Encryption Protects the Column, Not the Schema

June 9, 2026

A product encrypts document bodies before storage and ships the keys somewhere out of the service's scope. The privacy claim is narrow but deliberate: we cannot read your documents. Ciphertext lands in the database. An attacker gains broad read access to prod and finds column after column of meaningless bytes.[1]

Then the attacker runs this query:

```
SELECT grantor_user_id, recipient_user_id, count(*) AS shared_documents
FROM document_grants
GROUP BY grantor_user_id, recipient_user_id
ORDER BY shared_documents DESC;
```

The documents are still unreadable. The sharing graph is not.

That gap is what LINDDUN calls Linkability: the schema answers questions about relationships between data subjects without the attacker ever touching a plaintext value. In almost every production schema, that linkability surface is wider than the privacy contract acknowledges, and it has nothing to do with the strength of the cryptographic guarantees on the content.

The API and the Schema Are Different Interfaces

Most security reviews focus on approved access paths: whether one user can call an endpoint to read another user's data, whether authorization checks are consistent, whether resolvers enforce tenant boundaries. Those questions matter. They are not the whole picture.

The API exposes the questions the product intended to support. A schema encodes a query surface. Not the one the product designed, but the one the storage layout permits. Plaintext relationships can be queried. Stable tokens can be counted. Precise timestamps build timelines. Search tokens cluster records that share a hidden term. The application controls what its API answers. The API is reviewed for what it exposes. The schema is reviewed for who can reach it. What it reveals to someone who already has a stolen backup, a misconfigured read replica, a compromised analytics pipeline — that is the question that falls between those two reviews.

LINDDUN's Linkability category formalizes what RFC 6973 calls correlation: combining pieces of information that, taken together, identify or relate to an individual (Cooper et al., 2013). A database dump is exactly that: a correlation engine with the ciphertext stripped out.

What the Documents Table Says Without the Ciphertext

Start with a simple table. One row per document, one column for the encrypted blob, everything else is operational metadata the application needs to do its job.

```
CREATE TABLE documents (
  id uuid PRIMARY KEY,
  owner_id uuid NOT NULL REFERENCES users(id),
  organization_id uuid NOT NULL REFERENCES organizations(id),
  ciphertext bytea NOT NULL,
  file_size bigint NOT NULL,
  mime_type text NOT NULL,
  created_at timestamptz NOT NULL,
  updated_at timestamptz NOT NULL
);
```

Pull a row out of the dump and read it without the ciphertext:

```
owner_id    | organization_id | file_size  | mime_type    | created_at
| updated_at
c56bbc50... | 4e1c...         | 12,847,392 | image/dicom  | 2026-04-12 09:14:02
| 2026-04-12 09:14:02
```

No decryption happened. The row still says: a specific user owns this object, that user belongs to a specific organization, the object is roughly thirteen megabytes of medical imaging, and it was uploaded at 9:14 on April 12th and has not been touched since. Every one of those facts lives in a plaintext column the application needed for operational reasons and stored without a second thought. OWASP's cryptographic storage guidance is direct on this: the best way to protect sensitive data is not to store it. Most teams hear that and go straight to the ciphertext column. The dump threat model extends it to every column.

`owner_id` is the easiest column in the world to justify. Authorization checks need it. Your billing department would be pissed if you removed it. Customer support needs to function somehow. Most schemas put it on every table without a second thought. To an attacker, it generates a leaderboard:

```
SELECT owner_id, count(*) AS document_count
FROM documents
GROUP BY owner_id
ORDER BY document_count DESC;
```

Join to organizations, and the most active tenants surface. Join to `created_at`, and you now have a sketch of a typical tenant's day at work. Join to `mime_type`, and you can take a guess at the type of work being done. None of those joins touched a ciphertext column. If the privacy claim is "we cannot read your documents," the result is technically fine. If the claim is anything stronger, you're lying to your consumers.

That is the line that must be drawn: which leakages are operational metadata you accept, and which are privacy claims the schema is silently contradicting.

Sharing Tables Are Social Graphs

To the engineer who wrote it, a grants table is access control plumbing. To a database-only attacker, every row is a directed edge.

```
CREATE TABLE document_grants (
  id uuid PRIMARY KEY,
  document_id uuid NOT NULL REFERENCES documents(id),
  grantor_user_id uuid NOT NULL REFERENCES users(id),
  recipient_user_id uuid NOT NULL REFERENCES users(id),
  role text NOT NULL,
  expires_at timestamptz,
  revoked_at timestamptz,
  created_at timestamptz NOT NULL
);
```

Three columns — grantor, recipient, document — are all it takes to reconstruct who works with whom, on what, and how often. Edge weights fall out of a group-by. Tightly connected users are working on the same things; sparse connections are not. The most widely shared documents surface from a count on active grants. Neither query touches a ciphertext column, and neither needs to.

The query that tends to get overlooked is not about active access at all:

```
SELECT recipient_user_id, count(*)
FROM document_grants
WHERE expires_at IS NOT NULL OR revoked_at IS NOT NULL
GROUP BY recipient_user_id;
```

Revoked and expired grants don’t disappear — they accumulate. That query recovers who had access and lost it: contractors whose visibility window closed after three weeks, employees who were offboarded, reorganizations that show up as access patterns before they show up anywhere else. Current grants show who has access. Revoked ones show who was trusted, for how long, and when that trust ended — often the more sensitive fact. Revoking access was supposed to end the story. The row says otherwise.

Search Tokens Still Snitch

Search is the hardest feature to ship under a zero-access storage model. Users still expect to find their documents. Some structure has to live server-side.

A common shape:

```
CREATE TABLE document_search_tokens (
  token bytea NOT NULL,
  document_id uuid NOT NULL REFERENCES documents(id),
  created_at timestamptz NOT NULL,
  PRIMARY KEY (token, document_id)
);
```

The token column has been derived with an HMAC or similar PRF. The database cannot tell that `0x9d...` means “foo,” “bar,” or “baz.” It can tell that the same hidden term appears across many documents.

```
SELECT token, count(*) AS matching_documents
FROM document_search_tokens
GROUP BY token
ORDER BY matching_documents DESC;
```

That gives the attacker frequency without the dictionary. Rare terms identify people.[2] Common terms describe the product’s center of gravity. Joining the token back to documents clusters records by search term. Joining further to owners or organizations groups users by shared concepts. No plaintext required.

The SSE literature has been explicit about this for nearly two decades. The standard SSE model defined by Curtmola et al., 2006 does not claim the server learns nothing; it defines security against a leakage function. More recent quantitative work by Boldyreva, Gui, and Warinschi, 2024 demonstrates that even carefully constructed SSE schemes leak information that can be practically exploited, and that leakage categories differ substantially between a one-time database dump and persistent access.[3]

Search tokens are not inert. They are frequency maps. If the database can group encrypted documents by a hidden term, that grouping belongs in the privacy model, not in a footnote.

Audit Logs Are Behavior, Not Plumbing

Audit logs get treated as defensive plumbing. To an attacker, they are timestamped behavior.

You write facts about what happened: who acted, what object they touched, whether the attempt worked, where it came from, when it happened.

```
CREATE TABLE audit_events (
  id uuid PRIMARY KEY,
  actor_user_id uuid,
  organization_id uuid,
  object_id uuid,
  object_type text NOT NULL,
  action text NOT NULL,
  ip_address inet,
  user_agent text,
  success boolean NOT NULL,
  created_at timestamptz NOT NULL
);
```

The document stays unreadable. The event record does not. It tells a story.

```
SELECT created_at, object_type, action, success
FROM audit_events
WHERE actor_user_id = $1
ORDER BY created_at;
```

That query does not recover a single sentence from a document. It recovers a day.

> *Login at 9:02. Document opened at 9:06. Failed export at 9:07. Retry at 9:08. Share at 9:30. Revoke at 10:11. Another failed export at 10:13. Admin action at 10:19.*

The attacker doesn’t know what the user read. They know how the user spent their day, where they encountered problems, which actions succeeded, and which did not. That is not metadata around sensitive data. That is the sensitive data. The NIST Privacy Framework makes the point clear: privacy risk is not limited to unauthorized access failures, but can arise from ordinary system operations across the data lifecycle.

Audit logs are the obvious case. The same pattern hides in any table that records workflow state. A four-column verification table does not store a single document, but its status column still tells an attacker who cleared checks, who was rejected, and whose account was revoked. The authors of the schema didn’t intend to publish that. The schema published it anyway.

The answer is not “delete the logs.” The answer is to stop pretending logs are outside the threat model. Ask what the dump gets for free, because every field of unnecessary precision, every plaintext identifier, and every event class kept past its useful window is paid for in blast radius. Logs aren’t just how you explain the breach afterward; in a database dump, they’re part of it.

Tokenization Hides Names, Not Edges

The natural response is to make identifiers look less useful. Replace plaintext UUIDs with derived tokens. The values stop looking like identifiers and start looking like noise. Done well, it is a real improvement. If the token keys stay outside the blast radius, a database-only attacker cannot reverse

the token and recover the original user or document ID.

The problem is that reversing the token is not necessary. A stable token reused across contexts is a join key. This is the standard pseudonymity failure mode named in RFC 6973 §6.1.2: pseudonymity is strengthened only when the same pseudonym is used across fewer contexts. The attacker does not know the person's name, but they do know that person's outline. Every table where the same token appears becomes joinable. Tokenization can hide identity while preserving correlation. If the same token can appear across multiple domains, it has become a product-wide identity handle, even if it looks random.

This is not a failure of HMAC construction — it's a failure of proper scoping. ENISA's pseudonymisation guidance frames this as the unlinkability goal: reducing the risk that privacy-relevant data can be linked across different data processing domains (ENISA, 2019). Per-purpose and per-tenant key separation mean that a token used for document grants cannot be automatically joined to one used in audit logs.[4] When that join becomes an operational necessity, it should be treated as intentional or acknowledged leakage, documented as such.

Schemas should be reviewed by the graph they expose, not by the entropy of the values they contain.

What a Schema Review Actually Produces

Reading the database the way an attacker would means treating the schema as an interface: not the API the documentation describes, but the one the storage design ships. The review is one question asked three ways: what can this column or table actually answer, does that answer align with the privacy contract the product has made, and if the answer must exist, can it be narrower?

The output of that review is not a passing grade. It is a posture assessment written in operational terms. It names what a database-only attacker obtains. For each table, it asks whether the leakage is intentional and in scope, or a silent contradiction of the privacy contract.

That last category tends to be larger than expected. `mime_type` stored in plaintext was a convenience, not a conscious decision to publish doc types. `created_at` at microsecond precision was the database default. `user_agent` in the audit log was copied from the request because it was just there. None of these were privacy claims. They became ones the moment they were stored.

The Contract Has to Name What It Covers

The phrase "we cannot read your documents" is technically narrow. It commits to the ciphertext itself and nothing else.

Encryption protects the values in the columns it covers. The rest of the schema answers questions the product may not have chosen to answer. The difference between a system that leaks by accident and one that leaks by design is whether those questions have been read, named, and accounted for in the privacy contract — not whether the cryptography applied to the content is secure.

References

Boldyreva, A., Gui, Z., & Warinschi, B. (2024). Understanding Leakage in Searchable Encryption: a Quantitative Approach. *Proceedings on Privacy Enhancing Technologies*. Link

Cash, D., Grubbs, P., Perry, J., & Ristenpart, T. (2015). Leakage-Abuse Attacks Against Searchable Encryption. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. Link

Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., & Smith, R. (2013). RFC 6973: Privacy Considerations for Internet Protocols. *RFC Editor*. Link

Curtmola, R., Garay, J., Kamara, S., & Ostrovsky, R. (2006). Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. *IACR Cryptology ePrint Archive*. Link

ENISA. (2019). *Pseudonymisation techniques and best practices*. European Union Agency for Cybersecurity. Link

Islam, M. S., Kuzu, M., & Kantarcioglu, M. (2012). Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS 2012)*. Link

NIST. (2025). Getting Started with the NIST Privacy Framework. *National Institute of Standards and Technology*. Link

OWASP. (2024). Cryptographic Storage Cheat Sheet. *OWASP Cheat Sheet Series*. Link

Footnotes

1. The threat model throughout: an attacker with read access to stored data, via database dump, stolen backup, misconfigured read replica, compromised analytics pipeline, or read compromise of a service account, but without access to key material, application logic, or live query traffic. Models with key compromise, API-only access, or live query observation produce different leakage profiles and are out of scope. The SSE literature distinguishes these as snapshot versus persistent attacker models; see [3] for the leakage-category implications. ↩
2. The post's query demonstrates the leakage-abuse attack pattern formalized in Cash, Grubbs, Perry, and Ristenpart (2015), exploiting the leakage profile of searchable encryption schemes to recover queries and plaintext without breaking the underlying cryptography. Earlier work by Islam, Kuzu, and Kantarcioglu (2012) established access-pattern disclosure as the foundational attack vector. The token-frequency variant is particularly applicable to static database dumps, where access patterns aren't available but token-equality and frequency distributions are. ↩
3. Searchable encryption leakage depends on the construction and attacker model; see Cash et al. (2015) for the standard taxonomy. A one-time dump exposes index structure, token equality, and document frequency; long-lived access additionally reveals query equality, access patterns, and response volume. Those are different leakage categories and should be treated as such. ↩ ↩[2]
4. HMAC-derived tokens only help if (a) secrets remain secret and (b) tokens are separated by purpose and scope. Low-entropy or attacker-enumerable inputs can make blind tokens guessable. ↩