DCR: DataCapsule Replication System

Hanming Lu
John D. Kubiatowicz, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-267
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-267.html

December 16, 2022

DCR: DataCapsule Replication System

by Hanming Lu

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:

Professor John D. Kubiatowicz
Research Advisor

December 15, 2022
(Date)

* * * * * * *

Professor Anthony D. Joseph
Second Reader

December 13, 2022
(Date)

DCR: DataCapsule Replication System

Hanming Lu

Department of Electrical Engineering and Computer Sciences

University of California, Berkeley

December 16, 2022

Professor John D. Kubiatowicz, Advisor

Abstract

The DataCapsule Replication System (DCR) is a continuous replication system using DataCapsule, a cryptographically hardened data container. The system uses DataCapsules as its underlying storage objects to provide a secure and efficient storage system on untrusted infrastructure. In particular, it uses an in-enclave proxy with HMAC channels to minimize write latency, lower compute and network requirements on clients, enable network efficiency, while maintaining data integrity, confidentiality, and provenance. In addition, it introduces optimizations such as periodic signatures to further reduce computation workload on clients, while maintaining provenance on every piece of stored data. In our benchmarks, the DCR has shown great performance optimizations from proxies, where throughput is at most 62% higher than the baseline. Also, a novel anti-entropy failure recovery mechanism is designed using the DataCapsule data structure to enable a compute- and network-efficient algorithm that handles server and network failures. At 30000 records, the DAG-based design's pairing latency is at most 76% lower than the baseline.

Contents

1 Introduction

In the world where more and more computing devices are closer to the end users, cloud-centric computing is losing its dominance [29]. Preferred by Internet of Things (IoT), Edge computing [20] is a computing paradigm that allows data collection and processing to happen on the same location, instead of sending data to a remote compute resource such as the cloud [5]. Edge computing provides properties that cloud computing is not able to match: lower latency and higher bandwidth than ever. However, edge computing brings security concerns. Different from computing power provided by credible cloud providers, edge computing resources on the edge can be provided by anyone, thus less trustworthy [27]. On the other hand, end users need to send code and input data for computations, which could be sensitive information. Thus, it is a challenging problem to protect users from untrusted infrastructure where the code is run and the data is stored. One of existing solutions is Global Data Plane (GDP) [15], a network infrastructure that solves these challenges by providing a robust networking and storage infrastructure.

GDP is a data-centric network that uses DataCapsules [15] as storage objects. In essence, a DataCapsule is a cryptographically hardened container that stores encrypted application data in the structure of a merkle tree [14]. It utilizes read-only metadata and append-only data to ensure data integrity, confidentiality, and provenance. With signatures and append-only semantics, DataCapsule prevents any unauthorized party to access decrypted data, or alter data without being detected. Also, it allows any third-party to audit the producer of any piece of data in an efficient manner. DataCapsule can be used in any kind of network, but is especially compatible to GDP. GDP provides a robust networking infrastructure where name-based routing is used, based on flat 256-bit GDP names, instead of network addresses in IP-network. This provides a convenient layer for users to directly address objects, such as DataCapsules, in the GDP system. GDP provides a robust networking infrastructure to efficiently interact, migrate, and address DataCapsules. However, while GDP aims to provide a robust infrastructure to DataCapsule routing, the current GDP design does not provide a clear storage solution.

GDP network is a robust networking layer, however, it is relatively complex to use without simple user-facing APIs. Paranoid Stateful Lambda (PSL) [6] is a solution that uses the GDP infrastructure, and provides an easy-to-use key-value store (KVS) interface to users. PSL is a Function-as-a-Service (FaaS) framework, similar to AWS Lambda [2], while extending its functionality to provide privacy-preserving stateful executions. It utilizes trusted execution environments (TEE), such as Intel Software Guard Extensions (SGX) [10], to provide provable privacy-preserving execution. In addition, it uses an in-enclave cache called memtable to enable stateful executions. PSL efficiently uses DataCapsule as its storage object and exploits its Conflict-free Replicated Data Type (CRDT) [19] property to provide eventually consistent ordering among PSL workers. Also, it exploits DataCapsule security properties to provide secure and private communications. With TEE and DataCapsule, PSL is able to provide an efficient, secure, and scalable computation

layer between DataCapsules and end users to provide a robust infrastructure and simple KVS APIs at the same time. However, existing PSL does not have a robust continuous replication mechanism to efficiently persist and replicate PSL execution results.

Both GDP and PSL projects surface the need for a storage solution that can solve the following challenges: 1) continuous replication with minimal-latency acknowledgements; 2) network efficiency and low burden on clients for both writes and acks; 3) an efficient failure recovery mechanism to synchronize replicas; 4) efficiently support data durability, confidentiality, integrity, and provenance.

We introduce a novel storage solution to solve these challenges: DataCapsule Replication System (DCR). DCR provides a continuous replication mechanism for DataCapsules. It efficiently supports storing multiple replicas of the PSL key-value store embedded in DataCapsules, and serve as the storage component in GDP. The system uses DataCapsules as its underlying storage objects to provide a secure and efficient storage system on untrusted infrastructure. In particular, it uses a in-enclave proxy with HMAC [3] channels to minimize ack latency, lower computation and network burden on clients, enable network efficiency, while maintaining the same level of security. In addition, it introduces optimizations such as periodic signatures to alleviate the burden on client to sign every write message, while maintaining provenance on every piece of stored data. In our benchmarks, DCR has shown great performance optimizations from proxies, where throughput is at most 62% higher than the baseline. Also, an Anti-entropy failure recovery mechanism is specifically designed around the DataCapsule data structure to enable a compute- and network-efficient algorithm that handles server and network failures. At 30000 records, the DAG-based design's pairing latency is at most 76% lower than the baseline.

In this report, I present the design and implementation of DataCapsule Replication system, a storage solution for both GDP and PSL. The rest of the report is organized as follows. Section 2 will give a more comprehensive background on DataCapsule, Global Data Plane, and Paranoid Stateful Lambda. Section 3 will discuss the threat model that the system is aiming to address. Section 4 and 5 give the full picture on the architecture design and optimizations. Section 6 talks about the implementation details. Section 7 evaluates the system and gives performance comparisons. Section 8 discusses future work and Section 9 gives a conclusion.

2 Background

2.1 Components Overview

Figure 1 gives a high-level overview on all components involved in a full picture. From top to bottom, PSL is the user-facing interface, which provides a key-value store interface. PSL is responsible for providing secure and fast computation, using an eventually consistent in-memory cache, while CapsuleDB acts as a secure next-level cache for PSL workers. CapsuleDB provides consistency and fast data retrieval for PSL workers, and handles

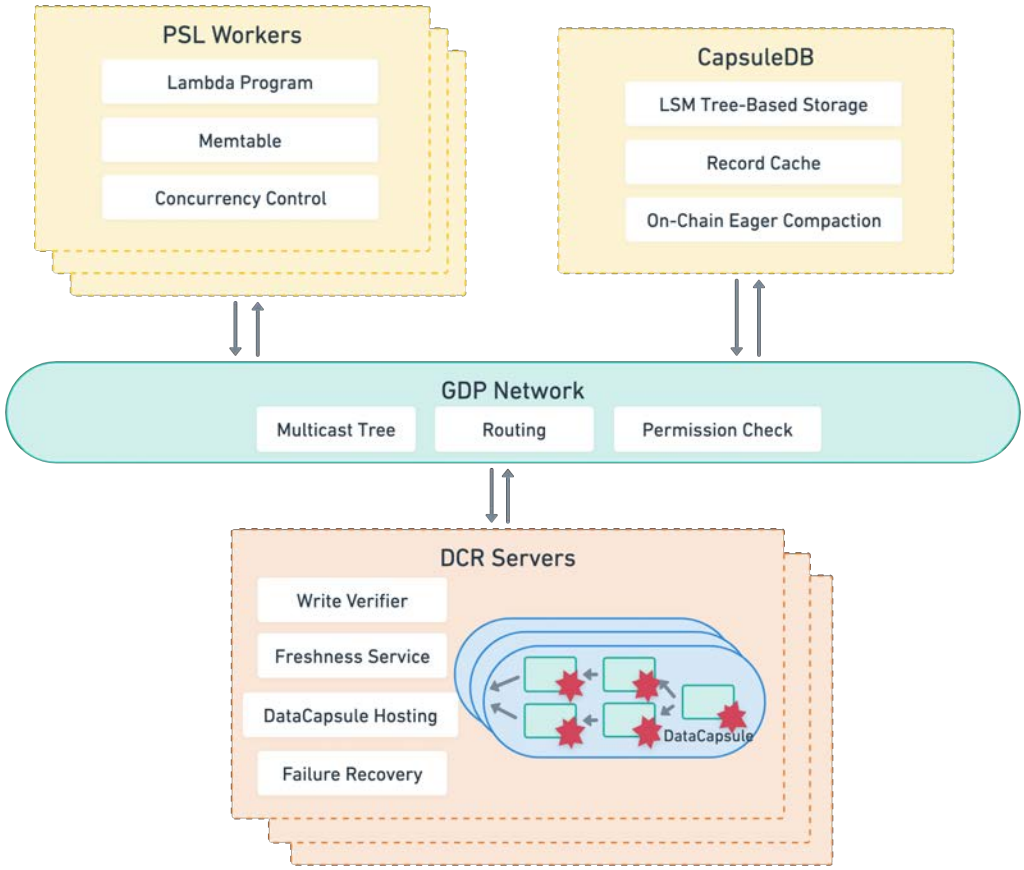

Figure 1: Overview of components.

failure recovery by incorporating DCR servers. In the middle, Global Data Plane (GDP) network acts as the router among three components: Paranoid Stateful Lambda (PSL) workers, CapsuleDB, and DCR servers. The GDP network provides efficient and secure data-centric routing to 256-bit destinations, and provides server multicast when multiple destinations have the same name. Individual components need to register with the GDP network for permission checks. At the bottom, the DataCapsule Replication system (DCR) serves to provide durability and replication. In particular, DCR servers are responsible for collecting writes from upper components like PSL workers and CapsuleDB, and then persist and replicate them. DCR servers are responsible for ensuring data durability under failures.

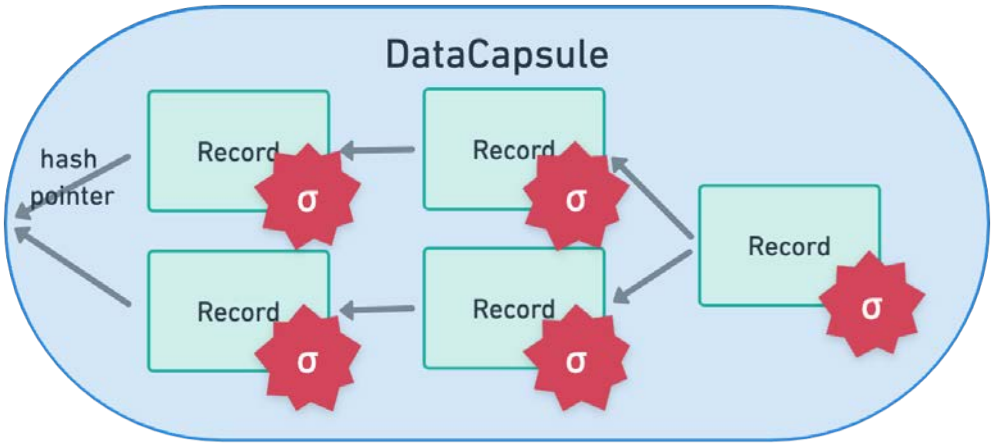

Figure 2: Overview of a DataCapsule.

2.2 DataCapsule

A DataCapsule [15] is the underlying storage object for the DataCapsule Replication system. As shown in Figure 2 , a DataCapsule consists of multiple records of data, connected by cryptographic hash pointers, with the first records pointing to its read-only metadata. Every DataCapsule has a public/private key pair that identifies the owner of the DataCapsule. The public key is included in the metadata, and the private key is stored and protected by the DataCapsule owner. With the private key, the DataCapsule owner has the ability to append new records to the DataCapsule, but is not able to modify existing records. Each write appends a new signed record to an existing record in the DataCapsule using cryptographical hash pointers. Each write must be signed by the private key to be considered a valid write. After construction, this chain of records has a structure of a merkle tree, which has two properties: 1) it is Conflict-Free Replicated Data Type (CRDT), meaning two partial DataCapsules can easily synchronize by taking a union of records; 2) it prevents malicious parties from forging or altering existing records without being detected. In summary, a DataCapsule has a cryptographically hardened structure of a merkle tree, which consists of records connected by hash pointers, wrapped around by metadata.

In addition, DataCapsules are storage units designed for global distributions, so it is critical to design a storage system that can efficiently store, organize, and migrate them globally. This system also needs to secure the data inside each DataCapsule maintains its integrity and provenance, as well as preventing unauthorized entities from altering or accessing sensitive information.

2.3 Global Data Plane (GDP)

The Global Data Plane (GDP) [15] is a data-centric framework for routing to where DataCapsules are stored and transmitted. The GDP is a robust networking infrastructure using name-based routing instead of network addresses as in IP networks. In particular, every identity in the GDP network has a unique cryptographically derived flat 256-bit name, which is called GDP name and is used for routing. The GDP network focuses on routing security, ensuring that in-transit DataCapsule records cannot be tampered by adversaries without being detected. As a feature, the GDP network supports unicast and multicast, which is a great feature DataCapsule Replication system may use to optimize replication performance.

2.4 Intel Software Guard Extensions (SGX1)

Intel Software Guard Extensions (SGX) [10] provides a Trusted Execution Environment (TEE). It exploits hardware-based memory encryption to create a private region of memory, named enclaves, that is isolated from other processes. Intel SGX protects the code and data running inside the enclave from other processes, even from more privileged processes like the operating system. Security features of Intel SGX include data confidentiality, data integrity, and code integrity. In particular, data inside an SGX enclave is confidential such that no unauthorized entities have access to decrypted data. Intel SGX guarantees data and code integrity such that no unauthorized entities can alter or modify them. In addition, Intel SGX also supports remote attestation, which allows another entity to authenticate the code running inside the enclave. This allows other entities to trust the code running inside an enclave after attestation.

We are aware of the fact that Intel SGX could be undermined by security attacks such as side channel attacks [7, 9, 21]. However, there are active research efforts to mitigate or prevent such attacks [7, 18, 21, 22]. Thus, we consider it orthogonal to our report and will not discuss it further.

2.5 Paranoid Stateful Lambda (PSL)

Paranoid Stateful Lambda (PSL) [6] is a secure Function-as-a-Servie (FaaS) framework, which provides privacy-preserving execution using Intel SGX. Each PSL worker (i.e., a lambda) runs in an enclave, specifically Intel Software Guard Extensions (SGX), to ensure privacy-preserving execution. Each PSL worker maintains a local memtable, which is an in-enclave cache of the most recently updated key-value pairs, stored in plain text. Individual PSL workers collaborate with one another to perform large parallel computations. The secure Concurrency Layer (SCL) [6] facilitates communication among PSL workers. It is a communication protocol between PSL workers that achieves eventual consistency among PSL workers computation results. In particular, every write is multicasted to all other PSL workers, and ordered by SCL, resulting in an eventually consistent ordering of writes. All

communications among PSL workers are encrypted, private, and unforgeable. PSL utilizes DataCapsule Replication system for persistent storage. In particular, PSL multicasts an ordered and connected chain of DataCapsule records to DCR, which will then store and replicate the records for persistence.

2.6 CapsuleDB

CapsuleDB [16] is an eventually consistent in-enclave key-value store. It is an LSM tree-based storage [17] that provides consistency and secure access to records to PSL workers. Specfically, PSL workers can save and coordinate state information through CapsuleDB, to achieve consistency and fast retrieval of records. CapsuleDB acts as a next-level cache and provides failure recovery in case of crash failures. CapsuleDB employs a novel compaction mechanism to enable fast recovery, less signature overhead, and adjustable write amplification. CapsuleDB utilizes the DCR system to persist and replicate data blocks. Similar to PSL, CapsuleDB multicasts ordered and connected chains of DataCapsule records to DCR, which will then store and replicate the records for persistence.

3 Threat Model

In this section, we discuss the threat model that the DataCapsule Replication system aims to address. Overall, we assume a federated storage model, where autonomous storage resources, either on the edges or in data centers, are collectively organized to form a widely distributed system for data storage. This means that anyone can contribute storage resources and become a part of the system.

We use the typical assumptions for cryptographical tools in the report, including signatures, symmetric and asymmetric encryptions, and hash functions. And we assume that adversaries have reasonably limited computation power so they cannot launch brute-force attacks. We also assume a malicious network, where packets may be discarded, and network can partition or fail completely. Another assumption is each entity can protect its secret keys.

3.1 Trusted DataCapsule Writer

For each DataCapsule, its writer is the only trusted entity. In particular, every modification to the DataCapsule from the writer, either intentional or accidental, is considered to be the source of truth and will be trusted. At the same time, a writer does not need to store a DataCapsule locally, instead, it delegates that responsibility to DCR, and employs the hash chain of merkle tree to ensure data integrity. Readers of a DataCapsule should trust the changes from the verified writer.

3.2 Untrusted Infrastructure

Confidentiality and Integrity: Our system is based on a federated storage model where anyone can participate in the network and provide storage services. With a large and diverse collection of storage providers, we should provide minimal trust to the infrastructure and their administrative entities. Therefore, despite the fact that DataCapsules being physically stored in the DCR servers, these servers should not have access to the decrypted data and should not have the capability to tamper with data without being detected.

Availability and Persistence: Although we do not trust service providers with data confidentiality and integrity, DataCapsule owners pay service providers for hosting their DataCapsules. Therefore, service providers have the financial incentives to provide service availability and data persistence. Thus, we trust service providers to provide availability and provide persistence as promised. However, in scenarios where individual nodes cannot guarantee such two properties, either intentionally or unintentionally, the system should provide alternate solutions to mitigate the situation. In the worst case where no alternates are possible, the clients should still be able to ensure there is no violation of confidentiality or integrity.

3.3 Untrusted Third Parties

The federated storage model allows any third party to join the network and contribute. The system should do an effective job of defending against any reasonable adversaries.

4 Design

4.1 Architecture Overview

The key task of the DataCapsule Replication system is to provide continuous, persistent, replicated, and secure storage for DataCapsules. The essence of DCR is that every DCR server securely persists a complete replica of assigned DataCapsules using untrusted infrastructure. By providing a failure recovery synchronization scheme among the servers, DCR is guaranteed to provide durability under network partition, network failure, or f server failures. Also, it handles disk corruption and partial loss of data by replicating a DataCapsule on multiple DCR servers.

Figure 3 provides an overview of the DCR system architecture. The DCR system utilizes a federated storage model, where there are multiple DCR servers, each represents a physical server administrated by an autonomous storage provider. Within each physical server, a DataCapsule is used as the underlying storage object. Each physical server hosts one or more DataCapsules, each DataCapsule is hosted with a delegation certificate, which is issued by the DataCapsule owner to certify that the server is authorized to host the DataCapsule. For networking, each server has a 256-bit GDP name. The underlying GDP network infrastructure uses the GDP names to achieve name-addressable networking.

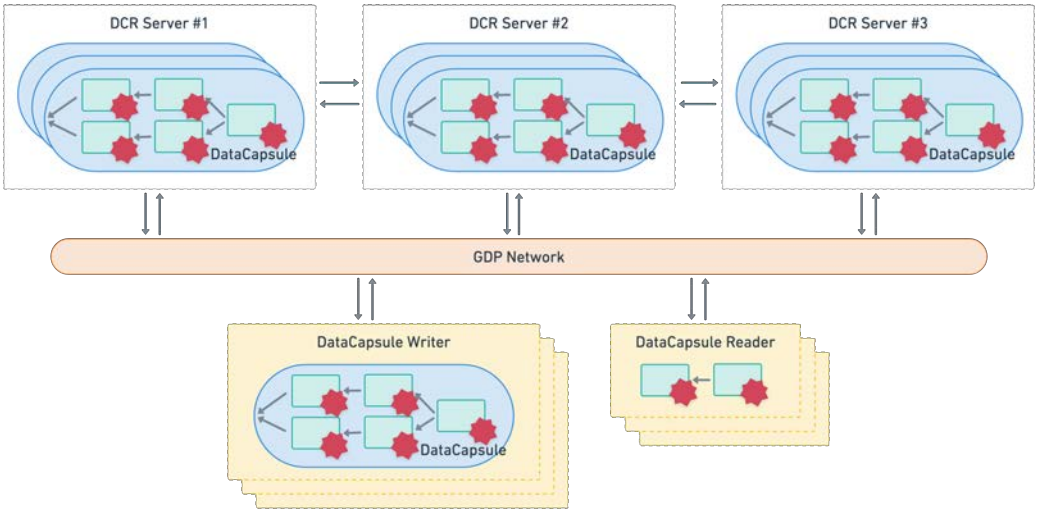

Figure 3: The architecture of DataCapsule Replication system. Each DCR server hosts one or more DataCapsules. DCR servers communicate with the GDP network to receive write and read requests. Failure recovery is achieved by anti-entropy via pairing. DataCapsule writer and DataCapsule readers communicate with the GDP network for write and read requests, respectively.

For each record write from the DataCapsule writer, we exploit the GDP's multicast functionality to propagate write updates to all servers hosting the addressed DataCapsule. When a DCR server receives a record to append, it verifies the cryptographic hashes and signatures using the DataCapsule writer's public key, then locally appends the record to its parent. After that, it will send an acknowledgement message to the sender. On the other hand, the sender will receive one ack from each server hosting its DataCapsule. When a quorum is reached, the sender is able to mark the record persistent and is safe to discard it locally. For a DCR client to retrieve a record, it can simply send a read query, containing the record's hash, to any server hosting the DataCapsule. It is safe to do so because all DataCapsule records are read-only, so a verified persistent record cannot be modified by anyone, and it can be verified by the reader using the DataCapsule writer's public key upon receiving the record.

By exploiting the CRDT property of DataCapsules, there is no synchronization required among DCR servers if there is no failure. If failures occur, either server or network failures, and causes missing records in a DataCapsule replica, anti-entropy will resolve it. Anti-entropy is a specially designed failure recovery mechanism for synchronizing DataCapsule replicas among DCR servers. By considering a DataCapsule as a DAG, anti-entropy optimizes network and compute efficiency while providing eventual consistency semantics to DataCapsule replicas.

4.2 DCR Server

A critical component in the DataCapsule Replication system is the DCR server. A DCR server is a physical server that a storage provider provisions and manages. DCR servers have the responsibility of hosting DataCapsules and serve client requests. To host a DataCapsule, a DCR server will receive a delegation request from the DataCapsule's writer, it will then advertise its DataCapsule replica in the GDP network. After that, the DCR server will receive write or read requests from the GDP network, and serve them accordingly. A single DCR server is capable of hosting multiple DataCapsules, depending on its local resource capacity. Note that each DataCapsule will be advertised separately to the GDP network for routing purpose.

Besides serving requests from clients, DCR server also serves synchronization requests from each other. In particular, when a server detects missing records, or after a timeout, it will send a sync request to its peers as a step in the failure recovery process, which will be discussed in Section 4.6. After synchronization, the DCR server will then become complete, which provides eventual consistency.

DCR servers have the responsibility of providing persistent storage, and do not have any authority over what is in the DataCapsule. As long as an incoming record is properly signed, the DCR server should accept it, store it, and reply with an ack. The only possible reason it can reject an update is service unavailable, which is a transient failure.

Integrity: Integrity means no unauthorized party is able to alter or discard data without being detected. Since DCR is based on a federated storage model, DCR servers are considered untrusted infrastructure as well. We prevent DCR servers from altering data, including both the encrypted payload and hash chain structure, by utilizing a signature on each record's encrypted payload and cryptographic pointer. Thus, if there is any unauthorized modification, any entity with the DataCapsule verification key can detect it.

Confidentiality: Similarily, DCR servers should not have access to decrypted payload data, despite the fact that they physically store the data. This is achieved by separating storage and permission to access. DataCapsule writers encrypt data before sending it to DCR servers for persisetence. Thus, DCR servers store encrypted data and do not possess decryption keys because they do not need it to provide persistent and verifiable storage. On the other hand, authorized readers have access to decrypted data by retrieving encrypted data from a DCR server and then decrypting it locally.

Provenance: The DCR is designed to provide provenance for every bit of data, which means any third-party auditor can verify the producer of every stored record. This is done using the same signature mentioned above, where each record is signed by the DataCapsule writer. Thus, any third-party auditor can retrieve the record and its signature from a DCR server, then verify it using the DataCapsule writer's public key.

Fault Tolerance: Each DataCapsule can be replicated multiple times. In particular, we use "full replication" where each replica is a full copy of a DataCapsule. Thus, the DataCapsule Replication system can tolerate f server failures when there are $f+1$ servers.

It can tolerate more server failures than the $2f + 1$ server model because each server is hosting a complete DataCapsule and DataCapsules have the properties of append-only and provenance. Since the DataCapsule is append-only, no existing record can be removed or altered; since every record has a signature, all records' integrity and provenance can be verified; since each DataCapsule replica is a complete one, one non-faulty replica is sufficient to reconstruct the entire DataCapsule. Note that a replica needs to complete only if f servers fail completely, otherwise, all we need is a record to be present in one DataCapsule. Thus, any new DCR server or client can reconstruct a complete DataCapsule in a verifiable manner using only one non-faulty replica.

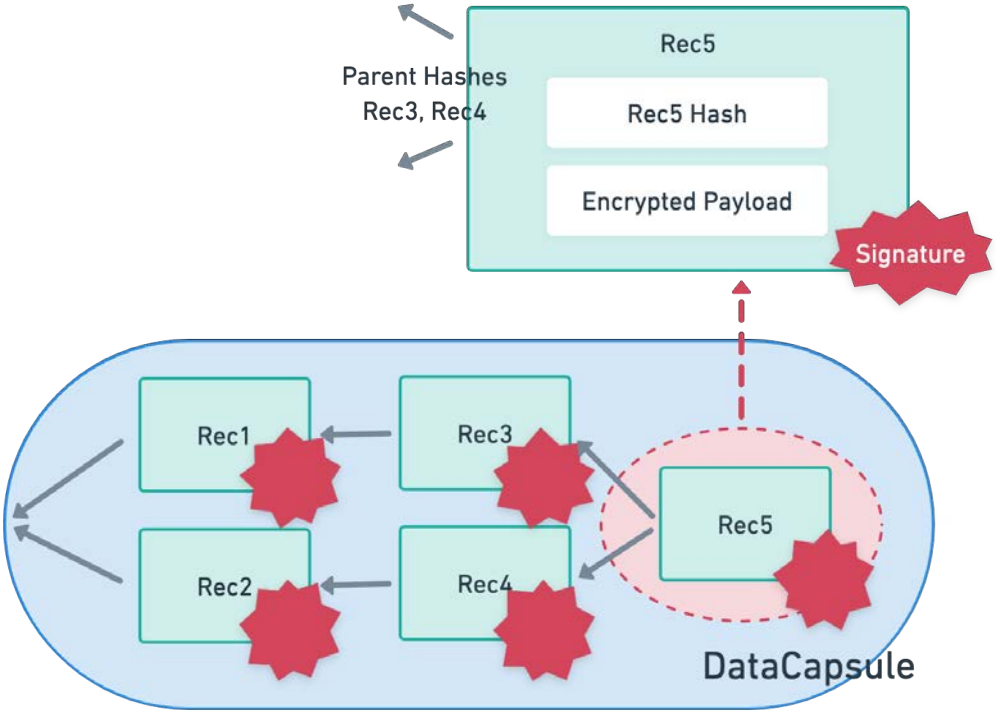

Figure 4: The detail of a DataCapsule.

4.2.1 DataCapsule in Detail

DCR servers use the DataCapsule structure for organizing data. Figure 4 gives a detailed visual representation of a DataCapsule. As shown in the figure, each record contains an encrypted payload, a hash of the record, a signature, and one or more parent hashes. The encrypted payload contains actual application-level data. It is Advanced Encryption

Standard-encrypted (AES) by the DataCapsule writer. The signature is generated over the entire record using the DataCapsule writer's private key, which is an Elliptic Curve Digital Signature Algorithm (ECDSA) signing key. Parent hashes point to the record's parents, which are previous records in the DataCapsule. There can be one or branches in a DataCapsule, for instance, *Rec1* and *Rec2* are in two branches. The actual data structure used to store each DataCapsule is dependent on implementations and should be indifferent from the perspective of design. We will discuss the DataCapsule implementation in Section 6.

4.2.2 DCR Client

A DCR client interacts with the DCR servers for the purpose of producing or consuming DataCapsules. There are three types of DCR clients: DataCapsule owner, DataCapsule writers, and DataCapsule readers. The DataCapsule onwer creates the DataCapsule and delegates it to DCR servers to host the DataCapsule. The DataCapsule writer produces records and writes to the DataCapsule. The DataCapsule writer possesses an AES encryption key for encrypting application-level data in each record. It also has an ECDSA signing key to generate signature on every record. It is the only entity that has the permission to write to DataCapsule. The DataCapsule reader is the consumer of DataCapsules. In addition to verifying DataCapsule records, it also has the permission to read decrypted payloads. It has an ECDSA verifying key and an AES decryption key, for verifying records and decrypting record payloads, respectively.

4.3 Append-Only Write

One responsibility of a DCR server hosting a DataCapsule is to handle write requests from clients. After registering the DataCapsule with the GDP network, DCR server will then start receiving write requests.

DataCapsule writer is responsible for generating records and determining the hash chain ordering. Each write to the DataCapsule is essentially an append operation, which adds a new record. Specifically, the DataCapsule writer encrypts application-level data with its AES encryption key, signs it with its ECDSA signing key, sets its parent record, and then sends it to the GDP routing network. The GDP routing network will then propagate the write to DCR servers that host the DataCapsule.

4.3.1 Write Verification

For each write received, the DCR server verifies the signature with its ECDSA verifying key, finds its parent in the DataCapsule, and then append it to the parent by storing it locally to disk. After that, the DCR server sends a signed acknowledgement message to the writer. The writer must verify the signature on the ack to make sure that it is from one of the organizations that the owner has delegated to physically host a replica of the

DataCapsule. The writer can mark a record as persisted when it receives a quorum of verified acks.

4.3.2 GDP Network Multicast

DCR utilizes the multicast functionality in the GDP network infrastructure. In particular, the DataCapsule writer sends every write to the GDP network, which will then multicast the write to all delegated DCR servers. Multicast enables the GDP network to efficiently propagate every write. This optimizes scalability, especially when a DataCapsule writer wants additional durability by delegating its DataCapsule to tens or hundreds of DCR servers. To utilize this multicast tree, every DCR server needs to register with the GDP network with a delegate certificate from the DataCapsule writer, which we will discuss further in Section 4.7. However, the DCR system does not rely on reliable multicast for correctness. As long as a quorum of DCR servers receive the writes, the DataCapsule writer will receive a quorum of acks and mark the records as persisted.

4.4 Hash-based Read

Similar to serving write requests, DCR servers also have the responsibility of serving read requests from clients. Again, a DCR server will start receiving read requests after registering the DataCapsule with the GDP network. In addition to supporting multicast, the GDP network supports anycast; thus a read request will be routed to the closest registered DataCapsule.

Every record has a hash value, which is generated using an SHA256 hash function on the record. A client can retrieve a specific record using its hash by sending a hash-based request to the GDP network. Since DataCapsule is an append-only data structure and every record can be independently verified, a client can retrieve the record from any delegated DCR server. Thus, when a read request is sent to the GDP network, it is forwarded to only one delegated DCR server. When the DCR server receives the hash-based read request, it will lookup the hash in its DataCapsule, and then reply it directly to the client. When the client receives the record, it will verify its signature for integrity and provenance, and then consume it.

Since this mechanism only needs to read from one server, it minimizes read latency by utilizing GDP network to optimize for locality. However, this mechanism does not rely on GDP network for correctness. Even if the GDP network routes the request to a distant server, the client should still receive a record, even though the latency could be higher. In the rare case where a desired record is not on the requested DCR server, the DCR server is responsible for fetching the record from another DCR server, and then serve the request. In the worst case where the requested server is malicious, the client can still verify if the retrieved record is tampered with by using the DataCapsule verifying key.

4.5 Freshness Service

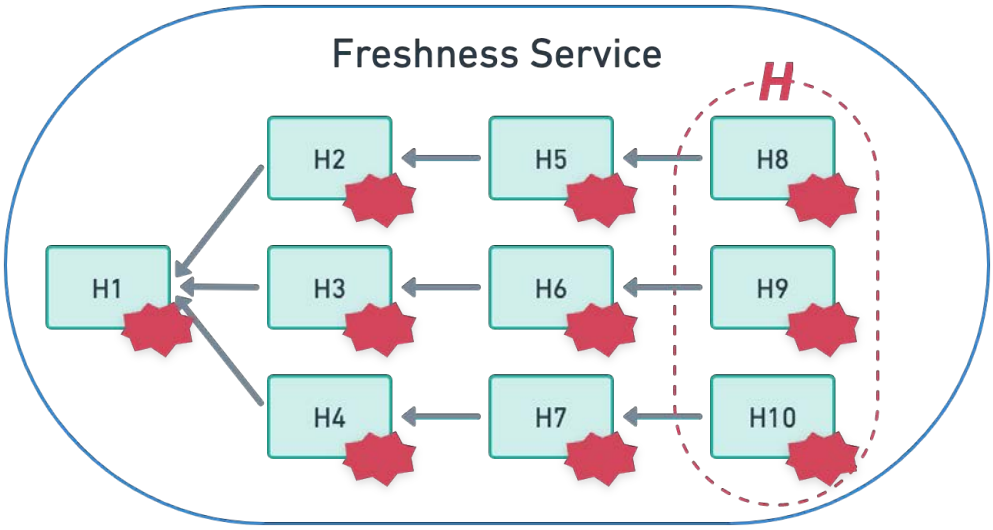

Figure 5: A visualization of a DataCapsule.

In situations where a client wants to know the most recent records in a DataCapsule, hash-based reads may not be sufficient. For instance, if a writer crashes and wants to reconstruct the DataCapsule locally, it needs to know which existing records to append new records to. Thus, in addition to retrieving a record via hash, DCR provides a *freshness service*, where a client can also fetch the most recent record for every branch in a DataCapsule. In particular, a client will send a freshness read request of a DataCapsule to the GDP network, which will multicast the request to all DCR servers hosting that DataCapsule. When a DCR server receives a freshness read request, it will send a response with the latest records' hashes on each branch, which is the set H in Figure 5. As long as a client receives $f + 1$ responses, where f is the number of faulty servers, it is able to construct the latest records' hashes. In detail, since every record is unforgeably defined because of signatures and hashes, a client can verify a record's integrity and use its timestamp to find the latest record on a branch.

4.5.1 Freshness Service Security

In a federated storage model, we should consider the possibility of faulty DCR servers. We assume there are potentially f faulty servers, either malicious or crash faulted. In this case, as long as a client receives $f + 1$ responses, the client will verify every response's

integrity to filter out potentially forged responses. After that, the client can use each record's timestamp to find the latest record of each branch.

Since the freshness service is based on all DCR servers hosting the DataCapsule, there is less potential vulnerability of DDoS attacks compared to a centralized design. In addition, each freshness read requires a light workload on the servers, which is only returning a set of hashes. Therefore, this design can mitigate DDoS attacks pretty well.

4.6 Anti-Entropy Failure Recovery

In a federated storage model, it is beyond any service provider's ability to act as a central service to coordinate and handle failures in all DataCapsule replicas. A DCR system exploits anti-entropy, a failure recovery mechanism that, for each pair of replicas, compares the data of them, and then update each other to be the union of the two. It is a pair-wise, epidemic synchronization mechanism that allows all replicas to become eventually synchronized without the need to reach out to a centralized service.

A naive anti-entropy design is that for each pair-wise update, two DataCapsule replicas exchange all of their current records' hashes, and fill out each other's missing records. This naive design requires sending one or both DataCapsule replicas' complete hash chains over the network. Most of the time, when two DataCapsule replicas exchange states, they are in complete agreement or have minimal differences. This property arises because DCR utilizes GDP multicast to allow all replicas to receive every new record, so DataCapsule replicas should be identical under no failures. In such cases, the naive design sends over at least one complete replica's hash chain and compares the entire chain of records. Given the size of a complete DataCapsule can be in the size of GBs, TBs, or even PBs, the naive design could be prohibitively expensive. Thus, we introduce a novel design that exploits the DAG structure of DataCapules.

4.6.1 DataCapsule as a DAG

A DataCapsule is a directed acyclic graph (DAG) [25]. It is a DAG because it has an append-only structure and every new record must reference one or more existing records as its parent(s). Thus, each hash pointer can be seen as a relationship based on time, where it points from a later record to an earlier record. In this setup, any cycle would have to include a pointer in which an earlier record points to a later record, which is impossible given DataCapsule's append-only design.

Before introducing the novel anti-entropy algorithm on DAGs, we need to define three terms: sink, source, and DAG digest. We define a sink to be a record without local outgoing edges. Similarly, we define a source as a record without local incoming edges. Lastly, we define a DAG digest of a DataCapsule to be the set of all sinks and sources. As shown in Figure 6, each DataCapsule replica can have one or more sinks and one or more sources. Also, by definition, the DAG digest of Figure 6 contains the two sinks and the three sources.

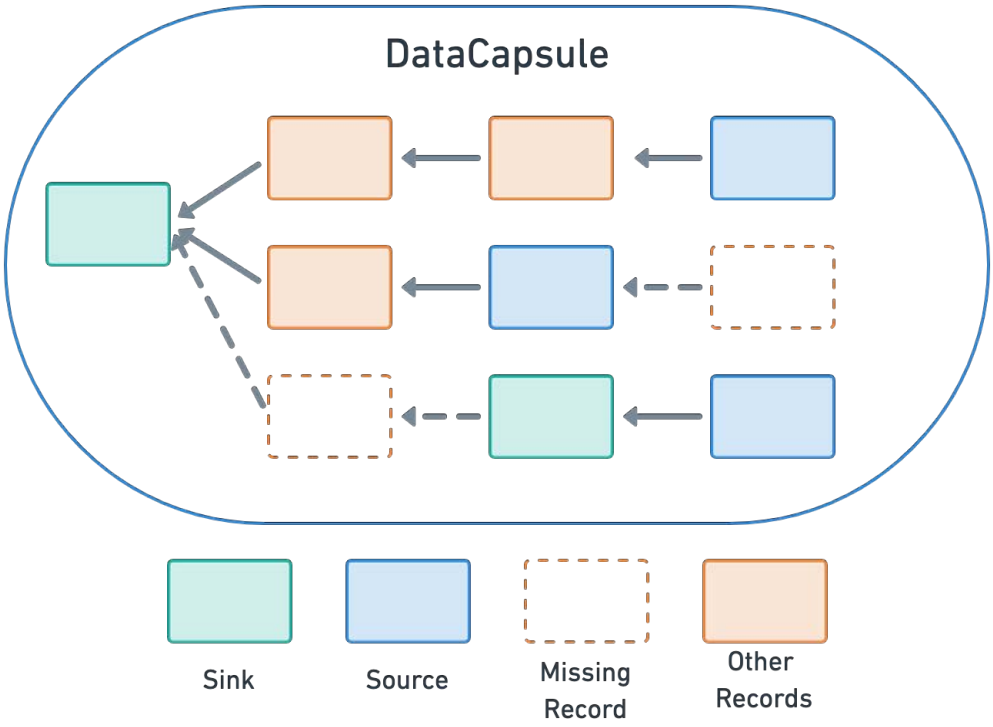

Figure 6: A DataCapsule as a directed acyclic graph (DAG) with two sinks and three sources.

4.6.2 Anti-Entropy on DataCapsules

We introduce a novel Anti-Entropy mechanism that exploits DataCapsule's DAG structure to optimize network and compute efficiency. Periodically, each DataCapsule replica randomly selects another replica, and starts pairing. The pairing requester, A, sends a pairing request with its replica's DAG digest (i.e., sinks and sources) to the requestee, B. After receiving the pairing request from A, replica B compares A's DAG digest with its own, and then identifies the records that A is missing. The DAG digest comparison algorithm is shown in Algorithm 1. With this algorithm, B can generate a set of records that A is missing, and then send it back to A. Along with these records, B also sends its DAG digest to A. A will receive, verify, and then store these records locally. At this point, A's replica is a union of A's and B's. A will then compare its updated DAG digest with B's DAG digest, using the same algorithm above. If there are any missing records, A will send them back to B, who will similarly verify and store the received records. By the end of this pairing session, replica A and B will have the same copy, which is the union

Algorithm 1 DAG Digest Comparison

$sinkA$ = all sinks of replica A
$sinkB$ = all sinks of replica B
$sourceA$ = all sources of replica A
$sourceB$ = all sources of replica B
$nodeB$ = all records of replica B
L = records to return

// To handle holes in a chain and stale branches
for $v \in sourceA$ **do**
    if $v \in nodeB$ and $v \notin sourceB$ **then**
        add all connected nodes ahead of v in $nodeB$ to L, until we reach a node in $sinkA$ or in L
    end if
end for
for $v \in sinkA$ **do**
    if $v \in nodeB$ and $v \notin sinkB$ **then**
        add all connected nodes after v in $nodeB$ to L, until we reach a node in $sourceA$ or in L
    end if
end for

// To handle missing branches
for $v \in sourceB$ **do**
    if $v \notin sourceA$ and $v \notin L$ **then**
        add connected component of v to L
    end if
end for
for $v \in sinkB$ **do**
    if $v \notin sinkA$ and $v \notin L$ **then**
        add connected component of v to L
    end if
end for

of them. By periodically pairing and randomly selecting pairing targets, this pair-wise, epidemic synchronization mechanism allows all DataCapsule replicas to become eventually synchronized.

In DCR, DataCapsule replicas are usually up-to-date because of the efficient GDP multicast. This algorithm is network and compute efficient because most pairing sessions only require the requester to send its DAG digest to a requestee, which verifies both replicas are up-to-date and simply sends back an acknowledgement. Also, since the number of sources and sinks is only related to the number of branches, but not related to DataCapsule size, this method is scalable. For instance, a DataCapsule with one record and a DataCapsule with one branch of many records have the same DAG digest size: one sink and one source.

In cases where several records are missing, this algorithm only requires exchanging missing records, optimizing network efficiency. In rare cases where two replicas have significant differences, such as a replica has just joined the network, the requestee can send a partial DataCapsule, and ask the requester to pair with another replica to receive the rest. This limits performance impacts on any individual requestee, while allowing the requester to catch up as soon as possible. In the worst case where a requestee is hacked and becomes malicious, the worst it can do is to send empty or modified records. This will not affect correctness because the modified records will fail verification, and the requester can receive correct updates from another replica in the next pairing session.

4.7 Security Model

4.7.1 Security Goals

DataCapsule Replication system has three key security goals: data integrity, confidentiality, and provenance. For data integrity, our system assures that no unauthorized party has the capability of altering or discarding stored data without being detected, this includes the infrastructure that hosts the data. In terms of data confidentiality, we ensures that only authorized clients have access to decrypted data. Lastly, we provides data provenance, which means any third party will have the capability of tracing and verifying the producer of every piece of data. For a replication system built on top of autonomous storage resources, these security goals are critical to protect clients from malicious parties.

4.7.2 Key Management

There are four parties considered in the DataCapsule Replication system: service providers, physical servers, DataCapsule owners, and DataCapsule readers. In particular, a service provider is an organization that provides storage services, such as IBM. A physical server is the server that stores and persists DataCapsules, an example would be a server provisioned by IBM. A DataCapsule owner is the owner of a DataCapsule, who possesses and protects the DataCapsule's private key as well as data encryption key. A DataCapsule reader is a reader of a DataCapsule, who is authorized to possess data decryption key.

A private/public key pair will be generated for each party mentioned above, except for DataCapsule readers. Public keys are accessible by anyone for the purpose of verification, and private keys are stored and protected by each entity. Data encryption key is possessed by DataCapsule writer, while data decryption key is possessed by authorized DataCapsule readers. The type of asymmetric keys generated depends on implementation and is indifferent for the purpose of this report. Also, the processes of key generation and key distribution are assumed to be done out of band, and will not be addressed in this report.

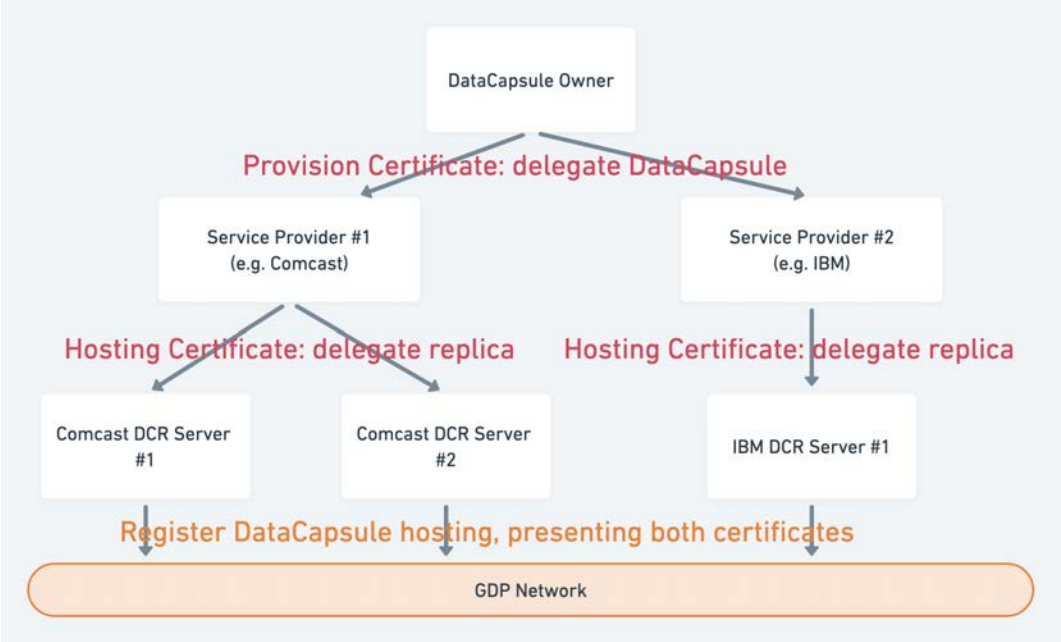

Figure 7: Delegation Certificates in the DCR system.

4.7.3 Delegation Certificates

Delegations are used to establish trusted relationships among these parties. A DataCapsule owner needs service providers to provision DataCapsule replicas, and a service provider needs physical servers to physically store and host DataCapsule replicas. For a service provider to provision DataCapsule replicas for the DataCapsule's owner, the DataCapsule owner needs to grant the service provider a Provision Certificate. For a service provider's physical server to host a DataCapsule replica, the service provider needs to grant the physical server a Hosting Certificate. As shown in Figure 7 , for a physical server to register its hosted DataCapsule in the GDP network, it needs to prove both its identity and its authorization to host, which requires showing both the Provision Certificate and the Hosting Certificate.

4.7.4 Provision Certificate

In the DCR system, a DataCapsule owner pays service providers to provision DataCapsule replicas on behalf of the owner. A Provision Certificate is a cryptographic delegation certificate from a DataCapsule owner to a service provider (e.g., IBM) to authorize it to provision DataCapsule replicas. In particular, a Provision Certificate will contain the DataCapsule's GDP name, the service provider's GDP name, and then signed by the DataCapsule's ECDSA signing key. After receiving the certificate, the service provider will store the certificate locally as well as forward it to its physical servers that are responsible for hosting a replica. When a physical server registers a DataCapsule replica with the GDP network, it will present the Provision Certificate, along with Hosting Certificate, to prove its identity. For further security, a Provision Certificate has an expiration time, and can be renewed periodically.

4.7.5 Hosting Certificate

For a service provider to provision DataCapsule replicas, it requires its physical servers to physically store and host each replica. A Hosting Certificate is a cryptographic delegation certificate from a service provider to its physical server to authorize it to host a specific DataCapsule replica. A Hosting Certificate proves the ownership relationship between a service provider and a physical server that it owns. In particular, a Hosting Certificate will contain both the service provider's GDP name and the physical server's GDP name, signed by the service provider's ECDSA signing key. After receiving the Hosting Certificate, a physical server will store it and present it to the GDP network whenever verification is needed.

When the GDP network receives a registration request from a physical server, it will receive a Provision Certificate and a Hosting Certificate. The GDP network will verify both certificates through the DataCapsule's public key and the service provider's public key, respectively. If successful, the GDP network will register the physical server to be hosting a DataCapsule replica, and then forward corresponding client requests to it in the future.

5 Optimization

In this section, I discuss system optimizations that improve operation latency and throughput, while maintaining the same level of security.

5.1 Proxy Server

In the baseline design, a client receives acknowledgement messages from all delegated DCR servers. In cases where the number of replicas is large, this can be a burden on clients' networking and computing resources. To alleviate that, we introduce a proxy that collects

acks from DCR servers and then sends an aggregated ack to the client. Since we assume DCR servers and the proxy to be untrusted infrastructures, we need a signature scheme that does not rely on trust among DCR servers, the proxy, and the client. We exploit Threshold Signature, a signature scheme that allows an aggregated ack to verifiably prove a quorum of DCR servers have persisted the record.

5.1.1 Threshold Signature Scheme (TSS)

A Threshold Signature Scheme [4] is a cryptographic signature scheme that allows multiple parties to establish a group, and allows a quorum of parties to sign on behalf of the group. Each party is considered an individual signer, which generates its own private key, and has the capability to sign a share of a message. To sign a message on behalf of the group, TSS requires at least m of n shares of signatures to be available. When m shares all sign a message, a single threshold signature is generated, and it means the group has signed the message. For any verifier, it can simply verify the threshold signature using a public key to know a quorum of parties have signed the message.

In the context of the DCR system, a DCR server is an individual signer. Each DCR server has their own private key and can sign a share of the ack after persisting a record. The proxy has the responsibility of collecting a quorum, m, of signatures from DCR servers, then generating a threshold signature on the ack. After that, the proxy will send the threshold signature with ack to the client, which will then verify the threshold signature received. By using the threshold signature, the client can verify only one ack to know that the record has been persisted by a quorum of DCR servers. TSS reduces clients' verification cost from linear to constant, saving them from receiving and verifying potentially tens or hundreds of acks.

5.2 Trusted In-Enclave Proxy Server

In the federated storage model, the network and infrastructure are untrusted. Therefore, DCR servers need to verify every write received. If we can ensure that sender is the DataCapsule writer in another way, then DCR servers do not need to verify signatures on every write. Similarly, if the client can ensure that the proxy is collecting acks properly, then the client does not need to verify ack signatures. We introduce in-enclave proxy server, a trusted proxy server that runs inside an enclave. An enclave is a secure area of a main processor which is isolated from other processes running in the system. It protects code and data running inside the enclave, and allows third parties to attest and verify that the code is indeed running inside an enclave. Therefore, any third entity can trust the in-enclave proxy with secrets.

5.2.1 Hash-Based Message Authentication Code (HMAC)

With a trusted in-enclave proxy, third parties can establish a HMAC [3] channel for efficient, secure, and verifiable communication. HMAC is a crypographic authentication scheme that enables authentication and data integrity in a conversation, without the need for signatures and asymmetric cryptography on every write.

Our threat model in Section 3 mentions that infrastructures, including proxies, are untrusted. Thus, HMAC channels with non-enclave proxy does not alleviate the burden on signatures because the server would have to verify signatures on every write. On the other hand, Hash-based Message Authentication Code (HMAC) becomes a more efficient option for an in-enclave proxy and thus can be trusted.

For the write workflow from a client to DCR servers, we establish an HMAC channel for each of the client-server pair, with the same symmetric key. As a result, a client can simply send the write to the GDP network, which will multicast the write to all delegated DCR servers. Since they have the same symmetric key, all DCR servers can properly handle the write.

For the write ack workflow from DCR servers to the client, we establish an HMAC channel for each of the server-proxy pair, with different symmetric keys so the proxy can verify the ack's identity; and an HMAC channel for the proxy-client pair to reduce signature traffic and load on the proxy. After receiving acks from a quorum of DCR servers, the proxy sends an aggregated ack to the client.

As a result, DCR servers do not need to verify the signature on every write for provenance authentication check. Instead, each DCR server can simply store the trusted record locally, and generate an ack. Since there is an HMAC channel with the in-enclave proxy, the DCR server does not need to sign the ack. However, the DCR server still needs to verify the signatures on data, since HMAC cannot be given to anyone else. Thus, any third party in the future needs the signature to verify data integrity and provenance in the future.

By employing HMAC channels, a client can save the cost of signing and verifying for each write request. DCR servers also do not have to verify every write, or signing ack messages. This design improves write latency and saves compute resources.

5.2.2 Periodic Signatures and Read Proofs

If there is no signature on records at all, even though a DCR server can safely assume every write request comes from the DataCapsule writer, it is not able to prove it to others, which undermines provenance. We introduce periodic signature verification, where a DCR server checks signature every several writes as shown in Figure 8. Let's say the last verified record is $R100$ and the second last one is $R103$. For records between $R100$ and $R103$ (i.e., $R101$ and $R102$), they will be marked verified whenever $R103$ is marked verified.

To serve read requests, only verified records can be returned, and the DCR server needs to generate a proof using a signature of a later record and a hash chain of records including

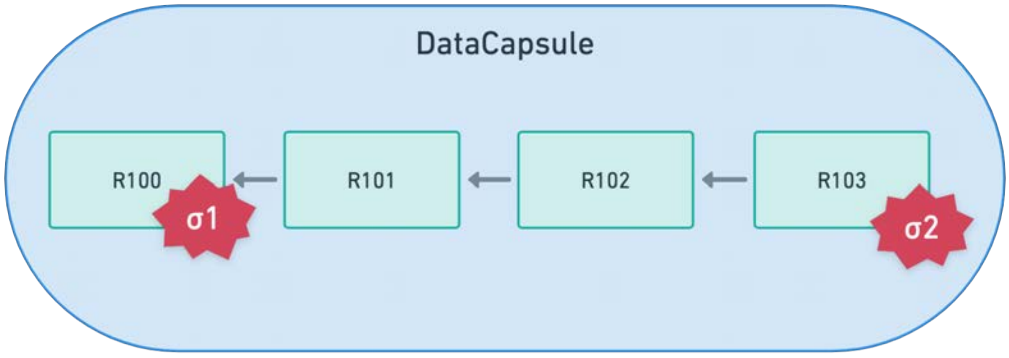

Figure 8: DataCapsule structure with periodic signatures.

the returned record. For a record with a signature, such as $R103$ in Figure 8, it can be simply returned because the reader can verify its signature. For a verified record without a signature, such as $R101$ in Figure 8, a cryptographic proof is generated. In particular, the proof consists of the signatures before and after the requested record, which are $\sigma 1$ and $\sigma 2$, along with the hash chain between $R103$ and $R100$. With these two signatures and the hash chain, a reader can first verify that $R100$ and $R103$ are valid, and then verify the hash chain is properly constructed, thus all records in the hash chain are valid. Further, during the period after $R101$ and $R102$ have been written but before $R103$ arrives, the server cannot serve $R101$ or $R102$ to readers, since there is no way to prove that they are valid. In fact, if the writer were to crash before writing $R103$, the unverified data in $R101$ and $R102$ would ultimately have to be discarded. Thus, the HMAC scheme would ultimately involve a negotiation between writers and servers as to the amount of data wrtiers are allowed to write without signing.

6 Implementation

The DataCapsule Replication System has a codebase of 3,500 lines of code in C++ excluding third-party networking and storage components as well as experiment scripts. All hash functions are SHA-256; symmetric encryption/decryption schemes are AES [23]; asymmetric signature schemes are ECDSA [13]. All cryptographic components mentioned above are based on OpenSSL [24] implementation, an open-source general-purpose cryptography library. In-enclave components are implemented on top of Open Enclave [12], which provides agnostic enclave platform support. In-enclave components of our experiments are run on Intel Software Guard Extensions (SGX1).

6.1 Storage

A DataCapsule record is the base unit of data transfer between DCR servers and clients. Each DataCapsule record is structured and transmitted using Protocol Buffers [26], a cross-platform data format used to serialize data. Each DataCapsule record has several fields, including an encrypted payload that stores application-level data, a parent hash, a signature, and the sender address. Every DCR server hosts one or more DataCapsules on local disk. Each DataCapsule is stored in a Key-Value Store, with each record's hash being the key, and the complete record being the value. Specifically, we use RocksDB [11], a log-structured merge-tree embedded KVS. We use RocksDB because it is an easy-to-use embedded KVS that integrates easily to the DCR system.

6.2 Networking

ZeroMQ (ZMQ) [28], an embedded networking library, is used to implement networking components. All unicast communications are done using TCP in ZMQ. For multicast, since GDP routing is under development, we simulate it using multiple unicasts.

6.3 Writes

We have implemented DataCapsule writers to send write requests to DCR servers. For each write, a DC client creates a record with an encrypted payload, its address, the hash on the record, and a parent hash. The client will then sign the record using its ECDSA signing key, serialize it to a string, and send it to all DCR servers via ZMQ simulated multicast. Specifically, the write API has the format of $put(hash, record)$. When a DCR server receives a write request, it will deserialize the string, verify its signature, store it locally in the corresponding DataCapsule KVS, which is a KVS that stores the DataCapsule for storing and retrieving records. The DCR server will then update its local DAG digest, including sinks and sources. After that, the DCR server will generate an ack, sign it, and send it back to the client via ZMQ unicast. When the client receives a quorum of verified acks, the record is considered durable.

6.4 Reads

For each read, a DC client creates a read request on a hash, a DataCapsule name, and its address. It then serializes it to a string, and sends it to a DCR server. Specifically, the hash-based read API has the format of $get(hash)$. When a DCR server receives a read request, it simply fetches the corresponding record from its local DataCapsule KVS using the given DataCapsule name and record hash. If the DCR server does not have the record, it will fetch from another delegated DCR server, verify it, and then return it to the client. After receiving the record, the client will verify its signature, decrypt payload with its AES

key, and then use it. Similarly, a client can send a freshness request to $f+1$ DCR servers, in the format of $get_fresh()$, and receive $f+1$ sets of latest record hashes for every branch.

6.5 Failure Recovery

After a DCR server joins the network, it periodically initiates pairing with other DataCapsule replicas. When a DCR server initiates a DataCapsule pairing, it first locks the DataCapsule KVS to ensure no race condition happens between new records and pairing response. It then randomly selects another DataCapsule replica among registered replicas. It will sign and send over its DAG digest, which includes all sources and sinks. When the requestee receives the pairing request, it verifies the request and then generate a set of records to return. After that, the requester restores missing records by storing them into the DataCapsule KVS, and unlock the KVS.

6.6 In-Enclave Server

An in-enclave proxy is implemented using Open Enclave on top of Intel SGX. Each DCR server initiates an HMAC channel with the in-enclave proxy using the shared secret key. Periodic signature verification is done by the writer periodically signing records, and the server verifies signatures whenever there is a signature. It is up to the writer to decide how frequently they send signatures.

7 Evaluation

7.1 Experiment Setup

We evaluate DCR servers on Azure Standard_D4s_v3 instances, which run on the Intel(R) Xeon(R) CPU E5-2673 v3 @ 2.40GHz with 4 virtual CPUs and 16GB memory. It has maximum cache and temporary storage throughput of 8000/64 IOPS/MBps and expected network bandwidth of 2000 Mbps. The machine uses Ubuntu 20.04.5 LTS 64bit with Linux 5.15.0-1021-azure. We evaluate the DCR in-enclave proxy on Azure Standard_DC2s_v3 instances, which run on the Intel(R) Xeon(R) Platinum 8370C CPU @ 2.80GHz with 2 virtual CPUs and 16GB memory. It supports Intel SGX2 and has encrypted memory (EPC) of 8GB. The machine uses Ubuntu 20.04.5 LTS 64bit with Linux 5.15.0-1021-azure. We run DCR client on Azure Standard_DS1_v2 instances, which run on Intel(R) Xeon(R) CPU E5-2673 v3 @ 2.40GHz with 1 virtual CPU, 3.5GB memory, and network bandwidth of 20Mbps. We run OpenEnclave version v0.18.0. We report the average of experiments that are conducted 5 times.

7.2 Proxy Server Evaluation

7.2.1 Benchmark Design

We evaluate DataCapsule Replication system's performance using Yahoo! Cloud Serving Benchmark (YCSB) [8] workloads. YCSB is an open-source specification used to evaluate performance of computer programs. YCSB generates traces of writes and reads to key-value pairs, it is the standard performance benchmark. For YCSB workloads to work with the DCR system, the client uses a hash table to store the mapping from a key to its most recent record hash. When the client writes to a key, it generates a record and stores the mapping from the key to the record's hash in the hash table. When the client reads from a key, it finds its corresponding hash, and sends a hash-based read to the DCR system.

We use four workloads generated by YCSB traces: write-only, write-heavy, read-heavy, and read-only. In particular, workload A is write-only, where 100% of operations are writes; workload B is write-heavy with 50:50 reads to writes; workload C is read-heavy with 95:5 reads to writes; and, workload D is read-only.

We have three levels of optimizations: base case, collector proxy, and in-enclave proxy. For the base case write, a writer signs every write record and sends it to all DCR servers via GDP network anycasts. A DCR server receives it, verifies signature, stores it, and sends an ack directly back to the writer. After that, the writer receives and verifies acks from all DCR servers, and can mark the record replicated. A base case read is to send a hash-based read to a DCR server, which will send back the record.

For the collector proxy, a writer still signs every write record and then multicasts it to all DCR servers via GDP multicast. When a DCR server receives a record, it verifies signature, stores it, and sends an ack to a collector proxy. This collector proxy will receive acks from a quorum of DCR servers, then signs a threshold signature on the ack, and send back to the client. The client will verify the threshold signature and mark the record replicated. Reads are the same as in the base case.

For in-enclave proxy, an HMAC channel is established between the proxy and each of writers and DCR servers. Then, a writer sends, via HMAC channel, write records to the in-enclave proxy, which will forward the record to DCR servers via HMAC. DCR servers store the record and sends an ack via HMAC back to the in-enclave proxy. The in-enclave proxy collects a quorum of acks and then send one aggregated ack back to the client. Then, the client will mark the record replicated. Again, reads are the same as in the base case.

We measure three sets of metrics, write latency, read latency, and throughput. We measure for three levels of optimizations and two payload sizes, 16 bytes and 64KB. For throughput, we measure the number of operations per second with 5 replicas and a quorum of 3 for each of the YCSB workloads above. For operation latency, we measure the time between a read or write request is sent, and a record or ack is received, respectively.

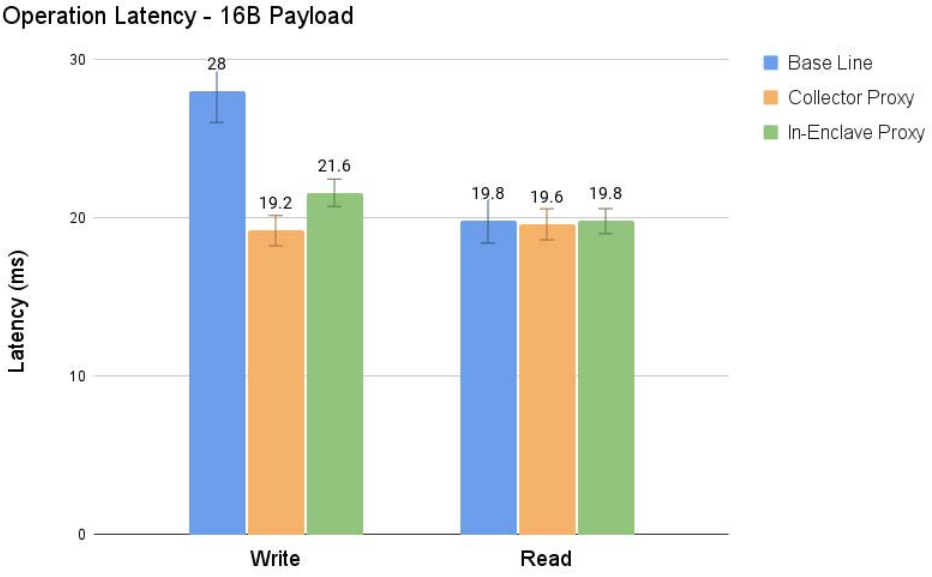

Figure 9: Latency for one write and one read under three levels of optimizations with 16B payloads.

7.2.2 Overall Performance - Operation Latency

Write Latency For write latency, we evaluate the time between when a client sends a write request to five replicas and receives a quorum of 3 ack messages. We evaluate write latency with a small payload of 16B and a large payload of 64KB. In the situation of a 16B payload, cryptographic operations and system overheads are the main factors for latency. From Figure 9, we see that the base case has a higher write latency because the client has limited computation power and network bandwidth. Both collector proxy and in-enclave proxy have a lower latency, which is because the proxy has a better network bandwidth and reduces the latency to multicast. In-enclave proxy has a slighter higher latency than collector proxy because of enclave's system overheads, including ocalls and ecalls.

In the situation of a 64Kb payload, network bandwidth's impact becomes the significant factor. As a result, we see that the base case has even higher latency as shown in Figure 10. In the base line case, a client uses multiple anycasts to send writes to the DCR servers, which becomes the bottleneck on the client with limited networking resources. As a result, the latency is significantly higher than the proxy cases. On the other hand, both the collector proxy case and in-enclave proxy case use multicast, which significantly reduces the networking bottleneck on the client, resulting in lower write latencies.

Read Latency For read latency, we evaluate the time between a client sends a hash-

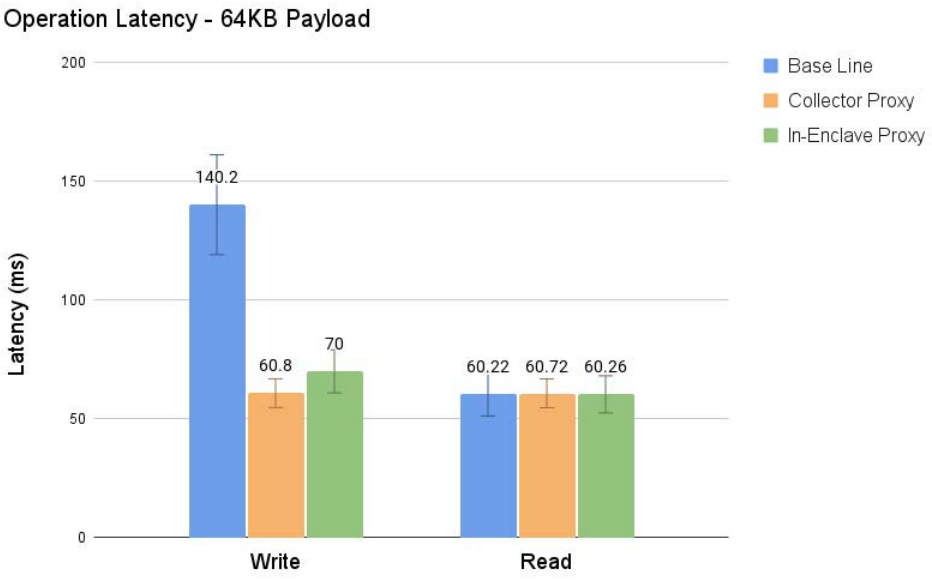

Figure 10: Latency for one write and one read under three levels of optimizations with 64KB payloads.

based read request and receives the DataCapsule record. From Figure 9 and 10, we see that all three optimizations have similar levels of read latency under both payloads. This is because read workflows are identical in three optimizations.

7.2.3 Overall Performance - Throughput

Throughput For throughput, we evaluate four workloads: write-only, write-heavy, read-heavy, and read-only. Similar to operation latency, we evaluate with two payload sizes: 16B and 64KB. With 16B payloads, cryptographic operations and system overheads are bottlenecks. From Figure 11, we see that for 16B payload write-only workloads, the base line has a lower throughput because of its limited computation power and network bandwidth. In-enclave proxy has a higher throughput than collector proxy because it utilizes HMAC channels to reduce cryptographic operation costs compared to collector proxy. For write-heavy and read-heavy workloads, in-enclave proxy optimization has the highest throughputs. It is because HMAC channels significantly reduces computation workloads on DCR servers, allowing them to provide more resource to serve read requests. On the other hand, the base line throughput is limited by its computation power and network bandwidth. For read-only workload, throughputs are the same for three operations because they have the same workflows.

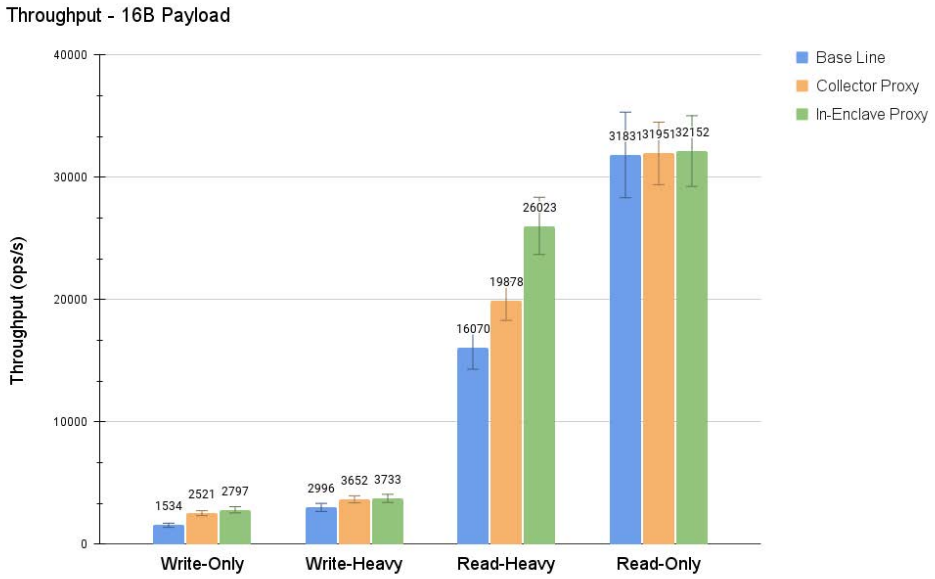

Figure 11: Throughput for three levels of optimizations with 16B payloads.

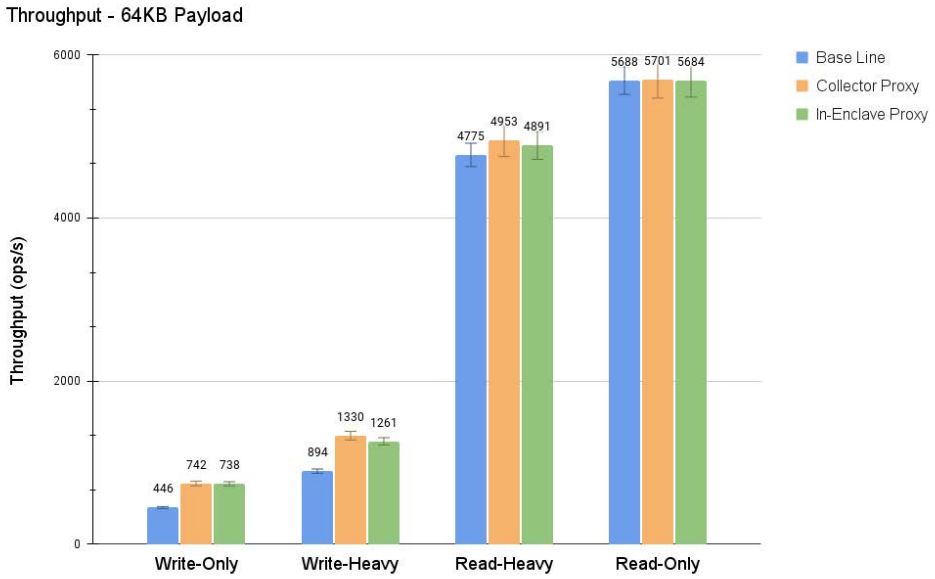

Figure 12: Throughput for three levels of optimizations with 64KB payloads.

For large payloads of 64KB, networking overheads are more significant. From Figure 12, we see that the collector proxy and in-enclave proxy have similar throughputs for both write-only, write-heavy, and read-heavy workloads, it is because both of them experience the same amount of networking overheads, while the additional system overheads from collector proxy slightly reduces throughput. The base line has lower throughputs because the client has limited network bandwidth. Again, read-only workloads have the same throughputs.

7.3 Anti-Entropy Evaluation

7.3.1 Benchmark Design

We evaluate DataCapsule Replication system's failure recovery performance. In particular, we compare between a naive anti-entropy design and the DAG-based anti-entropy design. We measure the time between a pairing request is sent and the two replicas are fully synced up under three situations: one complete replica and one empty replica; one complete replica and one almost complete replica; and two complete replicas. Again, we evaluate under two payload sizes of 16B and 64KB.

For one complete and one almost complete replica case, the almost complete replica is randomly generated by creating a DataCapsule of 1000, 10000, 20000, and 30000 records, with a random number of branches between 1 and 5. After that, 1% of the records are randomly removed.

7.3.2 Overall Performance

From both Figure 13 and Figure 14, we see that DAG-based anti-entropy is more efficient and scalable in both two complete replicas and one almost complete replica cases. In particular, for two complete replicas case, the naive design requires exchanging the hashes of all records in the DataCapulse, while DAG-based design only requires exchanging sinks and sources, which are independent from the size of the DataCapsule. As shown in the figures, as the number of records increases, the latency for DAG-based design remains the same or increases slightly, while the base line latency increases significantly. The compute latency is consistent at around 10ms for all cases because the DAG digest, including sinks and sources, is cached. And the rest of the latency is networking latency. At 30000 records, the DAG-based design's latency is 76% and 63% lower than the base line in two complete and one almost complete cases, respectively. Lastly, when a replica is completely empty, two designs spent the same amount of time to sync up as shown in 15. This is because the main bottleneck is record transfer overhead from the complete replica to the empty replica. Overall, the DAG-based design is significantly more efficient in the most common cases, where two replicas are complete or almost complete.

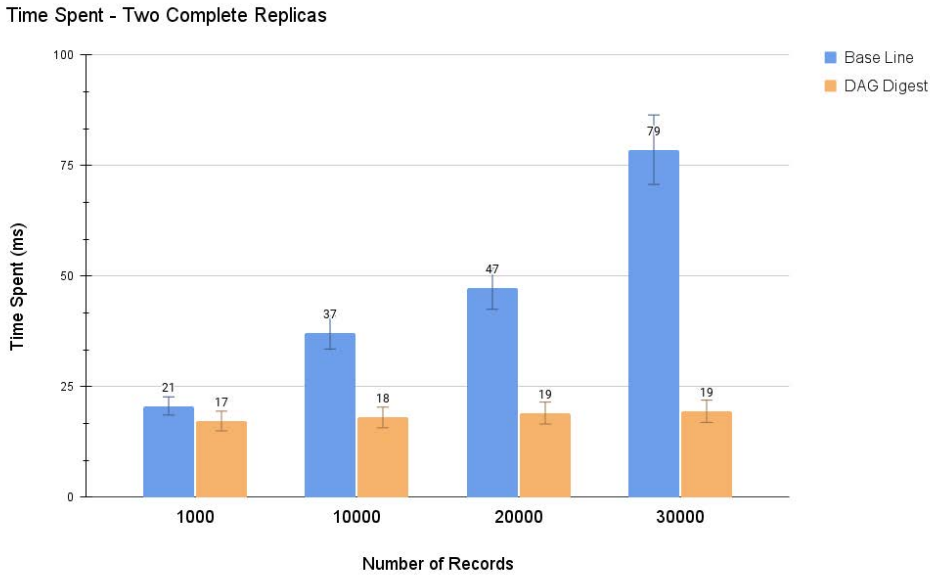

Figure 13: Time spent for two complete replicas.

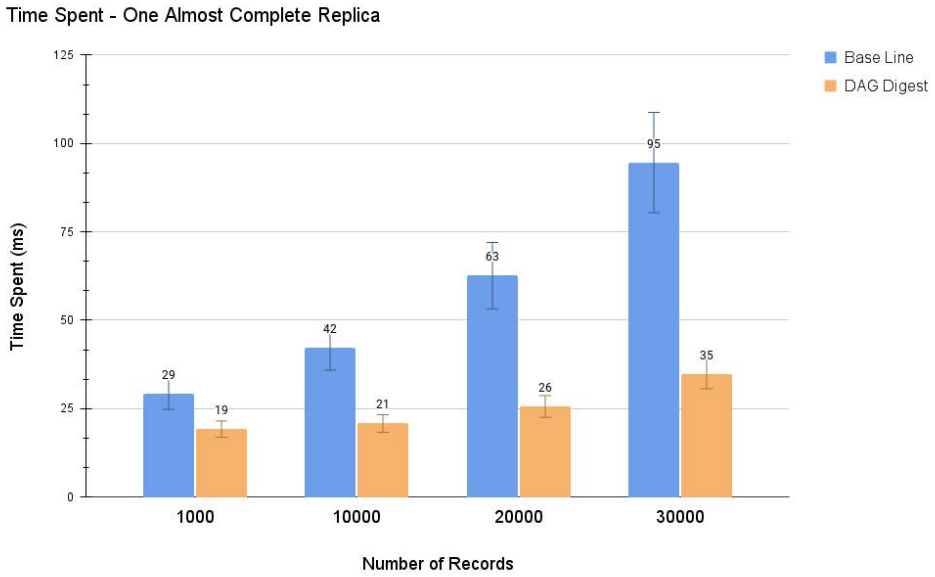

Figure 14: Time spent for one almost complete replica and one complete replica.

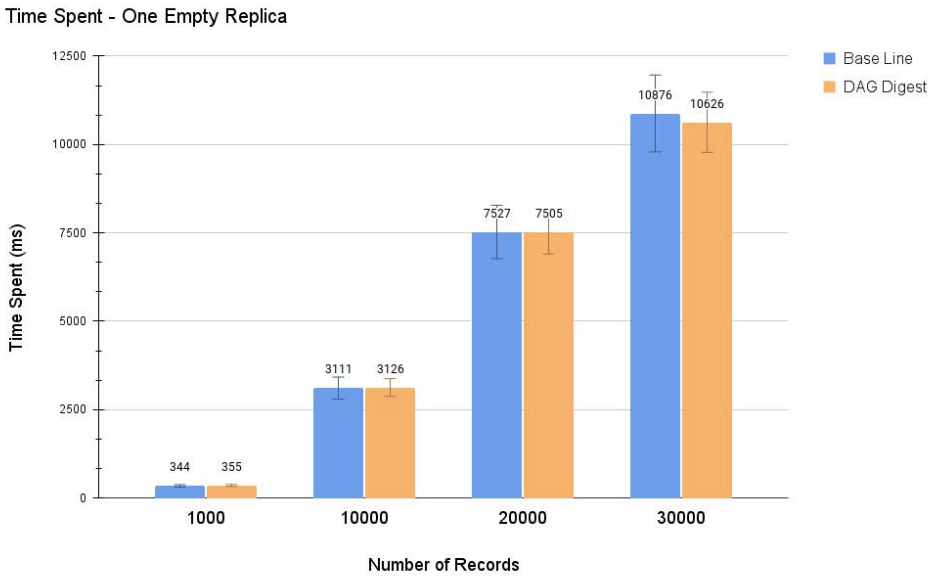

Figure 15: Time spent for one empty replica and one complete replica.

8 Future Work

8.1 Multi-Writer with Bitcoin Wallet

In the report, the DCR system is under the assumption of one writer identity, that means a unique private writer key. In the future, we aim to expand the DCR system to multiple writer identities, which means multiple private writer keys with mechanisms to assign new writer keys and revoke existing writer keys. A challenge of having multiple writer identities is to timely access the most updated information, which is likely to be stored in a DataCapsule. Given a DCR server currently does not have access to decrypted data in a DataCapsule, a new mechanism needs to be designed and built. One preliminary design uses Bitcoin Wallet [1], where a root private/public writer key pair can be used to derive and verify subsequent writer keys. In particular, the root public writer key can be included in the metadata of a DataCapsule, and then DCR servers can subsequently derive the next generation's public keys using it. On the other hand, writers can derive their next generation's private writer key, sign the record using it, and then send the signed record to DCR servers. DCR servers will have the information on which generation to derive the public key with, and then validate the signature. Subsequently, the DCR servers can store the signed records as usual.

8.2 DataCapsule In Transit

In this report, we discuss DataCapsules that are fully loaded in a DCR server, which uses indexing and caching to enable efficient operations. Currently, the effort required to fully load a DataCapsule requires parsing through the DataCapsule, generates a hash table from hashes to records, and creates a table of sinks and sources. This loading mechanism requires a non-trivial computation power to accomplish, which can be optimized. In the future, if we want to transfer a DataCapsule from one DCR server to another, the DataCapsule needs to be transformed to another form. In particular, compaction may be required to transfer a DataCapsule faster. In addition, how to efficiently compact and then load from a compacted form are also important. Thus, a future work is to design and build a mechanism that can efficiently compact and serialize a fully loaded DataCapsule, transfer it to another DCR server, and also enable efficient loading and indexing from the compact form.

8.3 In-Enclave Proxy Optimization

As shown in Section 7, in-enclave proxy optimization can efficiently alleviate computation bottlenecks on DCR servers. A limit of this design is the system overheads of enclave ecalls and ocalls, which are relatively expensive, and could offset benefits to a certain extent. In the future, we aim to reduce this system overhead. In particular, we can use mechanisms to

improve communications between an in-enclave process and an out-enclave process. One example is that switchless calls can be employed to reduce system overheads.

8.4 Freshness Service Server

As shown in Section 4.5, the current design requires the client to receive at least $f + 1$ sets of record hashes from $f + 1$ DCR servers. This requires the client to determine the latest set of record hashes by comparing the timestamps. This design relies on timestamps, which may not be available for applications other than PSL. Therefore, a future work is to design a freshness service server that determines the latest record on behalf of the client, and then send one set of latest records to the client.

9 Conclusion

We introduced DataCapsule Replication System, a storage system that provides continuous, persistent, replicated, and secure storage for DataCapsules. We focus on the security, replication, and efficiency aspects of the DCR system. It utilizes multicast from GDP network to optimize network efficiency, employs in-enclave proxy to alleviate networking pressure on DCR clients, uses HMAC channels to reduce cryptographic operation overheads on clients and servers, while maintaining the same level of security guarantees of data integrity, confidentiality, and provenance. It also uses a novel DAG-based anti-entropy mechanism to enable epidemic, pair-wise synchronization to handle failures. In our benchmarks, DCR has shown great performance optimizations from proxies, where throughput is at most 62% higher than the baseline. Also, the efficient failure recovery mechanism optimizes pairing latency to be at most 76% lower than the baseline. In addition, the DCR system is in use by the CapsuleDB project and serves as its persistence solution. In conclusion, the DCR system provides a great replication solution to applications such as GDP and PSL.

10 References

[1] Andreas M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. 1st. O'Reilly Media, Inc., 2014. ISBN: 1449374042.

[2] AWS. *AWS Lambda*. URL: https://aws.amazon.com/lambda/.

[3] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying Hash Functions for Message Authentication". In: *Advances in Cryptology — CRYPTO '96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–15. ISBN: 978-3-540-68697-2.

[4] Gerrit Bleumer. "Threshold Signature". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg. Boston, MA: Springer US, 2005, pp. 611–614. ISBN: 978-0-387-23483-0. DOI: 10.1007/0-387-23483-7_429. URL: https://doi.org/10.1007/0-387-23483-7_429.

[5] Flavio Bonomi et al. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 13–16. ISBN: 9781450315197. DOI: 10.1145/2342509.2342513. URL: https://doi.org/10.1145/2342509.2342513.

[6] Eric Chen et al. *SCL: A Secure Concurrency Layer For Paranoid Stateful Lambdas*. Tech. rep. UCB/EECS-2022-232. EECS Department, University of California, Berkeley, Oct. 2022. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-232.html.

[7] Sanchuan Chen et al. "Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '17. Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2017, pp. 7–18. ISBN: 9781450349444. DOI: 10.1145/3052973.3053007. URL: https://doi.org/10.1145/3052973.3053007.

[8] Brian Frank Cooper. "Yahoo! cloud serving benchmark". In: (). URL: https://github.com/%20brianfrankcooper/YCSB.

[9] Intel Corporation. *Intel(R) Software Guard Extensions SDK for Linux* OS*. https://download.01.org/intel-sgx/linux-1.8/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.8_Open_Source.pdf. 2017.

[10] Victor Costan and Srinivas Devadas. "Intel SGX Explained." In: *IACR Cryptol. ePrint Arch.* 2016.86 (2016), pp. 1–118.

[11] Siying Dong et al. "RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications". In: *ACM Trans. Storage* 17.4 (Oct. 2021). ISSN: 1553-3077. DOI: 10.1145/3483840. URL: https://doi.org/10.1145/3483840.

[12] Open Enclave. "Open Enclave: hardware-agnostic open source library for developing applications that utilize Hardware-based Trusted Execution Environments". In: (). URL: https://openenclave.io/sdk/.

[13] Don Johnson, Alfred Menezes, and Scott Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". In: *Int. J. Inf. Secur.* 1.1 (Aug. 2001), pp. 36–63. ISSN: 1615-5262. DOI: 10.1007/s102070100002. URL: https://doi.org/10.1007/s102070100002.

[14] Ralph C. Merkle. "A Certified Digital Signature". In: *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*. Vol. 435. Lecture Notes in Computer Science. Springer, 1989, pp. 218–238. DOI: 10.1007/0-387-34805-0_21.

[15] Nitesh Mor. "Global Data Plane: A Widely Distributed Storage and Communication Infrastructure". PhD thesis. EECS Department, University of California, Berkeley, Jan. 2020. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-10.html.

[16] William Mullen. "CapsuleDB: A Secure Key-Value Store for the Global Data Plane". MA thesis. EECS Department, University of California, Berkeley, May 2022. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-168.html.

[17] Patrick O'Neil et al. "The Log-Structured Merge-Tree (LSM-Tree)". In: *Acta Inf.* 33.4 (June 1996), pp. 351–385. ISSN: 0001-5903. DOI: 10.1007/s002360050048. URL: https://doi.org/10.1007/s002360050048.

[18] Oleksii Oleksenko et al. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 227–240. ISBN: ISBN 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/oleksenko.

[19] Marc Shapiro et al. "Conflict-Free Replicated Data Types". In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 386–400. ISBN: 9783642245497.

[20] Weisong Shi et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

[21] Ming-Wei Shih et al. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". In: Jan. 2017. DOI: 10.14722/ndss.2017.23193.

[22] Shweta Shinde et al. "Preventing Page Faults from Telling Your Secrets". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '16. Xi'an, China: Association for Computing Machinery, 2016, pp. 317–328. ISBN: 9781450342339. DOI: 10.1145/2897845.2897885. URL: https://doi.org/10.1145/2897845.2897885.

[23] Information Technology Laboratory (National Institute of Standards and Technology). *Announcing the Advanced Encryption Standard (AES) [electronic resource].* English. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, MD, 2001, 52 p. :

[24] The OpenSSL Project. "OpenSSL: The Open Source toolkit for SSL/TLS". www.openssl.org. Apr. 2003.

[25] K. Thulasiraman and M. N. S. Swamy. *Graphs: Theory and Algorithms.* USA: John Wiley amp; Sons, Inc., 1992. ISBN: 0471513563.

[26] Kenton Varda. *Protocol Buffers: Google's Data Interchange Format.* Tech. rep. Google, June 2008. URL: http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html.

[27] Yinhao Xiao et al. "Edge Computing Security: State of the Art and Challenges". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1608–1631. DOI: 10.1109/JPROC.2019.2918437.

[28] ZeroMQ. "Zeromq: An open-source universal messaging library". In: (). URL: https://zeromq.%20org/.

[29] Ben Zhang et al. "The Cloud is Not Enough: Saving Iot from the Cloud". In: *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing.* HotCloud'15. Santa Clara, CA: USENIX Association, 2015, p. 21.