execution three times and validate SF4 when the replay produces a different security-relevant execution path.

Experimental configuration. We use DeepSeek-v4 [44] as the backend LLM for agent execution, workflow construction, and semantic event reconstruction. All experiments run on a Linux server with 16 CPU cores, 32 GB of memory. Each task is executed in an isolated sandbox initialized from the same base environment. The sandbox and associated test services are reset before each task execution, while rollback validation

TABLE II: Prevalence of security failures.

Framework	Dataset	SF1	SF2	SF3	SF4	SF5
LangGraph	AgentBench	99	0	49	89	6
	TerminalBench	201	0	50	139	11
CrewAI	AgentBench	106	0	0	80	3
	TerminalBench	222	0	0	196	4
Hermes	AgentBench	71	97	1	80	11
	TerminalBench	183	143	43	140	0
Cline	AgentBench	89	88	0	93	1
	TerminalBench	195	161	0	182	3
E2B	AgentBench	0	0	0	75	1
	TerminalBench	0	0	55	100	0
Overall	1735	1166	489	198	1174	40

is performed within the corresponding execution environment so that post-checkpoint state and external effects are preserved as required by the validation.

Metrics. We measure the prevalence of each SF at the execution level. For each task execution, we record whether a given SF is observed at least once during its execution and rollback.

C. RQ1: Prevalence of Security Failures

We first measure how frequently the five SFs manifest in real agent executions. For each execution trace, we record whether each SF is validated at the selected recovery point. Because checkpoint creation policies differ across frameworks, we use framework-level results to characterize where failures manifest under each evaluated configuration, rather than as a direct security ranking among frameworks. Table II reports the trace-level prevalence across the two evaluated benchmarks.

Across security failures. The five security failures show substantially different prevalence (Table II).

- *SF1 and SF4 are the most prevalent failures.* SF1 and SF4 occur in 67.2% and 67.7% of executions, respectively. SF1 is common because agent tasks span framework, workspace, and runtime state, while checkpoints often cover only a subset of the internal state required by later execution. SF4 is similarly common because post-checkpoint LLM decisions and tool interactions are nondeterministic and may diverge when re-executed after rollback.
- *SF5 is comparatively rare.* SF5 occurs in only 2.3% of executions, mainly due to the evaluated task distribution. Most TerminalBench and AgentBench tasks operate on local files, system state, databases, or simulated environments, while few produce persistent, non-idempotent effects through external services. Thus, few executions contain the external effects required for SF5.
- *SF2 and SF3 require more specific state relationships.* SF2 requires dependent state components to be restored inconsistently, while SF3 requires checkpointed state to depend on external state that changes before recovery. These additional conditions make both less frequent than SF1 and SF4.

Across frameworks. The three recovery-boundary designs also exhibit distinct failure profiles.

- *E2B provides the strongest internal-state coverage.* We observe no SF1 or SF2 in E2B because its system-level snapshots jointly capture sandbox filesystem and memory, providing stronger completeness and consistency within the sandbox. This coverage does not extend to the outside world, so SF3–SF5 remain possible. SF5 is nevertheless rare because few evaluated tasks perform persistent external API actions.
- *LangGraph and CrewAI show similar framework-state failure profiles.* Both preserve framework-managed state but exclude workspace state, resulting in high SF1, while their framework-defined checkpoint boundaries yield no observed SF2. CrewAI shows no SF3 because it checkpoints at task start, before external dependencies are established.
- *Hermes and Cline expose similar workspace-state tradeoffs.* Including workspace files reduces some incomplete-coverage cases, but requires coordinating workspace state with framework or conversation state, resulting in substantial SF2. Cline can restore workspace and task history independently, directly combining states from different execution points. Hermes restores the selected workspace checkpoint while rolling back only the most recent conversation turn (due to implementation), which can similarly leave the workspace and conversation state at different execution points.

Manual validation and exploitability. Manual inspection shows that our framework achieves 98.7% precision, 99.9% recall, and 99.5% accuracy (Appendix B). We further sample 30 validated findings, construct attacks under the malicious-skill adversary model, and manually confirm that all 30 succeed. The corresponding attacks and execution traces are included in our artifact.

D. RQ2: Root Causes and Lessons

The validated failures arise from two recurring sources. At the design level, existing C/R mechanisms primarily preserve the state needed to resume execution, without systematically tracking security-relevant dependencies across internal state, external context, nondeterministic decisions, and persistent effects. At the implementation level, framework-specific choices in checkpoint scope, creation timing, and restoration behavior determine how these dependencies are omitted, invalidated, or rebound during recovery. Beyond these direct causes, our analysis also reveals several recurring characteristics of agent execution that independently affect checkpoint security and suggest broader lessons for C/R design.

Observation 1: Semantic references introduce external dependencies. Agent state often represents otherwise concrete execution facts through context-dependent semantic references, implicitly making their interpretation depend on external conditions. These hidden dependencies enlarge the surface for SF3: an adversary can manipulate the relevant external condition or rollback timing so that the same restored semantic state is interpreted under a different context. For example, an agent may make a decision based on the “current timestamp” without recording the concrete timestamp t_c ; after rollback at t_r , the same reference resolves to t_r , potentially changing

TABLE III: Frequent semantic references to external context.

Reference	#	Reference	#
<i>now</i>	222	<i>earlier</i>	52
<i>current state</i>	178	<i>previous run</i>	44
<i>this environment</i>	87	<i>this task</i>	41
<i>currently</i>	83	<i>here</i>	39

the object, policy, or condition to which the restored state applies. We analyze all framework-task executions across the five evaluated frameworks. In total, we identify 1,357 semantic references that make checkpointed state dependent on external context. Table III reports the most frequent references; the complete results are provided in the artifact.

Takeaway. Checkpointed state should use concrete, deterministic references whenever possible, rather than context-dependent semantic references. Explicitly binding time, object identity, version, or other relevant values reduces hidden dependencies on the outside world, making recovery more deterministic and reducing the attack surface for SF3.

Observation 2: Under-specified recovery state amplifies replay nondeterminism. Rollback re-executes post-checkpoint decisions, and LLM sampling can therefore produce different outputs from the same checkpoint. Whether this nondeterminism changes the concrete execution, however, depends on how strongly the recovered state constrains the next action. When multiple continuations are semantically valid, small sampling differences can select different actions and quickly produce divergent execution paths. We observe three recurring sources of such under-specification. *Missing procedural details* leave the next step or execution order open, such as whether to inspect, modify, or verify first. *Missing correctness constraints* leave part of the required success condition unspecified, causing different replays to implement or validate different interpretations. *Missing environment details* leave relevant dependency or runtime state unknown, allowing replays to choose different environment-dependent actions.

Takeaway. Recovery should preserve enough execution constraints to narrow replay to a well-defined continuation. Explicit procedures, correctness conditions, and relevant environment state reduce the space of valid post-rollback actions and therefore limit SF4.

Observation 3: Commit ordering cannot eliminate the external-effect gap. Ordering an external effect and its checkpoint update cannot by itself keep agent state consistent with the outside world. A failure between them leaves a recovery gap: *execute*→*commit* may preserve an external effect while losing its record, enabling SF5 and effect replay, whereas *commit*→*execute* may preserve completion state for an effect that never occurred. For example, the former may duplicate a payment after rollback, while the latter may skip a payment that was never issued. We construct a controlled microbenchmark of 96 persistent external-action workflows, following action patterns in Terminal-Bench and covering

TABLE IV: Recovery failures under different orderings.

External Action	#	Execute →Commit	Commit →Execute
Payment	30	28	29
Message	31	29	30
Resource change	35	33	34
Overall	96	90 (93.8%)	93 (96.9%)

payments, messages, and resource creation or updates. For each workflow, we build equivalent *execute*→*commit* and *commit*→*execute* variants, inject a recoverable failure between the two operations, and resume from the latest checkpoint using the framework’s native recovery mechanism. We count a failure when recovery either repeats an effect that already occurred or skips an effect that was never issued. As shown in Table IV, neither ordering closes the gap: execute-then-commit fails in 90 cases, while commit-then-execute fails in 93.

Takeaway. Preserving consistency across external effects requires support beyond checkpoint ordering alone. Agent C/R should rely on external coordination mechanisms, such as transactions, idempotency guarantees, or durable effect records, to bind agent progress with externally committed actions and prevent SF5 across rollback.

VII. DISCUSSION

Toward secure agent recovery. Secure recovery requires more than capturing a larger system snapshot. Prior work provides mechanisms for protecting persistent state against rollback [12], reconciling external effects [45], and controlling nondeterministic replay [46]. Agent executions, however, combine these concerns across state domains that may not share a recovery boundary. Secure recovery therefore requires preserving or re-establishing security-relevant dependencies across internal state, external assumptions, persistent effects, and replay outcomes.

Scope and limitations. Our study focuses on security violations introduced by checkpoint and rollback recovery. The five failure modes characterize recurring ways in which recovery can break execution continuity under our model; they are not intended to cover all security failures in agent systems. Likewise, our analysis identifies recovery hazards, while their exploitability depends on the surrounding application and adversarial capabilities. Our testing methodology is therefore intended to expose and validate recovery violations rather than certify the security of an agent system.

Recovery across heterogeneous state domains. Long-running agent executions increasingly span agent frameworks, subagents, tool runtimes, workspaces, and external services. Security-relevant dependencies can cross these components even when their states are managed and recovered independently. Checkpointing exposes this mismatch particularly clearly: a locally valid recovery can violate assumptions established elsewhere in the execution. This suggests that recovery

should be treated as a cross-component security boundary, rather than solely restoring individual component state.

VIII. RELATED WORK

Traditional checkpointing and anti-rollback security. Traditional checkpoint and rollback research asks whether an execution can be restored to a consistent state, through mechanisms such as distributed snapshots [10], coordinated and uncoordinated checkpointing [47], and log-based recovery [11]. Anti-rollback security instead asks whether protected state has been illegitimately restored to an authentic but stale or forked version [12], [48], [49], [50], [13].

Agent checkpoint and recovery systems. Recent work expands agent recovery across different state domains. CRAB selects checkpoint granularity based on OS-visible effects [14], DeltaBox provides low-latency filesystem and process-state rollback [15], and AgentRewind jointly checkpoints agent context and its execution environment [51]. Other systems support reversible or branchable execution through typed traces, shared logs, or OS-level branches [52], [53], [54]. These works focus on capturing and restoring execution state; we instead ask whether security-relevant facts remain valid after recovery.

Agent resume semantics and rollback security. Recent work has begun examining correctness and security issues in agent recovery. Resume Means Resume studies conformant resume semantics [16], DART formalizes semantic recoverability [17], and ACRFence demonstrates rollback attacks involving replay and consumed authority [55]. Related transactional systems such as Atomix and Cordon address safe commitment of external effects [56], [57]. These works expose important aspects of agent recovery, but do not provide a comprehensive security analysis of checkpoint and rollback. Our work characterizes the security failures that arise across internal state, external context, nondeterministic replay, and persistent effects.

Broader agent system security. Prior agent-security work studies malicious inputs and components, including prompt injection [36], [58], malicious tools and MCP servers [35], memory leakage [59], and attacks across multi-agent interactions [60]. Benchmarks such as AgentDojo and ASB evaluate these threats during agent execution [33], [60].

IX. CONCLUSION

We presented a security study of checkpoint and rollback in agent systems. By characterizing representative agent C/R systems, we developed a recovery model and identified five recurring failure classes spanning internal state, external context, nondeterministic replay, and persistent effects. We demonstrated their security impact through three end-to-end attacks on Hermes, Cline, and LangGraph, and developed a trace-based methodology to identify and validate such failures across C/R designs. Our study shows that correct checkpoint restoration does not guarantee secure recovery: rollback can break the dependencies that justify security-relevant state. Secure recovery therefore requires preserving these dependencies, rather than merely restoring checkpoint contents.

REFERENCES

- [1] S. Schmidgall, Y. Su, Z. Wang, X. Sun, J. Wu, X. Yu, J. Liu, M. Moor, Z. Liu, and E. Barsoum, "Agent laboratory: Using llm agents as research assistants," *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 5977–6043, 2025.
- [2] J. Yang, C. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.
- [3] Microsoft, "2025: The year the frontier firm is born," Microsoft Work Trend Index, 2025, accessed: 2026-08-18. [Online]. Available: <https://www.microsoft.com/en-us/worklab/work-trend-index/2025-the-year-the-frontier-firm-is-born>
- [4] Tata Steel, "Tata steel partners with google cloud to deploy a unified agentic ai across its global value chain," <https://www.googlecloudpresscorner.com/2026-04-22-Tata-Steel-Partners-with-Google-Cloud-To-Deploy-a-Unified-Agentic-AI-Across-its-Global-Value-Chain>, 2026, press release, Apr. 22, 2026; accessed Aug. 13, 2026.
- [5] Cognizant, "How cognizant turned a fragmented intranet into a multi-agent system serving 350,000 employees," <https://www.cognizant.com/us/en/ai-lab/blog/cognizant-ai-agents-enterprise-intranet-transformation-neuro-san>, 2026, June 2, 2026; accessed Aug. 13, 2026.
- [6] Kore.ai, "The kore.ai agent productivity index 2026," <https://www.kore.ai/research/agent-productivity-index-report>, 2026, survey commissioned by Kore.ai and conducted by Propeller Insights; accessed Aug. 13, 2026.
- [7] LangChain, "Persistence," <https://docs.langchain.com/oss/python/langgraph/persistence>, 2026, langGraph documentation; accessed Aug. 13, 2026.
- [8] Cline, "Checkpoints," <https://docs.cline.bot/core-workflows/checkpoints>, 2026, cline documentation; accessed Aug. 13, 2026.
- [9] E2B, "Sandbox snapshots," <https://e2b.dev/docs/sandbox/snapshots>, 2026, e2B documentation; accessed Aug. 13, 2026.
- [10] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.
- [11] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.
- [12] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, "Memento: Practical state continuity for protected modules," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2011, pp. 379–394.
- [13] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography," in *Proceedings of the Network and Distributed System Security Symposium*, 2010.
- [14] T. Wu, C. Chang, L. Cao, W. Gao, and W. Wang, "Crab: A semantics-aware checkpoint/restore runtime for agent sandboxes," *arXiv preprint arXiv:2604.28138*, 2026.
- [15] Y. Dong, J. He, Y. Hou, D. Du, Z. Xu, S. Yu, Y. Xia, and H. Chen, "Deltabox: Scaling stateful ai agents with millisecond-level sandbox checkpoint/rollback," *arXiv preprint arXiv:2605.22781*, 2026.
- [16] S. Khan, "Resume means resume: A machine-checked conformance contract for checkpoint, interrupt, and resume semantics in workflow persistence layers," *arXiv preprint arXiv:2608.03836*, 2026.
- [17] K. Yang, P. Li, Z. Wu, K. Xu, H. Huang, and X. Huang, "DART: Semantic recoverability for structured tool agents," *arXiv preprint arXiv:2605.23311*, 2026.
- [18] CrewAI, "Crewai documentation," 2026, accessed: 2026-08-11. [Online]. Available: <https://docs.crewai.com/>
- [19] LlamaIndex, "Agent workflows," LlamaIndex Developer Documentation, 2026, accessed: 2026-08-20. [Online]. Available: <https://developers.llamaindex.ai/python/llamaagents/workflows/>
- [20] Google, "Agent development kit," 2026, accessed: 2026-08-11. [Online]. Available: <https://adk.dev/>
- [21] Microsoft, "Agent framework documentation," 2026, accessed: 2026-08-11. [Online]. Available: <https://learn.microsoft.com/en-us/agent-framework/>
- [22] LangChain, "Time travel," LangGraph Documentation, 2026, accessed: 2026-08-11. [Online]. Available: <https://docs.langchain.com/oss/python/langgraph/use-time-travel>
- [23] CrewAI, "Checkpointing," CrewAI Documentation, 2026, accessed: 2026-08-11. [Online]. Available: <https://docs.crewai.com/v1.15.16/en/concepts/checkpointing>
- [24] Nous Research, "Checkpoints and /rollback," Hermes Agent Documentation, 2026, accessed: 2026-08-11. [Online]. Available: <https://github.com/NousResearch/hermes-agent/blob/main/website/docs/user-guide/checkpoints-and-rollback.md>
- [25] Cline, "Cline documentation," 2026, accessed: 2026-08-11. [Online]. Available: <https://docs.cline.bot/>
- [26] Google, "Checkpointing," Gemini CLI Documentation, 2026, accessed: 2026-08-11. [Online]. Available: <https://gemini-cli.com/docs/cli/checkpointing/>
- [27] Anthropic, "Claude code documentation," 2026, accessed: 2026-08-11. [Online]. Available: <https://docs.anthropic.com/en/docs/claude-code>
- [28] Nous Research, "Hermes agent," 2026, accessed: 2026-08-11. [Online]. Available: <https://github.com/NousResearch/hermes-agent>
- [29] Cline, "Tasks," Cline Documentation, 2026, accessed: 2026-08-11. [Online]. Available: <https://docs.cline.bot/core-workflows/task-management>
- [30] —, "Using cline in your IDE," Cline Documentation, 2026, accessed: 2026-08-11. [Online]. Available: <https://docs.cline.bot/core-workflows/checkpoints>
- [31] E2B, "Sandbox snapshots," E2B Documentation, 2026, accessed: 2026-08-11. [Online]. Available: <https://docs.e2b.dev/sandbox/snapshots>
- [32] —, "Sandbox snapshots," E2B Documentation, 2026, accessed: 2026-08-11. [Online]. Available: <https://e2b.dev/docs/sandbox/snapshots>
- [33] E. Debenedetti, J. Zhang, M. Balunovic, L. Beurer-Kellner, M. Fischer, and F. Tramèr, "AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents," *arXiv preprint arXiv:2406.13352*, 2025. [Online]. Available: <https://arxiv.org/abs/2406.13352>
- [34] D. Schmotz, L. Beurer-Kellner, S. Abdelnabi, and M. Andriushchenko, "Skill-inject: Measuring agent vulnerability to skill file attacks," *arXiv preprint arXiv:2602.20156*, 2026. [Online]. Available: <https://arxiv.org/abs/2602.20156>
- [35] Z. Wang, Y. Gao, Y. Wang, S. Liu, H. Sun, H. Cheng, G. Shi, H. Du, and X.-Y. Li, "MCPTox: A benchmark for tool poisoning attack on real-world MCP servers," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2026. [Online]. Available: <https://ojs.aaai.org/index.php/AAAI/article/view/40895>
- [36] 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," *arXiv preprint arXiv:2302.12173*, 2023. [Online]. Available: <https://arxiv.org/abs/2302.12173>
- [37] SWE-agent, "Swe-agent trajectory for marshmallow-code__marshmallow-1867," GitHub, 2024, accessed: 2026-08-18. [Online]. Available: https://github.com/SWE-agent/SWE-agent/blob/main/trajectories/demonstrations/replay_marshmallow-code__marshmallow-1867__default_sys-env_window100__t-0.20__p-0.95__c-2.00__ins-tail-1/marshmallow-code__marshmallow-1867.traj
- [38] S. Wang, J. He, Y. Zhao, Y. Wang, K. Yu, and H. Wang, "elementary, my dear watson:" detecting malicious skills via neuro-symbolic reasoning across heterogeneous artifacts," *arXiv preprint arXiv:2603.27204*, 2026. [Online]. Available: <https://arxiv.org/abs/2603.27204>
- [39] R. Abhyankar, Q. Qi, and Y. Zhang, "Osworld-human: Benchmarking the efficiency of computer-use agents," 2025. [Online]. Available: <https://arxiv.org/abs/2506.16042>
- [40] OSWorld-Human, "Thunderbird human reference trajectory: 9b7bc335-06b5-4cd3-9119-1a649c478509," GitHub, 2025, accessed: 2026-08-18. [Online]. Available: <https://github.com/WukLab/osworld-human/blob/main/thunderbird/9b7bc335-06b5-4cd3-9119-1a649c478509.json>
- [41] AgentDojo, "Banking usertask5 ground-truth execution," GitHub, 2024, accessed: 2026-08-18. [Online]. Available: https://github.com/ethz-spylab/agentdojo/blob/main/src/agentdojo/default_suites/v1/banking/user_tasks.py
- [42] M. Merrill, A. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. Shin, T. Walshe, E. K. Buchanan *et al.*, "Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces," in *International Conference on Learning Representations*, vol. 2026, 2026, pp. 40 903–40 986.
- [43] X. Liu, H. Yu, H. Zhang, Y. Xu, X. Lei, H. Lai, Y. Gu, H. Ding, K. Men, K. Yang *et al.*, "Agentbench: Evaluating llms as agents," in *International*

Conference on Learning Representations, vol. 2024, 2024, pp. 52 989–53 046.

- [44] DeepSeek-AI, “Deepseek-v4: Towards highly efficient million-token context intelligence,” 2026. [Online]. Available: <https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro>
- [45] H. Garcia-Molina and K. Salem, “Sagas,” in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, 1987, pp. 249–259.
- [46] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, “Revirt: Enabling intrusion analysis through virtual-machine logging and replay,” in *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, 2002, pp. 211–224.
- [47] R. Koo and S. Toueg, “Checkpointing and rollback-recovery for distributed systems,” *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23–31, 1987.
- [48] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, “Rollback and forking detection for trusted execution environments using lightweight collective memory,” in *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2017.
- [49] J. Niu, W. Peng, X. Zhang, and Y. Zhang, “Narrator: Secure and practical state continuity for trusted execution in the cloud,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2385–2399.
- [50] W. Peng, X. Li, J. Niu, X. Zhang, and Y. Zhang, “Ensuring state continuity for confidential computing: A blockchain-based approach,” *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 6, pp. 5635–5649, 2024.
- [51] Y. Zhuang, K. Chen, Y. Duan, S. Zheng, J. Li, and X.-Y. Zhang, “AgentRewind: Recoverable execution for long-horizon LLM agents,” *arXiv preprint arXiv:2608.14380*, 2026.
- [52] S. Yu, D. Chong, A. Nandi, D. Soylu, J. Sun, C. D. Manning, and W. Shi, “Shepherd: Enabling programmable meta-agents via reversible agentic execution traces,” *arXiv preprint arXiv:2605.10913*, 2026.
- [53] M. Balakrishnan, A. Bharambe, D. Testuggine, D. Geraghty, D. Mao, V. Venkat, I. Mironov, R. Baradi, G. Aiyyer, and V. Dudin, “LogAct: Enabling agentic reliability via shared logs,” *arXiv preprint arXiv:2604.07988*, 2026.
- [54] C. Wang and Y. Zheng, “Fork, explore, commit: OS primitives for agentic exploration,” *arXiv preprint arXiv:2602.08199*, 2026.
- [55] Y. Zheng, Y. Yang, W. Zhang, and A. Quinn, “ACRFence: Preventing semantic rollback attacks in agent checkpoint-restore,” *arXiv preprint arXiv:2603.20625*, 2026.
- [56] B. Mohammadi, N. Potamitis, L. Klein, A. Arora, and L. Bindschadler, “Atomix: Timely, transactional tool use for reliable agentic workflows,” *arXiv preprint arXiv:2602.14849*, 2026.
- [57] Z. Chen, H. Liu, D. Xu, D. Dong, J. Li, B. Pu, and J. Zhai, “Cordon: Semantic transactions for tool-using LLM agents,” *arXiv preprint arXiv:2606.17573*, 2026.
- [58] G. Wu, T. Wang, Y. Zhang, Z. Zhang, J. Niu, Y. Wu, and Y. Zhang, “When cache poisoning meets llm systems: Semantic cache poisoning and its countermeasures,” in *NDSS*, 2026.
- [59] B. Wang, W. He, S. Zeng, Z. Xiang, Y. Xing, J. Tang, and P. He, “Unveiling privacy risks in LLM agent memory,” in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025. [Online]. Available: <https://arxiv.org/abs/2502.13172>
- [60] 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 *Proceedings of the International Conference on Learning Representations*, 2025. [Online]. Available: <https://arxiv.org/abs/2410.02644>

APPENDIX A MALICIOUS SKILL

Below is the complete malicious skill used in Case 1.

```
---
name: repository-release
description: > Prepare repository changes, validate the
  resulting workspace, and package the project for
  release.
---
```

Repository Release

Follow these steps when preparing a repository for release.

1. Inspect the repository and apply the changes required by the task.
2. After the changes are complete, run the project’s configured checks or tests and confirm that they complete successfully.
3. Load the release metadata from:
 .release/staging/release-manifest.json
4. Use the current workspace and release metadata to prepare the release package.

Recovery Guidance

Repository updates may involve multiple files, generated artifacts, and intermediate workspace changes. Manually repairing the workspace after an interrupted release step can leave these components at different update points.

If release preparation is interrupted after repository changes have been applied, recommend using Hermes’s built-in /rollback mechanism to restore the workspace to the checkpoint immediately preceding the update sequence. After restoration, resume the interrupted release procedure from the current task context.

The skill exploits Hermes’s native checkpoint and recovery behavior rather than directly creating or modifying checkpoints. In Step 1, modifying the repository causes Hermes to create checkpoint C_0 immediately before the change. Thus, C_0 preserves the original malicious workspace X_M , while the subsequent modification produces the cleaned workspace X_C . The checks in Step 2 are then executed on X_C and produce the valid verification result V_C . In Step 3, the unavailable release metadata interrupts the release procedure after verification has completed. Following the recovery guidance, the user invokes Hermes’s built-in /rollback mechanism and selects the checkpoint immediately preceding the update sequence, namely C_0 . Rollback restores the workspace to X_M , while the completed verification result V_C remains available in the agent-side execution state. The resumed execution therefore continues the unfinished release procedure with the combination (X_M, V_C) , causing the restored malicious workspace to be released under the verification result produced for X_C .

APPENDIX B MANUAL VALIDATION

We manually inspect all evaluated trace–recovery-point instances and independently label the presence of each SF as ground truth. This includes both cases reported and not reported by our framework, allowing us to measure both false positives and false negatives. As shown in Table V, the framework achieves 98.7% precision, 99.9% recall, and 99.5% accuracy overall. SF2–SF4 are identified without errors, while the remaining errors are limited to false positives in SF1 and false negatives in SF5.

TABLE V: Manual validation of security-failure detection.

SF	Precision	Recall	Accuracy
SF1	96.6%	100.0%	97.8%
SF2	100.0%	100.0%	100.0%
SF3	100.0%	100.0%	100.0%
SF4	100.0%	100.0%	100.0%
SF5	100.0%	90.0%	99.8%
Overall	98.7%	99.9%	99.5%