# Network Jitter Raises the Price of the Oracle, Not Its Existence

*October 6, 2026*

---

An employee leaves on bad terms, and his payroll records stay behind. They have to: the IRS requires employers to keep employment tax records for at least four years after filing (IRS, 2026 (https://www.irs.gov/businesses/small-businesses-self-employed/employment-tax-recordkeeping)), so three years of his pay stubs, each one carrying his Social Security number, remain in the payroll app's database. The app is a Supabase project that serves pay stubs through PostgREST, and the RLS policy on that table is correct. Every employee sees their own stubs and nothing else.

The attacker is a coworker who's still there, whether curious or holding a grudge of their own. They're an ordinary authenticated user, with a valid session, the same REST endpoint every employee calls, and no access to the database, its logs, or its query plans. Each request gives them two things: a response body and how long it took to arrive.

When they filter the pay-stub table for his rows (PostgREST, 2026 (https://docs.postgrest.org/en/latest/references/api/tables_views.html)), the body is `[]`: two bytes, every time, whether the filter matched a withheld row or nothing at all. Only the arrival time differs. About five hundred pairs of those identical responses were enough to recover the pseudo-SSN, every time.

Part 1 (/blog/row-level-security-hides-the-row-not-the-time/) ended on an open question: does the attack survive the network? The published measurements were taken over direct database connections (Dar, Hershcovitch, and Morrison, 2023 (https://doi.org/10.1145/3588943); Espiritu and Cash, 2026 (https://www.usenix.org/system/files/usenixsecurity26-espiritu.pdf)). Over HTTP, a request also passes through a network path, a gateway, PostgREST, and a connection pool before it reaches the query, with the database adding noise of its own. Each of those adds delay, but none of them changes the work the policy does on rows it filters out, so that work still shows up in response times from the far side of a live deployment. More noise on the path doesn't hide the signal. It just means the attacker needs more requests to read it.

Each run has two views, and they don't mix. The attacker's view is timing alone: requests out, arrival times in, a candidate value at the end. The researcher's view adds the one thing only the person who seeded the target has, the true value, which is used to score each run after it finishes. Nothing from the researcher's view feeds back into the search.

## The Attacker Knows the Format, Not the Value

The target is a nine-digit value in the shape the Social Security Administration documents for its numbering scheme: a three-digit area, a two-digit group, and a four-digit serial, with areas limited to the range the agency has issued (SSA, n.d. (https://www.ssa.gov/employer/randomization.html)). That leaves 888,931,098 candidates, a 30-bit domain.[1] The attacker knows that format and nothing else about the value.

The value itself sits on the researcher's side. I seeded it as a deliberately voided identifier that can never be issued, and its digits stay redacted here as they are everywhere else in this project. The digits aren't the finding; the fact that `[]` gave them up is.

## The Fastest Trial Is the Honest One

Espiritu and Cash measured the size of the policy's per-row work over a direct database connection. Against a policy that does a per-row joined lookup, the fastest response to a probe matching nothing took 119 microseconds, one matching an authorized row took 474, and one matching a withheld row took 509, with standard deviations of 6 to 17 microseconds. The signal is the few hundred microseconds the policy spends evaluating a row.

Pulling the server's share of the time back out takes two steps: keep the fastest trial, then subtract a reference request that traveled the same path. Keeping the fastest trial works because delay on the path only ever pushes a response later. A congested hop can make it arrive late, but nothing can make it arrive early. That's why both papers, and this work, take the minimum of $k$ trials rather than the average. Slow trials drag an average around without moving the minimum, and with enough trials the minimum settles at the fastest the full round trip can go.

## Two Requests in One TLS Record Share Their Jitter

Even the fastest response still includes a trip across the network. The minimum strips out congestion and queueing, but what it settles on is the server's time plus the quickest the path can carry a request. The client receives those as one number, with no way to tell how much of it was Postgres and how much was the internet.

So the probe never travels alone.

Van Goethem et al. (2020) (https://www.usenix.org/conference/usenixsecurity20/presentation/van-goethem) hit the same wall with remote timing attacks, and their fix was to make two requests travel as one. It's the same primitive Kettle's single-packet attack later built on for race

conditions (/blog/race-conditions-read-write-gap/). The attacker sends two requests at once, packed into a single TLS record on one connection. The first is the probe, which asks whether the value falls in a given range. The second is a reference built to match no rows, so the policy never runs for it; it's the baseline the probe is measured against. The two leave the client together, cross the network together, and reach PostgREST together. Whatever the path does to one, it does to the other.

The network's share is still unknown, and it doesn't need to be known, because it's the same for both requests. Subtract the reference's time from the probe's and it cancels.[2] What remains is the extra work the database did for the probe: a few hundred microseconds of per-row policy cost when the probe's range covered a withheld row, close to nothing when it didn't.

The order of the two steps matters. The search takes the fastest probe response and the fastest reference response across all $k$ trials first, then subtracts. Subtracting pair by pair and keeping the smallest difference fails: any trial where the reference happens to arrive early produces a difference that's too small, sometimes negative, so the smallest difference keeps drifting lower as trials accumulate. Each side's own minimum has a floor to settle on, and the difference between two floors holds still.

---

## Each Run Calibrates Its Own Threshold

A margin of a few hundred microseconds only means something next to cases where the answer is already known. So before the search starts, each run measures two of them: the control and the anchor.

The control pairs the reference with itself: two requests that both match nothing. Whatever margin that produces is the pairing's own noise, the gap between two responses that should be identical. The anchor probes a row the attacker's own credential can read. The policy does the same per-row work there that it would do on a withheld row, and the answer is known to be "present." Every search probe's margin lands somewhere between those two; if it's closer to the anchor, the call is "present."[3] Neither measurement needs anything beyond the attacker's own credential.

Those two measurements also settle the question Part 1 left open about the REST stack. Part 1 named two places the pairing could break: a gateway that splits the pair across upstream connections, and a connection pool that makes one request wait on the other. Either would push extra delay onto one side of the pair, and the anchor's margin would sink toward the control's. On this path, the anchor's margin stayed clear of the control's in every run.

Put together, one oracle call looks like this:

1. Take the slice of the domain still in play and split it at the midpoint.
2. Build a probe whose range covers one half, using the same range filter from Part 1: `?and=(subject_ref.gte.LOW,subject_ref.lt.MID)`.
3. Send it $k$ times, each time coalesced with the reference.
4. Keep the fastest probe response and the fastest reference response, and subtract.
5. Compare that margin against the midpoint between the control and anchor margins.
6. If present, the value is in that half; if absent, it's in the other. Keep the half that holds the value and repeat.

Thirty of those calls take the domain from 888,931,098 candidates down to one.

---

## The Path Decides How Many Trials

How many trials are enough? That depends on two things each run measures for itself: how far apart the anchor and control margins sit, and how much variance the path adds to each measurement.

On any single trial, a response has some chance of arriving near its floor and some chance of arriving late. The minimum stays stuck high only if all $k$ trials arrive late. One late trial is likely; sixteen in a row is not. Each added trial multiplies the odds of staying stuck by the same fraction,[4] so the error falls exponentially in $k$: doubling $k$ roughly squares the error rate instead of halving it.[5]

A wide gap on a quiet path needs few trials; a narrow gap on a noisy one needs more. Each run solves for $k$ at the start, from its own control and anchor measurements. Across the runs here, that came out between 16 and 18.

The per-call error matters because the calls don't stand alone. Part 1 walked through the compounding: a 30-level binary search makes 30 dependent calls, and one wrong call sends every call after it into the wrong half. At 99 percent accuracy per call, only about three runs in four come out right. Here, each call's modeled error came in between 2.4 and 3.3 in ten thousand. Summed across thirty calls, that put each run's chance of failure under one percent.[6]

---

## The Request Count Is the Path's Price

Three runs against the same seeded value:

| run | derived $k$ | timed probe pairs | HTTP requests | verdict |
|---|---|---|---|---|
| A | 18 | 540 | 1,080 | correct value recovered |
| B | 17 | 510 | 1,020 | correct value recovered |
| C | 16 | 480 | 960 | correct value recovered |

The counts cover the search itself; each run's two calibration arms add a small fixed cost on top.[7] They're this instance's price under the load it carried while I ran it: about five hundred pairs on average. An instance under less load, or one behind a quieter route, settles each call in fewer trials, and a busier one needs more. The extra trials raise the cost, but they arrive at the same answer.

| run | $k$ | modeled per-call error | run-level (Boole, $\times 30$) |
|---|---|---|---|
| A | 18 | $2.48 \times 10^{-4}$ | 0.74% |
| B | 17 | $2.41 \times 10^{-4}$ | 0.72% |
| C | 16 | $3.29 \times 10^{-4}$ | 0.99% |

Run B reached a lower per-call error than run A with fewer trials. Its path measured cleaner, and because each run solves for $k$ against what it measures, a cleaner path clears the same error target with less work. The error figures are modeled, under the assumption of exponentially distributed excess delay and independent errors per call.

Under that same model, the request count comes down to one ratio: the gap between the anchor and control margins, divided by the jitter left after differencing. Call it signal over noise.[8] Solving the per-call error for a 1 percent run-level target gives the trials each call needs, and plugging in each run's measured ratio reproduces the 16 to 18 above.[9] Across a range of hypothetical paths:

| signal ÷ noise | $k$ | timed probe pairs | HTTP requests |
|---|---|---|---|
| 0.25 | 59 | 1,770 | 3,540 |
| 0.5 | 30 | 900 | 1,800 |
| 0.85–0.92 (these runs) | 16–18 | 480–540 | 960–1,080 |
| 1 | 15 | 450 | 900 |
| 2 | 8 | 240 | 480 |
| 5 | 3 | 90 | 180 |
| 10 | 2 | 60 | 120 |
| 15 and up | 1 | 30 | 60 |

The environment sets both halves of that ratio. A lightly loaded instance with an idle connection pool shrinks the noise; a costlier per-row policy, or a probe whose range covers more withheld rows, widens the signal. Requests scale linearly with noise over signal, so a path twice as noisy costs twice the requests. The floor is 60 requests, one pair per level, once the signal is roughly fifteen times the noise.

Scoring a run takes something the attacker never has. Because I seeded the value, the harness compares each run's emitted candidate against it and labels the run LEAK or WRONG-EMISSION. That check exists only to score the research. The search itself only ever sees timing: requests out, arrival times in, a candidate value at the end. An attacker running the same procedure against someone else's deployment has exactly that lane, and it's enough.

---

## Noise Changes the Price, Not the Outcome

Picture a coin that lands heads 51 percent of the time. Flip it ten times and it's indistinguishable from a fair coin; flip it a hundred thousand times and the bias is obvious. Each comparison in this search is that coin: right more often than a coin flip, by some edge. A noisier path shrinks the edge, and the cost grows faster than the edge shrinks. Halve the edge and each comparison carries a quarter of the information, so recovering the same value takes about four times the requests.[10]

That quadratic cost belongs to a comparator that reduces each trial to a yes-or-no vote. The paired minimum doesn't throw the arrival times away, and it uses the fact that noise only ever adds delay, which is why the request table above grows linearly instead. The coin is the pessimistic case: what the attack would cost if each trial could only vote.

As long as the edge stays above zero, every request still carries some information about the value.

What the attacker controls is budget. More requests buy resolution on a thinner edge. A search can also stop and resume cheaply: the only state it carries is the slice of the domain still in play, which halves at every level, so a restart picks up wherever the last one stopped.

---

## Single Values Were the Easy Case

Every oracle call here asked one question: is the value in this half of the domain? The only thing assumed up front was the domain itself, a nine-digit column in a format the SSA publishes, which fixes the search space at 30 bits before the first request goes out. The value came entirely from timing. Espiritu and Cash go further, running the same primitive against every attribute's domain and joining the results with conjunctions to rebuild whole rows a policy was hiding. The next post takes that on: starting from a harmless identifier an attacker could already know, like a name or an internal ID, and recovering the rest of that person's record, column by column.

---

## Key Takeaways

1. RLS timing leakage survives a live HTTP deployment. Identical `[]` responses still carry the policy's per-row cost.
2. Network delay only ever adds time, so the minimum of $k$ trials converges on the fastest round trip.
3. Coalescing the probe with a reference in one TLS record makes the network's share identical for both; subtracting their minimums removes it.
4. Control and anchor calibration set each run's decision threshold, and double as a black-box check that the pairing survived the gateway and connection pool.
5. Error falls exponentially in $k$, so a few more trials buy orders of magnitude in accuracy. The right $k$ comes from the path.
6. Noise sets the price in requests. The paired minimum's cost grows linearly with noise over signal; even a comparator limited to yes-or-no votes pays only quadratically. Either way, recovery stays within reach.

---

---

# 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), Article 89. 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

Internal Revenue Service. (2026). Employment Tax Recordkeeping. https://www.irs.gov/businesses/small-businesses-self-employed/employment-tax-recordkeeping

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

Social Security Administration. (n.d.). Social Security Number Randomization. https://www.ssa.gov/employer/randomization.html

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, 1985-2002. https://www.usenix.org/conference/usenixsecurity20/presentation/van-goethem


# Footnotes

1. $898 \times 99 \times 9999 = 888{,}931{,}098$, and $\lceil \log_2 888{,}931{,}098 \rceil = 30$. ↩
2. Each response's elapsed time is `ArrivedAt - HeadersSentAt`. Every stored and reported number is a delta: each pair's difference, or the margin between two floors. ↩
3. Concretely, a probe with margin $m$ is present iff $2m > c + a$, where $c$ and $a$ are the control and anchor margins. That is the midpoint test written without division, so rounding can't move the boundary. ↩
4. Treat each side's delay above its floor as independent and exponentially distributed with mean $\mu$. The minimum exceeds $t$ only if all $k$ delays do: $\Pr(\min_i X_i > t) = \prod_i e^{-t/\mu} = e^{-kt/\mu}$. The minimum is therefore exponential too, at rate $k/\mu$. ↩
5. Let $\lambda_p$ and $\lambda_r$ be the probe and reference rates, and $g$ the gap between the anchor and control margins. The error probability is then $\max\left(\frac{\lambda_p}{\lambda_p+\lambda_r} e^{-\lambda_r g/2},\ \frac{\lambda_r}{\lambda_p+\lambda_r} e^{-\lambda_p g/2}\right)$. This follows from the memoryless property: once one delay is fixed, the time until the other passes it is still exponential at its own rate. It's the standard competing-risks result from queueing theory and reliability analysis. ↩
6. If each of the 30 calls carries an independently allocated share of the error, Boole's inequality bounds the run's failure probability by the sum of the shares, the same reasoning as a Bonferroni correction across multiple comparisons. ↩
7. Calibration is two arms, control and anchor, each measured at the same $k$, so a run's full wire cost is the search count plus a fixed additive term. ↩
8. This is an amplitude ratio, not the variance-based SNR common in side-channel analysis. Under the exponential model in footnote 4, $\mu$ is both the mean and the standard deviation of the excess delay, so $g/\mu$ is the gap measured in noise standard deviations. A variance-based SNR would be its square, and requests would scale as the inverse square root of that SNR rather than linearly. ↩

9. With probe and reference seeing the same noise, footnote 5 reduces to a per-call error of $\frac{1}{2}e^{-kg/2\mu}$. A 1 percent run-level budget split across 30 calls is $1/3000$ per call, so $k = \lceil 2\ln(1500)\,\mu/g \rceil \approx \lceil 14.6\,\mu/g \rceil$, and the search costs $60k$ HTTP requests. The runs' reported errors imply $g/\mu$ of 0.85, 0.90, and 0.92, which give $k$ = 18, 17, and 16. ↩

10. Write $\epsilon$ for how far the comparator's correct-call rate sits above a coin flip. Expanding the binary symmetric channel's capacity $1 - H(1/2 - \epsilon)$ around $\epsilon = 0$ gives $C \approx 2\epsilon^2/\ln 2$ bits per shot, so the requests needed for a fixed amount of information grow roughly as $1/\epsilon^2$. For a single paired trial under footnote 9's model, the error is $\frac{1}{2}e^{-g/2\mu}$, so $\epsilon \approx g/4\mu$ for small $g/\mu$. Voting on single trials would therefore cost on the order of $(\mu/g)^2$ trials, against the paired minimum's $\mu/g$. ↩

---

https://blog.gtfo.dev/blog/network-jitter-raises-the-price-of-the-oracle/