The Chicken-and-Egg Problem of Zero-Knowledge User Lookup

March 17, 2026

Shipping every column encrypted. Private keys, recovery materials, document encryption keys—all opaque to the server, all derived client-side.

On paper, promising zero-knowledge was cut & dry: *the server never saw any of your plaintext data*.

Then I remembered the login endpoint.

Every auth request, whether that be login, salt retrieval, recovery, all started the same way: the client sent a plaintext email address so the server could compute `HMAC-SHA256(normalize(email), BLIND_INDEX_KEY)` and find the user. The blind index protected the email at rest. But TLS only guards the wire—once that request arrives, the worker process handled the plaintext email in memory. The server *saw* the identity it promised not to know.

This isn't unusual. It's actually standard practice[1]. Bitwarden's login sends email in the clear to its Identity server. 1Password transmits the account identifier before SRP begins. The term "zero-knowledge" has become conflated with end-to-end encryption of *data at rest*— but it says nothing about the metadata the server handles in transit. The server can't read your vault. But it sure as hell knows who's unlocking it.

I didn't like this. That said, I am also not the first to notice—researchers at ETH Zurich and Università della Svizzera italiana published a comparative security analysis in February 2026 that cataloged 27 successful attack scenarios across Bitwarden, LastPass, and Dashlane, all exploiting exactly these trust assumptions under a malicious-server threat model. The architecture is sound. The metadata leakage is the blind spot.

I wanted to close this gap. The question was deceptively simple:

> *How do you find a user without the server ever knowing who they are to begin with?*

This post covers the design and rationale. The math behind it — Poisson occupancy modeling, tail-bound overflow analysis, k-anonymity, and coupled constraint scaling — lives in Truncation Math: Why a Bucket Match Tells You Nothing (/series/).

The Paradox

Blind indexing is the established answer to "how do I search encrypted data without decrypting it." The canonical pattern—documented in Paragonie's CipherSweet (https://ciphersweet.paragonie.com/internals/blind-index) and the `blind_index` gem (https://ankane.org/blind-index-1-0)—is pretty straightforward:

1. Encrypt the sensitive value with a symmetric AEAD (e.g. AES-256-GCM)
2. Compute a keyed hash—HMAC, Argon2id, bcrypt—of the plaintext using a separate secret key
3. Store the hash as a blind index column; query against it for equality.

Effective against database breaches. But every implementation assumes the **server** holds the blind index key (BIK) and performs the derivation[2].

Do you see the issue? The plaintext *must* reach the server in order for the hash to be computed.

Now you're probably saying to yourself: "Well, dummy, just shift the derivation client-side then!" Key derivation at runtime is not the problem. The problem is **what the key is derived from**. Before authentication, the only inputs available to the client are:

- **A static bundle secret**: hardcoded in JS, extractable via devtools in minutes[3]. Not a secret.
- **A key derived at runtime from public inputs**: anyone with the source code reproduces the same key. Kind of an issue in an open-source project.
- **The email itself**: the only user-specific value available pre-auth. But, a blind index keyed solely by email is just a deterministic hash of a small, enumerable input space.

Here's why this is tricky, not obvious. Most client-side key derivation in ZK systems follows a pattern: the client sends an identifier, the server returns the salt, and the client derives a key locally. The server never sees the key. No chicken-and-egg because the *lookup* happens first (by plaintext email), and the *derivation* happens second.

A blind index flips that order. It **is** the lookup key, e.g. `WHERE login_bidx = $1`. It must be computable *before* any server contact, because the server can't do help you until it knows who it needs to help. Normal key derivation calls the server first, computes second. BIKs require computing first, calling second.

Every client-side approach collapses to the same weakness: **without a per-user secret, the derivation is reproducible**. And obtaining a per-user secret (like a password or recovery key) requires identifying the user first—which is the exact problem I was trying to solve.

> *The chicken-and-egg: you need a per-user secret to derive the lookup token, but you need the lookup token to find the user who holds the secret.*

What I Tried (and Rejected)

Bare hashing

Traditional password storage uses two layers to protect hashes: a **salt** and a **pepper**. The salt prevents precomputation, the pepper adds defense in depth—even with the full database, the attacker can't verify a guess without a separate server-side secret.

Both assume that the hash is a *verification* step: the server already knows which user it's looking at (found by email or username or other user-bound ID), retrieves their salt, and then checks whether the candidate password produces the stored hash. Find user first, verify second.

A blind index flips this. The hash *is* the lookup. There is no "find user first" step; the hash is how you find the user. This was a problem. A per-user salt requires finding the user's row to retrieve it, but the whole point was that I didn't know which row yet. And a pepper is exactly the server-sided model I was trying to eliminate. One leaked pepper, although that would indicate a much larger issue, would yield all BIKs confirmable. Every user's security would be correlated through a single point of failure. The goal was no shared secret in the derivation chain at all, such that each user would be on a cryptographically independent island.

Server-side isolation

HMAC via Enclave. Moving the HMAC key into an enclave seemed like a good idea. It keeps it out of the application memory and by nature if the enclave is compromised, I would have much larger issues to deal with. The problem is entirely architectural.

The enclave must be invoked on every login, salt retrieval, and recovery request—every operation that touches the blind index. In production you'd run multiple enclaves, but that shifts the problem from "single device" to "enclave fleet as an infrastructure tier"—every auth request still requires a synchronous round-trip to one of them. The implications compound: latency, horizontal scaling, availability, and blast radius—all of which were non-starters.

OPRF on parent server. An Oblivious PRF [RFC 9497 (https://www.rfc-editor.org/rfc/rfc9497)] hides the input from the server and the key from the client. Cryptographically elegant. This meant that I would be able to let the client blind the email, send it to the parent process (the hub that proxies vsock and KMS calls to the enclave), and receive a PRF evaluation; all without the parent ever learning the email. But the parent currently didn't hold any keys of its own; it proxies, it doesn't sign or encrypt. It introduces new key material to a process that currently holds none, promotes it from orchestrator to active auth participant, and puts it on every login path. The constraint was explicit: no new responsibilities or key material on parent. OPRF violates both.

Client-side keying

Password-derived blind index. The most intuitive approach: the user has the password at login time, so why not derive the blind index from it? `login_bidx = Argon2id(password, SHA-256("<service>-login-bidx-v1" || email))`. No server-side key needed, no static bundle key, chicken-and-egg solved—the user's own credentials are the key.

The problem was architectural. OPAQUE's stored registration record is uncrackable without the server's `oprf_seed`[4] — that's the entire point of adopting it. Eliminating stored password-verifiable material from the database entirely. Adding a password-derived blind index put it right back. That's not a tradeoff; that's a regression.

There was a secondary exposure too: an attacker with a database dump and a known email can compute `Argon2id(guessed_password, f(email))` for each guess and compare against the stored `login_bidx`. With strong password policies (zxcvbn >= 3, HIBP rejection), the brute-force risk was bounded — not catastrophic. But the regression was reason enough on its own.

Static bundle key. `Argon2id(email,static_key)` where `static_key` ships in the JS bundle. This is effectively public, with it being extractable from the bundle, minified source, or git history. Because the key is known, anyone can compute blind indexes for candidate emails. An attacker who obtains the database and has access to an email list simply runs `Argon2id(email, static_key)` for each address and checks for matches. At ~61ms/derivation[5], 1M emails takes roughly 17 hours. You see the problem.

Ed25519 from email. Derive a keypair from the email, use the public key as the index. Sounds great, but the private key is never used for anything. The public key is just a more expensive hash of the same input, with no security advantage over using Argon2id.

The Insight: OPRF-based Truncated Bucket Routing

The constraints:

- No static secret in the client bundle — it's not a secret
- No server handling of plaintext email — even transiently in worker memory
- No enclave or external service on the critical auth path — latency, availability, attack surface
- No bare hash without keying material — rainbow tables
- No password material in the stored lookup value — crackable from DB dump, regression from OPAQUE
- Server-side key material is acceptable *if a blinding protocol prevents it from ever seeing the input*

The last constraint is what unlocked this design for me. When I reject OPRF on parent originally, I rejected the *placement*, not the *protocol*. The worker, already the auth service, already on the login path, can hold the OPRF key without any architectural promotion. The blinding protocol means that the worker is able to evaluate a random-looking curve point and return the result. It never sees the email.

The Blinding Protocol

An Oblivious Pseudorandom Function [RFC 9497 (https://www.rfc-editor.org/rfc/rfc9497)][6] is a two-party protocol: the client holds an input, the server holds a secret key, and they jointly compute a PRF output such that the server learns absolutely nothing about the input and the client learns absolutely nothing about the key. The protocol works over any prime-order elliptic curve group where reversing a scalar multiplication has no known efficient solution.

The protocol works in three steps. The client hashes the email to a curve point and multiplies by a fresh random scalar `r`. The result is uniformly random and carries no information about the email. The server multiplies the blinded point by its OPRF key and returns the result; it never learns what input produced it. The client multiplies by `1/r`, canceling the blind factor that we just did. The deterministic PRF output is yielded without the server seeing the email or the client learning the key.

The server side OPRF key lives as an encrypted environment variable, separate from the database entirely. This is a deliberate design model for v1. The economics and timeline didn't justify a multi-service threshold deployment yet, and the failure mode is bounded—as the next section shows, truncation ensures that even with the OPRF key in hand, all the attacker confirms is that a candidate email *could* be in a bucket, not the specific user. The OPRF protocol naturally extends to a threshold scheme, where the key can be split across independent services, so no single compromised node holds the full key.

Truncation: Deliberate Collisions

The final step is counterintuitive: *throw away the precision*. Every database course drills the same lesson: collisions are bad, unique indexes are correct, and your lookup key should resolve to exactly **one** row. I chose to truncate the OPRF output, deliberately introducing an anti-pattern. Multiple emails map to the same bucket. The collisions are the feature.

A bucket is a deterministic identifier, more formally: the truncated output of our Hash function, an integer determined by the OPRF function, and the bit count. Every email that produces the same truncated value shares a bucket. The database uses this value as a lookup key: `WHERE login_bidx = 7,291` returns every user row whose OPRF output, after truncation, produced 7,291.

Okay, but why? Consider what happens without truncation. A unique blind index, even one derived via OPRF, lets an attacker with both the database and the OPRF key confirm whether a specific email is a user: compute `oprf_key * H("ethan@gtfo.dev")`, hash and look up the result. Confirmed.

Ethan is a user.

Truncation destroys that confirmation. Here's what the attacker actually sees with the aforementioned key and database dump at 14 bits:

1. Pick an email, any email from a breach corpus: `target@example.com`
2. Compute `truncate(SHA-256(oprf_key * H("target@example.com")), 14)` -> bucket **7,291**
3. Query `WHERE login_bidx = 7291` -> **6 rows** come back
4. The attacker now knows two things: I map to bucket 7,291, and 6 real user rows exist in that bucket. All 6 rows are real users. But ~61,000 emails from the corpus also produce the truncated value 7,291. The attacker cannot determine which 6 of those ~61,000 emails correspond to the 6 actual rows. Alice might be one of the 6 users, or she might not be.

Corpus size (`C`) is scope-dependent; a consumer platform might face billions of candidate addresses; a healthcare app serving Ohio providers narrows the pool to tens of thousands. `C` is the upper bound on how many email addresses an attacker would need to enumerate given the application's audience.

The math: given `k` truncation bits, `U` users, and a breach corpus of `C` emails: `2^k` buckets, `U/2^k` users per bucket, `C/2^k` candidates per bucket. Attacker confidence: `U/C` — the base rate. At 100k users against a 1B corpus, that's **0.01%** regardless of bit count[7]. Truncation doesn't change the base rate; it prevent the attacker from narrowing beyond it. Without truncation, a match is 100% confirmation. With it, a match tells the attacker nothing they didn't already know.

This guarantee is scope-dependent. The 0.01% figure above assumes a consumer-scale corpus. For a narrow-audience application, `C` shrinks and `U/C` rises. Consider that healthcare platform I was talking about earlier: the corpus of plausible email addresses might be 200,000, yielding `U/C = 2.5%`, a 250x weaker guarantee. Truncation still prevents the attacker from determining which emails correspond to which rows, but the bucket-match signal carries far more information when the corpus is small relative to the user count. Applications with narrow, enumerable user bases should factor corpus size into their bit selection and may need additional mitigations to compensate.

The tradeoff is latency: more users per bucket means more OPAQUE candidates to evaluate. But the opposite extreme is worse—*too many bits* destroys anonymity altogether.

More bits = more buckets, fewer users per bucket. At 25 bits with 100k users, λ drops to 0.003 — 99.7% of buckets are empty. An occupied bucket becomes a near-certain confirmation.

More padding = larger responses, more client-side compute. Padding hides bucket size from the *client* during login, but also means more candidate interactions. Scaling padding to compensate for sparse buckets means an increasingly large number of responses per login. Even at the sweet spot, the padding cap must cover the Poisson tail—if any bucket exceeds the cap, its response size leaks information.

Multi-Candidate OPAQUE

With truncation, `WHERE login_bidx = $bucket` returns N candidate rows instead of run. The server runs `opaque.server.startLogin()` against each candidate's registration record and sends all N response to the client (padded to a fixed count with dummy response to prevent the client from learning true bucket size).

The client loops through each response, calling `.finishLogin()`. On wrong candidates `null` is returned (no exceptions, no early exists), and `clientLoginState` is an immutable string reusable across calls. Internally, Argon2id runs unconditionally before the MAC check, so the cost of a wrong candidate is that of a correct one. Bye bye side-channel.

Total round trips: 3 (OPRF eval -> multi-candidate OPAQUE KE2 -> OPAQUE KE3). One more than standard OPAQUE. The OPRF round trip is a single scalar multiplication server-side.

The `finishLogin` calls are independent. The client processes all `PADDING_COUNT` slots (real + dummy) to prevent timing leaks. Latency scales linearly with the padding count.

Recovery: Unchanged

The recovery flow uses a different blind index entirely: `recovery_bidx = Argon2id(recovery_key, SHA-256("<service>-recovery-bidx-v1" || normalize(email)))`. 256 bits of CSPRNG entropy generated at registration. The sun goes first.

> *We have successfully answered which came first: the chicken or the egg. The client doesn't need a per-user secret to derive the lookup token. The server doesn't need to know what the client sent in order to validate over the wire.*

Does it Hold Up?

A design is only as good as its threat model. Let's take a look at the severity levels and their respective consequences:

Database only: Nothing. With the OPRF key still held server-sided, separate from the database, nothing is revealed.

Database + OPRF key: This assumes a single key, non-split. The attacker can map candidate emails to buckets, but each bucket matches ~61,000 candidates from a 1B corpus. Confidence that a specific email belongs to a specific user: ~0.0.1%. Still no way to link a specific email to a specific *row* within a bucket.

Database + OPRF key + OPAQUE seed: This is the worst-case scenario — and a failure that implies a breach well beyond this design's scope. With all three components, the attacker can fully decrypt the targeted user's data. What happens next is architecture-dependent. In this implementation: even at this

compromise level, the attacker still faces the user's password as the final barrier. Password policy (zxcvbn ≥ 3, HIBP rejection) ensures the exposure is per-user and bounded by password strength. The entity structure also requires two additional secrets split across enclaves, meaning full compromise demands more than this combination alone.

What It Costs

One extra round trip. Standard OPAQUE requires 2 round trips. The OPRF eval adds a third. But the OPRF round trip is a single scalar multiplication server-side. Network latency dominates, not the computation.

Multi-candidate OPAQUE latency. The client processes all `PADDING_COUNT` response slots, both real and fake, to avoid timing side-channels. The padding count must be ≥ λ + 5σ to avoid Poisson tail leakage. For a once-per-session operation, this is within acceptable range. With web workers on any non-dinosaur mobile phone, tablet, or computer, login range is between 50 and 500 ms.

Server-side OPRF key. Inherent to the design. Introducing another server-side var was acceptable within context, as previously explained the almost mathematically infeasible leakage of this key would only yield a very small % chance an email *could* be in a bucket.

Key Takeaways

1. **Blind indexing assumes server-side derivation:** The server must see the plaintext input. Moving derivation to the client creates a chicken-and-egg problem.
2. **Password-derived blind indexes are a regression:** They solve the chicken-and-egg but reintroduce stored password-verifiable material, undoing the security property that makes OPAQUE worth adopting.
3. **OPRF blinding separates input from key:** The server holds a key without ever seeing the input. The client blinds the email, the server evaluates, the client unblinds.
4. **Truncation converts exact match to approximate match:** Deliberate collisions ensure that a bucket match tells the attacker nothing beyond the base rate.
5. **Multi-candidate OPAQUE handles the ambiguity:** Try all candidates in a bucket; `finishLogin` returns `null` on mismatches.
6. **A database dump alone yields nothing:** No path to emails, passwords, or decryption keys. A strict improvement over the server-side HMAC model, where one leaked key compromised every user.

References

Biryukov, A., Dinu, D., & Khovratovich, D. (2021). RFC 9106: Argon2 Memory-Hard Function. IRTF. https://www.rfc-editor.org/rfc/rfc9106

Bourdrez, D., Krawczyk, H., Lewi, K., & Wood, C.A. (2025). RFC 9807: The OPAQUE aPAKE Protocol. IRTF. https://www.rfc-editor.org/rfc/rfc9807

Davidson, A., Faz-Hernandez, A., Sullivan, N., & Wood, C.A. (2023). RFC 9497: OPRFs Using Prime-Order Groups. IRTF. https://www.rfc-editor.org/rfc/rfc9497

de Valence, H., Grigg, J., Hamburg, M., et al. (2023). RFC 9496: The ristretto255 and decaf448 Groups. IRTF. https://www.rfc-editor.org/rfc/rfc9496

Jarecki, S., Krawczyk, H., & Xu, J. (2018). OPAQUE: An Asymmetric PAKE Protocol. IACR ePrint 2018/163. https://eprint.iacr.org/2018/163

OWASP. (2024). Password Storage Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

Paragonie Initiative Enterprises. (2023). CipherSweet: Blind Indexing Internals. https://ciphersweet.paragonie.com/internals/blind-index

Bitwarden. (2024). Encryption Key Derivation: Argon2id. https://bitwarden.com/help/kdf-algorithms/

Cloudflare. (2018). Validating Leaked Passwords with k-Anonymity. https://blog.cloudflare.com/validating-leaked-passwords-with-k-anonymity/

Meta Engineering. (2026). How Advanced Browsing Protection Works in Messenger. https://engineering.fb.com/2026/03/09/security/how-advanced-browsing-protection-works-in-messenger/

Scarlata, M., Torrisi, G., Backendal, M., & Paterson, K. (2026). Zero Knowledge (About) Encryption: A Comparative Security Analysis of Three Cloud-based Password Managers. IACR ePrint 2026/058. https://eprint.iacr.org/2026/058

Intruder. (2025). Secrets in Your Bundle(.js): The Gift Attackers Always Wanted. https://www.intruder.io/research/secrets-detection-javascript

Polášek, V. (2019). Argon2 Security Margin for Disk Encryption Passwords. Masaryk University Faculty of Informatics. https://is.muni.cz/th/yinya/thesis.pdf

Red Hat Research. (2019). How expensive is it to crack a password derived with Argon2? Very. https://research.redhat.com/blog/article/how-expensive-is-it-to-crack-a-password-derived-with-argon2-very/

Footnotes

1. This is standard practice even among products that market “zero-knowledge” encryption. Bitwarden’s Password Login Strategy authenticates via `POST /connect/token` on the Identity server, where the request includes the user’s email address in plaintext for account lookup [Bitwarden Contributing Docs, “Authentication”]. 1Password uses the Secure Remote Password (SRP) protocol, which avoids sending the password itself, but the email/account identifier is still transmitted to the server for user lookup prior to the SRP exchange [1Password Security White Paper, “A modern approach to authentication”; 1Password Blog, “How we use SRP”]. In both cases, the server knows *which user* is authenticating — the email is visible in transit to the worker. In February 2026, researchers at ETH Zurich and Università della Svizzera italiana (Scarlata, Torrisi, Backendal, Paterson) published “Zero Knowledge (About) Encryption: A

Comparative Security Analysis of Three Cloud-based Password Managers” [ePrint 2026/058], identifying 27 successful attack scenarios across Bitwarden (12), LastPass (7), and Dashlane (6) under a malicious-server threat model, with 2 additional scenarios against 1Password that the company confirmed were already documented in their Security Design White Paper [1Password Blog, “Zero knowledge vs. a malicious server”]. The paper will be presented at the USENIX Security Symposium in August 2026. ↩

2. CipherSweet’s architecture explicitly assumes a server-held key. The CipherSweet internals documentation states: “A blind index is calculated by using hash functions and/or key-stretching algorithms of the plaintext, using an appropriate key” (CipherSweet). Paragonie’s original 2017 blog post states: “The general idea is to store a keyed hash (e.g. HMAC) of the plaintext in a separate column. It is important that the blind index key be distinct from the encryption key and unknown to the database server” (Paragonie, 2017). The `blind_index` Ruby gem follows the same model (ankane, 2023). In all these implementations, the plaintext must reach the application server. ↩
3. Research by Intruder (2025) analyzed approximately 5 million single-page web applications and found secrets — API keys, tokens, and credentials — embedded in JavaScript bundles across a significant percentage of them [Intruder, “Secrets in your Bundle(.js)”]. The OWASP Top 10 Client-Side Security Risks lists “Sensitive Data Leakage” and specifically calls out “crypto secrets” and “API tokens” stored in “JavaScript variables” as a known vulnerability class [OWASP Client-Side Top 10]. ↩
4. RFC 9807 (Section 5) defines `credential_identifier` as the server’s lookup key (RFC 9807). The `@serenity-kit/opaque` library (v1.1.0) maps `userIdentifier` to `credential_identifier`. In the multi-candidate design, the client’s `finishLogin()` returns `null` on mismatches (MAC verification failure after unconditional Argon2id). No early exit, no timing side-channel. ↩
5. The 61ms figure is from native (non-WASM), CPU-bound benchmarking of Argon2id with parameters 64 MiB / 3 iterations / 4 parallelism on an Intel i7-13700K, single-threaded. This is a CPU-only figure — GPU-accelerated cracking of Argon2id at 64 MiB memory is severely constrained by memory bandwidth, making 17 hours a conservative (attacker-favorable) estimate for a CPU-only cracking rig. 1,000,000 derivations × 0.061 seconds = 61,000 seconds ≈ 16.9 hours. For broader context: research at Masaryk University (summarized by Red Hat Research) found that “it can take thousands of machines and hundreds of millions of dollars over ten years to crack an eight-character LUKS2 password using Argon2” (Red Hat Research). ↩
6. RFC 9497 (December 2023) specifies Oblivious Pseudorandom Functions (OPRFs) using prime-order groups. An OPRF is a two-party protocol: the client holds the PRF input, the server holds the secret key. The “obliviousness” property ensures the server does not learn anything about the client’s input during evaluation, and the client does not learn anything about the server’s key (RFC 9497). Meta’s Messenger team uses OPRF as a component of their Advanced Browsing Protection feature for private URL-checking within end-to-end encrypted chats. Their full construction layers additional techniques (Path ORAM, AMD SEV-SNP confidential computing, Oblivious HTTP) on top of OPRF to handle non-exact URL matching — the OPRF step alone is not sufficient for their use case (Meta Engineering Blog). ↩
7. The k-anonymity property of truncation: at 14 bits with 100k users and a 1B corpus (application-scoped universe of possible users), each bucket contains ~6 real user rows. ~61,000 corpus emails produce the same truncated value (1B / 16,384), but only 6 have actual database rows. The attacker cannot determine which 6 of the ~61,000 correspond to real rows. Confidence: `6 / 61,000 ≈ 0.01%` — equivalent to the global base rate `U/C`. Cloudflare’s password breach checking service uses a similar truncated-hash approach for k-anonymity (Cloudflare). ↩
