Row-Level Security Hides the Row, Not the Time It Took to Skip It

September 29, 2026

A SaaS product stores one table of customer records, one row per person, isolated by a tenant column. Row-level security is enabled, a policy restricts each tenant to its own rows, and the policy is correct: no query by tenant A ever returns a row belonging to tenant B. The team audits it, writes tests against it, and moves on.

That guarantee covers the result set. It says nothing about how long the query took to produce it.

When a client-controlled predicate uses operators whose underlying functions are marked `LEAKPROOF`, the planner can push that predicate into an index scan, visit rows the policy will later withhold, and apply the policy as a post-filter. The client gets back an empty array either way. The clock does not.

LEAKPROOF Is a Property of Operators, Not of Your Policy

None of this is new. Postgres evaluates a row-security policy expression for each row before any condition from the user's query, with one stated exception: leakproof functions, which are guaranteed not to leak information, may be applied by the optimizer ahead of the row-security check (PostgreSQL, 2026 (https://www.postgresql.org/docs/current/ddl-rowsecurity.html)).

That exception exists for performance. Forcing the policy to run first can prevent the planner from using an index on the queried column, leaving a sequential scan. The `LEAKPROOF` marking is the escape hatch: many ordinary equality and ordering operators are backed by functions with that marking, depending on the operator and data type, so a predicate built from those comparisons can be reordered ahead of the policy and served by an index.

The consequence is stated plainly in the same documentation set: invisible tuples are never handed to possibly-insecure functions. The same page notes that a user may still infer things about unseen data by reading the query plan or measuring how long queries take, and that an attacker may learn something about the volume of unseen data, its distribution, or its most common values (PostgreSQL, 2026 (https://www.postgresql.org/docs/current/rules-privileges.html)).

Consider a tenant-isolation policy that reads the tenant from a session claim:

```
CREATE POLICY tenant_read ON records FOR SELECT
  USING (tenant_id = current_setting('request.jwt.claims', true)::json ->>
'tenant');
```

Run a range probe against an indexed column on a test instance and ask the planner what it did. A vulnerable plan might look like this (illustrative, not a measured plan for the policy above):

```
Index Scan using records_subject_ref_idx on records (actual rows=0)
  Index Cond: ((subject_ref >= '...') AND (subject_ref < '...'))
  Filter: (tenant_id = ...request.jwt.claims...)
  Rows Removed by Filter: 1
```

The index condition ran first. Postgres fetched one row from the heap, and the policy dropped it. The response body is `[]`. A probe whose range brackets nothing removes zero rows and returns the same `[]`, having done strictly less work.

That difference in work is what makes an existence oracle possible. `Rows Removed by Filter` is one falsifiable artifact: an operator with access to `EXPLAIN (ANALYZE)` can test the actual policy, index, query, role, and data to see whether the client-controlled predicate reaches rows that the policy later discards. A plan with a different shape doesn't establish this mechanism.[1]

The Existence Oracle Is Old; Binary Search Is What Made It Matter

The timing channel itself was characterized by Dar, Hershcovitch, and Morrison (2023) (https://doi.org/10.1145/3588943) and independently by Rasin et al. (2024) (https://doi.org/10.1145/3676288.3676306). Both showed that query execution time reveals whether an RLS-protected column contains an attacker-chosen value. The response from the ecosystem was muted, and understandably so: testing candidate values one at a time is useless against a high-entropy domain. Probing a nine-digit identifier linearly is a billion queries.

Espiritu and Cash (2026) (https://www.usenix.org/system/files/usenixsecurity26-espiritu.pdf) removed that limitation. They showed the oracle survives rich predicates: ranges, set membership, and conjunctions, not just equality. Once ranges work, the attacker binary-searches the domain instead of enumerating it. Recovering the set of values present in a column takes at most on the order of $|S| \cdot \log|D|$ oracle calls rather than $|D|$, where $|S|$ is how many distinct values actually appear and $|D|$ is the size of the domain. Locating one value in a

billion-value domain takes about 30 levels of search, before calibration and repeated timing measurements; recovering every present value takes more. Conjunctions then stitch recovered values across columns back into whole rows.

Three results from that evaluation matter for anyone assessing their own exposure:

- **Noise is not necessarily a defense.** Under the paper's join-based policy, the oracle exceeded 99.9% accuracy at up to 95% CPU load with a handful of repetitions per probe. Its inline session-setting policy was substantially noisier. Cross-region separation of roughly 3,000 km reduced accuracy by a fraction of a percentage point in the reported test; repetitions reduced the loss.
- **Cardinality contributes to the signal.** The timing delta can track the size of the intermediate result before the policy runs, because more candidate rows can mean more policy evaluations. A value carried on one withheld row may produce a weaker signal than one repeated across hundreds of rows. A denormalized ledger or event table can therefore amplify the mechanism when a probe matches many withheld rows. Its exposure still depends on the policy cost and the chosen plan.
- **Errors compound.** Binary search over a billion-value domain is about 30 dependent oracle calls. At 99.9% per-call accuracy that is roughly 97% end to end; at 99.0% it falls to about 74%. Oracle accuracy decides whether the attack works at all.

The engine can return precisely the rows its policy permits while leaking information through the cost of not returning the others. The issue is a confidentiality side channel even when result-set enforcement works as designed.

Auto-Generated REST Layers Hand Clients the Rich Query Interface

The published work is careful about its threat model. Its strongest results assume direct access: an adversary who can submit arbitrary SQL, such as an analyst with a database role or an attacker inside a compromised internal service. It also discusses app-mediated access, where a trusted application may expose a fixed set of query templates and the attacker can only reach the database through that narrow interface. The authors explicitly leave reconstruction through that kind of constrained interface as future work.

App mediation doesn't make the query interface narrow by itself, and an auto-generated REST layer shows why.

PostgREST, the API layer behind Supabase's Data REST API, translates URL filters into database predicates. Its documented operators include `eq`, `gt`, `gte`, `lt`, `lte`, `neq`, `in`, `like`, and `ilike`; conditions combine with AND by default, with explicit `or`, `and`,

and negation available (PostgREST, 2026 (https://docs.postgrest.org/en/latest/references/api/tables_views.html)). A client can express a bracketing range as `?and=(subject_ref.gte.X,subject_ref.lt.Y)` and add predicates on other exposed columns. Those are the query shapes the paper's amplification needs, though how they plan and time depends on the deployment. Listing an API operator does not establish that its underlying PostgreSQL function is `LEAKPROOF`.

Unless the team restricts its exposed schema or adds a narrower interface, there is no application-defined query template between a client's URL and PostgREST's generated query; database grants and RLS control access to the rows. Supabase places the API behind a gateway and key authentication. It intentionally exposes a publishable API key to the browser; signed-in clients also send their own user JWT (Supabase, 2026 (https://supabase.com/docs/guides/getting-started/api-keys)). The caller can therefore be an ordinary client of a correctly configured system, not an intruder.

The mediation is real, but the query interface can still be broad. An architecture chosen to avoid hand-written backend CRUD may expose the predicates needed to turn an existence oracle into a search. That establishes the prerequisites, not an exploit; nothing in this post was measured against PostgREST or Supabase. Whether the oracle survives an HTTP path is the open question, and network noise may be less of an obstacle than it looks. Part 2 (/blog/network-jitter-raises-the-price-of-the-oracle/) measures it.

Network Jitter May Not Be the Defense It Looks Like

The published experiments used direct client-to-server connections, and the authors flag intermediaries such as load balancers as a source of variance they did not model. An HTTP path can add TLS and proxy processing, request scheduling, connection-pool queueing, and public-internet jitter. The client still waits for the database result, but those components add time that may obscure the query-execution signal.

None of those components necessarily removes the engine-level channel. They may raise the variance an attacker has to average out, increasing the number of requests needed and making oracle errors more likely.

That assumes the attacker measures absolute response times. Van Goethem et al. showed that an attacker doesn't have to (Van Goethem et al., 2020 (https://www.usenix.org/conference/usenixsecurity20/presentation/van-goethem)). Their timeless timing attacks send two requests so they arrive at the server together, then observe which response returns first. Because the result depends only on response order, network jitter largely drops out.

Whether that carries over to a REST-fronted database is untested. The technique only works if both probes are handled identically until the query runs, so that query cost alone decides which response returns first. A REST stack adds two places where that can break. A gateway that receives both probes over HTTP/2 but forwards upstream over HTTP/1.1 has to send them on separate connections, and one may get a head start. PostgREST then runs queries through a limited pool of database connections; if only one is free, the second probe waits for the first to finish, and its response comes back later regardless of cost. Either step can decide response order before the query does. The test described below is built to measure exactly that.

I tested it on infrastructure I control. The client reaches Caddy over HTTP/2 and TLS 1.3 on a persistent connection, and Caddy forwards to PostgREST over HTTP/1.1. Part 2 (/blog/network-jitter-raises-the-price-of-the-oracle/) reports the results: the oracle survives the path, and noise changes the number of requests it costs, not whether it works.

Encryption at Rest Does Not Close the Channel

Transparent data encryption is a reflexive answer to “but the column is sensitive.” The tested disk-encryption configuration did not close this channel. PostgreSQL 18 has no native TDE; the paper’s authors reproduced their microbenchmarks with LUKS2 enabled on the database directories and found the timing distributions essentially unchanged. Cached queries do not incur additional disk I/O from LUKS2, so it added no noise that would help a defender in those tests.

Disk encryption is transparent to the query engine by design. The column therefore remains plaintext-ordered and range-queryable inside the index, which is exactly the property the binary search consumes. Encryption at rest defends against someone who walks off with the disk. It has no opinion about a legitimate session asking ordered questions.

The Fix Binds the Tenant Into the Index, Not Just the Policy

The generic mitigations are unattractive. Constant-time evaluation of arbitrary queries is impractical, and delay padding requires estimating worst-case execution time for variable workloads while giving up throughput. Physically separating tenants into distinct indexes or databases removes this shared-index channel, at the cost of the consolidation that motivated shared-index multi-tenancy.

A more targeted fix makes the tenant predicate part of the same index access as the queried attribute: a composite index led by the tenant column, plus a policy the planner can see into instead of an opaque function call. In the paper's benchmark, that combination brought the oracle's accuracy to about a coin flip; either change alone did not. That benchmark policy was keyed on `current_user`, which doesn't carry over to REST-fronted deployments where many users share one database role; there, the tenant has to come from the request, such as a JWT claim.

The approach has limits. It covers policies reducible to a leading-index constraint; complex per-row logic may not fit. Composite indexes on many filterable columns add storage and write costs, roles that bypass RLS may see slower queries, and any remaining access path that scans other tenants' values needs its own review.

Two measures belong alongside it. Narrow client-composed filtering on sensitive columns by exposing a bounded endpoint and revoking direct access to the underlying table, keeping in mind that PostgREST can still filter views and some row-returning functions. And treat probe volume as a weak detection signal: a targeted lookup needs far fewer requests than full reconstruction, so rate limiting raises cost without closing the channel.[2]

Key Takeaways

1. RLS guarantees the correctness of the result set. It makes no claim about execution time, and the Postgres documentation says so directly.
2. Operators backed by `LEAKPROOF` functions can be pushed ahead of the policy for performance. That reordering can create the timing channel when the resulting plan visits withheld rows.
3. Equality-only probing was impractical against high-entropy domains. Range predicates turn the same oracle into a binary search, and conjunctions reassemble whole rows.
4. A predicate matching more withheld rows can yield a stronger signal; denormalization may amplify it, depending on the plan and policy cost.
5. Auto-generated REST APIs can give ordinary clients rich predicates. Whether recovery works through an HTTP deployment is measured in Part 2 (/blog/network-jitter-raises-the-price-of-the-oracle/).
6. The authors' LUKS2 disk-encryption test left the timing distributions essentially unchanged.
7. For the paper's function-wrapped policy, a composite tenant-leading index plus a planner-visible policy subquery neutralized the tested oracle. Other policy shapes require their own plan and timing tests.

References

Dar, C., Hershcovitch, M., & Morrison, A. (2023). RLS Side Channels: Investigating Leakage of Row-Level Security Protected Data Through Query Execution Time. Proceedings of the ACM on Management of Data, 1(1). https://doi.org/10.1145/3588943

Espiritu, Z., & Cash, D. (2026). Plaintext Recovery Against Post-Filtering Access Control. 35th USENIX Security Symposium. https://www.usenix.org/system/files/usenixsecurity26-espiritu.pdf

PostgreSQL Global Development Group. (2026). Row Security Policies. PostgreSQL 18 Documentation. https://www.postgresql.org/docs/current/ddl-rowsecurity.html

PostgreSQL Global Development Group. (2026). Rules and Privileges. PostgreSQL 18 Documentation. https://www.postgresql.org/docs/current/rules-privileges.html

PostgREST. (2026). Tables and Views: Horizontal Filtering and Operators. PostgREST Documentation. https://docs.postgrest.org/en/latest/references/api/tables_views.html

Rasin, A., Herbick, J., Lenard, B., Scope, N., & Wagner, J. (2024). On Vulnerability of Access Control Restrictions to Timing Attacks in a Database Management System. SSDBM. https://doi.org/10.1145/3676288.3676306

Supabase. (2026). Data REST API; API keys. Supabase Documentation. https://supabase.com/docs/guides/api; https://supabase.com/docs/guides/getting-started/api-keys

Van Goethem, T., Pöpper, C., Joosen, W., & Vanhoef, M. (2020). Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. 29th USENIX Security Symposium. https://www.usenix.org/conference/usenixsecurity20/presentation/van-goethem

Footnotes

1. `EXPLAIN (ANALYZE)` on a policy-protected table is the diagnostic, but plan visibility is itself listed in PostgreSQL's discussion of security-barrier views as a possible inference channel. PostgREST's plan endpoint is disabled by default. Whether untrusted clients can see plans depends on deployment configuration. ↩
2. The published tuple-assembly results show that conjunctions are most stable when built starting from the most selective attribute. A targeted lookup against a known identifier can be cheaper and quieter than full-table reconstruction, making volume-based detection less reliable. ↩