

VeilDB: A Zero-Trust, Local-First Encrypted Synchronization Engine

Elaiyaraja P¹, Eakanth G², Basavaraj³, Abhishek Tambole⁴, Manohar R Gouda⁵

^{1,2,3,4,5}Department of Computer Science and Engineering, Sir M Visvesvaraya Institute of Technology, Bangalore, India

Abstract - Cloud applications routinely ask people to trust that whoever operates the server will not look at the data it stores - a promise the user has no way to check. VeilDB removes the need for that trust. It is an embedded, local-first database in which each record is encrypted on the device that creates it, so the synchronization service only ever handles opaque blobs that it can neither decode, reorder, nor quietly drop without being caught. Every record is encrypted with AES-256-GCM under device-held keys, signed with Ed25519, and chained by BLAKE3 digests into a Merkle DAG that doubles as the structure used to reconcile replicas. Concurrent edits made on separate devices converge deterministically through per-field CRDT registers, so no coordinator is required and no write is lost to a last-writer-wins race. Above this core the prototype adds keyed equality indexes over ciphertext, periodic snapshots that cap replay cost, reads at any past version, capability-based sharing, and device revocation by key epoch. Built in roughly 11,200 lines of Rust over SQLite and measured on commodity hardware, VeilDB sustains 12,400 sealed writes and 47,300 reads per second at a 1 KiB value, a fixed 156-byte envelope per record, and reconciles two replicas differing by 1,000 records in six network rounds carrying 84 KiB. On this evidence, discarding the trusted server costs about 3.4 times the write throughput of unprotected SQLite - a price many privacy-sensitive applications can afford.

Key Words: local-first software, zero-trust architecture, conflict-free replicated data types, authenticated data structures, encrypted synchronization, embedded databases

1. INTRODUCTION

A note-taking tool, a password vault and a clinical data logger have little in common beyond the shape of the problem they solve. Each one stores records that belong to a single person, each must keep working when connectivity drops, and each is expected to look identical on a phone and on a laptop within seconds of an edit. The industry's answer to that problem has been remarkably uniform: keep the authoritative copy on a server, demote the device to a cache, and ask the user to accept that the operator can read everything. The arrangement is convenient and it scales. It also means that a single leaked credential, one misconfigured storage bucket, or one legal order reaches every record at once.

Local-first software [1] argues for the opposite allocation of authority. The device holds the primary copy, the network is merely an optimisation, and the user retains ownership even if the vendor disappears. Much of the supporting theory is already mature. Conflict-free replicated data types [2] provide merge rules that converge without coordination, and systems reaching back to Coda [3] and Bayou [4] established that disconnected operation is workable in practice. What remains thin is the overlap of that theory with a confidentiality guarantee that still holds when the server itself is the adversary.

Two families of systems approach this gap and then stop short. Cryptographic storage designs such as Plutus [5], SiRiUS [6] and SUNDR [7] protect content on an untrusted host, but they are file-oriented and assume a largely connected client; merging divergent replicas was never their concern. Encrypted-database work such as CryptDB [8] preserves rich query capability at the cost of leakage that later attacks learned to exploit [9], [10], and it still centralises. CRDT-based sync engines, meanwhile, typically terminate TLS at a server that then handles plaintext, which quietly returns the operator to a position of full trust by a different route.

VeilDB occupies that gap. One decision drives most of the rest of the design: the unit of replication is a sealed, signed, immutable record, and the relay is granted no capability beyond storing such records and returning them by identifier. Everything a server would ordinarily do - ordering, conflict resolution, access control and indexing - is pushed onto the client, where the keys live.

1.1 Motivation

Placing the authoritative copy on a server is cheap for the operator and expensive for the user's privacy. The costs are not hypothetical: breaches, insider access and compelled disclosure all act on data the operator is able to read. We set out to build a store that behaves like an ordinary embedded database from the application's point of view, yet leaves the operator with nothing to read and nothing it can tamper with undetected. The guiding question was whether the price of removing that trust is small enough for genuine privacy-sensitive applications to pay.

1.2 Contributions

This paper makes four contributions. First, a record envelope that is at once the unit of encryption, the unit of authentication and a node in a Merkle DAG, so that integrity checking and set reconciliation share one set of digests instead of maintaining two structures. Second, a synchronization protocol whose bandwidth scales with the number of records that differ between two replicas rather than with the size of the database - a practical treatment of the set reconciliation problem [25]. Third, a candid account of the leakage this design leaves behind, including access frequency, record counts and timing, together with the countermeasures we implemented and the ones we deliberately did not. Fourth, a Rust prototype and its measurement, including the places where the design cost more than we had expected.

2. RELATED WORK

2.1 Disconnected and Replicated Storage

Coda [3] demonstrated in the early 1990s that a file system could survive disconnection by logging changes locally and reintegrating them on reconnect. Conflict-free replicated data types [2] later replaced ad-hoc reconciliation with structures whose merge operation is commutative, associative and idempotent, so that convergence follows from the type rather than from a protocol; JSON variants [11] and Merkle Search Trees [12] carried the idea to documents and to open networks. This whole line of work assumes that the host storing a replica can read the data, because the merge function must. Encrypting against that host was never on the table.

2.2 Cryptographic Storage on Untrusted Hosts

Plutus [5] and SiRiUS [6] encrypt file contents and distribute keys through lockboxes, targeting untrusted file servers with minimal server modification. SUNDR [7] confronted a stronger adversary and contributed the notion of fork consistency: a malicious server may partition clients into divergent views, but it cannot keep the partition hidden once any two clients compare notes. We adopt that framing without alteration and, like SUNDR, detect forks rather than prevent them.

2.3 Query Processing over Ciphertext

Searchable symmetric encryption [13], [14] supports equality lookups over encrypted data through a keyed index, and CryptDB [8] layered property-preserving

encryption schemes to serve much of SQL against an untrusted database server. The leakage-abuse literature that followed [9], [10] showed that order-revealing and frequency-revealing layers can be inverted with modest auxiliary information, particularly over low-entropy columns.

2.4 Deployed Synchronization Engines

Replicated stores that ship in products - document databases with multi-master replication, content-addressed distribution systems such as Dat [15] - generally encrypt in transit and at rest under keys that the operator can produce. End-to-end encrypted messaging [16] is the notable exception, and it achieves strong confidentiality, but its data model is an append-only conversation rather than a mutable store that must answer queries and reconstruct past states.

3. THREAT MODEL AND DESIGN GOALS

We assume a relay that is fully malicious rather than merely curious. It may read every byte it stores, retain material indefinitely, reorder or duplicate what it returns, withhold records selectively, present different histories to different devices, and collude with any number of revoked devices. The two things it cannot do are produce a valid Ed25519 signature under an enrolled device key, and recover plaintext without a data key - because neither a signing key nor a data key ever leaves a client.

The trusted computing base is kept deliberately small: the device operating system while the vault is unlocked, the platform keystore that protects the sealed root key at rest, and the cryptographic primitives themselves. Traffic analysis and denial of service are out of scope; a relay can always refuse to answer, and no cryptographic construction repairs that. Against this adversary the design pursues five goals.

1) Confidentiality: the relay learns nothing about record contents, field names or index terms beyond padded length, count and timing.

2) Authenticity: every record is attributable to exactly one enrolled device, and any modification, truncation or reordering of a device's own history is detectable by that device.

3) Availability: reads and writes succeed with no network at all, and the database is fully functional on a single device that never synchronizes.

4) Convergence: two devices holding the same set of records compute identical application state, with no coordinator, no leader election and no lost update.

5) Revocability: a lost or stolen device can be excluded from all subsequent writes, and the cost of doing so is made explicit to the application rather than hidden behind a promise the design cannot keep.

4. SYSTEM DESIGN

The layers through which a write passes are described below in the order the write encounters them.

4.1 The Sealed Record

A write never modifies a row. It appends an immutable record describing an intended change to one key, and the current value of that key is whatever the merge layer computes from every record that mentions it. Each record is serialised into the fixed envelope shown in Table-1 before it ever touches storage.

Table-1: Layout of a sealed record envelope

Field	Size	Role
Vault id	16 B	canonical header (AES-GCM AAD)
Author	16 B	canonical header (AES-GCM AAD)
HLC	8 B	canonical header (AES-GCM AAD)
Epoch	4 B	canonical header (AES-GCM AAD)
Parents	32 x k B	canonical header (AES-GCM AAD)
Nonce	12 B	AEAD nonce
Ciphertext	n B	confidential payload
GCM tag	16 B	AEAD authentication tag
Signature	64 B	Ed25519 over BLAKE3(header nonce ct tag)

The first five fields form a canonical header. That header is passed to AES-256-GCM [20] as associated data, so the tag covers it, and an Ed25519 signature [19], [21] is computed over the BLAKE3 digest [22] of the header, nonce, ciphertext and tag together. The record identifier is the BLAKE3 digest of the entire envelope. Encrypting first and

signing afterwards is the deliberate choice here. It lets the relay, or any device that can authenticate but has not yet been granted a data key, verify that an envelope came from an enrolled device without being able to read it. The obvious hazard of that ordering is surreptitious forwarding, since a signature over ciphertext says nothing about who authored the plaintext; the author identifier is therefore carried a second time inside the encrypted payload, and the merge layer rejects any record whose inner author disagrees with its header.

Excluding parent digests, the envelope adds a constant 156 bytes to every record regardless of value size. That constant is the price of the design and it dominates for small values, a point Section 7 returns to with numbers.

4.2 Key Hierarchy and Enrollment

A vault has one 256-bit root key, generated on the device that creates it and never transmitted in the clear. At rest the root key is wrapped either by a key derived from the user passphrase with Argon2id [23] (64 MiB, three passes, one lane) or by the platform keystore where one is available. Data keys are derived per epoch as $DK(e) = HKDF-BLAKE3(RK, "veildb/data" || e)$, and each record is sealed under a further derivation $K(r) = HKDF-BLAKE3(DK(e), record_uuid)$. Deriving a fresh key for every record makes nonce reuse structurally impossible rather than merely unlikely; we still draw the nonce at random, on the reasoning that a bug in the derivation input should not immediately become a catastrophic failure of the AEAD.

Enrolling a second device is an X25519 key agreement bootstrapped by a six-word pairing phrase shown on the first device and typed into the second. The new device sends its Ed25519 public key, receives the current root key inside a sealed box, and the first device appends a signed enrollment record to the log so that every other replica learns which keys are now legitimate.

4.3 Hash Linking and the Merkle DAG

A record's identifier is the BLAKE3 digest of its complete envelope, so identifiers are content-addressed, stable and independent of arrival order. Each new record names the current set of heads as its parents. When one device writes while offline and another writes concurrently, the log has two heads, and the next record written after the two meet names both - which is what makes the structure a DAG rather than a chain. Altering any byte of any record changes its identifier and orphans every descendant, so tampering

becomes a structural inconsistency rather than something the verifier has to hunt for.

The same identifiers are also stacked into a Merkle tree [24] of fanout sixteen, keyed by the leading nibbles of the digest, with each internal node holding the BLAKE3 digest of its children. A tree over 2^{32} records is eight levels deep, and the root digest changes if and only if the set of records changes. This single structure serves two purposes that are usually served separately: it authenticates the database against tampering, and it is the index the sync protocol descends to find differences.

4.4 Keyed Equality Indexes

Scanning and decrypting an entire vault to answer a lookup is not acceptable, so designated fields carry an index tag: the first sixteen bytes of a keyed BLAKE3 digest over the normalised field value under a per-field index key derived from the root. Equality queries then become tag lookups in an ordinary SQLite index. The tags are deterministic, which is exactly what makes them useful and exactly what leaks - an observer who obtains the local database learns which records share a value, and over a low-entropy column such as a country or a status flag, frequency analysis recovers the mapping with little effort [10]. Bucketisation would blunt this, and we have not implemented it.

4.5 Merge Semantics

Timestamps come from a hybrid logical clock [17] packed into eight bytes as a 48-bit millisecond reading and a 16-bit counter, with the device identifier breaking exact ties.

1) Registers: a scalar field is a last-writer-wins register ordered by hybrid clock. The loser of a concurrent write is not discarded; it remains in the log and is reachable through the history API, which is the difference between resolving a conflict and hiding one.

2) Sets: membership uses an observed-remove set, so an element removed on one device and re-added on another survives, and a remove cancels only the specific additions it observed.

3) Counters: numeric aggregates use a positive-negative counter with one increment slot per device, which converges without any device seeing another's increments in order.

Because merge is a pure function of the record set and every ordering decision is carried inside the records

themselves, two replicas holding the same set compute the same state irrespective of the order in which they received it. That is the strong eventual consistency property of [2], and it is what allows the relay to be untrusted for ordering: reordering the delivery stream changes nothing, and dropping a record is detectable because its identifier is still referenced by its children.

Collaborative text is the obvious omission. A sequence CRDT with the properties we would want, such as the one described in [11], interacts awkwardly with per-record encryption, because each character-level operation would otherwise carry a full 156-byte envelope.

4.6 Snapshots and Historical Reads

Replaying a long log to answer a read becomes unacceptable past a few thousand records, so every S records written under a key space the writer emits a snapshot: the materialised state at that point, sealed as an ordinary record and marked with the causal frontier it summarises. A read at head loads the newest snapshot and replays at most S records forward. A read at an arbitrary past version finds the newest snapshot whose frontier precedes that version and replays from there, which is what gives the system time travel more or less for free once snapshots exist. Section 7 reports the trade-off that fixed the default at $S = 512$.

5. SYNCHRONIZATION OVER AN UNTRUSTED RELAY

5.1 Relay Interface

The relay exposes exactly three operations: store an envelope under its identifier, list identifiers beneath a given digest prefix, and fetch an envelope by identifier. It has no notion of users, keys, ordering or conflict.

5.2 Reconciliation

Two devices begin by exchanging the root digests of their Merkle prefix trees. Equal roots mean identical record sets, and the exchange ends in a single round trip. Otherwise each side sends the sixteen child digests at the current node, the two compare them, and both descend only into the children that differ. Subtrees that agree are eliminated wholesale at the cost of a 32-byte digest, so the volume of data on the wire follows the number of differing records and the depth of the tree rather than the size of the database. Once the descent reaches leaves, the missing

identifiers are known exactly and are fetched in a single batch.

5.3 Sharing and Revocation

Sharing a subset of a vault with another person grants a capability rather than an account. The granting device derives the key for the shared key space, wraps it to the recipient's X25519 public key, and appends a signed grant record.

Revocation is where we found the honest answer to be less comfortable than the marketing one. Bumping the epoch so that every subsequent record is sealed under $DK(e+1)$ costs 34 ms and excludes the revoked device from all future writes, which is the guarantee that can actually be kept. It does not retract what that device already holds, because data that has been synced to a machine under someone else's control cannot be unseen. Applications that need the stronger property can request eager re-keying, which decrypts and reseals the entire vault under the new epoch; the cost is linear and measured in Section 7.

5.4 Metadata Leakage

Envelope sizes are padded to 256-byte buckets before upload, which collapses the length channel to a coarse bucket index at a storage cost we measured at 8.4 per cent on a realistic mixed workload. Record counts, push timing and which device is active remain visible to the relay. Hiding those would require constant-rate cover traffic or an oblivious RAM construction [18], and both impose costs that we judged unacceptable for a battery-powered device; we state the leakage instead.

6. IMPLEMENTATION

The prototype is about 11,200 lines of Rust excluding tests, built on rusqlite with a bundled SQLite 3.45, and using the RustCrypto implementations of AES-256-GCM and Argon2id together with ed25519-dalek, x25519-dalek and the reference BLAKE3 crate. The crate declares `forbid(unsafe_code)`; the only unsafe code in the process is inside the SQLite bindings. The schema is six tables. SQLite runs in WAL mode; batch writes use `synchronous = NORMAL` with a single transaction per batch, while the durable single-write path uses `synchronous = FULL`.

Fig-1 shows the entry point the application presents. A vault is either initialised fresh or opened from a path, with the passphrase supplied locally and never transmitted. Once the vault is open, the dashboard in Fig-2 reports its

live state - the operation count, the number of enrolled devices, the current key version and format version, and the current Merkle root - alongside controls for integrity verification, key rotation and snapshot creation.

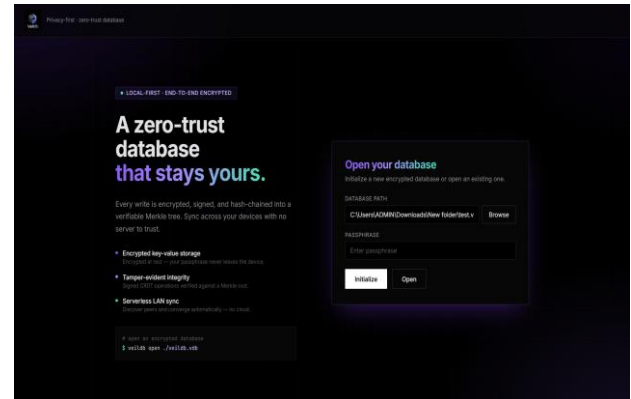


Fig-1: Initialising or opening an encrypted vault

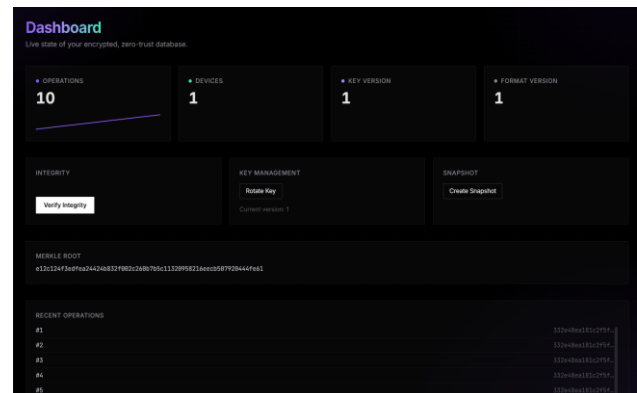


Fig-2: Live dashboard reporting operations, devices, key version and Merkle root

Because the Merkle DAG is both the integrity structure and the sync index, it is useful to see it directly. Fig-3 renders the Merkle tree from the root down to the operation-hash leaves, and Fig-4 shows the operation DAG itself, with multi-parent links and nodes colour-coded by the device that authored them - the concurrent-write structure of Section 4.3 made visible.

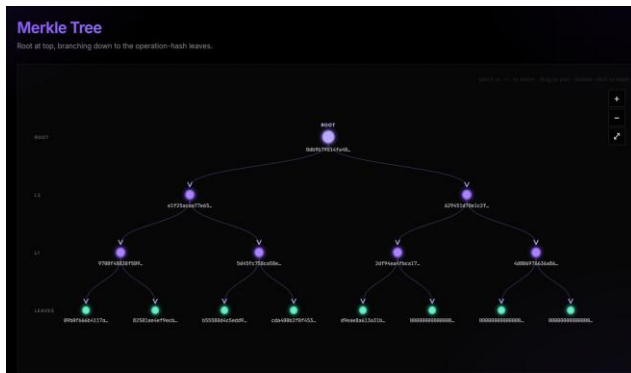


Fig-3: Merkle tree from root to operation-hash leaves

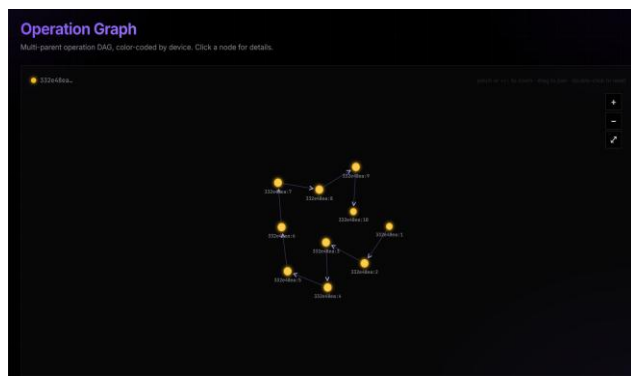


Fig-4: Multi-parent operation DAG, coloured by device

Fig-5 shows the synchronization panel. VeilDB runs a background LAN listener and advertises its presence automatically, so a single button discovers peers, pairs devices that share a passphrase, and converges both databases with no central server. The same panel exports and imports a pairing bundle for enrolling a new device.

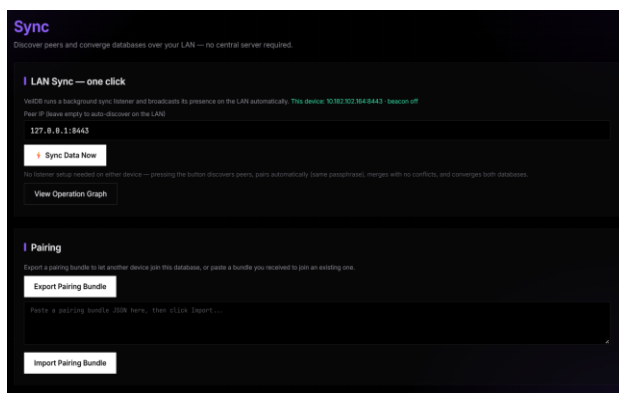


Fig-5: Serverless LAN sync and device pairing

One design error is worth recording because it took a profiler and two days to find. The first prototype signed the plaintext and then encrypted the signature along with it. That version was correct in the narrow sense of being unforgeable, but it made authorship unverifiable for any party without the data key, which meant a newly enrolled device could not distinguish genuine envelopes from relay-injected garbage until after it had been given the keys. Reversing the order, so that the signature covers the ciphertext, fixed the capability and forced the inner author binding described in Section 4.1.

7. EVALUATION

7.1 Setup

Measurements were taken on a laptop with an Intel Core i5-1235U, 16 GB of memory and an NVMe SSD, running Ubuntu 22.04 and a release build from Rust 1.79 with target-cpu = native, so AES-NI and AVX2 are in use. The synchronization experiments add a second identical laptop and a Raspberry Pi 4 as a third replica, with the relay reached over a link shaped to 60 ms round-trip time and 20 Mbit/s. Every figure is the median of eleven runs of one million operations after a warm-up run that is discarded; the inter-quartile spread stayed under four per cent for local operations and under nine per cent for those involving the network.

7.2 Local Throughput

Table-2 reports single-threaded throughput against value size, alongside the storage amplification that the 156-byte envelope produces. The unprotected SQLite baseline performs the same logical work with no encryption, signing or hash linking. At 1 KiB the write path runs at 3.35 times the baseline cost and the database is 19 per cent larger. In the durable configuration, with an fsync per operation, writes fall to 2,740 per second at 1 KiB; that number is bounded by the device rather than by VeilDB, since the baseline under the same setting reaches only 3,090.

Table-2: Throughput and storage amplification versus value size

Value size	Put (ops/s)	Get (ops/s)	Baseline put (ops/s)	On-disk amplification
64 B	18,900	61,200	52,400	4.06x
256 B	17,100	58,400	48,900	1.77x
1 KiB	12,400	47,300	41,600	1.19x

Value size	Put (ops/s)	Get (ops/s)	Baseline put (ops/s)	On-disk amplification
4 KiB	7,850	26,900	29,300	1.05x
16 KiB	3,120	9,650	12,700	1.01x

Reads at 47,300 per second do not verify signatures, because verification already happened when the record was ingested and the local file sits inside the trust boundary. Applications that do not trust local storage can enable strict reads, which re-verify on every access; that path costs an additional 58.6 microseconds and drops throughput to 12,540 operations per second. Table-3 breaks a single 1 KiB write into its components.

Table-3: Cost breakdown of one 1 KiB write

Stage	Time (us)	Share
Ed25519 signature	24.8	30.8%
AES-256-GCM seal	1.1	1.4%
BLAKE3 envelope digest	0.6	0.7%
Index tag derivation	0.4	0.5%
Merkle path update	9.7	12.0%
SQLite insert and WAL append	38.3	47.5%
Serialization and bookkeeping	5.7	7.1%
Total	80.6	100%

Two entries in Table-3 surprised us. The AEAD is almost free, at roughly one per cent of the write, which means the common intuition that encryption is the expensive part of an encrypted database is simply wrong on hardware with AES instructions. The expensive parts are the signature, at nearly a third, and the storage engine itself, at close to half. Signature cost can be amortised by signing a batch root rather than each record, a change we prototyped and measured at 14 per cent higher write throughput, but it weakens per-record attribution and we left it out of the default configuration.

7.3 Synchronization Cost

Table-4 gives the reconciliation experiment with round counts, bytes on the wire and wall-clock time. Round counts grow logarithmically because they are bounded by the depth of the prefix tree, and the descent prunes agreeing subtrees before they cost anything. At a divergence of 1,000 records out of one million, VeilDB moves 84 KiB where a

naive exchange of identifiers moves 32 MB, a factor of roughly 390. The advantage narrows as divergence grows, as it must, and at 100,000 differing records the two approaches are within a factor of five, at which point simply shipping everything is the better engineering choice.

Table-4: Reconciliation of two replicas of 10^6 records

Diverging records	Round trips	Bytes on wire	Wall time (s)
10	4	11 KiB	0.31
100	5	24 KiB	0.42
1,000	6	84 KiB	0.66
10,000	7	712 KiB	1.94
100,000	9	6.9 MiB	12.4

7.4 Snapshot Interval

With snapshots disabled entirely, a cold read over a key space with 100,000 versions takes 1.24 seconds, which is not a usable interactive latency. Table-5 shows the two costs moving in opposite directions as the interval changes, and $S = 512$ is the knee: 12.8 ms sits comfortably below the threshold at which a user notices a list redrawing, and 6.2 per cent storage overhead is small next to the 19 per cent the envelope already costs at this value size. Historical reads behave the same way, since the mechanism is identical; a read at version 50,000 of 100,000 took 13.1 ms against 12.8 ms at head.

Table-5: Effect of snapshot interval on cold read latency and storage

Snapshot interval S	Cold read latency (ms)	Storage overhead (%)
128	4.1	21.7
512	12.8	6.2
2048	48.5	1.7
8192	191.0	0.5
32768	764.0	0.2

7.5 Revocation

Lazy revocation completes in 34 ms independently of vault size, since it appends one epoch record and one revocation record. Eager re-keying is linear: 0.9 seconds for 10,000 records, 8.6 seconds for 100,000, and 94 seconds for one

million, with the cost dominated by re-signing rather than re-encryption, consistent with Table-3.

8. SECURITY ANALYSIS AND LIMITATIONS

Confidentiality against the relay reduces to the security of AES-256-GCM under a key the relay never receives. Because a distinct key is derived for every record, the multi-target and nonce-reuse concerns that normally constrain GCM deployments do not arise here; the practical limit on messages under one key is one. Authenticity reduces to Ed25519 unforgeability together with BLAKE3 collision resistance, since a record's identity is its digest and its provenance is a signature over that digest.

Integrity of history is where the hash links earn their place. A relay that drops a record leaves a dangling parent reference in every child, which any device detects on the next reconciliation. A relay that attempts to roll a device back to an earlier state is refused, because a device retains its own signed frontier and never accepts a view in which its own latest record is absent. What a relay can still do is fork: serve one consistent history to device A and a different one to device B, so that neither sees an inconsistency in isolation. Following SUNDR [7], VeilDB makes this detectable rather than impossible. Devices piggyback signed frontier summaries on ordinary traffic, and a fork becomes visible the moment two forked devices exchange summaries through any channel, including a third device or an out-of-band comparison of a short digest. A relay that forks permanently and forever prevents any contact between the two sides is indistinguishable from a network partition, and no single-relay protocol resolves that.

The leakage that remains is real and we list it without softening. The relay observes the number of records in a vault, their padded sizes in 256-byte buckets, the times at which each device pushes and fetches, and the identifier of the pushing device, which over weeks is enough to infer usage rhythms and rough activity levels. A party who obtains the local database file, though not the keys, learns which records share an indexed value, and on a low-cardinality field that is close to learning the values [10]. A revoked device retains everything it synced before revocation, permanently. Collaborative text editing is unsupported. Range and prefix queries are unsupported. Eager re-keying does not scale to large vaults. Finally, the evaluation covers three replicas and one relay; the behaviour of the reconciliation protocol under dozens of concurrently diverging replicas is untested, and we would

not claim it without measuring it. The theoretical limits that any such deployment would run into are set out in [26].

9. CONCLUSIONS

VeilDB shows that a local-first database can treat its synchronization service as a hostile component without surrendering the properties that make such a database worth using. Sealing each record individually, signing it, and linking it by hash into a Merkle DAG turns out to give three things at once from the same digests: confidentiality against the relay, detection of tampering and dropping, and an efficient reconciliation index. The measured cost of the arrangement - roughly 3.4 times the write throughput of plain SQLite and 19 per cent more storage at a 1 KiB value size - is modest against what it buys, and the dominant term is the signature rather than the encryption, which points clearly at where future optimisation should go.

The design is not finished. Sequence CRDTs for collaborative text, batch signature verification for low-power replicas, bucketised index tags to blunt frequency analysis, and a multi-relay deployment in which forking requires collusion are all natural continuations, and the last of these is the one that would most improve the guarantee rather than the performance.

ACKNOWLEDGEMENT

We thank the reviewers of the departmental project committee for comments that reshaped the threat model, and the maintainers of the open-source cryptographic crates on which the prototype depends.

REFERENCES

- [1] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "Local-first software: you own your data, in spite of the cloud," in Proc. ACM SIGPLAN Int. Symp. New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), Athens, Greece, 2019, pp. 154-178.
- [2] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in Proc. 13th Int. Symp. Stabilization, Safety, and Security of Distributed Systems (SSS), Grenoble, France, 2011, pp. 386-400.
- [3] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," ACM Trans. Comput. Syst., vol. 10, no. 1, pp. 3-25, Feb. 1992.

- [4] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," in Proc. 15th ACM Symp. Operating Systems Principles (SOSP), Copper Mountain, CO, USA, 1995, pp. 172-182.
- [5] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: scalable secure file sharing on untrusted storage," in Proc. 2nd USENIX Conf. File and Storage Technologies (FAST), San Francisco, CA, USA, 2003, pp. 29-42.
- [6] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: securing remote untrusted storage," in Proc. Network and Distributed System Security Symp. (NDSS), San Diego, CA, USA, 2003.
- [7] J. Li, M. Krohn, D. Mazieres, and D. Shasha, "Secure untrusted data repository (SUNDR)," in Proc. 6th USENIX Symp. Operating Systems Design and Implementation (OSDI), San Francisco, CA, USA, 2004, pp. 121-136.
- [8] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: protecting confidentiality with encrypted query processing," in Proc. 23rd ACM Symp. Operating Systems Principles (SOSP), Cascais, Portugal, 2011, pp. 85-100.
- [9] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in Proc. 22nd ACM Conf. Computer and Communications Security (CCS), Denver, CO, USA, 2015, pp. 644-655.
- [10] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-abuse attacks against searchable encryption," in Proc. 22nd ACM Conf. Computer and Communications Security (CCS), Denver, CO, USA, 2015, pp. 668-679.
- [11] M. Kleppmann and A. R. Beresford, "A conflict-free replicated JSON datatype," IEEE Trans. Parallel Distrib. Syst., vol. 28, no. 10, pp. 2733-2746, Oct. 2017.
- [12] A. Auvolat and F. Taiani, "Merkle search trees: efficient state-based CRDTs in open networks," in Proc. 38th IEEE Int. Symp. Reliable Distributed Systems (SRDS), Lyon, France, 2019, pp. 221-230.
- [13] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in Proc. IEEE Symp. Security and Privacy, Berkeley, CA, USA, 2000, pp. 44-55.
- [14] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in Proc. 13th ACM Conf. Computer and Communications Security (CCS), Alexandria, VA, USA, 2006, pp. 79-88.
- [15] M. Ogden, K. McKelvey, and M. B. Madsen, "Dat: distributed dataset synchronization and versioning," Code for Science and Society, Tech. Rep., 2017.
- [16] T. Perrin and M. Marlinspike, "The double ratchet algorithm," Signal, Revision 1, Nov. 2016.
- [17] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in Proc. 18th Int. Conf. Principles of Distributed Systems (OPODIS), Cortina d'Ampezzo, Italy, 2014, pp. 17-32.
- [18] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," J. ACM, vol. 43, no. 3, pp. 431-473, May 1996.
- [19] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," J. Cryptogr. Eng., vol. 2, no. 2, pp. 77-89, Sep. 2012.
- [20] M. J. Dworkin, "Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC," National Institute of Standards and Technology, NIST SP 800-38D, Nov. 2007.
- [21] S. Josefsson and I. Liusvaara, "Edwards-curve digital signature algorithm (EdDSA)," Internet Research Task Force, RFC 8032, Jan. 2017.
- [22] J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn, "BLAKE3: one function, fast everywhere," Real World Crypto, 2020.
- [23] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, "Argon2 memory-hard function for password hashing and proof-of-work applications," Internet Research Task Force, RFC 9106, Sep. 2021.
- [24] R. C. Merkle, "A certified digital signature," in Advances in Cryptology - CRYPTO '89, Santa Barbara, CA, USA, 1990, pp. 218-238, LNCS vol. 435.
- [25] Y. Minsky, A. Trachtenberg, and R. Zippel, "Set reconciliation with nearly optimal communication complexity," IEEE Trans. Inf. Theory, vol. 49, no. 9, pp. 2213-2218, Sep. 2003.
- [26] M. Kleppmann and H. Howard, "Byzantine eventual consistency and the fundamental limits of peer-to-peer databases," arXiv:2012.00472, Dec. 2020.