CaspuleDBv2: A Secure Key-Value Store for Edge

Shubham Mishra Sam Son Nithin Chalapathi
University of California, Berkeley

Abstract

Secure stateful computation in the edge environment is a difficult problem to solve due to limitations on storage capacity and compute hardware. With the advent of Internet-of-Things, the need for confidential computing and secure storage on top of untrusted hardware is increasing.

In this premise, we present CapsuleDB, a key-value store for edge devices that provides confidentiality through trusted execution environments (TEEs). CapsuleDB stores data in cryptographically hardened containers called DataCapsules, which allow for maintaining data integrity and provenance even on top of remote untrusted storage systems. Instead of porting existing key-value stores into a TEE, we build the application from scratch, providing several optimizations in caching, compaction and indexing to overcome the memory limitations of a TEE. We find that even with all TEE and cryptographic overheads, CapsuleDB is only 1.9-3.1x slower than RocksDB, a state-of-the-art unsecure key-value store, for moderate workload sizes.

1 Introduction

Edge computing is a computing paradigm that involves using resources at the edge of a network, rather than relying on centralized resources in data centers (e.g., cloud computing). The main advantage of edge computing is that it can offer low latency and high network bandwidth. However, edge computing is not widely used in practice due to security concerns. Because edge resources are often managed by small groups, they may not be seen as trustworthy as larger cloud providers. In addition, the physical security of edge resources may be more vulnerable to being compromised. These security concerns have limited the use of edge computing in practice.

To address this issue, the *Global Data Plane* (GDP) project [14] proposed a systematic approach based on trusting data rather than infrastructure. GDP uses a storage abstraction called a *DataCapsule* to securely utilize storage resources at the edge. A DataCapsule is a cryptographically hardened directed acyclic graph (DAG) of records, similar to a blockchain. Each record in a DataCapsule is signed and encrypted, including the pointers between records, and is addressed by its hash name. This design ensures the confidentiality, integrity, and provenance of records written to DataCapsules without having to trust the underlying infrastructure.

While the Global Data Plane (GDP) and DataCapsule make it easy and reliable for applications to use edge storage resources, the DataCapsule interface is not user-friendly for applications. DataCapsules only allow a single writer and has an append-only interface for writing new records, which may not be suitable for all types of applications. Additionally, DataCapsules do not provide any performance optimization features, such as caching or signature reduction. To address these limitations, there is a need for storage systems that provide familiar interfaces or Common Access APIs (CAAPIs), such as key-value stores and file systems, and incorporate performance optimization features. On top of that, such storage systems should also be able to securely process data on untrusted machines in order to take advantage of edge resources for computation.

To this end, we present **CapsuleDBv2**, a key-value store that can keep input key-values in memory secure from the untrusted machine it runs on and securely stores data to untrusted remote storage. While CapsuleDBv2 achieves the latter requirement naturally by using DataCapsules for storage back-end, it achieves the former goal through the use of Trusted Exeuction Environment (TEE). Although CapsuleDBv2 mainly targets deployment in edge computing infrastructure, it can be used for any case where an application developer wants to run a key-value store on a cloud machine without exposing their data to the cloud provider.

The design of CapsuleDBv2 poses several challenges. First, it has to be designed to show comparable performance with existing key-value stores while providing strong securiy guarantees. To achieve this, CapsuleDBv2 follows the design of LSM tree-based key-value stores and implements several performance optimizations that take into account the use of TEEs and DataCapsules. Second, the behavior of CapsuleDBv2 must respect the interfaces provided by DataCapsule. To meet this requirement, we define the record format and the DataCapsule structure for CapsuleDB. In addition, CaspuleDB supports eventual consistency and a recovery mechanism in coordination with DataCapsules.

We prototype CapsuleDBv2 using Openenclave and use the following **metrics of success** to evaluate it.

1. We compare CapsuleDBv2 with the unsecure state-of-the-art key-value store, RocksDB in YCSB, and measure how much slowdown it causes compared to other secure key-value stores.

2. We test the correctness of the consistency and recovery mechanism of CapsuleDBv2. We had to manually verify this part as we could not build a standard multi-user benchmark to check the consistency guarantees.

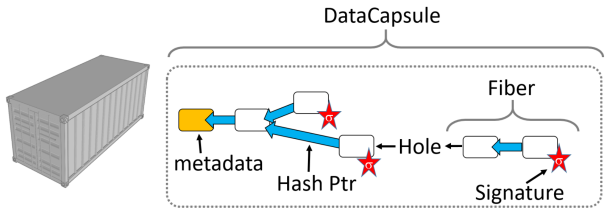

Figure 1: The structure of DataCapsule. The hash of the metadata uniquely defines a single DataCapsule and works as hash name.

2 Background and Motivation

2.1 Trusted Execution Environment

A Trusted Execution Environment (TEE) is a secure memory region that is protected by hardware-based security and mechanisms used to isolate the secure region [8]. TEEs are designed to protect sensitive data and code running in an untrustworthy environment. The isolated area of the machine's memory, known as an *enclave*, is protected by the processor, which denies access from everything else, including the privileged software such as the operating system and hypervisor, but the code in enclave. This ensures that the sensitive code and data running in the TEE are protected from unauthorized access or tampering, even if the rest of the system is compromised.

One downside of using a TEE is the overhead it can introduce. First, TEEs introduce additional context-switching overhead when the control flow switches between enclave code and host code. The context switching into enclave code requires security check for access control, and the switching out of enclave requires the flush of cache and TLB in order to maintain the isolation of the enclave memory. Additionally, the limited size of enclave memory (e.g. 128 MB on Intel SGX) can be a constraint. Although most vendors support the paging of enclave memory to make it possible for applications to use more memory, though this introduces additional overhead as the paged memory must be encrypted and check-summed for security.

2.2 DataCapsule

DataCapsule is a secure storage platform that uses cryptographic techniques to protect data stored on the platform. It is designed for use in untrustworthy infrastructure, where the confidentiality, integrity, and provenance of data must be ensured without relying on the underlying infrastructure. A DataCapsule is a directed acyclic graph (DAG) of DataCapsule records that are encrypted and signed by the writer and use hash pointers to link to each other, as shown in figure 1. The true potential of DataCapsules are realized with a routing infrastructure, which allows DataCapsules to be accessed using their hash names without knowing their exact location. With such an infrastructure, DataCapsule can be shared, replicated and migrated across multiple nodes very flexibly.

One problem with DataCapsules is the intentionally limited interface in order to make the consistency simple and to provide flexibility. DataCapsules allow append-only access for writing and record-level access. This means that a storage system must be built on top of DataCapsules to provide a more familiar interface for applications.

2.3 LSM Tree-based Key-value Store

The idea of Log-structured Merge Trees (LSM Trees) was introduced by O'Neil et al. in [16] as a data structure for fast key-value pair writes and lookups. A simple LSM Tree consists of two components: a fixed size Memtable located in volatile memory and disk-stored collection of sorted string tables (SSTables). When a key-value pair is written to an LSM tree, the write goes to the Memtable. When a Memtable fills up, its contents are marked immutable and the keys are sorted to create an SSTable. The generated SSTable is flushed to disk. This process is called *Minor Compaction*. Periodically, a background thread combines multiple SSTables together to remove redundant keys. This process is known as *Major Compaction*. Google uses the LSM tree as the basis for LevelDB and BigTable [5].

Facebook introduced RocksDB [10] with leveled compaction. In RocksDB, SSTables are organized into multiple levels with increasing sizes. The first level (L0) remains the same as before, whereas, L1 and above contain SSTables with non-overlapping sorted key ranges such that a given key can only exist in one SSTable per level. If layer L_i is filled, one of the SSTables is selected for eviction. The chosen SSTable is combined with SSTables containing overlapping key ranges in layer L_{i+1}. In this way, old key-value pairs trickle down the layers, reducing the tail latency of reads.

3 Design

CapsuleDB is an LSM tree-based key-value store that uses a Trusted Execution Environment (TEE) for secure computation and DataCapsules for secure remote storage. CapsuleDB guarantees the confidentiality and integrity of data even in untrustworthy environments. Additionally, the use of DataCapsules for storage allows for the persistent state of the database to be location-independent, replicated, and migrated easily. Designed for use with edge resources, CapsuleDB is also suitable for any scenario where strong privacy and remote computation are jointly required.

3.1 Overall Architecture

The figure 2 shows the overall architecture and components of CapsuleDB. CapsuleDB is designed for remote users and

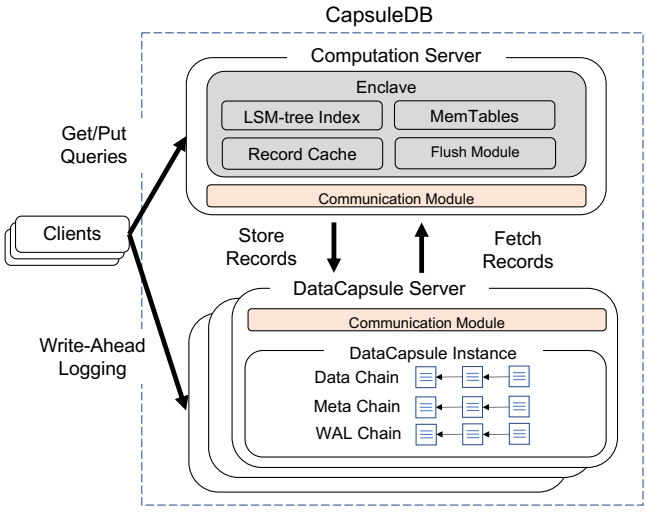

Figure 2: The overall architecture of CapsuleDB

remote storage while supporting eventual consistency for remote clients (3.7). We refer to the machine on which the CapsuleDB's key-value store management program runs as the *computation server*. Machines that store DataCaspules and provide the DataCapsule interface are *DataCapsule servers*. The black arrows between the components represent the communication through the network which has the capability to route DataCapsule.

The computation server is responsible for maintaining in-memory states required for LSM-tree structure and serving key-value store queries from the clients (e.g., `Get()`, `Put()`) within the enclave. CapsuleDB assumes that there is secure communication channel between the clients and the computation server, or that the input key-values are properly encrypted and check-summed. Similar to other LSM tree-based database, the computation server first buffers key-value pairs in a `Put()` query into *Memtable*s and flushes(3.6) to a persistent table (3.4) when a MemTable becomes full. The flush thread wraps the table in a DataCapsule record and sends it to the DataCapsule servers. The server maintains the hash of newly created records in the LSM-tree index, categorizing each record by the level it belongs to.

A `Get()` query is first served from the MemTables. If the query is not found in the MemTables, the server searches through the LSM-tree index to find the target record, which contains the queried key-value. The index contains a bloom filter and the keys of each record for efficient search (3.4). If the target record is found in the enclave's *record cache* (3.5), where recently fetched records are cached, the query is served promptly. Otherwise, the computation server sends a record fetch request to the DataCapsule server and then serves the query.

Threat Model. CapsuleDB assumes two different threat models depending on the type of server. For computation servers, the "cloud/edge attacker" model [6] is assumed, where the attacker has full control over the system software but is not able to compromise the processor's TEE. For DataCapsule servers, the "untrusted utility provider" [14] model is assumed. The owner of the DataCapsule servers is responsible for providing DataCapsule services but may still attempt to access the data stored in DataCapsules. CapsuleDB does not guarantee defense against availability attacks or side-channel attacks, inline with existing works. One may replicate DataCapsules over servers with different owners to guarantee high availability, but this is not considered in CapsuleDB's security guarantees.

3.2 DataCapsule Server

A single DataCapsule server is responsible for managing one DataCapsule instance of CapsuleDB. A DataCapsule may be replicated over different servers. In the case of replication, the servers are collectively referred to as *DataCapsule servers*, and each server is responsible for managing consistency between replicas. This job is relatively simple because DataCapsules are CRDTs [13, 14].

The primary job of DataCapsule servers is to store and fetch records in the DataCapsule requested from the computation server. If there are multiple replicas, the computation server simply sends a write request to every server and uses the result of a read request coming from any server. Upon receiving a new write request, each server verifies its signature to check if the record is sent from the writer of the DataCapsule. The record is encrypted by the writer (computation server). After verification, each server stores the record in its local storage as is and updates the storage index with the hash name. The records sent from the computation server form a Data chain or Meta chain depending on the type of the record (see 3.4). For a read request, each server simply looks up its storage index with the hash name and returns the record to the computation server. It is the computation server's responsibility to check if the hash of the returned record matches the hash name.

In addition, DataCapsule servers manage a write-ahead log of CapsuleDB. CapsuleDB assumes the clients (applications) do write-ahead logging before sending a query to the computation server. With this coordination with the clients, the computation server can be recovered when crashes happen. The write-ahead logs are maintained in a separate chain per client called WAL chain.

Besides, we expect DataCapsule servers to provide one more service to CapsuleDB: a freshness service. The freshness service returns the sources (the latest records of each branch) in DataCapsule DAG. This service is needed for CapsuleDB to recover its state after a normal shutdown or crash (discussed in 3.8). However, the freshness service can be used as a means of replay attack because our threat model assumes a DataCapsule server may act maliciously. To solve this issue, we replicate the DataCapsule over $3f+1$ servers and use a Byzantine quorum among the replicas in generat-

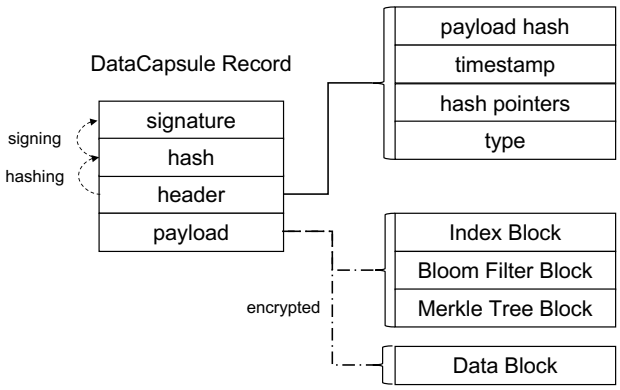

Figure 3: DataCapsule record format in CapsuleDB

ing the result of freshness service under the assumption that there are at most f malicious DataCapsule servers. Note that replay attacks are not possible while CapsuleDB is operating normally as it knows what the latest records are.

3.3 Switchless Call

CapsuleDB is designed to minimize the overhead of switching between enclave and host functions in the computation server. To facilitate this, CapsuleDB uses a switchless design. Enclave functions can indirectly invoke host functions through a communication buffer, and vice versa, instead of calling ecalls and ocalls directly. CapsuleDB uses separate worker threads for the enclave and host functions to serve indirect invocations. When a caller wants to initiate an indirect call, they put the function name and arguments into the communication buffer. The worker threads poll the buffer to see if it has been filled. When a request is found, the worker thread invokes the requested function in its thread context. The result of the function is returned through the communication buffer.

3.4 Table and Record Format

CapsuleDB has a table format that is based on SSTable [5]. A table in CapsuleDB includes four blocks: a data block, an index block, a bloom filter block, and a Merkle tree block. The data block stores all of the key-value pairs flushed from the MemTable. The index block contains an index for binary search; an array of (key, offset in data block) pairs that are sorted by key is stored. The bloom filter block is a bloom filter used to filter out most false positives. The Merkle tree block contains a Merkle tree whose leaves are the key-value pairs in the data block. The Merkle tree is used to verify the integrity of the key-value pairs when a table is moved from host memory to enclave memory.

Although the four blocks conceptually belong to a single table, CapsuleDB separates them into two separate records: a *DataRecord* for the data block, and a *MetaRecord* for the other blocks. This is useful when the key distribution is

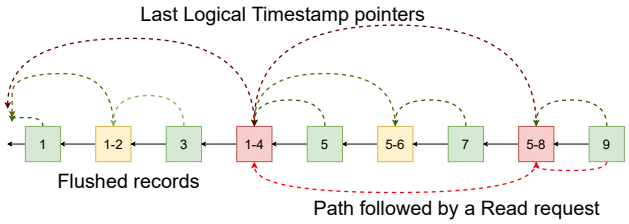

Figure 4: Eager Compaction when $c = 2$, $d = 2$

highly skewed and some records in the working set do not have any keys. The index and bloom filter blocks can be used to check whether a queried key exists in the current set of records. This removes the need to load irrelevant data blocks from the DataCapsule server into memory. By storing the DataRecord and MetaRecord in different hash chains (Data chain, Meta chain), the computation server can avoid loading the DataRecord unless absolutely necessary.

Figure 3 shows DataCapsule record format used in CapsuleDB. The payload of a record is the encrypted data of either the data block of the table or the other three blocks depending on the record type. Each record has a record header and the hash of the header which uniquely identifies a single record (i.e., the hash name). Finally, it contains a cryptographic signature forged from the header hash and CapsuleDB's writer key. The signature proves the CapsuleDB's identity to DataCapsule servers and guarantees the integrity of the whole record. The header includes the following fields: the hash of the payload, the timestamp it is created, the hash pointers to the previous records, and the type of the record. In our prototype, each record only has a hash pointer to the most recently written record in the same chain.

3.5 Enclave Caching

Since a DataCapsule record contains encrypted data and is sent from unreliable storage and the network, its contents and signature need to be decrypted and verified in the enclave. The verification causes the latency of a single read involving record fetching to be very high. Fortunately, the records are accessed in the reverse order of flushing time (i.e., the latest one is accessed first). Real-life workloads have a high degree of locality; recently written keys will be accessed more often than the older keys, and keys written together tend to be accessed around the same time. Therefore, the cost of the record fetching can be reduced by caching it. Concretely, CapsuleDB maintains an in-enclave cache of the recently fetched records. The enclave cache has a fixed number of slots and stores an entire record that is decrypted and verified in the enclave in each slot. It follows Least-Recently-Used (LRU) for cache eviction policy.

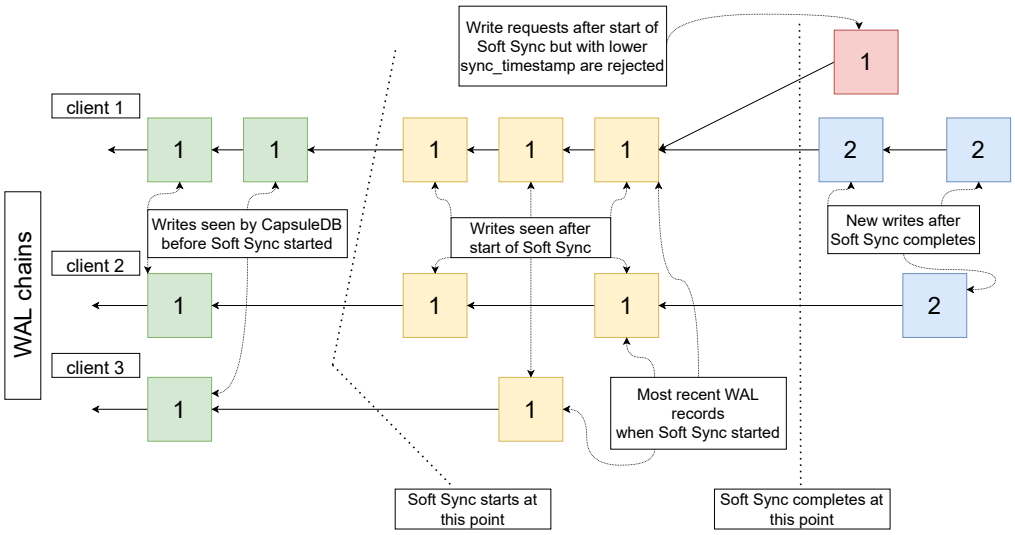

Figure 5: The Soft Sync Protocol. Numbers denote *sync_timestamp* recorded by each WAL record.

3.6 Eager Compaction

The process of compaction is essential to reduce the number of records searched for a key. Implementing Leveled Compaction like RocksDB is difficult in CapsuleDB for three reasons. (1) Reading older records separately for compaction means more network bandwidth usage. (2) Enclaves cannot efficiently handle multiple threads, especially if they are pinned to the same physical core. Hence, the compaction and main read-write procedures need to be multiplexed onto the same thread, causing unnecessary long blocks in reads and writes. (3) Since Leveled Compaction compacts older records, reading in older records will evict recent hot records from the cache.

Not compacting Memtables at all may lead to very long tail latency in Read requests. To mitigate this problem, we introduce *Eager Compaction*. CapsuleDB periodically combines a few recently flushed records with the next record to be flushed. Eager Compaction has two parameters: skip-length, c, and max-height, d. For every c^i-th ($i \in \{1,2,...,d\}$) record to be flushed, the last $(c^i - 1)$ flushed records are combined with it. The MetaRecord corresponding to each compacted record contains a pointer (*last_logical_timestamp*) to the $(c^i - 2)^{th}$ record before it. While searching for a key, the previous $(c^i - 1)$ can be skipped as all their information is effectively stored in the new compacted record. The entries for these records in both the index and the cache can be purged.

Eager compaction reduces the number of records to be searched by a factor $O(c^d)$ and only requires $O(d\log(c))$ more comparisons per record during binary searching for a key.

Algorithm 2 describes the full flushing process with synchronization (Section 3.7).

3.7 Consistency and Synchronization

CapsuleDB provides an eventual consistency guarantee. Although a network adversary might partition the network for brief periods of time, we work with the assumption that the network behind the multicast provides ordered and eventual delivery. We also assume that CapsuleDB knows a unique identifier endowed with a total ordering for all of its writers.

Writing to CapsuleDB involves two stages:

- First, the writers send the write request to the DataCapsule server, where writes form separate hash-chains for each writer. This acts as Write-ahead logging (WAL) for CapsuleDB.

- After the first step successfully completes, the same write request is sent to the computation server, which accepts or rejects the writes based on a lossy vector clock (Section 3.7.1) which the writer presents in each write request.

These two steps can be unified by broadcasting the write request over a multicast tree, but then the client should make sure that writes to DataCapsule servers always pass, or have DataCapsule servers call back to the computation servers with confirmation. As there can be arbitrary network delay between the first and the second steps, the computation server periodically synchronizes between the DataCapsule servers and itself (Section 3.7.2). It is important to note that the synchronization procedure described below is different from the *Sync Report* based synchronization used in CapsuleDBv1. In this design, writes are not blocked, although they can be rejected. Also, the need for an explicit Sync Report to be flushed is eliminated. Instead of CapsuleDB globally instructing all clients to stop writing, it now informs the clients of

potential conflicts and leaves it up to them to decide whether to coordinate with each other or not.

3.7.1 Lossy Vector Clock

The state of a client c_i is given by a Vector Clock, $V_i = \{(c_1,t_1),(c_2,t_2),...\}$, where the tuple (c_j,t_j) denotes that client c_i interacted with client c_j at timestamp t_j. We also use the notation $V_i.c_j$ for t_j. For $i \neq j$, t_j is incremented by one each time c_i exchanges message with c_j. Client c_i's own timestamp t_i is incremented when it sends a message to CapsuleDB.

However, to reduce the memory and bandwidth overhead of using the Vector Clock with every message, a client only retains the timestamp of a few other clients besides itself. This idea is borrowed from the design of Dynamo [9]. A client is free to choose which other clients' timestamps it wants to retain, defaulting to retaining the timestamps of clients with which it has to interact.

We also define a merge operation on two clocks V_i and V_j as $merge(V_i,V_j) = \{(c_1,t_1),(c_2,t_2),...\}$ such that if c_k is exclusively in V_i or V_j, $t_k = V_i.c_k$ or $t_k = V_j.c_k$, respectively. Otherwise, $t_k = max(V_i.c_k,V_j.c_k)$.

CapsuleDB causally orders writes to the same key using the Vector Clocks which are sent to CapsuleDB with each write. For causally concurrent writes, CapsuleDB breaks the tie using the predefined total order of the clients' unique identifiers. The key-level granularity of ordering is achieved by storing the timestamp $V_i.c_i$ with each key for the last writer c_i of the key. However, due to the lossy nature of these clocks, two clients' clocks may not have each other's entries. In that case, the causal relation may become indeterminate. CapsuleDB defaults to a First-Writer-Wins policy here. This is not a serious problem in practice, as Soft Sync (Section 3.7.2) happens with a short time period and a failing writer can retry after a Soft Sync with an updated clock.

CapsuleDB maintains a global vector clock in the computation server which is merged with the vector clock of each successful write request. If a client fails to write to a key, it updates its own clock using CapsuleDB's global vector clock and retries the request.

3.7.2 Soft Sync

In order to periodically synchronize the WAL hash-chains in DataCapsule with the state of Memtables, CapsuleDB launches a Soft Sync protocol that synchronizes these entities without blocking any writes or adding data to DataCapsule (hence the name *Soft* Sync). This protocol is loosely based on the Chandy-Lamport algorithm for global snapshots. [4]

CapsuleDB's global clock also maintains a monotonically increasing atomic counter called *sync_timestamp*. Periodically, CapsuleDB uses DataCapsule server's freshness service to get the latest WAL entries for each user and merges the Vector Clocks in these records to get a target clock. When the incoming writes cause the global clock of CapsuleDB to be equal to the target clock, *sync_timestamp* is incremented and its new value, with the latest global clock is broadcast to all the clients using the multicast tree. The clients update their vector clocks using this global clock broadcast.

Clients also include the last *sync_timestamp* seen by them in the write requests. Writes are only accepted by CapsuleDB if the *sync_timestamp* values match with that stored by the global clock. This also helps to mitigate the problem of overflow in timestamps of the vector clock. [19] A write request with higher *sync_timestamp* than the last write that updated the same key always succeeds.

The Soft Sync protocol is also performed before every flush and a snapshot of the global clock is included in the MetaRecord.

Although we do not prove here formally due to lack of space, the combination of Vector Clocks and Soft Sync guarantees the following conditions:

- **Condition 1**: A write request is successful if and only if it is done to a new key or if it causally happens after the previous write of the key or if (for concurrent writes) the new writer is ordered higher than the old writer.
- **Condition 2**: All write requests after Soft Sync causally happen after any write request prior to the Sync.

Algorithm 1 summarizes the Soft Sync protocol.

3.7.3 Handling Stateless Clients

The synchronization protocol described above dictates the clients store a vector clock as a state. However, in general-purpose usage, many applications may need a stateless client. We suggest two ways for achieving this:

1. The clients query the global clock from CapsuleDB before every write. This makes CapsuleDB the sole point of coordination in the system.
2. A client proxy runs as a sidecar to CapsuleDB, which handles sessions with multiple clients and maintains the vector clock for each session.

The second method is more geared towards general-purpose usage, where CapsuleDB can be used as a drop-in replacement for any eventually consistent database.

3.8 Recovery

CapsuleDB provides atomic writes, but does not provide any multi-operational transaction guarantee. Also, the writes are not immediately flushed to the DataCapsule (similar to No-Force policy in [11]). Hence, it suffices to have our WAL hash-chains act as a logical redo log.

When a crash or normal shutdown happens, CapsuleDB can recover the in-memory state of the computation server up to the last successful write to the Memtable before the event. The recovery happens in two phases as follows:

- Using the freshness service provided by the DataCapsule servers, the computation server identifies the last flushed record in the MetaChain and rebuilds the index for that record first. Then it follows the last logical timestamp pointers (see 3.6) to build the in-memory index for all the necessary flushed records.
- The global clock snapshot in the latest MetaRecord, V_s, corresponds to one WAL record for each user. In this phase, using the freshness service, the computation server finds out the latest WAL record for each user. Then for each user, it walks backward in the WAL chain to the first entry which causally happened before V_s. CapsuleDB now combines this per-user analysis of the writes into a global order of writes to apply. Then it applies them to its Memtable in a forward pass.

Assuming that the writes to the DataCapsule servers are atomic, this recovery operation is idempotent. Therefore, if a crash happens during recovery, the same process can be applied again to regain the same state.

4 Implementation

We have developed a performant prototype for CapsuleDB in C++ using the OpenEnclave SDK[1], which is a cross-platform vendor-agnostic framework for developing enclave applications. Our codebase contains about 7600 lines of code, excluding code for tests and client. Now we discuss the salient features of our implementation.

Reducing copy overhead between host and enclave. To reduce the amount of data copied between the host and enclave during switchless calls (Section 3.3), we exploit the fact that host-side buffers allocated in the heap memory are visible to the enclave. We identify large buffers that are passed to and from enclave and host (e.g., the buffer containing the result of a read query). We allocate these buffers on the host side and pass the pointer as a raw 64-bit address to the enclave. We also make sure that no sensitive enclave data is ever placed on these buffers.

Memtable design. We use have tried a few lookup data structures for our Memtable. Initially, we developed our Memtable using C++ STL's `std::map`. This design stores keys in sorted order, hence we do not need to sort the keys separately during flushing. However, the average case complexity for insertion and lookup is $O(\log(N))$ which proved as a bottleneck in large workloads. Hence, we switched to `std::unordered_map` with MurmurHash3[2] hash function. This reduces the average time complexity to $O(1)$, boosting our performance. In the future, we want to experiment with other data structures like Skip Lists, Tries, and B-trees.

Fast Binary Search in DataCapsule Records. If the index stored the key strings in lexicographical order, each comparison during binary search would take time linear in the lengths of the keys. To reduce this overhead, we compute 128 bit hash of each key using MurmurHash3 and store $\langle hash, key \rangle$ in sorted order. While searching for a key, we first hash the key and primarily use this hash for binary search and only resort to string matching if the hashes match. This allows us to compare two strings in amortized constant time (only 2 64-bit integer checks).

Bloom Filter. While searching for a key, in order to reduce the number of index lookups in records where the key is not present, we implement a simple Bloom Filter. Given a desired false positive rate and Memtable size, we compute the ideal size of the bitset and the ideal number of hash functions.[3] However, using very large Bloom Filter reduces the throughput due to extra overhead of storing and flushing a large bitset. We have experimentally determined the optimal size of a Bloom Filter.

Communication. We have implemented two separate transport mechanisms for communication between clients, computation server, and DataCapsule servers.

1. We implement a name-based overlay routing protocol using gRPC[4], called `towncrier`. Instead of specifying IP addresses, processes can join a `towncrier` network, by registering its name to a daemon running in the machine. For example, CapsuleDB can register with the name `cdb`, DataCapsule servers as `dc1`, `dc2` and so on and clients as `client1`, `client2` and so on. Messages can be directed to a specific name (e.g., `dc1`) or can be broadcast to all names with a given prefix (e.g., `dc`). The `towncrier` daemon tries to forward the messages it receives to the locally registered names. If not possible, it broadcasts the messages to other peering daemons over the network. We use this transport mechanism for our WAL and Soft Sync (Section 3.7) operations, but since gRPC uses HTTP underneath, the communication overhead is very high.

2. For low overhead communication, we make the communication between the client and CapsuleDB on the same machine using Named Pipes. The communication between CapsuleDB and DataCapsule servers happens over ZeroMQ[5]. We use this setup for our performance evaluation.

[1] https://openenclave.io
[2] https://sites.google.com/site/murmurhash/
[3] Adapted from https://corte.si/posts/code/bloom-filter-rules-of-thumb/
[4] https://grpc.io/
[5] https://zeromq.org/

Table 1: Machine Configurations

	DC8s_v3	D2ads_v5
CPU	Intel Xeon 8370C 2.8GHz * 8	Intel Xeon 8272CL 2.6GHz * 2
Memory	64 GB	8 GB
Storage	X	75 GB, 9000 IOPS
TEE	Intel SGXv2	X

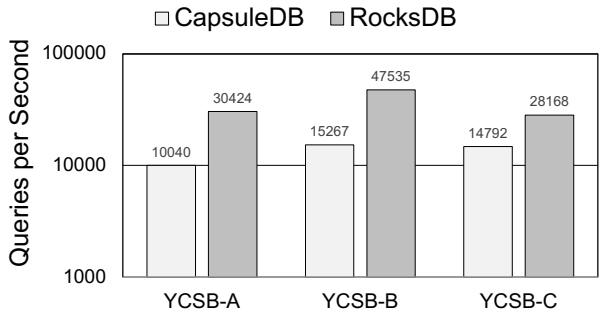

Figure 6: Overall throughput of CapsuleDB and RocksDB. 1M keys and 1M queries used in YCSB-B and YCSB-C, and 500K keys and 1M queries used in YCSB-A. Read-Update ratio is 50:50, 95:0, and 100:0 in YCSB-A, YCSB-B, and YCSB-C, respectively

5 Evaluation

Methodology. We use the Yahoo Cloud Serving Benchmark (YCSB) [7]. YCSB is a standard benchmarking tool for DBMS's with support for varying workloads; we develop a CapsuleDB java client to link with YCSB. Across all comparisons, unless specified, we use 1M queries with 200B records. We use Azure DC8s_v3 for the computation server and D2ads_v5 for the DataCapsule servers (see table 1). The client processes are spawned in the same machine with the computation server for fair comparison as it is difficult to simulate remote clients for RocksDB.[6]. To simulate the network cost incurred by CapsuleDB when communicating with the Data Capsule Server, we use an NFSv4 for disk storage for RocksDB.

5.1 Comparison with RocksDB

Performance. The figure 6 shows the processed queries per second of CapsuleDB and RocksDB in YCSB benchmarks. RocksDB shows 3.0x, 3.1x, and 1.9x faster performance than CapsuleDB. Despite a little slower performance, CapsuleDB still can offer comparable performance to the state-of-the-art non-secure key-value store while providing security guarantees and flexibility. This makes CapsuleDB a promising option for those looking for a secure and flexible key-value store. As a comparison, secure CapsuleDBv1 was 10x slower than non-secure CapsuleDBv1, which is much slower than RocksDB [15]. In addition, although the results were measured at a much larger scale, Tweezer and SPEICHER showed a slowdown of about 5x-30x compared to RocksDB [2, 12].

[6]While we tried to implement a YCSB java client for both Tweezer and Speicher, we ran into several errors. We were unable to contact Tweezer's authors to sort this issues out.

DB Type	Median Latency (μs)	Max Latency (μs)
CapsuleDB	55	1141759
RocksDB	22	103551

Table 2: Median and max latency of processing each read query in YCSB-C. The median represents the case the query hits in the MemTables, and the max represents the case the query causes fetching a record (or a block for RocksDB) from the storage.

Overhead Analysis. We have identified two factors that affect the slower performance of CapsuleDB compared to RocksDB. The first factor is that the processing of read queries in the MemTables is slower in CapsuleDB than in RocksDB, despite our efforts to optimize the MemTable implementation and use switchless calls. The first column in table 2 shows the median latency of CapsuleDB and RocksDB in processing read queries in the YCSB-C workload, where a read query hits the MemTables. CapsuleDB's median latency is 2.5x longer than RocksDB's median latency. We suspect that the switchless architecture of Openenclave is not efficient enough to hide microseconds-scale latency, and our MemTable implementation may not be as efficient as RocksDB's.

The other factor is CapsuleDB's limited ability to fetch data, as it can only retrieve whole records from the DataCapsule, while RocksDB can read blocks of data in 4KB page granularity from NFS storage. This design choice is enforced by the interface provided by the DataCapsule, but it means that CapsuleDB must read entire records even when only a few key-value pairs are needed. The second column in table 2 shows the maximum latency in processing read queries, representing the case fetching a record (or a 4KB block for RocksDB) from the remote storage. CapsuleDB's maximum latency is about ten times slower than that of RocksDB. This is because CapsuleDB has to fetch the entire record, which is tens of thousands times bigger than a single 4KB block, even when it only needs one key-value pair from the record. Improving this aspect of CapsuleDB's performance would require changes to the DataCapsule interface to allow for more fine-grained data retrieval.

Overhead of Cryptographic Functions. Contrary to expectations, cryptographic functions do not appear to be the primary performance bottleneck in the current prototype and workloads. In CapsuleDB, the cryptographic functions are heavily used in processing record creation and enclave cache miss operations. We measured the effect of these functions on

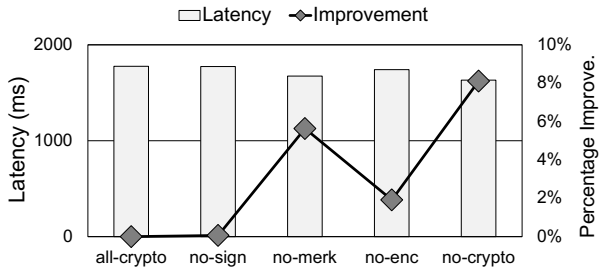

Figure 7: The effect of cryptographic functions on record creation. "all-crypto" represents the case where all of signature creation, merkle tree creation, and encryption are enabled, while "no-sign," "no-merk," and "no-enc" represent the cases where only one of them is disabled, respectively. "no-crypto" represents the case where all of them are disabled.

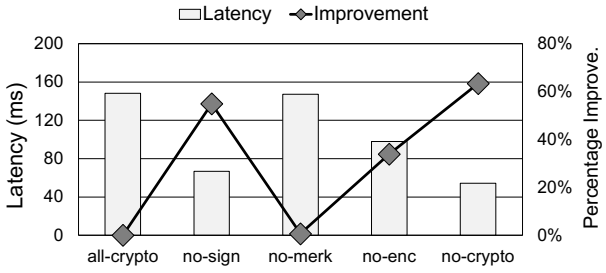

Figure 8: The effect of cryptographic functions on the enclave cache miss. The setup is the same as the previous figure. The latency represents the time to verify and decrypt a single record when the enclave cache miss happens.

these operations. Figure 7 shows the latencies of record creation with and without cryptographic functions. Even when all cryptographic functions are disabled, the overall latency only improves by 8.1%. Instead, composing the blocks of the table and copying data take up most of the time.

Figure 8 shows the latencies of verifying and decrypting a record when an enclave cache miss occurs, with and without cryptographic functions. In this case, disabling signature verification and decryption improves latency by 54.9% and 33.9%, respectively. The improvement increases up to 63.4% when all functions are disabled. However, the overall latency in enclave cache miss is relatively smaller than the record fetching time, and the cache misses occur not so frequently. Therefore it does not significantly affect overall throughput. This overhead could be a serious performance bottleneck in workloads where enclave cache misses occur frequently, but we did not test this case due to limitations in the current prototype, explained in 5.3.

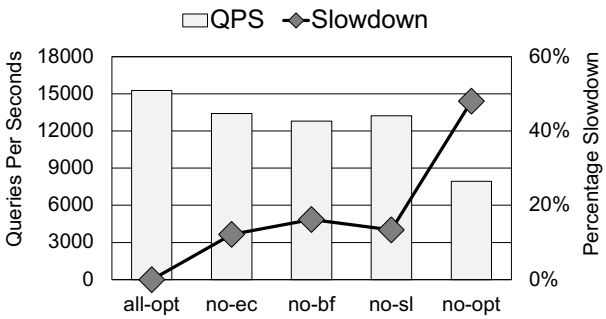

Figure 9: The effect of performance optimizations on throughput. "all-opt" represents the case where all optimizations are enabled and tuned in CapsuleDB, while "no-ec," "no-bf," and "no-sl" represent the cases where only one of eager compaction, bloom filters, and switchless calls is disabled, respectively. "no-opt" represents the case where all three optimizations are disabled. The YCSB-B workload is used in all cases.

5.2 The Effect of Optimizations

We measure the performance of CapsuleDB in cases where some of the performance optimizations are disabled in order to quantify their effect on the end-to-end performance. Figure 9 shows the queries per second of CapsuleDB in YCSB-B, measured in 5 different setups. "no-ec", "no-bf", and "no-sl" cases demonstrate that eager compaction, bloom filer, and switchless architecture improves the end-to-end performance by 12.2%, 16.2%, and 13.4%, respectively. They work more effectively when used together than when applied independently. As a result, the overall performance of "all-opt" is measured 48% faster than "no-opt" configuration, larger than the sum of the effects of each optimization.

5.3 Current Limitations

The prototype of CapsuleDB is experiencing a thrashing issue with the enclave cache in large-scale workloads. The current design uses whole records as the unit of caching, which makes it easy to verify their integrity using the DataCapsule header. However, when the working set of clients is distributed over multiple records, and the size of these records is larger than the enclave size (e.g. 128MB in Intel SGX), the cache entries need to be replaced frequently, leading to thrashing.

One solution to this problem is to split the data blocks into smaller chunks that can be verified independently, and use these chunks as the unit of caching. This design reduces the overhead of each cache replacement and allows for larger working sets. Other TEE key-value stores that use LSM trees have successfully implemented this approach to avoid thrashing. SPEICHER [2] uses the 32KB chunk size and builds the Merkle tree for each record whose leaves are the hash of the 32KB chunks. Tweezer [12] makes every single key-value

pair to be verifiable by augmenting each key-value pair with HMAC of it. Implementing this design in CapsuleDB is an important future task in order for it to be used in more realistic scales.

6 Applications

We built CapsuleDB as a general-purpose secure key-value store which can be used as a drop-in replacement of any other database with similar eventual consistency guarantees. This can be done with the help of client proxy services as described in Section 3.7.3. The client model described in this paper is, however, more suited to stateful clients who themselves are running code on TEEs. One such environment is Paranoid Stateful Lambda (PSL) [6], a secure Function-as-a-service (FaaS) platform. PSL consists of workers running code in TEEs. They themselves maintain a Memtable cache and communicate with each other through a multicast tree. CapsuleDB runs as a special worker in the PSL environment and provides a method for secure concurrency.

7 Future Works

Aside from the future task in 5.3, we have some future plans to extend CapsuleDB.

Verification of concurrency guarantees. Currently, our eventual concurrency has not been tested or proven rigorously. We would like to develop a multi-user workload to verify the correctness experimentally and also prove the correctness of a formal model using model checkers.

Sharding and Load Balancing. We currently assume CapsuleDB to run in a single machine. However, as the load increases, we should distribute it in multiple machines. Two ways of performing this are sharding, where key ranges are assigned to a machine, and State Machine Replication (SMR). We want to investigate the security implications of Load Balancing among different machines and ways to perform SMR with encrypted data.

8 Related Work

Persistent TEE Key Value Databases. There are currently two existing persistent key value stores which leverage a TEE, specifically Intel SGX: Speicher [2] and Tweezer [12]. Speicher first introduced the concept of a secure persistent KVS and extends the RocksDB codebase to address three security concerns: (1) data confidentiality, (2) data integrity, and (3) detection of stale data. In order to authenticate the LSM tree stored in untrusted memory, Speicher computes a MAC for each SSTable data block and builds a Merkle tree from each block MAC. To meet the low memory constraints of enclaves and to reduce EPC paging, Speicher performs two optimizations. First, the RocksDB MemTable is split into a key MemTable and value MemTable, where the key MemTable is stored in enclave memory and the value MemTable is encrypted and stored in untrusted memory. Second, in order to prevent context switching between the enclave and host memory process, a separate unsecured thread processes I/O system calls. Finally, to ensure data freshness Speicher uses a custom asynchronous monotonic counter secured by the synchronous SGX monotonic counter to perform versioning over the log and MANIFEST files.

Tweezer builds on the ideas of Speicher by addressing the scalability issues encountered in Speicher. Tweezer generates a unique MAC key for each SSTable. Using this key, freshness is checked by building a Merkle tree across SSTables. The same MAC key is used to generate a new MAC over each key-value pair within an SSTable, whereas Speicher generates a MAC for each data block. These two design choices reduces read amplification and the number of decryption calls performed. To address versioning, Tweezer uses a hash chain since the synchronous SGX monotonic counter is deprecated; the chain is extended by heartbeat transactions run by a client.

In contrast to both works, CapsuleDBv2 utilizes the Global Data Plane and its benefits, including location-independence, replication, and ease of migration. CapsuleDBv2 provides many of the same guarantees afforded by Tweezer and Speicher. However, CapsuleDBv2 currently lacks certain features from RocksDB that both Tweezer and Speicher inherit due to being extensions of RocksDB. We anticipate that, with time, CapsuleDBv2 will provide some of the essential features and comprehensive testing found in RocksDB.

Shielded Execution Environments. CapsuleDBv2 uses the Openenclave SDK but there exists a variety of execution environments to run an unmodified codebase within an enclave. Notably, Scone [1] (used by Tweezer and Speicher), Haven [3], Graphene-SGX [18], and Panoply [17] are all environments which port an existing codebase to run in an enclave. However, porting an entire codebase, especially for large persistent key-value stores, will run into EPC memory limitations and severe overhead from EPC paging.

9 Conclusion

In this paper, we have presented the design of CapsuleDB as a TEE-based secure Key-Value store running on top of remote untrusted storage. We have suggested several performance optimizations over the previous designs and experimentally proven the efficacy of these design choices. We have also benchmarked it against RocksDB, an unsecure key-value store, and discussed our overheads over the same. Finally, we also provide a simple protocol to establish eventual consistency and discussed potential applications for the same, especially in secure FaaS environments running on the edge.

References

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 689–703, USA, 2016. USENIX Association. event-place: Savannah, GA, USA.

[2] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. Speicher: Securing lsm-based key-value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 173–190, 2019.

[3] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.*, 33(3), August 2015. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[4] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, feb 1985.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[6] Kaiyuan Chen, Alexander Thomas, Hanming Lu, William Mullen, Jeffery Ichnowski, Rahul Arya, Nivedha Krishnakumar, Ryan Teoh, Willis Wang, Anthony Joseph, et al. Scl: A secure concurrency layer for paranoid stateful lambdas. *arXiv preprint arXiv:2210.11703*, 2022.

[7] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[8] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Paper 2016/086, 2016. https://eprint.iacr.org/2016/086.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating System Principles*, 2007.

[10] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), oct 2021.

[11] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, dec 1983.

[12] Igjae Kim, J Hyun Kim, Minu Chung, Hyungon Moon, and Sam H Noh. A log-structured merge tree-aware message authentication scheme for persistent key-value stores. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 363–380, 2022.

[13] Mihai Letia, Nuno Preguiça, and Marc Shapiro. CRDTs: Consistency without concurrency control.

[14] Nitesh Mor, Richard Pratt, Eric Allman, Kenneth Lutz, and John Kubiatowicz. Global data plane: A federated vision for secure data in edge computing. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1652–1663. IEEE.

[15] William Mullen. Capsuledb: A secure key-value store for the global data plane. 2022.

[16] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, jun 1996.

[17] Shweta Shinde, Dat Le, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB Linux Applications with SGX Enclaves. January 2017.

[18] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 645–658, USA, 2017. USENIX Association. event-place: Santa Clara, CA, USA.

[19] Lixia Zhang, Robert T. Braden, and Van Jacobson. TCP Extension for High-Speed Paths. RFC 1185, October 1990.

A Algorithms for Soft Sync and Flushing

In this section, we formally provide the pseudocode for Soft Sync and flushing protocols.

Algorithm 1 Soft Sync

```
function SoftSync
    freshWAL ← Fresh WAL records for each user from DataCapsule
    target_clock ← new VectorClock
    for record ∈ freshWAL do
        if record.sync_timestamp = global_clock.sync_timestamp then
            target_clock[record.writer] ← record.vector_clock[record.writer]
    while target_clock > global_clock.vector_clock do
        Update global_clock with each incoming write
    global_clock.sync_timestamp++
    Broadcast global_clock to all writers.
```

Algorithm 2 Flushing procedure

```
Global State: flush_timestamp: Number of Memtables flushed till now.
Parameters: c: skip-length, d: max-height
Input: mt: Memtable (a set of key-value pairs) to flush
function Flush(mt)
    SoftSync( )
    flush_timestamp++
    mt' ← {}
    last_logical_timestamp ← flush_timestamp − 1
    for i ← d to 1 do                                   ▷ Eager Compaction
        if c^i | flush_timestamp then
            for j ← flush_timestamp − c^i + 1 to flush_timestamp − 1 do
                Read j^th DataRecord, dr_j from cache.
                mt' ← mt' ∪ dr_j                        ▷ Order Sensitive
            last_logical_timestamp ← flush_timestamp − c^i
            break
    mt' ← mt' ∪ mt
    Sort(mt')
    dr ← new DataRecord from the key-value pairs of mt'
    mr ← new MetaRecord
    mr.index ← sorted keys from mt'
    mr.mtree ← new Merkle tree from key-value pairs in mt'
    mr.hint ← new BloomFilter from mr.index
    mr.hash ← Hash(dr)
    mr.last_logical_timstamp ← last_logical_timestamp
    mr.global_clock_snapshot ← global_clock
    Encrypt and Sign dr and mr and send it to DataCapsule servers.
```