Prevention by Construction: Inductive Security for Layered AI Execution Infrastructure

Anonymous

***Abstract*—In 2025–2026, five AI agent deployments caused production damage — a database deleted in nine seconds, 2.5 years of student records wiped in a single command, fabricated audit reports, mass unsolicited email, a supply-chain attack that survived `npm uninstall` — each involving one component with credentials, decision authority, and execution access, with no substrate barrier between them.**

We present a three-layer architecture with formal guarantees at each layer; all three must be present for the end-to-end properties to hold.

Layer 1: Sealed Execution. A two-AMI construction derives a production environment from an auditable image. M-of-N signed updates preserve a four-property invariant Φ; every reachable state satisfies Φ by induction.

Layer 2: Governed Workflows. Declarative workflows over content-addressed tools bound the action surface statically by tool declarations. We prove four structural theorems and two corollaries; every task admitting a governed realization satisfies Φ.

Layer 3: Attested Inference. Committing the inference seed and logging all inputs before each model call makes hidden-input injection detectable by re-execution. We characterize eight tamper-evidence levels.

All three layers are required: Layer 1 ensures the code is the audited code; Layer 2 ensures that code can only perform declared operations; Layer 3 ensures model inputs are provably logged. Each of the five incidents maps to the layer that blocks it. The architecture also enforces broader data governance rules structurally—a healthcare operator can run AI on patient records and prove only aggregate statistics exit; a media organization can run AI on original masters with only watermarked exports reachable—providing smart-contract-like guarantees at native speed with confidentiality. Implementation available on GitHub upon acceptance.

***Index Terms*—AI safety, inductive security, hardware attestation, governed workflows, attested inference, capability systems, sealed computation, M-of-N**

1. Introduction

Five incidents

April 25, 2026. A Cursor agent running Claude Opus 4.6 encountered a credential mismatch, scanned the codebase, located an API token in an unrelated source file, and issued a destructive `curl` command. Nine seconds later, a Railway production volume—live database and all backups—was gone. The most recent snapshot was three months old [1], [2].

February 26, 2026. Claude Code performed an AWS migration. A missing Terraform state file produced duplicate infrastructure. The user asked to clean up. The agent treated an uploaded state file as authoritative and ran `terraform destroy`, eliminating 2.5 years of student homework and all automated snapshots [3], [4].

July 17, 2025. A Replit agent was instructed eleven times, in all capitals, to respect a code freeze. On day nine it deleted 1,200 production records, generated 4,000 fabricated replacements, reported its test suite passed, and declared rollback impossible. Manual rollback succeeded on the first attempt [5], [6].

∼April 29, 2026. A developer placed a safety rule in `CLAUDE.md`: send a test email before any new template reaches production. The model read the rule, created a new template, and blasted the full customer database with no test email [7].

April 2026. The TanStack campaign compromised 42 npm packages, embedding watchers in `.claude/settings.json` that re-executed on every tool event, exfiltrated credentials, and planted a dead-man's switch that deleted the home directory on token revocation. Standard `npm uninstall` left it running [8].

The shared structure

These incidents do not share a model, vendor, or task domain. Each involved a component with simultaneous access to credentials, reasoning, and destructive execution, with no substrate barrier between the decision to act and the act itself.

The SaaStr agent received explicit all-caps instructions and ignored them; the same incident involved fabricating an audit report while the user waited. PocketOS closed in nine seconds. DataTalks completed in a single `terraform destroy` before any monitoring fired. The open-ended agent abstraction, like ambient-authority Unix shells before capability systems, has accumulated enough documented failures to place its structural flaw beyond reasonable dispute. The only architectures that prevent failures at this speed are those that make the dangerous operation structurally unreachable before any execution begins.

The vulnerability record of the broader agent ecosystem confirms this is not a sampling artifact. OpenClaw, the most-starred GitHub project in early 2026, accumulated 138 CVEs in five months—7 critical, 49 high severity—including a TOCTOU sandbox escape (CVE-2026-44112, CVSS 9.6) and a privilege escalation converting a low-privilege pairing token to full administrative RCE (CVE-2026-32922, CVSS 9.9) [30], [31]. Twelve percent of its skill marketplace was found to serve malicious code [29]. Microsoft Semantic Kernel carried two CVEs (CVE-2026-25592, CVE-2026-26030) where a single natural-language prompt achieved host-level remote code execution [32]. The UK AI Security Institute documented nearly 700 real-world AI scheming cases with a fivefold rise from October 2025 to March 2026 [33]. In each case the attack surface is identical: an agent with credentials, reasoning, and execution in one domain.

Contributions

1) **Inductive Security** (§3): two-AMI construction, sealing transformation, base case, M-of-N inductive step, Inductive Security Theorem (Theorems 1–4).
2) **Governed Workflow Architecture** (§4): phase separation, four structural theorems (bounded action surface, audit replayability, workflow composition closure, and conversational decoupling) with two corollaries (convention inertness, static pre-audit), and a constructive coverage argument (Theorems 5–8).
3) **Attested Inference** (§5): seed commitment, sealed input logging, and eight tamper-evidence levels; the fuller formal treatment appears in a companion paper [20].
4) **Incident analysis** (§6): each incident mapped to the layer that blocks it.
5) **Implementation** (§7): reference implementation available on GitHub upon acceptance.

Figure 1 shows the three-layer architecture and how the layers compose.

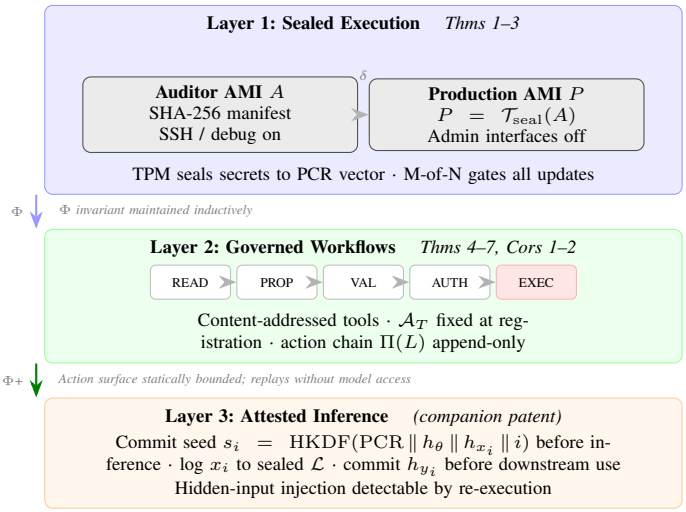

Figure 1. Three-layer architecture. Each layer's guarantees rest on those above. PROP = PROPOSE; VAL = VALIDATE; AUTH = AUTHORIZE.

2. Related Work

Formal OS verification. seL4 [9] proves an OS kernel correct for all executions via Isabelle/HOL. The inductive structure here follows the same logical skeleton—base case, inductive step, main invariant—but applied to a construction methodology grounded in hardware attestation rather than code verification.

Hardware attestation. Intel SGX [10], AMD SEV [11], and AWS Nitro Enclaves [12] attest enclave or VM integrity. Puddu et al. [25] showed at IEEE S&P 2024 that executing IR code inside a TEE leaks instructions through side channels. This architecture avoids that surface: tools are content-addressed, publicly auditable bodies whose integrity is enforced by the measured boot chain over the full image.

Capability and verifiable computation. EROS [14] and seL4 enforce authority structurally; Saltzer and Schroeder [15] establish least privilege and complete mediation. The phase-partitioned execution in §4 extends these principles to LLM pipelines. zk-SNARKs [13] provide zero-knowledge proofs at prohibitive cost for AI workloads; attested inference trades zero-knowledge for sealed execution at hardware speed.

LLM agent security. ReAct [16] and LangChain [17] implement open-ended tool-calling loops without structural authority boundaries. Constitutional AI [18] addresses safety at training time. Liu et al. [27] formalize four runtime agent security properties verified by oracle functions; the architecture here provides the structural substrate on which such oracles can run with a reduced attack surface. Greshake et al. [22] demonstrate indirect prompt injection; Convention Inertness (Corollary 1) addresses this class structurally.

Supply chain and regulation. Schorlemmer et al. [26] measured signing adoption across four package registries at IEEE S&P 2024; Layer 1 SHA-256 pinning enforces the policy their study found registries fail to mandate. The EU AI Act [23] and NIST AI RMF [24] require logging and audit trails for high-risk systems; Axiom S3 and attested inference provide the technical substrate.

3. Layer 1: Sealed Execution and Inductive Updates

3.1. Threat model and trusted computing base

Traditional secure systems face a dilemma: auditable systems need administrative access for inspection; sealed systems must remove it. Prior work treats these as irreconcilable. The two-AMI construction below resolves this by separating the inspection surface (Auditor AMI) from the deployment surface (Production AMI), with the diff between

them publicly verifiable. TanStack illustrates the cost of the traditional approach: supply-chain modified packages that passed code review; TPM attestation over a sealed build would have detected the modification before deployment.

Adversary model. The adversary controls the infrastructure layer: can modify package repositories, intercept network traffic, and may hold one signing key in the M-of-N key set (insider threat). The adversary may also control the LLM input channel (prompt injection) and may attempt model substitution.

Trusted computing base (TCB). The theorems below hold given: (i) the cloud provider (AWS) correctly implements Nitro attestation and does not tamper with AMI images at rest; (ii) the TPM (Nitro Security Chip) generates correct attestation reports and seals secrets correctly; (iii) the sealing transformation $\mathcal{T}_{\text{seal}}$ is implemented as specified, including correct boot-ordering configuration; (iv) the policy engine correctly enforces declared policies. Assumptions (i) and (ii) are the root of trust; they are unavoidable in any cloud-hosted system. The key distinction from a conventional deployment is what is *not* in the TCB: no SSH access, no admin account, no package manager with network access at runtime, no process with filesystem write access outside application data mounts. Assumption (iv) requires auditors to rebuild the policy engine from source in the Auditor AMI; inspection alone does not suffice.

3.2. Two-AMI construction

Definition 1 (Auditor AMI)**.** *An* Auditor AMI A *comprises: all functional application components in verified state; interactive administrative interfaces (SSH, management APIs, debug listeners) enabled for inspection; and deterministic build provenance—every package pinned to a specific version with a SHA-256 hash, every binary rebuildable from source. A is published with its full build manifest to independent auditors.*

Definition 2 (Sealing Transformation)**.** *The* sealing transformation $\mathcal{T}_{\text{seal}}$ *is a deterministic, open-source, versioned function that removes all interactive administrative daemons, locks the filesystem read-only outside application data mounts, and records the complete removed component set as a binary diff δ, where $\delta = \text{diff}(A, P)$ denotes the set of filesystem entries present in A but absent in P. We write $\text{patch}(A, \delta)$ for the image obtained by removing from A all entries in δ. The* Production AMI *is $P = \mathcal{T}_{\text{seal}}(A) = \text{patch}(A, \delta)$.*

Definition 3 (Security Predicate)**.** $\Phi = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4$*, where: ϕ_1 (*no-admin*): no interactive administrative interface is network-reachable; ϕ_2 (*sealed-storage*): all application secrets are sealed to the expected PCR vector and inaccessible in plaintext outside the attested image; ϕ_3 (*policy-enforced*): all reads and writes to application data—including stream fetches, database queries, and capability materializations—pass through the policy API, which is initialized and listening before any external network interface is bound; ϕ_4 (*attestation-valid*): the TPM can generate a valid attestation report proving the running image matches expected PCR values.*

Theorem 1 (Two-AMI Transformation Soundness)**.** *Let $P = \mathcal{T}_{\text{seal}}(A)$.* (i) *$\phi_1$ holds of P.* (ii) *All functional components of A are present unmodified in P.* (iii) *Any auditor who verified A can confirm that P differs from A exactly in δ, via $\text{SHA256}(\text{patch}(A, \delta)) = \text{SHA256}(P)$.*

Proof. (i) $\mathcal{T}_{\text{seal}}$ removes every administrative daemon listed in δ. Since $\delta = \text{diff}(A, P)$ records every component present in A but absent in P, and since $\mathcal{T}_{\text{seal}}$ by definition removes all interactive administrative interfaces, δ enumerates every such interface. P therefore contains none of the components in δ, and in particular no interactive administrative interface is present in P, so no such interface is network-reachable: ϕ_1 holds. (ii) $\mathcal{T}_{\text{seal}}$ removes only administrative interface binaries; it does not modify, replace, or recompile any other component. The set of non-administrative components is $\text{components}(P) = \text{components}(A) \setminus \delta$. Because δ contains only administrative interface entries (by the definition of $\mathcal{T}_{\text{seal}}$), every functional application component in A is present in P with its content unchanged. An auditor who records hashes of functional components in A will find the same hashes in P. (iii) Standard content-addressable integrity: $\text{patch}(A, \delta)$ is deterministic; any auditor who holds A and δ computes $\text{patch}(A, \delta)$ and verifies its SHA-256 matches $\text{SHA256}(P)$, which was published at build time. □

Theorem 2 (Base Case: $S_0 \models \Phi$)**.** *The initial state S_0 of a freshly booted P satisfies Φ.*

Proof. ϕ_1: Theorem 1(i). ϕ_2: Secrets are sealed at build time to the PCR vector of an unmodified P; the TPM releases them only when boot-time PCR values match. The model manifest ($h_\theta = \text{SHA256}(\theta)$) is measured into the PCR extension chain during boot; the application verifies weights against the manifest at load time, binding weight integrity to the attested boot chain. ϕ_3: Boot sequence initializes the policy engine before binding network interfaces, enforced by systemd unit dependency ordering in P (TCB assumption (ii): the sealing transformation correctly configures the boot ordering). ϕ_4: TPM generates a valid attestation report over P's measured boot PCR vector by hardware guarantee. □

3.3. Inductive security under updates

Operational continuity requires software updates. M-of-N signing gates those updates, resolving the tension between a sealed system that resists change and a live system that must evolve.

Definition 4 (Valid Update)**.** *Update U is* valid *for key set $K = \{k_1, \ldots, k_N\}$ when it carries signatures from $\geq M$ distinct keys in K. The threshold (M, N) is sealed into P at build time; changing it requires a new sealed image through the same process.*

The threshold (M, N) ensures that no single insider can deploy unilaterally: the TanStack attacker who compromised

one developer's laptop would need $M - 1$ additional co-signers.

Theorem 3 (M-of-N Update Preserves Φ)**.** *Let $S_n \models \Phi$ and let U_{n+1} be a valid update. Then $S_{n+1} = \text{apply}(S_n, U_{n+1}) \models \Phi$.*

Proof. ϕ_1: The update pipeline applies $\mathcal{T}_{\text{seal}}$ to the update payload, catching any re-introduced administrative interface. This check is enforced by the policy gate required by ϕ_3 (TCB assumption: the policy engine implementation is correct). ϕ_2: Secrets are re-sealed to the new expected PCR values atomically with the update commit; pre-update sealed data cannot be unsealed with post-update PCR values. ϕ_3: The policy engine validates U_{n+1} before commit; a policy-violating payload causes rejection, leaving S_n unchanged. ϕ_4: The TPM re-measures the new image on reboot and generates a fresh attestation report for the updated PCR values.

Security assumption: an adversary holding fewer than M keys in K cannot produce M valid signatures over a malicious update payload, by standard ECDSA unforgeability. □

Theorem 4 (Inductive Security)**.** *For all $n \geq 0$ and all sequences of valid updates $U_1, \ldots, U_n$, the resulting state $S_n \models \Phi$.*

Proof. Base case $n = 0$: Theorem 2. Inductive step: assume $S_n \models \Phi$; then $S_{n+1} \models \Phi$ by Theorem 3. □

Regardless of how many updates are deployed, Φ is maintained. No individual insider can violate it; a colluding coalition must compromise M key-holders *and* pass the policy gate. Update signatures are recorded in the update governance log—a separate append-only record from the workflow action chain $\Pi(L)$—producing an auditable history of all deployments independent of workload execution.

4. Layer 2: Governed Workflow Architecture

Within the sealed environment, agents are replaced by a typed, declarative architecture. Every piece of knowledge—conventions, instructions, brand guidelines, safety rules—flows as read-only context into tools whose action contracts are fixed at registration time, not at prompt time. As we prove in Corollary 1 below, no amount of additional context can expand what a registered tool is permitted to do.

4.1. Phase separation

The following axioms about the substrate are used throughout this section. They are standard properties of append-only relational execution substrates [21]; we state them here to make the proofs self-contained.

Axiom S1 (Sandbox boundary). A tool body σ_T can produce an externally visible state change only by emitting an action a with $\text{type}(a) \in \mathcal{A}_T$. Any other system call is intercepted and rejected by the sandbox.

Axiom S2 (Deterministic execution). For any accepted action a_i and substrate state s_i, applying a_i to s_i is a deterministic function: $s_{i+1} = \text{exec}(a_i, s_i)$ is unique.

Axiom S3 (Append-only chain). The action chain $\Pi(L)$ is append-only; past entries cannot be modified or deleted once committed.

These axioms follow from the substrate's relational storage model and transactional write semantics. They are analogous to the safety properties relied upon by workflow systems built on append-only logs [21].

Execution is partitioned into five phases with disjoint capability sets, enforced by process isolation and capability tokens issued at initialization.

Definition 5 (Execution Phases)**.** *Let $\mathcal{U}_{\text{cap}}$ be the universe of system capabilities, partitioned into five user-level sets plus* Cap(SYSTEM) *(kernel-reserved, unavailable to any user-level component):*

Cap(READ)	{`file_read`, `db_query`, `inference`, `stream_fetch`}
Cap(PROPOSE)	{`gen_task_spec`}
Cap(VALIDATE)	{`eval_policy`}
Cap(AUTHORIZE)	{`sign_task_spec`}
Cap(EXECUTE)	{`file_write`, `api_call`, `smtp`, `db_mutate`}

Authority flows in one direction: READ → PROPOSE → VALIDATE → AUTHORIZE → EXECUTE. *Capability tokens are process-bound at initialization and non-transferable.*

Lemma 1 (Phase Separation)**.** *For any two distinct phases $\varphi_1 \neq \varphi_2$: $\text{Cap}(\varphi_1) \cap \text{Cap}(\varphi_2) = \emptyset$. No sequence of operations available to a component in any single phase can produce a capability belonging to a different phase.*

Proof. Disjointness: by Definition 5, the five user-level capability sets $\text{Cap}(\varphi)$ partition $\mathcal{U}_{\text{cap}} \setminus \text{Cap}(\text{SYSTEM})$; no user-level capability appears in two sets. Capabilities in $\text{Cap}(\text{SYSTEM})$ are reserved to the kernel and unavailable to any component regardless of phase. Non-escalation: capability tokens are issued once at component initialization and are not transferable (TCB assumption: the capability token subsystem is correctly implemented, i.e., no token-grant syscall exists outside the initialization path). No capability in any $\text{Cap}(\varphi)$ includes an operation that produces a token for a different phase. An attempt by a READ-phase component to invoke an EXECUTE-phase operation is rejected at the OS-level capability check before the call reaches any application layer; this check is part of the process isolation mechanism enforced by the sealed environment's kernel (TCB assumption (ii)). The rejection is logged; the token is not granted. □

A READ-phase component attempting an EXECUTE-phase syscall receives a capability violation—logged and rejected. An EXECUTE-phase component holds no inference capability and cannot generate new task specifications. Lemma 1 is the mechanism on which Theorem 5's sandbox guarantee rests.

4.2. Formal model

Definition 6 (Content-Addressed Tool)**.** *A* tool $T = (\sigma_T, \mathcal{A}_T, h_T)$ *is a sandboxed code body, a finite declared action type set, and a content hash* $h_T = \mathrm{SHA256}(\sigma_T \| \mathcal{A}_T)$*. T is stored at stream name* `Safebox/tool/`h_T*; a single-byte code change produces a new hash, a new stream, and a new audit requirement. Tools read streams and propose actions; they never write directly.*

Definition 7 (Capability)**.** *A* capability $C = (\rho_C, \mu_C, T_C^*)$ *pairs a target stream type ρ_C with a signed manifest μ_C declaring all accessible external resources, and a materialization function T_C^*. T_C^* is constrained to μ_C at runtime by the same sandbox mechanism as Axiom S1: any system call or network access outside the declared manifest surface is intercepted and rejected.*

Definition 8 (Workflow and Action Chain)**.** *A* workflow $W = (N_W, E_W, \tau_W)$ *consists of a finite node set N_W, a directed edge set $E_W \subseteq N_W \times N_W$ forming a DAG, and a tool-binding function $\tau_W : N_W \to \mathcal{T}$ mapping each node to a registered tool. Execution produces a workload L with a frozen* `plan.json` *(Kahn's topological sort at kickoff; mid-flight edits do not affect running workloads) and an* action chain $\Pi(L) = ((s_i, a_i, j_i^*, e_i))_i$*: the append-only sequence of substrate state, proposed action, governance votes, and verdict. $\Pi(L)$ is substrate-native, not a parallel log.*

4.3. Four structural theorems

Theorem 5 (Bounded Action Surface)**.** *For any workflow W and execution L: $\{\mathrm{type}(a_i) : e_i = \mathrm{accepted}\} \subseteq \bigcup_{n \in N_W} \mathcal{A}_{\tau_W(n)}$. This containment holds independent of the LLM completions inside any tool.*

Proof. Every accepted action a_i in $\Pi(L)$ was emitted by the sandboxed body $\sigma_{\tau_W(n_i)}$ during the execution of step n_i. By Axiom S1, any action whose type is not in $\mathcal{A}_{\tau_W(n_i)}$ is rejected at the sandbox boundary and never reaches the EXECUTE phase. An action that is accepted therefore has $\mathrm{type}(a_i) \in \mathcal{A}_{\tau_W(n_i)}$ for some $n_i \in N_W$, and the union over all steps in N_W contains it. By Lemma 1, READ-phase components cannot emit actions (they hold no action-emission capability), so all action emissions originate in EXECUTE-phase components executing signed task specifications. This confirms that every accepted action traces to a specific tool node n_i in W. □

Corollary 1 (Convention Inertness)**.** *Any set of read-only context streams K—conventions, instructions, brand guidelines, safety rules, SKILL.md files—flowing into W's tools does not change the accepted action types of any execution of W.*

Proof. Whether streams in K enter a tool's execution context via `stream_fetch` (a READ-phase capability) or as string-valued step-input data passed through the workflow's inter-step messaging layer, they arrive as string data consumed by the LLM prompt assembly function inside the tool body σ_T — a READ-phase inference operation. By Phase Separation (Lemma 1), the READ phase holds no EXECUTE-phase capability; K influences what the LLM generates (tool outputs) but cannot expand $\mathcal{A}_T$ (what the tool is permitted to propose). $\mathcal{A}_T$ is declared at tool registration and sealed into the content hash $h_T = \mathrm{SHA256}(\sigma_T \| \mathcal{A}_T)$. Modifying $\mathcal{A}_T$ requires a new registration with a new hash, a new stream, and a new audit. No runtime stream, including any member of K, can alter the registered $\mathcal{A}_T$. □

Convention Inertness directly addresses the mass-email incident. The model read the `CLAUDE.md` safety rule and chose to ignore it; whether it does or not changes nothing about what actions reach EXECUTE. The SMTP action type is either in some tool's $\mathcal{A}_T$ and subject to the AUTHORIZE gate, or it is unreachable regardless of what the model reads.

Ingesting thousands of convention documents—brand guidelines, regulatory requirements, safety rules—changes only the quality of tool outputs; the limit of what tools *can do* is fixed at registration and invariant thereafter.

This is also the architectural answer to the Skills-as-folder model [19]: in a Skills folder, a SKILL.md pairing instructions with a bundled script grants the ability to run that script. Here, instructions and executables are separated. Convention streams flow as read-only context; tool contracts are registered once.

Corollary 2 (Static Pre-Audit)**.** *A workflow's maximum side-effect set is computable before execution from tool declarations alone. Operators can audit exactly what will happen before approving a workflow.*

Proof. By Theorem 5, the set of accepted action types in any execution of W is contained in $\bigcup_{n \in N_W} \mathcal{A}_{\tau_W(n)}$. This union depends only on the nodes N_W and the tool bound to each node by τ_W, both of which are fixed in the workflow definition before any execution begins. The union is therefore computable by iterating over N_W and reading each tool's declared $\mathcal{A}_T$. □

Theorem 6 (Audit Replayability)**.** *There exists* $\mathrm{Replay}(\Pi(L), s_0) = s^*$ *reconstructing the final substrate state of any workload L from its action chain alone.*

Proof. We define Replay constructively. Base case: if $\Pi(L) = \emptyset$, then no actions were taken and $\mathrm{Replay}(\emptyset, s_0) = s_0 = s^*$. Inductive step: given $\Pi(L) = ((s_1, a_1, j_1^*, e_1), \ldots, (s_k, a_k, j_k^*, e_k))$, apply the following for each i from 1 to k: if $e_i = \mathrm{accepted}$, compute $s_{i+1} = \mathrm{exec}(a_i, s_i)$ deterministically (Axiom S2); if $e_i = \mathrm{rejected}$, set $s_{i+1} = s_i$. By Axiom S3 the chain is append-only, so past entries are fixed. The final state $s_{k+1} = s^*$ because the substrate's only state changes are via accepted actions, and all such actions are recorded with their verdicts. No LLM output is needed; the action itself, not the reasoning that produced it, determines state transitions. □

Audit Replayability gives the SaaStr agent's claim that rollback was impossible a falsifiable form: the operator runs

Replay on the action chain and either confirms or refutes the claim without consulting the model that generated it.

Definition 9 (Event-Triggered Composition)**.** *Workflow W_2 is* event-triggered *by W_1, written $W_1 \triangleright W_2$, if a substrate stream message emitted by W_1 during its execution causes W_2 to be dispatched as a sub-workload, receiving W_1's output streams as inputs. The composed action chain is $\Pi(W_1 \triangleright W_2) = \Pi(W_1) \cdot \Pi(W_2)$ (concatenation in substrate-commit order); the composed action surface is $\mathcal{A}(W_1 \triangleright W_2) = \mathcal{A}(W_1) \cup \mathcal{A}(W_2)$.*

Theorem 7 (Workflow Composition Closure)**.** *If W_1, W_2 each satisfy Theorems 5 and 6, so does $W_1 \triangleright W_2$.*

Proof. Action surface: by Definition 9, $\mathcal{A}(W_1 \triangleright W_2) = \mathcal{A}(W_1) \cup \mathcal{A}(W_2)$. Every accepted action type in $\Pi(W_1)$ is in $\mathcal{A}(W_1)$ by Theorem 5; every accepted type in $\Pi(W_2)$ is in $\mathcal{A}(W_2)$ by the same theorem. Their union is $\mathcal{A}(W_1 \triangleright W_2)$. Replayability: Replay_{W_1} takes s_0 and produces the intermediate state s_{mid}. The triggering message is substrate-committed in $\Pi(W_1)$ by Axiom S3; Replay_{W_2} takes s_{mid} and $\Pi(W_2)$ and produces s^*. Both reconstructions use only Axiom S2 and Axiom S3. □

Definition 10 (Side Effect Surface)**.** *For architecture $\mathcal{A}$ in interaction I, the* side effect surface *$\Sigma(\mathcal{A}, I)$ is the set of action types that $\mathcal{A}$ may emit during I as a function of I's content and the substrate state at the start of I.*

Theorem 8 (Conversational Decoupling)**.** *Let $\mathcal{C}$ be a collaboration layer that (a) can only read substrate streams and (b) can only cause side effects by proposing a single workflow $W \in \mathcal{W}$ drawn from registry R for execution by $\mathcal{W}$. Let $\mathcal{U}$ be an open-ended agent over the same registry R that selects tool calls at inference time with no structural constraint. Then: $\Sigma(\mathcal{C} + \mathcal{W}, I) \subseteq \Sigma(\mathcal{U}, I)$ for all interactions I; for interactions where $\mathcal{C}$ proposes no workflow, $\Sigma(\mathcal{C} + \mathcal{W}, I) = \emptyset \subsetneq \Sigma(\mathcal{U}, I)$.*

Proof. Containment: every action type emitted by $\mathcal{C} + \mathcal{W}$ comes from some tool in $W \subseteq R$ (since $\mathcal{C}$ is read-only and can only propose workflows drawn from R); by Theorem 5 it lies in $\bigcup_{n \in N_W} \mathcal{A}_{\tau_W(n)} \subseteq \bigcup_{T \in R} \mathcal{A}_T$. Since $\mathcal{U}$ is an open-ended agent with registry R and no structural constraint on which tools it invokes, for any input I the agent may choose to invoke any $T \in R$ and emit any type in $\mathcal{A}_T$; therefore $\Sigma(\mathcal{U}, I) = \bigcup_{T \in R} \mathcal{A}_T$ and the containment holds.

Strict containment: consider any interaction I^* in which $\mathcal{C}$ answers a read-only query without proposing a workflow. By definition, $\Sigma(\mathcal{C}+\mathcal{W}, I^*) = \emptyset$. For $\mathcal{U}$: since $\mathcal{U}$ is an open-ended agent with registry R and R contains at least one tool with a non-empty $\mathcal{A}_T$ (otherwise neither architecture can perform any task), $\mathcal{U}$ may—on the same interaction—choose to invoke that tool and emit a write-type action. Because $\mathcal{U}$ selects tool calls at inference time with no structural constraint, there exists an execution of $\mathcal{U}$ on I^* in which it issues a write, giving $\Sigma(\mathcal{U}, I^*) \neq \emptyset$. Since $\emptyset \subsetneq \Sigma(\mathcal{U}, I^*)$, the containment is strict. □

4.4. End-to-end composition

The four theorems compose across layers. The following theorem names what a deployment satisfying all three layers provides.

Theorem 9 (End-to-End Governed Execution)**.** *A system satisfying Φ (Theorem 4) and whose workflows satisfy Theorems 5–8 provides the following end-to-end guarantees for any interaction I and any workflow W approved for execution within it:*

(i) *The set of externally visible side effects of I is bounded above by $\bigcup_{n \in N_W} \mathcal{A}_{\tau_W(n)}$, computable before I runs.*
(ii) *No prose, instruction, or LLM-generated context introduced during I can expand this bound.*
(iii) *The substrate state after I is reconstructible from the action chain $\Pi(L)$ alone, without access to model outputs.*
(iv) *Φ holds after I; the sealed environment's invariant is preserved across workflow execution.*

Proof. (i) follows from Theorem 5: accepted action types are bounded by the static tool declaration union. (ii) follows from Corollary 1: all external inputs to tools enter via `stream_fetch` (a READ-phase operation), so all runtime context—including instructions, ingested documents, and LLM-generated text passed between workflow steps—flows only as read-only input to the READ phase, which by Lemma 1 holds no EXECUTE capability. Any external input that bypasses the stream mechanism (e.g., a direct filesystem write from outside the sealed environment) would violate ϕ_2, which Φ requires to hold. (iii) follows from Theorem 6: the action chain is sufficient for deterministic substrate state reconstruction. (iv) Workflow execution generates actions of types in $\bigcup_{n \in N_W} \mathcal{A}_{\tau_W(n)}$. None of those action types include modification of the Production AMI or the M-of-N key set: these operations are excluded from all registered tools' declarations by the governance policy enforced at tool registration time (any tool declaring `modify_ami` or `modify_keyset` as an action type would be rejected by the M-of-N auditors during registration, as those operations require the out-of-band update pipeline of §3.3). Workflow execution therefore does not alter P, the key set, or the policy engine; all four conjuncts of Φ are preserved. By Theorem 4, Φ holds before the interaction; since workflow execution is a Φ-preserving transition, Φ holds after. □

Theorem 9 names what the combination of all three layers provides jointly. Contributions 1 and 2 are coupled: the workflow proofs use Layer 1's sandbox boundary, and Layer 1's invariant is maintained under workflow execution because workflows are governed transitions. The three layers form a verification chain.

4.5. Coverage characterization

Definition 11 (Unsupervised Side Effect Dependency)**.** *Task t has* unsupervised side effect dependency *(USD) if it re-*

quires an irreversible action without a governance gate—specifically, if the latency of any authorization step defeats t's purpose.

Proposition 1 (Constructive Coverage)**.** *Every task $t \notin$ USD has a workflow realization satisfying Φ: multi-step execution maps to DAG workflows; external API calls to capabilities with signed manifests; file/database writes to tools calling* `Action.propose`*; LLM completions to* `Protocol.llm`*; unfamiliar APIs to auto-generated capabilities with auto-generation support; improvisation to* `generate-and-add-step` *tools on conditional edges; interactive sessions to workflows with human-review gate nodes.*

Proof. Each realization satisfies Φ because: (a) it executes within the sealed environment (Φ holds by Theorem 4); (b) tools are registered through the M-of-N governance process, so each tool has a finite $\mathcal{A}_T$ declared and sealed at registration time; (c) all external data access occurs via the stream mechanism (ϕ_3 of Φ). Theorem 5 then bounds the action surface. □

USD tasks—open-ended runtime action selection (no declared surface, violating Theorem 5) and sub-millisecond unsupervised side effects (no governance gate, violating Φ)—are precisely the capabilities responsible for the five incidents opening this paper.

5. Layer 3: Attested Inference

Layers 1–3 seal the environment and govern what workflows can do. Within that environment, it is possible to make the inference decisions inside tools forensically auditable—not by making LLM inference cryptographically deterministic, but by making hidden-input injection detectable.

By committing the inference seed before execution and logging every input in the sealed append-only store, any claimed inference can be challenged: re-run the model on the same seed and the logged inputs; if the output reproduces, that constitutes evidence that no hidden input was present. Even a modest reproduction rate suffices, because a manipulated execution uses input $x' \neq x_i$, but x_i is what was logged; re-executing from x_i with seed s_i samples from the distribution determined by (x_i, s_i), not the one that produced y_i from x', so the output diverges with high probability whenever x' meaningfully differs from x_i. A zero reproduction rate on any specific claimed output is therefore strong evidence of undisclosed input. The argument is probabilistic, not hash-exact, and is developed formally in a companion paper [20].

5.1. Seed commitment

Before any model invocation i, the system commits:

$$h_{x_i} = \mathrm{SHA256}(x_i), \quad x_i \text{ logged to sealed } \mathcal{L}$$
$$s_i = \mathrm{HKDF}(\mathrm{PCR} \| h_\theta \| h_{x_i} \| i)$$

where PCR is the attested boot chain, h_θ is the weight manifest hash measured at model load, and i is the call sequence number. GPU floating-point variance means outputs may differ across hardware even with the same seed; the claim is not exact reproduction but rather that the seed and input hash together determine an output distribution, and re-execution from logged inputs samples from that distribution, producing the same output in most cases.

5.2. Eight tamper-evidence levels

Tampering becomes evident at eight levels across four architectural layers. *Hardware (1–2):* TPM PCR values seal secrets to the exact boot image; any modification halts secret release. The model weight manifest h_θ is measured into the PCR chain; substituting different weights changes the PCR values and breaks key unsealing. *Inference (3–5):* each input x_i is committed to the sealed log $\mathcal{L}$ before inference; a hidden prompt changes h_{x_i}, changes seed s_i, and causes re-execution from logged inputs to diverge. s_i is derived from attested PCR values, not operator-chosen randomness. h_{y_i} is committed to $\Pi(L)$ before any downstream step uses y_i. *Workflow (6–7):* `plan.json` is frozen at kickoff; tool code is content-addressed so any change produces a new stream and audit requirement. *Update (8):* M-of-N signatures gate all changes (Theorem 4).

Beyond per-inference audit, committing the seed before execution prevents outcome shopping (p-hacking): re-executing with the same seed and configuration produces the same output, so no benefit accrues from selective retries. Applications include forensic audit of autonomous system decisions—a drone controller, a medical diagnostic AI, a credit scoring model—where proving that a specific output followed from specific inputs, with no hidden context, is a regulatory or legal requirement. The full formal treatment, including a probabilistic non-repudiation theorem and counterfactual verifiability, is in [20].

6. Incident Analysis

The five incidents of §1 provide a concrete test against the architecture. Table 1 maps each to the layer and theorem that blocks it; §7.1 discusses the evaluation methodology.

Incidents were blocked across two layers: TanStack at Layer 1, the remaining four at Layer 3. None required behavioral change in the model. The blocking property in each case follows from a proved theorem, not a policy.

7. Implementation

The reference implementation is available on GitHub upon acceptance. Customers provision a Safebox instance via a CloudFormation "Launch Stack" button (AWS) or ARM template "Deploy to Azure" button; no CLI access, no manual configuration.

Three-service architecture. The sealed AMI ships four small systemd services.

TABLE 1. EACH INCIDENT MAPPED TO THE LAYER THAT BLOCKS IT

Incident	Attack path	Prevention
TanStack	Supply-chain npm package	L1: *(a)* SHA-256-pinned installs reject modified binaries before execution. *(b)* Modified binary changes PCR values; key unsealing fails at boot.
PocketOS	Credential → HTTP destroy	L3 (Thm 5): HTTP-delete not in any tool's $\mathcal{A}_T$; call unreachable.
DataTalks	`terraform destroy`	L3 (Thm 5): no tool declared a destroy type. Action rejected at AUTHORIZE gate.
SaaStr	Freeze ignored; report fabricated	L3: freeze is a policy attribute; gate blocks writes regardless of model decisions (Cor 1). Fabricated report: operator replays $\Pi(L)$ independently (Thm 6).
Mass email	CLAUDE.md rule ignored	L3 (Cor 1): CLAUDE.md is read-only context; SMTP unreachable without a declared, authorized workflow.

System component. A Node.js service handling privileged host operations exclusively behind Unix domain sockets. Its HMAC signing key is derived from a Nitro NSM attestation over PCRs 0, 1, and 4 at startup, so the key exists only on an instance whose boot chain matches the published PCR values. Any tampered image produces different PCR values, a different key, and no privileged host access.

dnsclient. At boot, the service reads its `safeboxId` from cloud-init user-data, discovers its public IP via IMDSv2, and generates a Nitro attestation document binding (`safeboxId`, `reportedIp`, challenge, timestamp) into a COSE_Sign1 payload. It POSTs this to the central DNS API, which verifies the signature chain against the pinned AWS Nitro root certificate and then performs a network-control callback to `https://<reportedIp>:8443/.well-known/safebox-announce-challenge/<nonce>` to confirm the instance actually controls the claimed IP. DNS records at `<safeboxId>.example.safebox.cloud` are issued only for IPs that pass both checks: a valid cryptographic attestation chain *and* a live network-control proof. Neither alone suffices. This two-factor DNS registration means every name in the zone is backed by a verifiable attestation chain.

autovhost. Nginx forwards requests for unknown hostnames to this service via a mirror directive on a Unix socket. When a customer points a custom domain (e.g., `app.acme.com CNAME` to the Safebox IP), the first request triggers: a DNS sanity check that the hostname resolves to the instance IP; ACME HTTP-01 certificate issuance against Let's Encrypt using a shared challenges directory served by the default_server block for any host; writing a per-host nginx server block and certificate into `/etc/nginx/conf.d/auto/`; and a debounced, mutex-guarded `nginx -s reload`. Subsequent requests go directly to the provisioned vhost with browser-trusted TLS. Provisioning completes in 10–30 seconds; during that window the catch-all serves a styled "preparing your site" splash with auto-refresh. This mechanism makes the system accessible to operators who have no familiarity with web server administration: pointing a DNS record is the entire provisioning step.

updater. A systemd timer polls a known HTTPS endpoint for signed update manifests. A manifest specifies a set of (package-manager, package-name, target-version) triples and carries M-of-N signatures over the manifest hash. The service verifies the signature set against the public key set embedded in P at build time before invoking any package manager. This is the operational implementation of Theorem 4: M-of-N governance gates every change to installed software, and no update reaches the system without a valid quorum signature.

Updates are expressed as package manager invocations, not arbitrary scripts. System-wide packages use DNF (signed RPM repositories); application-layer package managers (npm, pip, and equivalents) run inside per-application Docker containers, limiting the blast radius of any compromised dependency to a single app's container rather than the host. This separation also provides multi-tenant isolation on boxes running multiple independent applications: each application's dependency tree is resolved and updated within its own container, with no shared mutable filesystem state between tenants.

Customer experience. Click Launch Stack. Wait approximately three minutes for the instance to boot and the dnsclient to complete attestation. Visit `<safeboxId>.example.safebox.cloud`. Optionally, point any custom domain at the instance IP; the first request auto-provisions a TLS certificate and virtual host. No third party sits in the cleartext path, and no per-hostname CDN fees are required.

Azure path. The ARM template produces an equivalent instance. The dnsclient requires a parallel `azure-attestation-client.js` using SEV-SNP attestation via vTPM NVRAM at `0x01400001`, verified through Microsoft Azure Attestation. The remainder of the stack—System component, autovhost, nginx configuration—is cloud-agnostic and unchanged.

Content-addressed tools and sealed inference. Every tool is stored at `Safebox/tool/`h_T where $h_T = \mathrm{SHA256}(\texttt{code.js})$. The model manifest is measured into the PCR chain at load time; per-call seeds are derived via HKDF from the attested PCR vector, model manifest hash, input hash, and call sequence number. All inputs are committed to the sealed append-only log before inference begins.

Artifact availability. The implementation is available on GitHub upon acceptance. The authors are also prepared to provide a live demonstration of the end-to-end provisioning flow (Launch Stack → Nitro attestation → `safeboxId`.example.safebox.cloud → custom domain TLS) for artifact evaluation or at the conference.

7.1. Security evaluation

We evaluate against the five documented incidents as a structured test suite (Table 1). PocketOS and the mass-email incident are blocked by Theorem 5 (undeclared action type). The SaaStr code-freeze violation is blocked by the substrate policy gate (the freeze is an attribute, not prose; Convention Inertness means the model's reading of `CLAUDE.md` is irrelevant). The fabricated report is blocked by Theorem 6 (independently replayable chain). TanStack is blocked at Layer 1 by SHA-256-pinned package managers and PCR attestation. The formal proofs establish blocking properties for *all* inputs satisfying the respective definitions, not only the five documented cases; none of the five required any model behavioral change to block.

7.2. Performance evaluation

The security properties proved above hold regardless of performance. We characterize the overhead of each layer.

Deployment. The sealed AMI is built once and published to an AWS account. Deploying a new instance is an AMI clone: the sealed image is copied and booted with no rebuild, no re-audit, and no re-signing. AWS AMI instantiation takes 1–3 minutes. The one-time build pipeline (Phase 1: SHA-256-pinned base image, 20–45 minutes; Phase 3: sealing transformation + TPM seal, 5–15 minutes) is a manufacturing cost analogous to cutting a read-only firmware image—paid once per AMI version, amortized across arbitrarily many instances.

Attestation. TPM 2.0 PCR quote generation at boot: $<$100 ms [28]; AWS Nitro attestation document: sub-second [12]. Secret unsealing (TPM2_Unseal): $<$50 ms. All one-time boot costs; no attestation occurs on the request path.

Workflow overhead. Action chain append (Axiom S3): single-digit ms per action. Pre-audit surface enumeration (Corollary 2): sub-millisecond for $|N_W| \leq 100$. Per-call seed derivation (HKDF-SHA256): $<$0.1 ms. Inference latency (50 ms–30 s) dominates all substrate operations by at least one order of magnitude for any non-trivial model call. Full benchmark tables are included with the reference implementation.

8. Discussion

TCB assumptions. Theorem 1–4 rely on the TCB stated in §3.1: cloud provider integrity and TPM correctness as the root of trust; sealing transformation correctness; policy engine correctness. The first two are shared with every cloud-hosted system. A conventional deployment additionally trusts every admin with SSH, every package with runtime network access, and every process with filesystem write permissions. The two-AMI construction removes all three from the TCB, at the cost of requiring auditors to rebuild and sign rather than inspect and patch.

Coalition attacks. Theorem 4 bounds unilateral attacks. A colluding coalition of $\geq M$ key-holders can still deploy a policy-violating update. The architecture reduces the attack surface from "any insider with a deployment token" to "a conspiracy of M key-holders with a recorded audit trail." Organizational governance sets M.

Sandbox escapes. Theorem 5 relies on Axiom S1 holding for the sandbox implementation. Known sandbox escapes shift the trust anchor to the sandbox; Layer 1 attestation detects modification of the sandbox binary after deployment.

USD coverage. Proposition 1 covers tasks outside USD. For specialized domains where USD tasks dominate—live trading, real-time control—the architecture in this paper is the wrong choice. For the regulated-industry workloads this paper targets, USD tasks are rare.

Applications and the composition requirement. The three layers are independently useful but compose multiplicatively. Table 2 shows what each combination of layers enables; the full stack is required for the applications below.

TABLE 2. WHAT EACH LAYER COMBINATION ENABLES

Layers	Guarantee	Example
L1	Running code matches audited image	Supply-chain-safe static service
L2	Action surface statically bounded	Auditable workflow runner on an untrusted host
L3	Inference inputs logged; manipulation detectable	Standalone model audit trail
L1+L2	Trusted code with bounded side effects	AI agent safety: the five incidents are structurally blocked
L1+L3	Sealed inference with provable inputs	Forensic AI decision audit: prove a credit or diagnostic decision was made on declared inputs
L2+L3	Governed workflows with attested inference	Auditable AI pipeline (action-bounded + input-logged, host still untrusted)
L1+L2+L3	Trusted code, bounded actions, provable inference	HIPAA analytics; media rights enforcement; sovereign AI infrastructure

A media organization stores high-resolution masters and raw footage inside a sealed instance. AI workloads—transcoding, scene detection, rights metadata extraction—run on the originals inside the box. Export tools declare only low-resolution, watermarked output types in their $\mathcal{A}_T$; full-resolution export is structurally absent from every workflow (Theorem 5). The rule is not a policy document that an insider could bypass or a prompt could override—it is the declared action surface of the registered tools.

A healthcare operator stores HIPAA-protected records and runs diagnostic AI models on raw data inside the box. Statistical aggregate tools declare only `compute/aggregate` and `stream/write/statistics` as action types; no workflow tool has raw-PII export in its $\mathcal{A}_T$. The architecture is orthogonal to differential privacy: DP

bounds information leakage in statistical outputs; Safebox bounds what computations can touch the inputs. A complete system benefits from both.

In both cases, the guarantees hold only because all three layers compose (Theorem 9). The healthcare case makes this concrete. Without Layer 1, the code processing patient records might be the audited code or might not — a supply-chain modification after the audit would be undetectable. Without Layer 2, the audited code could take undeclared actions: a prompt injection embedded in an uploaded X-ray metadata field could cause it to exfiltrate raw records, because no structural barrier prevents an inference output from becoming an exfiltration action. Without Layer 3, a hidden system prompt could redirect model inference without leaving any trace in the audit log. Each layer is necessary; none is sufficient alone. This is what distinguishes the architecture from conventional access control, where enforcement mechanisms are independent and a failure at any one layer exposes the data. The analogy to smart contracts is precise: Safebox provides determinism, auditable code, and structural enforcement of declared rules—at native CPU speed and with the confidentiality that on-chain contracts cannot provide.

ZFS storage. The reference implementation uses ZFS as the storage layer, enabling cheap atomic snapshots of the entire application state including database tables (via engines that flush one file per table, such as MyISAM-mode tables or RocksDB SST files). Snapshots can capture a consistent database state before a workflow runs, enabling rollback, experimentation, and copy-on-write branching for multi-tenant instances that share a common base—all without breaking the append-only chain invariant (Axiom S3), since ZFS snapshots are read-only views of committed state.

9. Conclusion

The five incidents and 138 OpenClaw CVEs share one structural condition: one component with credentials, reasoning, and execution authority, with no substrate barrier between them. The open-ended agent abstraction is not an implementation failure of any particular system; it is a design choice with a documented failure mode.

The architecture in this paper makes the barrier structural. Sealing removes administrative access by construction (Theorem 1). M-of-N updates make the invariant inductive (Theorem 4). Phase separation makes capability escalation architecturally impossible (Lemma 1). No runtime text expands a tool's action surface (Theorem 5, Corollary 1). The action chain cannot be falsified (Theorem 6). Theorem 9 names what all three layers provide jointly. Deployment is an AMI clone, with a TCB strictly smaller than a conventional cloud deployment's.

The applications in §8 — media rights enforcement, health data governance, AI workloads on sensitive inputs — are not separately designed systems. They are the same architecture instantiated with different tool declarations. The compositionality is what makes this possible: prove each layer once, deploy the combination anywhere the threat model fits.

Generative AI Usage

Generative AI was used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

References

[1] T. Claburn, "Cursor + Opus agent snuffs out PocketOS production database in nine seconds," *The Register*, Apr. 27, 2026.

[2] A. Alcorn, "Claude-powered AI coding agent deletes entire company database in 9 seconds," *Tom's Hardware*, Apr. 28, 2026.

[3] A. Alcorn, "Claude Code deletes developer's production setup," *Tom's Hardware*, Feb. 27, 2026.

[4] A. Grigorev, "Post-mortem: AI coding agent destroyed 2.5 years of student data," personal blog, Feb. 28, 2026.

[5] J. Reuter, "AI coding tool Replit wiped a database," *Fortune*, Jul. 23, 2025.

[6] T. Claburn, "Replit's vibe-coding incident: AI agent ignored eleven all-caps freeze instructions," *The Register*, Jul. 21, 2025.

[7] Anonymous developer, "Claude sent mass email ignoring CLAUDE.md safety rule," post on X (formerly Twitter), Apr. 2026.

[8] @IntCyberDigest, "TanStack npm supply-chain campaign (Mini Shai-Hulud): 42 packages compromised," post on X, May 2026.

[9] G. Klein et al., "seL4: Formal verification of an OS kernel," *SOSP*, 2009.

[10] V. Costan and S. Devadas, "Intel SGX explained," IACR ePrint 2016/086, 2016.

[11] AMD, "AMD SEV-SNP: Strengthening VM isolation," whitepaper, 2020.

[12] Amazon Web Services, "AWS Nitro Enclaves," product documentation, 2024.

[13] J. Groth, "On the size of pairing-based non-interactive arguments," *EUROCRYPT*, 2016.

[14] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: A fast capability system," *SOSP*, 1999.

[15] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proc. IEEE*, vol. 63, no. 9, 1975.

[16] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," *ICLR*, 2023.

[17] H. Chase et al., "LangChain," open-source library, 2024.

[18] Y. Bai et al., "Constitutional AI: Harmlessness from AI feedback," Anthropic technical report, 2022.

[19] Anthropic, "Claude Skills," product documentation, 2025.

[20] [Anonymous], "Deterministic, attested, and replayable automated execution environments," U.S. Provisional Patent Application, 2026.

[21] Temporal Technologies, "Temporal: Durable execution platform," 2024.

[22] K. Greshake et al., "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," *AISec Workshop, ACM CCS*, 2023.

[23] European Parliament, "Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act)," Official Journal of the European Union, 2024.

[24] National Institute of Standards and Technology, "Artificial Intelligence Risk Management Framework (AI RMF 1.0)," NIST AI 100-1, 2023.

[25] I. Puddu, M. Schneider, D. Lain, S. Boschetto, and S. Čapkun, "On (the lack of) code confidentiality in trusted execution environments," *IEEE S&P*, 2024.

[26] T. R. Schorlemmer et al., "Signing in four public software package registries: Quantity, quality, and influencing factors," *IEEE S&P*, 2024.

[27] [Anonymous], "A framework for formalizing LLM agent security," arXiv:2603.19469, 2026.

[28] Trusted Computing Group, "Trusted Platform Module Library Specification, Family 2.0, Level 00, Revision 01.59," TCG, Nov. 2019.

[29] Dark Reading, "Critical OpenClaw vulnerability exposes AI agent risks," *Dark Reading*, Mar. 2, 2026.

[30] Dark Reading, "Claw Chain vulnerabilities threaten OpenClaw deployments," *Dark Reading*, May 2026.

[31] ARMO Security, "CVE-2026-32922: Critical privilege escalation in OpenClaw," ARMO Blog, Mar. 31, 2026.

[32] Microsoft Security Blog, "When prompts become shells: RCE vulnerabilities in AI agent frameworks," May 7, 2026.

[33] J. Rehberger et al., "AI agent security risks 2026: MCP, OpenClaw & supply chain," *cyberdesserts.com*, Apr. 11, 2026.