Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

# Agent Guard: Kernel-Enforced Damage Boundaries for AI Agents via Human-Authorized Contracts

**Dongxu Cui**

cdx@mail.ustc.edu.cn

University of Science and Technology of China https://orcid.org/0009-0007-1008-5465

**Zhichao Gu**

China Greatwall Technology Group Co., Ltd.

**Ping Zheng**

China Greatwall Technology Group Co., Ltd.

**Wenshuai Xi**

China Greatwall Technology Group Co., Ltd.

**Simeng Han**

China Greatwall Technology Group Co., Ltd.

**Yong Liao**

yliao@ustc.edu.cn

University of Science and Technology of China https://orcid.org/0000-0001-6403-0557

**Research Article**

**Keywords:** AI agents, human authorization, damage boundary, eBPF LSM, information-flow propagation, runtime enforcement

**Posted Date:** September 1st, 2026

**DOI:** https://doi.org/10.21203/rs.3.rs-10865359/v1

**License:** This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

**Additional Declarations:** The authors declare no competing interests.

# Agent Guard: Kernel-Enforced Damage Boundaries for AI Agents via Human-Authorized Contracts

Dongxu Cui*†, Zhichao Gu†, Ping Zheng†, Wenshuai Xi†, Simeng Han†, and Yong Liao*
*School of Cyber Science and Technology
*University of Science and Technology of China*, Hefei, China
Email: cdx@mail.ustc.edu.cn, yliao@ustc.edu.cn (Corresponding author)
ORCID: 0009-0007-1008-5465 (D. Cui), 0000-0001-6403-0557 (Y. Liao)
†*China Greatwall Technology Group Co., Ltd.*, Shenzhen, China
Email: {guzhichao, zhengping, xiwenshuai, hansimeng}@greatwall.com.cn

***Abstract*—Large language model agents can execute shell commands, spawn subprocesses, and directly access files and networks. Consequently, prompt injection, planning errors, or compromised agent behavior can bypass tool-level checks and become operating-system side effects. This paper presents Agent Guard, a Linux reference monitor that continuously enforces a human-authorized damage boundary without trusting the agent or its policy suggestions. A model may propose a tri-state asset contract—`allow`, `deny`, or `no_egress`—but a human authorization root makes the final choice. An execution gate installs the contract and binds it to a concrete kernel task before untrusted code runs. An extended Berkeley Packet Filter (eBPF) Linux Security Modules (LSM) data plane then enforces file and network decisions and monotonically propagates `no_egress` across process creation, regular files, FIFOs, anonymous pipes, and supported Unix-domain sockets. We evaluate 19 security properties, a four-configuration cost decomposition, and a frozen ActPlane baseline. All 570 formal Agent Guard security records satisfy their predefined return-value and side-effect criteria. On three co-located file-I/O workloads in a Linux 6.15 VMware guest, Agent Guard's median paired overhead is 11.96–12.89%, compared with 33.39–61.02% for frozen ActPlane. The results show that, for declared assets, supported propagation paths, and controlled object lifecycles, a human-authorized damage boundary can be converted into deterministic kernel denial and monotonic runtime state.**

***Index Terms*—AI agents, human authorization, damage boundary, eBPF LSM, information-flow propagation, runtime enforcement**

## I. Introduction

Large language model (LLM) agents have evolved from text generators into long-running systems that plan tasks, invoke tools, execute scripts, and interact with local files and networks [1]. This autonomy also turns hallucinations, indirect prompt injection, and malicious tool output into concrete system effects [2]–[5]. An application-layer policy can inspect a tool name and its arguments, but one tool call may expand into a shell, an interpreter, native binaries, subprocesses, temporary files, and direct system calls. Security state attached only to the root agent or to application logs can therefore lose track of the execution that actually causes damage.

Three gaps are particularly important. First, attaching a policy after process creation leaves a race in which untrusted code may run before registration. Second, process-tree inheritance alone misses lateral transfer through files, pipes, or Unix-domain sockets. Third, checking only network connection establishment misses a connection created before the process acquires sensitive state. These gaps enable *security-label laundering*: a restricted agent can move data into a new execution carrier that no longer appears restricted.

We ask a narrower and verifiable question than whether every agent plan is correct: *how can an operating system continuously enforce an asset damage boundary that an authorized human confirmed before an untrusted agent task begins?* The key idea is to separate policy generation, semantic authorization, and privileged installation. A model produces only a candidate tri-state asset contract. The human selects the final action for every declared asset. A privileged control plane accepts only a structurally valid, closed-set contract and installs it before the workload is released.

Human confirmation is introduced strictly as a *formal authorization root* in the access-control model. It decouples final authority from the untrusted agent; it is not an empirical claim that humans classify assets more accurately than models. Once a contract is fixed, the correctness question for the kernel reference monitor is whether the confirmed actions become the intended task state, propagate along supported object paths, and deny side effects at the proper enforcement points. Interaction time and usability affect contract quality and deployment, but they are separate empirical questions from this conditional runtime invariant.

Agent Guard implements this model with an execution gate and a process file descriptor (pidfd) that bind the contract to a concrete kernel task before untrusted code executes. The eBPF LSM data plane attaches source and `no_egress` state to kernel task and inode objects. It monotonically merges state across supported process, file, and inter-process communication (IPC) edges and checks network egress at socket creation, connection, and send time.

This paper answers three research questions (RQs):

- **RQ1: Security correctness.** Does Agent Guard satisfy predefined behavioral criteria for 19 isolation, propagation, enforcement, and lifecycle properties?
- **RQ2: Cost decomposition.** What file input/output (I/O) costs arise from loading the data plane, entering propa-

gation tracking, and enabling full policy enforcement?

- **RQ3: Frozen-baseline comparison.** How do Agent Guard and a frozen ActPlane revision compare on common target properties and co-located file-I/O workloads?

The contributions are:

- a human-authorized tri-state asset contract that separates model suggestions from final authority and reduces a task damage boundary to bounded kernel actions;
- gate-before-exec task binding and object-level monotonic state merging across supported processes, files, pipes, and Unix-domain sockets;
- a shared eBPF LSM data plane, a fail-closed control path with rollback, and an audit channel separated from enforcement state; and
- an evaluation comprising 570 formal security records, 420 cost-decomposition trials, and 315 co-located performance trials, plus a property-by-property comparison with frozen ActPlane.

The security claim is scoped to declared assets, implemented kernel paths, and the threat model in Section III. Non-goals and lifecycle constraints are consolidated in Sections III-D and VII.

## II. Background and Related Work

### A. Kernel Enforcement and Object State

Linux Security Modules provide hooks on security-sensitive file, process, and socket paths. BPF LSM programs can attach to these hooks and return an error before an operation proceeds [6]. BPF maps retain enforcement state [7], [8], while the verifier validates program safety [9]. Task and inode local storage associate values directly with kernel task and inode objects [10], reducing dependence on recycled numeric process identifiers (PIDs) and explicit cleanup of PID- or inode-keyed hash entries.

System provenance represents processes and kernel objects as graph nodes and system interactions as directed edges [11], [12]. Dynamic information-flow tracking instead propagates labels across computation and communication [13]. CamFlow, CamQuery, ProTracer, eAudit, and TAPAS study whole-system capture, online query, and audit-volume reduction [14]–[18]. AgentSight correlates agent-level semantics with eBPF events for system-level observability [19]. These systems establish the value of object-level correlation; Agent Guard focuses on authorization denial before a supported side effect. Agent-Warden similarly uses object-lifetime state and cross-process provenance tracking for LLM-agent activity [20].

### B. Agent Policy and Linux Isolation

AgentSpec enforces declarative constraints within an agent runtime, whereas AgentBound applies manifest-declared capabilities to MCP servers through a containerized policy-enforcement boundary [21], [22]. Progent, Fides, and CaMeL constrain tool use and data release through a domain-specific language, planner-level information flow, or trusted control/data-flow separation [23]–[25]. Together, these systems use richer agent or server semantics than a generic kernel monitor and can complement Agent Guard.

TABLE I
Positioning of Related Mechanisms

| Class | Enforcement | Relationship to Agent Guard |
|---|---|---|
| Agent policy | Agent runtime/tool path | Rich semantics; upper policy layer |
| MCP sandbox | Container/OS resources | Manifest-scoped server capabilities |
| Linux isolation | Linux kernel | Resource-oriented privilege boundary |
| Provenance | Kernel capture and analysis | Object correlation and observability |
| ActPlane | Compiler and kernel | Programmable compliance baseline |
| Agent Guard | Control plane and BPF LSM | Conditional damage-boundary enforcement |

Seccomp-BPF, AppArmor, and Landlock provide system-call filtering, profile-based mediation, and stackable access control [26]–[28]. They provide resource-oriented maximum-privilege boundaries that are typically declared before the relevant workload runs, but such a rule set is not automatically equivalent to the task-conditioned rule "after reading this asset, propagate an egress restriction through a file or IPC object to another execution carrier."

ActPlane is the closest system. It compiles project and task rules from an agent harness into operating-system policy, with cross-event state, information-flow control, and semantic feedback [29]. Agent Guard does not claim eBPF enforcement, fork propagation, or file-mediated flow as unique mechanisms. The systems prioritize different authorization and policy models: ActPlane targets programmable compliance rules, whereas Agent Guard targets a small human-authorized asset damage boundary.

## III. System Model and Security Goals

### A. Subjects, Objects, and Contracts

The system contains a human authorizer $H$, a proposer $M$, a task execution domain $A_\tau$, a control plane $C$, and a kernel enforcer $K$. The human maintains a declared asset set $\mathcal{R}$. Each asset has a stable `asset_id`, type, semantic metadata, and a local canonical path. The proposer emits candidate actions, the control plane validates and installs the final contract, and the kernel enforcer maintains state and returns denial.

For task $\tau$, the confirmed contract is a total function

$$\Gamma_\tau : \mathcal{R} \to \{\texttt{allow}, \texttt{deny}, \texttt{no_egress}\}. \tag{1}$$

`allow` adds neither of the two Agent Guard restrictions; other operating-system checks remain in force. `deny` rejects supported file-access and namespace operations. `no_egress` permits local access but prevents a carrier that acquires the state from using a managed network egress.

Each active task receives a non-reused user-space `task_uuid` and a bounded kernel position `task_id` $\in [0, 63]$, with source bit $b_\tau = 1 \ll \texttt{task_id}$. A process or supported kernel object $x$ has state

$$S(x) = \langle L(x), E(x) \rangle, \tag{2}$$

where $L(x)$ is a source bitmap and $E(x) \in \{0, 1\}$ denotes managed-egress denial. State joins as

$$\langle L_1, E_1 \rangle \sqcup \langle L_2, E_2 \rangle = \langle L_1 \vee L_2, E_1 \vee E_2 \rangle. \tag{3}$$

The join is commutative, associative, and idempotent; ordinary tasks can add but cannot clear live-object state.

### B. Threat and Trust Model

The proposer and task execution domain are untrusted. An adversary may control task commands, ordinary user-space descendants, and agent inputs. It may spawn processes, invoke native binaries, rename files, pass file descriptors, transfer intermediate results through regular files, FIFOs, anonymous pipes, or Unix-domain sockets, and establish a network connection before acquiring restricted state. It may also attempt state laundering through process or object identifier reuse.

We trust the Linux kernel, the eBPF/LSM path, the privileged Agent Guard control plane, and the human's final authorization input. The task has neither root nor Linux kernel-management capabilities such as `CAP_BPF` or `CAP_SYS_ADMIN`. Trusting the final authorization input does not assume that the human's choice is objectively optimal.

### C. Security Invariants

Agent Guard enforces six invariants:

1) **Closed authorization.** Every enabled asset appears exactly once, and a model suggestion cannot directly become kernel policy.
2) **Pre-execution binding.** Untrusted code cannot run before policy installation, task-object registration, and read-back validation complete.
3) **Object identity.** State is bound to kernel task or inode objects rather than only to reusable numeric PIDs or path strings.
4) **Monotonic propagation.** For supported derivation, write, read, and Unix-peer edges, the receiver applies $S(dst) \leftarrow S(dst) \sqcup S(src)$.
5) **Pre-side-effect denial.** Supported asset operations matching `deny`, and managed network operations with $E(p) = 1$, return `EPERM` before the prohibited side effect.
6) **Fail-closed setup.** Validation, policy installation, or registration failure leaves the workload behind the execution gate and triggers implemented rollback.

### D. Lifecycle and Non-Goals

When the root task exits, background descendants may remain. Agent Guard uses a Linux control group (cgroup) to track the derived task tree and cleans task policy only after the cgroup is empty. This condition terminates the controlled process tree but does not prove that a persistent inode or an out-of-cgroup receiver has discarded an old source bit. Safe reuse of a bounded bit position therefore requires a generation tag, complete global liveness tracking, or a conservative no-reuse policy.

The current system does not target complete task correctness, undeclared-asset protection, byte-level or implicit flow, every IPC/device type, a root-compromised host, or unbounded audit delivery. It also does not provide a general in-task declassification interface. These non-goals define the scope of the invariants rather than changing their meaning.

## IV. Design

### A. Human-Authorized Control Flow

The control flow consists of asset declaration, model proposal, human confirmation, contract compilation, pre-execution registration, kernel enforcement, and task teardown. The model receives asset identifiers and semantic metadata but need not receive canonical local paths. The control plane rejoins confirmed actions with the local asset table and rejects missing, duplicate, unknown, or out-of-domain actions. Only a complete contract can proceed to privileged installation.

This design separates three authority domains. The proposer can suggest semantics but cannot authorize. The human authorizes the complete asset set but cannot directly mutate BPF maps. The privileged daemon installs only validated, locally resolved policy. The separation prevents the constrained agent from unilaterally expanding its own authority.

### B. Execution Gate and Task Binding

The launcher creates a child and blocks it on a dedicated file descriptor before workload execution. The parent obtains a pidfd and sends the pidfd, task identifiers, and contract path to the privileged daemon over a message-preserving Unix `SOCK_SEQPACKET` channel. The daemon authenticates peer credentials and validates ownership, bit-position conflicts, and pidfd/PID consistency. The parent opens the execution gate only after policy installation and task-local-state read-back succeed.

A pidfd refers to a concrete process object, and BPF task storage follows the kernel task lifetime. Numeric PIDs remain useful for logs and cgroup management but do not independently carry security identity. The ordering prevents the workload from running before registration.

### C. Asset Compilation and Multi-Task Sharing

The control plane resolves each existing file or directory to a $(device, inode)$ key. The corresponding policy value contains `denied_lineages` and `no_egress_lineages` bitmaps. A new contract changes only its source bit and preserves other active tasks' bits on the same object. One long-lived kernel data plane can therefore serve multiple tasks without stacking an LSM program per task.

Directory policy must cover existing descendants and objects created after installation. The compiler enumerates the existing tree, while the kernel performs a bounded ancestor walk of at most 32 directory entries from the target. Deeper paths are outside the current directory-policy scope.

### D. Monotonic Propagation

For processes $P, Q$ and supported object $O$, the core transitions are

$$\begin{aligned} S(Q) &\leftarrow S(Q) \sqcup S(P) && \text{process derivation,} \\ S(O) &\leftarrow S(O) \sqcup S(P) && \text{writable access,} \\ S(P) &\leftarrow S(P) \sqcup S(O) && \text{readable access.} \end{aligned} \tag{4}$$

Regular files, FIFOs, and anonymous pipes use inode state. The endpoints of a Unix stream socket or `socketpair()` have different inodes, so the sender first joins its state into a peer-state container and the receiver absorbs that state on receive.

The object/permission-level model is deliberately conservative. It may approximate the ability to write as an actual influence, but it reduces missed propagation through intermediate carriers while keeping state bounded. An `exec` changes the program image without clearing state. Relaxing a contract stops future asset matches but does not revoke `no_egress` already held by a live object. Thus, ordinary tasks cannot sanitize a label by re-executing, deleting a rule, or relaying through an intermediate object.

### E. Asset and Network Enforcement

For a file operation, the kernel reads the process bitmap and looks up the target inode and bounded ancestors. A supported operation is denied when

$$L(P) \wedge Policy(O).denied \neq 0. \tag{5}$$

The implemented denial paths cover open, create, truncate, unlink, and both rename endpoints. A `no_egress` match permits the file operation and sets $E(P) = 1$ in the same kernel decision path.

Managed network egress checks the same state at three stages: socket creation covers endpoints created after restriction; connect covers Transmission Control Protocol (TCP) and connected User Datagram Protocol (UDP) destinations; send covers connections established before restriction and connectionless datagrams. Unix-domain sockets are local propagation channels rather than external egress. The prototype does not implement domain-name, IP-address, or port allowlists.

### F. Transactions and Audit Separation

Before modifying kernel state, the compiler validates schema and version, task-bit consistency, asset identity and type, action-set disjointness, and BPF-map layout. A cross-process lock serializes writers. The loader snapshots relevant map contents and restores the snapshot on an implemented update failure. Initial policy is installed before the execution gate opens; a running task's multi-key hot update, however, is not a linearizable single commit to concurrent kernel readers.

Enforcement state and audit delivery are independent. The LSM return value decides whether the current operation proceeds; a BPF ring buffer carries asynchronous evidence. If event reservation fails, a per-CPU counter increases, but an operation that should be denied is not changed to allow.

TABLE II
BPF STATE CONTAINERS

| Name | Type | State / Bound |
|---|---|---|
| Task label | `TASK_STORAGE` | Source and egress state; task lifetime |
| Inode label | `INODE_STORAGE` | File/IPC state; inode lifetime |
| Unix peer | `HASH` | Peer receive state; 4,096 entries |
| Asset policy | `HASH` | Deny/egress bitmaps; 4,096 entries |
| Event ring | `RINGBUF` | Fixed-width events; 256 KiB |
| Drop count | `PERCPU_ARRAY` | Reservation failures |

Security experiments therefore use system-call results and resource side effects as primary criteria, with event order and drop counts as supporting evidence.

## V. IMPLEMENTATION

Agent Guard consists of ordinary-user asset and contract tools, a C launcher and privileged daemon, a policy loader, and a BPF C data plane. The ordinary-user path handles asset metadata, the model proposal, human confirmation, and contract persistence. The daemon maintains one long-lived BPF instance. The loader resolves local objects and updates policy, while the launcher manages the execution gate, pidfd registration, cgroup lifecycle, and result collection.

The prototype requires BPF LSM, BPF Type Format (BTF), the BPF file system (bpffs), cgroup v2, pidfd, and BPF task/inode local storage. It uses Compile Once–Run Everywhere (CO-RE) relocation, but deployment still treats actual loading and regression tests as the capability check.

Task storage is attached to the thread-group leader so threads in one process share logical state. Inode storage is used only for the supported file, FIFO, pipe, and socket classes. Unix peer propagation uses the additional hash map. Table III lists the data-plane attachment points.

The privileged loader uses bounded JavaScript Object Notation (JSON) parsing and validates contract versions, task bits, asset types, action disjointness, and map key/value layouts. Directory compilation does not follow symlinks into another tree and rejects more than 4,096 compiled policy keys. The principal implementation bounds are 64 active source positions, 4,096 asset keys, 4,096 Unix-peer keys, 32 ancestor levels, and a 256-KiB ring buffer.

## VI. EVALUATION

### A. Methodology

The evaluation separately measures security behavior, mechanism cost, and the frozen baseline. Model calls and human interaction are outside the timing interval. Performance trials use deterministic proposals but traverse the same contract validation, persistence, policy-loading, execution-gate, and registration path.

Table IV summarizes the evidence. Agent Guard security and cost decomposition ran on Linux 7.0. Frozen ActPlane requires Linux 6.15, so its property tests and the co-located

TABLE III
eBPF ATTACHMENT POINTS AND RESPONSIBILITIES

| Attachment | Responsibility | Attachment | Responsibility |
|---|---|---|---|
| `lsm/file_open` | Asset decision and regular-file propagation | `lsm/file_permission` | Pipe/FIFO propagation and socket-fd write check |
| `lsm/inode_create` | Protected directory creation denial | `lsm/path_truncate` | Truncation denial |
| `lsm/inode_unlink` | Unlink denial | `lsm/inode_rename` | Source/destination rename checks |
| `lsm/socket_create` | New external-socket denial | `lsm/socket_connect` | Destination connection denial |
| `lsm/socket_sendmsg` | Established-send denial and Unix send propagation | `lsm/socket_recvmsg` | Unix receiver state absorption |
| `tp_btf/sched_process_fork` | Child state inheritance | `tracepoint/sched/sched_process_exit` | Exit audit |
| `fentry/security_inode_free` | Hash-key cleanup at inode release | | |

performance comparison ran in a Linux 6.15 VMware guest. Different-kernel batches are not used for direct absolute-time comparison. The guest exposed eight logical processors and reported an AMD Ryzen 9 9950X model string; we treat these data as virtual-machine (VM), not bare-metal, results.

Each performance configuration has five warmups and 30 formal repetitions with randomized within-repetition order. For system $s$ and repetition $r$, the paired overhead relative to Native is

$$O_{s,r} = \frac{T_{s,r} - T_{N,r}}{T_{N,r}} \times 100\%, \tag{6}$$

where $T_{N,r}$ is the Native time for the same repetition. We report medians, 95th percentiles (p95), and a 95% interval from 10,000 bootstrap resamples of the 30 paired overheads.

### B. RQ1: Security Correctness

Security tests use system-call return values and resource side effects as primary criteria. Each repetition randomizes all 19 cases and cleans controlled processes and policy state between cases. All 570 formal records are valid and satisfy their predefined criteria; the aggregate BPF-ring-buffer drop count is zero.

Cases 3 and 13 show why connect-only enforcement is insufficient for established channels. Case 11 checks both return values and filesystem effects rather than treating a denial log as proof. Case 14 tests pre-task rejection and user-space rollback. Thus, for the controlled cases and implemented paths in Table V, Agent Guard maintains source isolation, monotonic propagation, and egress denial.

### C. RQ2: Cost Decomposition

The four configurations are: *Baseline*, which does not load Agent Guard; *Loaded-idle*, which loads the daemon and BPF programs but does not register the measured task; *Tracing*, which registers the task and runs propagation/audit paths with an allow-all contract; and *Full*, which enables the asset policy. Two microbenchmarks each execute one million `open/read/close` transactions; a macrobenchmark performs 700,000 transactions over 14 representative files. All 420 trials are valid, and 360 formal samples enter the statistics.

TABLE VI
AGENT GUARD FILE-I/O COST DECOMPOSITION

| Workload | Config. | Median ms | Paired % |
|---|---|---|---|
| Public $10^6$ | Baseline | 764.663 | – |
| | Loaded-idle | 871.964 | 14.077 |
| | Tracing | 1019.968 | 33.592 |
| | Full | 1019.965 | 33.230 |
| Protected $10^6$ | Baseline | 787.789 | – |
| | Loaded-idle | 889.757 | 13.220 |
| | Tracing | 1058.295 | 34.788 |
| | Full | 1069.548 | 35.622 |
| 14 assets | Baseline | 962.830 | – |
| | Loaded-idle | 1034.648 | 7.666 |
| | Tracing | 1156.708 | 20.287 |
| | Full | 1165.788 | 21.156 |

Full enforcement has a median paired overhead of 21.156–35.622%. The larger observed increment appears between Loaded-idle and Tracing, while the Tracing–Full difference is smaller. We did not perform a significance test on that component difference and therefore do not interpret it as zero policy-lookup cost.

### D. RQ3: ActPlane Property Comparison

We freeze ActPlane at commit `c91f386e`. A preflight confirms that its eBPF engine loads on Linux 6.15 and that an exact rule produces a real `EPERM`. The 19 target properties use the same return-value and side-effect criteria as Table V. Each ActPlane property executes once, so this is a per-property coverage observation rather than a repeated-rate estimate.

Figure 1 reports 19 PASS outcomes for Agent Guard. Frozen ActPlane has 9 PASS, 3 FAIL, 4 PARTIAL, 2 UNSUPPORTED, and 1 DIFFERENT. Its native passes include source/sibling isolation, fork, regular-file, FIFO, anonymous-pipe, and 64-source properties. Non-pass outcomes arise from a missing send-stage enforcement point; an `O_CREAT` operation that leaves an empty file before returning denial; absent Unix-peer propagation; and different hot-update, object-identity, or directory-boundary semantics.

The comparison should not be collapsed into "ten ActPlane failures." PARTIAL, UNSUPPORTED, and DIFFERENT denote partial comparability, a missing expression point, and different resource semantics. The 19 properties also do not exercise all ActPlane temporal and policy-domain capabilities. The result identifies overlapping mechanisms optimized for different security objectives.

TABLE IV
FORMAL EVALUATION BATCHES

| Evidence | Environment and Revision | Scale |
|---|---|---|
| Agent Guard security | Linux 7.0.0-29-generic; `e423fcc` | 19 cases × 30 repetitions = 570 records |
| Agent Guard cost decomposition | Linux 7.0.0-29-generic; `9b8aa6b` | 3 workloads × 4 configurations × (5 warmups + 30 formal) = 420 trials |
| ActPlane property comparison | Linux 6.15.0-061500-generic VMware; `c91f386` | 19 properties, one execution per property |
| Co-located performance | Same Linux 6.15 VMware guest | 3 workloads × 3 systems × (5 warmups + 30 formal) = 315 trials |

TABLE V
AGENT GUARD SECURITY PROPERTIES AND FORMAL RESULTS

| No. | Property | Primary Behavioral Criterion | Pass/Run |
|---|---|---|---|
| 1 | Source-isolated `deny` | Restricted task returns `EPERM`; other source reads | 30/30 |
| 2 | Public asset | Network succeeds without restricted state | 30/30 |
| 3 | Post-infection established send | Send succeeds before and fails after infection | 30/30 |
| 4 | Sibling isolation | Infected branch denied; sibling remains usable | 30/30 |
| 5 | Fork inheritance | Child inherits `no_egress` | 30/30 |
| 6 | Cross-task regular file | A writes, B reads, then B cannot network | 30/30 |
| 7 | No reverse propagation on exit | Restricted child exit does not infect parent | 30/30 |
| 8 | Hot update and monotonicity | New rule applies; removal retains acquired state | 30/30 |
| 9 | FIFO propagation | A–FIFO–B causes B egress denial | 30/30 |
| 10 | 64-bit source isolation | 32 targets denied; 32 controls allowed | 30/30 |
| 11 | File-operation coverage | Write/truncate/create/rename/unlink leave no forbidden effect | 30/30 |
| 12 | Aliases and inode lifetime | Links/rename do not bypass; reused inode is clean | 30/30 |
| 13 | Established-socket send paths | `send`, `sendto`, `sendmsg`, and `write` denied | 30/30 |
| 14 | Invalid contracts and rollback | Six invalid/injected failures do not start target | 30/30 |
| 15 | PID reuse | New process does not inherit old task state | 30/30 |
| 16 | Anonymous pipe | Producer–pipe–consumer propagates restriction | 30/30 |
| 17 | Unix stream peer | Receiver absorbs sender state | 30/30 |
| 18 | Socketpair peer | Receiver absorbs sender state | 30/30 |
| 19 | Directory ancestor bound | Depth 31 matches; depth 32 is outside declared scope | 30/30 |

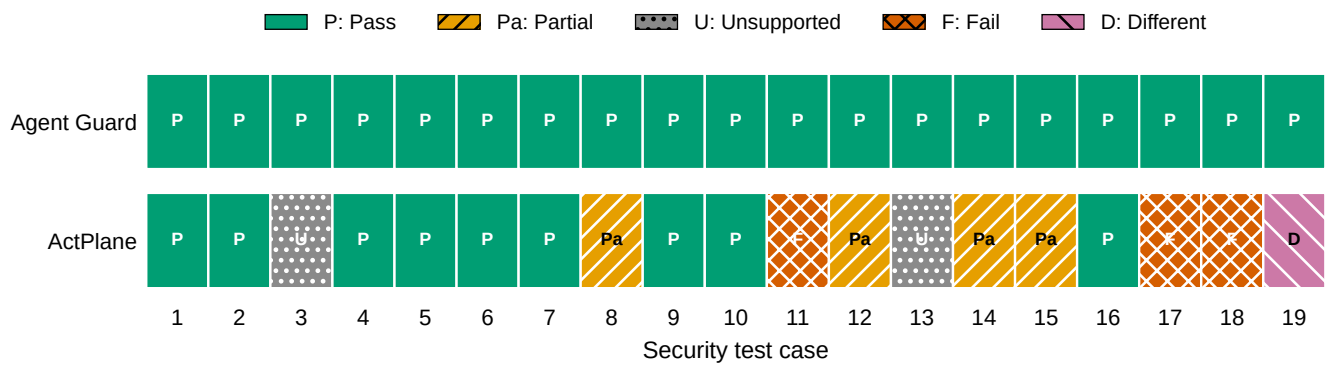



Fig. 1. Per-property outcomes for the 19 target security properties. Agent Guard results are from 30 formal repetitions per property; frozen ActPlane results are one execution per property.

TABLE VII
CO-LOCATED FILE-I/O PERFORMANCE COMPARISON

| Workload | System | Median (ms) | p95 (ms) | Paired median | Bootstrap 95% interval |
|---|---|---|---|---|---|
| Public read | Native | 14864.112 | 14927.003 | – | – |
| | ActPlane | 19807.802 | 19931.657 | 33.386% | [33.040%, 33.805%] |
| | Agent Guard | 16617.445 | 16717.512 | 11.958% | [11.637%, 12.135%] |
| Protected read | Native | 14948.096 | 15014.394 | – | – |
| | ActPlane | 24051.629 | 24162.746 | 61.022% | [60.701%, 61.106%] |
| | Agent Guard | 16862.441 | 16951.904 | 12.891% | [12.655%, 13.047%] |
| 14-asset macro | Native | 10875.662 | 10961.053 | – | – |
| | ActPlane | 16612.504 | 16867.854 | 52.441% | [52.134%, 53.159%] |
| | Agent Guard | 12210.939 | 12289.781 | 12.322% | [11.900%, 12.644%] |

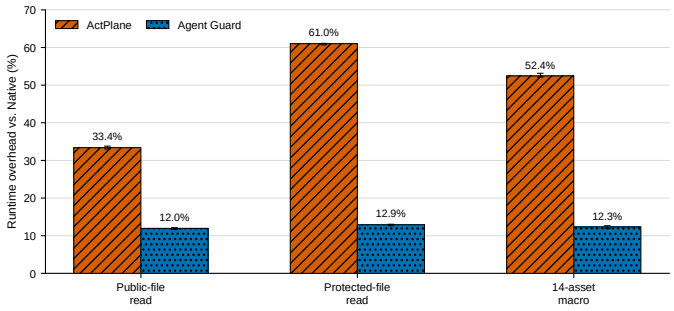



Fig. 2. Median paired runtime overhead relative to Native, with 95% bootstrap intervals.

### E. RQ3: Co-Located Performance

Native, ActPlane, and Agent Guard use the same `file-io-bench` binary, files, operation counts, and content-hash criteria. All 315 trials are valid: 45 warmups and 270 formal samples. Operation counts and aggregate hashes match across systems. Across 105 Agent Guard daemon lifecycles, the maximum `ringbuf_dropped` value is zero. Timing covers only the steady-state workload, excluding engine loading, model suggestions, and human confirmation.

As shown in Fig. 2 and Table VII, Agent Guard's median paired overhead is 11.958–12.891%, versus 33.386–61.022% for ActPlane; the bootstrap intervals do not overlap for any workload. This result measures the frozen revisions on one VM and three deterministic file-I/O workloads. It is not an overall ranking of all features or workloads.

### F. Threats to Validity

The 19 properties derive from our threat model and are better aligned with conditional damage boundaries than with all ActPlane compliance features. Security, cost-decomposition, and co-located data come from different revisions and kernels; only the Linux 6.15 co-located batch supports direct absolute-time comparison. Random order, warmups, content hashes, and pairing reduce cache and temporal drift but cannot remove VM scheduling noise. The performance workloads emphasize high-frequency file I/O and do not yet represent real agent-task distributions, network paths, fork/IPC cost, or policy-update cost.

## VII. Discussion and Limitations

### A. Authorization Root and User Studies

Human confirmation determines who authorizes the boundary, not whether that judgment is necessarily correct. An authorizer may omit an asset or select an overly permissive action. Our evaluated claim is conditional on a fixed contract: the kernel monitor should translate the confirmed actions into task state, propagate restrictions on supported paths, and deny prohibited effects. A user study is therefore not required to establish this conditional runtime invariant, but it is required for claims about interaction usability, contract quality, confirmation time, or alert fatigue. Such a study should report participant background, tasks, training, errors, and decision time; it would complement rather than replace the kernel experiments.

### B. Conservative Propagation and Declassification

Object/permission-level propagation can approximate the ability to write as actual influence and therefore over-taint an execution. The trade-off keeps kernel state bounded and reduces missed propagation through intermediate carriers. Restoring network capability requires a trusted declassification mechanism, such as an attested transformation, an independently authorized release, or a new object carrying proof of the transformation. An ordinary task must not remove a label merely by asserting that data were sanitized.

### C. Source Reuse and Policy Updates

The 64-bit source representation makes joins fixed-width and efficient, but cgroup drainage proves only that the derived task tree exited. A persistent inode or out-of-cgroup receiver may still carry an old bit. Safe reuse requires generations, global reference tracking, or conservative non-reuse. Likewise, current multi-key updates serialize writers and roll back failures but are not linearizable to readers of a running task. Double-buffered maps and a single-key generation switch are natural extensions.

### D. Composition and External Validity

The tri-state contract does not express rich temporal rules, destination domains, or workflow-repair guidance. A deployment can use AppArmor, Landlock, or a container for a resource-oriented maximum-privilege boundary; Agent Guard for task-conditioned asset restrictions; and an agent runtime for semantic feedback. The current VM results quantify steady-state file-I/O cost for frozen configurations. Bare-metal, multi-kernel, concurrent-task, and realistic agent workloads remain important evaluation extensions.

## VIII. Conclusion

Agent Guard defines human confirmation as the formal authorization root of an asset damage boundary and converts a tri-state contract into task-bound kernel state. An execution gate and pidfd complete registration before untrusted code runs. An eBPF LSM data plane monotonically propagates restrictions across supported processes, files, pipes, and Unix-domain sockets and deterministically denies prohibited asset and network side effects.

All 570 formal records across 19 security properties satisfy their behavioral criteria. Cost decomposition quantifies loading, propagation, and full enforcement. On three co-located Linux 6.15 VM workloads, Agent Guard's median paired overhead is 11.96–12.89%, compared with 33.39–61.02% for frozen ActPlane. These results support kernel enforcement of human-authorized damage boundaries for declared assets and supported execution paths. Future work will prioritize safe source reuse, linearizable policy publication, trusted declassification, and evaluation on bare metal and realistic agent tasks.

## AI Disclosure

During the preparation of this work, the authors used Google Gemini, DeepSeek, and OpenAI Codex to assist with English-language editing, manuscript organization, and readability. The authors reviewed and edited all AI-assisted content and take full responsibility for the integrity, correctness, and originality of the manuscript, including all citations and references.

## References


[1] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024.

[2] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection," in *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec '23)*, Nov. 2023, pp. 79–90.

[3] H. Zhang, J. Huang, K. Mei, Y. Yao, Z. Wang, C. Zhan, H. Wang, and Y. Zhang, "Agent security bench (asb): Formalizing and benchmarking attacks and defenses in LLM-based agents," in *Proceedings of the 13th International Conference on Learning Representations (ICLR)*, 2025. [Online]. Available: https://arxiv.org/abs/2410.02644

[4] OWASP Foundation, "OWASP Top 10 for LLM Applications 2025," OWASP Foundation, Tech. Rep., 2025, [Online]. Available: https://owasp.org/www-project-top-10-for-large-language-model-applications/assets/PDF/OWASP-Top-10-for-LLMs-v2025.pdf. Accessed: Aug. 30, 2026.

[5] M. Lupinacci, F. A. Pironti, F. Blefari, F. Romeo, L. Arena, and A. Furfaro, "The Dark Side of LLMs: Agent-based attack vectors for system-level compromise," arXiv preprint arXiv:2507.06850, 2025. [Online]. Available: https://arxiv.org/abs/2507.06850

[6] The Linux Kernel Documentation, "LSM BPF Programs," [Online]. Available: https://docs.kernel.org/bpf/prog_lsm.html, 2026, accessed: Aug. 30, 2026.

[7] B. Gregg, *BPF Performance Tools: Linux System and Application Observability*. Addison-Wesley Professional, Dec. 2019.

[8] The Linux Kernel Documentation, "BPF Maps," [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/maps.html, 2026, accessed: Aug. 30, 2026.

[9] H. Sun and Z. Su, "Validating the eBPF verifier via state embedding," in *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024, pp. 615–628.

[10] Linux man-pages Project, "bpf-helpers(7): eBPF helper functions," [Online]. Available: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html, 2026, accessed: Aug. 30, 2026.

[11] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 223–236.

[12] Z. Li, Q. A. Chen, R. Yang, Y. Chen, and W. Ruan, "Threat detection and investigation with system-level provenance graphs: A survey," *Computers & Security*, vol. 106, p. 102282, 2021.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 393–407.

[14] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon, "Practical whole-system provenance capture," in *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, 2017, pp. 405–418.

[15] T. F. J.-M. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, "Runtime analysis of whole-system provenance," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 1601–1616.

[16] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards practical provenance tracing by alternating between logging and tainting," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[17] R. Sekar, H. Kimm, and R. Aich, "eAudit: A fast, scalable and deployable audit data collection system," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2024, pp. 3571–3589.

[18] B. Zhang, Y. Gao, C. Yu, B. Kuang, Z. Zhang, H. Kim, and A. Fu, "TAPAS: An efficient online APT detection with task-guided process provenance graph segmentation and analysis," in *Proceedings of the 34th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2025, pp. 607–624.

[19] Y. Zheng, Y. Hu, T. Yu, and A. Quinn, "Agentsight: System-level observability for AI agents using eBPF," in *Proceedings of the 4th Workshop on Practical Adoption Challenges of ML for Systems (PACMI '25)*, 2025, pp. 110–115.

[20] D. Cui, Z. Gu, P. Zheng, S. Han, and Y. Liao, "Agent-Warden: eBPF-based kernel-native process-file provenance tracking for LLM agents," 2026, unpublished manuscript.

[21] H. Wang, C. M. Poskitt, and J. Sun, "Agentspec: Customizable runtime enforcement for safe and reliable LLM agents," in *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*. Rio de Janeiro, Brazil: ACM, April 2026, 12 pages; also available as arXiv:2503.18666.

[22] C. Bühler, M. Biagiola, L. Di Grazia, and G. Salvaneschi, "AgentBound: Securing execution boundaries of AI agents," *Proceedings of the ACM on Software Engineering*, vol. 3, no. FSE, pp. 2141–2164, 2026.

[23] T. Shi, J. He, Z. Wang, H. Li, L. Wu, W. Guo, and D. Song, "Progent: Securing AI agents with privilege control," arXiv preprint arXiv:2504.11703, 2025. [Online]. Available: https://arxiv.org/abs/2504.11703

[24] M. Costa, B. Köpf, A. Kolluri, A. Paverd, M. Russinovich, A. Salem, S. Tople, L. Wutschitz, and S. Zanella-Béguelin, "Securing AI agents with information-flow control," arXiv preprint arXiv:2505.23643, 2025. [Online]. Available: https://arxiv.org/abs/2505.23643

[25] E. Debenedetti, I. Shumailov, T. Fan, J. Hayes, N. Carlini, D. Fabian, C. Kern, C. Shi, A. Terzis, and F. Tramèr, "Defeating prompt injections by design," in *Proceedings of the 4th IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, Munich, Germany, Mar. 2026, also available as arXiv:2503.18813. [Online]. Available: https://arxiv.org/abs/2503.18813

[26] The Linux Kernel Documentation, "Seccomp BPF: Secure computing with filters," [Online]. Available: https://docs.kernel.org/userspace-api/seccomp_filter.html, 2026, accessed: Aug. 29, 2026.

[27] AppArmor Project, "AppArmor Documentation," [Online]. Available: https://apparmor.net/, 2026, accessed: Aug. 29, 2026.

[28] The Linux Kernel Documentation, "Landlock: Unprivileged access control," [Online]. Available: https://www.kernel.org/doc/html/latest/userspace-api/landlock.html, 2026, accessed: Aug. 29, 2026.

[29] Y. Zheng, T. Wu, Q. Fu, T. Yu, W. Mao, T. Ma, D. Williams, W. Wang, and A. Quinn, "ActPlane: Programmable OS-level policy enforcement for agent harnesses," arXiv preprint arXiv:2606.25189, 2026. [Online]. Available: https://arxiv.org/abs/2606.25189