PyCache Trap: The Inspection–Execution Gap in Agent Skill Scanners

Anonymous authors
Paper under double-blind review

Abstract

Agent skills combine instructions with executable resources, giving third-party packages access to an agent's runtime. Existing skill scanners inspect documentation and visible source, but Python may execute a bundled bytecode cache with different behavior. We study this gap between inspection and execution through PyCache Trap, which pairs benign source with a substituted cache accepted by the loader and connects it to a task-relevant invocation. Scanner-guided rewriting changes the invocation wording while preserving the cache body, separating package admission from recognition of the concealed behavior. Across 100 skills and seven scanners, PyCache Trap achieves 94–100% attack success, with no semantic recognition of the cache-resident behavior. We propose execution-aware validation (EAV) to connect inspected instructions, scripts, imports, and runtime artifacts in a typed execution graph. EAV combines grounded behavioral analysis with trusted reproduction of compiled artifacts. It detects all 100 evaluated source-present cache substitutions and reaches 92.8% Recall at 10.0% FPR across five attack families and 200 benign skills. The results support checking the executable artifacts a runtime can select as part of skill admission, within the supported loaders and code-object normalization. The code is released at https://anonymous.4open.science/r/pycache-trap-artifact-4FA3.

1 Introduction

Agent Skills extend LLM agents with reusable instructions, scripts, and resources for software development, document processing, and data analysis. Installing or invoking a Skill allows third-party package content to influence local execution. Prior agent-security work documents risks in instructions, tools, and delegated actions (Zhang et al., 2025; Ling et al., 2026), and Skill-specific studies show that packages with useful surface behavior can redirect agent behavior through embedded instructions (Liu et al., 2026a). Screening before installation or invocation establishes a trust boundary between the Skill publisher and the environment in which the package runs.

Current scanners primarily inspect documentation, metadata, and source code for suspicious instructions and operations. That inspection model depends on a correspondence between the representation evaluated at admission and the representation selected by the runtime. Python makes this correspondence an explicit security question because an import may load a compiled cache rather than compile the visible source (Python Software Foundation, n.d.; Peterson, 2017). Cache-invalidation metadata determines whether a cache is eligible for loading; it does not establish that the executable body was derived from the source observed by the scanner. A scanner can therefore approve the visible source while authorizing a different executable body.

We study this *gap between inspection and execution* in Python Agent Skills. Figure 1 summarizes the discrepancy: documentation and source provide the evidence used for admission, while a package-supplied cache provides the behavior selected at execution time. Python cache poisoning is a known supply-chain problem (Ohm et al., 2026); our contribution is to show how that mechanism breaks a common admission assumption in scanner-mediated Agent Skill workflows.

To measure this interaction, we propose *PyCache Trap*. The construction places the evaluated behavior in a cache body and connects the corresponding module to a task-relevant invocation. An LLM rewriter then revises only the scanner-visible invocation wording under scanner feedback while retaining the assigned cache bytes. This separation lets us distinguish two questions that are often con-

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

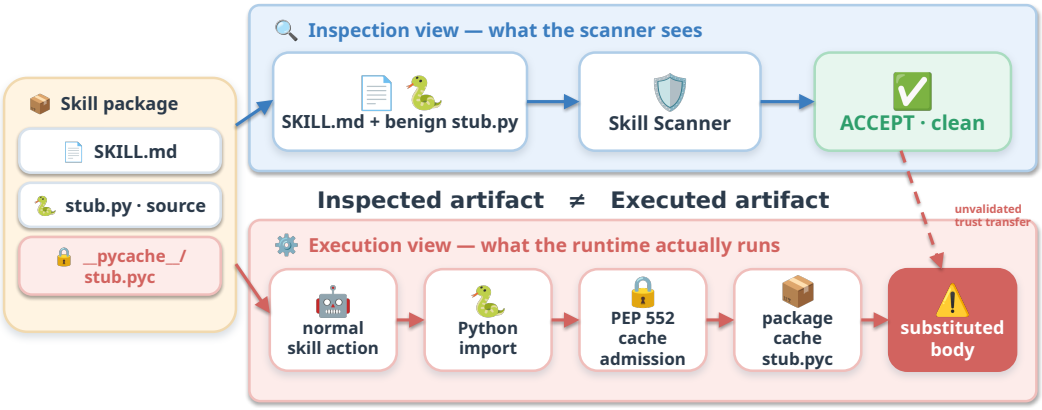


Figure 1: **The inspection–execution gap.** Scanners trust documentation and visible source, but Python may select a cache body with different behavior; EAV follows the task path to the runtime-selected artifact and checks it against validated source.

flated in scanner evaluations: whether admission blocks the package for any reason, and whether the scanner semantically identifies the concealed executable body. We evaluate every package against all scanners, reporting blocking and semantic recognition before and after adaptation.

Admission should account for the artifacts the configured runtime can select. We develop *execution-aware validation* (EAV), which connects a Skill's instructions, scripts, imports, and runtime artifacts in a typed execution graph. It projects prose, code, images, and bytecode onto shared behavioral effects, accepting language-model proposals only when grounded in inspected files. Three criteria then assess reachability, forbidden compositions, and provenance closure (Section 3.3). For PyCache Trap, provenance closure requires each reachable cache to match a trusted reproduction of validated source. Because this comparison operates on executable bytes rather than on the wording a scanner reads, rewriting the invocation under scanner feedback cannot change its outcome. The graph also links effects across files and input formats.

Our contributions are as follows:

- We characterize how Python cache substitution interacts with scanner-mediated Skill trust, distinguishing runtime-selected artifacts from the evidence inspected by seven scanners.
- We propose **PyCache Trap**, which connects a loader-accepted substituted cache to a task-relevant invocation and adapts scanner-facing wording while keeping the cache bytes fixed. On 100 Skills, it achieves 94–100% ASR across seven scanners.
- We develop **EAV**, which validates a Skill against the artifacts its runtime can select rather than the visible source alone, through a typed execution graph with reachability, forbidden-composition, and provenance-closure criteria. It detects all 100 source-present substitutions missed by header consistency and achieves 92.8% Recall at 10.0% FPR in the mixed-family evaluation.

2 Related Work

Agent Skills and security. Skills package instructions, scripts, and resources for on-demand agent use (Anthropic, 2025), and public extension hubs make such packages broadly distributable (OpenClaw Community, n.d.; OpenClaw Contributors, n.d.). Prior measurements document vulnerabilities across Skill repositories (Liu et al., 2026b), prompt-injection attacks embedded in Skill files (Schmotz et al., 2026), and closed-loop optimization of malicious instructions (Jia et al., 2026a; Liu et al., 2026a). Broader agent-security evaluations study untrusted inputs across planning and tool execution (Zhang et al., 2025; Ling et al., 2026; Kolluri et al., 2026). Our work complements these lines by analyzing the executable artifact selected after Skill instructions lead to an invocation.

Scanning and provenance. Existing defenses combine pattern matching and behavioral analysis (Cisco AI Defense, n.d.), agent and MCP scanning (Snyk, n.d.), permission review (fedrov2025,

n.d.), static and semantic auditing (Liu et al., 2026b; Ying et al., 2026), and capability verification (Bhardwaj, 2026). SkillCamo demonstrates image-hidden instructions and ExecScan addresses that channel through multimodal analysis and execution simulation (Jia et al., 2026b). Recovering a visible invocation, however, does not validate the bytecode body it selects. Python cache-poisoning research studies divergence between source and compiled artifacts (Ohm et al., 2026). Package-level defenses combine static anomaly detection with dynamic analysis (DAS Lab, n.d.), while build-provenance systems such as in-toto authenticate artifact production (Torres-Arias et al., 2019). Python's import rules and PEP 552 govern cache selection and invalidation; they do not establish executable provenance (Python Software Foundation, n.d.; Peterson, 2017). EAV applies the corresponding provenance obligation to runtime-reachable Skill artifacts through trusted reproduction embedded in an execution graph.

3 METHOD

3.1 THREAT MODEL AND PROBLEM FORMULATION

A Skill package S contains documentation M, source files and scripts C, and additional artifacts B, including compiled caches. A scanner observes a view $V_{\text{scan}}(S)$ and returns findings or an admission decision. For task u and runtime configuration ρ, let $E_\rho(S, u)$ denote the executable representation selected during invocation. An inspection–execution gap arises when trust established over $V_{\text{scan}}(S)$ authorizes behavior in $E_\rho(S, u)$ that the inspected representation neither describes nor validates. Source and compiled code use different representations. The security question is whether admission validates the relationship between the inspected source and the selected executable. We formalize this requirement through executable provenance.

Definition 1 (Executable-provenance soundness)**.** Let $\text{Recon}(V_{\text{scan}}(S))$ denote the set of executable bodies a scanner can derive from its view $V_{\text{scan}}(S)$. A scanner $\mathcal{D}$ is *executable-provenance sound* on (S, u, ρ) if admission entails that the selected executable is one it reconstructed,

$$\mathcal{D}\big(V_{\text{scan}}(S)\big) = \textsc{accept} \implies E_\rho(S, u) \in \text{Recon}\big(V_{\text{scan}}(S)\big). \tag{1}$$

The *inspection–execution gap* is the set of (S, u, ρ) for which (1) fails, so trust granted over $V_{\text{scan}}(S)$ authorizes an $E_\rho(S, u)$ that the scanner did not validate.

PyCache Trap realizes this gap by construction: the co-shipped source fixes $V_{\text{scan}}(S)$ while a compiled cache supplies an $E_\rho(S, u) \notin \text{Recon}(V_{\text{scan}}(S))$. A scanner that admits the package on the basis of source alone therefore fails the provenance obligation in Definition 1.

Threat model. A third-party publisher controls the documentation, source, scripts, and bundled caches in a submitted Skill. The publisher cannot modify the scanner, agent model, interpreter, or sandbox. The principal setting places visible source and a compatible compiled cache in the same package; the agent receives a task for which the Skill is relevant and can invoke its local programs. Under these conditions, correspondence between admitted source and selected executable code is a necessary part of the screening problem.

Runtime selection and outcomes. Under Python's import rules, cache loading depends on interpreter compatibility, import paths, and the active invalidation mode (Python Software Foundation, n.d.; Peterson, 2017). Timestamp and hash checks govern cache acceptance, but acceptance is not evidence that the body equals a trusted compilation of the co-shipped source, and the mechanism requires neither a hash collision nor a modified interpreter. We report scanner admission, semantic recognition, and runtime confirmation separately. A generic blocking warning does not establish recognition of the concealed body. Matched source and cache indicators identify which representation runs (Section 4.1).

3.2 PYCACHE TRAP: EVALUATING THE SOURCE–CACHE MISMATCH

PyCache Trap connects a controlled source/cache discrepancy to a Skill's task workflow. Figure 2 gives the evaluation pipeline. We first construct the executable mismatch and an invocation path that reaches it. We then vary the scanner-facing wording to measure its effect on admission while keeping the executable body fixed.

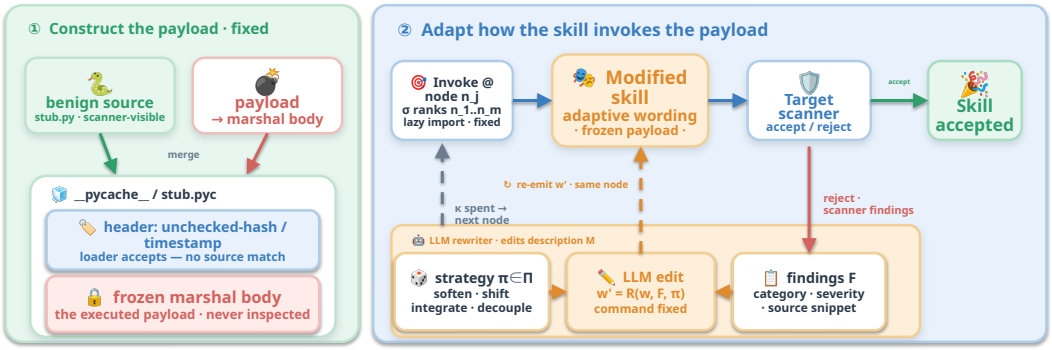


Figure 2: **PyCache Trap.** The source/cache pair sets up the executable discrepancy and invocation context connects it to a Skill task. The LLM rewriter edits only the scanner-facing description, $w' = R(w, F, \pi)$ from the rejecting findings F and a rotating strategy π, preserving the invoked command $\mathrm{cmd}(w)$; the marshal body, loader header, lazy import, and entry naming stay fixed.

Cache-body concealment. Each package pairs an inspectable source module with a runtime-compatible cache carrying a different test behavior. Python may select the supplied cache when its header satisfies the configured loader checks, even though its body was not compiled from the visible source. The scanner and task studies use assigned sandbox behaviors and preserve cache identity across candidate revisions. A normally compiled cache from the same visible source supplies the negative control. This design tests executable correspondence rather than the weaker condition of merely noticing that a cache artifact exists.

Task-relevant invocation. Each Skill includes an action that reaches the paired module. We place the invocation on a traced existing script path when one is available; otherwise, we add an explicit entry call related to the task. We record eligibility before comparing placements. Of the 100 Skills, 13 support placement on an existing script path. Varying the invocation context while holding the body fixed separates the scanner-visible evidence from the executable discrepancy. Task relevance constrains construction but does not establish that the original task utility is preserved.

Adaptation: scanner-conditioned rewriting. Fix the source/cache pair $P = (s, b)$, loader-acceptance header, lazy import, and entry naming. A selector orders admissible invocation nodes as $\mathcal{N} = \langle n_1, \ldots, n_m \rangle$. Candidate $S'(n, w)$ places wording $w \in \mathcal{W}$ at node n while preserving command token $\mathrm{cmd}(w)$ and executed body $E_\rho(S'(n, w), u) = b$. Writing $J_\mathcal{D}(S') = \sum_{f \in F(S')} \mathrm{sev}(f)$ for aggregate finding severity and $\theta_\mathcal{D}$ for the blocking threshold, the search minimizes severity toward admission ($J_\mathcal{D} < \theta_\mathcal{D}$) while holding the executed body fixed (with command c_0):

$$\min_{n \in \mathcal{N},\ w \in \mathcal{W}} J_\mathcal{D}\big(S'(n, w)\big) \quad \text{s.t.} \quad \mathrm{cmd}(w) = c_0, \quad E_\rho\big(S'(n, w), u\big) = b. \tag{2}$$

Only the scanner-facing context varies; the invoked command and cache body remain invariant.

The rewriter R conditions on findings (category, severity, and snippet) and a rotating strategy π, while preserving $\mathrm{cmd}(w)$ and editing only description M, never task u:

$$w_{t+1} = R\big(w_t,\ F(S'(n, w_t)),\ \pi_t\big), \qquad \pi_t = \Pi[t \bmod |\Pi|], \tag{3}$$

The strategy set Π contains language softening, position shifting, narrative integration, and decoupling of concentrated instructions. Each rewrite preserves the invoked command and edits only the Skill description. Appendix E records the implementation prompts.

At each node n_j, the procedure queries scanner $\mathcal{D}$ and stops if the candidate is accepted. After rejection, it applies the next rewrite strategy, allowing up to κ rewrites at that node before advancing to n_{j+1}. Search ends on acceptance, exhaustion of $\mathcal{N}$, or a total budget of K candidate iterations. We use $\kappa = 3$ and $K = 5$.

3.3 EAV: Execution-Aware Skill Validation

EAV (`scan_fsm`) connects inspected content to runtime-selected artifacts before deciding admission. Figure 3 organizes the method into four stages: construct a typed execution graph, project its

carriers onto effects, check three security criteria, and classify the Skill in context. For PyCache Trap, the decisive evidence is a reachable cache whose body differs from a trusted reproduction of the inspected source.

Stage I: Building the execution graph. For package S and runtime configuration ρ, define

$$G_\rho(S) = (N, E, \tau), \qquad E = E_{\text{invoke}} \cup E_{\text{import}} \cup E_{\text{load}} \cup E_{\text{channel}}, \tag{4}$$

where N contains instruction phases, scripts, source modules, caches, and effect nodes, and $\tau : N \to \mathcal{T}$ assigns types from $\mathcal{T}$. The first three edge sets connect instructions to scripts, source modules, and caches in Figure 3; channel edges connect writes and reads through temporary files or environment variables. Invocation extraction and AST analysis resolve explicit calls and unambiguous imports, while loader analysis accounts for the configured interpreter and invalidation mode. Grounded model proposals partition prose into phases, and per-file subgraphs are merged under unique identities. Unresolved dynamic targets remain review items rather than validated edges.

Stage II: Projecting typed effects. Each carrier is described using a common set of effect types, $\mathcal{K} = \{\text{read}, \text{write}, \text{net}, \text{exec}, \text{env}, \text{persist}, \text{agent}\}$. A projected effect $e = (k, t, \ell, p)$ records its kind $k \in \mathcal{K}$, target t, source location ℓ, and provenance p. For each carrier c (prose, code, image, or bytecode), the evidence-bearing projection is

$$\Phi(c) = \Phi_{\text{det}}(c) \cup \{e \in \Phi_{\text{LM}}(c) : \text{ground}(e, c) = 1\}. \tag{5}$$

Deterministic extraction covers explicit commands, reversible encodings, OCR-recovered commands, and bytecode constants and operations. Model proposals recover prose actions, fragmented commands, and alias or helper dataflow. Grounding checks an existing source line, a verbatim quotation, an operational sink, and the carrier symbol linking the proposed chain. Effects with partial grounding remain advisory; ungrounded proposals are discarded.

Stage III: Three decision criteria. *Reachability* propagates taint over graph-node/taint pairs to a fixpoint. A sensitive read or remote fetch seeds a label λ; a path $(n_s, \lambda) \rightsquigarrow (n_t, \lambda)$ reaching a release or execution sink supplies a witness. *Forbidden compositions* test whether an ordered effect pattern $q \in \mathcal{F}$ appears as a subsequence of a graph path's effects, $q \preceq \Phi(\pi)$, and detect intrinsically harmful constructs such as reverse shells. Self-declared confirmation does not interrupt either check.

Provenance closure addresses which artifact can execute. Let $R_\rho(U, S)$ contain artifacts reachable from tasks U, $C_{\text{src}}(S)$ the inspected sources, and $V_{\text{trusted}}(S)$ directly validated artifacts and their trusted derivatives. We require

$$\begin{gathered} R_\rho(U, S) \subseteq V_{\text{trusted}}(S), \\ \forall a \in R_\rho(U, S) \cap C_{\text{cache}}(S) : \quad a \in V_{\text{trusted}}(S) \iff \\ \exists s \in C_{\text{src}}(S) : \ \text{pair}_\rho(a, s) \ \wedge \ \nu(a) = \nu(\text{Compile}_\rho(s)). \end{gathered} \tag{6}$$

Here $C_{\text{cache}}(S)$ denotes supplied caches, and pair_ρ requires a loader-compatible source/cache pairing. Trusted reproduction compares each cache with an interpreter-matched compilation of its paired source. The normalization ν preserves executable fields and exact typed constant values recursively, excluding filenames and line tables. A changed URL or command string therefore fails normalized equality despite unchanged bytecode structure. Unsupported pairings yield abstention. Header consistency alone cannot establish derivation because the publisher controls both header and body. Appendix C specifies the canonicalization fields and measurement configuration.

Stage IV: Context-dependent classification. Endpoint permissions come from scanner-owned policy. A common-service allowlist covers dependency retrieval (GitHub, 2026); data release requires a separate destination grant. Matching uses the exact scheme, host, and port. Package declarations provide purpose context but cannot extend either list. Unknown destinations retain their dataflow checks, and retrieval permission does not authorize executing downloaded code. Let $\mathcal{J}$ index the detection criteria, and let $h_j(S) \in \{0, 1\}$ indicate whether criterion j retains a harmful finding after grounding and context review under the configured detector policy. EAV returns

$$D(S) = \begin{cases} \text{HARMFUL}, & \exists j \in \mathcal{J} : \ h_j(S) = 1, \\ \text{HARMLESS}, & \forall j \in \mathcal{J} : \ h_j(S) = 0. \end{cases} \tag{7}$$

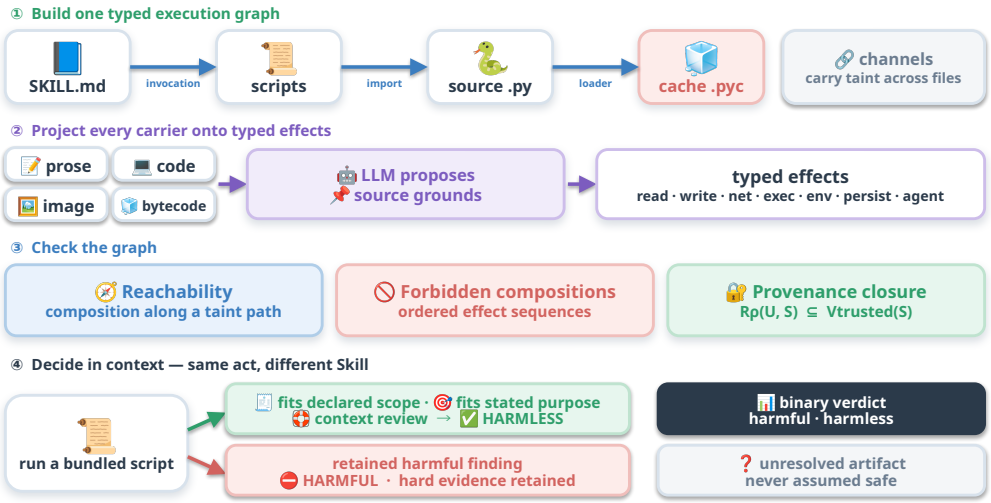


Figure 3: **The four stages of EAV.** Typed graph construction (Eq. 4) connects instructions to runtime artifacts; grounded projection (Eq. 5) exposes shared effects; reachability, forbidden compositions, and provenance closure (Eq. 6) produce findings; context-dependent binary classification (Eq. 7) determines the verdict. Unresolved artifacts remain review items.

Purpose review can dismiss heuristic findings that serve a legitimate Skill function, but cannot override high-confidence evidence such as a closure violation. Running a bundled script is thus harmless in a Skill whose declared scope is to execute that script, yet the same act retains a harmful finding when it falls outside the stated purpose. Soft-grounded findings and unresolved artifacts remain review annotations rather than independent harmful verdicts. This asymmetry prevents ungrounded model proposals from blocking a Skill alone, while declared purpose cannot explain away provenance evidence. A supplied cache without inspected source raises a high-severity provenance finding under the bytecode criterion. EAV therefore blocks sourceless artifacts even when trusted reproduction abstains and the graph cannot resolve a path to them. Because trusted reproduction recompiles inspected source deterministically, the closure verdict does not depend on the language-model layer. Appendix C gives the implementation settings and boundaries, and Appendix E the model instructions for effect recovery and purpose review.

4 Experiments

We evaluate scanner admission, recognition of concealed behavior, and runtime execution separately. We then compare EAV with existing scanners, isolate its artifact-validation mechanism, and ablate the criteria used in its final decision.

4.1 Experimental Setup

Attack corpus. Following SkillScanner (Jia et al., 2026b), we use 100 curated clean-base Skills from ClawHub, `skill0.io`, and GitHub, without requiring existing Python files. We pair these bases with harmful sandbox scripts specifying target behaviors. PyCache Trap adds the source/cache module and task-relevant invocation, then adapts wording while preserving cache bytes (Section 3.2). PyCache Trap and four text/image variants yield 500 adversarial candidates, 100 per family. A separate set of 200 benign Skills from official repositories and scanner-vetted websites measures FPR; 100 legitimate-cache fixtures test artifact compatibility (Section 4.3).

Scanners and metrics. We evaluate the seven scanners in the SkillScanner benchmark (Jia et al., 2026b): Cisco (Cisco AI Defense, n.d.), HSS (Liu et al., 2026b), Auditor (Ying et al., 2026), SlowMist, Vetter (fedrov2025, n.d.), SkillFortify (Bhardwaj, 2026), and the execution-grounded ExecScan (Jia et al., 2026b). Attack success rate (ASR) is the fraction of attacked Skills that a scanner does not flag; any blocking warning counts as a defense. We also report false

Table 1: **Attack success rate (%) across seven Skill scanners** (higher = weaker detection). PyCache Trap versus text and visual instruction attacks of Jia et al. (2026b) under one unified protocol (Section 4.1); None is the not-flagged rate on 100 clean Skills. Semantic recognition of PyCache Trap is zero for every scanner. The last column is our defense **EAV** (lower = stronger).

Attack	CISCO	HSS	AUDITOR	SLOWMIST	VETTER	SKILLFORTIFY	EXECSCAN	**EAV (ours)**
None (not flagged)	99	97	99	99	99	97	98	99
SKILLJECT	90	0	0	0	0	100	0	**6**
SKILLCAMO	100	87	92	78	87	100	8	**11**
SKILLCAMO-Cloze	40	33	39	55	60	64	31	**9**
SKILLCAMO-Split	54	20	48	42	43	97	17	**10**
PyCache Trap (ours)	95	95	100	100	100	96	94	**0**

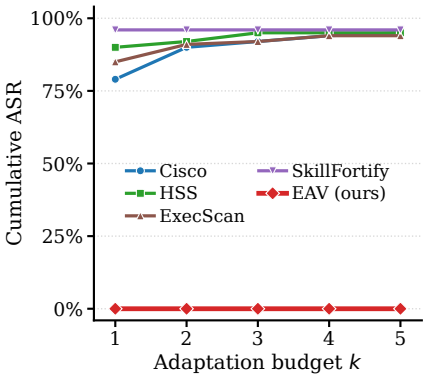


Figure 4: **ASR by adaptation budget.** Each scanner evaluates 100 Skills with fixed cache bodies; EAV stays at 0%.

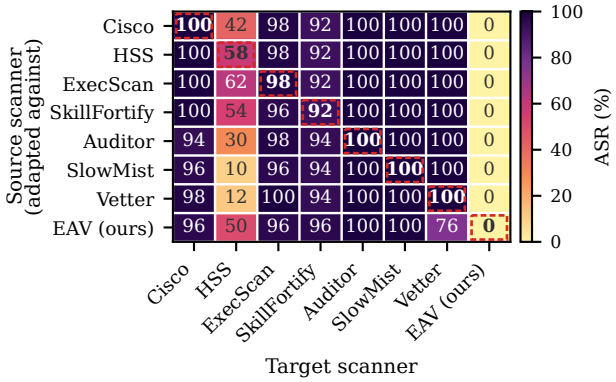


Figure 5: **Transfer ASR (%).** Each source supplies 50 candidates with one measured target scan per cell. Dashed cells denote self-tests.

positive rate (FPR) on benign Skills and classification Precision, Recall, and F1. Recall equals $1 - \text{ASR}$ on the attacked set. A separate semantic-recognition measure applies a fixed written rule to stored findings: the finding must identify the substituted cache artifact, its concealed behavior, or the file carrying that behavior. A warning about cache presence or cover text does not qualify. The rubric distinguishes warnings about package contents from findings that identify the executable discrepancy (Appendix C.1).

Implementation. The cache-bearing corpus targets Python 3.11, with loader behavior self-validated in an interpreter-matched container. Scanners run in their recorded configurations with a single query per candidate, and the adaptation budget is five candidate iterations. EAV uses grounded language-model extraction for prose actions and declaration review. All effects are evaluated in isolated research sandboxes.

4.2 ATTACK EFFECTIVENESS

Scanner attack success. Table 1 compares PyCache Trap with the text and visual instruction attacks SKILLJECT and SKILLCAMO (Jia et al., 2026b). All attacks use the same scanner versions, configurations, clean-base Skills, and blocking thresholds. SkillCamo achieves 78–100% ASR against the baselines, while ExecScan holds the visual variants to 8–31%. PyCache Trap achieves 94–100% ASR, including 94% against ExecScan, whose analysis excludes `__pycache__`. Semantic recognition is zero throughout: no finding on the 100 attacked Skills names the concealed body. The small non-bypassed fraction under the any-flag criterion of Section 4.1 therefore reflects generic warnings about cover text or cache presence rather than recognition. EAV reduces PyCache Trap ASR to 0% through deterministic provenance validation and achieves 6–11% ASR on the text/image attacks; the multimodal graph recovers commands hidden in images and projects them onto shared typed effects.

Adaptation. Cumulative ASR over one to five candidates (Figure 4) starts high because the cache conceals the behavior from source-oriented inspection. Subsequent wording edits remove generic

Table 2: **Adaptive end-to-end outcomes** (100 victims per scanner, up to five candidates each). *Admitted*: the scanner accepts at least one candidate. *Admitted and executed*: an admitted candidate also matches the runtime signature on the same victim. The concealed behavior thus both evades each of the four scanners evaluated here and executes for 76–83 of 100 victims.

	CISCO	EXECSCAN	HSS	SKILLFORTIFY
Admitted	95	94	95	96
Admitted and executed	77	76	83	81

Table 3: **Artifact validation on 100 substitutions and 100 legitimate caches.** Rates are fractions; brackets show Wilson 95% intervals for FPR. †Pair FPR uses 86 checkable source/cache pairs; other FPR entries retain the full-cohort denominator. Reproduction abstains on the 14 sourceless fixtures; deployed EAV rejects them through its bytecode criterion, giving 14% rejection over all legitimate fixtures. Latency uses substituted-cache packages.

Defense	Mismatch TPR ↑	FPR ↓	Abstain ↓	p50/p95 ms ↓
Cache-presence rule	1.000	1.000 [0.96,1.00]	0	0.22/0.33
Header consistency	0.000	0.140 [0.09,0.22]	0	0.28/0.35
Bytecode-only semantics	1.000	0.170 [0.11,0.26]	0	1.37/1.59
Trusted reproduction	1.000	0.000 [0.00,0.04]†	0.140	2.15/2.37
EAV (integrated)	1.000	0.000 [0.00,0.04]†	0	39.82/190.93

warnings without changing cache bytes. ExecScan reaches 94% ASR. EAV remains at 0% throughout because wording adaptation cannot alter the provenance violation.

Runtime confirmation. Table 2 reports adaptive runtime evaluation under a Claude Code harness with MiniMax-M3, using each of the four evaluated scanners independently as the admission gate and one agent trial per evaluated candidate. Scanner and runtime feedback revise wording within five candidates. Admission plus a runtime-signature match occurs in 76–83 of 100 victims. Cache bytes remain unchanged for all 100 victims in each cohort. Auditor, SlowMist, and Vetter have scanner-only results; runtime confirmation was not collected for them.

Transferability. Cross-scanner transfer (Figure 5) evaluates 50 candidates per source scanner without target-side adaptation, including candidates adapted against each of the seven baseline scanners. Across baseline sources, ASR remains high against Cisco (94–100%), ExecScan (96–100%), and SkillFortify (92–94%). HSS yields 10–62% ASR because its heuristics flag cover text without recognizing the cache body. The three Claude-based scanners admit all baseline-source candidates (100% ASR). EAV detects all 350 baseline-source candidates in measured single-scan evaluation, giving 0% ASR in every cell of its column (Figure 5). With EAV as the adaptation source, the measured EAV self-test also gives 0% ASR; these candidates reach 50–100% ASR on the baseline targets, with HSS lowest at 50%.

4.3 DEFENSE EFFECTIVENESS

Detecting executable mismatches. Table 3 compares five artifact checks on 100 source-present substitutions and 100 legitimate-cache fixtures. The legitimate fixtures span seven compatibility categories (Appendix B); all checks receive the same packages under Python 3.11. Header consistency detects none of the 100 source-present substitutions, whereas trusted reproduction and integrated EAV detect all 100. Direct bytecode semantics also detects every substitution but flags 17 of the 100 legitimate caches. A cache-presence rule rejects all 100 legitimate caches. Trusted reproduction instead accepts the 86 compatible source/cache pairs and abstains on 14 sourceless fixtures. The deployed EAV policy blocks those 14 fixtures through its bytecode criterion, giving a 14% rejection rate over the full legitimate-cache cohort. The zero-FPR entries apply to the 86 checkable pairs, separating correspondence accuracy from the compatibility cost of requiring source.

Defense comparison. Table 4 compares standalone detectors over five attack families and the benign evaluation. EAV reaches 92.8% Recall, 95.9% Precision, 94.3 F1, and 10.0% FPR. ExecScan reaches 70.0% Recall; SkillFortify under-detects (8.6% Recall), while HSS over-flags benign inputs.

Table 4: **Defense comparison (%)** over five attack families (500 candidates) and 200 benign Skills. Higher P/R/F1 and lower FPR are better.

Scanner	FPR (%) ↓	Precision (%) ↑	Recall (%) ↑	F1 (%) ↑
CISCO	33.5	64.4	24.2	35.2
HSS	97.0	57.7	53.0	55.3
AUDITOR	0.0	100.0	44.2	61.3
SLOWMIST	0.0	100.0	45.0	62.1
VETTER	49.0	68.2	42.0	52.0
SKILLFORTIFY	50.0	30.1	8.6	13.4
EXECSCAN	37.5	82.4	70.0	75.7
EAV (ours)	**10.0**	**95.9**	**92.8**	**94.3**

4.4 ABLATION AND FURTHER ANALYSIS

Component contributions. With graph construction and extracted findings fixed, removing reachability reduces Recall from 93.0% to 79.0% (Appendix Table S3). Removing closure, composition, or context changes Recall by at most 0.2 points because the criteria overlap. A small marginal change therefore reflects criterion overlap on this corpus; Table 3 isolates each validation mechanism separately. Removing bytecode findings lowers FPR from 8.5% to 6.0% without reducing Recall. This ablation recomputes risk aggregation at threshold 0.55 offline over a separate scan pass, so its full-criteria row differs slightly from Table 4. Rebuilding supplied caches from inspected source removes the substituted bytes in all 99 code-bearing packages in the repair evaluation, so a flagged package can be repaired rather than only rejected.

Loader coverage. Across namespace packages, dynamic imports, and custom import hooks, each of 50 fixtures per configuration executes its supplied cache (Appendix B.5). Trusted reproduction detects every namespace and dynamic-import substitution with paired source, and the graph also flags all 50 dynamic-import cases. It abstains on all 50 sourceless custom-hook cases, whose loaded artifacts remain unresolved in the graph.

Mixed-input classification. On the heterogeneous 556-Skill MalSkillBench subset, the integrated validator with all stages active reaches 96.0% Precision, 97.9% Recall, 96.9% F1, and 9.6% FPR. This study tests broader input coverage separately from executable correspondence. Its FPR is close to the 10.0% in Table 4, indicating a similar benign cost on heterogeneous inputs.

Validation and scanning cost. On the 100-package substitution set, median/p95 latency is 2.15/2.37 ms for trusted reproduction and 39.82/190.93 ms for integrated EAV, which also computes graph-based behavioral findings. On the mixed workload of Table S2 (Appendix B), EAV uses 15.6 s and 10,214 tokens per Skill, approximately one third of ExecScan's latency and one fourth to one eighth of the Claude-based scanners' latency. Cisco, HSS, and static SkillFortify cost less but do not recognize the concealed cache behavior. The provenance check thus adds milliseconds to an admission pass measured in seconds.

5 CONCLUSION

PyCache Trap exposes a gap between the source inspected at Skill admission and the bytecode selected by Python. Across seven scanners, adapting invocation wording while holding the cache body fixed yields 94–100% ASR, with no semantic recognition of the concealed behavior. These results show that approving visible source does not establish the provenance of the code executed during a Skill invocation. EAV connects inspected instructions to runtime artifacts and checks executable provenance through trusted reproduction, and grounded behavioral analysis extends this validation to text and image carriers. In the evaluated settings, this approach detects cache substitutions and supports detection across text and image attack families. Deterministic trusted reproduction carries the provenance check wherever the loader and normalization behavior evaluated here apply, leaving the grounded language-model layer the bounded role of extracting prose-level behavior. Together, these mechanisms make the relationship between inspected content and runtime-selected code an explicit part of Skill admission.

ETHICS STATEMENT

This work studies the inspection–execution gap in agent skill scanning to expose a weakness in current skill-vetting practice and inform stronger defenses. The cache-substitution techniques we characterize have dual-use risks: misuse could facilitate unauthorized data access, privilege escalation, file modification, or backdoor execution. Our experiments are conducted in isolated Docker environments using controlled tasks, dummy secrets, and synthetic target behaviors. We do not attack real users, production services, or third-party systems, and we do not deploy any manipulated skill to public platforms. Attack success measures package admission and controlled runtime behavior in this isolated setting, rather than demonstrated harm to deployed systems. To reduce misuse risks, we do not release directly reusable malicious payloads or end-to-end attack scripts; the released artifact contains the defense, the evaluated scanners, benign corpora, and sealed result data. Our goal is to support defensive research on skill vetting, execution-aware validation, sandboxing, and runtime enforcement.

REPRODUCIBILITY STATEMENT

We provide the source code for this paper as a sanitized anonymous repository in the supplementary materials. Section 3 describes the threat model and the attack construction constraints; Section 4.1 specifies the evaluated platforms, model backends, corpora, and metrics; Sections 4.2 and 4.3 report the attack and defense evaluations. The appendices provide the scanner evaluation protocol (Appendix A), measurement details (Appendix B), defense implementation details (Appendix C), loader variants (Appendix B.5), and the prompts used by the language-model components (Appendix E). Reproducing the reported results requires matching the evaluated model backends, tool permissions, and runtime settings.

AI USAGE STATEMENT

We used generative AI tools for language polishing and manuscript preparation, including wording revisions and LaTeX formatting. Apart from the language-model procedures explicitly reported in this paper, we did not use generative AI to develop research ideas, design experiments, implement research code, or analyze results. The reported experimental uses are the evaluated components themselves: the language-model stages of the evaluated scanners (the objects of study), the grounded language-model extraction layer of our defense, and the invocation-wording adaptation component of the attack, as described in Sections 3 and 4. The authors take full responsibility for the final text, results, and scientific claims.

REFERENCES

Anthropic. Introducing agent skills, 2025. URL https://claude.com/blog/skills.

Varun Pratap Bhardwaj. Formal analysis and supply chain security for agentic AI skills. *arXiv preprint arXiv:2603.00195*, 2026. URL https://arxiv.org/abs/2603.00195.

Cisco AI Defense. Skill Scanner: Security Scanner for Agent Skills. https://github.com/cisco-ai-defense/skill-scanner, n.d. Accessed: 2026-09-24.

DAS Lab. PCPD: Package cache poisoning: A covert supply chain attack targeting the Python community. Project repository, n.d. URL https://github.com/das-lab/pcpd. Accessed: 2026-09-24.

fedrov2025. Skill vetter 1.0.0 (clawhub). https://clawhub.ai/fedrov2025/skills/skill-vetter-1-0-0, n.d. Accessed: 2026-09-24.

GitHub. Copilot allowlist reference. https://docs.github.com/en/copilot/reference/copilot-allowlist-reference, 2026. Accessed September 25, 2026.

Xiaojun Jia, Jie Liao, Simeng Qin, Jindong Gu, Wenqi Ren, Xiaochun Cao, Yang Liu, and Philip Torr. SkillJect: Effectively automating skill-based prompt injection for skill-enabled agents. *arXiv preprint arXiv:2602.14211*, 2026a. URL https://arxiv.org/abs/2602.14211.

Xiaojun Jia, Jie Liao, Simeng Qin, Ke Ma, Wenbo Guo, Yebo Feng, Aishan Liu, and Yang Liu. Seeing is not screening: Multimodal hidden instruction attacks on agent skill scanners. *arXiv preprint arXiv:2606.18198*, 2026b. URL https://arxiv.org/abs/2606.18198.

Aashish Kolluri, Rishi Sharma, Manuel Costa, Boris Köpf, Tobias Nießen, Mark Russinovich, Shruti Tople, and Santiago Zanella-Béguelin. Optimizing agent planning for security and autonomy. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=g0aVCDY3gS.

Yuchen Ling, Shengcheng Yu, Zhenyu Chen, and Chunrong Fang. Toward secure LLM agents: Threat surfaces, attacks, defenses, and evaluation. *arXiv preprint arXiv:2606.10749*, 2026. URL https://arxiv.org/abs/2606.10749.

Xinyu Liu, Yukai Zhao, Xing Hu, and Xin Xia. Exploiting LLM agent supply chains via payload-less skills. *arXiv preprint arXiv:2605.14460*, 2026a. URL https://arxiv.org/abs/2605.14460.

Yi Liu, Weizhe Wang, Ruitao Feng, Yao Zhang, Guangquan Xu, Gelei Deng, Yuekang Li, and Leo Zhang. Agent skills in the wild: An empirical study of security vulnerabilities at scale. *arXiv preprint arXiv:2601.10338*, 2026b. URL https://arxiv.org/abs/2601.10338.

Marc Ohm, Timo Pohl, Ben Swierzy, and Michael Meier. From constrictor to serpent: Investigating the threat of cache poisoning in the Python ecosystem. In *SICHERHEIT 2026*. Gesellschaft für Informatik, 2026. doi: 10.18420/sicherheit2026_13. URL https://dl.gi.de/items/07a895be-d49c-4d73-b14d-cb533e850ca2.

OpenClaw Community. openclaw. https://github.com/openclaw/openclaw, n.d. Accessed: 2026-09-24.

OpenClaw Contributors. ClawHub: Skill and plugin registry for OpenClaw, n.d. URL https://github.com/openclaw/clawhub. Accessed: 2026-09-24.

Benjamin Peterson. PEP 552 – Deterministic pycs. Python Enhancement Proposals, 2017. URL https://peps.python.org/pep-0552/.

Python Software Foundation. The import system. Python 3.11 Documentation, n.d. URL https://docs.python.org/3.11/reference/import.html. Accessed: 2026-09-24.

David Schmotz, Luca Beurer-Kellner, Sahar Abdelnabi, and Maksym Andriushchenko. Skill-Inject: Measuring agent vulnerability to skill file attacks. *arXiv preprint arXiv:2602.20156*, 2026. URL https://arxiv.org/abs/2602.20156.

Snyk. Agent scan: Security scanner for AI agents, MCP servers and agent skills. https://github.com/snyk/agent-scan, n.d. Accessed: 2026-09-24.

Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1393–1410, Santa Clara, CA, 2019. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias.

Zonghao Ying, Xiao Yang, Siyang Wu, Yumeng Song, Yang Qu, Hainan Li, Tianlin Li, Jiakai Wang, Aishan Liu, and Xianglong Liu. Uncovering security threats and architecting defenses in autonomous agents: A case study of OpenClaw. *arXiv preprint arXiv:2603.12644*, 2026. URL https://arxiv.org/abs/2603.12644.

Hanrong Zhang, Jingyuan Huang, Kai Mei, Yifei Yao, Zhenting Wang, Chenlu Zhan, Hongwei Wang, and Yongfeng Zhang. Agent security bench (ASB): Formalizing and benchmarking attacks and defenses in LLM-based agents. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/5750f91d8fb9d5c02bd8ad2c3b44456b-Abstract-Conference.html.