IOArx: Agent-Aware Policy Enforcement Between AI Agents and Sensitive Data on the Endpoint

Ali Mohammadi Ruzbahani
IOArx Development Team
Calgary, Alberta, Canada
info@ioarx.com

PubArx:2026.00016v1 [cs.CR] 26 Nov 2026 · Preprint

***Abstract*—AI agents now read and change files, run commands and open network connections on the computers of the people who use them, and the operating system authorizes every one of these actions as the user, because it has no notion of an agent. A single injected instruction or misjudged step therefore reaches every credential, document and connection the user can reach. We present IOArx, which makes the agent the principal of access control on the endpoint without changing agents, models or the kernel. IOArx names the agent behind each kernel event by following the process chain to verified code-signing evidence, decides the event with a policy written for agents, pauses or stops the agent's entire process tree when the policy asks or forbids, refuses forbidden connections at their first packet, and inspects prompts, pastes and uploads before they reach AI chat services. We formalize the policy and prove that its decisions are deterministic, that rules for ordinary programs cannot grant an agent access, that added rules move decisions monotonically, that remembered answers bound the number of prompts, and how much an agent can still do before it is held; we also derive the exact error rates of the checksum and entropy validation in the prompt detectors. On a developer workstation, IOArx attributed 36 processes to their agents, 16 of them through their ancestors, without a single misattribution, and recognized no program outside an agent product among 3,449 executables. On Linux it stopped a violating agent within a median of 3.1 ms, a complete decision takes 19 μs, and on 32,771 real user prompts the default detectors raised no false alarm while finding 99.65% of the credentials and personal data planted in them.**

***Index Terms*—AI agents, endpoint security, access control, runtime enforcement, prompt injection, data loss prevention**

I. Introduction

AI agents have moved from suggesting code to carrying it out. A coding agent reads a repository, edits files, installs packages and runs tests on the developer's own computer, and desktop assistants and local model runtimes work beside it on the same machine. Every one of these actions reaches the operating system as a request from the user, because the agent runs under the user's account, and the operating system grants it, because it has no notion of an agent. Whatever the user may read, change, delete or send, so may every agent the user starts. A single instruction planted in a web page or an issue, or a single misjudged step, then places SSH keys, cloud credentials, private documents and the network at the agent's disposal. The browser carries the same exposure in another form, since people paste configuration files, logs and personal details into AI chat services as a matter of routine.

The defenses available today address parts of this problem from positions that cannot cover it. Defenses inside agent frameworks constrain how a model plans and calls its tools [1], [2], yet they protect only the agents whose vendors adopt them, and a single workstation runs agents from many vendors at once. A sandbox confines one program at a time and must be set up for it in advance, while an agent's work spreads across shells, interpreters and build tools. Endpoint detection reasons after the fact over every process of a machine and already struggles with the volume of its own alerts [3]. None of them supplies what the operating system lacks: a principal for the agent, and a policy that follows it into everything it starts.

This paper presents IOArx, which makes the agent the principal of access control on the endpoint without changing the agents, their models or the kernel. IOArx observes the file and process events that the kernel already reports, names the agent behind each event by following the process chain to the nearest program that carries an agent's evidence, and decides the event with a policy written for agents. When the policy asks or forbids, IOArx pauses or stops the agent together with every process it started. A forbidden connection is refused at its first packet, and in the browser an extension checks each prompt, paste and upload before it reaches an AI service (Fig. 1). Two properties make this design sufficient. Attribution follows an agent through every unrecognized program it launches, so no shell or interpreter can launder its access (Theorem 2). Containment is fast: on Linux it completed within a median of 3.1 ms after the violating operation, two orders of magnitude below the median latency of a single model inference on coding requests [4], which an agent needs before it can act on what it has just read (Theorem 5).

This paper makes the following contributions:

- An architecture for agent-aware enforcement on the endpoint that derives an agent's identity from kernel events, process lineage and verified code-signing evidence, and that needs no cooperation from agents, model providers or the kernel (Sections II and IV).
- A formal semantics of the IOArx policy with proofs of determinism, of non-escalation through ordinary programs, of monotone refinement, of a bound on prompts under remembered answers, of exposure bounds for asynchronous containment and of tamper evidence for the activity log, together with an exact analysis of the checksum and entropy validation used by the prompt detectors (Sections III and IV).
- IOArx 1.0 for Windows and Linux, built on one shared

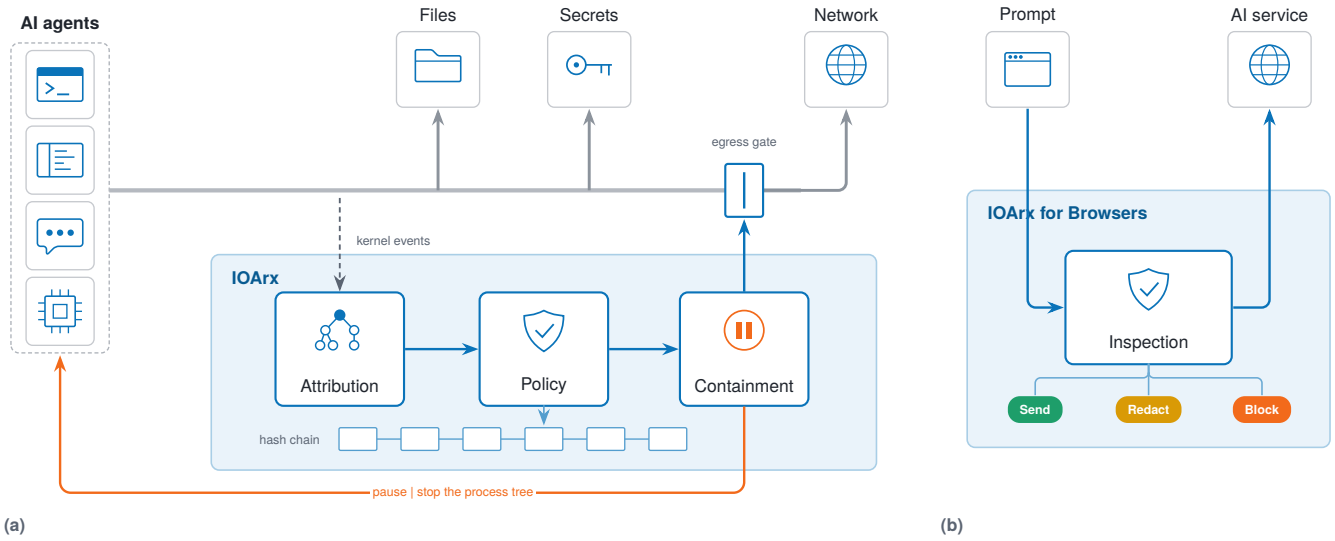


Fig. 1. Architecture of IOArx. (a) On the endpoint, IOArx names the agent behind each file event, decides it, contains violators and gates their network access. (b) In the browser, it inspects every submission to an AI service.

core, and a browser extension with 29 validated detectors for 19 AI services (Section V).

- An evaluation with the agents of a working developer machine, a stand-in agent driven to its limits, and 32,771 real user prompts, covering attribution, containment, cost and detection accuracy (Section VI).

II. Background and Threat Model

An AI agent couples a language model with tools. The model alternates between reasoning about a task and issuing actions whose results it reads back [5], and coding agents expose those actions as an interface of shell commands, file edits and test runs [6]. In deployment, each action becomes an ordinary process on the user's computer: a shell, an interpreter, a package manager or a compiler, started by the agent and running with the user's account. Research on language models in software engineering has grown into a field of its own [7], and the tools built on it now run on the machines that hold source code, signing keys and cloud credentials.

Operating systems were not designed for a principal that is neither the user nor an ordinary program. Access control on desktops is discretionary and keyed to the user, so a process started by an agent holds every right of the person who launched it. The principle of least privilege asks that every program operate with the least set of privileges necessary [8], but the kernel cannot tell the user reading a cloud credential file from an agent doing the same, and it therefore cannot apply a different policy to the agent.

The gap matters because agents follow instructions they find in content. Indirect prompt injection plants instructions in data that an application retrieves, so that an adversary without any direct interface to the application can exploit it, with data theft among the outcomes [9]. On the InjecAgent benchmark, a ReAct-prompted GPT-4 agent followed injected instructions in 24% of the cases, and in 47% once the attack was reinforced [10]; AgentDojo turns the same question into 97 realistic tasks and 629 security test cases [11]. Agents also cause harm without any adversary: in an emulated sandbox, even the safest agent evaluated showed risky failures 23.9% of the time [12]. What lies within reach is valuable. Secrets leak from more than 100,000 public repositories, with thousands of new ones every day [13], and a curated benchmark alone gathers 97,479 candidate secrets from 818 repositories [14]. In the browser, users of chat assistants disclose their own and other people's personal information [15], and personal data in a model's training text can later be extracted from the model [16].

Adversary. The adversary controls content that an agent consumes, such as a web page, an issue, a document or the output of a tool, or benefits from an agent's own error. The agent may then read, change or delete any file its user can access, start any program, and open network connections. It runs without administrator rights and acts through processes it starts.

Trusted base. The operating system kernel, IOArx with its administrator or root privileges, and processes outside the lineage of any agent are trusted. The browser and the extension are trusted to see what the user types.

Goals. IOArx must name the agent behind every file and network action, decide each action by a policy the user controls, hold a violating agent within a bounded time, refuse connections the policy forbids before they are established, stop credentials and personal data from reaching AI services through the browser, and keep a record of every decision that cannot be altered without detection.

III. A Formal Model of Agent-Aware Enforcement

Every IOArx platform shares one decision core, and the guarantees users rely on are properties of that core rather than of any operating system. This section states the core as

a small formal model and proves the properties that matter in practice. Decisions never depend on the order in which rules were written. A rule written for an ordinary program cannot be used to launder an agent's access. Adding a rule moves decisions in one predictable direction. A remembered answer ends prompting for everything it covers. Finally, the model bounds what an agent can still do between a violation and its containment, and it makes precise what the activity log can and cannot be silently changed to say.

A. Requests, Attribution and Policies

Definition 1 (Coverage)**.** *Let ν map a path to its normalized form, in which separators are unified, case is folded on Windows, and trailing separators are removed except at a root. A path p* covers *a path q, written $p \preceq q$, if $\nu(p)$ is a prefix of $\nu(q)$ and the remainder of $\nu(q)$ is empty, starts with a separator, or p is a root. We write $|p|$ for the length of $\nu(p)$.*

Since every path that covers q is a prefix of $\nu(q)$, two such paths of equal length are the same path. We use this fact twice below.

Definition 2 (Attribution)**.** *The chain of a process π is the sequence $c(\pi) = (g_0, g_1, \ldots, g_m)$ in which g_0 is the program of π and g_j that of its j-th ancestor. A catalog C maps each program to the name of an AI agent or to $\bot$. The agent of π is $\alpha(\pi) = C(g_j)$ for the least j with $C(g_j) \neq \bot$, and $\alpha(\pi) = \bot$ if no such j exists.*

Lemma 1 (Inheritance)**.** *If π' is a child of π and the catalog does not recognize the program of π', then $\alpha(\pi') = \alpha(\pi)$.*

Proof. The chain of π' is its own program followed by $c(\pi)$. Its first element maps to $\bot$, so the first recognized element is the first recognized element of $c(\pi)$. □

By induction, every process an agent starts through shells, interpreters, build tools or any other unrecognized program carries the agent's name, however deep the chain.

Definition 3 (Rules and policies)**.** *A request is $r = (\pi, q, o)$ with target path q and operation $o \in O = \{\mathsf{read}, \mathsf{write}, \mathsf{delete}, \mathsf{rename}\}$. A subject is one of any, agents, $\mathsf{agent}(a)$ or $\mathsf{image}(x)$; it applies to π always, if $\alpha(\pi) \neq \bot$, if $\alpha(\pi) = a$, or if $\alpha(\pi) = \bot$ and π runs program x, respectively, and then has rank $\kappa = 0, 1, 2, 3$ in that order. Actions are $D = \{\mathsf{Allow}, \mathsf{Ask}, \mathsf{Block}\}$ with strictness $s(\mathsf{Allow}) = 0 < s(\mathsf{Ask}) = 1 < s(\mathsf{Block}) = 2$. A rule $\rho = (p_\rho, sub_\rho, \Omega_\rho, d_\rho)$ applies to r if $p_\rho \preceq q$, its subject applies to π, and $\Omega_\rho = \emptyset$ or $o \in \Omega_\rho$. A policy $\Pi = (R, d_0)$ is a finite list of rules and a default action.*

Definition 4 (Decision)**.** *For a rule ρ that applies to r, let $k_r(\rho) = (|p_\rho|, \kappa(sub_\rho, \pi), s(d_\rho))$, ordered lexicographically. Let $A_\Pi(r)$ be the rules of R that apply to r. Then $\mathrm{Dec}_\Pi(r) = d_{\rho^*}$ for a rule ρ^* of maximal key in $A_\Pi(r)$, and $\mathrm{Dec}_\Pi(r) = d_0$ when $A_\Pi(r) = \emptyset$.*

The most specific path wins, then the most specific subject, then the strictest action. A user reads this order as "the closest rule decides, and a tie goes to caution".

B. Properties of Decisions and Remembered Answers

Theorem 1 (Determinism)**.** *$\mathrm{Dec}_\Pi(r)$ is well defined for every request and depends on R only as a set: permuting, duplicating or merging duplicate rules never changes a decision.*

Proof. Membership in $A_\Pi(r)$ and the key of a rule depend on the rule alone, not on its position. If two rules attain the maximal key, their third components agree, so $s(d_\rho) = s(d_{\rho'})$, and s is injective, hence $d_\rho = d_{\rho'}$. □

Theorem 2 (No escalation through programs)**.** *Let $\alpha(\pi) = a \neq \bot$ and let Π_a be Π without its rules of subject image. Then $\mathrm{Dec}_\Pi(r) = \mathrm{Dec}_{\Pi_a}(r)$ for every request $r = (\pi, q, o)$. Consequently no rule written for an ordinary program can admit an agent, or any process it started through unrecognized programs, past an Ask or Block rule.*

Proof. A subject $\mathsf{image}(x)$ applies only to processes with $\alpha = \bot$, so $A_\Pi(r) = A_{\Pi_a}(r)$ and both maxima coincide. The second statement follows from Lemma 1. □

Theorem 2 closes the most natural escape route. Agents act through shells, Python, Node.js and Git, and users write rules for exactly those programs. Without the rank restriction, a rule such as "allow git to read the project" would also open the project to every agent that runs Git.

Theorem 3 (Monotone refinement)**.** *Let Π^+ extend Π by one rule ρ^+ with action d^+. Then $\mathrm{Dec}_{\Pi^+}(r) \in \{\mathrm{Dec}_\Pi(r), d^+\}$ for every request r. In particular, adding a Block rule never makes a decision more permissive, and adding an Allow rule never makes one stricter.*

Proof. If ρ^+ does not apply to r, nothing changes. If it applies and $A_\Pi(r) = \emptyset$, the decision becomes d^+. Otherwise let k^* be the maximal key in $A_\Pi(r)$. If $k_r(\rho^+) < k^*$, the decision is unchanged; if $k_r(\rho^+) > k^*$, it is d^+; if $k_r(\rho^+) = k^*$, Theorem 1 gives $d^+ = \mathrm{Dec}_\Pi(r)$. The second statement holds because Block has the greatest strictness and Allow the least. □

When the user answers a held request $r = (\pi, q, o)$ with a verdict $v \in \{\mathsf{Allow}, \mathsf{Block}\}$ and chooses a scope $p \preceq q$, IOArx forms the rule $\rho_r = (p, sub_r, \{o\}, v)$, where $sub_r = \mathsf{agent}(\alpha(\pi))$ for an agent and image of the program of π otherwise. The rule is kept only if it attains the maximal key for r in Π extended by ρ_r; otherwise the user is shown the rule that would still decide. The class of r under p, written $[r]_p$, holds the requests (π', q', o) with $p \preceq q'$ whose process has the same agent as π, or, if π is no agent, runs the same program and is no agent either.

Theorem 4 (Remembered answers end prompting)**.** *Let ρ_r be kept. Every rule that outranks ρ_r at some $r' \in [r]_p$ has a path strictly longer than p. Hence, if no Ask rule lies strictly inside p, no request of $[r]_p$ is prompted again. If n requests are drawn independently from K classes with probabilities*

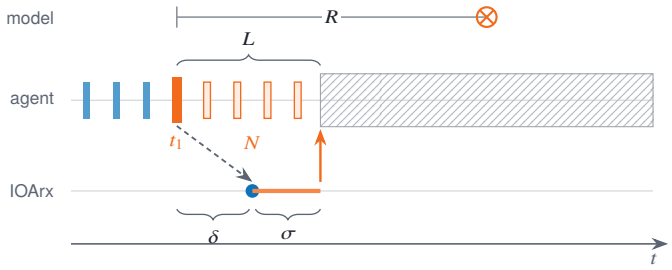


Fig. 2. One violating operation at t_1: delivery and decision (δ), suspension of the process tree (σ), and a model step of length R that never completes.

$w_1, \dots, w_K$ and each class is remembered at its first prompt, the expected number of prompts is at most

$$\sum_{k=1}^{K} \big(1 - (1 - w_k)^n\big) \;\le\; \min(n, K). \tag{1}$$

Proof. Let $r' = (\pi', q', o) \in [r]_p$. The rule ρ_r applies to r' and $k_{r'}(\rho_r) = k_r(\rho_r)$, because the path, the rank of the subject and the action are the same. Suppose a rule ρ applies to r' with $k_{r'}(\rho) > k_{r'}(\rho_r)$ and $|p_\rho| \le |p|$. The first components force $|p_\rho| = |p|$; both paths cover q', so $p_\rho = p$ and ρ covers q. Whether a subject applies, and with which rank, depends only on the agent of a process and, for image, on its program, which π and π' share; and $o \in \Omega_\rho$ or $\Omega_\rho = \emptyset$. Thus ρ applies to r with $k_r(\rho) = k_{r'}(\rho) > k_r(\rho_r)$, contradicting that ρ_r was kept. Any outranking rule therefore lies strictly inside p, and if none of them asks, r' is decided without a prompt. For the bound, a class causes at most one prompt, at its first occurrence, which happens with probability $1-(1-w_k)^n$. Each term is at most one, and by Bernoulli's inequality at most nw_k, whose sum is n. □

A monitor without memory prompts once per access, which is n prompts, whereas (1) grows only with the number of distinct situations. Prompts are a scarce resource, since users habituate to repeated warnings [17], [18], and this difference is what keeps Ask usable as a default.

C. Asynchronous Containment

File events reach IOArx from the kernel after the operation they describe. The guarantee therefore concerns what follows a violating operation: the agent's process tree is held within a measurable time of it, before the agent can act on what it obtained. Figure 2 shows the quantities involved.

Theorem 5 (Containment)**.** *Let an agent's process tree complete file operations at times $t_1 < t_2 < \cdots$, where the operation at t_1 is the first to be decided Ask or Block. Let IOArx decide it at $t_1 + \delta$ and complete the pause or stop of every process of the tree at $t_1 + L$, $L = \delta + \sigma$, after which no process of the tree completes an operation. Let $N = |\{i > 1 : t_i \le t_1 + L\}|$. Then (a) if consecutive operations are at least $1/r$ apart, $N \le \lfloor rL \rfloor$; (b) if the operations after t_1 form a Poisson process of rate λ independent of L, then $\mathbb{E}[N] = \lambda\mathbb{E}[L]$ and $\Pr[N = 0] = \mathbb{E}[e^{-\lambda L}] \ge e^{-\lambda\mathbb{E}[L]}$; and (c) an action that uses data returned at t_1 and is issued only after a model inference that starts no earlier than t_1 and lasts at least R is executed only if $L > R$, hence with probability at most $\Pr[L > R] \le \mathbb{E}[L]/R$.*

Proof. (a) The j-th operation after t_1 completes no earlier than $t_1 + j/r$, so $t_{1+j} \le t_1 + L$ implies $j \le rL$. (b) Given $L = \ell$, N counts the points of a Poisson process of rate λ in an interval of length ℓ, so N is Poisson with mean $\lambda\ell$ and $\Pr[N = 0 \mid L = \ell] = e^{-\lambda\ell}$. Taking expectations gives the first two claims, and Jensen's inequality for the convex map $x \mapsto e^{-\lambda x}$ gives the bound. (c) The action cannot be issued before $t_1 + R$, and from $t_1 + L$ on no process of the tree runs, so it requires $R < L$. Markov's inequality bounds $\Pr[L > R]$. □

Part (c) is the case that prompt injection depends on. An injected instruction must first make the model read a secret, and the stolen value can reach an attacker only through a further action that the model decides on after seeing it [9]. Section VI measures L and N directly.

D. Integrity of the Activity Log

IOArx writes each outcome as a record x_i and stores it with $h_i = H(h_{i-1} \,\|\, \texttt{LF} \,\|\, x_i)$, where H is SHA-256 [19] written as 64 hexadecimal digits, `LF` is a line feed and h_0 is a fixed genesis value, in the spirit of secure audit logs [20]. Because h_{i-1} has a fixed length, $h_{i-1}\|\texttt{LF}\|x_i$ determines both of its parts.

Theorem 6 (Tamper evidence)**.** *Let a verifier retain the number n of records and the head h_n of the log at some moment. If the first n records later read $x'_1, \dots, x'_n \ne x_1, \dots, x_n$ and their recomputed head equals h_n, then a collision of H follows from the two logs with at most n evaluations of H. Without a retained head, a change that leaves the stored hashes of later records in place is located at the first record whose stored hash disagrees with its recomputation.*

Proof. Let $y_i = h_{i-1}\|\texttt{LF}\|x_i$ and $y'_i = h'_{i-1}\|\texttt{LF}\|x'_i$, and let j be the largest index with $x'_j \ne x_j$. Walk down from $i = n$, where $h'_n = h_n$. For $i > j$ we have $x'_i = x_i$: if $h'_{i-1} \ne h_{i-1}$, then $y'_i \ne y_i$ while $H(y'_i) = H(y_i)$, a collision; otherwise $h'_{i-1} = h_{i-1}$ and the walk continues. If it reaches $i = j$ with $h'_j = h_j$, then $y'_j \ne y_j$ because $x'_j \ne x_j$, and again $H(y'_j) = H(y_j)$. Insertions, deletions and reorderings among the first n records change the sequence and are covered. The second claim holds because recomputation reproduces every stored hash up to the first changed record. □

IV. Design of IOArx

The endpoint component and the browser extension of Fig. 1 follow three design rules. Identity is taken from evidence that the operating system vouches for wherever such evidence exists, and every decision records how strong its evidence was. All observation and every decision stay on the device, so no file content, path or prompt is sent anywhere. And uncertainty resolves toward caution: a tie between rules goes to the stricter one, and a question the user leaves unanswered ends in a block.

A. Naming the Agent

A catalog describes each of 28 agents, from command-line coding agents and AI code editors to desktop assistants and local model runtimes, by the evidence it can be recognized by. Evidence comes in two strengths. A publisher in a verified code signature, optionally with the signed product name, and the identity of an installed app package are vouched for by the operating system, so renaming or copying a file does not produce them. A file name, a folder in the program's path or text in its command line also recognize agents that run inside Node.js or Python and carry no signature, but any program can imitate them. When two entries recognize the same program, the narrower evidence names it: an app package before a signed product, then a folder, a file name, a command line, and last a publisher's signature alone, which covers every product of that publisher. Every logged request records which evidence matched and whether it was verified, so the report tells the user how much trust a name deserves.

Agents work through other programs, which is why IOArx resolves names along the process chain (Definition 2). It renews the set of running agents every two seconds and follows their descendants through the kernel's process notifications, so a compiler or shell started by an agent is known as the agent's from its first event. When a short-lived child has already exited by the time its event is handled, the program and parent recorded at its start stand in for the lookup. A fixed list of operating system processes, such as the desktop window manager, the shell and the local security authority, is never paused or stopped, whatever a custom catalog entry matches.

B. Deciding and Containing

Decisions follow Section III. The first run protects Documents, Desktop, Pictures and Downloads with an Ask rule for every agent and leaves the rest to an Allow default, and the user then adds rules in either direction. Independently of the rules, IOArx judges from its path alone whether a target looks like a credential store: SSH keys, the sign-in files of cloud and cluster tools, environment files, password databases, the password and cookie stores of browsers, token files of package managers and Git, private keys, and documents named as holding secrets. Every request made on behalf of an agent and every request for such a file is logged whatever its outcome, and the reports single out agents that were allowed to touch credentials.

An Ask decision pauses the agent together with every process it started and raises an alert with a countdown. Allow resumes the tree, while Block or silence until the deadline, 30 seconds by default, stops it. A Block decision stops the tree at once. Because the same access by the same process within three seconds counts once, a tool that rereads a file in a loop raises one alert, not hundreds. The answer to an alert applies once or becomes a rule for the file or its folder, under the conditions of Theorem 4.

Network control works before the fact. A program on the block list cannot open connections, and in ask mode every program except IOArx itself is refused until the user places it on the allowed list, with traffic inside the computer exempt. A refused connection fails at its first packet, so the gate needs no containment latency at all. Outcomes are written to the hash-chained log of Theorem 6; on Windows the log lives in a folder that only administrators and the system can change, while agents normally run without administrator rights.

C. Inspecting Prompts

The extension watches the prompt box, pastes and file uploads of 19 AI services and checks them before they are sent. Twenty-nine detectors cover cloud and AI service keys, developer platform tokens, payment and messaging credentials, private keys, passwords and generic secrets, card and Social Insurance Numbers, and, if the user enables them, email addresses and phone numbers. Each detector pairs a format with validation: checksums, decoding of a token's structure, required context such as an access key identifier near an AWS secret, a minimum empirical entropy, and the rejection of documentation placeholders and published test numbers. Depending on the policy, a finding produces a warning, a block, or a redacted submission in which each finding is replaced by a typed placeholder. Two forms of validation admit exact analysis.

Proposition 1 (Checksums)**.** *Let $d_1 \dots d_n$ be digits whose last digit is uniform and independent of the others. The Luhn test accepts d with probability exactly $1/10$ given $d_1 \dots d_{n-1}$, and it rejects every string obtained from an accepted one by changing a single digit.*

Proof. The test accepts when $S = \sum_i \varphi_i(d_i) \equiv 0 \pmod{10}$, where φ_i is the identity for every second digit counted from the right, starting with the last, and the doubling map $x \mapsto 2x - 9\lfloor 2x/10 \rfloor$ otherwise. The doubling map permutes $\{0, \dots, 9\}$ as $(0, 2, 4, 6, 8, 1, 3, 5, 7, 9)$. Given the other digits, $S \equiv c + d_n$ for a constant c, which is uniform modulo 10. A change of one digit alters S by $\varphi_i(d_i') - \varphi_i(d_i) \not\equiv 0$, since φ_i is a bijection. □

A random number that already carries a card network's prefix therefore passes as a card with probability one in ten, while no issued card is lost, and the same holds for Social Insurance Numbers. The entropy threshold, in contrast, can reject a genuine credential, and the next statement gives the exact rate at which it does.

Proposition 2 (Entropy thresholds)**.** *Let a credential body consist of n characters drawn independently and uniformly from k symbols, and let $\hat{H} = \log_2 n - \frac{1}{n}\sum_c N_c \log_2 N_c$ be its empirical entropy, where N_c counts symbol c. Then*

$$\Pr[\hat{H} < \tau] = \sum_{\substack{\lambda \vdash n,\ \ell(\lambda) \le k \\ H(\lambda) < \tau}} \frac{n!}{\prod_i \lambda_i!} \cdot \frac{k!}{(k - \ell(\lambda))! \prod_j m_j(\lambda)!} \cdot k^{-n}, \tag{2}$$

where λ ranges over the partitions of n, $\ell(\lambda)$ is the number of parts, $m_j(\lambda)$ the number of parts equal to j, and $H(\lambda)$ the value of $\hat{H}$ for counts λ.

Proof. $\hat{H}$ depends on a string only through the multiset of its counts, a partition λ of n. Strings with counts λ arise by

TABLE I
EXACT MISS RATE OF ENTROPY THRESHOLDS FOR RANDOM CREDENTIALS

Credential body	n	k	τ	$\mathbb{E}[\hat{H}]$	$\Pr[\hat{H} < \tau]$
Hugging Face token	34	62	3.5	4.61	8.9×10^{-11}
AWS secret key	40	64	4.0	4.78	1.3×10^{-7}
`sk-` key, alphanumeric	32	62	3.2	4.55	2.4×10^{-13}
`sk-` key, hexadecimal	32	16	3.2	3.61	3.6×10^{-3}
Generic secret, shortest	16	62	3.0	3.77	6.3×10^{-5}
Twilio auth token	32	16	3.0	3.61	1.1×10^{-4}

choosing which distinct symbols take the parts, in $k!/((k-\ell)!\prod_j m_j!)$ ways, and arranging the positions, in $n!/\prod_i \lambda_i!$ ways. Each string has probability k^{-n}. □

Evaluating (2) over the at most 37,338 partitions involved yields Table I. For alphanumeric bodies the threshold costs nothing measurable; for hexadecimal keys it trades a miss rate of a few in a thousand for the rejection of identifiers and words that happen to fit a key's shape.

V. Implementation

The decision core, the agent catalog and the log are written once in Rust and used unchanged on every platform; only event collection and process control differ. The desktop application combines this core with a web interface through the Tauri framework, and the browser extension is plain JavaScript without dependencies.

On Windows, IOArx needs no kernel driver. With administrator rights it opens a real-time event tracing session that subscribes to two providers built into the kernel, one for file operations and one for process creation, and it discards every event whose process is neither a running agent nor one of its descendants after a single lookup. The session hands over full buffers at once and partially filled ones every second. Pausing uses a native call that suspends all threads of a process together, so a process cannot start a new thread between the suspension of two others, and the tree is paused from its deepest processes upward. Signatures are verified by the operating system's trust provider, including catalog signatures, and network control is built from filters of the Windows Filtering Platform that disappear with the session that created them.

On Linux, a root service collects events for every user's copy of the application. It marks each mounted file system with fanotify in notification mode, with directory entry names so that deletions and renames carry their paths, and it learns of every process start and exit from the kernel's process connector, which keeps short-lived children attributable after they end. Pausing and stopping use the stop and kill signals. While any user asks for network control, an nftables table sends the first packet of every new outbound connection to a netfilter queue, where the service finds the owning process through the socket tables in `/proc` and accepts or rejects the connection. The same core also builds a macOS client on Apple's Endpoint Security framework.

TABLE II
ATTRIBUTION ON THE WINDOWS WORKSTATION

Running processes	
Processes examined	314
Attributed to an agent	36
Claude Code, 4 instances	15
OpenAI Codex, 4 instances	8
Microsoft Copilot	11
GitHub Copilot	2
Attributed through an ancestor	16
Attributed to an unrelated program	0
Program files on disk	
Executables examined	3,449
with a verified signature	1,710
Recognized as an agent	69
by a verified signature	57
by folder evidence	12
Recognized outside an agent product	0
Time	
Process list / signatures of 89 files	37 ms / 365 ms
Attribution of all 314 chains	17.8 ms

The browser extension follows Manifest V3 and is packaged for Chrome, Edge and Firefox. Its content scripts check a prompt at the moment it is about to leave, whether by the Enter key, a send button or a form submission, check text as it is pasted, and check files that are chosen, dropped or pasted before the service receives them, reading text files of up to 2 MB. Each check either lets the submission continue, replaces it with its redacted form, or cancels it. Detection, redaction and the activity history stay in the browser profile.

VI. Evaluation

We evaluate IOArx 1.0.0 on four questions. Does attribution name the right agents on a working machine, and only those? How quickly is a violating agent held, and how much does it accomplish before that? What does a decision cost? How accurate are the prompt detectors on real prompts and real code, and what does each kind of validation contribute?

Attribution and the cost of the core were measured on Windows 11 (build 26200) on an AMD Ryzen 9 9950X3D with 16 cores and 64 GB of memory. Containment was measured with the released Linux packages on Ubuntu 26.04 with kernel 6.18 under WSL 2 on the same machine, with 32 virtual processors and 30 GB of memory. For containment, a stand-in agent, a small native program installed where the native installer of Claude Code keeps its versions, is recognized as Claude Code by folder evidence. It performs file operations in a folder covered by a rule and writes the completion time of each operation into shared memory, which raises no file events. An observer running as root reads that memory and polls the agent's process state every 20 μs, so it sees the moment the agent is paused or stopped.

A. Attribution on a Developer Workstation

The workstation on which this paper was prepared runs Claude Code, OpenAI Codex, GitHub Copilot, Microsoft

Copilot and Ollama in daily use. Table II summarizes one scan of its processes and of its program folders. All 36 attributed processes belong to agents, and every attribution rests on a verified signature. Sixteen of them are not agents themselves but consoles, command interpreters and PowerShell sessions that agents started, which IOArx attributed through their ancestors, the deepest three levels below the agent. The benchmark program that performed the scan was itself attributed to Claude Code, which had launched it through two shells. On disk, 69 of 3,449 executables matched the catalog. Fifty-seven carry the signatures of Anthropic, OpenAI, GitHub, Microsoft or Ollama; the remaining twelve are search tools that the Codex extension ships inside its own folder and that it runs on the agent's behalf. No program outside an agent product was recognized.

B. Containment

We measure containment on Linux, where fanotify delivers each event as it happens; on Windows, δ also contains the flush interval of the trace session (Section V). In 30 trials the stand-in agent deleted files as fast as it could, 100,000 per second at the median, in a folder under a Block rule; in 30 more it created and wrote files at 58,000 per second; and in 30 it read files at 219,000 per second under an Ask rule. IOArx held the agent in every trial (Fig. 3a). A Block stopped the deleting agent within a median of 3.1 ms after its first violating operation and within 3.6 ms in all 30 trials, and stopped the writing agent within a median of 3.2 ms. An Ask paused the reading agent within a median of 4.0 ms. Every operation that completes before the pause becomes a held request of its own, and after a burst of several hundred reads this backlog delayed the following decisions, so that the 90th percentile reached 9.3 ms and one of the 30 trials took 1.13 s. A paused agent stayed paused until the alert's deadline, two seconds in this experiment, and was then stopped.

Without a rate limit, the agent completed a median of 295 further deletions, 171 writes or 849 reads before it was held, as its speed and L imply. Agents driven by a model act far more slowly, because each step waits for an inference. Limiting the stand-in to 100, 1,000 and 10,000 operations per second gave the counts of Fig. 3b, which follow $r\,\mathbb{E}[L]$ over more than three orders of magnitude. At 100 operations per second, one deletion followed the violating operation in one of 20 trials and none did in the other 19, and no read followed it in any trial. In 118 of the 120 rate-limited trials, N stayed within $\lfloor rL \rfloor$; in the remaining two it exceeded that bound by one, because the agent's fixed schedule let two operations follow each other faster than $1/r$ after a short delay.

These latencies settle part (c) of Theorem 5. On production-sized coding requests, the median end-to-end latency of a single inference of a 70-billion-parameter model is 493 ms on H100 and 856 ms on A100 accelerators [4]. All 120 Block trials and 89 of the 90 Ask trials held the agent within 16 ms, more than thirty times faster than even the faster of these inferences, so by part (c) no action that depends on what the violating operation returned could have been issued in those trials.

TABLE III
Prompt detectors with validation switched off in turn

	Recall	Numbers	Placeh.	Dummies	Code
All on	99.65%	214	0	1,196	992
No checksums	99.65%	2,291	0	1,196	1,830
No entropy	99.66%	214	0	2,000	992
No placeholders	99.65%	214	726	1,196	994
None	99.66%	2,291	726	2,000	1,832

Recall over 14,500 planted values. Numbers: 4,000 random order and grouped numbers; Placeh.: 2,000 key placeholders; Dummies: 2,000 repetitive dummy values; Code: findings in 322 MB of source code.

C. Cost of a Decision

Mediation sits on the path of every event an agent causes. A policy decision with 16 rules takes 0.4 μs at the median, and the scan over rules grows linearly at about 32 ns per rule, to 24 μs with 1,024 rules (Fig. 3c). Naming the agent of a process chain costs about 7 μs per program with the 28 agents of the catalog, 26 μs for a chain of four programs, the longest that ended in an agent on the workstation (Fig. 3d). The complete decision for one event of an agent, covering attribution, the classification of the target, the policy, the handling of held requests and the append to the log, took 19.2 μs at the median and 43.5 μs at the 99th percentile over 50,000 events; an event of an ordinary program took 10.0 μs. Within that, linking a typical 617-byte record into the hash chain takes 0.32 μs and appending it 2.5 μs, or 0.46 ms when it is forced to disk. Verifying a log of one million records, 709 MB in size, took 2.3 s, and after one byte of record 500,000 was changed the verification reported exactly that record, as Theorem 6 predicts.

D. Prompt Inspection

We planted 500 values of each of the 29 detected kinds, generated in their documented formats with random bodies, into prompts drawn from the 32,771 distinct user prompts of the OpenAssistant Conversations dataset [21], at the end of the prompt or between its paragraphs and framed as a configuration line, a code block or running text. The prompts themselves served as real negatives. Hard negatives came from 16 classes of 2,000 values each that resemble secrets without being any: UUIDs, Git and SHA-256 digests, Base64 digests and data URIs, package integrity strings, random order numbers and grouped nine-digit numbers, hexadecimal request identifiers, IP addresses, timestamps, version strings, numeric settings, documentation placeholders for keys and passwords, and secret names assigned repetitive dummy values. Real code came from 21,465 source files, 322 MB, of 514 Rust crates and of the npm packages of the desktop interface.

With every detector on by default, IOArx found 14,449 of the 14,500 planted values, 99.65%. Twenty-six detectors found all 500 of their values; the exceptions were passwords in settings and code (93.4%), passwords inside connection URLs

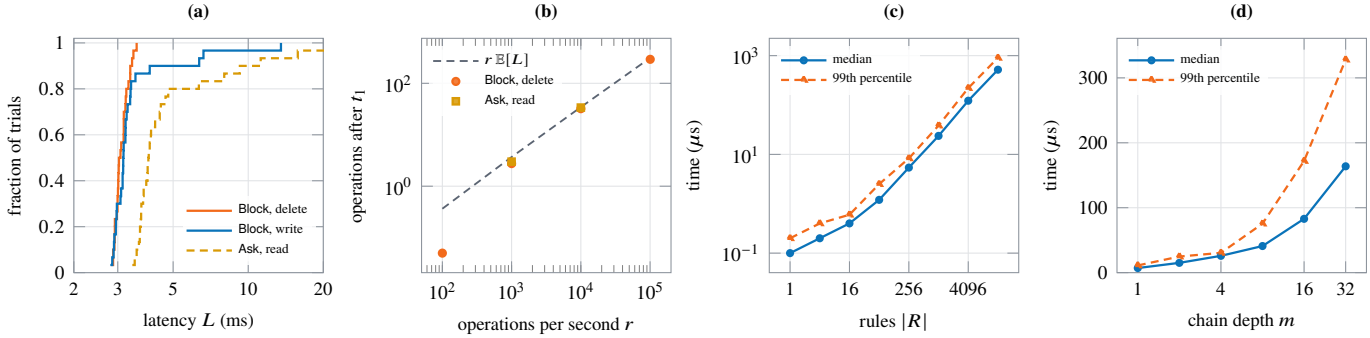


Fig. 3. (a) Time from the violating operation to the held agent. (b) Operations completed after it, against the agent's rate, with the bound of Theorem 5. (c) Cost of one policy decision. (d) Cost of naming the agent of a process chain.

(96.6%) and hexadecimal `sk-` keys (99.8%), where the single miss matches the rate of Table I. The real prompts produced no finding at all. Turning on the contact detectors, which are off by default, added 19 findings in six prompts, every one a well-formed email address or phone number in sample data that users had pasted. Thirteen of the 16 hard negative classes raised nothing.

Table III isolates the validations. None of them costs recall, since switching all of them off gains a single planted value out of 14,500. The checksums account for the largest single effect: random order numbers that carry a card network's prefix pass as cards 9.5 times as often without the Luhn test, and grouped nine-digit numbers pass as Social Insurance Numbers 11.3 times as often, close to the factor of ten that Proposition 1 predicts. Over two million random 16-digit strings, the fraction with a valid prefix that also passed the test was 0.0999. The placeholder filter alone removes every documentation placeholder, 726 of which would otherwise be reported, and the entropy threshold removes 40% of the repetitive dummy values. On source code, which the detectors meet when users paste it into a prompt, validation nearly halves the findings, to 3.1 per MB. Most of the remaining ones lie in generated operating system bindings with fields named *password*, in tables of large numeric constants, and in the test data of URL parsers, which embed credentials in URLs on purpose.

Scanning adds little to a submission. In Node.js 26 on the workstation, the detectors took 0.02 ms for a prompt of 1,000 characters, 0.25 ms for 10,000, 2.8 ms for 100,000 and 38.5 ms for one million.

VII. Related Work

IOArx draws on four lines of work: runtime enforcement, defenses built into agent frameworks, endpoint detection, and the study of how people answer security prompts.

Runtime enforcement. Complete mediation and least privilege [8] frame the problem, and the theory of execution monitors explains its shape. Monitors that observe a single execution can enforce only safety properties [22], and edit automata separate monitors that may only halt the target, called truncation automata, from those that may also suppress or insert actions [23]. The file path of IOArx is a truncation monitor that acts after a delay: it halts the agent's process tree once a violating prefix is observed, and Theorem 5 bounds the suffix that runs before the halt takes effect. The egress gate is a classical in-line monitor that suppresses the forbidden action itself.

Defenses inside agent systems. IsolateGPT isolates the applications of an LLM-based system from one another and keeps its overhead below 30% for 75.73% of the queries evaluated [1]. CaMeL separates the control flow of an agent from untrusted data and completes 77% of the AgentDojo tasks with provable security, against 84% without the defense [2], and AgentDojo itself measures how far simpler defenses such as tool filtering reduce attack success [11]. These designs act inside the agent and require each vendor to adopt them. IOArx acts at the operating system, below every agent, and applies one policy to agents whose internals it cannot see. The two layers compose: a framework defense lowers the rate of violations, and IOArx limits what any remaining violation can reach.

Endpoint detection and provenance. Dependency graphs over process and file events trace an intrusion back to its entry point [24], and provenance-based triage of the alerts of threat detection software is estimated to cut their false alarms by 84% [3]. Those systems reason after the fact over every process of a machine. IOArx needs only the lineage of the agents it knows, which makes attribution cheap enough to sit on the decision path of every event (Section VI).

Prompts and attention. Runtime permission prompts on mobile platforms let users decide in context, and a model of each user's earlier decisions predicted their privacy decisions 96.8% of the time, a fourfold reduction in error compared to the systems then deployed [25]. Attention is scarce and each needless interruption has a cost [17], and repeated warnings fade from perception after only a few exposures [18]. IOArx answers with scoped remembered rules whose number of prompts Theorem 4 bounds by the number of distinct situations rather than by the number of accesses.

Secrets and logs. Large-scale studies detect leaked credentials by their formats and entropy [13], and benchmarks label candidate secrets so that detectors can be compared [14]. The detectors of IOArx follow the same principle but add

checksums, decoding and context whose effect Propositions 1 and 2 make exact. The activity log follows the hash-chained design of secure audit logs [20].

VIII. Conclusion

Operating systems decide by user, and AI agents act as the user. IOArx closes that gap at the endpoint by giving the agent a name that follows it through every program it starts, a policy whose outcome does not depend on how its rules were written, and a containment path that holds a violating agent before its model can act on what it obtained, while the network and the browser are guarded before data leaves. The formal model turns these properties into theorems, and the measurements confirm them on real agents and real prompts: attribution without a misattribution, containment within milliseconds on Linux, decisions in microseconds and prompt inspection without false alarms on tens of thousands of real prompts. As agents take on more of the work done at a keyboard, the question of which program acts for whom becomes part of access control itself, and IOArx shows that the endpoint can answer it for every agent at once.

Acknowledgment

AI tools were used to polish the text, correct grammar, improve readability, and debug code.

References

[1] Y. Wu, F. Roesner, T. Kohno, N. Zhang, and U. Iqbal, "IsolateGPT: An execution isolation architecture for LLM-based agentic systems," in *Proceedings 2025 Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA: Internet Society, 2025, doi: 10.14722/ndss.2025.241131.

[2] E. Debenedetti *et al.*, "Defeating prompt injections by design," in *2026 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, Mar. 2026, pp. 587–618, doi: 10.1109/SaTML68715.2026.00040.

[3] W. U. Hassan *et al.*, "NoDoze: Combatting threat alert fatigue with automated provenance triage," in *Proceedings 2019 Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA: Internet Society, 2019, doi: 10.14722/ndss.2019.23349.

[4] P. Patel *et al.*, "Splitwise: Efficient generative LLM inference using phase splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2024, pp. 118–132, doi: 10.1109/ISCA59077.2024.00019.

[5] S. Yao *et al.*, "ReAct: Synergizing reasoning and acting in language models," in *The Eleventh International Conference on Learning Representations (ICLR 2023)*, 2023, doi: 10.48550/arXiv.2210.03629.

[6] J. Yang *et al.*, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024, pp. 50 528–50 652, doi: 10.52202/079017-1601.

[7] X. Hou *et al.*, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, Nov. 2024, art. no. 220, doi: 10.1145/3695988.

[8] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975, doi: 10.1109/PROC.1975.9939.

[9] 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)*. Copenhagen, Denmark: ACM, Nov. 2023, pp. 79–90, doi: 10.1145/3605764.3623985.

[10] Q. Zhan, Z. Liang, Z. Ying, and D. Kang, "InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents," in *Findings of the Association for Computational Linguistics: ACL 2024*. Association for Computational Linguistics, 2024, pp. 10 471–10 506, doi: 10.18653/v1/2024.findings-acl.624.

[11] E. Debenedetti, J. Zhang, M. Balunović, L. Beurer-Kellner, M. Fischer, and F. Tramèr, "AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents," in *Advances in Neural Information Processing Systems 37 (NeurIPS 2024), Datasets and Benchmarks Track*, 2024, pp. 82 895–82 920, doi: 10.52202/079017-2636.

[12] Y. Ruan *et al.*, "Identifying the risks of LM agents with an LM-emulated sandbox," in *The Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024, doi: 10.48550/arXiv.2309.15817.

[13] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? Characterizing secret leakage in public GitHub repositories," in *Proceedings 2019 Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA: Internet Society, 2019, doi: 10.14722/ndss.2019.23418.

[14] S. K. Basak, L. Neil, B. Reaves, and L. Williams, "SecretBench: A dataset of software secrets," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, May 2023, pp. 347–351, doi: 10.1109/MSR59073.2023.00053.

[15] Z. Zhang *et al.*, "'It's a fair game', or is it? Examining how users navigate disclosure risks and benefits when using LLM-based conversational agents," in *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*. Honolulu, HI, USA: ACM, May 2024, pp. 1–26, doi: 10.1145/3613904.3642385.

[16] N. Lukas, A. Salem, R. Sim, S. Tople, L. Wutschitz, and S. Zanella-Béguelin, "Analyzing leakage of personally identifiable information in language models," in *2023 IEEE Symposium on Security and Privacy (SP)*, May 2023, pp. 346–363, doi: 10.1109/SP46215.2023.10179300.

[17] R. Böhme and J. Grossklags, "The security cost of cheap user interaction," in *Proceedings of the 2011 New Security Paradigms Workshop (NSPW '11)*. Marin County, CA, USA: ACM, Sep. 2011, pp. 67–82, doi: 10.1145/2073276.2073284.

[18] B. B. Anderson, C. B. Kirwan, J. L. Jenkins, D. Eargle, S. Howard, and A. Vance, "How polymorphic warnings reduce habituation in the brain: Insights from an fMRI study," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. Seoul, Republic of Korea: ACM, Apr. 2015, pp. 2883–2892, doi: 10.1145/2702123.2702322.

[19] National Institute of Standards and Technology, "Secure hash standard (SHS)," National Institute of Standards and Technology, Federal Information Processing Standards Publication 180-4, Aug. 2015, doi: 10.6028/NIST.FIPS.180-4.

[20] B. Schneier and J. Kelsey, "Secure audit logs to support computer forensics," *ACM Transactions on Information and System Security*, vol. 2, no. 2, pp. 159–176, May 1999, doi: 10.1145/317087.317089.

[21] A. Köpf *et al.*, "OpenAssistant Conversations: Democratizing large language model alignment," in *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, 2023, pp. 47 669–47 681, doi: 10.52202/075280-2064.

[22] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, Feb. 2000, doi: 10.1145/353323.353382.

[23] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: Enforcement mechanisms for run-time security policies," *International Journal of Information Security*, vol. 4, no. 1–2, pp. 2–16, Feb. 2005, doi: 10.1007/s10207-004-0046-8.

[24] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. Bolton Landing, NY, USA: ACM, Oct. 2003, pp. 223–236, doi: 10.1145/945445.945467.

[25] P. Wijesekera *et al.*, "The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences," in *2017 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, May 2017, pp. 1077–1093, doi: 10.1109/SP.2017.51.