Postern: a Lean-verified access gateway for agentic data lakehouses

Secure Program Synthesis Hackathon 2026 — Track 3 research artifact

true

Abstract

We address access control for data lakehouses queried by LLM agents. An agent's effective rights are context-driven — which principal it acts for, which task it is invoked under, which scope the caller granted — and the static *identity* → *role* → *permission* chain of RBAC cannot encode any of those axes. Per-engine row- and column-level security does not survive the ETL boundary; physical tenant segregation forfeits the cross-source joins that motivate the lakehouse. We propose plan-level rewriting against a **Biscuit-Datalog** policy (Clever Cloud and contributors 2024) and present **Postern**, an artifact in three parts: (i) a rewriter rewrite : *Catalog* → *Policy* → *Principal* → *Plan* → *Option Plan* mechanised in Lean~4 (Moura and Ullrich 2021), inspired by Cedar's Lean authorization core (Cutler et al. 2024) but stated over plan-level outputs rather than per-request decisions, with nine `sorry`-free theorems (axioms bounded by `propext` and `Quot.sound`) plus a partly-mechanised Horn-fragment Datalog evaluator; (ii) a Rust capability-tracking layer inspired by Odersky et al. (Odersky et al. 2026) — invariant brand lifetimes, sealed types, opaque-receipt sinks; and (iii) a reference-conformance harness binding Rust to the Lean reference on 18 cases. We evaluate on the Kaggle `transactions-fraud-datasets` schema and identify Biscuit attenuation, audience, expiry, key rotation, cross-relation joins, and differentially-private aggregation as the principal open problems.

Introduction

Production agentic systems increasingly read context from data-lakehouse stores at inference time, compacting heterogeneous sources into engine-agnostic columnar formats — Apache Arrow Parquet on S3-class object storage — queried by in-process engines such as DuckDB (Raasveldt and Mühleisen 2019), often co-located with on-prem or edge runtimes hosting local models. The access-control consequence is sharp: the entity issuing queries is no longer a human analyst on a stable role but an LLM agent mediated by tool-call protocols such

as the Model Context Protocol (Anthropic 2024). Agent access is short-term, contextual, and task-oriented; role-based access control — and its temporally-scoped variant TRBAC — does not fit. Two existing responses each concede something. Per-engine row- and column-level security (PostgreSQL Global Development Group 2024) does not survive ETL into an engine-agnostic Parquet store, forcing per-source policy duplication that scales poorly with source count and churn. Physical tenant segregation across object-storage prefixes queried by disjoint engines restores safety but forfeits the cross-source joins that motivate the lakehouse in the first place. Neither addresses the deeper mismatch: an agent's effective permissions depend on the principal it acts for, the task it is invoked under, and the calling context — none of which the static *identity* → *role* → *permission* chain of RBAC encodes. The same agent code called by two principals, or invoked by one principal under two tasks, may legitimately need two different views. We propose plan-level rewriting against a column-grant policy as a third point in this design space, gated under the agent's task-scoped identity at query time. **Postern**, the artifact we present, has three components: a Lean~4-mechanised plan rewriter, a Rust capability-tracking layer constraining the agent's downstream computation, and a reference-conformance harness binding the two.

Contributions

1. A Plan IR (*Scan*/*Project*/*Filter*), a Biscuit-Datalog policy language (Horn-fragment with ground `right(p, r, c)` facts compiled from the surface column-grant syntax), and a rewriter, mechanised in Lean~4 (Moura and Ullrich 2021). The rewriter is inspired by Cedar's Lean-mechanised authorization core (Cutler et al. 2024); we transpose the technique from per-request *authorize* decisions to plan-level outputs over an IR, and so prove soundness of a *transformation* rather than a classification. The rewriter side comprises nine `sorry`-free theorems: output-column soundness, filter-predicate soundness, schema subset, no-new-columns, idempotence under repeated application, monotonicity in the policy, touched-relation preservation, and explicit-refusal lemmas for unknown relations and for filter predicates over forbidden columns (§4). The Datalog evaluator (`verifier/lean/Datalog.lean`) contributes a further `eval_monotone` theorem (proved modulo one isolated combinatorial obligation, `herbrandBound_mono`) plus four `sorry`-free specialisation lemmas covering the rule-free regime that all our scenarios use today. Two further evaluator meta-theorems — `eval_sound` and `eval_terminates` — are stated with `sorryAx` obligations named explicitly in `CheckAxioms.lean`. Axiom dependencies for proved declarations are bounded by `propext` and `Quot.sound`; two rewriter theorems depend on none.

2. A Rust capability-tracking layer (`postern-guardrail`) implementing three composable mechanisms: sealed `Cap<'sc, C>` tokens whose construction is private to the crate, an invariant brand lifetime `'sc`

enforced by `PhantomData<fn(&'sc ()) -> &'sc ()>` and gated by a universally-quantified scope combinator, and opaque-receipt sinks that consume both the `Cap` and the carrier `Tagged` without exposing the underlying value. Three lexical bypass attempts (forging `Cap`, projecting `Tagged::value`, escaping the brand) are pinned as `compile_fail` doctests. The agent-facing surface is `no_std`-compatible (§3).

3. A Rust implementation of the rewriter (`postern-core`) structurally mirroring the Lean reference, and a reference-conformance harness (`postern-diff`) that asserts byte-equivalence between the Rust output and the Lean reference on a corpus of 18 hand-curated cases (15 accept, 3 refusal). The gateway's Datalog evaluation surface is designed around `biscuit-auth`'s public `biscuit_auth::datalog::World` evaluator (Clever Cloud and contributors 2024); the in-tree `postern-core` migration to that backend is in progress, with the column-grant DSL serving as the byte-equivalent stand-in until the migration lands. We label the procedure *reference-conformance testing* rather than QuickCheck-style differential testing, reserving the latter term for property-based generation; the latter is among the open problems of §6.

4. A case study over the Kaggle `transactions-fraud-datasets` schema, with three principals (CRM, Card Operations, Fraud Risk) exercising PII redaction, cross-departmental refusal, and minimum-necessary disclosure (§5). Each row of the evaluation table corresponds to a corpus case driving the conformance harness.

Plan IR

We state the IR before the threat model so §2 may reference its operators directly. Plans are single-relation expressions built from three constructors:

$$\mathit{Plan} \ ::= \ \mathit{Scan}(r) \mid \mathit{Project}(\mathit{Plan}, cs) \mid \mathit{Filter}(\mathit{Plan}, c)$$

where $r \in \mathit{Relation}$, $c \in \mathit{Column}$, and $cs \in \mathit{List\ Column}$. We write $\sigma(q)$ for the output schema of plan q under a catalog *cat*, defined inductively: $\sigma(\mathit{Scan}(r)) = \mathit{cat}(r)$; $\sigma(\mathit{Project}(p, cs)) = \sigma(p) \cap cs$; $\sigma(\mathit{Filter}(p, c)) = \sigma(p)$. The asymmetry between *Project* (which alters the output schema) and *Filter* (which does not, despite reading c) is the source of the *filter side-channel* discussed in §2.

Threat model

The trusted computing base (TCB) consists of the gateway process itself, the catalog *cat* that it consults, the plan-to-executor lowering step that hands the rewritten plan to DuckDB, the principal extraction step that maps a verified capability token to a *Principal* string, and the DuckDB + Parquet store. All other parties are untrusted.

Component	Trust	Justification
LLM / agent planner	×	Susceptible to direct and indirect prompt injection.
Agent-generated code	×	Composes upstream-tainted context with downstream effects.
Capability tokens (Clever Cloud and contributors 2024)	~	Trust contingent on the gateway's signature verification.
Gateway process	✓	Hosts the Lean-verified rewriter and the policy.
Catalog ($r \mapsto$ columns)	✓	Assumed bound to the physical Parquet schema (§6).
Plan-to-executor lowering	✓	The rewritten plan is assumed honoured literally by DuckDB.
Principal-string extraction	✓	A bug in token verification invalidates every theorem of §4.
DuckDB + Parquet store	✓	Standard storage-engine assumptions apply.

The attacks within scope of the formal model are: (i) over- projection of forbidden columns; (ii) *reference* to a forbidden column inside a filter predicate, addressed by `rewrite_filter_sound` — the *Filter* constructor carries only a column name (no predicate value or operator), so the theorem rules out the principal *naming* a forbidden column in a filter, not value-probing exfiltration through allowed columns (`WHERE region = 'EU'` followed by row-count observation), which is out of scope (§6); (iii) scan of a relation absent from the catalog, addressed by `rewrite_refuses_unknown`; (iv) cross- departmental reach by a principal lacking matching grants; and (v) unknown principals, which we *collapse to an empty output schema* (not refusal) via the empty-allow convention — the rewriter returns *some* (*Project q* []), so the executor receives a syntactically valid plan that releases zero columns. A genuinely fail-closed variant that refuses unknown principals is a §6 follow-up.

The following are deliberately out of scope and discussed in §6: value-based predicate side-channels through allowed columns; aggregation and inference attacks; covert channels through latency or row-count observation; multi-relation joins; biscuit attenuation modelled inside the Lean proof; policy synthesis from natural language; and the planner-to-executor lowering step.

Design

Postern compiles a single policy artifact to plan-level enforcement.

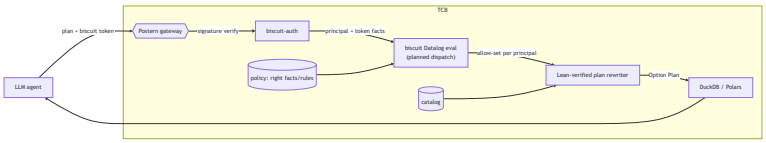

Figure 1: Postern architecture. The agent submits a plan paired with a biscuit token. Inside the trusted base, biscuit-auth performs Ed25519 signature verification, and then the same library's Datalog evaluator (`biscuit_auth::datalog::World`) combines the token's authenticated facts with the gateway-loaded policy's `right(principal, relation, column)` rules to derive the principal's allow-set. The Lean-verified plan rewriter then projects the submitted plan against that allow-set and the catalog, emitting an `Option Plan` for DuckDB/Polars to execute. The dashed box marks the trusted computing base of §2. The "biscuit Datalog eval" node is labelled *planned*: `postern-diff` today calls `postern_core::rewrite` directly against a column-grant `Policy`, and the second conformance corpus exercising `biscuit_auth::datalog::World` is the queued follow-up (§5).

Policy

A policy is a list of **column-grants** $\langle p, r, C \rangle$: "principal p may read columns C on relation r". Multiple grants for the same (p, r) flat-union. Anything outside the union is denied — fail-closed. **No deny-lists**: the policy language is deliberately monotone grant-only, which makes policy review additive (a new grant can only widen). Deny-lists and attribute-based predicates are §6.

A concrete policy from the financial-institution scenario of §5, in Postern's surface syntax:

```
grant CRM       on users_data        { id, name, region, age }
grant CardOps   on cards_data        { card_id, card_type, limit, activated }
grant FraudRisk on transactions_data { txn_id, card_id, amount, merchant, timestamp }
grant FraudRisk on users_data        { id, region }
```

Anything outside these grants is implicitly denied; `users_data.ssn`, `users_data.email`, and `cards_data.card_number` are never released regardless of which principal queries. The rewriter projects each plan's output schema down to the grant union under the querying principal and refuses plans whose filter predicates touch columns outside that union (§4).

The surface syntax shown above is *illustrative*: the artifact does not ship a parser for `.postern` files. Every catalog and policy exercised by the Lean reference (`Demo.cat`, `Demo.pol`) and the Rust mirror (`postern_core::demo_policy`) is constructed directly as a list of `Grant` records. A parser is small enough to add but is not on the publication path; the `scenarios/financial-institution/policy.postern` file is human-readable documentation of the same `Grant` list.

Rewriter

```
rewrite cat P prin q :=
  if cat q.touched = [] then     none                          -- unknown relation
  else if ¬ q.filterCols ⊆ allow then  none                    -- forbidden filter col
  else
    some (Project q (q.schema cat ∩ allow))
  where allow := P.allowed prin q.touched
```

Post-hoc projection is the simplest algorithm that admits a clean soundness proof. Predicate-pushdown variants can be verified against this rewriter as a reference; we leave that to future work.

Static reference, dynamic gateway

A confusion the artifact shape invites is whether Postern requires the lake to be batched or otherwise frozen. It does not. Every input to the rewriter is bound at *request* time: the plan q submitted by the agent over MCP, the principal p extracted from the verified biscuit token on that request, and the catalog snapshot *cat* that the gateway consults at evaluation. The rewriter is a pure function of

(cat, P, p, q) and never reads a row — only relation and column *names*. Underlying Parquet data may therefore be mutable, growing, partitioned, or remote; the lake can be queried online by long-running agent sessions, and a deployment may issue arbitrarily many distinct plans per principal without any pre-registration step. The column-grant policy P is the only artifact that must be loaded into the gateway ahead of time; live policy reload is supported operationally, and Theorem 6 (monotonicity) bounds the safety direction of an in-flight change — strengthening the policy may widen the released set but cannot grant a column the previous policy did not.

The Lean-side `lake exe postern-corpus` is a *build-time* conformance tool: it emits a JSON corpus of (*input*, *expected output*) pairs from the Lean reference rewriter, which `postern-diff` then asserts the Rust implementation honours byte-for-byte (§5). The corpus is not in the request path — the production gateway runs the structurally-identical Rust mirror of the Lean function. The runtime constraints that *do* apply are about plan *shape* and *catalog truthfulness*, not data motion: the Plan IR is single-relation (Scan/Project/Filter), so cross-relation joins, aggregations, window functions, and recursive CTEs are deliberately out of scope (§6 itemises each); the gateway is trusted to consult a catalog snapshot that faithfully describes the physical Parquet schema at evaluation time, and catalog drift between the snapshot and the store is itself an open problem listed in §6.

Cost at the gateway

Online enforcement is cheap by construction. The rewriter is a single bottom-up traversal of the Plan tree: $O(|q|)$ node visits, each performing one catalog hash-lookup for a *Scan*, one set-membership check for a *Filter*, and one column-set intersection for a *Project*. With the natural index $P : (\mathit{Principal}, \mathit{Relation}) \mapsto \mathit{Set\ Column}$, the per-node policy lookup is $O(1)$ amortised, and the dominant per-request cost is the biscuit Ed25519 verification done once at the front of the gateway — sub-millisecond on commodity hardware (Clever Cloud and contributors 2024). *Nothing in the rewrite path reads rows*: the gateway returns an *Option Plan* from inputs that fit in a few kilobytes, independent of lake size. The same algorithm runs in the browser-side WASM build powering §5's demo. Two practical consequences follow. First, the policy gate is comfortably online for interactive agent loops: the round trip is dominated by the downstream DuckDB/Polars execution and by network latency to the lake, not by the rewrite itself. Second, because the rewriter emits an explicit *Project* narrowing the output schema to the policy-allowed columns, downstream Parquet column-pruning (Raasveldt and Mühleisen 2019) often makes the rewritten plan strictly *cheaper* to execute than the original — the policy gate can reduce I/O rather than impose it. This contrasts with row-level alternatives: per-engine RLS (PostgreSQL Global Development Group 2024) evaluates a predicate per row, scaling with table size; an external policy-decision point such as OPA (Cloud Native Computing Foundation 2021) adds a network round-trip per query. Postern pays

neither cost.

Capability-bounded data flow

The rewriter of §3 (Layer 1) constrains what data reaches the agent. A separate layer (henceforth *Layer 2*) constrains what the agent's code may do with the values released. Odersky et al. (Odersky et al. 2026) propose Scala 3 capture-checking as a type-level mechanism for this purpose: capabilities are first-class program variables, and the compiler tracks each function's *capture set* — the capabilities it may use — so that agent-generated code cannot perform an effect for which it does not hold a capability. Rust has no capture-checking. We mechanise a weaker analog by composing three pure-Rust constructions, each closing one face of the gap.

Sealed capability tokens. `Cap<'sc, C>` carries no public constructor and contains a `Sealed` field whose constructor is private to the crate. Forging a `Cap` is therefore a privacy violation rejected at compile time (verified by a `compile_fail` doctest in the crate). Carrier values `Tagged<'sc, T, C>` likewise have a private `value` field; the inner T is unreachable by direct field access (also verified by `compile_fail`).

Invariant brand lifetime. Both `Cap` and `Tagged` carry a brand parameter $'sc$, made invariant by `PhantomData<fn(&'sc ()) -> &'sc ()>`. The sole entry point to the layer is a scope combinator

`run<T, C, R, F>(value, f) -> R` where `F: for<'sc> FnOnce(Cap<'sc, C>, Tagged<'sc, T, C>) ->`

The universal quantification of $'sc$ together with $R : 'static$ forces the closure's return type to be free of $'sc$, ruling out any path by which a `Cap` or `Tagged` might escape the scope. The construction is the same one used by `ghost-cell` for branded references. A third `compile_fail` doctest demonstrates that attempting to return the `Cap` from the closure is rejected.

Opaque-receipt sinks. Carrier extraction is governed by a fixed set of sink functions, each of which consumes both `Cap<'sc, C>` and `Tagged<'sc, T, C>` and returns a *receipt* type that contains no information derived from T beyond its serialised length:

```
pub fn to_llm<'sc, T, C, S>(cap: Cap<'sc, C>,
                            data: Tagged<'sc, T, C>,
                            serialize: S) -> LlmAck
  where S: FnOnce(T) -> String;
```

The agent has no public method that returns raw T; the prior draft's `Tagged::release(cap) -> T` is removed in favour of the sink interface above. The only operations on *Tagged* that the agent's code can perform are `map` and `and_then`, both of which preserve the brand and the kind.

The agent-facing surface is `#![no_std]`. The crate is partitioned so that the gateway-side integration with `postern-core` lives behind a `gateway` feature flag;

the agent-facing types and operations (`Cap`, `Tagged`, `run`, `sinks`) depend only on `core` and `alloc`. A downstream agent crate declaring `#![no_std]` and depending on `postern-guardrail` with `default-features = false` therefore has no link-level access to `std::println!`, `std::process::exit`, network sockets, or filesystem APIs, and the only side-effect channels available are those reached through the sanctioned sinks.

Residual. Inside a `map` closure body the agent has temporary access to a value of type T; Rust does not bound what that body may do with the value. Three classes of side-channel survive: `panic!` with formatted strings, timing observed by the host, and stash in thread-local storage when $T :' \mathit{static}$. We mitigate the obvious thread-move attack by making both `Cap` and `Tagged !Send + !Sync` via the `*const ()` phantom, but a determined adversary inside the agent runtime is out of scope for the present construction. A genuinely tight analog of capture-checking in Rust appears to require either a custom lint over closure bodies or a Wasm-class sandbox; both directions are discussed in §6.

Formal model

The development is mechanised in `verifier/lean/Postern.lean`; the per-theorem axiom set is reported by `CheckAxioms.lean`. We write *rewrite cat P p q* for the rewriter applied to catalog *cat*, policy P, principal p, and plan q. The output type is *Option Plan*; *none* denotes explicit refusal.

Theorem 1 (output-column soundness, `rewrite_sound`). For every $\mathit{cat}, P, p, q, q'$, if *rewrite cat P p q* = *some* q', then for every column $c \in \sigma(q')$, $c \in P.\mathit{allowed}\ p\ \mathit{touched}(q)$.

Theorem 2 (filter-predicate soundness, `rewrite_filter_sound`). Under the same hypothesis, every column read by a *Filter* predicate inside q' is also in $P.\mathit{allowed}\ p\ \mathit{touched}(q)$. Theorems 1 and 2 together rule out the side-channel in which a principal lacking read access to column c uses it as a row selector without projecting it.

Theorems 3 and 4 (`rewrite_schema_subset`, `rewrite_no_new_columns`). The output schema is contained in the input schema. Equivalently, $c \notin \sigma(q)$ implies $c \notin \sigma(q')$.

Theorem 5 (idempotence, `rewrite_idempotent`). If *rewrite cat P p q* = *some* q' and *rewrite cat P p* q' = *some* q'', then $\sigma(q'') = \sigma(q')$ as sets. The rewriter is a closure operator on schemas.

Theorem 6 (monotonicity in the policy, `rewrite_monotone`). If $P.\mathit{allowed}\ p\ r \subseteq P'.\mathit{allowed}\ p\ r$ for every p and r, then the output schema under P is contained in the output schema under P'. Strengthening the policy can only widen the released set.

Theorem 7 (touched-relation preservation, `rewrite_touched`).

touched$(q') = $ *touched*(q).

Theorem 8 (refusal under unknown relation, `rewrite_refuses_unknown`). *cat touched*$(q) = []$ implies *rewrite cat P p q* = *none*. A relation absent from the catalog is rejected explicitly rather than reduced to an empty output schema — relevant under catalog drift, where the physical store may diverge from the catalog.

Theorem 9 (refusal under forbidden filter, `rewrite_refuses_forbidden_filter`). If $c \in$ *filterCols*(q) and $c \notin P.$*allowed p touched*(q), then *rewrite cat P p q* = *none*. The contrapositive of Theorem 2.

`CheckAxioms.lean` audits the axiom dependencies of each theorem. For the rewriter side (Theorems 1–9) the set is bounded by {`propext`, `Quot.sound`}, Lean~4's foundational axioms; the proofs of `rewrite_touched` and `rewrite_refuses_unknown` depend on no axioms, and no proof uses `sorry`.

Datalog evaluator (`verifier/lean/Datalog.lean`). The policy language is mechanised independently. Eight supporting list-membership lemmas (`step_extensive`, `allMatches_subset_facts`, the joint `step_subset`, `iterate_succ_extensive`, `iterate_subset_le`, `iterate_subset_program`) are proved without `sorry`. Four specialisation lemmas (`step_no_rules`, `iterate_no_rules`, `eval_no_rules`, `eval_fact_mem`) cover the rule-free regime the financial-institution scenario uses today; they give an unconditional soundness direction for ground-fact policies and depend only on `propext`. The headline `eval_monotone` is proved modulo one isolated combinatorial obligation, `herbrandBound_mono` (a length-after-`eraseDups` arithmetic argument with no semantic content). Two further meta-theorems, `eval_sound` and `eval_terminates`, are stated with `sorryAx` obligations named explicitly in the audit so the residual proof surface is visible at CI time. The corresponding open problems are listed in §6.

Implementation and conformance testing

The Rust implementation in `prototype/crates/postern-core` mirrors the Lean types and the *rewrite* function structurally. The conformance harness `postern-diff` consumes a JSON corpus emitted by `lake exe postern-corpus` and asserts three equalities per case: that the Rust outcome kind (*accept* / *refuse*) matches the Lean reference; that, on accept, the rewritten plan, output schema, predicate read-set, and touched relation are structurally equal to the Lean reference; and that the input plan's *filterCols* matches the Lean auxiliary, independent of the rewriter.

JSON-corpus conformance is preferred to Lean-to-Rust extraction on the grounds that the corpus interface is stable across compiler-version churn in both languages and that divergence manifests as a CI failure rather than a build failure.

Datalog backend. The gateway is designed to evaluate policies through `biscuit-auth`'s public `biscuit_auth::datalog` module — `World::new()`, `add_fact`, `add_rule`, `run`, `query_match` — which is the same evaluator used inside the production token-verification surface, only without the token-handling layers we put out of scope (§6). The in-tree `postern-core` migration to this backend is in progress; the column-grant DSL serves as a byte-equivalent stand-in until the migration lands. A second conformance corpus, exercising Lean `eval` against `biscuit_auth::datalog::World` directly, is the natural pair for the rewriter corpus and is queued as the next work item.

The corpus comprises 18 cases (15 accept, 3 refuse): seven behavioural cases drawn from the financial-institution scenario of §5; three refusal regressions for known attack shapes (filter-on-forbidden-column, unknown-relation, one nested forbidden-filter variant); two acceptance regressions for the empty-projection collapse cases (unknown-principal collapses to empty schema; over-projection of forbidden columns drops to empty); and six policy-language edge cases (duplicate grants, catalog-absent columns, case-sensitive principal, trailing- whitespace principal, nonexistent project column, nested *Project* narrowing). All eighteen pass on the current Rust implementation.

Evaluation: a financial institution with three principals

The scenario is illustrative, not a deployment claim. We pick a public schema — the Kaggle `transactions-fraud-datasets` — that is small enough to reproduce end-to-end in the conformance harness yet realistic enough to surface the three failure modes the rewriter must rule out: PII over-projection, filter side-channels on forbidden columns, and cross-departmental reach. The schema has three tables — `users_data` (customer-level PII and demographics), `cards_data` (card metadata including PAN and limit), and `transactions_data` (ledger entries) — which is the minimal shape that allows a cross-source policy story while keeping the IR single-relation (§6 lifts the latter restriction). Around it we instantiate three department-scoped agent principals that a retail bank would typically run:

- **CRM** (Customer Relationship Management) — segmentation and customer-support lookups. Needs identifiers and demographics on `users_data`; never sees cards, transactions, or PII fields (`ssn`, `email`).
- **CardOps** (Card Operations) — issuance, activation, limit changes. Needs card metadata on `cards_data` but **never the full PAN** (`card_number`, PCI-DSS scope) and has no business with users or transactions in this scenario.
- **FraudRisk** (Fraud and Risk Analytics) — anomalous-spend investigation. Needs the full `transactions_data` plus a minimum-necessary slice of `users_data` (just `id` + `region`) to bucket by geography; does not need

`name`/`age`/PII.

A fourth principal `Marketing` appears in the case table to exercise the unknown-principal fail-closed path. The full policy is reproduced verbatim in `scenarios/financial-institution/policy.postern` and rendered inline in §3 Policy. The case table below summarises the behavioural rows of the corpus.

principal	plan	outcome	rewritten schema
`CRM`	`Scan users_data`	accept	`id, name, region, age`
`CRM`	`Project [ssn,email]` over above	accept	∅ (over-projection collapses)
`CRM`	`Filter on ssn`	**refuse**	—
`CardOps`	`Scan users_data` (cross-dept)	accept	∅ (no matching grant)
`FraudRisk`	`Scan users_data`	accept	`id, region` (minimum-necessary)
`Marketing`	`Scan users_data` (unknown prin.)	accept	∅ (empty allow)
`CRM`	`Scan credit_bureau_imports`	**refuse**	— (unknown relation)

Each row corresponds to a corpus case in the conformance harness; the rows annotated *refuse* exercise Theorems 8 and 9 of §4.

Related work

We are not aware of prior work that establishes a mechanised soundness theorem for a plan-level rewriter in an LLM-agent- facing lakehouse setting. The closest landmarks fall into four groups.

Verified authorization decision procedures. Cedar (Cutler et al. 2024) formalises and proves the soundness of an authorization-decision function in Lean. The axis of verification differs from ours: Cedar establishes authorize(*request*) $\in$ {*allow*, *deny*} correctly classifies a per-call request, whereas we establish that the *output of a plan transformation* is contained in the policy-allowed set. The two are complementary; we adopt the Cedar style of Lean-mechanised denotational semantics for our policy.

Capability-based enforcement runtimes. SEAL (Sadeghi et al. 2023) provides capability-based access control for analytic workloads at the runtime level; the policy core is not mechanically verified. Our development is the dual: the policy core is verified, and the runtime is correspondingly lighter. Biscuit (Clever Cloud and contributors 2024) is both the deployed capability-token distribution mechanism we assume on the front end *and* the policy-language layer we mechanise: the Horn fragment of its Datalog dialect underpins our policy semantics, and the

production-grade `biscuit-auth` Rust crate is the gateway's runtime evaluator (`biscuit_auth::datalog::World`). Block attenuation, audience, expiry, and key rotation are explicit out-of-scope items (§6); the column-grant surface syntax compiles to ground `right(principal, relation, column)` Datalog facts in the in-scope fragment.

Information-flow control for database-backed applications. Jeeves (Yang et al. 2012), Jacqueline (Yang et al. 2016), and the faceted-execution line (Schoepe et al. 2016) enforce IFC inside ORM-backed applications using a faceted-value runtime discipline. They predate the LLM-agent threat model and do not target the lakehouse setting; we view them as the closest PL-side relatives of Layer 2 of our development.

Defences for LLM-agent prompt injection. AgentDojo (Debenedetti et al. 2024) and CaMeL (Debenedetti et al. 2025) develop capability-flow defences at the agent boundary. Our development is complementary: the rewriter of §3–§4 enforces a policy at the lake-facing boundary on plans; their constructions enforce analogous properties on the agent's own emitted code. Closest to our Layer 2 specifically, Odersky et al. (Odersky et al. 2026) propose Scala~3 capture-checking as the type-level mechanism for tracking capabilities through agent code; we adapt the same intuition under Rust's weaker type-system commitments (§3).

Deployed alternatives we improve upon. PostgreSQL row security policies (PostgreSQL Global Development Group 2024) and equivalent CLS facilities require per-engine integration and do not compose across the heterogeneous ingest paths typical of lakehouse deployments. Open Policy Agent (Cloud Native Computing Foundation 2021) is general-purpose but does not reason about query outputs at the plan level. Tenant segregation forfeits cross-source analytics and is the silo case we discuss in §1.

Open challenges and future work

Three extensions of the Lean development are the natural next research questions.

Value-based predicate side-channels and a richer Filter. The current *Filter* constructor carries only a column name; the rewriter therefore enforces that a principal does not *reference* a forbidden column in a filter, but cannot reason about exfiltration through allowed columns whose values the agent probes (e.g. issuing one query per candidate `region` value and observing row counts). The natural extension introduces a predicate term φ into the IR and adds a coverage condition: every free variable of φ lies in $P.\mathit{allowed}\ p\ \mathit{touched}(q)$. The harder half is deciding when the *value* of an allowed column carries enough mutual information about a forbidden one to warrant blocking — the same problem space as the differential-privacy boundary below.

Bridging `Policy.allowed` and `Program.allowed`. `verifier/lean/Postern.lean` and `verifier/lean/Datalog.lean` today coexist without import: the rewriter consults `Policy.allowed prin rel : List Column` (a `flatMap` over `Grant.columns`), and the evaluator separately defines `Program.allowed prin rel : List Symbol` (a `filterMap` over ground `right`-atoms in `eval`). The natural bridge — a constructor `Policy.toProgram : Policy → Program` together with the theorem

$$\mathit{bridge\_allowed} : P.\mathit{allowed}\ p\ r = (P.\mathit{toProgram}).\mathit{allowed}\ p\ r$$

— is unstated. Closing it lifts the §4 rewriter theorems through the Datalog evaluator, removing the "as-if Datalog" gap between the surface column-grant DSL and the Horn-fragment policy language that the gateway is designed to dispatch through (§5 *Datalog backend*).

Cross-relation joins. The Plan IR is single-relation. The Rust implementation handles joins by per-leg rewriting but the composition is not under proof. The conjecture is a theorem of the form

$$\mathit{rewrite\_sound\_join} : \mathit{accept}(q_1) \wedge \mathit{accept}(q_2) \implies \sigma(\mathit{rewrite}(\mathit{Join}(q_1, q_2))) \subseteq \bigcup_i P.\mathit{allowed}\, p\ \mathit{touched}(q_i)$$

on a Plan IR extended with a *Join* constructor. The join-key leak — joining on a column c without projecting it, which leaks c's value distribution — mirrors the filter side-channel and admits the analogous coverage condition.

Aggregation with a differential-privacy boundary. A principal may be permitted to read SUM(*amount*) without permission to read individual rows. The rewriter extension here is straightforward in shape but admits a non-trivial soundness statement once the differential-privacy boundary is parameterised; SEAL (Sadeghi et al. 2023) and the faceted line (Schoepe et al. 2016) are the closest reference points.

Capability attenuation inside the proof. The Lean development takes *Principal* as a flat string and assumes the gateway has already verified the bearer of a biscuit token. Modelling biscuit's Datalog-based attenuation, expiry, and audience checks inside the Lean proof lifts the principal- extraction row out of the trusted base — a substantial strengthening of the artifact's overall claim. Adjacent open problems include catalog-integrity attestation and plan- integrity in transit.

Reproducibility

```
verifier/lean/   Lean 4 spec + theorems + corpus emitter
prototype/       Rust workspace: postern-core, postern-diff,
                 postern-guardrail, postern-wasm
scenarios/       Financial-institution case study
web/             Astro site with /paper, /slides, /demo
```

```
paper/          This document + figures + build.sh
scripts/        reproduce.sh - chains everything
```

Toolchains: **Lean 4.29.1** (pinned in `verifier/lean/lean-toolchain`), **Rust stable** (tested 1.93), and optionally `wasm-pack` 0.13 if you want the `/demo` WASM bundle rebuilt from source (skipped with a notice otherwise — the proofs and conformance harness do not depend on it). Single command:

```
scripts/reproduce.sh
```

Expected output ends with `18/18 cases pass (Lean reference == Rust impl)` and an axiom audit showing only `propext` and `Quot.sound`. With `wasm-pack` present, the last step emits `postern_wasm_bg.wasm` into `web/src/wasm/`. We do not quote a wall-clock budget — `time scripts/reproduce.sh` on the reader's hardware is the only honest measurement, and the dominant cost is the Lake fetch on a cold cache (Mathlib pull) rather than the proofs themselves.

References

Anthropic. 2024. *Model Context Protocol*. Https://modelcontextprotocol.io.

Clever Cloud and contributors. 2024. *Biscuit: A Bearer Token with Datalog-Based Offline Authorization*. Https://www.biscuitsec.org.

Cloud Native Computing Foundation. 2021. "Open Policy Agent and the Rego Language." *CNCF Graduated Project Documentation*.

Cutler, Joseph W., Craig Disselkoen, Aaron Eline, et al. 2024. "Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization." *Proc. ACM Program. Lang.* 8 (OOPSLA1).

Debenedetti, Edoardo et al. 2025. *Defeating Prompt Injections by Design*.

Debenedetti, Edoardo, Jie Zárate, and Tramer Florian Tramer. 2024. "AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents." *Advances in Neural Information Processing Systems (NeurIPS)*.

Moura, Leonardo de, and Sebastian Ullrich. 2021. *The Lean 4 Theorem Prover and Programming Language*. Https://lean-lang.org.

Odersky, Martin, Yichen Zhao, Yifan Xu, Oliver Bračevac, and Cao Nguyen Pham. 2026. *Tracking Capabilities for Safer Agents*. https://arxiv.org/abs/2603.00991.

PostgreSQL Global Development Group. 2024. *Row Security Policies*. Https://www.postgresql.org/docs/current/ddl-rowsecurity.html.

Raasveldt, Mark, and Hannes Mühleisen. 2019. *DuckDB: An in-Process SQL OLAP Database Management System*. Https://duckdb.org.

Sadeghi, Ahmad-Reza et al. 2023. "SEAL: Capability-Based Access Control for Data-Analytic Scenarios." *Proceedings of the 28th ACM Symposium on Access Control Models and Technologies (SACMAT)*. https://doi.org/10.1145/3589608.3593838.

Schoepe, Daniel, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. 2016. "Explicit Secrecy: A Policy for Taint Tracking." *IEEE European Symposium on Security and Privacy (EuroS&p)*. https://doi.org/10.1109/EuroSP.2016.14.

Yang, Jean, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. "Precise, Dynamic Information Flow for Database-Backed Applications." *PLDI*. https://doi.org/10.1145/2908080.2908098.

Yang, Jean, Kuat Yessenov, and Armando Solar-Lezama. 2012. "A Language for Automatically Enforcing Privacy Policies." *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2103656.2103669.