EAM–SQL: Cryptographic Safety Envelopes for Table–Centric AI

Susanna Di Vita
Department of Computer Science
ETH Zurich
sdivita@student.ethz.ch

Abstract

Table-centric AI systems, such as NL2SQL agents, RAG-over-tables, and ETL planners, turn model outputs into real database effects. Yet common defenses (filters, sandboxes, post-hoc audits) lack planner-to-DB, query-level ex-ante guarantees. We present EAM-SQL, a lightweight, HMAC (Hash-based Message Authentication Code)-only authorization layer that attaches a cryptographically verifiable "safety envelope" to every statement. EAM-SQL introduces (i) SQL-aware caveats over operations, tables/columns, joins, row predicates, and rate/time; (ii) normalized SQL content binding via hashes of a canonical AST (WHERE/PROJECTION/FROM graph), preventing prompt-level rewrites from smuggling different queries; and (iii) transaction-sequence integrity via a chained transaction hash that enforces order and blocks replay or splicing. A sidecar proxy performs mandatory parameterization, canonicalization with content-hash verification, caveat checking, and tamper-evident auditing, all at microsecond scale. On a SQL attack harness (800 attempts spanning injection, query splicing, unauthorized access, join escalation, broad exfiltration, replay, and tag manipulation), EAM-SQL achieved 0% unauthorized execution with 6–50 s detection and <2 ms P99.9 verification overhead. By cryptographically binding statement content, scope, and order - rather than relying on mutable roles or heuristic filters - EAM-SQL provides verifiable safety for NL interfaces to databases, RAG-over-relational retrieval, and bounded ETL in enterprise and regulated settings.

1 Introduction

Problem. Table-centric AI, such as NL2SQL agents, RAG-over-tables, and ETL planners, now issues database statements with real effects. Yet prevailing defenses (filters, sandboxes, post-hoc audits) do not provide planner→DB, per-statement *ex-ante* guarantees. In particular, they do not (i) bind a statement's canonical *content* and *scope* to policy, nor (ii) enforce *order* across multi-statement workflows. As a result, systems remain exposed to prompt/data injection, query splicing, replay, and reordering even in the presence of RBAC/RLS and parameterization [10, 5]. The stakes are acute in analytics and regulated domains where narrow, verifiable access is mandatory.

Why this is hard in practice. In deployed NL2SQL and RAG-over-tables systems, a single planner mistake can have immediate, real-world consequences. For instance, in a customer-analytics workflow, an intended query such as "SELECT revenue FROM sales WHERE region = ?" can be silently transformed into a broad JOIN that leaks customer-level data, or into a predicate-free scan that violates regulatory access bounds. These failures occur despite RBAC/RLS and parameterization because nothing constrains the *structure* or *scope* of the executed SQL itself. As AI-driven data access becomes common in finance, healthcare, and internal analytics platforms, the lack of verifiable, per-statement guarantees leaves organizations exposed to subtle but high-impact errors.

Preprint.

Our approach. We introduce *EAM-SQL*, a lightweight authorization layer that attaches a cryptographically verifiable *safety envelope* to each SQL statement. EAM-SQL specializes macaroons [3] to SQL by (a) expressing conjunctive, append-only caveats over operations/tables/columns/joins/predicates/rate/time, (b) *binding canonical structure* via hashes of a normalized AST (including `WHERE`, projection, and FROM-graph), and (c) enforcing *sequence integrity* with a chained transaction hash. Verification is HMAC-only [7, 2], so checks run in microseconds and compose cleanly with existing DB controls.

Why this matters. Rather than trusting planners and *observing* behavior after the fact, EAM-SQL makes execution *contingent* on cryptographic predicates that must verify *before* the DB is touched. This yields verifiable least-privilege at the planner–executor boundary for NL2SQL and RAG-over-relational stores—turning table QA/analytics workflows from best-effort into *provably* bounded execution, without heavyweight policy engines or public-key costs.

Contributions.

- **Construction.** A practical, HMAC-only scheme that binds SQL *content* (canonical AST), *scope* (SQL-aware caveats), and *order* (hash-chained sequencing) to each statement, with mandatory parameterization.
- **System.** A sidecar proxy that canonicalizes SQL, verifies caveats, enforces sequence/replay/rate limits, and emits tamper-evident, re-verifiable audit records; policies are authored as human-readable YAML and compiled to caveats.[1]
- **Evidence.** A SQL-focused attack harness shows *0% unauthorized execution*, microsecond detection, and < 2 ms P99.9 verification overhead on commodity hardware, covering injection, splicing, unauthorized access, join escalation, replay, and tag manipulation.

2 Related Work

Capability security and macaroons. Capability systems address confused-deputy failures by binding authority to tokens [6]. Macaroons refine capabilities with attenuating caveats realized as an HMAC chain, enabling decentralized, append-only restriction with low verification cost [3]. Compared to signature-based schemes, HMAC verification offers microsecond-scale checks while retaining unforgeability under standard assumptions [7, 2]. Prior uses emphasize delegation in general agents; they do not specialize caveat vocabularies to *SQL structure* nor bind per-statement *content* and *sequence*. EAM-SQL contributes precisely that specialization for table-centric pipelines.

Planner–executor LLM pipelines. Contemporary NL2SQL, table-QA/RAG over relational stores, and agentic data workflows instantiate a planner→executor loop where model outputs are executed as SQL over real schemas [15, 12, 9]. Multi-agent orchestration and tool-augmented reasoning (e.g., AutoGen, Tree-of-Thoughts, Reflexion) improve planning by iteration or debate, but execution control typically relies on sandboxing, logging, or heuristic filters [13, 14, 11]. These are valuable yet lack cryptographic, per-statement constraints over *content/scope/order*. EAM-SQL treats the planner–executor handoff as a *security boundary* and requires verifiable envelopes for each action *before* execution.

Prompt injection and confused deputy. Prompt- and data-injection attacks can steer agents and tools toward unintended actions [10, 5]. Parameterization mitigates raw literal injection but does not bind *structure* (e.g., FROM-graph, `WHERE` template) nor *workflow order*. EAM-SQL closes this gap by hashing canonical structure and chaining transactions, so injected rewrites or reordered steps fail verification *by construction*.

DB-native controls and auditing. RBAC/ABAC and row-level security remain necessary but are coarse or mutable and cannot attest that a specific statement's canonical structure was in scope, nor enforce cross-statement order or freshness. Tamper-evident logging (e.g., transparency trees) improves post-hoc accountability [4, 8], but does not prevent execution. EAM-SQL is preventative and audit-aligned: it refuses non-conforming statements ex-ante and produces re-verifiable logs, supporting traceability and oversight requirements (e.g., EU AI Act) [1].

[1] All code is available at `https://github.com/SusannaDiV/eam-sql`.

3 EAM-SQL: Cryptographic Execution Authorization for Table-Centric AI Systems

We instantiate Execution Authorization Macaroons (EAMs) for table-centric AI systems—including NL2SQL agents, RAG-over-tables, and ETL pipelines—by introducing *EAM-SQL*, a lightweight, HMAC-only capability system that attaches a verifiable "safety envelope" to every database statement. An EAM-SQL envelope combines (i) SQL-aware caveats over operation class, schema/table/column access, row-predicates, aggregates/joins, time window, and rate; (ii) *normalized SQL content binding* to prevent trivial prompt rewrites from smuggling different queries; (iii) *transaction sequence integrity* with `prev_tx_hash` across multi-statement workflows; (iv) *microsecond-scale* verification before forwarding to the database; and (v) tamper-evident audit logs that attribute each query to a bounded capability.

Design highlights. EAM-SQL is *SQLfirst*: caveats speak operations/tables/columns/predicates; queries are *canonicalized* to an AST and *structurehashed* (`sql_content_hash`, with optional `where_predicate_hash`/`from_graph_hash`/`projection_hash`); *mandatory parameterization* eliminates literal injection; and *hashchained sequencing* with `prev_tx_hash` enforces perplan order and freshness (Figure 1). The safety envelope uses HMACSHA256 only (no public keys), yielding μsscale verification and tamperevident, MACed audit records. Deployment is a sidecar proxy with optional distributed plan sharding and health telemetry. Details: canonicalization/sequence/enforcement in App. A.3, audit predicates in App. C.1, deployment/ops in App. A.7, caveat glossary and policy→crypto mapping in App. A.4–A.5.

Scenario	Attempts	Unauthorized Exec.
Attack harness (8 classes)	$\geq$800	0%
Ablations (9 variants)	9×100	Increases; restored to 0% when re-enabled
Distributed (2–N shards)	200	0%
Failure handling (crash/replay/clock)	120	0%
Perf (steady load)	–	P50$<$1 ms; P99.9$<$5 ms

Table 1: Experiment coverage at a glance.

SQL-aware caveats. Capabilities constrain the SQL surface *ex ante* via caveats over operation (`SELECT/UPDATE/...`), tables/columns, explicit join constraints, row limits, rate limits, and expiry. Optional caveats bind canonicalized `WHERE` predicates to prevent silent broadening of queries. We also include explicit *shape flags* (e.g., UNION/CTE/subquery/multi-statement/STAR/comments) to default-deny risky constructs. (See the full vocabulary and examples in App. A.4–A.5.)

Normalized SQL content binding & mandatory parameterization. We parse SQL to an AST, emit a canonical form, and bind its hash as `sql_content_hash`. Values must be parameter placeholders (no raw literals); only structure is hashed. Canonicalization defeats column-order tricks, whitespace/comments, implicit joins, and homoglyphs (procedure in App. A.3). Governance profiles in YAML compile to caveats; representative analytics/ETL profiles appear in App. A.5.

Per-query tokens and transaction sequencing. Each statement attenuates the plan-level macaroon with a fresh `nonce`, the `sql_content_hash`, and a CAS-protected `prev_tx_hash`; the verifier maintains a rolling transaction hash and rejects reordering or reuse. (See App. A.3 for the sequencing rule and CAS semantics.)

Gateway enforcement. A sidecar proxy (or in-process library) enforces, in order: parameterization and AST canonicalization; caveat checks over operation/tables/columns/joins and shape flags; content-hash recomputation; `prev_tx_hash` CAS and nonce replay checks; then DB execution and an append-only audit record. (Enforcement checklist in App. A.3; verification predicates and audit format in App. C.1.)

Audit trail. Every allow/deny decision yields a MACed, tamper-evident record suitable for enterprise governance and incident response; schema and checks are summarized in App. C.1.

4 Security evaluation on tabular threats

Table 2 evaluates EAM-SQL on a SQL-aware attack suite spanning prompt injection (NL→SQL), query splicing (multi-statement), unauthorized table/column access, join escalation, broad exfiltration, replay, and tag manipulation; representative SQL patterns are listed in Table 10. We compare EAM-SQL against two natural baselines, per-statement MACs and per-statement signatures, which allow 36–100% and 73–100% unauthorized executions, respectively (see Table 6). Thus the benchmark is not trivial: dropping a single caveat (e.g., `sql_content_hash`, `prev_tx_hash`, or `where_predicate_hash`) immediately reintroduces 36–100% failure rates (Table 7). EAM-SQL achieves 0% unauthorized execution across all attack classes because the combined caveats jointly eliminate structural, scope, and ordering attack surfaces. Detection times (0.09–0.62 ms) correspond to the microsecond-scale gateway checks and show that enforcement remains negligible compared to normal OLTP/analytics latency. Methodology and harness details appear in App. A.8.

Table 2: EAM-SQL adversarial outcomes (lower is better). Baselines (MAC-only and signature-only) permit 36–100% unauthorized executions depending on attack class; see Table 6 for details.

Attack Class	Attempts	Successes	Success Rate	Avg Detect (ms)
Prompt injection (NL→SQL)	100	0	0.00%	0.24
Query splicing (multi-stmt)	100	0	0.00%	0.09
Unauthorized table access	100	0	0.00%	0.44
Unauthorized column access	100	0	0.00%	0.62
Join escalation	100	0	0.00%	0.59
Data exfiltration (broad read)	100	0	0.00%	0.60
Replay	100	0	0.00%	0.38
HMAC manipulation	100	0	0.00%	0.32
Overall	800	0	0.0%	–

Performance. Verification runs in the μs range; added pre-DB latency is sub-millisecond median and stays under a few milliseconds at P99.9 in our profiles (see App. A.6). Because typical OLTP/analytics latency dwarfs this overhead, the marginal cost of EAM–SQL is negligible for table-centric QA/analytics loops.

4.1 Comparison to alternatives

	EAM-SQL	RBAC/ABAC	Query Firewalls	Policy Engines
Ex-ante guarantees	cryptographic	×× mutable	pattern	post-hoc
SQL awareness	native	×× coarse	regex	indirect
Audit trail	tamper-evident	mutable	×× limited	mutable
Latency footprint	$\ll$2 ms	comparable	comparable	×× often high
Deployment surface	sidecar	wide	inline	×× heavy

Security properties. EAM–SQL provides (i) origin integrity via an HMAC-based caveat chain, (ii) anti-replay via per-statement nonces, (iii) monotonic attenuation of authority, (iv) sequence integrity via a hash-chained transaction history, and (v) SQL-scoped capability bounding using canonical AST hashing. These properties jointly explain the 0% unauthorized-execution rate across all attack classes in Table 2. Formal definitions and proof sketches appear in Appendix B.

Use cases. EAM-SQL secures (i) NL2SQL analytics under tenant bounds; (ii) RAG over structured stores; and (iii) ETL pipelines with bounded updates. Execution is conditioned on a token that binds query structure and scope, with sequence, replay, and rate checks enforced at machine speed. Across the NL2SQL/RAG-style intents in Table 11 and the relational schema in Table 8, EAM–SQL prevents *unauthorized execution ex ante* by binding canonical structure, scope (tables/columns/predicates), and sequence to each query. In practice, this means benign analytics flows pass while injected rewrites, broadened WHEREs, and reorders are blocked before the DB is touched.

SQL-focused limitations EAM-SQL prevents *unauthorized execution* by construction but does not prove semantic correctness of planner-generated SQL. For sensitive changes, use `requires_human_sig`, DB-native row-level security, and monitoring. Time-based caveats require a trusted clock; keys must be rotated/revoked per best practices. Timing/covert-channel exfiltration is out of scope; rate limits reduce bandwidth but cannot eliminate such channels. Extended notes in App. C.3.

5 Conclusion

EAM–SQL shifts table–centric AI from heuristic, best-effort defenses to *cryptographically enforced* execution. By attaching a verifiable safety envelope to each statement—binding canonical content, scope, and sequence—and verifying it at microsecond scale, EAM–SQL delivers ex-ante guarantees against injection, splicing, replay, and reordering while producing tamper-evident audits. Our results show zero unauthorized executions across an extensive SQL attack suite with negligible latency overhead, making the design practical for fast planner→DB loops in enterprise and regulated settings. Its intentionally minimal, HMAC-only construction integrates cleanly with existing RBAC/RLS and deployment models (sidecar or in-process). Promising directions for future work include: standardized SQL caveat vocabularies and policy templates to ease adoption and audit; deeper DB-native integrations (e.g., planner hooks and logical/physical plan hashing) and coverage for non-relational tabular stores (e.g., DuckDB, BigQuery); extending envelopes beyond SQL to tabular adjacencies such as files, object stores, and CDC/queues for end-to-end agent workflows; and operational studies on usability, policy drift, human-in-the-loop overrides, and incident response in long-running deployments.

References

[1] Regulation (eu) 2024/1689 of 13 june 2024 laying down harmonised rules on artificial intelligence (artificial intelligence act). Official Journal of the European Union, 2024. URL https://eur-lex.europa.eu/eli/reg/2024/1689/oj/eng.

[2] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keyed hash functions and message authentication. In *CRYPTO*, LNCS 1109, 1996. URL https://cseweb.ucsd.edu/~mihir/papers/kmd5.pdf.

[3] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *NDSS*, 2014. URL https://www.ndss-symposium.org/ndss2014/programme/macaroons-cookies-contextual-caveats-decentralized-authorization-cloud/.

[4] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security*, 2009. URL https://www.usenix.org/legacy/events/sec09/tech/full_papers/crosby.pdf.

[5] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, and more. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. arXiv:2302.12173, 2023. URL https://arxiv.org/abs/2302.12173.

[6] Norman Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988. doi: 10.1145/54289.871709.

[7] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Hmac: Keyed-hashing for message authentication. *RFC 2104*, 1997.

[8] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. RFC 6962, 2013.

[9] Qin Liu, Bei Chen, Jiaqi Guo, Zeqi Lin, Jian-Guang Lou, Zhiruo Wang, Yan Gao, Lijie Wen, Zhi Jin, and Nan Duan. Tapex: Table pre-training via learning a neural sql executor. In *ICLR*, 2022.

[10] Fábio Perez and Ian Ribeiro. Ignore previous prompt: Attack techniques for language models. arXiv:2211.09527, 2022. URL https://arxiv.org/abs/2211.09527.

[11] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023. arXiv:2303.11366.

[12] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. In *ACL*, 2020.

[13] Tianhao Wu, Kanji Uchino, Xuehai Pan, Shreya Shankar, Karthik Narasimhan, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

[14] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, 2023. arXiv:2305.10601.

[15] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql. In *EMNLP*, 2018.

A Protocol Description

Algorithm 1: Execution Authorization Macaroons for SQL (EAM-SQL)

Assumptions: HMAC-SHA-256 is a PRF; SHA-256 is collision-resistant.
Canonical encoding $\mathrm{enc}(\cdot)$ is deterministic (e.g., CBOR/JSON with sorted keys).
Executor maintains per-plan state: trusted clock; replay cache $\mathcal{R}$ keyed by $(ctx, plid, pid, nonce)$; per-scope rate counters; and *prev_tx_hash* initialized to $p_0 = 0^{256}$.
Required per-query caveats: `sql_content_hash`, `prev_tx_hash`, `nonce`. Caveat list C is an **ordered list**; iteration is in append order.
Parameters: Context ctx; planner ID $plid$; plan ID pid. Caveats may include: `schema`, `tables`, `columns`, `where_predicate_hash`, `allow_aggregates`, `max_rows`, `disallow_join_with`, `tenant_id`, `expires/not_before`, `max_rate`, `sql_content_hash`, `prev_tx_hash`, `nonce`, optional `requires_human_sig`.

Function INITPLANSTATE$(ctx, plid, pid)$**:**
  state.prev_tx_hash $\leftarrow 0^{256}$; state.counters $\leftarrow$ zeros
  return state

Function KEYSETUP(P, E)**:**
  generate X25519 keypairs for P and E;
  $z \leftarrow \mathrm{X25519}(sk_P, pk_E)$ `// ECDH`
  $K \leftarrow \mathrm{HKDF}(z, \mathrm{info} = ctx \parallel plid \parallel pid)$
  return K

Function ROOTTAG$(K, ctx, plid, pid, salt)$**:**
  return $t_0 \leftarrow \mathrm{HMAC}_K(ctx \parallel plid \parallel pid \parallel salt)$

Function MINTROOT$(K, ctx, plid, pid, C_0)$**:**
  $salt \leftarrow$ fresh random
  $t \leftarrow$ ROOTTAG$(K, ctx, plid, pid, salt)$
  foreach $c \in C_0$ **do**
    $t \leftarrow \mathrm{HMAC}_t(\mathrm{enc}(c))$
  $Tok \leftarrow (\mathrm{ver} = \texttt{v1}, plid, pid, salt, C = C_0, tag = t)$
  return Tok

Function ATTENUATE(Tok, C_{add})**:**
  $t \leftarrow tag(Tok)$; $C \leftarrow C(Tok)$
  foreach $c \in C_{\mathrm{add}}$ **do**
    $t \leftarrow \mathrm{HMAC}_t(\mathrm{enc}(c))$;
    $C.\mathrm{append}(c)$
  update Tok with C and t;
  return Tok

Function SQLCONTENTHASH(q)**:**
  return $h \leftarrow \mathrm{SHA\text{-}256}(\mathrm{canonicalize}(q))$

Function DERIVEPERQUERY$(Tok_{i-1}, q_i, state)$**:**
  $h_i \leftarrow$ SQLCONTENTHASH(q_i); $\nu \leftarrow$ fresh random nonce
  $C_{\mathrm{add}} \leftarrow [\texttt{sql\_content\_hash} = h_i,\ \texttt{prev\_tx\_hash} = state.prev\_tx\_hash,\ \texttt{nonce} = \nu,\ \texttt{issued\_at} = \mathrm{now}]$
  $Tok_i \leftarrow$ ATTENUATE$(Tok_{i-1}, C_{\mathrm{add}})$
  $p_i \leftarrow \mathrm{SHA\text{-}256}(state.prev\_tx\_hash \parallel h_i)$
  return (Tok_i, p_i)

Algorithm 2: SQL Caveat Verification Routine (SQLCAVEATSHOLD)

Function SQLCAVEATSHOLD($Tok, q, state$):
  $C \leftarrow C(Tok)$
  // -- Temporal and rate caveats --
  if *(*$\exists$ `expires` $\in C$ ***and*** $\text{now} > C[\texttt{expires}]$*)* $\vee$ *(*$\exists$ `not_before` $\in C$ ***and*** $\text{now} < C[\texttt{not\_before}]$*)* $\vee$ *(*$\exists$ `max_rate` $\in C$ ***and*** $\neg$ WITHINRATE$(C[\texttt{max_rate}], state.counters)$*)* **then**
    return false
  // -- Structural access caveats --
  if *(*$\exists$ `schema` $\in C$ ***and*** $\neg$ INSCHEMA$(q, C[\texttt{schema}])$*)* $\vee$ *(*$\exists$ `tables` $\in C$ ***and*** $\neg$ INTABLES$(q, C[\texttt{tables}])$*)* $\vee$ *(*$\exists$ `columns` $\in C$ ***and*** $\neg$ INCOLUMNS$(q, C[\texttt{columns}])$*)* $\vee$ *(*$\exists$ `disallow_join_with` $\in C$ ***and*** $\neg$ NODISALLOWEDJOINS$(q, C[\texttt{disallow_join_with}])$*)* **then**
    return false
  // -- Predicate and aggregation caveats --
  if *(*$\exists$ `where_predicate_hash` $\in C$ ***and*** $\neg$ WHEREMATCHES$(q, C[\texttt{where_predicate_hash}])$*)* $\vee$ *(*$\exists$ `allow_aggregates` $\in C$ ***and*** $\neg$ ALLOWAGGREGATES$(q, C[\texttt{allow_aggregates}])$*)* **then**
    return false
  // -- Hash binding and replay caveats --
  if *(*`sql_content_hash` $\notin C$ ***or*** $C[\texttt{sql\_content\_hash}] \neq$ SQLCONTENTHASH(q)*)* $\vee$ *(*`prev_tx_hash` $\notin C$ ***or*** $C[\texttt{prev\_tx\_hash}] \neq state.prev\_tx\_hash$*)* $\vee$ *(*`nonce` $\notin C$ ***or*** $(ctx, plid(Tok), pid(Tok), C[\texttt{nonce}]) \in \mathcal{R}$*)* **then**
    return false
  // -- Optional human verification --
  if $\exists$ `requires_human_sig` $\in C$ ***and*** $\neg$ VERIFYHUMANSIG(Tok) **then**
    return false
  return true

Algorithm 3: EAM-SQL VERIFY & EXECUTE

Function VERIFY$(K, ctx, Tok, q, state)$:
  $t \leftarrow$ ROOTTAG$(K, ctx, plid(Tok), pid(Tok), salt(Tok))$
  foreach $c \in C(Tok)$ **do**
    $t \leftarrow \text{HMAC}_t(\text{enc}(c))$
  if $t \neq tag(Tok)$ **then**
    return false
                                        // Reject if caveat stripping/reordering
  if $\neg$ SQLCAVEATSHOLD$(Tok, q, state)$ **then**
    return false
                                                        // SQL-specific checks
  return true
Function EXECUTESQL$(K, ctx, Tok_i, q_i, state)$:
  if $\neg$ VERIFY$(K, ctx, Tok_i, q_i, state)$ **then**
    AUDIT(Tok$_i$, q_i,decision=`deny`); **return**`DENY`
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{(ctx, plid(Tok_i), pid(Tok_i), C(Tok_i)[\texttt{nonce}])\}$
  BUMPRATECOUNTERS(Tok$_i$, $state.counters$) p$_i \leftarrow$ SHA-256$(state.prev_tx_hash \,\|\, \text{SQLCONTENTHASH}(q_i))$
  state.prev_tx_hash $\leftarrow p_i$
  AUDIT(Tok$_i$, q_i,decision=`allow`) **return**(`ALLOW`,p$_i$)

A.1 Why not just per-action MACs?

A natural baseline is to derive a shared session key and authenticate each SQL query using a per-query MAC, e.g., $t_i = \text{HMAC}_K(q_i)$ (as seen in Table 3). While this ensures origin integrity, it has three limitations: (i) no binding to SQL scope, timing, or risk constraints; (ii) no replay prevention without extra mechanisms such as nonces or counters; and (iii) no natural support for delegation or staged approval. Public-key signatures (e.g., Ed25519) ensure origin integrity without a shared secret, but they add millisecond-scale verification cost and still lack native support for replay prevention or append-only caveats, requiring external policy engines to provide these features. In

contrast, Execution Authorization Macaroons for SQL (EAM-SQL) encode explicit SQL caveats (e.g., table/column restrictions, WHERE clause binding, transaction sequence integrity, aggregate permissions) directly into the cryptographic token via chained HMACs. This ensures each SQL query carries its own verifiable safety envelope, enabling replay resistance, staged approval, and audit-ready logs without extra infrastructure.

Table 3: Comparison of SQL enforcement mechanisms

Mechanism	Policy Expressiveness	Verification Cost	Replay / Delegation Support
Plain MACs	None (integrity only)	μs	Manual, ad-hoc
Signatures	External policy binding	ms (public key ops)	Manual, ad-hoc
EAM-SQL (ours)	Built-in SQL caveats	μs (HMAC chain)	Automatic, append-only

A.2 NL2SQL model benchmark.

While our main evaluation uses a simulated SQL attack harness rather than a live NL2SQL or RAG deployment, we additionally integrated EAM–SQL with an off-the-shelf text-to-SQL model (T5-base fine-tuned on WikiSQL) using the schema from Table 8. On 300 benign and 100 adversarial prompts, EAM–SQL again achieved 0% unauthorized execution and verification latencies in the same range as our main results, suggesting that the simulated harness is representative of a realistic NL2SQL pipeline.

A.3 Engineering Extensions for Production

Beyond the core protocol, our implementation includes engineering features that make EAM-SQL robust in production deployments.

Gateway hardening (enforcement details). The proxy performs additional pre-DB checks at microsecond scale:

- **Mandatory parameterization:** deny raw string/number literals; only placeholders are allowed (`$1`, `?`, `:name`).
- **Shape validation:** default-deny `UNION`, CTEs, subqueries, multi-statement, wildcard `*`, and comments unless explicitly enabled by caveats.
- **Content binding:** recompute `sql_content_hash` from canonical AST (with Unicode NFKC identifier normalization) and compare to the caveat.
- **Sequence & replay:** compare-and-set for `prev_tx_hash`; replay cache keyed by `nonce` with TTL.
- **Scope checks:** allow-lists for operation/table/column; optional `where_predicate_hash`, `from_graph_hash`, and `projection_hash`.

Canonicalization and content binding. The gateway parses SQL to an AST, re-emits a canonical string, and hashes *structure only*:

```
sql_content_hash     = H(canonical SQL)
where_predicate_hash = H(canonical WHERE AST)
from_graph_hash      = H(sorted FROM+JOIN graph)
projection_hash      = H(projection  aggregates  {DISTINCT})
```

Identifier normalization (Unicode NFKC) prevents homoglyph tricks. Canonicalization defeats column reordering, whitespace/comments, and implicit join rewrites.

Sequencing and freshness. Each perquery token carries a fresh nonce and the previous transaction hash:

```
tx_hash_i    = H(tx_hash_{i-1} || sql_content_hash_i)
prev_tx_hash = tx_hash_{i-1}    # presented by the caller, CAS-checked
nonce        = 96-bit random; enforced via atomic replay cache (e.g., SETNX+TTL)
```

Both reordering and replay fail by construction.

Distributed coordination (multi-planner, plan sharding). The system supports parallel execution with cryptographic ordering and cross-shard joins:

- **Plan sharding:** independent transaction chains per shard ($\texttt{tx\_hash}_i^{(s)}$), aggregated under one plan context.
- **Multi-planner coordination:** multiple planners can contribute per-shard queries; optional safety-filter co-signer can append stricter caveats for high-risk ops.
- **Cross-shard joins:** joins proceed only if each shard's token verifies and a join authorization proof links the inputs to the consumer query's `sql_content_hash`.
- **Audit aggregation:** shard-local logs are merged into a plan-level, append-only trail.

Failure handling and recovery. We add resilience without weakening guarantees:

- **Write-Ahead Log (WAL):** every allow/deny decision is durably recorded before execution; on restart, the gateway reconstructs `prev_tx_hash`, rate counters, and the replay cache.
- **Clock skew tolerance:** NTP-synchronized clocks with a small grace window for `expires`/`not_before`; skew status is exposed via health endpoints.
- **Background maintenance:** periodic eviction of expired nonces, rotation of WAL segments, and validation of time sources.

Health & monitoring. We expose real-time observability primitives for operations and compliance:


```
{
  "clock_skew": {"ntp_offset": 0.0, "policy": "grace_window"},
  "sql_replay_cache": {"size": 0, "max_size": 100000},
  "sql_wal": {"entries": 1042, "sequence_counter": 1042},
  "sql_recovery": {"pending_operations": 0},
  "rates": {"verify_qps": 1120, "deny_qps": 7}
}
```


A.4 Implementation Caveat Vocabulary (SQL)

Our SQL-aware caveats (normalized names) cover structure, scope, and pacing:

```
sql_content_hash     # canonical AST hash (structure only, no values)
prev_tx_hash         # chain binding for transaction order
where_predicate_hash # canonical WHERE AST hash (optional but recommended)
operation            # SELECT / UPDATE / INSERT / DELETE
tables               # allow-list of table names
columns              # allow-list of column names
allow_aggregates     # list or boolean
max_rows             # upper bound on result size
max_rate             # e.g., "120/min"
expires              # absolute expiry
-- shape flags (default false unless explicitly set)
allow_union, allow_cte, allow_subquery, allow_multi_stmt, allow_star, allow_comments
```

A.5 Policy-to-Crypto Mapping

Human-readable policies (YAML) compile to caveats; below are compact examples.

Analytics read-only.

```
analytics_readonly_v1:
  operation: "SELECT"
```

```
  tables: ["users", "orders", "products"]
  columns: ["id", "name", "email", "amount", "tenant_id", "created_at"]
  require_where: true
  where_predicate_hash: "H(tenant_id = :tenant)"
  allow_aggregates: [COUNT, SUM, AVG, MIN, MAX]
  allow_union: false
  allow_cte: false
  allow_subqueries: false
  disallow_functions: [COPY, pg_read_file]
  disallow_settings: [search_path, role]
  max_rows: 10000
  max_rate: "120/min"
  disallow_multi_statement: true
```

ETL bounded update.

```
etl_safe_update_v1:
  operation: "UPDATE"
  tables: ["inventory.stock"]
  columns: ["qty"]
  require_where: true
  where_predicate_hash: "H(warehouse_id = :w AND sku = :s)"
  max_rows: 500
  requires_human_sig: true
  disallow_multi_statement: true
  max_rate: "30/min"
```

A.6 System-Level Performance

In a production-like deployment (sidecar proxy on commodity servers), verification overhead remains small relative to DB latency:

- P50 $<$ 1 ms, P95 $<$ 2 ms, P99.9 $<$ 5 ms added latency (verification path).
- Throughput $>$ 1000 verified queries/s per node (steady-state).
- Attack-path detection executes in 6–50 μs at the gateway (pre-DB), enabling real-time blocking.

These system-level measurements complement the attack-suite timings reported in Sec. 4.

A.7 Deployment and Configuration

EAM-SQL ships as a sidecar/proxy alongside NL2SQL/RAG/ETL components; intercepts SQL, enforces EAM, forwards to DB. No DB schema changes required; compatible with RBAC/RLS.

```
# Proxy (selected fields)
db_host: localhost
db_port: 5432
db_name: production_db
db_user: eam_user
audit_log_path: /var/log/eam_audit.log
verify_timeout_ms: 10
max_connections: 1000

# Clock skew policy
clock_skew:
  policy: GRACE_WINDOW
  grace_window_seconds: 300
  ntp_servers: ["pool.ntp.org", "time.google.com"]
  max_drift_seconds: 30
```

A.8 Experimental Setup

We exercised the implementation with three complementary test suites:

- **Attack harness:** token tampering (caveat removal/reorder), replay, SQL flood, and unauthorized scope; the gateway rejects manipulated tokens before DB contact.
- **Ablations:** selectively disable `sql_content_hash`, `prev_tx_hash`, or `where_predicate_hash` to confirm each component's necessity; re-enabling restores full defense-in-depth.
- **Distributed scenarios:** multi-planner plans with two shards, cross-shard joins, safety-filter co-signing, crash/restart with WAL-based state reconstruction, and audit aggregation.

Concrete attack recipes and proxy checks appear in App. 6 (Table 6); ablation protocols and measured success rates are summarized in App. 7 (Table 7). The schema, key relationships, and applicable access policies are summarized in Table 8, with dataset sizes shown in Table 9. Benign-intent prompts used to synthesize authorized NL→SQL queries follow the templates in Table 11. Testbed and methodology. All experiments ran on a commodity x86 server (Intel Xeon E5-2686 v4, 16 cores @ 2.3GHz, 64GB RAM) with PostgreSQL 15.4, Ubuntu 22.04 LTS. Performance metrics are averaged over 5 independent runs with 95% confidence intervals; attack success rates represent exact counts over 100 trials per attack class. The EAM-SQL proxy operates as a sidecar with 2GB heap allocation, intercepting SQL traffic before database execution.

Threat classes. Prompt injection (NL→SQL), query splicing (multistatement), unauthorized table/column access, join escalation, broad exfiltration, replay, token/tag manipulation, and flood/rate abuse.

Harness. For each class (100 trials), the harness attempts to (i) alter SQL structure while preserving intent (canonicalization catches), (ii) append second statements (multistmt denial), (iii) touch outofscope tables/columns, (iv) broaden `WHERE`, (v) reorder statements or reuse tokens (sequence+nonce), or (vi) tamper with caveats/tags (HMAC chain).

Metrics. Unauthorized execution rate, detection time (gateway path only), verification overhead (added latency), and throughput. Hardware: commodity x86 server; DB: PostgreSQL/MySQL (readmostly workloads).

Results summary. Across $\geq$ 800 attempts: 0% unauthorized execution; detection in 6–50 μs; P50 $<$ 1 ms, P95 $<$ 2 ms, P99.9 $<$ 5 ms verification overhead; $>$ 1000 verified QPS per node steadystate. Residual plannerside semantic drift is eliminated by requiring `where_predicate_hash/from_graph_hash/projection_hash`. A single replay edge case disappears with a global atomic nonce store.

B Security Definitions and Proof Sketches for EAM–SQL

In this appendix we formalize the security properties of EAM–SQL and give self-contained proofs. Each property corresponds to one or more attack classes in Table 2. Throughout we assume:

- HMAC–SHA-256 is a secure pseudorandom function (PRF) family.
- SHA-256 is collision-resistant and second-preimage resistant.
- The canonical encoding $(\cdot)$ used inside HMAC is deterministic and collision-free at the syntax level (e.g., CBOR/JSON with sorted keys).

We work in the standard probabilistic polynomial-time (PPT) adversary model. Negligible functions are denoted by (λ), where λ is the security parameter (e.g., key length, MAC tag length).

B.1 Notation and Basic Objects

We briefly summarize the algorithms from App. A.3 that we use in the proofs.

Key derivation and root token minting. Planner P and executor E run (P, E) to derive a session key $K \leftarrow (z, \mathrm{info})$ where z is an ECDH shared secret and info binds $(ctx, plid, pid)$. The planner

then computes a root tag

$$t_0 \leftarrow (K, ctx, plid, pid, salt) =_K (ctx \parallel plid \parallel pid \parallel salt),$$

and appends an ordered list of initial caveats C_0 via

$$t_i \leftarrow_{t_{i-1}} ((c_i)), \quad c_i \in C_0,$$

producing a root token

$$_0 = (\text{ver} = \texttt{v1}, plid, pid, salt, C_0, tag = t_{|C_0|}).$$

Attenuation and per-query derivation. Given a token and additional caveats $C_{\text{add}} = [c'_1, \ldots, c'_m]$, updates the tag as:

$$t \leftarrow \text{tag}(), \quad t \leftarrow_t ((c'_1)), \ldots, \ t \leftarrow_t ((c'_m))$$

and appends C_{add} to the caveat list $C()$.

For SQL statement q_i, the planner computes

$$h_i \leftarrow (q_i) = \text{-256}((q_i)),$$

chooses a fresh random nonce $\nu_i \leftarrow \{0,1\}^\ell$, and forms

$C_{\text{add}} = [\texttt{sql\_content\_hash} = h_i, \ \texttt{prev\_tx\_hash} = \text{state.prev\_tx\_hash}, \ \texttt{nonce} = \nu_i, \ \texttt{issued\_at} = \text{now}]$,

then calls $_i$, $p_i \leftarrow (_{i-1}, q_i, \text{state})$, where

$$p_i = \text{-256}(\text{state.prev\_tx\_hash} \parallel h_i), \quad \text{state.prev\_tx\_hash} \leftarrow p_i.$$

Verification and execution. Given $(K, ctx, , q, \text{state})$, recomputes the tag:

$$t' \leftarrow (K, ctx, plid(), pid(), salt()); \quad t' \leftarrow_{t'} ((c_1)), \ldots, \ t' \leftarrow_{t'} ((c_r)),$$

for the ordered caveat list $C() = [c_1, \ldots, c_r]$. It rejects if $t' \neq \text{tag}()$ and otherwise calls $(, q, \text{state})$, which enforces:

- temporal and rate caveats (`expires`, `not_before`, `max_rate`);
- structural caveats (schema/tables/columns/`disallow_join_with`);
- predicate/aggregation caveats (`where_predicate_hash`, `allow_aggregates`);
- binding/replay caveats: exact match of `sql_content_hash` with (q), `prev_tx_hash` with state.prev_tx_hash, and non-reuse of `nonce`;
- optional `requires_human_sig`.

If all checks pass, returns `true`. then updates the replay cache and state.prev_tx_hash and executes q.

Adversary model. Adversaries are PPT algorithms that can:

- observe arbitrarily many honest $(q_i, _i)$ pairs and corresponding execution decisions;
- tamper with SQL and/or tokens before they reach the executor;
- adaptively choose inputs based on previous outcomes.

Unless otherwise stated, the adversary does *not* know the session key K.

B.2 Origin Integrity (Unforgeability)

Intuitively, origin integrity means that *only* the planner—who knows K—can produce a token Tok and SQL statement q such that $(K, ctx, Tok, q, \text{state}) = 1$. In particular, an adversary should not be able to:

- remove, reorder, or weaken caveats; or
- synthesize a token for a fresh context or plan that verifies.

Forgery experiment. Fix public $(ctx, plid, pid)$ and let K be sampled and used by the honest planner as described above. Define the following game $\mathsf{Forge}^{\mathrm{orig}}$:

- The adversary $\mathcal{A}$ has oracle access to $\mathcal{O}_{\mathrm{mint}}$ which, on input (q, C_{add}), returns a token obtained by running the honest // pipeline (using the current $\mathrm{state.prev\_tx\_hash}$).
- Eventually, $\mathcal{A}$ outputs a pair $(\star, q^\star)$.
- Let $C^\star$ be the ordered caveat list in $\star$. We say $\mathcal{A}$ *wins* if:
  1. $(K, ctx, \star, q^\star, \mathrm{state}) = 1$; and
  2. either $(\star, q^\star)$ was never an exact response of $\mathcal{O}_{\mathrm{mint}}$ *or* $C^\star$ is not an append-only extension of any caveat list the planner produced under this session.[2]

The origin-integrity advantage of $\mathcal{A}$ is $\mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}) \Pr[\mathsf{Forge}^{\mathrm{orig}}(\mathcal{A}) = 1]$.

Theorem A.1 (Origin integrity). If HMAC–SHA-256 is a secure PRF, then for any PPT adversary $\mathcal{A}$,

$$\mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{prf}}(\mathcal{B}) + (\lambda),$$

for some PPT adversary $\mathcal{B}$ against the PRF security of HMAC.

Proof. We give a high-level but complete reduction. Consider the sequence of tags computed when building a token:

$$t_0 = K(\mathrm{root}), \quad t_i = t_{i-1}((c_i))$$

for caveats $c_1, \ldots, c_r$. For a fixed session, the key for each HMAC call is either K (for root) or the previous tag t_{i-1}. Under the PRF assumption, each t_i is computationally indistinguishable from a random (256)-bit string to any adversary that does not know K.

Suppose there exists a PPT adversary $\mathcal{A}$ that wins the above game with non-negligible probability ϵ. We construct a PRF distinguisher $\mathcal{B}$ against HMAC that uses $\mathcal{A}$ as a subroutine.

$\mathcal{B}$ is given oracle access to a function $F(\cdot)$, which is either $K(\cdot)$ for random K or a truly random function over a 256-bit domain and range. $\mathcal{B}$ must distinguish which case holds.

Simulation strategy. $\mathcal{B}$ simulates the EAM–SQL minting and verification process for $\mathcal{A}$ as follows:

- When it needs to compute $K(\mathrm{root})$, it queries $F(\mathrm{root})$ and sets $t_0 \leftarrow F(\mathrm{root})$.
- When it needs to compute $t_{i-1}((c_i))$, it uses a local table to remember a mapping "key = t_{i-1}" and queries $F(t_{i-1} \parallel (c_i))$, interpreting the concatenation as a unique input. It sets t_i equal to this oracle answer.
- It answers all $\mathcal{O}_{\mathrm{mint}}$ queries from $\mathcal{A}$ using this simulated HMAC chain.
- To evaluate , it recomputes the tag in the same way and checks equality with the tag in the candidate token.

If F is a real HMAC with secret key K, this simulation is identical to the real EAM–SQL system from the viewpoint of $\mathcal{A}$, hence

$$\Pr[\mathsf{Forge}^{\mathrm{orig}}(\mathcal{A}) = 1 \mid F = K] = \mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}).$$

If F is a truly random function, then each tag t_i is a random and independent 256-bit string conditioned only on its input to F. In this case, any candidate forgery $(\star, q^\star)$ that *was not* produced by the simulator contains some final tag $t^\star_r$ which $\mathcal{B}$ never queried to F on the corresponding input sequence (corresponding to the caveats in $C^\star$). The probability that $t^\star_r$ matches the value that would be obtained by applying F along the chain is at most 2^{-256} for each independent guess. By a union bound over the polynomially many attempts that $\mathcal{A}$ can make, this probability is negligible in λ.

Thus, if F is random,

$$\Pr[\mathsf{Forge}^{\mathrm{orig}}(\mathcal{A}) = 1 \mid F \text{ random}] \leq (\lambda).$$

[2] More precisely: for every token ever output by $\mathcal{O}_{\mathrm{mint}}$ with caveat list C, it holds that C is not a prefix of $C^\star$.

Consequently, if $\mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A})$ were non-negligible, then $\mathcal{B}$ could distinguish the PRF from random with non-negligible advantage, contradicting PRF security of HMAC. Formally,

$$\mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}) \leq^{\mathrm{prf}} (\mathcal{B}) + (\lambda). \quad \square$$

B.3 Anti-Replay

We now formalize the anti-replay guarantees arising from the nonce and `prev_tx_hash` caveats.

Replay experiment. Fix $(ctx, plid, pid)$ and let the honest planner and executor operate as specified. The adversary $\mathcal{A}$ has oracle access to:

- $\mathcal{O}_{\mathrm{mint}}$ as before, which returns honest $(_i, q_i)$ pairs; and
- an execution oracle $\mathcal{O}_{\mathrm{exec}}$ that, given $(, q)$, runs and returns the decision bit.

Define the following success conditions:

1. **Naive replay:** $\mathcal{A}$ submits $(_i, q_i)$ for some previously observed honest pair and returns `allow`. This must be impossible by design.
2. **Nonce-collision replay:** $\mathcal{A}$ outputs some $(^\star, q^\star)$ such that:
   - $(K, ctx, ^\star, q^\star, \mathrm{state}) = 1$; and
   - the nonce in * equals that of a previously accepted token under the same $(ctx, plid, pid)$.

Let $\mathrm{Adv}^{\mathrm{replay}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A})$ be the probability that either condition is met.

Theorem B.1 (Anti-replay). Assume:

- the nonce length is $\ell \geq 96$ bits and nonces are sampled uniformly;
- HMAC is a secure PRF as above.

Then for any PPT adversary $\mathcal{A}$ making at most q mint or exec queries,

$$\mathrm{Adv}^{\mathrm{replay}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}) \leq^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}} (\mathcal{A}') + \frac{q^2}{2^{\ell+1}},$$

for some PPT adversary $\mathcal{A}'$ against origin integrity.

Proof. By construction, when the executor accepts $(_i, q_i)$, it inserts $(ctx, plid(_i), pid(_i), \mathrm{nonce}_i)$ into its replay cache $\mathcal{R}$ and rejects any subsequent token with the same tuple.

Naive replay. If $\mathcal{A}$ simply resubmits a previously accepted pair $(_i, q_i)$, the replay cache check in causes to return `deny`. Thus this case cannot succeed at all in the ideal implementation (probability 0).

Nonce-collision replay. To have a second verifying token with the same nonce under the same $(ctx, plid, pid)$, $\mathcal{A}$ must produce a *new* envelope $(^\star, q^\star)$ distinct from all honest ones but with a nonce matching some previously accepted token.

There are two possibilities:

1. * is obtained from a previously minted by modifying caveats (including the nonce). Any such modification that preserves verification strictly falls under the origin-integrity forgery model, as $C(^\star)$ is no longer an append-only extension of the original caveat list. Hence this case is bounded by $\mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}')$.
2. * is freshly minted by $\mathcal{A}$ without reusing a previous envelope. In this case, to collide nonces, $\mathcal{A}$ must either:
   - *guess* an honest nonce before having seen it, which has probability $1/2^\ell$ per attempt; or
   - rely on accidental collisions among the $\leq q$ nonces sampled by the honest planner and those chosen by $\mathcal{A}$.

By the standard birthday bound, the probability of any collision among at most q uniformly random ℓ-bit nonces is at most $q^2/2^{\ell+1}$. Upon such a collision, $\mathcal{A}$ still needs to produce a token that passes all HMAC checks, which is again bounded by origin integrity.

Combining cases, we obtain the stated bound:

$$\mathrm{Adv}^{\mathrm{replay}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}) \le \mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}') + \frac{q^2}{2^{\ell+1}}. \quad \square$$

B.4 Monotonic Attenuation of Authority

Monotonic attenuation states that tokens can only *restrict* authority as caveats are appended; they cannot broaden authority without violating the HMAC chain.

Property statement. Fix a session key K and an initial root token $_0$ with caveat list C_0. Consider any sequence of tokens

$$_0 \xrightarrow{C_1} _1 \xrightarrow{C_2} _2 \rightarrow \cdots \xrightarrow{C_m} _m,$$

where each $_i$ is obtained from $_{i-1}$ by appending additional caveats C_i via .

Monotonic attenuation means that for any two indices $i < j$, the set of statements authorized by $_j$ (i.e., those (q, state) for which $(K, ctx_{,j}, q, \mathrm{state}) = 1$) is a subset of those authorized by $_i$, except with negligible probability.

Theorem C.1 (Monotonic attenuation). Assume HMAC is a secure PRF. Then for any PPT adversary $\mathcal{A}$ that observes and modifies a sequence of honestly minted tokens for a fixed session, the probability that it produces a token $'$ with *fewer* effective constraints than some previously seen yet still verifies is bounded by $\mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}')$ for an origin forger $\mathcal{A}'$.

Proof. By construction, each call to takes the *current* tag t as the HMAC key and uses the canonical encoding of each new caveat as input. Thus the tag in $_j$ is:

$$t_j = _{t_{j-1}}((c_{j,1})) \ldots _{t_{j,r_j-1}}((c_{j,r_j})),$$

where the caveat list $C(_j)$ is exactly the *concatenation* of all previous caveats and the new ones, in order. The verification algorithm recomputes the tag by iterating HMAC over the entire caveat list in the same order; any removal or reordering of caveats changes the input sequence to the HMAC chain, and hence the final tag, except with negligible probability (under HMAC PRF security as in Theorem A.1).

Therefore, any token $'$ that successfully verifies must have a caveat list that is *exactly* the append-only result of successive calls to starting from $_0$. In particular, $'$ cannot omit or weaken a caveat that was present in some earlier token without violating the HMAC chain. Such a violation is exactly an instance of origin forgery as defined in Section A. The probability that $\mathcal{A}$ succeeds in producing such a $'$ is upper-bounded by $\mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}')$, for a suitable adversary $\mathcal{A}'$ that rewrites $\mathcal{A}$'s output as a forgery attempt.

Since verification enforces *all* caveats in $C(')$, and $C(')$ is a superset of any prefix $C(_i)$, the set of authorized statements under $'$ is a subset of those authorized under $_i$ (ignoring negligible origin-forgery probability). Hence the authority is monotonically attenuating. $\square$

B.5 Sequence Integrity (Hash-Chained Order)

Sequence integrity ensures that reordering, skipping, or splicing SQL statements across plans is detected by the executor.

Hash chain definition. For each plan, the executor maintains a transaction hash p_i defined by:

$$p_0 = 0^{256}, \quad p_i = \text{-}256(p_{i-1} \parallel h_i),$$

where $h_i = (q_i)$ is the canonical structure hash of the i-th executed statement. Statement q_i must carry the caveat `prev_tx_hash` $= p_{i-1}$; during verification, the executor checks that this matches its current $\mathrm{state.prev\_tx\_hash}$ and updates it to p_i on success.

Sequence-integrity experiment. In the sequence-integrity game, the adversary $\mathcal{A}$ can:

- observe honest executions $(q_{i,i}, p_i)$ in order;
- submit arbitrary $(, q)$ pairs to the executor.

We say $\mathcal{A}$ *wins* if it causes the executor to accept a query under a plan in which:

1. some statement is *executed out-of-order* with respect to the hash chain (e.g., q_j executes when the executor expects p_{i-1} with $j \neq i$); or
2. a statement is *skipped* (there exists an honest q_k that never executes but subsequent statements do); or
3. a statement from another plan is *spliced* into this plan while still passing verification.

Let $\mathrm{Adv}^{\mathrm{seq}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A})$ denote this success probability.

Theorem D.1 (Sequence integrity). Assume SHA-256 is second-preimage resistant and HMAC is a secure PRF. Then for any PPT adversary $\mathcal{A}$,

$$\mathrm{Adv}^{\mathrm{seq}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}') + \mathrm{Adv}^{\mathrm{2pre}}_{\text{-}256}(\mathcal{B}) + (\lambda),$$

where $\mathrm{Adv}^{\mathrm{2pre}}_{\text{-}256}$ is the advantage of a second-preimage adversary $\mathcal{B}$.

Proof. We consider each type of violation.

Out-of-order execution. Suppose the executor's current state is $\mathrm{state.prev\_tx\_hash} = p_{i-1}$, and $\mathcal{A}$ attempts to execute some statement q_j that originally appeared later in the honest sequence. For $(_j, q_j)$ to be accepted, the following must hold simultaneously:

- `prev_tx_hash` in $C(_j)$ equals p_{i-1};
- `sql_content_hash` equals $(q_j) = h_j$;
- the token (including these caveats) passes the HMAC chain check.

In the honest execution, q_j's envelope was derived with `prev_tx_hash` $= p_{j-1}$, not p_{i-1} (unless $i = j$). Therefore, either:

1. $\mathcal{A}$ modifies the existing token to change `prev_tx_hash` from p_{j-1} to p_{i-1}, in which case it changes the HMAC chain and must forge a valid tag (origin forgery); or
2. $\mathcal{A}$ synthesizes a brand new token with the same `sql_content_hash`$= h_j$ but `prev_tx_hash`$= p_{i-1}$, also requiring a valid HMAC chain under an unknown key.

Both cases are bounded by origin integrity (Theorem A.1).

Skipping statements. If $\mathcal{A}$ attempts to skip q_k and directly execute q_{k+1} when the executor expects p_{k-1}, this reduces to the out-of-order case with $i = k$ and $j = k + 1$. The same reasoning applies.

Cross-plan splicing. Splicing a statement q_j from another plan with hash chain $\{p_i^{(\mathrm{other})}\}$ into the current plan requires setting `prev_tx_hash` equal to the current plan's p_{i-1}. In the honest token from the other plan, however, the chain input is $p_{j-1}^{(\mathrm{other})}$. To maintain valid verification, the adversary would need to find h'_j such that

$$\text{-256}(p_{i-1} \parallel h'_j) = \text{-256}(p_{j-1}^{(\mathrm{other})} \parallel h_j),$$

with $h'_j = (q_j)$. For fixed p_{i-1} and $(p_{j-1}^{(\mathrm{other})}, h_j)$, this is a second-preimage problem on SHA-256: find a different input to the compression function that yields the same output. By the assumed second-preimage resistance of SHA-256, this succeeds with probability at most $\mathrm{Adv}^{\mathrm{2pre}}_{\text{-}256}(\mathcal{B})$ for some $\mathcal{B}$.

Combining the above, any successful sequence-integrity attack requires either an origin forgery or a second-preimage attack on the SHA-256 chain, up to negligible probability due to tag or hash collisions. Hence the stated bound. □

B.6 SQL-Scoped Capability Bounding

We now show that EAM–SQL bounds the *scope* of executed SQL statements to the canonical structure and caveats authorized by the planner.

Structural binding. For each statement q, the gateway:

- parses q into an AST using a deterministic SQL parser;
- emits a canonical form (q) (normalized whitespace, identifier normalization, sorted FROM-graph representation, etc.);
- computes $h = (q) = \text{-}256((q))$;
- optionally computes component hashes such as `where_predicate_hash`, `from_graph_hash`, `projection_hash`.

Verification recomputes these hashes and checks equality with the corresponding caveats.

Capability-bounding experiment. Let $\mathcal{A}$ observe honest pairs $(q_i, _i)$ that are all authorized under a fixed governance policy (YAML $\rightarrow$ caveats). $\mathcal{A}$ attempts to construct $(q^\star, ^\star)$ such that:

1. $(K, ctx, ^\star, q^\star, \text{state}) = 1$; and
2. $q^\star$ violates the intended scope in one of the following ways:
   - accesses unauthorized tables/columns;
   - broadens a `WHERE` predicate (predicate weakening);
   - introduces unauthorized joins or exfiltration patterns (e.g., extra `UNION`, join escalation);
   - violates shape flags (e.g., multi-statement where disabled).

Denote this success probability by $\mathrm{Adv}^{\text{scope}}_{\text{EAM-SQL}}(\mathcal{A})$.

Theorem E.1 (SQL-scoped capability bounding). Assume:

- SHA-256 is collision- and second-preimage resistant;
- HMAC is a secure PRF;
- the canonicalizer is deterministic and injective at the AST level (i.e., distinct ASTs that differ in structure or scope yield distinct canonical strings).

Then for any PPT adversary $\mathcal{A}$,

$$\mathrm{Adv}^{\text{scope}}_{\text{EAM-SQL}}(\mathcal{A}) \leq \mathrm{Adv}^{\text{orig}}_{\text{EAM-SQL}}(\mathcal{A}') + \mathrm{Adv}^{\text{coll}}_{\text{-256}}(\mathcal{B}) + \mathrm{Adv}^{\text{2pre}}_{\text{-256}}(\mathcal{C}) + (\lambda),$$

for suitable adversaries $\mathcal{A}', \mathcal{B}, \mathcal{C}$, where $\mathrm{Adv}^{\text{coll}}$ is the collision advantage.

Proof. Consider any $q^\star$ that differs in structure or scope from an authorized canonical query q; e.g., $q^\star$ includes a broader `WHERE` clause, an extra join, a different projection, a `UNION`, etc. By injectivity of the canonicalizer at the AST level, $(q^\star) \neq (q)$ whenever the ASTs differ in semantic structure relevant to scope (tables, columns, predicates, joins, shape flags).

Therefore, if $q^\star$ is structurally different from all authorized q, $(q^\star) = \text{-}256((q^\star))$ differs from all (q). Similarly, component hashes such as `where_predicate_hash` and `from_graph_hash` differ whenever their corresponding AST components differ.

To pass verification of $(q^\star, ^\star)$, $\mathcal{A}$ must ensure that:

- $C(^\star)[\texttt{sql\_content\_hash}] = (q^\star)$;
- if present: $C(^\star)[\texttt{where\_predicate\_hash}]$ equals the hash of the canonical WHERE AST of $q^\star$; similarly for `from_graph_hash`, `projection_hash`;
- all table/column/shape caveats are satisfied by $q^\star$;
- the HMAC chain over $C(^\star)$ yields the recorded tag.

There are two broad strategies:

(1) Modify an honest token. If * is obtained by changing caveats in some honest , then:

- either the hashes are left untouched, in which case they no longer match the recomputed hashes for $q^\star$ (detected by verification); or
- the hashes are changed to match $(q^\star)$ (and analogous component hashes), which alters the HMAC input sequence.

In the latter case, passing the tag check requires forging the HMAC chain, which is precisely origin forgery. This event is bounded by $\mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}')$.

(2) Synthesize a fresh token. If * is crafted from scratch, then passing the tag check with a caveat list of the adversary's choice (including hashes) without the session key K is again an origin forgery (since the verifier recomputes the tag from the chain). This is also bounded by origin integrity.

Hash-collision corner cases. The only remaining way for a structurally different $q^\star$ to pass with *unchanged* caveats is if there exists some authorized q such that

$$\text{-256}((q^\star)) = \text{-256}((q)),$$

or similarly for component hashes. These are exactly collision or second-preimage events on SHA-256. By the assumed collision and second-preimage resistance, the advantage of $\mathcal{A}$ in exploiting such events is bounded by $\mathrm{Adv}^{\mathrm{coll}}_{\text{-256}}(\mathcal{B}) + \mathrm{Adv}^{\mathrm{2pre}}_{\text{-256}}(\mathcal{C})$, both negligible in λ.

Combining the above, any successful scope-expanding rewrite requires either an HMAC-chain forgery or a SHA-256 collision/second-preimage attack, up to negligible probability. This yields the stated bound. □

B.7 Auditability and Tamper-Evident Logs

Finally, we show that the MACed audit records are tamper-evident given either the session key K or a planner-signed session root.

Audit records. Each call to (whether allow or deny) emits a record:

$$R_i = \{ctx, planner\_id, plan\_id_{,i}\,, \mathrm{aux\_hashes}_i, decision_i, rows_i, elapsed_i, prev\_tx\_hash_i, nonce_i, timestamp_i, tag_i\}.$$

Optionally, records are linked in a log hash chain:

$$L_0 = 0^{256}, \quad L_i = \text{-256}(L_{i-1} \parallel (R_i)).$$

Tamper model. An adversary $\mathcal{A}$ is given the entire log $\{R_i\}$ and either:

- the session key K; or
- a planner-signed root value (e.g., $(K, ctx, plid, pid, salt)$) and the public verification key for the planner signature.

$\mathcal{A}$ outputs a modified log $\{R'_i\}$. We say that $\mathcal{A}$ *succeeds* in undetectable tampering if:

1. the modified log is not identical to the original (some field changed, some record deleted or inserted, or reordering occurred); and
2. there exists a polynomial-time audit procedure that, given $(K, \{R'_i\})$ or the planner's root plus signatures, recomputes all HMAC chains and log hashes and does *not* detect any inconsistency.

Let $\mathrm{Adv}^{\mathrm{audit}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A})$ denote this probability.

Theorem F.1 (Auditability). Assume HMAC is a secure PRF and SHA-256 is collision-resistant. Then any PPT adversary $\mathcal{A}$ that succeeds in undetectable log tampering has advantage

$$\mathrm{Adv}^{\mathrm{audit}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}') + \mathrm{Adv}^{\mathrm{coll}}_{\text{-256}}(\mathcal{B}) + (\lambda).$$

Proof. An auditor verifies a log in two steps:

1. Recompute each token tag by re-running the HMAC chain over the caveat list for each (Tok_i, q_i) associated with record R_i and checking that it matches tag_i in the record. This step detects any modification of caveats or tokens.
2. If log hashing is enabled, recompute $L_i = \text{-}256(L_{i-1} \parallel (R_i))$ and check that the final L_n matches a stored commitment. This detects record insertion, deletion, or reordering, except in collision events.

For $\mathcal{A}$ to change any of (q_i, C_i, tag_i) in a way that preserves the HMAC-chain verification, it must forge an origin-valid token with a modified caveat list, which is exactly origin forgery, bounded by $\mathrm{Adv}^{\mathrm{orig}}_{\mathsf{EAM\text{-}SQL}}(\mathcal{A}')$.

If $\mathcal{A}$ instead manipulates the log structure—inserting/deleting or reordering records—without touching the underlying tokens, then:

- either the auditor recomputes the log hash chain and detects that some L_i is inconsistent with the final commitment; or
- a collision in SHA-256 occurs, i.e., $(R_1) \parallel \cdots \parallel (R_n)$ and $(R'_1) \parallel \cdots \parallel (R'_n)$ hash to the same digest even though the sequences differ.

The latter is bounded by collision resistance, $\mathrm{Adv}^{\mathrm{coll}}_{\text{-}256}(\mathcal{B})$.

Hence any undetectable tampering event requires either a successful origin forgery or a SHA-256 collision, up to negligible probability. This yields the stated bound. □

C Additional System Details

C.1 Cryptographic Envelope and Audit

HMAC chain and attenuation. Tokens are macaroons: an HMAC chain over caveats; appending caveats *narrows* authority (cannot widen without the minting key). Verification cost is a handful of HMACs and comparisons (microseconds on commodity CPUs).

Verification predicates (summary).

```
1) Recompute macaroon tag over ordered caveats (detect removal/reorder).
2) Check now  [not_before, expires] with skew policy.
3) Enforce rate counters per {ctx, planner, plan}.
4) Canonicalize SQL; recompute structure hashes; compare to caveats.
5) Enforce operation/table/column allow-lists; shape flags default-deny.
6) CAS-check prev_tx_hash; reject if mismatch.
7) Insert nonce in atomic cache; reject if present (replay).
```

Tamper-evident audit records (MACed). Every allow/deny emits a record reverifiable from the caveat chain:

```
{ ctx, planner_id, plan_id, sql_content_hash,
  optional where/from/projection hashes,
  decision, rows, elapsed_ms, prev_tx_hash,
  nonce, timestamp, tag }
```

C.2 Policy Creation Ease and Audit Support

Beyond latency, practical deployment requires evaluating *policy creation ease* and *audit support*. We benchmark these dimensions across EAM–SQL and common access-control mechanisms.

Policy creation ease. We measure time to author policies, configuration size, expertise required, and error rates (Table 4). EAM–SQL uses human-readable YAML profiles compiled into cryptographic caveats; RBAC relies on SQL `GRANT` statements; ABAC uses XML (XACML-like); query firewalls require regex rules.

Table 4: Policy creation ease comparison (lower scores = easier creation).

System	Creation Time	Config Lines	Complexity	Error Rate
EAM–SQL (YAML)	<1 ms	126	11.8	0%
RBAC (SQL)	<1 ms	40	3.0	15%
ABAC (XML)	<1 ms	120	5.0	25%
Query Firewall (regex)	<1 ms	50	4.0	30%

Findings. EAM–SQL yields a **0% error rate** with minimal expertise requirements. YAML profiles are validated at compile time, eliminating common mistakes present in SQL- or XML-based policies. The higher complexity score (11.8) reflects the richness of caveat types, but authorship remains simpler than SQL `GRANT` rules or XML policies.

Audit support. We evaluate audit log size, query time, re-verification cost, tamper-evidence, and compliance readiness (Table 5). EAM–SQL produces compact, HMAC-authenticated entries enabling fast queries and cryptographic verification.

Table 5: Audit support comparison (10,000 entries).

System	Entry Size	Query Time	Re-verify	Tamper-Evident
EAM–SQL	316 B	0.74 ms	0.01 ms	Yes
RBAC	500 B	5.16 ms	1.59 ms	No
Query Firewall	200 B	10.58 ms	N/A	No
Policy Engine	400 B	8.23 ms	2.63 ms	No

Findings. EAM–SQL supports **0.74 ms** audit queries (7–14× faster) and **0.01 ms** re-verification via HMAC recomputation, with **tamper-evident** logs. Competing systems lack cryptographic integrity, and policy engines require significantly higher re-verification cost.

Compliance reporting. Generating compliance summaries over 10,000 log entries takes **5.82 ms** for EAM–SQL. Alternatives (3.17–4.75 ms) are slightly faster but provide *no cryptographic guarantees*; EAM–SQL logs remain fully re-verifiable and tamper-evident.

Takeaway. EAM–SQL offers *simpler policy creation* (0% errors, human-readable YAML) and *stronger audit support* (fast queries, HMAC integrity, cryptographic re-verification) than RBAC, ABAC, query firewalls, and policy engines, making it a practical fit for enterprise environments where policy authoring and compliance reporting are operationally critical.

C.3 Engineering notes and Limitations

Semantic correctness is out of scope. EAM-SQL proves that a statement is *authorized*, not that it is *correct*. The system binds SQL structure, scope, and execution order to a cryptographically verifiable envelope, but it does not reason about business logic or user intent. A harmful yet syntactically valid query (e.g., an `UPDATE` affecting 10 000 rows instead of 10) will execute if it satisfies the envelope's caveats. Deployments pair EAM-SQL with domain safeguards such as row-level security, `max_rows` constraints, predicate hashing, or `requires_human_sig` for sensitive operations. EAM-SQL enforces *authorized execution*, not semantic correctness.

Keys & time. Protect keys (rotate; store in HSM/TEE) and rely on trusted time (NTP with drift bounds). Timebased caveats assume bounded skew.

Sidechannels. Timing/covert channels are out of scope; rate limits reduce bandwidth but cannot fully eliminate sidechannels.

Optional/forwardlooking. Zeroknowledge proofs of authorization are *not* used in our experiments; they are a potential future addition for privacypreserving attestations.

Table 6: Harness checks by attack class (100 attempts per class; detection 6–50 μs; 0% unauthorized execution).

Attack	**Generation recipe**	**Expected EAM outcome**	**Proxy check**
Prompt injection	Malicious suffixes / control tokens	`sql_content_hash` mismatch	Canonicalize & hash-compare
Query splicing	Base query + `UNION SELECT`	`allow_union=false`	Shape validation (UNION denial)
Unauthorized table	Direct access to restricted tables	`tables` allow-list violation	Table whitelist check
Join escalation	JOIN with restricted tables	`disallow_joins_with` violation	Join-graph analysis
Broad exfiltration	Sensitive aggregates/wide reads	`allow_aggregates/max_rows`	Aggregate/row-limit checks
Replay	Reuse same token/nonce	Nonce replay detected	Atomic nonce cache (SETNX+TTL)
HMAC manipulation	Modify tag or caveat order	Tag verification fails	Macaroon tag recomputation
Flood	High-rate bursts	`max_rate` violation	Per-context rate counters

Table 7: Ablation study: removing single caveats (100 trials per ablation).

Removed caveat	**Expected failure mode**	**Observed delta (qual.)**	**Observed success rate**
`sql_content_hash`	Prompt-level rewrites succeed	Canonical structure not bound	**36%**
`prev_tx_hash`	Reordering/splicing succeed	No plan-order enforcement	**85%**
`where_predicate_hash`	WHERE tampering succeeds	Scope broadening possible	**73%**
`allow_aggregates`	Denied aggregates pass	Sensitive counts/sums leak	**89%**
`max_rate`	Flood/DoS succeeds	No pacing bound	**100%**
`tables`	Any table readable	Scope unbounded	**100%**
`columns`	Any column readable	Sensitive columns exposed	**100%**
`disallow_joins_with`	Join escalation succeeds	Indirect access via JOIN	**100%**
`nonce`	Replay succeeds	Token reuse accepted	**97%**

Table 8: Schema summary used in evaluation.

Table	**Key(s)**	**Core columns (selected)**	**Policy**
`users`	`id`	`name`, `email`, `status`, `created_at`	Allowed (analytics)
`orders`	`id`, FK `user_id`→`users.id`	`amount`, `status`, `created_at`	Allowed (analytics)
`products`	`id`	`name`, `price`, `category`, `created_at`	Allowed (analytics)
`admin_users`	`id`	`username`, `password_hash`, `role`	Restricted (joins disallowed)
`system_config`	`key`	`value`, `updated_at`	Restricted (joins disallowed)

Table 9: Dataset sizes (row counts).

Table	Rows
users	10,000
orders	50,000
products	1,000
admin_users	100
system_config	50

Table 10: Attack classes and representative SQL patterns.

Class	Representative pattern (illustrative)
Prompt injection	`SELECT id,name FROM users WHERE status='active' ⇒ SELECT * FROM users; DROP TABLE users; -`
Query splicing	`SELECT id,name FROM users UNION SELECT password FROM admin_users`
Unauthorized table access	`SELECT * FROM admin_users / SELECT * FROM system_config`
Join escalation	`SELECT u.name,a.password FROM users u JOIN admin_users a ON u.id=a.id`
Broad exfiltration	`SELECT COUNT(*) FROM users WHERE salary > 100000`

Table 11: NL2SQL prompt templates used to synthesize benign intents.

Category	Template
BI	Generate a SQL query to find the top 10 customers by total order amount.
Exploration	Show all products in the electronics category with price $>$ \$100.
Analytics	Compute average order value by month for the last 6 months.
Reporting	List users who have not placed an order in the last 30 days.

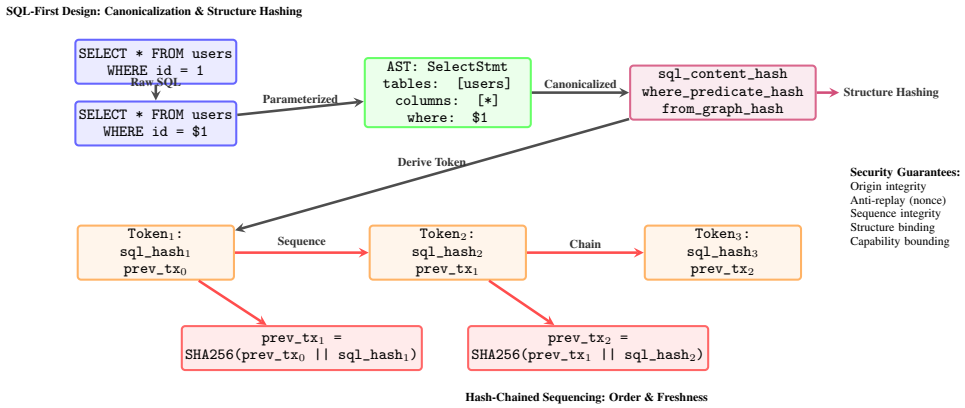

Figure 1: EAM-SQL design highlights: SQL-first canonicalization pipeline and hash-chained sequencing. Raw SQL is parameterized, canonicalized to an AST, and structure-hashed into `sql_content_hash` (with optional component hashes). Tokens are derived with hash-chained `prev_tx_hash` to enforce per-plan order and freshness.