CapsuleDB: A Secure Key-Value Store for the Global Data Plane

William Mullen

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-168
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-168.html

May 31, 2022

CapsuleDB: A Secure Key-Value Store for the Global Data Plane

by William Mullen

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor John D. Kubiatowicz
Research Advisor

May 19, 2022

(Date)

* * * * * * *

[illegible]

Professor Anthony D. Joseph
Second Reader

May 19, 2022

(Date)

CapsuleDB: A Secure Key-Value Store for the Global Data Plane

William Mullen
wmullen@berkeley.edu

Abstract

The Global Data Plane (GDP) is a data-centric network infrastructure that provides federated access to compute resources. However, an environment in which code runs on untrusted hardware introduces new security challenges for data management. Furthermore, DataCapsules, which are the high-level logical units of transfer that move through the GDP, can store persistent data but are difficult to search through efficiently, requiring users to understand their somewhat complex security structures for correct usage. CapsuleDB is the first database and key-value store designed for the GDP. It exports a simple interface for developers without jeopardizing any of the security guarantees inherent to DataCapsules. Furthermore, it uses a novel indexing system to track active data and naturally age out older data by leveraging the unique structure of DataCapsules. Finally, it uses Intel SGX to protect against malicious host operating systems, providing an extra layer of security for deployments to nodes owned by other entities.

CapsuleDB: A Secure Key-Value Store for the Global Data Plane

William Mullen
wmullen@berkeley.edu

1 Introduction

The explosion of wireless sensor devices, along with the ability to use cloud service,s has led to a rise in distributed computing and a new paradigm of computing called the Internet of Things (IoT). However, there are many privacy, security, scalability, durability, and latency concerns. In particular, protecting data from potentially untrusted cloud providers has emerged as a difficult problem [20]. A user may not want to send their personal data to a service provider, and the provider likely would not want to send proprietary algorithms to the user's local device. For many IoT devices, there may not even be enough computation resources locally to complete complex tasks. One way to solve this problem is to leverage existing edge computing resources in a distributed fashion. Unfortunately, this edge computing model presents several of its own challenges [16]. Rather than attempt to fix every problem, there are new paradigms that natively address these concerns. One such system is the Global Data Plane (GDP) [12], a data-centric system that enables secure management (transport and storage) of data among clients on both the edge and in the cloud. This federated infrastructure enables the seamless access and management of networked resources through a structure called a DataCapsule (described in Section 2 and Figure 1).

However, while the GDP introduces a comprehensive communications layer, its API can be complex to use, especially for non-streaming applications. The Paranoid Stateful Lambda (PSL) [5] project is a function-as-a-service (FaaS) framework that leverages the GDP's architecture to enable large-scale parallel, secure, and stateful computation among edge and cloud environments. PSLs use DataCapsules and Trusted Execution Environments (TEEs) [11] to provide integrity, confidentiality, and provenance to data. For a lambda to be stateful, it should be able to communicate with other lambdas to share and store communicated key-value pairs. These lambdas are also lightweight enough to run on edge devices in a secure environment while the GDP enables access to additional storage and computation resources. Using PSLs, functions can be run in a cloud environment without needing to trust the underlying hardware provider.

While the PSL project achieves our goal of secure computation on the edge, it does have several major limitations. Currently, key-value (KV) pairs are stored in-memory, in a data structure called a memtable, and in a DataCapsule, the only persistent storage in the system. As the size of the memtable grows, the memory footprint used by each enclave also increases, eventually hitting the memory bottleneck imposed by Intel SGX [11]. Intel SGX2 has dynamic memory allocation and so could delay the problem, but we would still eventually run out of space for large data sets. Thus, PSL has a scaling issue. Second, since PSL stores each KV pair directly into the DataCapsule, even if each lambda was not constrained by the size of the memtable it would be slow to read a value from the DataCapsule. This latency would only increase as the number of KV pairs increased, since the lambda may have to search the entire capsule to find a single KV pair. Finally, when an enclave crashes, PSL nodes must replay all the past records in the DataCapsule to reconstruct its memtable, as the system uses the DataCapsule like a write-ahead log. To address these issues, PSLs need a mechanism to quickly store and retrieve data from persistent storage without jeopardizing security.

In light of the new infrastructure paradigm proposed by the GDP, as well as the needs of an application such as PSLs, my research centered on a novel database designed as a first-class option for the GDP. This key-value store (KVS), called CapsuleDB, leverages a level-based structure to take advantage of the structure of DataCapsules. It exports a standard KVS interface with `put()` and `get()` functions to hide the complexities of interacting with the DataCapsule directly from developers. Originally conceived as a tightly integrated part of the PSL project to offset the aforementioned issues, CapsuleDB has since become a standalone system that targets general usage. Furthermore, it can be run in a trusted execution environment (TEE) to strongly guarantee the safety of signing keys and data (described further in Section 3.3).

In this paper I present the design and implementation of CapsuleDB. While it exports a standard KVS interface, the system itself is powered by a novel indexing and data management scheme that is DataCapsule native and takes advantage of a capsule's structure. The rest of this paper is organized as follows. Section 2 provides additional information on the GDP, PSL project, and log-structured merge-trees. Section 3 introduces the threat model for CapsuleDB as well as important security background. Section 4 introduces the design of CapsuleDB while Section 5 discusses the specifics of my implementation. Section 6 discusses benchmarking results and analysis. Section 7 describes several open areas of further development for CapsuleDB, and Section 8 concludes.

2 Background

To provide context for my work, I begin by discussing the GDP, DataCapsules, the PSL project, and log-structured merge-trees (LSM trees). Both the GDP and DataCapsules are critical components from a structural standpoint, and I assume their functionality is as described in the paper *Global Data Plane: A Federated Vision for Secure Data in Edge Computing* by Mor et al [12]. While the PSL project is not critical to understand my work, it does provide context for some of the original inspiration for CapsuleDB. Finally, CapsuleDB leverages the properties of LSM trees to provide its core functionality.

2.1 Global Data Plane

The GDP [12] is a federated data-centric infrastructure network. There is no central authority. Instead, GDP servers route requests to specific services and clients using GDP names as identifiers. The high-level logical unit of data in the GDP is a structure called a DataCapsule, a format-

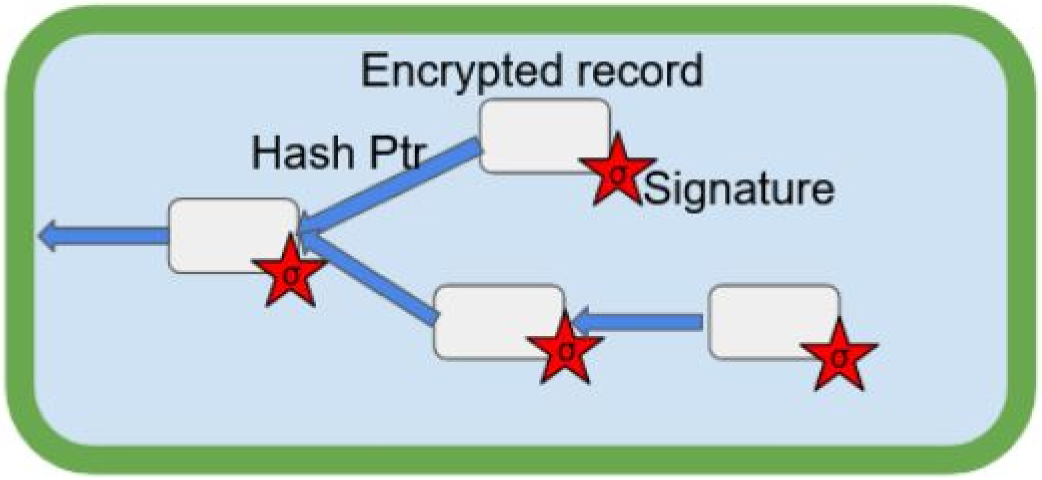


Figure 1: A basic DataCapsule with hash pointers and signatures.

agnostic and cryptographically secure container for data. See the next Section for a more detailed description of DataCapsules. These capsules are replicated and distributed throughout the GDP, acting as storage, communication channels, transaction records, and entity identifiers. Together with the GDP's infrastructure, they provide a robust networking architecture, able to connect computing resources regardless of physical location. Because each capsule follows a standardized metadata format, they are universally routable and manageable, regardless of the actual data contained within them. Through additional locality optimizations, clients are also able to satisfy additional performance requirements by pulling DataCapsule replicas close to them. This paper will not discuss details of or improvements to the GDP's implementation.

2.2 DataCapsules

A DataCapsule is a cryptographically hardened, append-only data structure that is globally addressable in the GDP and acts as the atomic unit of data transfer. Each capsule is made up of a metadata wrapper and a series of records. In their most basic form, these records contain arbitrary data, a hash over the data, and potentially a series of hash pointers to older records. Each record must also specifically contain a signed hash pointer of the previous record, thus creating a Merkle tree as shown in Figure 1. The first record in the capsule is a special metadata record that uniquely defines the DataCapsule itself. It contains ownership information, a public key for signing records, and other application specific information. The GDP name of this capsule is defined as the hash of this metadata record. In turn, each record in the capsule can also be uniquely identified by its hash. DataCapsules are also extremely flexible in the type and amount of data each can carry, constrained only by the available resources.

The structure of DataCapsules enforce several powerful security guarantees. Records are immutable, well-ordered, and cryptographically signed. In addition, they contain hashes over their data as well as hash pointers to previous records. Thus, it is straightforward to quickly verify the integrity of records and the overall capsule. This structure also establishes a chain of provenance, storing the data's history for the life of the capsule. For use cases that require even stricter security, proofs can be returned that verify a DataCapsule's validity. Since each record is data agnostic, it is also straightforward to encrypt data, providing confidentiality without disrupting the capsule's structure. Finally, because DataCapsules have the owner's public key embedded into the metadata, only authorized users with the correct key are able to write to it. Thus, the structure of DataCapsules provides strong integrity, confidentiality, authenticity, and provenance guarantees without needing to rely on third-party infrastructure.

In the GDP, DataCapsule management is handled by a series of DataCapsule servers owned by infrastructure providers. These servers handle securely appending new data to DataCapsules as well as retrieving specific records by their hash. At the time of writing, DataCapsule Server development is a separate ongoing project, and I will not be discussing it in-depth, as it is orthogonal to my work. I assume there is an interface available that allows for new data to be appended to specific capsules and for old data to be retrieved by the hash of a record.

2.3 PSL

The Paranoid Stateful Lambda project (PSL) is an implementation of a function-as-a-service (FaaS) framework similar to AWS Lambda [1] and Google Cloud Functions [2], but built on the GDP. However, in contrast to the aforementioned services, PSLs can maintain state between remote nodes by using CapsuleDB as shared persistent storage. Each lambda node also contains an eventually consistent in-memory KVS called a memtable. These memtables provide a cache to enable fast retrieval of recently used data. Through a system called the Secure Concurrency Layer (SCL), workers can also multicast new KV pairs to other workers and to CapsuleDB.

However, PSLs may be operating on sensitive data. For example, in an AI use case, a user may not want their data uploaded to an third-party's server, and the service provider may not want to put their proprietary model directly onto a user device for fear of intellectual property loss. To overcome this problem, PSL executes almost entirely in TEEs, specifically Intel SGX [11] enclaves. These TEEs allow for all parties to verify the enclave and the code inside is being executed as expected through an attestation process. Furthermore, enclaves protect the data they contain from the host OS and applications. As a result, PSLs can safely run on the edge or in the cloud, regardless of who owns the underlying hardware, without leaking any information. The only portion of the system that is not inside the enclave is the DataCapsule itself, as its structure natively protects its data as discussed above.

The SCL provides critical synchronicity functionality while also connecting the various components of the PSL system together. Using the SCL, PSL nodes can securely share new KV pairs and synchronize their memtables using special end-of-epoch messages. The SCL also hosts a coordinating enclave and a key-management enclave. The coordinating enclave takes requests from clients and manages the overall execution of the user's code. The key-management enclave distributes identity information via security keys to other entities on the SCL and manages the attestation process. With these components, PSLs provides a robust and secure FaaS framework that has a wide range of applications.

As mentioned in the Introduction (Section 1), CapsuleDB was first conceived as an answer to several limitations of PSLs. Although CapsuleDB has now developed into its own standalone project, there are three aspects of the database that are built with PSL compatibility in mind. First, CapsuleDB ingests `kvs_payloads` to extract the relevant data for storage. This structure is the exact same as the structure that is transferred between lambdas in the PSL project, consisting of a key, a value, a timestamp, and a message type, though CapsuleDB only stores the KV pair and its associated timestamp. Second, CapsuleDB implements the same memtable found in the PSL project. The memtable is an in-memory data structure that holds the most recent `kvs_payloads`, implemented as a hash table. Finally, CapsuleDB uses ZeroMQ [3] for networking and socket management. This is partially a legacy decision, since the SCL uses ZeroMQ for its networking and I wanted CapsuleDB to be compatible with the SCL. However, ZeroMQ's cross-language capabilities and simple interface have provided new benefits that make it useful to CapsuleDB even outside the context of PSLs. In my work, I mostly used the basic ZeroMQ socket functionality to listen for string based messages. These components are discussed further in Sections 4 and 5.

2.4 Log-Structured Merge-Trees

A log-structured merge-tree (LSM tree) [14] or level tree is a data structure designed to quickly serve the most recent version of data while also preserving older versions. In its most basic form for KVS's, data is split into different levels labelled as level 0 (L0), level 1 (L1), level 2 (L2), and so on. Each level is larger than the previous, with L0 being the smallest and often kept in memory. New data is first stored in L0. Once L0 is full, a process called compaction is triggered which moves data from L0 into L1. Any overlapping keys in L1 are replaced with the fresher data incoming from L0. This process then repeats down the tree until no levels are full or have overlapping keys. LSM trees are beneficial because they naturally sort data into more and less recently used groups. Fresher keys are kept at the top in L0 and can be quickly returned, whereas less often used keys are stored in larger levels that take longer to search.

I chose LSM trees as the basis for the design of CapsuleDB because they are uniquely suited to match well with DataCapsules. In both structures, data is never deleted. Furthermore, data can be easily partitioned into levels through the use of hash pointers. When new data is added to the DataCapsule, in addition to including a hash pointer to the previous record, a second hash pointer can be used to link the new content to the last record of a level, regardless of that record's location in the chain. See Figure 2 for an example. These attributes make the LSM tree an excellent foundation for CapsuleDB. A brief note on notation: various sources have discussed LSM trees using different terminology for the same design. In this paper, I consider "lower levels" to have higher numbers. Thus, L2 is lower than L1 and L0 is higher than L1.

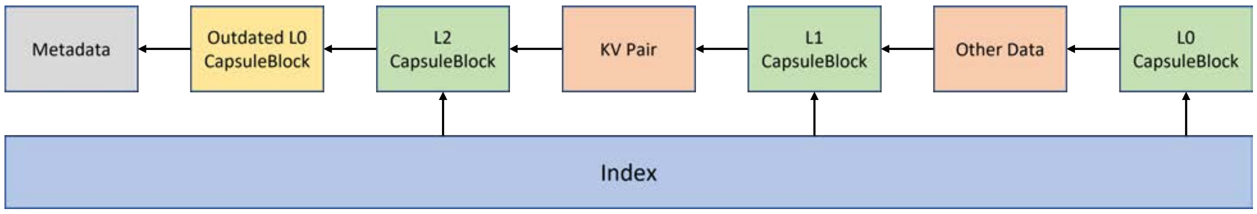


Figure 2: CapsuleDB tracks specific records in the DataCapsule using its index, making it easy to find where data is regardless of its age. The underlying Capsule naturally keeps fresh data towards the front of the chain. See Section 4 for an explanation of terminology.

3 Security

CapsuleDB is designed to provide data security through the use of DataCapsules without substantially impacting overall performance. While the current implementation can be run as a standalone application, CapsuleDB provides the strongest security guarantees when used in conjunction with the GDP. In this Section, I will discuss some background before presenting the threat model for CapsuleDB.

3.1 Security Features

CapsuleDB leverages SHA-256 hashes, elliptic curve digital signature algorithm (ECDSA) signatures, and authenticated encryption with associated data (AEAD) symmetric encryption to provide integrity, authenticity, and confidentiality to stored data. When data is stored in CapsuleDB, it is hashed before being written to permanent storage. When the data is later retrieved, the data is rehashed to verify it has not been modified. A signature is also taken over the hash that verifies it is the correct hash and approves the data to be appended to a DataCapsule. Finally, the AEAD encryption simultaneously verifies and encrypts the data, making attacks designed to read the data at rest impossible. See Figure 6 for a diagram of the layers of security.

3.2 Trusted Execution Environments

While these security features provide powerful guarantees, they rely on a trustworthy hardware operator. In on-premises installations, the user is likely to also own the hardware. However, in modern cloud computing and in the model presented by the GDP, the user likely does not own the hardware on which CapsuleDB is running. In these scenarios, users may not feel comfortable providing security keys to CapsuleDB as they have no guarantees of the database's operation or the security of the data it contains. Like the PSL project, CapsuleDB overcomes this limitation through a Trusted Execution Environment (TEE). As defined by Sabt, Achemlal, and Bouabdallah, a TEE is a "tamper-resisted processing environment that guarantees the authenticity of executed code, the integrity of runtime states, and the confidentiality of its data stored in persistent memory" [15] with an attestation process for trustworthiness verification by third parties. For CapsuleDB, this means it can be protected from a malicious host OS when run within a TEE. Thus, keys and stored data are protected as a malicious actor would not be able to extract them without detectable modifications to the underlying hardware. Furthermore, users can be additionally sure of the TEE's authenticity through the attestation mechanism, and can verify both the TEE and CapsuleDB are running as intended. As discussed in more depth in Section 4.7, CapsuleDB uses Intel SGX enclaves [11] and Asylo [8] as its TEE.

3.3 Threat Model

With this background, I now present the threat model for CapsuleDB. CapsuleDB's signing keys are the root of trust, as the signatures verify the correctness of the hashes which in turn verify the correctness of the encrypted data. Furthermore, the signing key allows CapsuleDB to append new data to the DataCapsule. As such, CapsuleDB needs to protect this key from all potential attackers. CapsuleDB's threat model assumes a malicious host OS, and only assumes that a correct TEE attestation mechanism is available. To protect against a malicious host modifying the CapsuleDB implementation, the TEE will provide an attestation report of CapsuleDB as well as its own code. The user can then verify the information is correct and that CapsuleDB is operating as expected. Since TEEs protect the data they contain from the host OS, a user can now instantiate and safely use a signing key within CapsuleDB, even in the presence of an attacker. If a party were to modify any data, an integrity violation would be detected by recalculating and comparing against the stored hashes. Even if an attacker were able to append false data to the DataCapsule, they would not be able to correctly generate the corresponding signature and thus their data would be ignored. The worst attack an actor could launch is a denial of service attack by attempting to breach the TEE, causing an integrity violation and forcing CapsuleDB to terminate for safety. Given that cloud and GDP infrastructure providers would want to retain customers, a denial of service attack would be unlikely as it would drive users away from their service. Thus, I do not consider it a vulnerability CapsuleDB needs to address.

By these descriptions, the only persistent state is stored in the DataCapsules. As later sections discuss, there are optimizations that allow for fast startup, either normally or after a crash, but the role of CapsuleDB is mainly to act as a trusted interface for a DataCapsule. As such, the user provides a signing key that CapsuleDB uses and trusts that the security optimizations embedded into the system, namely the signatures, are also trusted, so the underlying DataCapsule does not need to be queried. Therefore, a malicious entity who controls either CapsuleDB or the signing key can effectively gain full control over the stored KV pairs in the DataCapsule, since a user would have no reason to check the underlying log (the DataCapsule). As such, the signing key must not be allowed to leak.

4 Design

CapsuleDB is a key-value store inspired by LSM trees but backed by DataCapsules and the GDP. It is built to specifically take advantage of the properties of DataCapsules while exporting a more con-

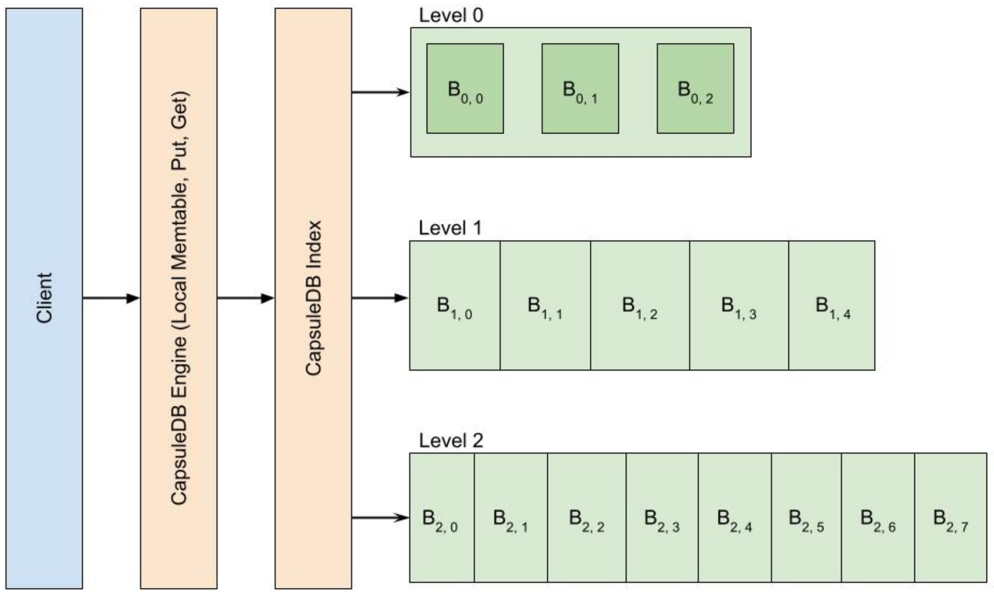


Figure 3: The structure of CapsuleDB. The index contains block ranges and hashes that help quickly retrieve blocks from the DataCapsule which may contain the requested key.

venient programming interface. CapsuleDB abstracts away the complexities of interacting directly with a DataCapsule by tracking the location of active data. As the database expands, CapsuleDB limits its own growth locally, storing only the minimum amount of data necessary for operation and reducing the need for high-powered hardware.

CapsuleDB is broken into three core components, the engine, an index, and CapsuleBlocks. Clients submit PUT and GET requests to the engine. The engine then processes the requests and dispatches the relevant data to the index. For PUTs, the index simply adds the new payload to the current memtable, potentially triggering a compaction. For GET requests, the index checks the key and references its internal list of DataCapsule record hashes to find the matching CapsuleBlock. The CapsuleBlock is then retrieved from the DataCapsule and the data returned to the engine and the client. These interactions are shown in Fig 4.

4.1 Level-Based Design

CapsuleDB stores KV tuples of key, value, and timestamp in increasingly larger levels. As new data is added to the database, older data gets pushed to lower levels. When a memtable is filled, it is marked read-only and written to L0. This level is small enough that it can be stored in memory. Over time, L0 will expand beyond its maximum size, prompting a compaction (described below in Section 4.5) which moves data into lower levels. If the same key appears in that lower level, the newer value overwrites the older one as determined by the timestamp attached to the payloads. If the compaction causes a level to fill, then data is compacted down into even lower levels. These lower levels are substantially larger than L0, but are sorted. Thus, as CapsuleDB grows, searches remain fast. Given the access pattern findings by Boissier et al in [4], having smaller levels for more recent data improves latency for the bulk of GET requests, since most are for recent data.

Analyzing the runtime, each key can only appear once in a level by definition. Therefore, the worse case number of searches for a key is the number of keys in L0 + the number of levels. This is because keys in L0 are unsorted, and thus must be linearly searched. As shown later in Section 6, the compaction process can be costly. Therefore, to keep accesses to recent data fast and given that L0 is relatively small, it is kept unsorted. Note that a payload could be in any level, making accesses to older data more costly as additional DataCapsule records would need to be retrieved to determine whether the requested key was present. However, it is still substantially faster than searching through every record in the underlying DataCapsule. Thus, the level structure of CapsuleDB substantially accelerates data management by tracking which DataCapsule records to access directly while simultaneously keeping common access patterns fast.

4.2 Engine Interface

CapsuleDB exports a simple interface for developers already familiar with other KVS's. `put()` accepts a `kvs_payload` to be stored in the database. `get()` takes a key and returns the associated value and timestamp of when the pair was originally inserted. This interface abstracts the CapsuleDB storage process and DataCapsule operations from clients. When used in an enclave, the same interface is used, though the requests are first processed by a translation layer. When used

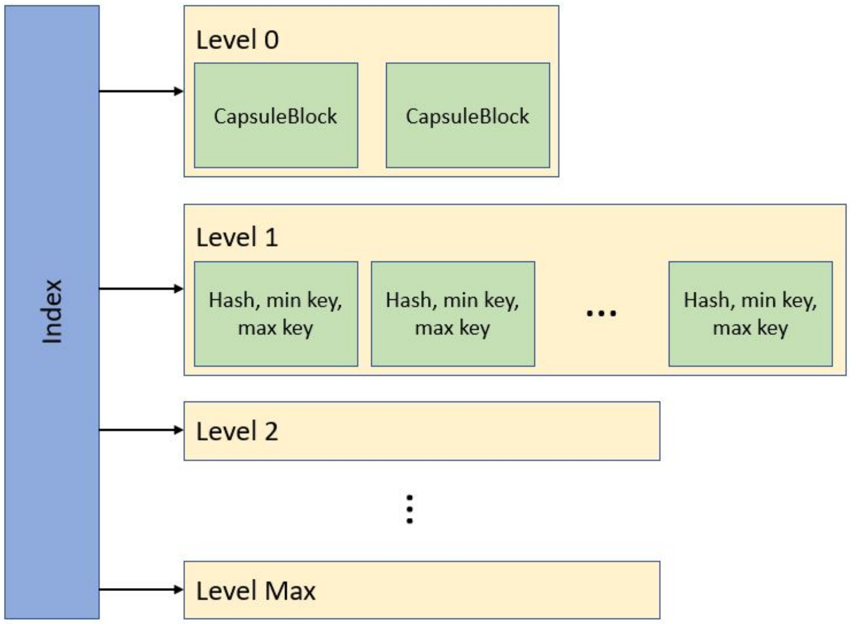


Figure 4: A more detailed view of the data stored in the index. L0 stores pointers to CapsuleBlocks while lower levels store the hash of the DataCapsule record that contains the block.

as a server, `PUT` and `GET` requests are translated into the same function calls. This process as well as the enclave subsystem are described below in Sections 4.7 and 5.6.

4.3 CapsuleBlocks

CapsuleBlocks are the basic unit of storage in CapsuleDB. Each one contains a total number of KV tuples set by the user. This value determines both the size of the CapsuleBlocks and, by extension, the size of the memtable. At L0, these CapsuleBlocks are read-only versions of old memtables. At lower levels, they are runs of tuples sorted by key. Thus, all the entries in L1 form a sorted run from the level's minimum to maximum key, but are stored across multiple CapsuleBlocks. Lower levels reflect this same pattern, though are larger. Each CapsuleBlock stores what level it is associated with, a list of payloads, and the minimum and maximum keys it contains.

From a DataCapsule perspective, each CapsuleBlock is stored as a single record. Thus, whenever a value is requested, an entire CapsuleBlock in retrieved from the DataCapsule. Similarly, during the compaction process described in Section 4.5, CapsuleDB will write out full CapsuleBlocks rather than individual tuples.

4.4 Index

The index is the core mechanism of CapsuleDB and serves several critical roles. Primarily, it tracks the hashes of active CapsuleBlocks to quickly lookup keys. It does so by organizing blocks into levels as shown in Figure 4. Each level beyond L0 sorts all of its contained keys in ascending order. This run of keys is broken into chunks and stored in CapsuleBlocks, which the index tracks via the hash of the associated DataCapsule record. L0 simply contains immutable memtable copies, and thus must be searched linearly. However, it is kept relatively small for this specific reason. Whenever a block is added, removed, or modified, the hash list is updated in the index. When compaction occurs, the index updates which levels the moved CapsuleBlocks are now associated with. In this way, CapsuleDB can always quickly find the most recently updated CapsuleBlock that may have a requested key.

At first glance, it may seem strange that the list of hashes must be updated for any operation that modifies a CapsuleBlock. However, recall that DataCapsules are an append-only data structure with immutable records. If a CapsuleBlock needs to be modified, a new record containing the block must be appended to the end of the chain. This is the second major function of the index, it determines which blocks have active data. Currently, nothing can ever be deleted from a DataCapsule, so old data that no longer holds the correct KVS state is still present. By updating the record hashes in the index, CapsuleDB effectively removes any references to those old blocks. Although still technically accessible by scanning through the DataCapsule, a user does not have to worry about accidentally retrieving stale data. The index acts as an effective source of truth on the most up to date CapsuleBlocks, and by extension the KV tuples they contain, to guarantee that the values returned are fresh.

4.5 Compaction

Compaction is critical to managing the data in any level-based system and is the process by which data is moved down and merged into lower levels. CapsuleDB's compaction process uses key insights from the flush-then-compact strategy of SplinterDB [6] to limit write amplification. Recall each level has a maximum size; a level is full once the summed sizes of the CapsuleBlocks in that level meet or exceed the level's maximum size. This triggers a compaction.

All writes to CapsuleDB are first stored in the in-memory memtable. Once the memtable fills, it is marked as immutable and written to L0 as a CapsuleBlock, also simultaneously appending the block as a record to the DataCapsule. Once L0 is full, compaction begins by sorting the keys in L0. They are then inserted into the correct locations in L1 such that after all the keys are inserted, the level is still a monotonically increasing run of tuples sorted by key. In the event of a collision, the timestamps for the tuples are compared, and whichever is newer is retained. Finally, the CapsuleBlocks and their corresponding hashes are written out to the DataCapsule and updated in the index, marking the end of compaction.

Now consider a slightly more complex situation. When compaction starts after L0 is full, CapsuleDB also evaluates the size of L1. If the the size of L0 and L1 together would exceed the maximum size of L1, then CapsuleDB knows L1 also needs to be compacted into L2. Now, rather

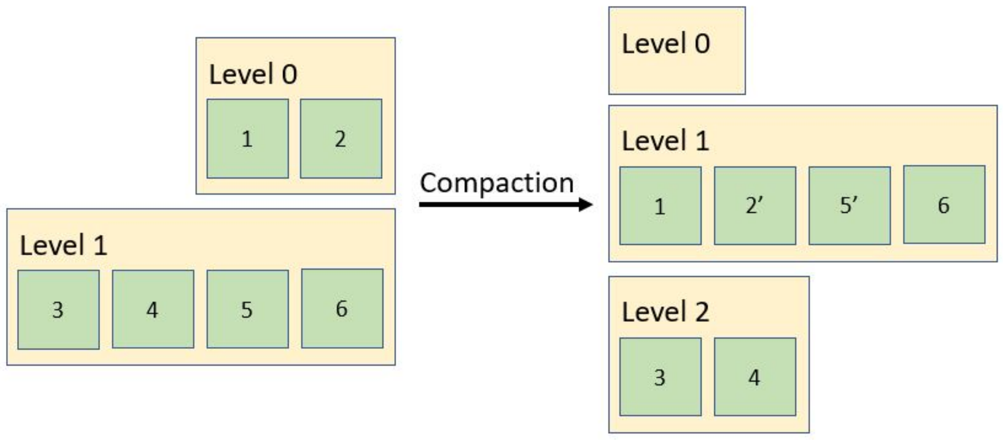


Figure 5: An example of compaction. In this case, L0 reached its maximum size and is compacted into L1. However, doing so causes L1 to fill, and so blocks 3 and 4 are flushed to L2 since they overlapped with the incoming block 1. Finally, keys in block 2 overlapped with those in 5, and so two new blocks were created from those keys: blocks 2' and 5'.

than flushing and compacting L0 into L1 and then the new L1 into L2, the database first determines which blocks in L1 would be affected by the incoming L0 blocks. These blocks are then flushed from L1 and compacted into L2. Next, the new blocks from L0 would be flushed and compacted into the now smaller L1. Finally, the index is updated to reflect the changes made to the levels. If L2 would fill up as a result of L1's compaction, the same process would occur between L2 and L3, and so on. An example is presented in Figure 5.

Through this strategy, each KV tuple is guaranteed to only move once. This reduces write amplification, defined as the degree to which data is repeatedly written. In this case, CapsuleDB ensures each tuple is written only once, rather than potentially being written at one level, then needing to be compacted down into the next level, and so on. This is critical since any modification to a CapsuleBlock would result in a new record being appended to the DataCapsule, a relatively slow process. If a level was fully compacted and then had to potentially edit those CapsuleBlocks again to reflect keys being moved to lower levels, it would substantially inflate the number of appends and cryptographic function calls needed.

4.6 Storage

When CapsuleBlocks need to be written to storage, they are first serialized to a string using Protocol Buffers (protobufs) [9], a cross-language library from developers at Google. A hash is then taken over that string, and is signed. This data is then sent to the DataCapsule server for storage. The server returns the hash of the newly appended record which CapsuleDB stores in its index.

During retrieval, the hash stored in the index is used to retrieve the block from the DataCapsule server. In both cases, the retrieved block's signature and hash are verified to prevent tampering. After deserialization, the relevant data can be retrieved from the block.

4.7 Enclaves

So far, CapsuleDB's security guarantees rely on its code not being tampered with and a trustworthy OS. However, in the vision of computing described by the GDP, users may not know where their CapsuleDB instance is running. Today, we see this phenomena with cloud computing vendors, though most would consider the current large vendors trustworthy. However, TEEs can be used to ensure that code is executed as expected and that malicious OS's cannot affect the database's operation. To access the powerful security model provided by TEEs, CapsuleDB uses Asylo [8] for enclave development, a library that can package the same code for a variety of hardware TEE implementations. The enclaved version of CapsuleDB function almost identically to the non-enclave version with only one difference. First, rather than client requests going directly to the CapsuleDB engine, they instead first go to the enclave interface. Here, the client requests are encoded into a protobuf message. Asylo then uses the protobuf to securely transfer the data into the enclave. From there, the request is decoded and sent to the engine as normal. On a GET request, the process is done in reverse, again using a protobuf message to cross the enclave boundary. With this added layer of security, CapsuleDB can be attested as running correctly while keeping all the information it contains, including security keys, safe from outside actors, including the OS itself.

However, introducing an enclave gives users a trade-off. Unfortunately, to preserve security, crossing the enclave boundary is extremely expensive due to the encryption and decryption of memory pages as well as register data getting flushed (see [18] for an example of the impact on performance). Furthermore, in older implementations, available memory may be severely limited. To mitigate this problem, CapsuleDB's core code is purposely kept small and lightweight to fit within the enclave. However, using an enclave provides critical benefits. Most importantly, users can store encryption keys and be confident that the data produced from within the enclave has not been tampered with. As such, an encrypted CapsuleBlock, its hash, and the signature over that hash must also all be correct. All this is possible without needing to trust the host machine or needing to go directly to the DataCapsule, making the enclaved version of CapsuleDB a compelling option for cloud and GDP deployments where security is a major concern, as the user does not control the underlying hardware in those instances.

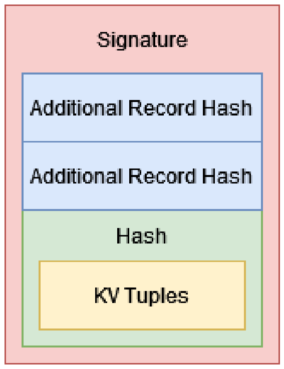


Figure 6: The layers of security used when storing a CapsuleBlock.

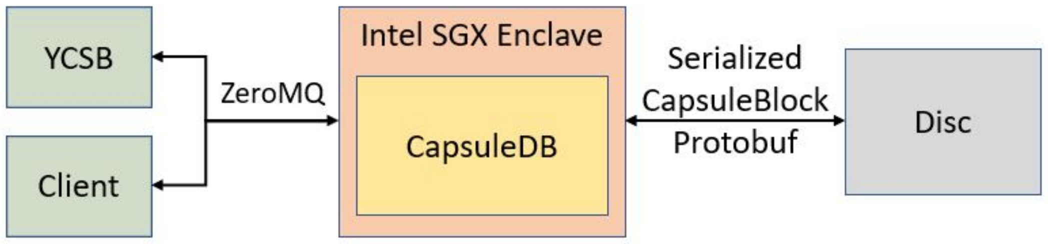


Figure 7: The CapsuleDB pipeline. Note that requests must pass through the enclave barrier to reach CapsuleDB.

Note that in all cases, CapsuleDB must hold a private signing key that allows it to write into a DataCapsule or to disk. As shown in Figure 6, this signing key forms the root of trust for the blocks CapsuleDB writes. This signing key cannot leak under any circumstances, as it would allow an attacker to arbitrarily modify CapsuleDB's state by manipulating DataCapsule records. As such, the enclave is a critical portion of CapsuleDB to protect against the malicious host threat model discussed in Section 3.3. Without it, there is a much higher risk of leakage, especially when CapsuleDB is deployed to non-owned hardware.

5 Implementation

CapsuleDB is approximately 10,700 lines of C++ and leverages the existing memtable implementation and `kvs_payload` structure from the PSL project. Furthermore, I used the same Docker container as the PSL project to replicate the operating environment. All hashing uses SHA-256. Currently, CapsuleDB only supports Intel SGX enclaves for the enclave back-end, partially due to limitations with Asylo and partially because only Intel SGX hardware is available. As additional hardware backends are made available through Asylo, CapsuleDB should be compatible with no issues.

My implementation diverges from the above design in two ways. First, due to ongoing development with the GDP and DataCapsule servers, my implementation stores all data to a local disk rather than a DataCapsule. Even if I had used the GDP, a local disk could be used as a cache for DataCapsule records to avoid making repeated network requests. Furthermore, by design the GDP acts as networked storage, and so saving data to local disk emulates a fast version of the GDP.

Second, my standalone implementation does not sign records. In the PSL project, there is an embedded version of CapsuleDB that does sign records but is missing several features of the standalone version. This is because the SCL provides a built-in key distribution mechanism that the standalone version does not have, and not implementing signatures accelerated development substantially. However, the standalone implementation does also provide important insights into the core mechanisms of CapsuleDB while also being usable when the host OS is trusted. For example, in on-premises server farms, users can trust the hardware because they own it. As such, the strict security guarantees signatures provide may be unnecessary for some use cases. This implementation does not have signatures, but future versions will likely use them by default with users able to toggle them off if desired. The full pipeline as implemented is shown in Figure 7.

5.1 Blocks and Levels

CapsuleBlocks are the base unit of data transfer for CapsuleDB. Within each block, KV tuples are stored in a vector. The block also stores which level it is associated with, as well as the minimum and maximum keys it contains. The maximum size of the memtable matches the maximum size of the CapsuleBlocks.

Each level follows a similar structure, storing the level number they represent as well as a vector of block headers, rather than the CapsuleBlocks themselves. Each block header contains the hash of a DataCapsule record that corresponds to a CapsuleBlock, as well as the minimum and maximum key of the block. This way, CapsuleDB can reduce the amount of data it needs to store without impacting performance or correctness. This is also a critical memory management technique, as storing every CapsuleBlock in memory would be prohibitive in older TEE implementations. Each level also stores the minimum and maximum key in the level to accelerate look-ups. Finally, the level contains the number of blocks currently in the level and the level's maximum size. By default, each level in CapsuleDB is ten times larger than the previous level.

5.2 Index

The index follows a similar design to the levels and CapsuleBlocks. It contains a vector of levels, which corresponds to the active levels in CapsuleDB. The index also contains the number of levels and the size of the CapsuleBlocks. Note that any time a block is modified, added, or deleted, the corresponding entry in a level must be updated to reflect the newly appended record's hash in the DataCapsule. This is because records that have already been added to the DataCapsule cannot be modified, so a new block must instead be used. While the index does handle searching through the levels and compaction, each individual level is responsible for the direct tracking of each CapsuleBlock. Thus, the index can still serve requests to other levels while a different level is applying updates to its list of associated hashes.

5.3 Compaction

After a new memtable is written to L0, CapsuleDB always checks whether this write causes L0 to fill and require a compaction. If a compaction is needed and would not cause L1 to fill, the database performs a simple merge sort to combine the two runs of blocks. If it would cause L1 to fill completely, blocks that would be affected by the merge are first flushed from L1 into L2. This is done by evaluating minimum and maximum keys to determine which blocks overlap. L1 and L2 are then merged using the same merge sort implementation as before. Then, since space is now available in L1, the original merge is performed. This process could recursively occur several times at L2 and below before space is finally available for the L0 – L1 merge. Once all newly created CapsuleBlocks have been written and hashed, the index is updated to reflect the new active blocks.

As discussed in Section 4.1, CapsuleDB keeps higher levels small to reduce the amount of time needed to find the correct CapsuleBlock for recent data. Other level databases, such as RocksDB [17] or SplinterDB [6], provide similar arguments for keeping upper levels small, though both have different storage mechanisms. Interestingly, only RocksDB chooses to keep L0 unsorted. The authors found taking the time to sort and organize L0 did not provide a substantial enough performance benefit over a simple linear scan to warrant the processing time. Given this conclusion, CapsuleDB follows Rocks' design choice to keep L0 unsorted.

5.4 Storage

CapsuleDB uses Protocol Buffers (protobufs) [9], a cross-language library from developers at Google, to serialize CapsuleBlocks for storage. Each block is first serialized to a string before being hashed using SHA-256. By default, the string version of the block is then written to disc, with the hash assigned as the name for the file. When the block is later retrieved for a read, the hash is checked for correctness. In this way, CapsuleDB can detect any tampering with the block on disk by a third party. The worst an attacker could do is a denial of service of attack by changing the hash the block is stored under, as it would no longer match what was stored in the index.

In the embedded version implemented in the PSL project, the serialization process is slightly more involved. Like the local storage version, the CapsuleBlock is first serialized and hashed. However, DataCapsules require additional security features. Thus, the data that is hashed is actually an encrypted version of the serialized block created using AEAD encryption. A ECDSA signature is then taken over the hash of the encrypted data. This model was also developed without the DataCapsule server, and so is simply stored in an emulated DataCapsule.

5.5 Managing Blocks

In general, CapsuleBlocks are stored as records in the DataCapsule or locally on disk. However, there are a few notable exceptions. The entirety of L0 is kept in memory, due to its small size and relatively high rate of access. CapsuleBlocks in lower levels may also be present in memory because they recently served a GET request. In this case, the block is not evicted from memory until the space is needed due to the limitations imposed by SGX. If a later GET request requires a key in the same block, a costly DataCapsule server or disk access can be avoided. After reading in so many blocks that the enclave runs out of memory, the least recently used block is evicted to make space for the new incoming block. The block is also evicted if it is no longer considered active by the index.

Due to this method of managing the CapsuleBlocks, the key retrieval process requires a slight adjustment. When the hash of the block that may contain the requested key is determined, the cache is first checked to see whether the block is already present in memory. If so, the cached copy can be accessed directly as any updates would cause it to be evicted.

5.6 Networking

While the design described would use the GDP for all networking, the routing infrastructure is currently under development. Therefore, CapsuleDB currently uses ZeroMQ [3] to listen on a socket for incoming PUT and GET requests. ZeroMQ is a cross-language networking library that exports the same interface in all languages while hiding the complex connection management. During operation, CapsuleDB runs in dealer mode on a ZeroMQ socket. Since ZeroMQ supports sending strings, protobufs are used again to encode requests into a `DBRequest` message. CapsuleDB's networking module first deserializes an incoming message to determine whether it is a PUT or GET request. If it is a PUT, the module simply inserts the included payload into the database and begins to processes the next request. If it is a GET request, the module will retrieve the payload using the included key. See the Appendix for a discussion of parallelism. When found, the module will then create a new `DBRequest` message with the payload attached and send it back to the user client. The same basic structure is used with the enclaved version of CapsuleDB as well, though requests are routed through the enclave interface rather than being sent directly to the CapsuleDB Engine. In Section 6, I discuss using the Yahoo! Cloud Services Benchmark (YCSB) [7] for evaluating CapsuleDB's performance. Here, a benefit of ZeroMQ is that YCSB's benchmarking code (implemented in Java) can be easily connected to CapsuleDB (written in C++) without needing complex translation layers.

5.7 Writing to CapsuleDB

Writes to CapsuleDB are formatted as a PSL `kvs_payload`, containing the key, the value, a timestamp, and the message type. After the payload is received, the message type is dropped and the remaining data is stored. All key and value entries are strings. CapsuleDB utilizes the same memtable implementation as PSL, only modifying it to check if it is full and needs to be written to a CapsuleBlock. This includes the locking mechanism used by PSL to ensure correct concurrent behavior. As in the PSL memtable, any incoming payloads with the same key as existing payloads but with an earlier timestamp are also automatically discarded. If the memtable is full, it is written to a CapsuleBlock and a compaction check occurs. At the same time, a new memtable is spawned and begins servicing incoming writes.

5.8 Reading from CapsuleDB

CapsuleDB takes in a key that corresponds to the desired value. First, the current memtable is searched, again utilizing a lock for correct concurrent operations. If not present, the key goes to the index to sequentially check each level, terminating early if the key is found. In levels other than L0, the index evaluates which block to pull by checking the requested key against the maximum and minimum keys contained in each block. If found, the corresponding payload is returned. Otherwise, CapsuleDB informs the requester that the key is not present.

Note that when used with the PSL project, CapsuleDB may not yet have the most up to date values as they could still be in transit when the read request is processed. However, they will

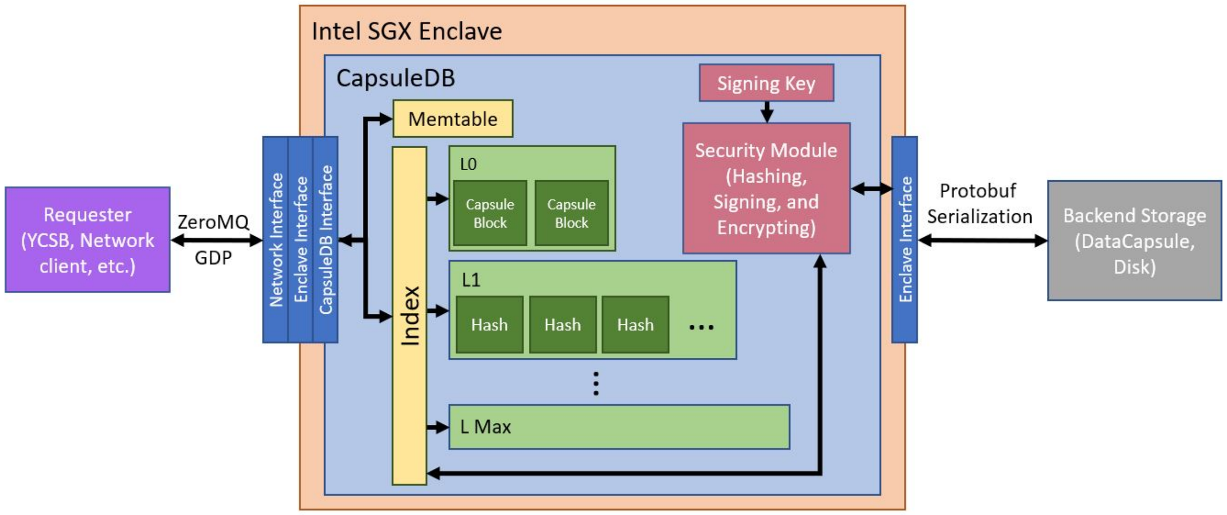


Figure 8: A diagram of the full CapsuleDB system.

eventually be delivered by the SCL. As such, CapsuleDB only guarantees the same loose consistency model proposed by the PSL project.

5.9 Full System Diagram

I end this Section with an overview of CapsuleDB's core components. As shown in Figure 8, CapsuleDB first takes in a request from a requesting entity, either through ZeroMQ or the GDP. It is processed by the networking interface, then the enclave interface, and finally the CapsuleDB interface. From there, the request is dispatched to the memtable and index. Writes are written directly to the memtable. If it is a read, each level is read sequentially to find the relevant block after checking the memtable. In the event the key is found in L1 or below, the index retrieves the associated hash and sends it to the security module. The security module requests a CapsuleBlock from storage using the hash (from either a DataCapsule or local disk) and performs all security checks to verify the block has not been illegally modified. If a block is instead being written out, the security module hashes and signs the block using the user provided signing key. In both cases, protobufs are used to serialize and deserialize the data. All components within the Intel SGX Enclave in Figure 8 are in memory.

5.10 Note on Intel SGX Enclave Vulnerabilities

I am aware of recent literature that describes major vulnerabilities in the underlying implementation of SGX [13, 19, 10]. While these issues are concerning, I specifically chose to use Asylo because of its portability features. While current implementations of SGX may be insecure, future versions of

SGX and other TEEs will likely not exhibit the same problems. With Asylo, CapsuleDB should be easy to deploy on those platforms with little to no modification. Given Asylo's portability, and like many other projects that rely on SGX enclaves, I consider these security issues to be orthogonal to my work.

6 Evaluation

CapsuleDB's performance is evaluated using the Yahoo! Cloud Serving Benchmark (YCSB) [7] core workloads. Each workload follows a specific access pattern. Workload A is 50:50 reads:writes whereas Workload B is 95:5 reads:writes. Workload C is 100% reads, and Workload D is reads of the most recently inserted records. Finally, Workload F is a read-modify-write pattern. Note that Workload E is skipped as CapsuleDB does not support range scans. These workloads are run against both the enclaved and non-enclaved versions of CapsuleDB. As discussed earlier, Workload D with its reads of recent data will likely be the most common access pattern, though the other workloads also provide insights into CapsuleDB's performance on reads and writes.

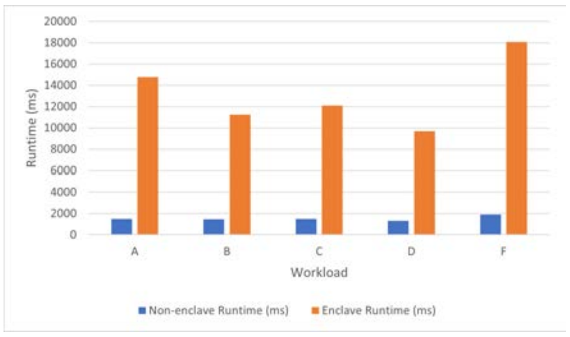
6.1 Experimental Setup

CapsuleDB is evaluated on a Intel NUC 7PJYH, equipped with Intel(R) Pentium(R) Silver J5005 CPU @ 1.50GHz with 4 physical cores (4 logical threads). The processor has a 96K L1 data cache, a 4MiB L2 cache, and 16GB of memory. The machine runs Ubuntu 18.04.5 LTS 64bit. The machine is equipped with Intel SGX2 and runs Asylo version 0.6.2. In each test (except for the CapsuleBlock size test), YCSB is run with default settings and a memtable size of 50 entries. Note that this version of CapsuleDB stores data to disk and does not sign the data. As such, this Section is evaluating the performance of CapsuleDB's core components only. It can be assumed that using the DataCapsule server for storage and enabling signatures will degrade performance, as they add networking time and complex cryptographic operations.

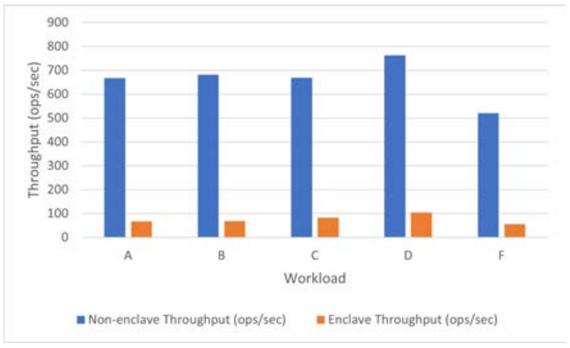
6.2 Overall Performance

Evaluating CapsuleDB using each of the aforementioned YCSB workloads reveals an interesting result. As Figure 9 shows, the enclaved version of CapsuleDB performs substantially worse by a factor of ten in both runtime and throughput in comparison to the non-enclaved version. This result was expected given crossing the enclave boundary is extremely expensive, especially because registers need to be flushed and several pages need to be encrypted or decrypted. Reducing the number of crossings or hiding the cost through batching could improve enclaved performance.

Now examining the relative performance of CapsuleDB on each of the workloads, Figure 10 displays the overall runtime for the two environments. In both cases, CapsuleDB performed best on Workload D and worst on Workload F. This is also an expected result, as Workload D is mostly reading recent data, a task CapsuleDB excels at thanks to the memtable and small upper levels. In contrast, the read-modify-write heavy Workload F is especially punishing. In the other workloads,

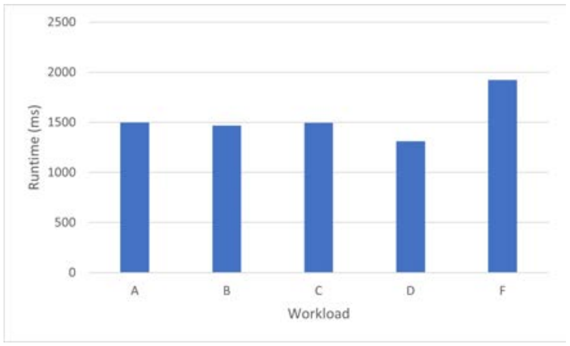


(a) Runtime

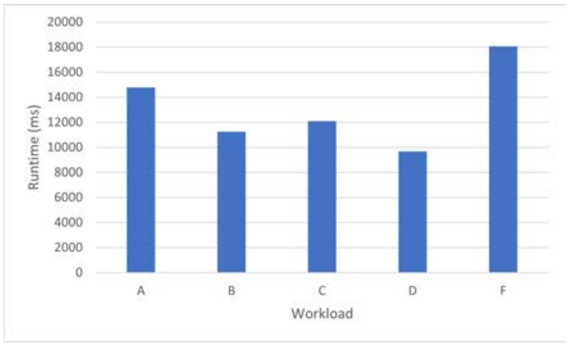


(b) Throughput

Figure 9: These graphs show the overall runtime and throughput of the YCSB workloads run against non-enclaved and enclaved CapsuleDB. Lower is better for runtime and higher is better for throughput.



(a) Non-enclaved overall runtime

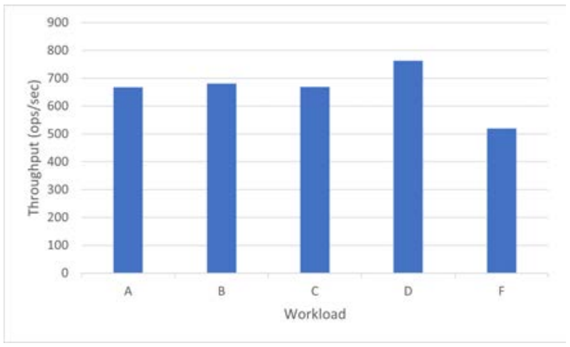


(b) Enclaved overall runtime

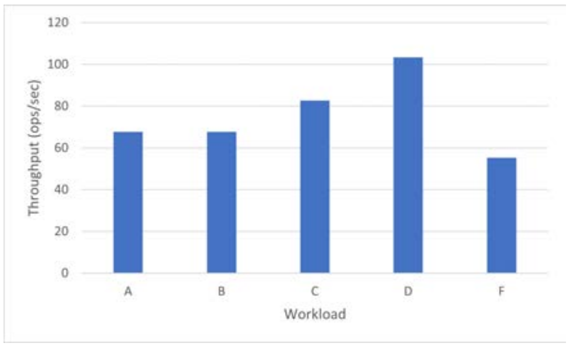
Figure 10: Overall Runtime of YCSB Workloads run against CapsuleDB.

the cost of PUTs is fairly light and can mostly be hidden by ZeroMQ sending the next request, since it does not need to wait for a response. However, repeated read-modify-write operations cause CapsuleDB to incur the cost of a read multiple times as well as a write, losing the slight pipeline parallelism gained from ZeroMQ.

Now examining the throughput in Figure 11, there is once again a similar pattern. Workload D has the best performance while Workload F suffers the most. Interestingly, there is similar performance for workloads A and B across tests, even though their read:write ratios are substantially different. As the next Section shows, this is likely because any number of read operations will drastically eclipse write operations in latency.



(a) Non-enclaved overall throughput



(b) Enclaved overall throughput

Figure 11: Overall Throughput of YCSB Workloads.

6.3 Reads versus Writes

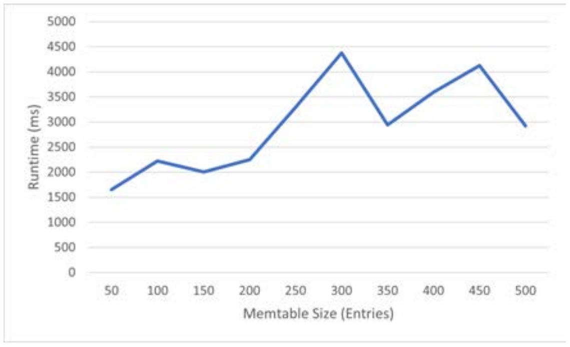
Analyzing the data, there is clearly a major difference in read and write performance. As table 1 lists, read latency is much worse, usually by more than a factor of ten. As discussed earlier, CapsuleDB does not respond to client PUT requests, allowing ZeroMQ to begin processing the next request while CapsuleDB is still adding a payload to the memtable. However, during a read, ZeroMQ must wait for CapsuleDB to retrieve the requested payload or search through all the levels. As such, reads require round trip networking time and potentially multiple requests to disk in comparison to writes which need half the networking time and a single memory access in the majority of cases.

	Reads	Writes
Average Latency	2435.19	140.085
Min Latency	314	33
Max Latency	109823	3533
95th Percentile Latency	2295	275
99th Percentile Latency	66111	394

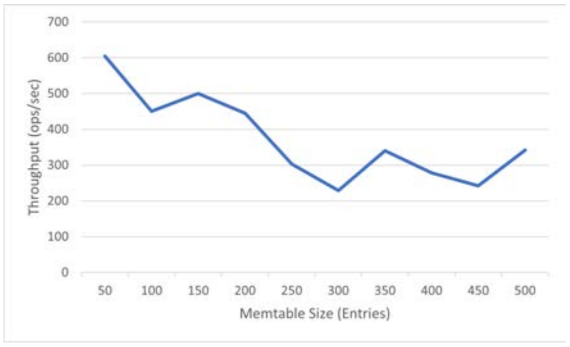
Table 1: Read and write latencies for Workload A. All entries are in μs.

6.3.1 Writes Over Time

As mentioned previously, writes usually only require a single memory access to the memtable to complete. However, some PUT requests will cause the memtable to fill, triggering a compaction. Figure 12 shows write latency over time using a workload consisting entirely of PUT requests. The data shows three spikes across the dataset, one at the start, one at the 20 ms mark, and one at the 90 ms mark. The first is likely due to the memtable being written out to L0 and the second occurs when L0 gets written to L1. The extremely large spike is likely due to a second L0



(a) Runtime



(b) Throughput

Figure 13: Runtime and throughput as a function of memtable size.

compaction to L1 with the required merge. Clearly, unlike the first occurrence where L1 was empty, taking the time to sort even the relatively small L0 into L1 incurs a major performance penalty. Additionally, as CapsuleDB expands and compactions at lower levels are needed, performance may decrease further. There are ways to offset this cost through multithreading and more efficient sorting, though compactions happens so rarely at lower levels and only have a minor impact at upper levels compared to GET requests that the performance impact is mostly hidden.

A potential improvement is through asynchronous compaction. Rather than not processing incoming requests during compaction, requests could instead be served while compaction takes place in the background. Once a level has finished compacting, the index can be updated to reflect the new blocks. Under this model, levels would likely be slightly oversized while compaction takes place, but CapsuleDB would not need to stop serving requests. All the data would still be present and there is no situation in which old data could be served before new data (as compaction happens bottom up from the lowest and oldest data). As a result, compaction would no longer occur on the critical path. However, this method could be problematic when so much new data comes in that a second compaction is triggered while a compaction is underway, causing upper levels to grow substantially larger and impact performance.

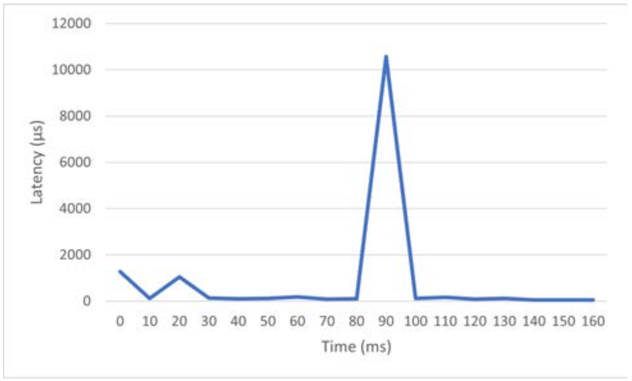


Figure 12: Write latency over time. This test was performed against CapsuleDB in an enclave using a workload consisting entirely of PUT requests.

6.4 Effect of CapsuleBlock Size

Finally, I evaluated the size of the memtable's effect on CapsuleDB's performance using Workload C (all reads) shown in Figure 13. Recall that the size of the memtable also determines the size of each CapsuleBlock. Somewhat counter-intuitively, performance actually declines as the memtable increases in size. Initially, this may not make sense as a larger memtable means more tuples kept in memory, especially more recent payloads. However, it also means more locks to track in the memtable and larger files to read in during retrieval. Thus, developers tuning this parameter should favor smaller sizes. However, given that each level substantially increases in size the lower it is in the database, use cases that retrieve mostly older data may actually want larger memtable sizes. This would cause CapsuleDB to have fewer CapsuleBlocks of larger size at lower levels, potentially speeding up searches while also bringing more data into the cache on a read at the cost of slower memtable accesses for fresh data.

6.5 Comparison with Other Systems

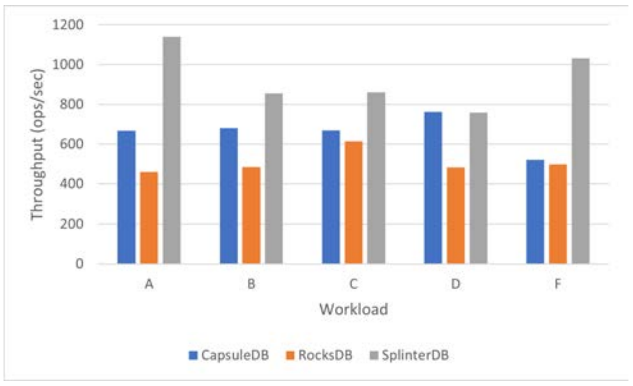


Figure 14: Throughput of CapsuleDB, RocksDB, and SplinterDB compared using YCSB.

In evaluating CapsuleDB against other systems, we see an interesting trend shown in Figure 14. I compared CapsuleDB against SplinterDB [6], a brand new cutting edge system, and RocksDB, a common industry standard, using the data collected by the SplinterDB team here [6]. Both are level-based KVS's and are evaluated using YCSB. Note that the data found in the SplinterDB paper was collected using a larger dataset, though throughput should be comparable. Interestingly, both RocksDB and CapsuleDB have an unsorted L0 and a memtable, but CapsuleDB generally has higher throughput. Therefore, I believe the indexing system and reduced write-amplification help keep CapsuleDB's throughput high over RocksDB, as RocksDB suffers from poor write management. In contrast, SplinterDB performs substantially better than CapsuleDB for the majority of workloads. Splinter uses a highly optimized data structure to achieve this performance, though I expect CapsuleDB's performance to approach this level with the addition of the optimizations described below in Section 7.

While this comparison is somewhat skewed because a direct comparison could not be made, it does reveal some interesting results about the core components of each system. In general, CapsuleDB's storage mechanisms can keep pace with RocksDB and SplinterDB, even when unoptimized. In addition, further development on CapsuleDB will only improve performance. However, it is unclear how CapsuleDB will perform with the larger dataset, especially when considering how quickly searches can be made when the KVS grows larger. While this is an area of active investigation, it is clear that CapsuleDB's core design provides compelling initial results, and that CapsuleDB will be able to perform comparably to other state of the art systems.

7 Future Work

While the core components of CapsuleDB do present a cohesive system, I also designed several additional features and optimizations. Note that several features were originally designed with the PSL in mind and rely on a DataCapsule server.

7.1 Accelerating Reads

As shown in Section 6, reads in CapsuleDB still suffer performance-wise when keys are not in the memtable due to CapsuleBlock requests from disk or from the DataCapsule server. While the cost of CapsuleBlock retrieval cannot be avoided, there are several optimizations that can accelerate other portions of the read pipeline. Currently, CapsuleDB uses a simple linear search to find the CapsuleBlocks that may contain a key within each level. This process can be accelerated through faster search algorithms, such as binary search, especially since levels are sorted. To limit the number of CapsuleBlock requests needed, bloom filters may provide a useful mechanism to determine whether a payload *could be* in a block. A bloom filter is a probabilistic data structure that has a zero percent false-negative rate. Thus, attaching a bloom filter to each `BlockHeader` structure could be used to determine whether there is even a possibility that a key is in the corresponding block before needing to pull the block itself, potentially saving on the number of needed requests. However, attaching a filter to each structure would cause the size of each level to balloon substantially, especially in the lower, larger levels with more blocks. However, with the introduction of Intel SGX2, the memory limit restriction has been removed. As such, bloom filters may now be feasible, especially since their size can be controlled depending on the desired accuracy. Furthermore, with the additional space granted by SGX2, more aggressive caching policies may also be possible. While it is unlikely the entirety of CapsuleDB could fit in memory for large datasets, given that each level increases in size by a factor of ten, keeping more CapsuleBlocks in memory would substantially increase read performance.

7.2 Compressing CapsuleBlocks

In environments with low amounts of or expensive storage, storing a timestamp for every KV pair may not be desired. Instead, a potential optimization is to assign the entire block a timestamp for when it was written to the DataCapsule by CapsuleDB. This way, block sizes would be substantially smaller without jeopardizing merge correctness, as new blocks would always have a more recent timestamp than older blocks. Furthermore, for PSL compatibility, the block's timestamp would always be older than entries that have propagated between lambdas but had yet to reach CapsuleDB. Extra care would have to be taken with the memtable, as those entries have yet to be written to a CapsuleBlock and thus would not yet have a shared timestamp. However, in exchange for this data compression, substantial temporal granularity is lost. Whether this is a problem will depend on the application, as well as its CapsuleDB usage patterns. Given the price of storage is constantly declining, this feature should be made a developer toggle to give users the option to decide which setup is better for their application.

7.3 Restoration After Failure

In the PSL system, individual lambdas maintain a local memtable that is synchronized with every other node in the system. However, in the event of a failure, the crashed PSL node would have to scan back through the entire DataCapsule to bring their memtable up to date. While technically feasible, this is slow and impractical, especially for larger data sets. CapsuleDB solves this problem by acting as stable storage that can quickly retrieve requested values. Then, rather than every memtable needing to be synchronized, a restarted lambda can immediately start serving requests by pulling any missing entries it needs from CapsuleDB.

However, this prompts the question, what happens if CapsuleDB fails? With a slight change, the index can serve as a recovery mechanism for CapsuleDB. When the index is updated, it can be written out to either the disk or the DataCapsule. Then, a CapsuleDB instance can be instantly restored simply by loading the most recent index. When used in conjunction with the GDP, CapsuleDB could also be stopped and started across different hardware locations and instances by loading an index, allowing users to easily move their data between CapsuleDB instances via the universally addressable DataCapsules. Once the index is loaded, only the records since the last index in the DataCapsule need to be played forward to restore the most recent KV pairs that were in CapsuleDB's memtable. This fast recovery system would be a powerful feature of CapsuleDB and provides a compelling example for the GDP's globally addressable networking.

7.4 Multiple Indexes

In addition to fast recovery, storing instances of the index in the DataCapsule would also provide a snapshot feature for CapsuleDB. Since the index defines the active data, a user could easily restore CapsuleDB to previous states by rolling back to a previous index. Since a DataCapsule is an append-only structure, the entire life of the database can be viewed in terms of these old indices. Furthermore, as data enters CapsuleDB, older levels may not need to be included in new indices. Instead, rather than storing old CapsuleBlock data, a new index could hold a reference to a DataCapsule record that holds the information for that level. While it would require an extra DataCapsule request for older data, it would also help keep index sizes small and manageable.

7.5 Deletion

While DataCapsules themselves do not currently have a mechanism to delete data, CapsuleDB could implement such a feature by removing the payload's reference in a CapsuleBlock. Then, when the CapsuleBlock is written back to the DataCapsule, that payload would have effectively been "deleted" as it is no longer considered active by the index. If combined with the multiple index system above, previous snapshots would still have that old data, ensuring the database would be able to recover those deleted keys. However, deletion causes substantial problems for CapsuleDB in terms of fragmentation. Excessive deletions would cause levels to require substantially more CapsuleBlocks, potentially causing the index to grow to an unbounded size. As storage continues to get cheaper, this issue will become less and less of an issue, and a compaction would cause the level to return to a run of mostly filled CapsuleBlocks. However, there are still several open questions about deletion's impact on overall performance and on the specific implementation details to ensure level size calculations remain correct that warrant additional investigation.

7.6 GDP Integration

By its original design, CapsuleDB was always proposed to be used in conjunction with the GDP. In the future, a few minor changes must be made to ensure compatibility. First, the local disk should be treated as a cache. If a file is not found locally for a read, only then should a request go to the DataCapsule server. Writes should be written both locally and to the server. Since the disk operations are already implemented, this would only require an extra check and then a call to the DataCapsule server. The second change is formatting. Right now, data is serialized to the disk using a simple protobuf structure, but it will need to change to match the format the DataCapsule server is expecting. This is a mostly a simple reformatting change, but will require some modifications to the way CapsuleDB starts. Specifically, a signing key will need to be provided so CapsuleDB has the authority to write the data into the DataCapsule. Finally, adjustments should be made to the way CapsuleDB starts-up to better utilize the GDP. Whenever CapsuleDB shuts down, it should write its current index into the DataCapsule. Then, on initialization, a user would have the option to point CapsuleDB towards an existing index in addition to starting a fresh copy. This would require the ability to write indices to a DataCapsule, as well as a loading mechanism to later read in a retrieved index. These changes are all relatively simple to make, but may become more complex depending on changes to the DataCapsule server interface.

8 Conclusion

CapsuleDB presents a compelling future for key-value stores in a globalized computing environment. While there are still several opportunities for future work on optimizing performance, the database already demonstrates the benefits of several key features around security and usability. Furthermore, CapsuleDB simplifies DataCapsule data management and accelerates data retrieval while exporting a simple and efficient interface for users. Initial performance data also demonstrates that CapsuleDB matches and exceeds current standard systems, with plans to improve performance to match the current state of the art. Thus, CapsuleDB's persistent storage capabilities, coupled with its novel indexing and data tracking scheme, provides a compelling analysis of its design which will power new classes of applications on both PSLs and the GDP.

9 Appendix

Please find additional information about CapsuleDB here.

9.1 Artifacts

CapsuleDB's code can be found on GitHub in the GlobalDataPlane orgnaization here. The CapsuleDB YCSB bindings can be found in the forked YCSB repo on GitHub in the GlobalDataPlane organization here.

9.2 Discussion of Parallelism

In the current model, there would be a single CapsuleDB instance running within a data pipeline. With ZeroMQ, CapsuleDB can handle incoming messages from several clients simultaneously. However, there are two major changes that would improve CapsuleDB's overall performance. First, writes can be decoupled from reads. Currently, reads and writes are processed by the same thread. Since writes only ever go to the memtable, reads to lower levels can occur asynchronously without consistency issues. For memtable level reads, we already have a locking mechanism in place to avoid read-after-write errors. Thus, CapsuleDB would be able to serve read requests while also processing incoming write requests. The only time that reads would need additional synchronicity mechanisms is during a compaction, and it would only be for the levels involved with the compaction itself.

The second change is somewhat more complex. In this scheme, there are multiple CapsuleDB instances running and clients submit their request to the closest replica. While this would improve availability and performance for highly distributed systems, it introduces new synchronization complexities, namely around the index. While the backend storage may not be an issue, through a shared storage drive or through a DataCapsule, ensuring index updates are propagated across instances is necessary to guarantee correctness. A potential solution is to read in an index whenever it is appended to the DataCapsule, but this would cause additional networking overhead and require some type of notification system so that CapsuleDB replicas knew when to pull an updated index. There are many potential benefits, but also substantial difficulties.

References

[1] Aws lambda. https://aws.amazon.com/lambda/.

[2] Google cloud functions. https://cloud.google.com/functions/.

[3] ZeroMQ Authors. Zeromq: An open-source universal messaging library. https://zeromq.org/.

[4] Martin Boissier, Carsten Alexander Meyer, Timo Djürken, Jan Lindemann, Kathrin Mao, Pascal Reinhardt, Tim Specht, Tim Zimmermann, and Matthias Uflacker. Analyzing data relevance and access patterns of live production database systems. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 2473–2475, 2016.

[5] Kaiyuan Chen, Alexander Thomas, Hanming Lu, William Mullen, Jeffrey Ichnowski, John Kubiatowicz, Anthony Joseph, and Ken Goldberg. Scl: A secure concurrency layer for paranoid stateful lambdas. 2021.

[6] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the bandwidth gap for NVMe Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 49–63. USENIX Association, July 2020.

[7] Brian Frank Cooper. Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB.

[8] Google. Asylo. https://asylo.dev/.

[9] Google. Protocol buffers. https://developers.google.com/protocol-buffers.

[10] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[11] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–9. 2016.

[12] 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, 2019.

[13] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel sgx. *arXiv preprint arXiv:2006.13598*, 2020.

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

[15] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64, 2015.

[16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[17] Facebook Database Engineering Team. Rocksdb: A persistent key-value store for flash and ram storage. https://rocksdb.org/, May 2021. Accessed: 2021-05-25.

[18] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.

[19] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[20] Ben Zhang, Nitesh Mor, John Kolb, Douglas S Chan, Ken Lutz, Eric Allman, John Wawrzynek, Edward Lee, and John Kubiatowicz. The cloud is not enough: Saving iot from the cloud. In *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.