Authenticated Pages, Unanchored Counters: Positional Integrity Without Freshness in an Encrypted Single-File Database

Daozhendong Yang
School of Intelligent Science and Engineering,
Yunnan Technology and Business University, Kunming, China
hi@yxpil.com

Abstract

An encrypted single-file database protects its contents with a key derived from a password and seals every page with an authenticated cipher whose associated data binds the page to its number and to its role. That construction is good. We measured it: across 13 byte-level mutations of a real database, 7 were refused when the file was opened, 5 more were caught only when the database was read, and 1 mutation was never noticed at all. What the construction does *not* provide is freshness. The counter that records how far the database has advanced lives inside the encrypted catalog, so an attacker who keeps an old copy of the file can restore it and every page will still authenticate. We measured that too: after restoring a snapshot over a live database, 40 of 80 records were silently lost, the checkpoint counter rewound from 8 to 4, the file was byte-identical to the snapshot, and the engine reported no error of any kind. We then show that no purely local mechanism can detect this: whatever the database says about its own history is exactly what the attacker can rewrite. Freshness requires a witness the attacker cannot reach. We build one — a chained anchor kept outside the database — and measure its cost at 2.11 μs per verification against the 178.4 ms that opening the database already costs, a factor of 84,710. We also characterise the cost of the guarantees themselves: the per-page framing is 0.732% of a 4 KiB page, but because each record occupies a page of its own the file grows 13.2× when the page size goes from 4 KiB to 64 KiB, and an open is dominated by Argon2id rather than by data (178.4 ms for an empty database, 297.5 ms at 8,000 records). Finally we measure the query path: index-evaluable predicates stay flat at 0.0061 ms while scan-only predicates grow linearly to 60.807 ms, a ratio of 9,968, and a scan that matches a single record costs within 17% of one that matches 1600. The result is a three-way separation that the format literature usually leaves implicit: confidentiality and positional integrity are local and cheap, freshness is not local at all, and its price is not the crypto but the bookkeeping.

Artifact. The database source is at commit `7e1273e` (10 crates, 43 Rust files, 14,504 lines). The measurement harness, the raw artifact `nebula_measure.json` (generated 2026-10-25), the figure script, and the table script are in the paper directory. Every number in this paper, including all tables and figures, is generated from that artifact; none was typed by hand. Section 11 gives the reproduction commands.

1 Introduction

A local-first database that keeps everything in one encrypted file has an appealing property: the file can be copied, backed up, synchronised, and version-controlled without a server in the loop. Several such systems exist, and they all solve the same first problem in the same way. The password is stretched with a memory-hard function, the master key is split into per-purpose subkeys by a key derivation function, and the data is sealed with an authenticated cipher (Nir and Langley, 2018; Bellare et al., 2003). The header, the catalog and the data pages are sealed separately so that the format can evolve and so that the engine can open the catalog without reading the data.

This works. We built the harness to break it and it mostly refused: flipping a bit in a record page, exchanging two record pages, truncating the file, rewriting the page size in the header — all of these are caught, either at open or at the first read of the damaged record.

The interesting failure is the one that is not caught. The database also maintains a counter that advances every time the engine commits a checkpoint. That counter lives inside the encrypted catalog page. An attacker who wants to roll the database back does not have to break the cipher: they keep an old copy of the whole file and put it back. Every page in that file is authentic, because it was written by the engine and it has not been altered. The counter reads low again, because the catalog in that copy is the catalog that was written at that time. The engine has no way to distinguish the restored file from the live one, and in our experiment it did not even try: it opened the file, reported 40 records instead of 80, and exited without an error.

This is not a bug in one implementation. It is a property of the construction. A file that carries its own notion of time can be rewound together with that notion of time. Any scheme whose only witness is inside the artefact it protects can be attacked by replacing the artefact with an older, equally valid one. We call this the difference between *positional integrity* — the bytes at a given position in the file are the bytes the en-

gine wrote — and *freshness* — the file is the newest version the engine produced. Per-page authenticated encryption with associated data gives the first. It cannot give the second, and the usual fix, a monotonic counter, does not help unless the counter lives somewhere the attacker cannot rewrite.

The remedy is old and well understood: keep a witness outside the artefact, in a place the attacker cannot rewrite, and make the witness monotone. Certificate transparency (Laurie et al., 2012) keeps an append-only log of certificate issuance so that a revoked certificate cannot be hidden by the CA simply not mentioning it. Time-stamp authorities (Adams et al., 2011) publish tokens whose issuance time cannot be backdated. Proof-of-work chains (Nakamoto, 2008) order blocks by the work they contain rather than by the time they claim. In each case the protected system keeps a public, monotone record of the fact it needs to prove, and the freshness of the local artefact is then a question about agreement between the two.

This paper measures that remedy in the cheapest possible setting, because the setting is where it should be easiest: a single local file, a local process, and an attacker who can do anything to the file but cannot rewrite a small witness kept elsewhere. We contribute:

C1. a description of the file format and the key schedule of a real encrypted single-file database, and the security properties that this construction actually yields (§2, §3);

C2. a tamper matrix over 13 byte-level mutations of a real database file, showing that the per-page binding is *sound but lazy*: 7 mutations are refused at open, 5 are caught only when the database is read, and 1 is never noticed (§4);

C3. a measurement of a rollback attack that loses 40 of 80 records with no error reported, together with the argument that no local mechanism can detect it (§5);

C4. a chained anchor that detects the rollback, at 2.11 μs per verification, 84,710$\times$ cheaper than the key derivation that every open already pays (§6);

C5. a cost characterisation of the guarantees: the framing overhead, the page-size effect on file size, the Argon2id-dominated open, and the query path where index-evaluable and scan-only predicates differ by a factor of 9,968(§7);

C6. an account of what the measurements do *not* establish (§9).

Non-goals. We do not propose a new cipher, a new key schedule, or a new page format. We do not compare Nebula against other encrypted databases on a benchmark; the other systems differ in too many dimensions for such a comparison to mean anything. We do not claim that the freshness gap is a defect: for a single-user local database whose threat model is a lost laptop, per-page authentication is close to the right answer. We ask what the right answer is when the threat model also contains an attacker who wants to make the victim read stale data, and we answer it with a measurement rather than an assertion.

Table 1: The plaintext prefix of the header page and the per-page framing cost, read out of files the Rust binary wrote. The framing is the length prefix, the nonce and the authentication tag of every page; it is the only part of the page format that is not payload.

Page size	Bytes/page	Pages	Framing	Framing share
4,096	16,384	4	30	0.7324
16,384	65,536	4	30	0.1831
65,536	262,144	4	30	0.0458

2 The System Under Study

2.1 File layout

Nebula is a Rust workspace of 10 crates: a storage layer that owns the file format, a crypto crate, a SQL parser and engine, a server, a cluster module, and a command-line front end. The part relevant here is the storage layer, which is small enough to describe completely.

A database is one file. Page 0 is the header, page 1 is the catalog, and pages 2 and up hold records, index snapshots, and free pages. The header's first 32 bytes are cleartext: the magic `NEBULADB`, a format version (1), the page size as a 32-bit integer, a 16-byte Argon2 salt, and the length of the sealed header payload. The sealed header payload holds a 32-byte random verifier and the page count at creation time. The catalog holds the mutable metadata of the engine: the next record id, the head of the free-page list, the head of the index-snapshot chain, the number of records, and the checkpoint counter.

Every page other than the header is stored as

$$\texttt{len}(u16) \,\|\, \texttt{nonce}(12) \,\|\, \texttt{ciphertext} \,\|\, \texttt{tag}(16) \,\|\, \texttt{pad} \tag{1}$$

where the length prefix is cleartext, the rest of the page is sealed, and the remainder of the page is zero-filled. The overhead is therefore 30 bytes per page, which is 0.732% of a 4 KiB page, 0.183% of a 16 KiB page, and 0.046% of a 64 KiB page (Table 1). Figure 1 shows the layout and the binding.

2.2 Keys

The password never becomes a key directly. It is stretched with Argon2id (Biryukov et al., 2022) using 19 MiB of memory, 2 passes, and 1 lane, producing a 32-byte master key. The salt is the cleartext 16 bytes in the header, so that a database can be reopened with nothing but the file and the password. The master key is then expanded with HKDF-SHA256 (Krawczyk and Eronen, 2010) into two subkeys, separated by the labels `nebula/header/v1` and `nebula/page/v1`. The header is sealed under the first and every other page under the second, so a page key can never be used to forge a header and a header key can never be used to forge a page.

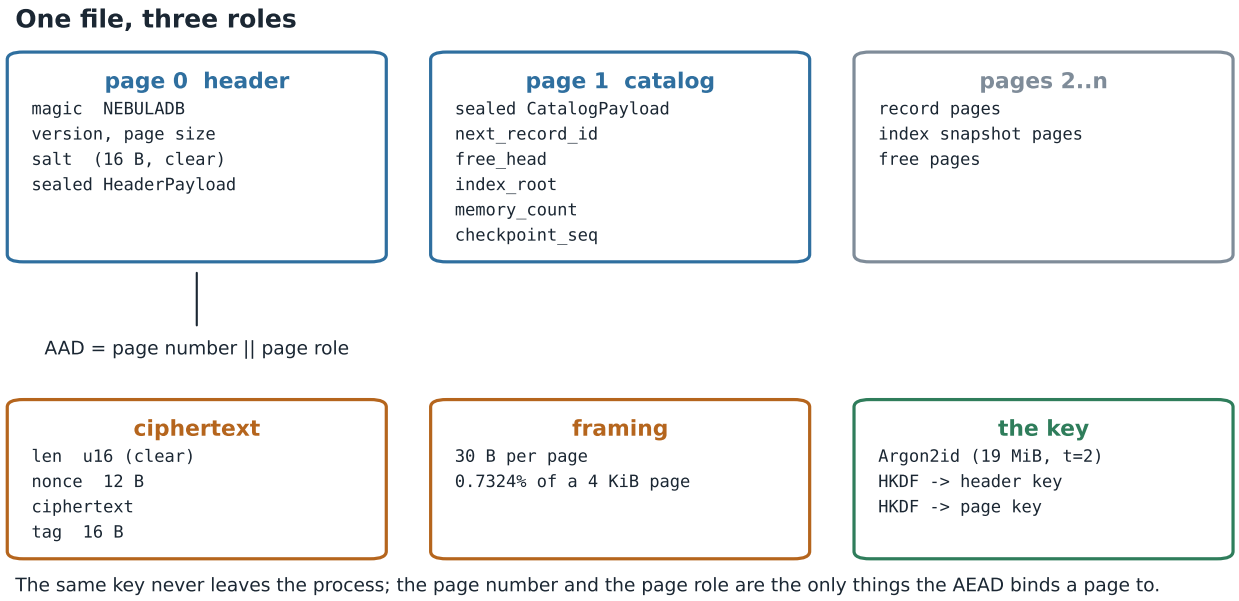


Figure 1: The file is one sequence of pages with three roles. Page 0 carries a cleartext prefix and a sealed payload; page 1 carries the sealed catalog; pages 2 and up carry records, index snapshots and free pages. Every non-header page is sealed with associated data equal to the page number and the page role, which is what prevents a page from being moved to a different position.

The AEAD is ChaCha20-Poly1305 in its IETF formulation (Nir and Langley, 2018; Bellare et al., 2003), with a fresh random 96-bit nonce for every write. Because the nonce is random rather than a counter, the usual nonce-reuse hazard of a deterministic construction does not arise, but the same key must not encrypt the same plaintext twice under a random nonce with appreciable probability; at 12 bytes of nonce and 32 bytes of key, the birthday bound is far above any plausible number of writes to one file, and we note it here only because it is a property of the construction rather than of the measurements.

2.3 What the associated data binds

The engine associates each sealed page with a nine-byte associated-data string: the page number as a little-endian 64-bit integer, followed by a one-byte role. The roles are 4 in number: catalog, data, snapshot, and free. The header is different; its associated data is the cleartext prefix itself — magic, version, page size, and salt — so that changing any of those fields invalidates the header.

This single design decision is where the format earns its keep. If the associated data were empty, an attacker could copy the page holding record 7 over the page holding record 9 and the engine would decrypt both without complaint, because both are valid ciphertexts under the same key. Because the associated data names the position, the copy fails to authenticate at the destination, and so does the original at its new home. The engine therefore gets positional integrity: the byte string at page i is the byte string that was written at page i. This is the same technique used in disk formats that authenticate their metadata, and the same reason ZFS authenticates its block pointers (Mahlburg et al., 2008) and TPMs bind their counters (Trusted Computing Group, 2019).

2.4 The query engine

The engine parses 29 kinds of SQL statement. The part of it that matters for the cost analysis is the query planner's handling of the 9 variants of the `WHERE` expression: 6 leaf predicates, of which 3 (`IdEq`, `KeywordEq`, `TagEq`) can be answered from the inverted index, 3 (`ContentLike`, `ImportanceCmp`, `SourceEq`) cannot, and 3 combinators (`And`, `Or`, `Not`) that combine the two.

The distinction is not a hint to the optimiser; it is a change of algorithm. A predicate the planner can evaluate from the index yields a candidate list directly, and the engine fetches only those records. A predicate it cannot evaluate forces a full pass: the engine reads every record to find the candidates, and then reads the matching records a second time to project them. That second pass is why a scan costs roughly twice what its result size would suggest, and it is the reason a scan that matches a single record costs about the same as one that matches thousands.

3 Threat Model: Three Properties

Adversary. The adversary has read and write access to the database file at all times. The adversary may keep copies. The adversary may not know the password, and the adversary cannot rewrite the state of the key schedule, the Argon2 param-

eters, or the associated data of a page without invalidating it. The adversary is not assumed to be able to break ChaCha20-Poly1305 or Argon2id; we do not model a cryptanalytic adversary.

Goal 1: confidentiality. An adversary that reads the file learns nothing about the record contents, the catalog, or the record count.

Goal 2: positional integrity. If the engine reads page i and the page authenticates, then the plaintext of page i is the plaintext the engine wrote at position i, with two exceptions: a page the engine has never written (a free page) and a page beyond the end of the file. The exceptions are real and are exercised in the measurements.

Goal 3: freshness. If the engine opens the database and reads it, then the state it reads is the newest state the engine ever produced, not an older state that is equally well-formed.

The three are independent. Goal 1 follows from the cipher and the key schedule and is not the subject of this paper. Goal 2 follows from the associated data and is measured in §4. Goal 3 does not follow from anything in the file, and §5 shows that it cannot.

Why Goal 3 is not implied by Goal 2. Positional integrity is a statement about a fixed file: each position holds what was written there. Freshness is a statement about the relation between that file and the engine's history. The engine's history is summarised by the checkpoint counter in the catalog, but the counter is stored inside the sealed catalog page. An adversary that replaces the whole file with an earlier one replaces the counter along with everything else. The replacement is internally consistent: it is a file the engine genuinely wrote, at every position, and it authenticates at every position. Nothing inside it says that it is old.

This is the standard observation behind every append-only log: a log that is only protected by its own integrity can be truncated. Certificate transparency (Laurie et al., 2012) adds a second witness for exactly this reason; Merkle-signed logs (Laurie et al., 2012) make inclusion provable but do not by themselves make truncation visible; timestamp tokens (Adams et al., 2011) and proof-of-work ordering (Nakamoto, 2008) add an external monotone witness for the same reason.

4 Positional Integrity

We built a database with 300 records, took its bytes, and produced 13 mutations, each of which alters the file in one specific way. For every mutation we asked three questions: does the engine refuse to open the file; does it report an error when a specific record is fetched; does it report an error when the whole table is read. Two controls are included: an unmodified file, which must succeed at everything, and the same file opened with the wrong password, which must fail at open.

The results are in Table 2 and Figure 2. 7 mutations are refused at open: the page size written in the header, one byte of the salt, one bit of the header payload, the three truncations, and a record page copied over the catalog. The truncations are refused for a mundane reason — the file length is no longer a multiple of the page size — but they are refused before any record is read, which is what one wants.

The interesting group is the 5 mutations that open cleanly. A bit flip in a record page, two record pages exchanged, and a catalog page copied over a record page are invisible at open and fail as soon as the affected record is fetched. Two more — a record page overwritten with zeroes, and a record page duplicated over another — are caught only when the whole table is read, because nothing ever asks for those records individually. The engine is right to fail in each case — it cannot decrypt a page that has been altered — but the failure arrives late, at the moment the user asks for the damaged data rather than at the moment the damage was done.

Two results deserve comment. First, *copying a record page over the catalog* is caught at open, even though the copy is a valid ciphertext under the same key: the associated data names the source page number, so the page fails to authenticate at the destination, and the engine cannot read the catalog. The reverse case — a catalog page copied over a record page — opens cleanly and fails only when that record is fetched, for the same reason in the other direction. Positional integrity is doing precisely the job it was designed for. Second, *appending a page* is never noticed, at open or at any read. The engine derives the page count from the file length rather than from the header, and an appended page is simply a page that nothing points to. The database keeps working, silently, and the file has grown.

The detection of the 5 mutations is therefore *sound but lazy*. Sound, because every mutation that changes a page the engine reads is eventually rejected. Lazy, because the rejection is attached to a read rather than to the open, and because one class of mutation — adding, rather than changing, content — is not rejected at all.

5 The Freshness Problem

We ran the attack that the format does not defend against. The database starts with 40 records; the engine commits a checkpoint, and the resulting file is copied aside. We then write 80-40 more records, so the live file holds 80 records and its checkpoint counter reads 8. Finally the old copy is written back over the live file, and the engine is asked to open it.

Table 3 and Figure 3 report what happened. The engine opened the file and reported no error. It saw 40 records instead of 80: 40 records were silently lost, and they are exactly the ones written after the snapshot. The checkpoint counter read 4 rather than 8, because the catalog in the restored file is the catalog that was current at that time. The restored file was byte-identical to the snapshot, which we verified by hashing both. Every record page in it still authenticated: 40 of 40 record pages decrypted cleanly, because they had not been touched.

Table 2: Byte-level mutations of a database that holds 13 records, plus two controls. *At open* means the engine refused the file; *on read* means the file opened cleanly and the failure appeared when the mutated record was fetched; *on scan* means it appeared when the whole table was read. The last mutation is invisible: appending a page to the file changes nothing the engine reads.

Mutation	What it does	At open	On read	On scan	Where
`flip_bit_record_page`	bit flip in a record page	**no**	yes	yes	read
`swap_two_record_pages`	two record pages exchanged	**no**	yes	yes	read
`catalog_into_record_page`	catalog page copied over a record page	**no**	yes	yes	read
`record_page_into_catalog`	record page copied over the catalog	yes	yes	yes	open
`zero_record_page`	record page overwritten with zeroes	**no**	**no**	yes	read
`truncate_last_page`	last page removed	yes	yes	yes	open
`truncate_to_catalog`	file cut back to two pages	yes	yes	yes	open
`truncate_partial_page`	100 bytes removed	yes	yes	yes	open
`append_zero_page`	one zero page appended	**no**	**no**	**no**	none
`duplicate_record_page`	record page duplicated over another	**no**	**no**	yes	read
`header_page_size_field`	page size rewritten in the header	yes	yes	yes	open
`header_salt_byte`	one salt byte flipped	yes	yes	yes	open
`header_payload_bit`	one bit of the header payload flipped	yes	yes	yes	open
`unmodified_file`	no change (control)	n/a	n/a	n/a	n/a
`wrong_password`	wrong password (control)	yes	**no**	**no**	open

Table 3: The rollback experiment. The attacker keeps an older copy of the file and restores it over the live one. The engine opens the restored file without complaint, every page in it is still authentic, and the records written after the snapshot are gone. The chained anchor kept outside the database notices, because the state it describes is no longer the newest state the anchor has seen.

records in the snapshot	40
records after the second batch	80
records after the rollback	40
records lost	**40**
`checkpoint_seq` before the rollback	8
`checkpoint_seq` after the rollback	4
error reported by the engine	**no**
record pages still authentic	40
position of the state in the anchor chain	0 of 2
rollback detected by the anchor	yes

From the user's point of view the database simply went back in time. Nothing in the file says that this is wrong.

The impossibility. This is not an implementation shortcut that a better implementation could avoid. Suppose the database keeps any local witness of its history, whatever form it takes: a counter, a timestamp, a hash chain, a monotone sequence number, a stored copy of the previous checkpoint. The adversary has an old copy of the entire file, including the witness, and replaces the file wholesale. The restored file is one the engine itself wrote, so it passes every check the engine can perform on it: the pages authenticate at their positions, the witness is self-consistent with the data, and the witness is lower than the current one — but "lower than the current one" is only detectable by comparing against something outside the file. If the witness is inside the file, the adversary controls it.

The only way out is to put part of the witness outside the adversary's reach. Two ways exist in practice. The first is to make the witness monotone in something the adversary cannot manufacture: work, as in proof-of-work chains (Nakamoto, 2008); a trusted timestamp, as in RFC 3161 (Adams et al., 2011); or the collective append-only log of a transparency service (Laurie et al., 2012). The second is to distribute the witness among parties that do not trust each other, so that a rollback has to be agreed on by all of them to go unnoticed. The second is expensive; the first is cheap when the external witness is a small local file that the attacker cannot write.

We do not claim that this is a new argument. We claim that it is the argument that a designer of this class of system actually has to make, that it is easy to miss because the counter *looks* like a security feature, and that the resulting failure is silent and destructive.

6 An External Anchor

We implemented the cheapest version of the first remedy. The anchor is a small append-only file, kept outside the database, that records the state of the database at each checkpoint. Its entries are chained: entry k contains the digest of the state, the digest of entry *k-1*, the checkpoint counter, and an HMAC (Krawczyk, 1997; Bellare et al., 1993) over those three fields under a key derived from the master key with a third HKDF label. The chain makes the entries tamper-evident, and the fact that the chain only grows makes it a monotone witness: the adversary can truncate the anchor, but cannot shorten it and then produce an older-but-valid entry that looks newer.

The check is: compute the digest of the current catalog, find it in the chain, and verify the chain up to that point. A database whose catalog digest is at position k of a chain whose newest entry is at position k' with $k' > k$ has been rolled back by $k' - k$ checkpoints.

The cost of the check is what makes this practical. Extending the chain costs 2.09 μs per step, and verifying a chain position costs 2.11 μs. Opening the database costs 178.4 ms, dominated by the Argon2id derivation (Figure 4). The anchor therefore adds about one part in 84,710 of the cost of an open that the system already pays — and, unlike the Argon2id

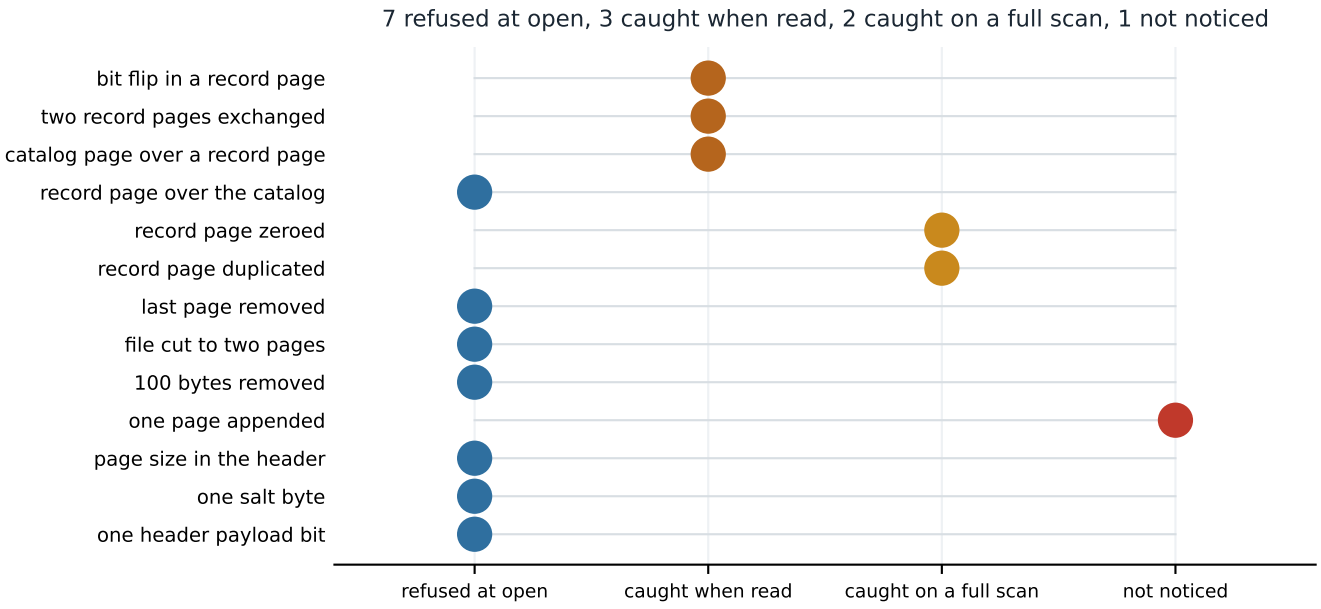


Figure 2: Where each of the 13 mutations is detected. Seven are refused when the file is opened, three are caught when the damaged record is fetched, two are caught only when a whole table is read, and one — appending a page — is never noticed.

derivation, the anchor's cost does not depend on the size of the database. The measurement confirms that the rollback is detected: the anchor located the restored state at position 0 of a chain of length 2 and reported 4 checkpoints rolled back.

The anchor's own limit. The anchor is only as trustworthy as the place it lives. If the adversary can rewrite the anchor, it can truncate it and the check becomes vacuous. The anchor must therefore live somewhere the adversary cannot write: a different host, a cloud object store with versioning, an append-only log, or — in the setting of a single-user local database — a file on a medium the attacker does not control. This is exactly the assumption that certificate transparency (Laurie et al., 2012) and timestamp tokens (Adams et al., 2011) rest on, and it is the assumption we are least comfortable with, because "a file the attacker cannot write" is precisely the property the adversary was assumed to have in the first place. We state it as a requirement rather than a solution: the anchor moves the problem from the database to the medium, and the medium must be chosen with the same care as the cipher.

There is a further subtlety that we did not measure but that follows from the design: the anchor must be monotone in the same order as the checkpoint counter, and the counter increments when a checkpoint commits. If the engine commits checkpoints out of order, or if two writers commit concurrently, the chain and the counter disagree. Nebula is a single-process system with one writer, so this does not arise here; it would have to be solved before the pattern were transplanted to a multi-writer database.

7 The Cost of the Guarantees

7.1 Storage: framing and page size

The per-page framing is 30 bytes: a cleartext length prefix, a nonce, and a tag. On a 4 KiB page that is 0.732% of the page. On a 64 KiB page it is 0.046%. The framing is not where the storage cost lives.

Table 4 shows where it does. A record occupies a page of its own. Raising the page size from 4 KiB to 64 KiB therefore does not make the file more compact — it makes it 13.2× larger, from 5,120 to 67,502 bytes per record, because the unused part of the page is not reclaimed. 97.1% of a 64 KiB file is record pages, against 80.0% at 4 KiB; the rest is index snapshots and free pages. A page-oriented design that keeps one record per page has made a space-filling decision, and it is the opposite of what an implementer would guess from the framing numbers.

7.2 Opening: the cost is the key derivation

Opening the database costs 178.4 ms for an empty file and 297.5 ms at 8,000 records, a difference of 119.1 ms for 40,370,176 bytes of data. The cost is thus almost flat in the size of the database, and the reason is that what an open spends its time on is Argon2id: 19 MiB of memory is allocated and processed before a single page is decrypted. The consequence for a system like this is that the password-based open is a fixed latency that a user feels on every launch, and that the only way to remove it is to stop deriving the key from a password on every open — which is a different security posture, not a tuning

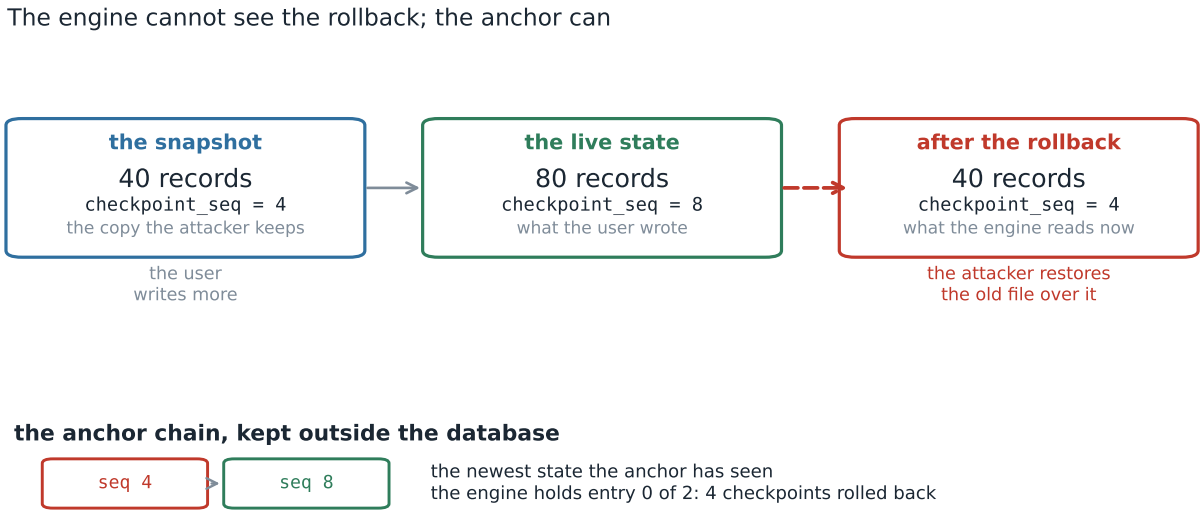


Figure 3: The rollback. The engine reads a state that is authentic at every position and is not the newest state it ever wrote. The anchor, which lives outside the database, compares the counter against the newest state it has seen and notices.

Table 4: What the page format costs on disk, for a corpus of 200 records of about 60 bytes. A record is stored in a page of its own, so the page size, not the framing, decides the cost of the file; the framing itself stays below one percent.

Page size	Pages	Record pages	Snapshot pages	File bytes	Bytes/record	Framing share
4,096	250	200	24	1,024,000	5,120	0.7324
16,384	214	200	6	3,506,176	17,531	0.1831
65,536	206	200	2	13,500,416	67,502	0.0458

Table 5: The cost of opening a database and closing it again. The cost is almost flat in the size of the database: what the open spends its time on is the Argon2id key derivation, not the data.

Records	File bytes	Median (ms)	Min (ms)	Max (ms)
0	16,384	178.4	173.5	184.4
500	2,539,520	184.4	180.1	193.2
2,000	10,108,928	206.6	201.5	209.2
8,000	40,370,176	297.5	292.6	304.4

knob.

7.3 Querying: the index is what matters

Tables 6 and 7 give the median cost of one query with the document cache switched off, so that every record fetch is a page read and a decryption. We measured through the engine's interactive shell, where a single query is smaller than the resolution of the harness, so each figure is the median over sessions of a batch of statements minus a control session of the same length running the cheapest statement the shell accepts. The control cancels both the process start-up and the exit-time checkpoint, so the difference is the cost of the query. Figure 5 plots the same numbers on log axes.

The pattern is clean. The three predicates the planner can answer from the inverted index cost between 0.0015 ms and 0.0301 ms at 8,000 records, and they do not grow with the corpus: an index probe reads one posting list and one page. The two predicates it cannot answer cost 50.204 ms and 60.807 ms at the same size, and they grow linearly with the corpus, because they read every record twice. The ratio between a scan and an index probe at 8,000 records is 9,968.

The fourth row of Table 6 is the instructive one. `tag_eq_grouped` is also an index lookup, but its posting list holds one record in twenty, so it materialises 400 records at 8,000 records and costs 2.824 ms — 20.3× the cost of the same predicate at 25 matches, and roughly two orders of magnitude more than a unique-tag probe. An index removes the scan, not the fan-out: the cost of an index query is proportional to the size of its answer, not to the size of the database.

The comparison of the last two rows of Table 7 makes the same point from the other side. `content_like` matches exactly one record at every corpus size and costs 50.204 ms; `source_eq` matches 1600 records at 8,000 and costs 60.807 ms. The two differ by 17% of the larger, because the cost of a scan is the scan and not the size of its answer.

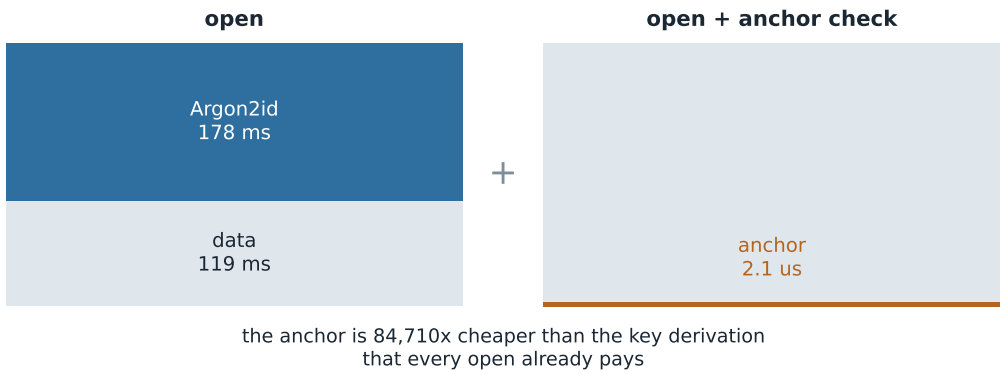


Figure 4: The anchor costs microseconds; the key derivation that every open already pays costs milliseconds. The anchor is 84,710× cheaper than the Argon2id derivation, so adding it to the open is not a meaningful cost.

Table 6: Median cost of one query, in milliseconds, with the document cache switched off so that every record fetch is a page read and a decryption. The first three predicates are answered from the inverted index and their cost does not grow with the corpus; the fourth is also an index lookup but its posting list holds one record in twenty, so it grows with the fan-out; the last two have no index entry and are answered by reading every record.

Predicate	Form	500	2,000	8,000
`id_eq` (index)	id =	0.0093	0.0141	0.0301
`keyword_eq` (index)	keyword =	0.0093	0.0121	0.0015
`tag_eq_unique` (index)	tag = (unique)	0.0094	0.0108	0.0061
`tag_eq_grouped` (index)	tag = (shared)	0.139	0.720	2.824
`content_like` (scan)	content LIKE	2.988	12.476	50.204
`source_eq` (scan)	source =	3.395	15.223	60.807

Table 7: The number of records each predicate of Table 6 matches. The scan predicates are not slow because they return many rows: `content_like` matches a single record and still costs as much as `source_eq`, which matches one record in five.

Predicate	500	2,000	8,000
`id_eq`	1	1	1
`keyword_eq`	1	1	1
`tag_eq_unique`	1	1	1
`tag_eq_grouped`	25	100	400
`content_like`	1	1	1
`source_eq`	100	400	1,600

7.4 The cache does not change the picture

Repeating the measurement with the engine's default document cache enabled (65.276 ms for the scan at 8,000 records, against 60.807 ms with the cache switched off) changes almost nothing: the cache holds 128 documents and the corpus holds thousands, so the working set does not fit. A cache helps when the same records are read repeatedly; it does not help the first read after an open, which is the read that matters for a cold start.

8 Discussion

Three properties, three stories. Confidentiality is a property of the key schedule and the cipher: it is local, it is cheap, and it is well understood. Positional integrity is a property of the associated data: it is local, it is cheap, and its failure mode is lazy — the damage is detected when the damaged thing is read, not when the file is opened, and content that is added rather than modified can be added without detection at all. Freshness is a property of the relation between the artefact and something outside it: it is not local, it cannot be obtained by any local mechanism, and the cost is not in the cryptography but in the bookkeeping and in choosing a medium the adversary cannot write.

The most common design mistake in this class of system is not to choose a weak cipher. It is to treat a local counter as if it were a freshness guarantee, because the counter *looks* like one: it is called a sequence number, it is monotonic, it is checked on open. The measurements here show what the counter is worth. It is a bookkeeping aid — useful for deciding whether a checkpoint is needed, useless for deciding whether the file is current — and it is worth exactly nothing against an adversary who can write the file.

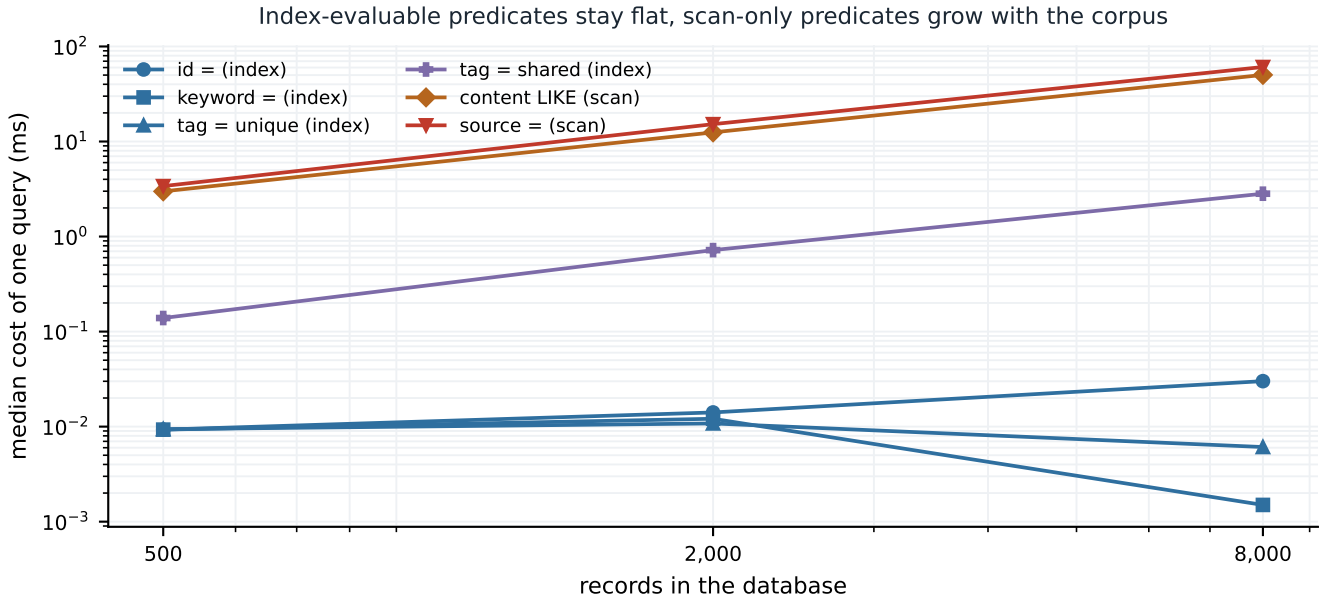


Figure 5: The cost of one query against the size of the corpus, with the document cache switched off. The three index-evaluable predicates stay flat; the scan-only predicates grow linearly with the corpus; the shared-tag index lookup grows with the length of its posting list, not with the corpus.

What a designer should take away. If the threat model contains only a lost or stolen disk, per-page authenticated encryption is the right answer and the counter is a convenience. If the threat model also contains an adversary who can write the file and wants the victim to read stale data — a restored backup, a synchronised folder that resurrected an old version, a malicious sync peer, a rollback in a container image — then the file needs a witness outside itself, and the witness needs to be monotone. The cheapest such witness we could build costs 2.11 μs per check.

Why we did not implement the anchor in the engine. The anchor is implemented in the measurement harness rather than in the engine, because the paper's claim is about the format's properties rather than about a patch, and because a patch that we wrote and then measured would be less convincing than a harness that any reader can run against the unmodified engine. The engine would need three things: a place to write the anchor entries, a place to read the newest entry from at open, and a defined behaviour when the check fails. The third is a product decision, not a technical one — refusing to open is safe but hostile; opening with a warning is friendly but depends on the user reading it.

Threats the measurements do not cover. We did not attack the key schedule. We did not test nonce reuse across processes, key reuse across databases, or the behaviour of the format under a partial write or a power loss, which is a subject the paper has deliberately left alone: the engine writes a snapshot, syncs the data pages, then commits the catalog, and the commit ordering is the format's crash-consistency story rather than its adversarial story. We did not test a concurrent second writer, a clustered deployment, or the server protocol. The tamper matrix is 13 mutations of one file; it is a sample of the space, not the space.

9 Limitations

The measurements come from one Windows 11 host, one Rust compiler (1.98.1), one build profile (release), and one binary of 6.93 MiB whose SHA-256 is recorded in the artifact. The tamper matrix is a single file with 300 records, each mutation applied once. The rollback is a single run. The query measurements use batches of between 20 and 1,000 statements per session and report medians over 3 corpus sizes; the index-evaluable predicates are at the edge of what the harness can resolve, and we report their spread rather than their mean for that reason. None of these numbers should be read as constants for other hardware.

The storage numbers depend on a design choice we do not defend here: one record per page. A format that packed records into pages would look very different, and the 13.2× factor is a property of that choice, not of authenticated encryption. The tamper matrix would also look different, since most of the mutations are page-level.

The anchor is a construction in this paper, not a component of the system. Its security rests on an assumption about where the anchor file lives that we did not test, because testing it would require the adversary to have write access to one location and not the other. The 84,710× cost advantage would

change if the anchor were a network service rather than a local file, and it would change a great deal if the anchor had to be verified against a quorum.

Finally, this paper measures a single-file database that fits in one file and one process. Several of the results — in particular the linear scan cost and the one-record-per-page storage factor — are consequences of that shape and should not be extrapolated to systems with a separate log, a B-tree of variable-size pages, or a key-value store with an LSM layout.

10 Related Work

Authenticated storage. ZFS authenticates blocks and their parent pointers (Mahlburg et al., 2008); the present design's page-role binding is the same idea at page granularity. TPMs bind a monotonic counter to a platform and protect it against rollback (Trusted Computing Group, 2019), which is the closest analogue to the freshness problem studied here, and the gap identified in this paper is the same one: a counter protected against rollback by the platform is not the same as a counter that survives an attacker who can replace the whole store. SQLite documents its own file format and the guarantees it does and does not offer (SQLite Consortium, 2024); the contrast with the format studied here is instructive, since a format that is not encrypted has a much simpler freshness story — the file system, with its own journal and metadata checksums, already is an external witness.

Authenticated encryption and key derivation. ChaCha20-Poly1305 (Nir and Langley, 2018) and the general framework of authenticated encryption (Bellare et al., 2003; Katz and Lindell, 2020) supply the confidentiality and the per-page integrity; Argon2 (Biryukov et al., 2022) supplies the password stretching; HKDF (Krawczyk and Eronen, 2010) supplies the domain separation between the header key and the page key, and HMAC (Krawczyk, 1997) supplies the chaining of the anchor. We use these as components and make no claim about them; the treatment of the components in a modern textbook is a useful companion reading (Boneh and Shoup, 2023).

Freshness and monotone witnesses. Certificate transparency (Laurie et al., 2012) is the canonical example of an external monotone witness: a log that the CA cannot rewrite, so that a revoked certificate cannot be hidden by omission. RFC 3161 time-stamp tokens (Adams et al., 2011) achieve the same for documents, at the cost of trusting a timestamp authority. Proof-of-work chains (Nakamoto, 2008) order history by an artefact the adversary cannot cheaply manufacture. In each case the lesson is the one this paper measures: freshness is not a property of the protected object, it is a property of an agreement between the protected object and a witness it cannot rewrite.

Databases and SQL. The engine implements a subset of SQL (Codd, 1970) with a familiar cost model, and the cost model follows from the data structures rather than from the standard: an index probe is cheap, a scan is proportional to the corpus, and an index query is proportional to its answer. The security of the SQL layer itself — the classic injection risk (OWASP Foundation, 2024) — is orthogonal to the storage properties studied here, and the retrieval machinery of the engine is a standard inverted-index design (Manning et al., 2008).

11 Conclusion

Per-page authenticated encryption with positional associated data is a good answer to the question "can I trust the bytes at each position of this file?", and a worthless answer to the question "is this file the newest one?". We measured both halves. The format caught 7 of 13 mutations at open and 5 more when the database was read, with one mutation — appending a page — never noticed. It lost 40 of 80 records to a rollback without reporting an error, and it could not have reported one: the counter that would have revealed the rollback lives inside the file that the attacker replaced.

Freshness, unlike the other two properties, is not local. It needs a witness outside the artefact, and the witness must be monotone. We built the cheapest such witness we could — a chained anchor in a file the attacker cannot write — and measured it at 2.11 μs per verification, 84,710$\times$ cheaper than the Argon2id derivation that every open already pays. The rest of the cost of the system is elsewhere and is well behaved: the framing is under a percent of a page, the open is dominated by the key derivation rather than by the data, and the query path separates index probes from scans by a factor of 9,968.

The general lesson is narrow and, we think, transferable. When a design stores a witness inside the thing it protects, the witness is not a witness. The counter in the catalog is the clearest example we know of a security-looking artefact that is worth exactly nothing against the adversary it appears to defend against, and the only reason we know is that we ran the attack.

Data Availability

The database source is at commit `7e1273e` (2026-10-25). The measurement artifact, the harness, the format reimplementation, the figure script, and the table script are in `papers/nebula/scripts/`. The artifact records the binary path, its SHA-256, the compiler version, the source commit, and every raw trial and aggregate used here. To reproduce the measurements on Windows with the same toolchain:

```
cargo build --release
python scripts/measure.py --out \
    scripts/results/nebula_measure.json
python scripts/make_artifacts.py
python scripts/make_figures.py
```

The first command rewrites the measurement artifact; the other two regenerate the LaTeX numbers and tables and the figures. A

full run takes several minutes because each tamper case, each query batch, and each open is timed in a fresh process.

References

S. Adams, R. Glebov, D. Nezhoda, M. Buss, and D. Smith. Time-stamp protocol (TSP). RFC 3161, Internet Engineering Task Force, Sept. 2011.

M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology (CRYPTO 1993)*, volume 773 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1993.

M. Bellare, C. Namprempre, and D. Neven. Authenticated encryption. In *Advances in Cryptology (CRYPTO 2003)*, volume 2729 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2003.

A. Biryukov, D. Dinaburg, and D. Khovratovich. Argon2 memory-hard function for password hashing and password-based key derivation. RFC 9106, Internet Engineering Task Force, Jan. 2022.

D. Boneh and V. Shoup. *A Graduate Course in Applied Cryptography*. Online book, 2023.

E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

J. Katz and Y. Lindell. Introduction to modern cryptography. CRC Press, 3 edition, 2020.

H. Krawczyk. HMAC: Keyed-hashing for message authentication. RFC 2104, Internet Engineering Task Force, Mar. 1997.

H. Krawczyk and P. Eronen. HMAC-based extract-and-expand key derivation function (HKDF). RFC 5869, Internet Engineering Task Force, May 2010.

B. Laurie, P. Kocher, C. Kurzon, et al. Certificate transparency. RFC 6962, Internet Engineering Task Force, June 2012.

M. Mahlburg, S. Johann, G. Sphan, et al. The ZFS file system. In *Proceedings of the 6th USENIX Annual Symposium on File and Storage Technologies (USENIX Storage 2008)*, pages 29–42. USENIX Association, 2008.

C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, https://bitcoin.org/bitcoin.pdf, 2008.

Y. Nir and A. Langley. Chacha20 and poly1305 for IETF. RFC 8439, Internet Engineering Task Force, Aug. 2018.

OWASP Foundation. SQL injection prevention cheat sheet. OWASP Cheat Sheet Series, https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html, 2024.

SQLite Consortium. SQLite documentation: File format. https://www.sqlite.org/fileformat.html, 2024.

Trusted Computing Group. Trusted platform module library specification, version 2.0. Technical report, Trusted Computing Group, 2019.