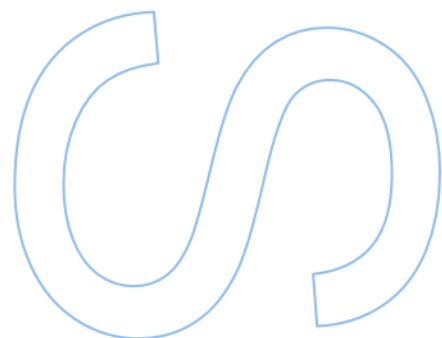
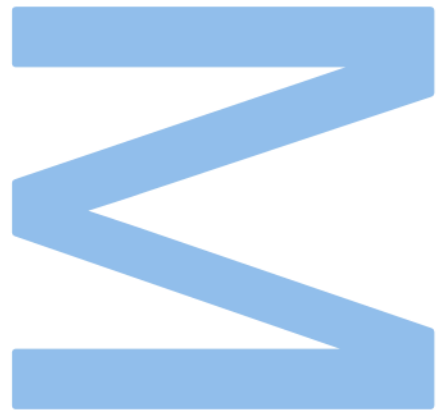


Secure Conflict-Free Replicated Data Types via Shamir Secret Sharing

Ricardo Araújo Amorim
Master in Information Security
Department of Computer Science
Faculty of Sciences of the University of Porto
2026



Secure Conflict-Free Replicated Data Types via Shamir Secret Sharing

Ricardo Araújo Amorim

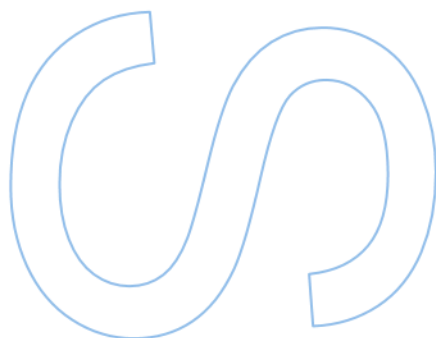
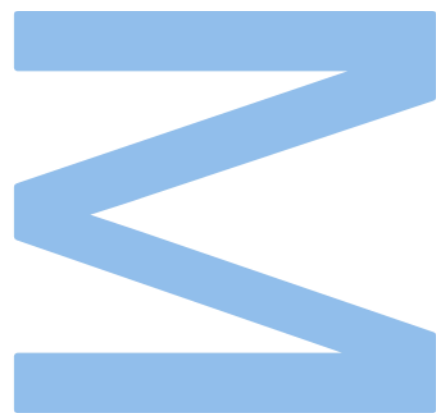
Dissertation carried out as part of the Master in Information Security
Department of Computer Science
2026

Supervisor

Bernardo Portela, Assistant Professor, Faculty of Sciences, University of Porto

Co-supervisor

Hugo Pacheco, Assistant Professor, University of Minho



“ A dream may set the destination, but the journey is defined by the people we choose to sail with, the kindness we show, and the memories we allow ourselves to make along the way. Perhaps only then are we truly ready to find the treasure. ”

Personal reflection inspired by *One Piece*

Acknowledgements

I would like to express my sincere gratitude to everyone who supported me throughout this journey.

To Professor Bernardo and Professor Hugo, thank you for your guidance, availability, valuable feedback, and support during the development of this dissertation. Your knowledge, advice, and encouragement were invaluable, and I am truly grateful for the opportunity to have worked with both of you and learned so much from you.

To my girlfriend, Ana, thank you for being such an important part of my life and academic journey over the past seven years, from the 11th grade until the completion of my master's degree. Thank you for always being by my side as my friend, my companion, and the best girlfriend anyone could ask for. Your patience, love, support, and belief in me, especially during the most difficult moments, meant more than words can express.

To my father, Alberto, and my mother, Teresa, the person I am today is only possible because of the hard work and sacrifices you made to give me opportunities you never had. I could not have asked for better parents or role models. Thank you for your love, support, lectures, and life lessons, which made this achievement possible.

To my little sister, Rita, thank you for being the greatest sister anyone could ask for. Your love, joy, and happiness for my achievements mean so much to me. I hope my journey motivates you and shows you that it is possible to pursue what you love, even when the path seems uncertain.

To my grandmother, Teresa, thank you for all the love, care, and concern you have always shown me. To my grandfather, Manuel, thank you for the games of Sueca on Sundays, the long conversations, and the stories from the past. Those moments will always hold a very special place in my heart.

To the rest of my family — my godfather, my godmother, my aunts, uncles, and cousins — and to Ana's family, who over the years have also become my family, thank you for all the love, motivation, and joy you have shown throughout the pursuit of my goals. Your support means a great deal to me.

To my friends, thank you for your companionship, encouragement, laughter, conversations, advice, and all the moments shared along the way. Your presence made this path lighter, more enjoyable, and even more special.

Resumo

As *Conflict-free Replicated Data Views* (CRDV) fornecem uma *framework* para usar *Conflict-free Replicated Data Types* (CRDTs) em sistemas práticos de gestão de dados replicados, expondo estado replicado convergente através de tabelas relacionais e vistas SQL. Por se basearem em CRDTs, as CRDV preservam a disponibilidade e a convergência sob concorrência e falhas parciais. No entanto, muitos sistemas baseados em CRDTs são concebidos assumindo que as réplicas podem armazenar e processar o estado replicado em claro. A *framework* CRDV preserva esta assunção: os valores replicados são visíveis para as réplicas que os mantêm. Isto limita a sua aplicabilidade em cenários onde os dados são sensíveis ou onde as réplicas pertencem a diferentes domínios administrativos.

Esta tese explora como a computação multipartidária segura, ou *Secure Multi-Party Computation* (MPC), pode ser integrada com CRDV para suportar gestão confidencial de dados replicados. É proposta uma extensão segura de CRDV na qual os valores sensíveis são representados por *handles* na camada de dados replicada, enquanto o seu conteúdo é partilhado secretamente por um conjunto de partes MPC. Operações seguras, como inserção, reconstrução, comparação, multiplicação e redistribuição de partilhas, são executadas sobre estas partilhas, permitindo que as réplicas manipulem dados privados sem os armazenar diretamente em claro. O desenho considera também alterações no conjunto de réplicas ativas, redistribuindo as partilhas para as partes associadas às réplicas que poderão mais tarde aceder aos dados protegidos.

O sistema é implementado sobre a *framework* de programação coreográfica Chorus e avaliado através de *microbenchmarks* de operações seguras individuais e de um *macrobenchmark* baseado em cargas de trabalho de comunicação veicular. Os resultados mostram que a abordagem é viável e preserva a modularidade do modelo CRDV, ao mesmo tempo que torna explícitos os custos de desempenho introduzidos pela computação segura e pela redistribuição de partilhas.

Palavras-chave: *Conflict-free Replicated Data Types*, *Conflict-free Replicated Data Views*, *Secure Multi-Party Computation*, *Shamir Secret Sharing*, Reconfiguração, Replicação Confidencial

Abstract

Conflict-free Replicated Data Views (CRDV) provide a framework for using Conflict-free Replicated Data Types (CRDTs) in practical replicated data management systems, exposing convergent replicated state through relational tables and SQL views. By building on CRDTs, CRDV preserve availability and convergence under concurrency and partial failures. However, many CRDT-based systems are designed under the assumption that replicas may store and process replicated state in plaintext. The CRDV framework preserves this assumption: replicated values are visible to the replicas that maintain them. This limits its applicability in settings where data is sensitive or replicas belong to different administrative domains.

This thesis explores how secure multi-party computation (MPC) can be integrated with CRDV to support confidential replicated data management. It proposes a secure extension of CRDV in which sensitive values are represented by handles in the replicated data layer, while their contents are secret-shared across a set of MPC parties. Secure operations such as insertion, reconstruction, comparison, multiplication, and resharing are executed over these shares, allowing replicas to manipulate private data without directly storing it in plaintext. The design also considers changes in the set of active replicas by redistributing shares to the parties associated with the replicas that may later access the protected data.

The system is implemented on top of the Chorus choreographic programming framework and evaluated through microbenchmarks of individual secure operations and a macrobenchmark based on vehicular communication workloads. The results show that the approach is feasible and preserves the modularity of the CRDV model, while making explicit the performance costs introduced by secure computation and share redistribution.

Keywords: Conflict-free Replicated Data Types, Conflict-free Replicated Data Views, Secure Multi-Party Computation, Shamir Secret Sharing, Reconfiguration, Confidential Replication

Table of Contents

List of Tables	xiii
List of Figures.....	xv
1. Introduction	1
1.1. Problem Statement	2
1.2. Objectives	3
1.3. Document Structure	3
2. Preliminaries.....	5
2.1. Conflict-free Replicated Data Types.....	5
2.1.1. Types of CRDTs	6
2.1.2. Common CRDT Structures	7
2.2. Secure Multi-Party Computation.....	10
2.2.1. Secret Sharing-Based MPC.....	11
2.3. Shamir Secret Sharing.....	12
2.3.1. Secret Generation	13
2.3.2. Secret Reconstruction.....	14
3. Related Work.....	17
3.1. Conflict-free Replicated Data Views	17
3.1.1. Architecture	18
3.1.2. CRDT Representation.....	18
3.1.3. Limitations	21
3.2. Secure CRDTs	22
3.2.1. Encryption-Based Approaches	22
3.2.2. Access-Control-Based Approaches	24
3.2.3. MPC-Based Approaches	25
3.3. Discussion.....	26
4. Secure CRDV Architecture	29
4.1. Architectural Overview	29
4.1.1. CRDV Layer	31
4.1.2. Secure Layer.....	31
4.2. Replicas as Sets of MPC Participants.....	32
4.3. Security Assumptions and Threat Model.....	33
4.4. Trust Quorums	35

5.	Secure Operations	37
5.1.	Secure MPC Operations.....	37
5.1.1.	Operation Types.....	38
5.1.2.	Private Input and Opening	39
5.1.3.	Addition and Subtraction.....	39
5.1.4.	Resharing	41
5.1.5.	Multiplication	41
5.1.6.	Declassification	42
5.2.	Secure CRDV Operations	43
5.2.1.	Secure Counters	44
5.2.2.	Secure Registers	45
5.2.3.	Secure Maps	46
5.2.4.	Secure Lists	46
5.2.5.	Secure Sets.....	47
5.3.	System-level Secure Operations	51
5.3.1.	Share Propagation	51
5.3.2.	Quorum Reconfiguration.....	52
6.	Implementation	55
6.1.	Technology Stack.....	56
6.2.	CRDV Integration.....	57
6.2.1.	Handle Generation	59
6.3.	Secure Runtime Interface	61
6.4.	Choreographic MPC Implementation	63
6.5.	Implementation Challenges and Limitations	66
7.	Experimental Evaluation	69
7.1.	Environment.....	70
7.2.	Microbenchmarks	70
7.2.1.	Methodology and Metrics.....	71
7.2.2.	Operation Benchmarks	72
7.2.3.	Reconfiguration	73
7.2.4.	Results	73
7.2.5.	Discussion	76
7.3.	Macrobenchmarks	77
7.3.1.	Case Study Scenario	78
7.3.2.	DENM and IVIM Data Model	80
7.3.3.	Mapping to Secure CRDV	82
7.3.4.	Workload and Configuration	83
7.3.5.	Results	87

7.3.6. Discussion	92
8. Conclusion and Future Work.....	95
8.1. Future Work	96
References.....	99
A. Detailed Macrobenchmark Results	105
A.1. Time-Binned Operation Latency	105
A.2. Time-Series Results.....	105

List of Tables

3.1.	Representation of common CRDT abstractions in CRDV.	19
3.2.	Simplified replicated data relation for a LWW register.	19
3.3.	Comparison of secure CRDT approaches.	26
5.1.	Secure operation types provided by the secure layer.	38
6.1.	Concrete handle formats used by the prototype.	59
7.1.	Hardware used in the experimental evaluation.	70
7.2.	Configuration used for the operation microbenchmarks.	72
7.3.	Configuration used for the reconfiguration microbenchmarks.	73
7.4.	Latency of individual CRDV operations.	74
7.5.	Reconfiguration latency for different quorum transitions.	75
7.6.	Mapping between the case-study model and secure CRDV abstractions.	82
7.7.	Application-level operations used in the macrobenchmark.	84
7.8.	Macrobenchmark workload distribution.	84
7.9.	Injected replication latency.	85
7.10.	Injected MPC party latency model.	86
7.11.	Macrobenchmark configuration.	87
7.12.	Authorised regional MPC quorum policy used in the macrobenchmark.	87
7.13.	Macrobenchmark throughput.	88
7.14.	Executed operations per macrobenchmark scenario.	89
7.15.	Median latency of macrobenchmark operations.	89
A.1.	Baseline macrobenchmark latency evolution by operation.	105
A.2.	Latency-injected macrobenchmark latency evolution by operation.	105

List of Figures

2.1.	G-Counter example with two replicas.	7
2.2.	LWW register example.	8
2.3.	2P-Set example.	9
2.4.	CRDT map example.	9
2.5.	General overview of Secure MPC.	10
2.6.	Secret sharing-based MPC workflow.	12
2.7.	SSS share generation.	14
2.8.	SSS share reconstruction.	14
3.1.	High-level overview of CRDV.	17
3.2.	Layered architecture of CRDV [18].	18
4.1.	High-level architecture of the secure CRDV system.	30
6.1.	High-level implementation overview of the secure CRDV prototype.	56
6.2.	Coordinator-facing interfaces to the choreographic MPC runtime.	62
6.3.	Endpoint projection overview.	64
6.4.	Staged communication structure used by the choreographic MPC runtime.	66
7.1.	Latency of register, counter, map, and list operations.	75
7.2.	Latency of set insertion.	75
7.3.	Reconfiguration latency for different quorum transitions.	76
7.4.	Case-study deployment topology used in the macrobenchmark.	80
7.5.	Simplified DENM data model used by the macrobenchmark.	81
7.6.	Simplified IVIM data model used by the macrobenchmark.	81
7.7.	Median latency per macrobenchmark operation in the baseline scenario (top) and in the injected-latency scenario (bottom).	91
A.1.	Evolution of p95 latency during the macrobenchmark.	106
A.2.	Evolution of throughput during the macrobenchmark.	106

Chapter 1

Introduction

Modern applications increasingly rely on distributed data management to provide high availability, fault tolerance, and low-latency access to information [1]. Rather than relying on a single centralised database, data is often replicated across multiple nodes, geographic regions, or administrative domains. Replication allows applications to continue operating despite individual failures, network delays, or temporary disconnections between participants.

This replicated setting introduces the challenge of maintaining a consistent state when different replicas accept updates independently. In highly available systems, requiring synchronous coordination before every operation may increase latency and reduce availability. Conflict-free Replicated Data Types (CRDTs) [2] address this challenge by defining data types whose concurrent updates can be merged deterministically, allowing replicas to execute operations locally and converge once they observe the same set of updates.

In addition to availability and convergence, distributed systems may also need to satisfy confidentiality requirements [1, 3]. Applications increasingly operate across multiple entities or administrative domains that collaboratively maintain and process shared state. In such settings, some of the managed information may be sensitive, and unrestricted disclosure of replicated values may be undesirable or unacceptable. Distributed data management must therefore consider not only robust replication and convergence, but also the protection of selected data.

In Multi-Party Computation (MPC)-based secure CRDTs, each logical replica is realised by a set of MPC parties rather than by a single trusted server. When replicas exchange updates, the replicated layer propagates the logical CRDT state or metadata, while the protected values must also be made available to the MPC parties associated with the receiving replica. Therefore, the party set associated with a replica determines not only where shares are stored, but also which entities are trusted to maintain and process that replica's confidential state.

1.1. Problem Statement

The need to combine convergent replicated data management with confidentiality requirements has motivated the study of secure CRDTs [4]: replicated data types that retain the availability and convergence benefits of CRDTs while protecting sensitive state from individual participants. Recent work has shown how general-purpose CRDTs implementations can be transformed into secure variants using secure MPC [5].

This thesis builds on two main lines of work: Conflict-free Replicated Data View (CRDV) as a relational and view-based framework for CRDT-based replicated data management, and MPC-based secure CRDTs as a mechanism for protecting replicated state through secret sharing and secure computation. Although previous work demonstrates the feasibility of general-purpose MPC-based secure CRDTs, two important limitations remain.

First, the mapping between logical CRDT replicas and the MPC parties responsible for realising them is assumed to be statically fixed. This means that the set of parties maintaining and processing the protected state cannot be dynamically selected or changed as part of the replicated data management system. This limitation is relevant in practical distributed environments, where the entities trusted to maintain confidential state may change over time. Nodes may become unavailable, new parties may be introduced, or different replicas may need to rely on different subsets of available parties. In such settings, secure replicated state should be able to adapt its distributed representation without disclosing or modifying the underlying logical value.

Second, existing secure CRDT implementations have mostly been evaluated as independent libraries. While this validates the secure data type constructions themselves, it leaves open how such mechanisms can be incorporated into a higher-level replicated data management framework. Such a framework should not only store replicated objects, but also provide abstractions for representing application state, support the management of multiple replicated objects and data types, and define how updates are propagated and made convergent across replicas.

This raises the question of how secure CRDT mechanisms can be integrated into a replication framework without losing the modularity and abstraction benefits provided by that framework. In particular, it is necessary to understand how protected replicated objects can be represented, propagated, reconfigured, and exposed through existing replication abstractions.

1.2. Objectives

The objective of this thesis is to incorporate confidentiality into an expressive replicated data management platform, while preserving the availability, convergence, and modularity benefits provided by CRDT-based replication. In particular, the goal is to allow selected replicated values to remain confidential even when replicas belong to different administrative domains or are not fully trusted to observe application data in plaintext.

This thesis addresses this objective through four concrete goals. First, it extends CRDV with a secure representation for private values, where the replicated layer stores handles and the secure layer stores the corresponding secret-shared state. Second, it defines secure operations for creating, updating, opening, comparing, propagating, and resharing private values. Third, it uses resharing to change the MPC parties associated with protected values, allowing the trust assumptions of a private value to change without reconstructing it in plaintext or modifying the logical CRDV state. Finally, it implements and evaluates a prototype integrated with an existing CRDV system, using microbenchmarks and an application-oriented macrobenchmark.

The secure execution layer is implemented using choreographic programming [6]. Instead of implementing each participant separately and manually coordinating message exchanges, a choreography describes the global communication structure of a distributed computation. In this thesis, choreographies are used to orchestrate the interaction between MPC parties during secure operations, including the operations needed to manipulate shared values and redistribute shares.

The resulting system separates the replicated data management layer from the secure execution layer: CRDVs provide the replicated data abstractions, while the secure layer manages protected values through handles, coordinators, quorums, and MPC parties.

The proposed system is validated experimentally at two levels. First, microbenchmarks evaluate individual secure operations and the cost of redistributing shares between quorums. Second, an application-oriented macrobenchmark inspired by vehicular communication workloads evaluates the complete prototype in a more realistic replicated data management scenario.

1.3. Document Structure

The remainder of this thesis is organised as follows. Chapter 2 introduces the background concepts required for this work, including CRDTs, secure MPC, Shamir's Secret Sharing

(SSS), and choreographic programming. Chapter 3 reviews related work on CRDV and secure CRDTs, identifying the gap addressed by this thesis. Chapter 4 presents the architecture of the proposed secure CRDV system, including its main components, trust policies, and security assumptions. Chapter 5 describes the secure operations supported by the system and explains how they are used to implement secure CRDTs. Chapter 6 presents the prototype implementation, focusing on the integration between the CRDV framework and the secure runtime, and on the use of choreographic programming to implement the MPC protocols. Chapter 7 evaluates the prototype through microbenchmarks and a Cooperative Intelligent Transport Systems (C-ITS)-inspired macrobenchmark based on Decentralized Environmental Notification Message (DENM) and Infrastructure to Vehicle Information Message (IVIM) workloads. Finally, Chapter 8 concludes the thesis by summarising the main results, limitations, and directions for future work.

Chapter 2

Preliminaries

This chapter introduces the main concepts required to understand the proposed secure replicated data types. It first presents CRDTs, which provide the replication and convergence model used by the system. It then introduces MPC and secret sharing-based protocols, which provide the cryptographic foundation for computing over confidential data. Finally, since the proposed mechanisms rely on SSS, it describes share generation and reconstruction.

2.1. Conflict-free Replicated Data Types

Data replication is a fundamental technique in distributed systems, where data is maintained across multiple nodes to improve availability, fault tolerance, and geographic scalability. By storing replicas of the same data at different locations, systems can continue operating despite node failures and can serve requests closer to users. However, replication introduces the challenge of maintaining a consistent state when updates occur concurrently at different replicas.

Traditional consistency mechanisms often rely on coordination between replicas before updates can be committed. This is the natural choice for applications that require strong consistency guarantees, such as banking systems, where it is preferable to delay an operation rather than allow clients to observe conflicting states. However, this coordination may increase latency and reduce availability, particularly in geographically distributed environments or during network partitions.

As an alternative, many distributed systems adopt *eventual consistency*, one of the most common consistency models for highly available replicated storage. Under eventual consistency, replicas may temporarily diverge and clients may observe stale information. However, the system is expected to avoid permanent inconsistencies: once updates stop and all propagated updates have been processed, all replicas eventually converge to the same state [7].

CRDTs provide a principled solution to this problem [2, 8]. CRDTs are replicated data structures designed so that replicas can process updates independently while guaranteeing deterministic convergence. This is achieved by embedding conflict-resolution semantics directly within the data type, allowing concurrent updates to be merged without requiring coordination during normal operation.

2.1.1 Types of CRDTs

CRDTs are generally classified into two categories: state-based CRDTs, also known as convergent replicated data types (CvRDTs), and operation-based CRDTs, also known as commutative replicated data types (CmRDTs) [8, 9]. Delta-state CRDTs have also been proposed as a communication-efficient variant of state-based CRDTs [10].

State-based CRDTs propagate their local state to other replicas. When a replica receives a remote state, it combines the two states using a deterministic merge function. To guarantee convergence, the state space forms a join-semilattice and the merge operation computes the least upper bound of the two states. Because merge operations are associative, commutative, and idempotent, replicas are guaranteed to converge regardless of message ordering or duplication.

Operation-based CRDTs update operations rather than entire states. Each update is applied locally and then disseminated to other replicas, which execute the same operation on their local copies. Correctness requires that operations be delivered reliably and, in most cases, in causal order. Compared to state-based CRDTs, operation-based approaches often reduce communication overhead by transmitting only the performed operations instead of complete states.

Delta-state CRDTs provide an intermediate approach between these two models. Instead of propagating the full state after each update, replicas propagate smaller state fragments, called deltas, which capture the effect of recent updates while preserving the convergence properties of state-based CRDTs [10].

These approaches provide eventual convergence while enabling replicas to process updates independently. The choice between them depends on factors such as communication costs, implementation complexity, and system requirements.

2.1.2 Common CRDT Structures

Several CRDT abstractions have been proposed to support different application requirements. Among the most commonly used are counters, registers, sets, and maps. These structures provide the building blocks for more complex replicated data models.

The examples in this section use state-based representations because the thesis focuses on replicated state that is stored, merged, and later protected by the secure layer. The proposed separation between public replicated metadata and secret-shared payloads is not inherently tied to this model: operation-based variants would propagate handles with the operations, whereas delta-state variants would propagate handle-bearing deltas. Both alternatives would require the secure layer to provide reliable, causally compatible share propagation. They may reduce replication bandwidth, but add coordination and delivery requirements that were not evaluated by this thesis.

Counters maintain a numerical value that can be updated concurrently by multiple replicas. One of the simplest examples is the Grow-only Counter (G-Counter), which supports only increment operations. Each replica maintains a local component of the counter, and the global value is obtained by summing all components.

Figure 2.1 illustrates a G-Counter with two replicas. Replica A performs one increment, and replica B performs two increments. During synchronisation, replicas merge their states using the component-wise maximum. After synchronisation, both replicas converge to the value 3.

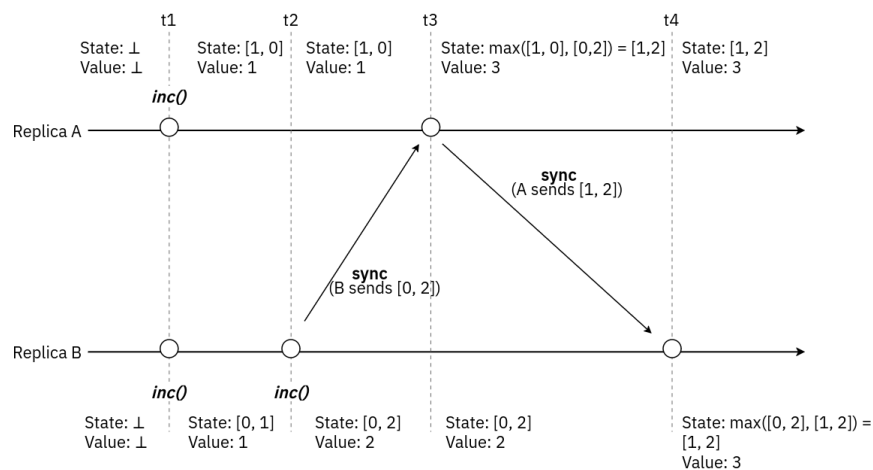


Figure 2.1: G-Counter example with two replicas.

Registers store a single value and are among the simplest replicated data structures. Different register variants define different policies for resolving concurrent updates.

A common example is the Last-Writer-Wins (LWW) Register, in which each update is associated with metadata such as a timestamp or a logical clock. Figure 2.2 shows a scenario in which concurrent updates are performed at different replicas. Replica A writes the value `red` with timestamp t_1 , while Replica B writes the value `blue` with timestamp t_2 . During synchronisation, the register keeps the value associated with the greatest timestamp, resulting in the converged value `blue`. In practice, implementations commonly use a logical clock, a hybrid logical clock, or a physical timestamp combined with a deterministic replica identifier to avoid requiring perfectly synchronised physical clocks; the latter only provides a total tie-breaking order and may not reflect real-time order under clock skew [8, 9].

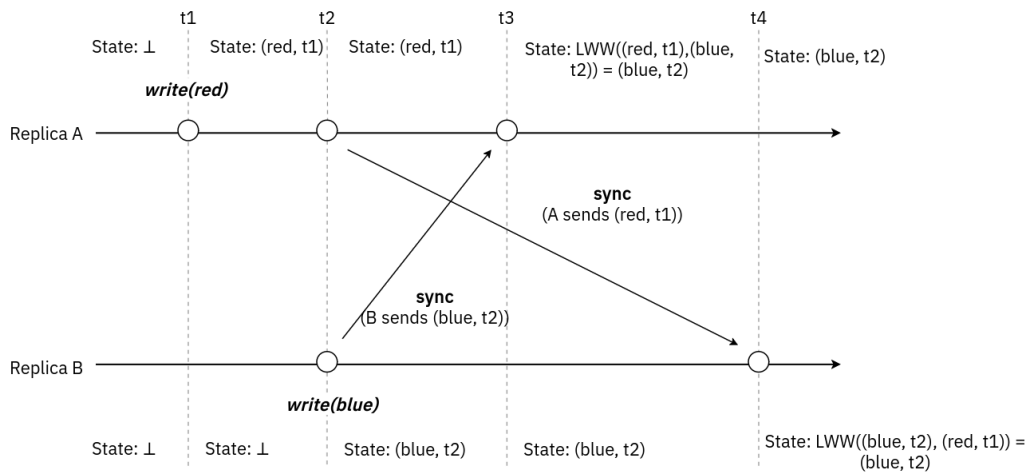


Figure 2.2: LWW register example.

Sets maintain collections of unique elements while supporting replicated updates. Different CRDT set variants provide different semantics depending on whether removals are supported and how concurrent additions and removals are handled.

The simplest variant is the Grow-only Set (G-Set), which supports only insertion operations. Once an element has been added, it can never be removed. In this case, synchronisation is performed by merging the elements added at each replica, resulting in the union of all inserted elements.

To support removals, the Two-Phase Set (2P-Set) maintains separate sets for additions and removals. An element is considered a member of the set only if it belongs to the add set and does not belong to the remove set. Figure 2.3 illustrates this behaviour. Replica A adds element b , while Replica B removes element a . During synchronisation, replicas merge the add set and the remove set independently using set union. The visible value is obtained by subtracting the remove set from the add set, resulting in $\{b\}$.

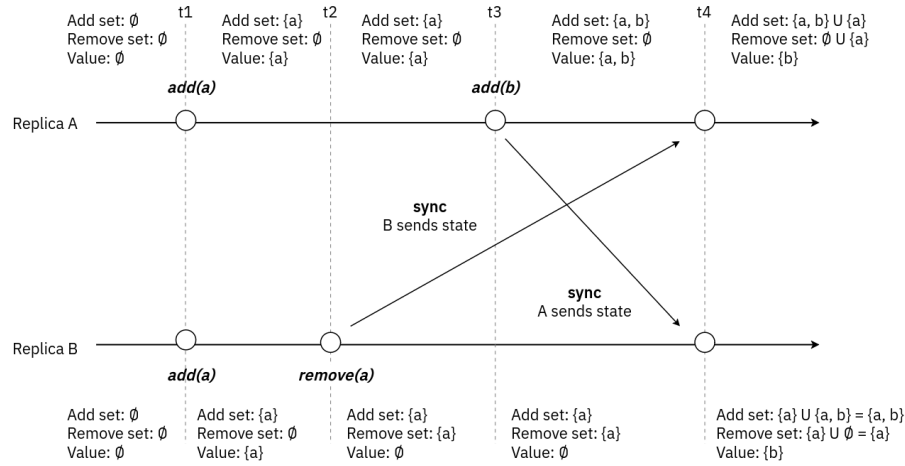


Figure 2.3: 2P-Set example.

This allows elements to be removed, but prevents them from being reinserted afterwards, since removals are recorded permanently as tombstones. Although this is restrictive, it illustrates how replicated sets encode additions and removals in a way that allows replicas to merge their states deterministically and converge despite concurrent updates.

Maps associate keys with values and are commonly used to represent structured application data. In CRDT systems, maps allow updates to different keys to be performed independently and later merged deterministically.

Figure 2.4 illustrates a simple CRDT map. Starting from the same initial state, Replica A adds the key `city` with value `Porto`, while Replica B concurrently adds the key `age` with value `25`. During synchronisation, both updates are preserved because they affect different keys, and the replicas converge to a map containing both entries.

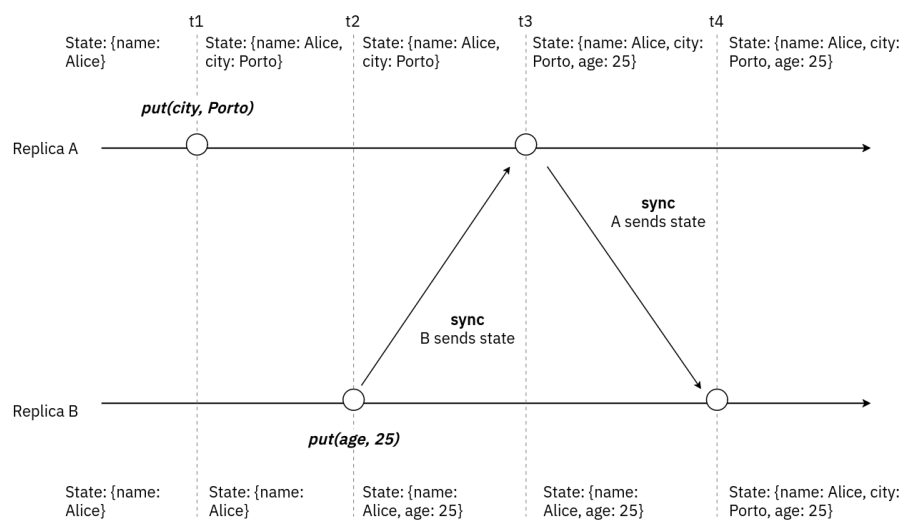


Figure 2.4: CRDT map example.

When concurrent updates affect the same key, the conflict is resolved according to the semantics defined for that entry. For example, a map can associate each entry with an LWW register, in which case concurrent writes to the same key are resolved by the register timestamp and tie-breaker; updates to different keys remain independent. In many CRDT systems, map entries may themselves contain replicated values, enabling maps to be composed with other CRDT primitives to represent more complex objects.

The composability of counters, registers, sets, and maps enables the construction of complex replicated objects from simpler CRDT primitives while preserving eventual consistency and deterministic convergence [2].

2.2. Secure Multi-Party Computation

MPC is a cryptographic paradigm that allows a set of parties to jointly compute a function over their private inputs without revealing those inputs to one another [11, 12]. Instead of relying on a trusted third party, participants execute a distributed protocol that reveals only the intended output of the computation.

Figure 2.5 illustrates the general MPC model. Each participant provides a private input to the protocol, which collaboratively evaluates a function over all inputs. During the execution, the inputs remain confidential, and only the final result is disclosed.

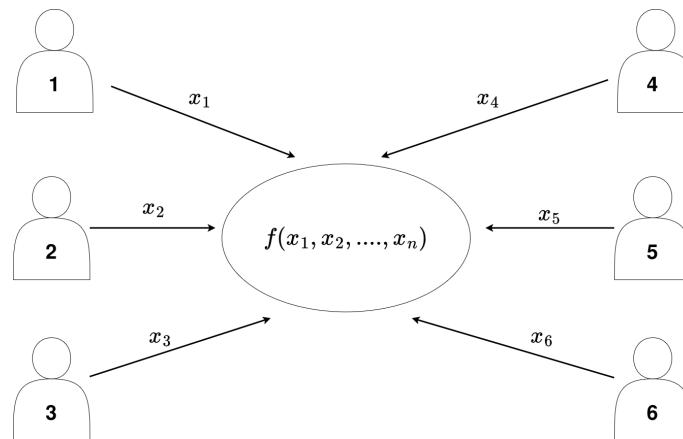


Figure 2.5: General overview of Secure MPC.

MPC protocols are typically analysed under a threat model that specifies the capabilities of potential adversaries. A common assumption is the *semi-honest* (or *honest-but-curious*) model, in which participants follow the protocol correctly but attempt to infer additional information from the messages they observe during execution. This is the adversarial model assumed throughout this thesis because it isolates the confidentiality implications

of integrating secret sharing with CRDV while avoiding the substantial protocol and performance cost of enforcing correct behaviour. Alternatively, adversaries can be active or malicious, if they can arbitrarily deviate from the protocol specification, for example by sending malformed messages, inconsistent values, or refusing to participate [12]. Extending the system to support this stronger model would require mechanisms such as verifiable secret sharing and maliciously secure MPC protocols; this extension is left for future work. Under the semi-honest model, a protocol is considered secure if no participant learns information about another participant's private input beyond what can be inferred from the final output itself.

In the context of this thesis, and under the semi-honest assumption, MPC provides the basis for executing operations over confidential replicated state. Instead of storing and processing plaintext values, replicas can operate over protected representations of those values, allowing secure operations to be integrated with replicated data structures.

2.2.1 Secret Sharing-Based MPC

Several techniques have been proposed for implementing MPC, including garbled circuits, homomorphic encryption, and secret sharing-based protocols [11]. In this thesis, the focus is on secret sharing-based MPC, a well-established approach for secure computation among multiple parties [13]. This approach is a natural fit for the replicated setting considered here: trust is distributed across several participants, values are represented by shares held by different parties, and linear arithmetic operations can often be evaluated directly over those shares.

The central idea is to divide a secret value into multiple pieces, called *shares*, and distribute them among a set of parties. Individually, these shares reveal no information about the original secret, but a sufficient subset of shares can be combined to reconstruct it. Rather than operating on plaintext values, participants perform computations directly on the shares, ensuring that intermediate values remain protected throughout protocol execution.

Figure 2.6 presents the general workflow of secret sharing-based MPC. Each party P_i starts with a private input x_i and secret-shares that input among the participating parties. For example, in a protocol with three parties, P_1 splits x_1 into the shares $[x_1]_1$, $[x_1]_2$, and $[x_1]_3$, keeping one share and distributing the others. After this distribution phase, each

party holds one share of every input. The parties then execute the MPC protocol over these distributed shares, producing shares of the result, such as $[y]_1$, $[y]_2$, and $[y]_3$.

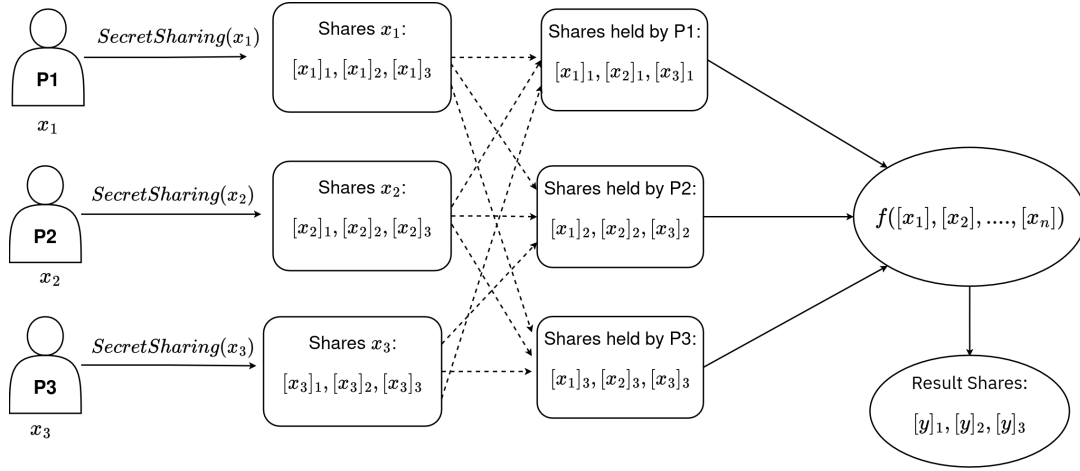


Figure 2.6: Secret sharing-based MPC workflow.

A key advantage of linear secret sharing schemes is that certain operations can be evaluated locally by each participant. For example, in additively homomorphic schemes such as Shamir Secret Sharing, additions and subtractions can be performed independently on the shares without requiring communication. More complex operations, such as multiplications or comparisons, generally require interaction between parties and additional protocol steps to preserve the security properties of the sharing scheme.

This approach distributes trust across multiple participants rather than relying on a single trusted entity. Security is maintained as long as the number of colluding parties remains below a predefined threshold determined by the underlying secret sharing scheme.

Among the various secret sharing techniques proposed in the literature, SSS [14] is one of the most widely used due to its simplicity, efficiency, and information-theoretic security guarantees. Since the mechanisms proposed in this thesis rely on SSS to protect replicated state and support secure operations, it constitutes a fundamental building block of the overall system.

2.3. Shamir Secret Sharing

SSS is a threshold secret sharing scheme proposed by Shamir in 1979 [14]. The scheme enables a secret value to be divided into multiple shares and distributed among a set of participants in such a way that only a predefined number of shares is required to reconstruct the secret.

More formally, a (t, n) -threshold scheme distributes a secret among n participants such that any subset of at least $t + 1$ participants can reconstruct the secret, whereas any subset of at most t participants obtains no information about it. This property allows confidentiality to be maintained even if some participants become compromised.

SSS is based on polynomial interpolation over a finite field and provides information-theoretic security, meaning that its confidentiality guarantees do not depend on computational assumptions. Due to its simplicity and efficiency, SSS has become one of the most widely adopted building blocks for secret sharing-based MPC protocols [15].

2.3.1 Secret Generation

The sharing phase consists of transforming a secret value into a collection of shares that can be distributed among multiple participants. The scheme operates over a finite field $\mathbb{F}_p$, where p is a prime chosen large enough to represent the secrets and to provide at least n distinct non-zero evaluation points. The secret is represented as an element $s \in \mathbb{F}_p$, and all arithmetic operations are performed modulo p .

Given a secret s and a (t, n) -threshold configuration, the dealer samples coefficients $a_1, \dots, a_t \in \mathbb{F}_p$ uniformly at random and defines a polynomial $f \in \mathbb{F}_p[x]$ of degree at most t :

$$f(x) = s + a_1x + a_2x^2 + \dots + a_tx^t$$

The secret corresponds to the constant term of the polynomial, that is, $f(0) = s$.

Shares are created by evaluating the polynomial at n distinct non-zero points $x_1, \dots, x_n \in \mathbb{F}_p$:

$$(x_1, f(x_1)), (x_2, f(x_2)), \dots, (x_n, f(x_n))$$

Each participant receives one share $(x_i, f(x_i))$. The points x_i are public and identify the shares, while the values $f(x_i)$ must remain private to the corresponding participants [14, 16].

Figure 2.7 illustrates this process. The secret is encoded as the polynomial's intercept, while the generated shares correspond to points on the polynomial distributed among the participants.

The security of the scheme relies on the randomness of the polynomial coefficients. Given at most t shares, the observed points are compatible with multiple degree- t polynomials, including one polynomial for each possible secret in $\mathbb{F}_p$. Therefore, an adversary with an

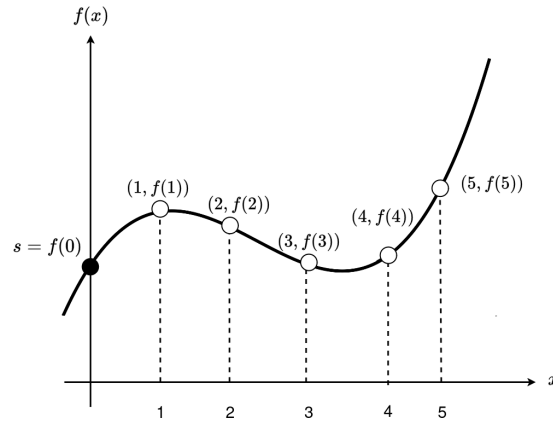


Figure 2.7: SSS share generation.

insufficient number of shares cannot distinguish the real secret from any other possible value in the field.

2.3.2 Secret Reconstruction

To recover the secret, at least $t + 1$ shares must be combined. Reconstruction is performed through polynomial interpolation over $\mathbb{F}_p$, which uniquely determines the original polynomial once a sufficient number of points are known.

Figure 2.8 illustrates this process. A subset of $t + 1$ shares is sufficient to interpolate the polynomial and recover its constant term, which corresponds to the original secret.

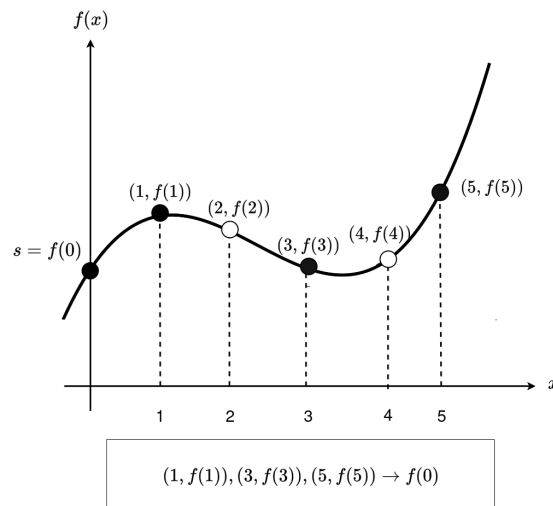


Figure 2.8: SSS share reconstruction.

In practice, reconstruction is typically performed using Lagrange interpolation [17]. Given a set of $t + 1$ shares (x_i, y_i) , polynomial interpolation can be used to uniquely determine

the degree- t polynomial from which the shares were generated. The reconstructed polynomial is given by

$$f(x) = \sum_{i=0}^t y_i l_i(x)$$

where the Lagrange basis polynomials are defined as

$$l_i(x) = \prod_{\substack{0 \leq j \leq t \\ j \neq i}} \frac{x - x_j}{x_i - x_j}$$

All additions, multiplications, and divisions are performed in $\mathbb{F}_p$. Since the secret corresponds to the constant term of the polynomial, recovering the secret simply requires evaluating the interpolated polynomial at $x = 0$:

$$s = f(0)$$

This threshold property makes SSS particularly suitable for distributed systems, as confidentiality can be preserved while tolerating failures and supporting collaborative computations across multiple participants.

Chapter 3

Related Work

This chapter reviews the work most closely related to the goals of this thesis. It first discusses CRDV, which provide the view-based replicated data management framework extended in this work. It then examines existing approaches to secure CRDTs, including encryption-based, access-control-based, and MPC-based techniques. The chapter concludes by identifying the gap between view-based replicated data management and confidentiality-preserving secure computation over replicated state, which motivates the design presented in the following chapters.

3.1. Conflict-free Replicated Data Views

CRDV was proposed as a framework for building replicated applications that combine the convergence guarantees of CRDTs with the declarative programming model of relational databases [18]. Rather than exposing CRDTs as opaque application-level objects or custom database types, CRDV represents replicated state using relational tables and SQL views, allowing developers to manipulate replicated data through standard SQL operations while benefiting from eventual consistency and automatic conflict resolution.

Figure 3.1 provides a high-level overview of the CRDV model. Applications interact with replicas through SQL queries, while the replicated data semantics are encoded inside the database using SQL views. Each replica can serve clients independently and later synchronise with other replicas, allowing applications to benefit from the convergence guarantees of CRDTs while using a relational interface.

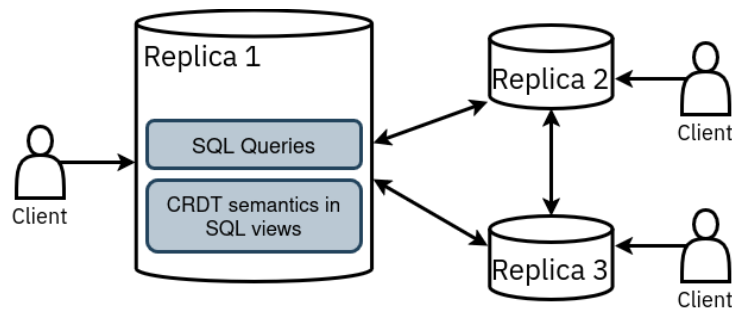


Figure 3.1: High-level overview of CRDV.

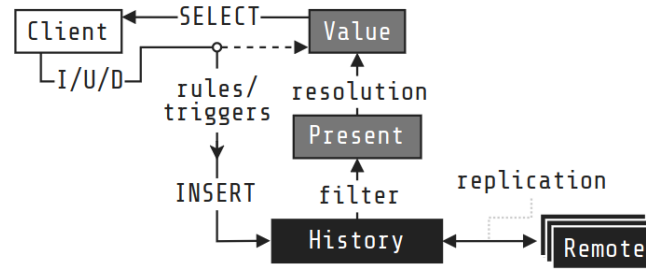


Figure 3.2: Layered architecture of CRDV [18].

3.1.1 Architecture

The architecture of CRDV is organised into three logical layers, illustrated in Figure 3.2. At the lowest level, a replicated *History* relation stores all updates together with the metadata required for replication and causality tracking. Above this layer, a *Present* layer filters obsolete versions and exposes the causally relevant state of each object. Finally, a *Value* view defines the conflict-resolution semantics visible to applications, producing the logical state of the replicated data structure.

This separation allows replication, causality management, and conflict resolution to be handled independently while remaining fully expressible in SQL. As a result, CRDV encodes replicated-data semantics in the database schema rather than embedding them within application-specific data structures.

3.1.2 CRDT Representation

A key aspect of CRDV is not the idea of storing a history of updates by itself, since history- or operation-based representations are common in CRDT systems, including pure operation-based CRDTs. Rather, the contribution of CRDV is to encode replicated state, metadata, and conflict-resolution logic using relational schemas and SQL views. Instead of encapsulating merge behaviour inside specialised data types, CRDV stores updates as relational tuples and derives the visible state of each object through view definitions. This allows conflict-resolution rules to be expressed in SQL and optimised together with application queries by the database engine.

The CRDV framework reproduces common CRDT abstractions by mapping each data type to relational state and defining the corresponding reconciliation rule as a Value view. Table 3.1 summarises how registers, sets, maps, lists, and counters can be represented in this model.

Table 3.1: Representation of common CRDT abstractions in CRDV.

Abstraction	Stored representation	Value derivation
Register	id: object identifier; data: assigned value.	Selects the visible assignment, e.g. using LWW.
Set	id: set identifier; key: element; op: add/remove.	Selects visible elements according to the set policy, e.g. add-wins or remove-wins.
Map	id: map identifier; key: entry key; data: entry value.	Derives the visible value independently for each key.
List	id: list identifier; key: position identifier; data: element.	Selects visible elements and orders them by position.
Counter	id: counter identifier; key: contribution identifier; data: contribution value.	Computes the logical value by aggregating contributions.

As a concrete example, Table 3.2 shows a simplified replicated data relation for a LWW register. Each assignment is stored as a row in the History layer, together with the meta-data required to track causality and resolve conflicts. Since registers do not require an internal key, the `key` column of the generic data relation is omitted from this example.

Table 3.2: Simplified replicated data relation for a LWW register.

id	type	data	op	site	lts	pts
r1	r	A	a	1	[1,0]	10
r1	r	B	a	2	[0,1]	12
r1	r	C	a	1	[2,1]	15

In this relation, all rows use `r1` because they record the update history of the same replicated register; they are not distinct register instances. The column `id` identifies the replicated object, `type` indicates the CRDT abstraction being represented, and `data` stores the application value. The `op` column records whether the row corresponds to an addition/update or a removal, while `site` identifies the replica that produced the update. The `lts` column stores the logical timestamp used to determine causal ordering in the Present layer, and `pts` stores the physical timestamp used by the LWW reconciliation rule.

The corresponding Value view defines the semantics exposed to applications by selecting the visible assignment for each register. Listing 3.1 shows a simplified SQL definition of this view.

In this simplified illustrative view, based on the PostgreSQL implementation of CRDV, the view exposes the selected register value through the columns `id` and `data` (lines 1–2). The inner query reads from the generic replicated relation `Data` and restricts the result to register entries using the predicate `type = 'r'` (lines 9–10). The ranking expression partitions entries by object identifier and orders them by decreasing physical timestamp, using additional attributes as deterministic tie-breakers (lines 5–8). Finally, the outer query keeps only the highest-ranked row for each register (line 12), corresponding to the value exposed by the view.

```

1 CREATE OR REPLACE VIEW RegisterLww AS
2     SELECT id AS id, data
3     FROM (
4         SELECT id, data,
5             rank() OVER (
6                 PARTITION BY id
7                 ORDER BY pts DESC, site, data, ctid
8             ) AS rank
9         FROM Data
10        WHERE type = 'r'
11    ) t
12    WHERE rank = 1;

```

Listing 3.1: Simplified SQL definition of the LWW register Value view.

In the example from Table 3.2, this rule selects the row with value `C`, since it has the greatest physical timestamp. The additional ordering attributes, such as `site`, `data`, and `ctid`, provide deterministic tie-breaking when timestamps are equal. The `ctid` attribute is PostgreSQL-specific and is used here only as a final tie-breaker in this implementation.

Other replicated data types follow the same general pattern. The underlying relation stores updates and metadata, while each Value view defines how the logical value is derived. For sets, the element is stored in the `key` column and the `op` column distinguishes additions from removals. For maps, `key` identifies the map entry and `data` stores the associated value, with conflict resolution applied independently for each map key. For lists, `key` stores an ordering identifier, allowing the Value view to reconstruct the sequence by selecting visible elements and ordering them by position. For counters, `key` identifies a partial contribution, such as the value associated with a site, and the logical counter value is computed by aggregating the partial values stored in `data`.

Overall, this representation makes CRDV particularly relevant to this thesis. By expressing replicated state, causal metadata, and reconciliation logic through a modular view-based model, CRDV provides a natural foundation for studying how confidentiality-preserving mechanisms can be integrated into a higher-level replicated data management framework.

3.1.3 Limitations

The relational and view-based structure of CRDV is particularly relevant to this thesis because it separates the storage of replicated updates from the derivation of application-facing values. Replicated updates and metadata are stored in the underlying relations, while the visible state of each object is computed through SQL views that encode the corresponding CRDT semantics. This separation suggests a compatible way of introducing confidentiality mechanisms: protected representations can be stored in the underlying relations, while the view layer can continue to derive the replicated object state from the available representation.

However, the original CRDV design does not address confidentiality as part of its system model. Its focus is on replication, availability, convergence, and the declarative representation of conflict-resolution logic. As a result, replicated relations store and process application data in plaintext, implicitly assuming that each participating replica is trusted to access the replicated state.

This assumption becomes restrictive in collaborative or multi-organisational environments, where replicas may be operated by different entities or deployed in less trusted infrastructure. Even when reconciliation rules depend mainly on metadata, replicated relations may still expose application values, keys, elements, or derived results. Consequently, CRDV provides useful abstractions for representing, propagating, and deriving convergent replicated state, but it does not provide confidentiality against replicas that store or process that state.

A related limitation follows from where CRDV semantics are executed. The Value views that define the visible state of replicated objects are evaluated by the database engine as regular SQL queries. This is convenient for plaintext replicated data, but it also means that a view is not an interactive process that can directly invoke an MPC protocol whenever its reconciliation logic needs to inspect a protected value. If conflict resolution or value

derivation depends on private data, that logic cannot be implemented simply by placing secret-shared values behind a SQL view.

Therefore, a secure extension of CRDV must distinguish between the parts of CRDT semantics that can remain inside SQL views and the value-dependent operations that must be executed by the secure layer. In particular, the system must either keep the required reconciliation metadata public, represent private payloads by handles whose ordering and visibility can be decided from public metadata, or move value-dependent computation to the secure layer before the CRDV view is evaluated.

Addressing these limitations requires mechanisms that protect selected values while preserving the main benefits of the CRDV model: modular replicated data definitions, propagation, convergence, and application-facing views.

3.2. Secure CRDTs

The confidentiality limitations of traditional replicated data types have motivated several lines of work on secure CRDTs. These approaches differ in the security property they prioritise and in the assumptions they make about the participants that store, access, or compute over replicated state.

This section groups existing work into three families: encryption-based approaches, access-control-based approaches, and MPC-based approaches. The discussion focuses on four dimensions that are relevant to this thesis: whether the approach protects stored replicated state, whether it controls who may access or update the data, whether it supports computation over protected values, and whether it integrates with view-based replicated data management frameworks.

3.2.1 Encryption-Based Approaches

Encryption-based approaches protect replicated state by encrypting data before storage or replication. Under this model, replicas store protected representations of the application state rather than plaintext values, reducing the risk of information disclosure in the event of storage compromise or unauthorised access. This is particularly useful when the main security goal is to protect data at rest or prevent untrusted replicas from directly inspecting replicated payloads.

One of the most influential works in this area is *Secure Conflict-free Replicated Data Types* by Barbosa et al. [4]. The authors introduce formal security definitions for CRDTs and propose cryptographic constructions that preserve convergence while protecting application data from unauthorised access. Their work demonstrates that confidentiality guarantees can be incorporated into CRDTs without sacrificing their core replication properties.

The main strength of this line of work is that it extends the CRDT model with explicit confidentiality guarantees. Instead of assuming that all replicas can inspect the replicated state, encryption-based approaches distinguish between the ability to store or propagate protected data and the ability to learn its plaintext contents. This is an important step towards deploying replicated data types in environments where storage nodes or synchronisation services are not fully trusted.

However, encryption-based approaches face limitations when the system must compute over confidential data. Replicas must often evaluate queries, apply updates, merge concurrent versions, or derive new states from replicated objects. For example, an application may need to compute the maximum of two replicated values, compare a candidate element with existing set elements, or update a counter based on the current value. With conventional encryption, these operations cannot generally be evaluated directly over ciphertexts. Unless the supported operations are compatible with the underlying encryption scheme, data must be decrypted by some trusted component before application-level logic can be applied.

The cost of encryption-based approaches depends strongly on the operations that the replicated data type must support. For data types whose merge logic does not inspect the protected payload, such as registers, encryption can be added with limited changes to the underlying CRDT. For data types that require replica-side computation over values, such as counters, stronger cryptographic techniques such as homomorphic encryption may be required [19]. This introduces a trade-off between the functionality supported over encrypted data, the information leaked by the construction, and the performance overhead imposed on clients and replicas.

This distinction is important for the goals of this thesis. Protecting the stored representation of a value is not sufficient when replicas must also execute operations whose result depends on that value. In a secure replicated data management setting, the system should be able to manipulate confidential state while limiting what each individual replica learns. Encryption-based secure CRDTs therefore address part of the problem, but do

not by themselves provide a complete model for computation over confidential replicated state.

3.2.2 Access-Control-Based Approaches

Another line of work focuses on access control for collaborative CRDT-based applications. Instead of hiding all replicated data cryptographically, these approaches define policies that determine which users or replicas are authorised to access, modify, or propagate specific parts of the shared state. Access-control mechanisms are particularly relevant in collaborative and local-first applications, where users may have different permissions over shared documents while still requiring availability under disconnection.

Rault et al. [20] propose a distributed access-control mechanism for collaborative applications using CRDTs. Their approach treats access-control policies themselves as replicated data, allowing permissions to be updated and propagated in decentralised applications. In their model, access-control lists are represented using CRDT structures, such as maps from users to sets of roles, and conflict-resolution strategies are defined for concurrent policy updates. The authors also discuss the interaction between data operations and policy operations, showing that conflicts between them require additional mechanisms, such as epochs, to preserve a consistent access-control context.

Renaux et al. [21] propose Secure RDTs, which enforce role-based access-control policies for offline-available JSON data. Their work addresses the problem that replicated clients may otherwise receive a full copy of the data, including fields they should not read, and may locally modify fields they should not write. To prevent this, a trusted leader projects both the data and the security policy according to each user's role. Clients receive only the fields they are allowed to read and a policy excerpt that allows well-behaved clients to check write permissions while offline. The leader then enforces write permissions when changes are synchronised.

The main advantage of these approaches is that they address an important practical requirement of collaborative replicated systems: not every participant should necessarily have the same permissions over the shared state. They show that access-control metadata can itself be replicated, projected, or enforced in ways that preserve availability and eventual consistency.

Access-control mechanisms are therefore orthogonal and complementary to the problem addressed in this thesis. Cryptographic techniques protect the data itself, ensuring that

only entities with the required cryptographic material can access it. Access-control mechanisms, in contrast, define and manage which users or components should be allowed to obtain that material, invoke operations, or receive reconstructed results.

For this reason, access control is important for secure replicated applications, but it does not by itself provide computation over confidential replicated state. This thesis focuses on the latter problem: how replicated values can remain protected while still being manipulated by the replicated system.

3.2.3 MPC-Based Approaches

More recent work has explored the use of MPC and secret sharing to protect CRDT state while still allowing replicas to perform operations over confidential data. Instead of relying on a single trusted node to decrypt and process values, these approaches distribute trust across multiple parties. Data can be represented through secret shares, and operations can be evaluated by secure protocols without revealing the underlying plaintext to any individual participant.

Portela et al. [5] propose General-Purpose Secure CRDTs, a framework that combines CRDTs with MPC techniques. Instead of storing plaintext values, data are represented using secret-sharing schemes and manipulated through secure protocols executed by multiple parties. This approach allows updates and conflict-resolution operations to be performed without revealing the underlying values to any individual participant.

Compared with encryption-based approaches, MPC-based secure CRDTs provide greater flexibility, since computations can be executed directly over protected state. This makes them particularly attractive for applications that require both strong confidentiality guarantees and collaborative manipulation of shared data. Rather than only protecting stored values, MPC allows the system to compute functions over those values while keeping them hidden from individual parties.

The main cost of MPC-based approaches comes from replacing local plaintext computation with protocols executed by multiple parties. Values must be stored as shares, operations may require communication and coordination between parties, and the replicated system must manage both replica-to-replica propagation and party-level secure execution. This overhead is the price paid for enabling computation over confidential state without relying on a single trusted component. Existing MPC-based secure CRDTs also assume a

fixed mapping between logical replicas and the parties that realise them, leaving dynamic party reconfiguration as an open system-level problem.

This line of work is therefore the closest to the goals of this thesis. It represents sensitive data as secret shares and allows generic computations to be executed over those shares, without revealing the underlying plaintext values to any individual party. However, existing MPC-based secure CRDTs are mainly presented as secure data-type abstractions. They do not focus on their integration as a secure back-end for a more expressive replicated data management framework such as CRDV, where state, metadata, and conflict-resolution logic are represented through relational tables and SQL views. In addition, they assume a fixed mapping between logical replicas and the parties that realise them, leaving dynamic reconfiguration of the parties maintaining protected state outside their main scope. These two aspects, integration with CRDV and reconfiguration of the parties holding shares, are the main gap addressed by this thesis.

Table 3.3 summarises the main strengths and limitations of the families of approaches discussed in this section.

Table 3.3: Comparison of secure CRDT approaches.

Approach	Main Strength	Main Limitation
Encryption-based approaches [4]	Provide a simple and minimal security layer for CRDT payloads.	Limited support for general computation over protected state.
Access-control-based approaches [20, 21]	Support permission management for replicated and offline-capable collaborative data.	Require trust in policy-enforcing components or do not hide data from authorised participants.
MPC-based approaches [5]	Enable secure computation directly over confidential replicated state.	No support for dynamic party reconfiguration.

3.3. Discussion

This chapter reviewed two lines of work relevant to this thesis. CRDV provide a view-based model for replicated data management, where CRDT semantics are expressed through relational tables and SQL views. Although other systems expose CRDTs through storage systems, geo-replicated databases, or programming models [9], such as Riak [22], AntidoteDB [23], and Lasp [24], this thesis focuses on CRDV because their relational model provides a natural integration point for confidentiality mechanisms. Secure CRDT

approaches, summarised in Table 3.3, provide mechanisms for confidentiality, access control, or computation over protected replicated state.

However, these directions address complementary parts of the problem. CRDV provide relational integration and declarative conflict resolution, but assume plaintext state and trusted replicas. Existing secure CRDT approaches improve the security of replicated data types, but are not generally designed as extensions of view-based frameworks where state, metadata, and reconciliation logic are represented through relational definitions.

Across these approaches, security is not only a matter of the protection mechanism used, but also of the system-level costs introduced by that mechanism: cryptographic functionality for encryption-based designs, policy management and enforcement for access-control-based designs, and communication and coordination between parties for MPC-based designs. These costs do not invalidate the approaches, but they show that security mechanisms must be considered together with the performance and deployment model of the replicated system.

Access control is an important part of secure replicated applications, since it determines which users or components are allowed to access data, invoke operations, or receive declassified results. However, it is orthogonal to the main goal of this thesis. This work does not focus on deciding who should be authorised to access protected data; instead, it focuses on how selected values can remain confidential even while the replicated system stores and manipulates them.

This leaves a gap between secure replicated data structures and view-based CRDT frameworks such as CRDV. The following chapter addresses this gap by presenting the design of a secure CRDV framework based on secret sharing and MPC. The goal is to preserve the modularity and declarative convergence model of CRDV, while preventing individual replicas from learning confidential application data and supporting reconfiguration of the parties responsible for maintaining protected values.

Chapter 4

Secure CRDV Architecture

The goal of the proposed system is to extend CRDV with support for a confidential replicated state. In the original CRDV model [18], replicas are responsible for storing, propagating, and resolving replicated updates, but application values are stored and processed in plaintext. We extend their design by introducing a secure computation layer based on SSS. The proposed architecture preserves the replication and convergence mechanisms of CRDV. Public data continues to be stored and processed directly by the CRDV layer. Private data, however, is no longer represented in the replicated state by its plaintext value. Instead, it is represented by a handle: a unique, non-secret identifier that names one logical secret-shared value in the secure layer. The handle is metadata, not a cryptographic key, ciphertext, or secret share; it lets the two layers refer to the same protected value without exposing its plaintext. This allows replicas to continue propagating and resolving replicated updates while avoiding the storage of sensitive application values in plaintext.

The chapter is organised as follows. Section 4.1 presents the two-layer architecture of the system. Section 4.2 explains how replicas are associated with sets of MPC participants. Section 4.3 states the security assumptions and threat model. Finally, Section 4.4 describes how acceptable party sets are represented as trust quorums.

4.1. Architectural Overview

The architecture of the proposed system is organised into two main layers: the CRDV layer and the secure layer. As shown in Figure 4.1, clients interact with CRDV replicas through the same SQL-facing interface used in a regular CRDV deployment. The CRDV layer preserves the functionality of the original CRDV framework: it stores replicated relations, propagates updates between replicas, maintains metadata, and applies the view-based conflict-resolution logic that defines the visible state of each replicated object.

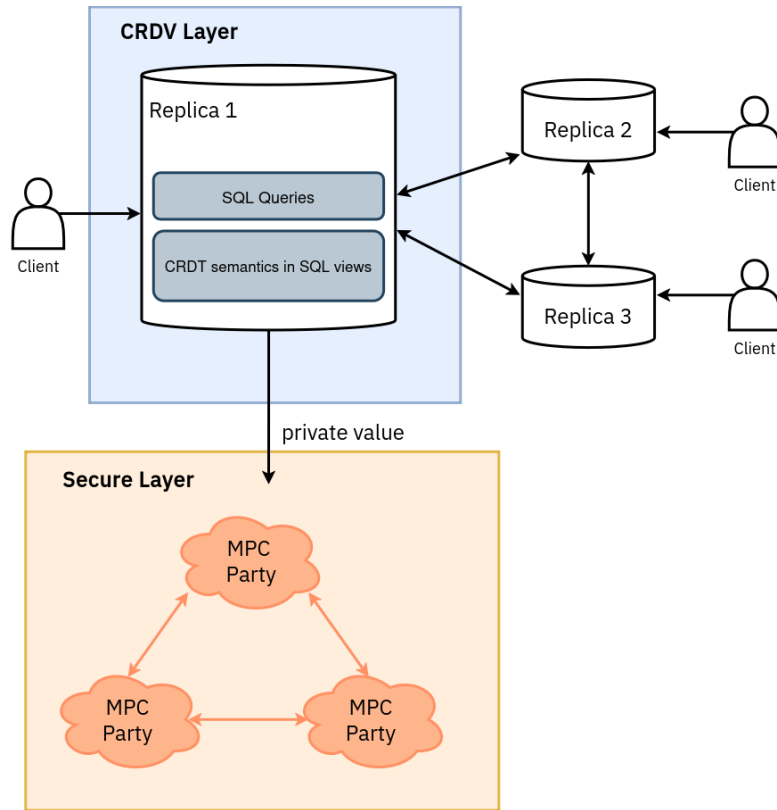


Figure 4.1: High-level architecture of the secure CRDV system.

The secure layer, also shown in Figure 4.1, is introduced to handle the computations required by CRDV over private values. Its purpose is not to change the CRDV programming model, but to act as a transparent execution layer: private values are never represented in plaintext at the replica side. Instead, replicas store and propagate handles, while the secure layer stores and manipulates the corresponding secret shares through MPC parties.

When an operation only involves public data, it is executed entirely within the CRDV layer. When an operation involves private data, the replica invokes the secure layer. The secure layer creates, updates, reconstructs, or reconfigures the secret-shared representation of the private value, and the CRDV layer stores only the corresponding handle in the replicated state.

This separation also creates two complementary propagation mechanisms. At the CRDV layer, replicas propagate handles and metadata through the normal replication mechanism. At the secure layer, the corresponding shares must be propagated or reshared to the MPC parties associated with the replicas that may later access those handles. Therefore, the handle is the replicated reference visible to CRDV, while the shares are the protected representation maintained by the secure layer.

4.1.1 CRDV Layer

The CRDV layer is responsible for the database-facing representation of replicated state. It stores public values directly and stores handles for values marked as private. These handles are part of the replicated CRDV state and are propagated through the normal CRDV mechanisms. As a result, replicas can converge on the same logical representation of an object without storing the private plaintext value.

From the perspective of the CRDV framework, a handle behaves as the value associated with a private field. It can be stored in the corresponding relations, propagated to other replicas, and used by the conflict-resolution logic of the underlying data type. However, the handle does not reveal the private value itself. It only identifies the secret-shared state maintained by the secure layer.

This separation does not hide the structure or metadata of the replicated state. The CRDV layer still processes object identifiers, data-type information, operation metadata, causal metadata, timestamps, and handles in plaintext. Consequently, replicas may learn how many replicated objects exist, which CRDT abstractions they belong to, and which operations are performed over them. The goal of this work is to protect selected application data values, not the existence of those values or the structure of the replicated objects.

Providing this additional level of privacy would require techniques for hiding access patterns and data-structure metadata, such as Oblivious RAM (ORAM) [25] or Oblivious Data Structures (ODSs) [26]. ORAM could hide which protected location is accessed, while oblivious data structures could hide properties such as collection size and insertion position. Integrating either approach would strengthen the model but would add communication, storage, and latency overhead, particularly to the replicated secure-set operations that already dominate the evaluation. These mechanisms are orthogonal to the goals of this thesis and are therefore outside its scope.

4.1.2 Secure Layer

The secure layer is responsible for all operations involving private values. It receives requests from the CRDV layer when an operation targets a private value and ensures that the corresponding computation is performed without storing that value in plaintext in the replicated database state.

Private values are represented using SSS. Instead of storing a private value directly in a replica, the system splits it into shares and distributes those shares among a selected set of MPC parties. Each party stores only its own share of the value. By itself, a single share does not reveal the underlying plaintext. A sufficient number of shares is required to reconstruct the value, according to the threshold defined by the secret sharing scheme.

The secure layer therefore acts as the computation backend for private CRDV values. Depending on the operation requested by the CRDV layer, MPC parties may create new shares, update existing shares, execute a secure computation, reconstruct a value when explicitly authorised, or redistribute shares to another set of parties. These operations are described in detail in Chapter 5.

This separation also distinguishes the communication patterns of the two layers. Communication at the CRDV layer concerns replica-to-replica propagation of replicated updates, handles, and metadata. Communication at the secure layer concerns interactions between MPC parties for share management, secure computation, share propagation, and reconfiguration.

As a consequence, there may be a temporary discrepancy between propagation at the CRDV layer and propagation at the secure layer. A replica may receive a replicated handle before the secure layer has completed the propagation of the corresponding shares to that replica's MPC parties. In that case, the handle is already part of the replicated CRDV state, but the secure layer cannot yet translate it into an operational secret-shared value at that replica. This is analogous to, but not identical with, the eventual delivery assumption of the underlying CRDV model. Both require reliable eventual delivery for convergence and availability. However, delivery of a CRDV update makes its handle visible, whereas completion of share propagation is additionally required before the receiving replica can perform secure operations on that handle. The system must therefore tolerate a temporary *not-yet-operational* handle and retry or delay dependent secure operations until the corresponding shares arrive. Once propagation completes, the two layers again refer to the same logical value.

4.2. Replicas as Sets of MPC Participants

In the proposed system, a secure replica is represented by a set of MPC parties rather than by a single trusted server. Each logical CRDV replica is associated with one such set in the secure layer. The parties in that set jointly constitute the secure representation

of the replica: they store and manipulate shares of the private values referenced by the replica, while no individual party stores those values in plaintext. The number of parties shown in Figure 4.1 is only illustrative; the design is not restricted to three parties per replica.

This gives each logical replica two complementary representations. At the CRDV layer, the replica stores replicated relations, metadata, public values, and handles. At the secure layer, the same replica is realised by the set of MPC parties associated with it. The private values referenced by the handles stored in the CRDV layer are represented by shares distributed among those parties. The CRDV layer can therefore propagate and resolve updates without storing the corresponding private values in plaintext.

Different replicas may be associated with different numbers of MPC parties. This allows deployments to reflect differences in availability, administrative domains, trust assumptions, or local infrastructure. A replica may use a small set of nearby parties in one deployment, while another replica may use a larger or administratively different set of parties.

The association between replicas and MPC parties is not fixed by the data model. A private value shared among n parties can later be redistributed to a different set of m parties without reconstructing the plaintext value. This property enables reconfiguration when parties are added, removed, fail, or when a replica changes the set of parties it is willing to rely on.

The source and target sets of a reconfiguration do not need to be disjoint. If the original set of parties and the new set of parties overlap, parties in the intersection may contribute to the new sharing. Conversely, disjoint sets can be used when stronger separation between administrative domains is required. The security conditions that make a target set acceptable are defined by the trust quorum policy described in Section 4.4.

4.3. Security Assumptions and Threat Model

The proposed system assumes a semi-honest adversary model, commonly used in secure computation [11, 12]. Participants are expected to follow the protocol correctly, but they may attempt to infer additional information from the messages, shares, handles, and metadata they observe during execution. The goal of the secure layer is to protect the plaintext contents of private values, even when individual CRDV replicas or some MPC parties are curious.

Private values are protected using SSS. A value shared among a quorum of MPC parties can only be reconstructed when a sufficient number of shares is available, according to the threshold defined by the scheme. In the notation used in this thesis, t denotes the privacy threshold: any set of at most t shares reveals no information about the value, while $t + 1$ shares are sufficient to reconstruct it.

The trust assumption is local to each sharing. Confidentiality is preserved as long as the number of compromised parties within the quorum responsible for that sharing remains at most t . When a value is propagated or reconfigured, the secure layer does not copy the same shares unchanged to another quorum. Instead, it uses resharing to create a fresh sharing of the same underlying value for the target set of parties. This means that changing the quorum also changes the set of entities trusted to store and process that protected value. In other words, the source and target quorums may correspond to different trust assumptions, even though they represent the same logical value. As a result, compromised parties from different sharings cannot simply combine their shares to reconstruct the value, unless enough shares from the same sharing are obtained.

The system does not aim to provide full metadata privacy. The choice of which application fields are private is made by the developer or schema configuration: fields marked as private are represented by handles in the CRDV layer and by secret-shared values in the secure layer, while fields marked as public remain stored directly in the replicated database. Independently of this choice, the CRDV layer still exposes the metadata required to represent and propagate replicated objects, such as object identifiers, operation identifiers, causal metadata, quorum identifiers, handles, access patterns, communication patterns, and the fact that a private value exists. The security goal is therefore limited to protecting the contents of fields configured as private, while preserving the replication and convergence mechanisms of CRDV.

The concrete prototype described in Chapter 6 also assumes that the component receiving private inputs is trusted to observe those inputs at submission time and to distribute the corresponding shares correctly. The system does not address malicious parties that deviate from the protocol, denial-of-service attacks, compromise of enough parties to satisfy the reconstruction threshold, or leakage through metadata and access patterns.

4.4. Trust Quorums

The previous sections explain that each replica is associated with a set of MPC parties, and that confidentiality depends on the number of compromised parties within each sharing. However, not every possible set of parties should necessarily be considered acceptable. In realistic deployments, the parties that are acceptable for storing and processing private data may depend on the client, application, administrative domain, or trust policy associated with that data. The system therefore uses trust quorums to specify which sets of MPC parties may be used to maintain and operate over private values.

A trust quorum represents a user- or application-defined trust assumption. It defines a set of MPC parties and a threshold under which the owner of the data is willing to place private state in that set. This follows the intuition of asymmetric trust and quorum systems, where different users may accept different coalitions of parties instead of relying on a single global trust assumption [27, 28]. In this thesis, trust quorums are used to determine which party sets are authorised to hold shares of a protected value.

For example, consider the available parties $\{P_1, P_2, P_3, P_4, P_5\}$. A client may allow the quorum $\{P_1, P_2, P_3\}$ with threshold $t = 1$, meaning that any single party in that quorum learns nothing and any two shares are sufficient to reconstruct the value. The same client may also allow $\{P_3, P_4, P_5\}$ with the same threshold, because it trusts that at most one party in that set will be compromised. However, this does not imply that every subset or combination of the five available parties is acceptable. For instance, the client may reject $\{P_1, P_4, P_5\}$ if those parties do not satisfy its trust policy, even though the quorum has the same size and threshold.

This distinction is important during propagation and reconfiguration. Moving a private value to a new set of parties is safe only if the target set satisfies the trust policy associated with that value. Therefore, before shares are propagated or reshared to a target quorum, the secure layer must check that the target quorum is authorised. If the trust policy does not authorise a quorum, the system must reject the propagation or reconfiguration, even if the quorum would be valid from a purely threshold-secret-sharing perspective.

Trust quorums therefore separate two concerns. The CRDV layer determines the logical replicated state and stores handles. The trust-quorum policy determines which MPC parties are allowed to hold the shares referenced by those handles. Changing the active quorum of a replica does not change the handle stored in the CRDV layer and does not

modify the logical value represented by that handle. It only changes which MPC parties are trusted to maintain and operate over the corresponding shares.

In the prototype, trust-quorum policies are defined statically and used to validate share propagation and reconfiguration. This captures the main mechanism needed to enforce quorum-based trust assumptions, but it does not yet implement a full dynamic policy language. Two extensions are left for future work. First, multiple clients could specify different trust-quorum policies, and a replica configuration would be accepted only if it satisfies all relevant client policies. Second, trust policies could evolve over time. If a client changes the set of parties it is willing to trust, the system could trigger resharing for values currently stored outside the new trust domain, moving them to quorums that satisfy the updated policy.

Chapter 5

Secure Operations

This chapter describes the operations supported by the secure CRDV system. While Chapter 4 introduced the architecture of the system, this chapter focuses on how private values are created, updated, propagated, reconstructed, and reconfigured.

The operations are organised into three levels. First, Section 5.1 presents the low-level secure MPC operations provided by the secure layer. These operations manipulate secret-shared values identified by handles and are independent from any particular CRDT. Second, Section 5.2 describes how these primitives are composed with the CRDV layer to support secure counters, registers, maps, lists, and sets. Finally, Section 5.3 describes system-level operations that manage the distribution of shares across quorums, namely share propagation and quorum reconfiguration.

5.1. Secure MPC Operations

The secure layer provides a set of basic MPC operations used to represent and manipulate private values. These operations are independent from the semantics of a particular CRDT. Instead, they operate over secret-shared values identified by handles. Secure CRDT operations, described in Section 5.2, are implemented by composing these lower-level primitives with the replication and conflict-resolution mechanisms of the CRDV layer.

As introduced in Chapter 4, a handle is a unique, non-secret identifier used by both layers to name one logical private value. Each handle is assumed to uniquely identify one logical private value within the secure layer. The same handle may be associated with different sharings of that value over time, for example after share propagation or quorum reconfiguration, but it must not be reused for unrelated values. This assumption ensures that all MPC parties interpreting a handle refer to the same protected value, even if the set of parties holding shares changes. In the design, handle generation is treated as a system responsibility, while the concrete handle formats used by the prototype are described in Section 6.2.1.

5.1.1 Operation Types

The secure operations supported by the system can be classified according to whether they require communication between MPC parties. This classification does not mean that non-interactive operations are message-free. Rather, it distinguishes operations where parties do not communicate with each other from operations where parties must exchange intermediate values as part of the protocol.

The first class consists of non-interactive operations. These operations may involve sending shares to parties or collecting shares from them, but the MPC parties do not exchange intermediate values with one another. This class includes input, opening, and linear updates such as addition and subtraction.

The second class consists of interactive operations. These operations require MPC parties to exchange intermediate values before the resulting sharing or opened value can be obtained. This class includes resharing, multiplication, and declassification.

This distinction is useful because interactive operations require coordination among MPC parties and usually incur additional communication cost. The following subsections describe each secure operation using this terminology.

Table 5.1 summarises the secure operations provided by the secure layer and their interaction class.

Operation	Class	Description
Input	Non-interactive	Creates a secret-shared representation of a private value and associates the resulting shares with a handle.
Open	Non-interactive	Collects enough shares associated with a handle and reconstructs the corresponding plaintext value when authorised.
Add/Sub	Non-interactive	Updates the shares associated with a handle by adding or subtracting a secret-shared delta.
Reshare	Interactive	Redistributes the shares associated with a handle from a source quorum to a target quorum without changing the represented value.
Mult	Interactive	Computes a shared representation of the product of two private values through interaction between parties.
Declassify	Interactive	Opens a shared value when the result of a secure computation is intended to become public.

Table 5.1: Secure operation types provided by the secure layer.

5.1.2 Private Input and Opening

Private input creates the secret-shared representation of a private value inside the secure layer. Given a unique handle, a private value, and a quorum, the Input operation generates Shamir shares of the value and stores one share at each party under that handle. The handle is the identifier inserted into the CRDV state, while the shares remain stored by the parties of the selected quorum.

In this thesis, Input denotes the system-level operation that introduces a private value into the secure layer. The underlying cryptographic step is the generation of Shamir shares. This distinction is useful because other operations may also generate temporary shares without storing them as persistent values associated with a CRDV handle.

Algorithm 1 $\text{Input}(h, v, Q)$

Input: handle h , private value v , quorum Q

Output: shares of v associated with h stored by the parties in Q

- 1: Let P be the parties in quorum Q
 - 2: Generate shares $(s_1, \dots, s_n)$ of private value v
 - 3: **for** $i \leftarrow 1$ to n **do**
 - 4: Send share s_i to party p_i
 - 5: Party p_i stores (h, s_i)
 - 6: **end for**
-

Opening performs the inverse system-level operation. Given a handle and a quorum whose parties hold shares associated with that handle, the secure layer collects shares from the parties and reconstructs the corresponding plaintext value from a sufficient subset of them. This operation is only performed when the value is allowed to be revealed to an authorised client or component.

In this thesis, Open denotes the complete system-level operation: it retrieves shares by handle and applies the underlying Shamir reconstruction procedure. The interpolation step itself is the cryptographic reconstruction procedure described in Section 2.3.2. Private values therefore remain represented by handles unless an operation explicitly opens the plaintext result. Opening does not change the type or representation stored in the CRDV state: the replica continues to store the handle. It only returns a plaintext result to an authorised recipient for that request.

5.1.3 Addition and Subtraction

The secure layer supports arithmetic updates over private values. Addition and subtraction update a shared value by a private delta. Since the delta is itself private, the secure

Algorithm 2 $\text{Open}(h, Q)$

Input: handle h , quorum Q

Output: plaintext value v , or failure if not enough shares are available

```

1: Let  $P = \{p_1, \dots, p_n\}$  be the parties in quorum  $Q$ 
2:  $S \leftarrow \emptyset$ 
3: for  $i \leftarrow 1$  to  $n$  do
4:   Party  $p_i$  retrieves its share  $s_i$  associated with handle  $h$ 
5:   Party  $p_i$  sends  $s_i$  to the opening component
6:    $S \leftarrow S \cup \{s_i\}$ 
7: end for
8: if  $|S| < t + 1$  then
9:   return failure
10: end if
11: Select a subset  $S' \subseteq S$  such that  $|S'| = t + 1$ 
12: Reconstruct the plaintext value  $v$  from  $S'$  using interpolation
13: return  $v$ 

```

layer first generates a temporary Shamir sharing of Δ for the parties in the quorum. This uses the same underlying share generation procedure as Input, but the resulting delta shares are not associated with a persistent CRDV handle. Each party then combines its temporary delta share with its current share of the value associated with h . Algorithm 3 summarises this operation.

Algorithm 3 $\text{AddSub}(h, \Delta, Q, op)$

Input: handle h , private delta Δ , quorum Q , operation $op \in \{\text{Add}, \text{Sub}\}$

Output: updated shares associated with handle h

```

1: Let  $P = \{p_1, \dots, p_n\}$  be the parties in quorum  $Q$ 
2: Generate temporary shares  $(d_1, \dots, d_n)$  of private delta  $\Delta$ 
3: for  $i \leftarrow 1$  to  $n$  do
4:   Send temporary share  $d_i$  to party  $p_i$ 
5:   Party  $p_i$  retrieves its current share  $s_i$  associated with handle  $h$ 
6:   if  $op = \text{Add}$  then
7:     Party  $p_i$  stores  $(h, s_i + d_i)$ 
8:   else
9:     Party  $p_i$  stores  $(h, s_i - d_i)$ 
10:  end if
11: end for

```

This operation does not require the parties to exchange intermediate values with one another. Each party only manipulates its own local share, and the plaintext value is never reconstructed. For this reason, addition and subtraction are the basic update primitives used by secure counter operations.

5.1.4 Resharing

Resharing changes the set of parties that hold shares of a private value without changing the value itself. This operation is related to secret redistribution in threshold sharing schemes, where shares are moved from one access structure to another without reconstructing the underlying secret [29]. Given a handle, a source quorum, and a target quorum, the secure layer produces a new sharing of the same underlying value for the target quorum. The CRDV layer can continue to refer to the value through a handle, while the secure layer changes the parties responsible for maintaining the corresponding shares. Algorithm 4 summarises this protocol.

Algorithm 4 Reshare(h, Q_s, Q_t)

Input: handle h , source quorum Q_s , target quorum Q_t

Output: shares associated with h stored by the parties in Q_t

- 1: Let P_s be the parties in source quorum Q_s
 - 2: Let P_t be the parties in target quorum Q_t
 - 3: **for all** $p_i \in P_s$ **do**
 - 4: **at source party** p_i :
 - 5: retrieve current share s_i associated with h
 - 6: create shares $(r_{i,1}, \dots, r_{i,m})$ of s_i for parties in P_t
 - 7: send $r_{i,j}$ to target party p'_j
 - 8: **end for**
 - 9: **for all** $p'_j \in P_t$ **do**
 - 10: **at target party** p'_j :
 - 11: collect pieces $\{r_{i,j} \mid p_i \in P_s\}$
 - 12: reconstruct new share s'_j
 - 13: store (h, s'_j)
 - 14: **end for**
-

Resharing preserves the logical value represented by the handle while changing the set of parties that maintain its shares. This operation is used to support quorum changes, share propagation, and reconfiguration.

5.1.5 Multiplication

Multiplication builds on the same redistribution pattern used by Reshare. In resharing, each source party creates a fresh sharing of its current share and sends one piece to each target party. Multiplication follows the same communication structure, but each party redistributes the product of its two local input shares instead of redistributing a single existing share.

If two values are shared using polynomials of degree t , then multiplying the local shares produces points of a polynomial of degree $2t$. This would require $2t + 1$ shares to reconstruct and would effectively increase the reconstruction threshold. To avoid this, multiplication combines local products with a degree-reduction step. Each party first computes the product of its two local input shares. Then, each party creates a fresh degree- t sharing of this local product contribution and distributes the resulting pieces to the other parties. Each receiver combines the pieces it receives to obtain one output share. The final result is therefore a fresh degree- t sharing of the product, rather than a degree- $2t$ sharing.

This protocol follows the standard approach for multiplying Shamir-shared values described by Gennaro, Rabin, and Rabin [30]. Under the notation used in this thesis, where degree- t sharings are reconstructed from $t + 1$ shares, the intermediate product polynomial requires $n \geq 2t + 1$ parties. Algorithm 5 summarises the multiplication protocol.

Algorithm 5 $\text{Mult}(h_z, h_x, h_y, Q)$

Input: output handle h_z , input handles h_x and h_y , quorum Q

Output: shares associated with h_z representing $x \cdot y$

- 1: Let P be the parties in quorum Q
 - 2: **for all** $p_i \in P$ **do**
 - 3: **at source party** p_i :
 - 4: retrieve shares s_i^x and s_i^y
 - 5: compute local product $m_i \leftarrow s_i^x \cdot s_i^y$
 - 6: create shares $(m_{i,1}, \dots, m_{i,n})$ of m_i for parties in P
 - 7: send $m_{i,j}$ to party p_j
 - 8: **end for**
 - 9: **for all** $p_j \in P$ **do**
 - 10: **at receiver party** p_j :
 - 11: collect pieces $\{m_{i,j} \mid p_i \in P\}$
 - 12: reconstruct local output share s_j^z
 - 13: store (h_z, s_j^z)
 - 14: **end for**
-

5.1.6 Declassification

Declassification is the explicit decision to make a shared value public inside the secure layer. While `Open` returns a plaintext value to an authorised external client or component, declassification is used when the result of a secure computation is intended to become public to the parties participating in the protocol. This distinction makes clear that private values remain protected unless an operation explicitly reveals them. Algorithm 6 summarises the declassification protocol.

Algorithm 6 Declassify(h, Q)

Input: handle h , quorum Q

Output: value associated with h stored as public in the parties of Q

```

1: Let  $P$  be the parties in quorum  $Q$ 
2: for all  $p_i \in P$  do
3:   at source party  $p_i$ :
4:     retrieve local share  $s_i$  associated with  $h$ 
5:     send  $s_i$  to the parties in  $P$ 
6: end for
7: for all  $p_j \in P$  do
8:   at receiver party  $p_j$ :
9:     collect shares  $\{s_i \mid p_i \in P\}$ 
10:    reconstruct public value  $v$ 
11:    store  $(h, v)$  as the declassified value
12: end for
  
```

After declassification, the value associated with the handle is no longer treated as a protected secret by the secure layer for that operation. Declassification is therefore used only when the result of a secure computation is intended to become visible to the participating parties.

The operations above form the core secure interface used by the system design. Other operations may appear in the implementation as auxiliary mechanisms. For example, batched resharing applies the resharing protocol to multiple handles at once, reducing repeated coordination overhead during reconfiguration. Random share generation can be used as an auxiliary primitive in protocols that require masking or probabilistic tests. Share deletion and state reset operations are management mechanisms used to maintain or initialise party state, but they do not define new secure data-type semantics. For this reason, they are discussed as implementation details rather than as separate abstract operations.

5.2. Secure CRDV Operations

The secure MPC operations described in the previous section provide the building blocks for manipulating private values. This section describes how those operations are composed to implement secure replicated data types. The goal is not to replace the CRDV representation of replicated state, but to extend it so that selected values can be protected through secret sharing while still participating in the normal replication and convergence mechanisms of CRDV.

A secure CRDT operation follows the same high-level structure as its public counterpart. The difference is that values marked as private are not inserted directly into the CRDV

state. Instead, the secure layer creates or updates a secret-shared representation of the value and returns a handle. The CRDV layer stores and propagates this handle as the representation of the private value. As a result, replicas can converge on the same logical object while the corresponding sensitive fields remain protected by the MPC layer.

Read operations are handled according to the representation stored in the CRDV layer. If the stored value is public, it can be returned directly by the CRDV layer. If the stored value is a handle, the system does not need a data-type-specific read protocol: when the client wants the plaintext value, the handle is passed to the Open operation defined in Section 5.1.2. For this reason, the following subsections focus mainly on update operations that create, modify, propagate, or compare private values.

The secure data types considered in this design correspond to the CRDT abstractions implemented by the underlying CRDV framework: registers, counters, sets, maps, and lists. These types provide the building blocks for representing private fields and structured replicated objects in the workloads evaluated in this thesis. More complex secure objects can be built by composing these secure data types.

5.2.1 Secure Counters

A secure counter represents a numerical value whose updates may be private. When the counter value is public, increments and decrements are applied directly by the CRDV layer. When the counter value is private, the system uses a stable handle for the local counter component at the current replica. The first private update creates the handle and initialises the corresponding shared state in the secure layer. Later increments and decrements reuse the same handle and update only the shares associated with it.

This design follows the local-component nature of replicated counters. Each replica maintains its own counter component, and therefore only the handle associated with the local replica is updated by a local increment or decrement. The CRDV layer stores the handle as the representation of the local private component, while the secure layer maintains the evolving numerical value. Algorithm 7 summarises this update procedure.

As a result, the handle stored in the replicated state remains stable over time, while the secret-shared value associated with that handle evolves inside the secure layer. The CRDV conflict-resolution logic remains unchanged, because the replicated state still contains the local counter component expected by the underlying counter representation.

Algorithm 7 SecureCounterUpdate(c, Δ, Q, op)

Input: counter identifier c , value Δ , quorum Q , operation $op \in \{\text{Inc}, \text{Dec}\}$

Output: counter update inserted into the CRDV state

```

1: if PrivacyTag( $c, \text{data}$ ) = public then
2:   insert public counter update ( $c, \Delta, op$ ) into the CRDV layer
3: else
4:    $h \leftarrow$  current local handle associated with counter  $c$ 
5:   if  $h = \perp$  then
6:      $h \leftarrow$  NewHandle( $c$ )
7:     Input( $h, \Delta, Q$ )
8:   else
9:     if  $op = \text{Inc}$  then
10:      AddSub( $h, \Delta, Q, \text{Add}$ )
11:     else
12:      AddSub( $h, \Delta, Q, \text{Sub}$ )
13:     end if
14:   end if
15:   propagate shares associated with  $h$  to active replica quorums
16:   insert counter update ( $c, h$ ) into the CRDV layer
17: end if

```

This protects the counter value and private deltas, but not necessarily the operation pattern: when increments or decrements are public constants and the initial value is public, an observer can infer the value from the visible sequence of operations. Such counters remain useful when deltas are private or irregular, but applications requiring this stronger metadata protection need an oblivious operation design, which is outside this prototype's scope.

5.2.2 Secure Registers

A secure register stores a single logical value, which may be public or private. Public register assignments are executed exactly as in the original CRDV implementation. Private assignments are handled by storing a handle in the register and maintaining the actual value as shares in the secure layer.

As with counters, private register assignments use a stable handle for the replica-local register entry. The first private assignment creates the handle and shares the value in the secure layer. Later assignments reuse the same handle and replace the secret-shared value associated with it. The CRDV layer continues to use its normal register metadata to determine which value is visible.

The conflict-resolution semantics of the underlying register remain unchanged. A last-writer-wins register can continue to use the timestamps or ordering metadata maintained

by the CRDV layer. The secure layer only changes the representation of private payloads: instead of storing the plaintext value, the register stores the handle that identifies the corresponding shares. Unlike encryption-only approaches, this representation also permits supported computations over the private value without first decrypting it. Encryption-based registers could be preferable for workloads that only store and retrieve values, but they would require decryption at an authorised endpoint for computation; supporting both representations is a possible extension with distinct key-management and operation semantics.

5.2.3 Secure Maps

A secure map associates keys with values, where the keys may remain visible and the values may be private. Public values are inserted directly into the map as in the original CRDV implementation. Private values are replaced by handles associated with the map entry being updated.

This handle policy allows different entries of the same map to evolve independently. Each private key-value association has its own handle, and updates to one key do not affect the secret-shared state associated with other keys. The CRDV layer can therefore continue to use public keys for indexing, replication, and conflict resolution, while the secure layer protects the private payload stored under each key.

This design supports structured replicated objects in which public structure is combined with private fields. For example, a replicated object may expose public attribute names or identifiers while keeping the corresponding values protected as secret-shared state. Protecting map keys as well is conceivable, for example when keys are personal identifiers, but direct indexing and conflict resolution would then require private equality or deterministic protected tags. The latter would leak equality across entries unless carefully domain-separated, while the former would incur the search cost observed for secure sets. This design therefore deliberately keeps keys public.

5.2.4 Secure Lists

Secure lists represent ordered collections that may contain private elements. Public elements are inserted directly into the list. Private elements are first introduced into the secure layer using Input and then represented in the CRDV list by a handle. The list

structure, including ordering metadata and the semantics of insert, append, and prepend, remains managed by the CRDV layer.

Each private list element receives a distinct handle associated with the list identifier, the local replica, and the position or element identifier used by the underlying list representation. This allows the CRDV layer to order and replicate the list normally, while the secure layer protects the payload associated with each element.

Append, prepend, and insert follow the same pattern. The CRDV layer determines the position of the new element, while the secure layer creates the secret-shared representation of the private payload. The replicated list stores the resulting handle as the element value. The confidentiality guarantee applies to the values stored in the list, not to the ordering metadata, list size, or access patterns required by the CRDV layer. Public and private elements may coexist, but the prototype does not define value-based operations, such as sorting or comparison, across them. Such an operation would require a secure comparison protocol and an explicit policy for revealing or retaining the result; it does not send the complete list to the secure layer by default.

5.2.5 Secure Sets

Secure sets require special handling because of how set elements are represented in the underlying CRDV framework. In a regular set, the element itself is stored as the key of the update. When the element is private, the plaintext value cannot be used directly as the replicated key. Therefore, secure set operations use handles as the elements stored in the CRDV layer.

Unlike counters and registers, secure set handles are not stable per object. Instead, the system creates a candidate handle for each local set operation. This is necessary because the plaintext element cannot be used as the replicated key. Candidate handles are separated according to the kind of set operation, so that additions and removals can be represented without exposing the private element. Here, a namespace is the public partition consisting of one set identifier and one operation kind (the add or remove component); it determines the existing private values considered by a membership test.

This means that a secure remove does not directly delete an existing add handle. Instead, it creates a new private representation of the value to be removed and inserts the resulting handle as a remove update in the CRDV layer. The underlying set semantics then interpret the add and remove updates to determine whether the element belongs to the logical set.

The main challenge in secure sets is deciding whether a private value is already represented in the set without revealing the value itself. In the public implementation, this check can be performed by directly comparing the element stored as the set key. In the secure setting, however, replicas only observe handles, and equality between the underlying private values must be tested by the secure layer.

There are different ways to design a secure set, depending on the desired trade-off between leakage and performance. At one extreme, the system could try to hide not only the values, but also the size of the set, the position of matches, and whether an insertion failed because the element was already present. This would require padding the set to a fixed maximum size and evaluating the same number of secure comparisons for every operation. Such an approach would provide stronger privacy, but would be expensive and would require choosing a maximum set size in advance.

At the other extreme, the system could compare the candidate element with existing elements one by one and stop as soon as a duplicate is found. This would be more efficient when duplicates occur, but it would directly reveal that a match was found and could also leak information through the number of comparisons performed.

This work adopts an intermediate point in this design space and makes the leakage of the functionality explicit. The secure set functionality leaks the size of the relevant namespace and the public outcome of the update, namely whether the candidate operation is accepted or rejected. However, the membership test does not leak which existing handle, if any, matched the candidate. The only value-dependent information revealed by the secure layer is the boolean result required by the set operation: whether the candidate is already represented in the relevant namespace.

To implement this trade-off, this work adapts a private equality protocol over secret-shared values and generalises it to a set-membership test. Instead of checking equality between only two private values, the protocol compares a candidate value against all private values already represented in the relevant set namespace. The output is public only at the level required by the set operation: it reveals whether the candidate is already represented, but not which existing handle matched it or what the private value is.

For both add and remove operations, the secure layer first creates a temporary sharing of the candidate value and compares it against the existing values in the corresponding namespace. For an add operation, the candidate is compared against existing add

handles. For a remove operation, the candidate is compared against existing remove handles. If the value is already represented, the candidate handle and temporary sharing are discarded and no replicated update is inserted. Otherwise, the handle is accepted, propagated, and inserted into the CRDV set. Algorithm 8 summarises this operation.

Algorithm 8 SecureSetUpdate(S, x, Q, op)

Input: set identifier S , value x , quorum Q , operation $op \in \{\text{Add}, \text{Remove}\}$

Output: set update inserted for x if the operation is accepted

```

1: if PrivacyTag( $S, \text{key}$ ) = public then
2:   insert public set update ( $S, x, op$ ) into the CRDV layer
3: else
4:   if  $op = \text{Add}$  then
5:      $N \leftarrow \text{add}$ 
6:   else
7:      $N \leftarrow \text{remove}$ 
8:   end if
9:    $h \leftarrow \text{NewCandidateHandle}(S, N)$ 
10:  Generate temporary shares  $[x]$  of candidate value  $x$  for quorum  $Q$ 
11:   $Y \leftarrow$  secure values represented by existing handles in namespace  $N$  for  $S$ 
12:  if not PrivateSetMembershipTest( $[x], Y$ ) then
13:    store temporary shares  $[x]$  under handle  $h$ 
14:    propagate shares associated with  $h$  to active replica quorums
15:    insert set update ( $S, h, op$ ) into the CRDV layer
16:  else
17:    discard candidate handle  $h$  and temporary sharing  $[x]$ 
18:  end if
19: end if

```

Private set membership test.

Private set membership is a well-studied problem in privacy-preserving computation, closely related to private set intersection and other secure set operation protocols. In general, these protocols aim to determine whether an element belongs to a set, or to compute relations between private sets, while limiting the information revealed beyond the intended output [31, 32]. In this thesis, the problem appears in a restricted form: the system only needs to determine whether a private candidate value is already represented in a secure CRDV set namespace.

The private set membership test is the key mechanism that makes secure set operations possible in this setting. It is derived from the equality-with-public-result protocol for secret-shared values described by Turban [33]. That protocol reduces equality between two private values to testing whether their secret-shared difference is zero. The difference is

multiplied by a fresh secret-shared random value, and only the masked result is declassified. This reveals the equality result while hiding the values being compared.

This work generalises this idea from a pairwise equality test to a set-membership test. Instead of comparing a candidate value with a single existing private value, the protocol compares the candidate against all private values already represented in the relevant set namespace. The output is public only at the level required by the set operation: it reveals whether the candidate is already represented, but not the candidate value, the existing values, or which existing handle caused the match.

Let $Y = \{[y_1], \dots, [y_n]\}$ be the existing private values represented by the handles in a given namespace, and let $[x]$ be the secret-shared candidate value. For each existing value $[y_i]$, the protocol computes the shared difference $[z_i] = [x] - [y_i]$, masks it with a fresh shared random value $[r_i]$, and obtains the masked difference $[d_i] = [r_i] \cdot [z_i]$. Instead of declassifying one masked difference per existing value, the protocol multiplies all masked differences into a single shared aggregate:

$$D = \prod_{i=1}^n r_i \cdot (x - y_i).$$

Only this aggregate is declassified. If $D \neq 0$, then all differences are non-zero and the candidate value is not present in the namespace. If $D = 0$, then at least one factor is zero, but this may be caused either by a true match or by one of the random masks being zero. To disambiguate these cases, the protocol also declassifies

$$R = \prod_{i=1}^n r_i.$$

If $R = 0$, the test is inconclusive and is repeated with fresh randomness. If $R \neq 0$, then the zero value of D must have been caused by at least one zero difference, meaning that the candidate value is already represented. Since the random masks are sampled from a finite field of large order, such as $\mathbb{F}_p$, the probability that some r_i is zero is negligible in practice, so the loop rarely needs more than one iteration.

Algorithm 9 abstracts this generalised membership protocol. The output is intentionally public, because the set operation must decide whether to insert a replicated update. However, the output only reveals whether the candidate is already represented in the namespace. It does not reveal the candidate value, the existing set values, or which existing handle caused the match.

This approach ensures that private set elements are never inserted into the replicated

Algorithm 9 PrivateSetMembershipTest($[x], Y$)

Input: private candidate value $[x]$, secure collection $Y = \{[y_1], \dots, [y_n]\}$

Output: **true** if x is already represented in Y ; **false** otherwise

```

1: if  $Y = \emptyset$  then
2:   return false
3: end if
4: repeat
5:   for  $i \leftarrow 1$  to  $n$  do
6:      $[z_i] \leftarrow [x] - [y_i]$ 
7:      $[r_i] \leftarrow \text{Random}()$ 
8:      $[d_i] \leftarrow [r_i] \cdot [z_i]$ 
9:   end for
10:   $[D] \leftarrow \prod_{i=1}^n [d_i]$ 
11:   $D \leftarrow$  declassified value of  $[D]$ 
12:  if  $D \neq 0$  then
13:    return false
14:  end if
15:   $[R] \leftarrow \prod_{i=1}^n [r_i]$ 
16:   $R \leftarrow$  declassified value of  $[R]$ 
17: until  $R \neq 0$ 
18: return true

```

state in plaintext. The CRDV layer only observes handles as set elements, while the secure layer decides whether a candidate add or remove operation should produce a replicated update. The contribution of this mechanism is the adaptation of pairwise private equality into a set-level membership test that can be used inside the CRDV set semantics without exposing private set elements.

5.3. System-level Secure Operations

The previous sections described how private values are represented and how secure CRDT operations manipulate those values. However, the system must also manage the placement of the shares that represent private values. System-level secure operations are responsible for moving or redistributing secret-shared state between quorums without changing the logical values stored in the CRDV layer.

5.3.1 Share Propagation

Share propagation makes the shares associated with a handle available to the quorums of other active replicas. It complements the normal CRDV propagation mechanism: the CRDV layer propagates handles and metadata, while the secure layer ensures that the corresponding shares are available to the MPC parties associated with replicas that may later access those handles.

When a private value is inserted into the CRDV state, the replicated object contains only a handle. This handle is sufficient for the CRDV layer to replicate and resolve the update, but it is not sufficient for another replica to execute secure operations over the value unless the parties associated with that replica also hold shares of the value. Share propagation addresses this by resharing the value from the local quorum to the quorums associated with the active replicas.

Conceptually, share propagation is a wrapper around Reshare (Algorithm 4). It applies the same secret redistribution idea at the system level: the logical value remains the same, but shares are made available to the quorums of other active replicas. Given a handle and a set of target replicas, the system determines the active quorum of each target replica and reshapes the value from the source quorum to each target quorum.

After share propagation, the handle stored in the CRDV state can be used by other replicas whose quorums received the corresponding shares. This preserves the separation between replication and confidentiality: the CRDV layer propagates the logical update, while the secure layer propagates the secret-shared state required to operate on the private value.

5.3.2 Quorum Reconfiguration

Chapter 4 introduced trust quorums as the mechanism used to determine which MPC parties may maintain and operate over private values. Quorum reconfiguration is the system-level operation that applies changes in those policies to secret-shared state that already exists in the system.

Conceptually, reconfiguration creates a fresh sharing of existing protected state under another authorised quorum. This can be understood as making the protected values available to a new secure representation, possibly associated with a new or updated replica configuration, without changing the logical values stored in the CRDV layer. The important point is not that the plaintext value moves, but that the set of MPC parties trusted to maintain and process the corresponding shares changes.

Like share propagation, reconfiguration is built on top of Reshare (Algorithm 4). For each handle affected by the quorum change, the secure layer creates a new sharing of the same underlying value under the target quorum. The plaintext value is not reconstructed during this process. The handle stored in the CRDV layer remains unchanged; only the set of MPC parties responsible for storing the corresponding shares changes.

After the affected handles have been reshared, the system updates the active quorum associated with the replica. Future secure operations issued at that replica use the new quorum. Shares stored only for the previous quorum are then removed from parties that are no longer part of the replica's active trust policy. This prevents the old quorum from continuing to act as the responsible party set after the policy change.

Reconfiguration is different from share propagation. Share propagation makes shares available to the quorums of other active replicas so that replicated handles can be used remotely. It does not necessarily change the trust policy of the source replica or remove the original shares. Reconfiguration changes the active quorum of a replica and removes shares from parties that are no longer part of that replica's trust policy.

Chapter 6

Implementation

This chapter describes the implementation of the secure CRDV prototype. Chapter 4 introduced the overall architecture, while Chapter 5 described the secure operations supported by the system. This chapter focuses on the concrete prototype choices used to realise those ideas.

The source code of the prototype developed for this thesis, together with the scripts used to run the experimental evaluation, is available in a public repository¹. The version corresponding to the implementation described in this chapter and to the evaluation reported in Chapter 7 is identified by the tag `v1.0-thesis`.

Figure 6.1 shows how the architectural layers are mapped to implementation components. The existing PostgreSQL-based CRDV implementation provides the replicated data layer and the SQL-facing interface. The secure layer is implemented in Rust and connects the CRDV layer to the choreographic MPC runtime.

The figure distinguishes two paths through which application requests may reach the secure layer. Application clients interact with the CRDV replica through SQL operations over public values and handles. Operations involving plaintext private inputs or outputs enter the secure layer directly, so that plaintext private values are not stored in the CRDV tables. Handle-based secure requests issued by the CRDV replica are also forwarded to the secure layer, where they are executed by the choreographic MPC runtime.

Solid arrows in the figure represent component-level interactions, such as SQL operations or handle-based secure requests, while dashed arrows represent inter-process communication. In this thesis, choreographic MPC refers to the implementation of MPC protocols using a choreographic programming model, where the communication structure of a secure protocol is described globally and then projected to the local behaviour of each participant. This runtime is introduced here as part of the implementation overview and described in detail in Section 6.4.

¹<https://github.com/ricarim/secure-crdv>

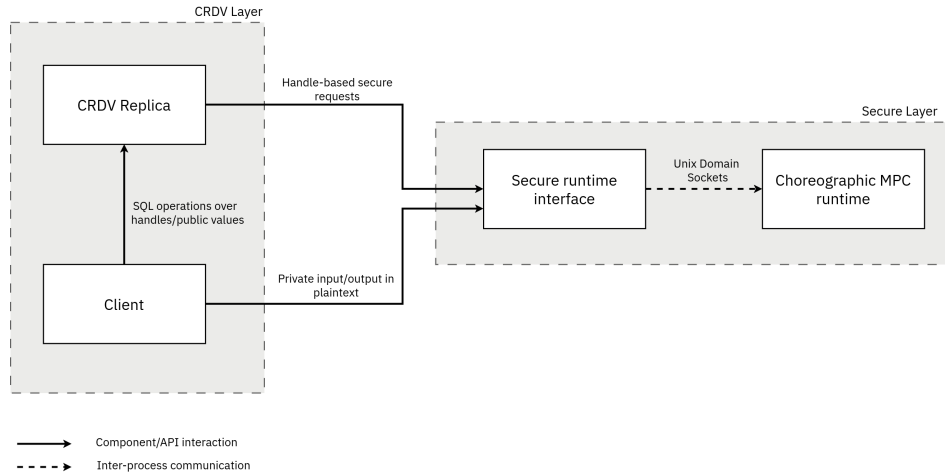


Figure 6.1: High-level implementation overview of the secure CRDV prototype.

The following sections describe the technologies used, the integration with the existing CRDV implementation, the concrete handle-generation scheme, the runtime interface, the choreographic implementation of the secure protocols, and the main limitations of the prototype.

6.1. Technology Stack

The prototype was implemented on top of the PostgreSQL-based implementation of the CRDV framework. Since CRDV is not inherently tied to a specific database system, PostgreSQL 16.14 [34] should be understood as the concrete backend used by this prototype rather than as a requirement of the model. This choice allowed the implementation to reuse the storage model, metadata, conflict-resolution logic, and SQL-facing interface already provided by the existing CRDV implementation.

The components added by this work were implemented in Rust 1.92.0 [35]. These components include the PostgreSQL extension, the secure runtime interface, the secure runtime, and the logic required to execute secure operations over private values. The Rust workspace and its dependencies were managed using Cargo 1.92.0 [36]. Rust was chosen because it is well suited for systems programming and provides strong type-safety and memory-safety guarantees, which are useful in an implementation that combines database extensions, runtime interfaces, concurrent execution, inter-process communication, and secure protocol execution.

The integration between PostgreSQL and Rust is implemented using `pgrex` 0.12.9 [37]. This framework allows Rust functions to be exposed as SQL-callable primitives. In the prototype, these primitives form the boundary between the PostgreSQL-based CRDV layer

and the secure runtime: CRDV operations invoke SQL functions, which in turn call Rust code through `pgrx`.

The secure runtime is implemented using Chorus 0.5.0 [38], through the `chorus_lib` crate. It is the software component that executes the secret-sharing protocols and manages party-local shares. Chorus is used to implement the choreographic MPC runtime described later in this chapter, where runtime participants are represented as choreography locations. Its confidentiality properties follow from the distribution of shares among parties under the stated threat model.

Communication with the secure runtime is implemented using Unix Domain Sockets. Request and response messages are serialised using `serde` and `bincode`, specifically versions 1.0.228 and 1.3.3, respectively [39, 40].

The prototype also includes an internal Rust implementation of SSS, based on Shamir's original construction [14] and the implementation-oriented description in [41]. This implementation provides the finite-field arithmetic, share generation, and reconstruction used by the secure runtime.

6.2. CRDV Integration

The integration was implemented by extending the SQL and PL/pgSQL functions of the existing PostgreSQL-based CRDV prototype. The original CRDV relations continue to store operation records, metadata, timestamps, and the information required for conflict resolution. Public values are stored and manipulated as before. Private values, however, are represented by handles before being inserted into the replicated state.

Applications continue to manipulate replicated objects through SQL operations, without interacting directly with MPC parties or managing secret shares. The main change is that the implemented CRDV operations now include privacy-aware branches. When a field is public, the operation follows the original plaintext execution path. When a field is private, the value received by the CRDV operation is already a handle that identifies the corresponding secret-shared value in the secure runtime. The CRDV layer stores this handle in the replicated state and uses it as the representation of the private value.

This means that plaintext private values are not routed through the PostgreSQL-based CRDV execution path. For operations that require a new private value, such as private insertion or update, the plaintext value is first submitted to the secure layer, which creates

or updates the corresponding secret-shared representation and returns a handle. The application then invokes the corresponding CRDV operation using that handle. Operations that are already expressed over existing handles, such as resharing, reconfiguration, or handle-based secure updates, are issued from the CRDV layer over those handles. The concrete coordinator-facing interfaces that support these two paths are described in Section 6.3.

Listing 6.1 shows the general pattern followed by the implemented operations. The listing is intentionally simplified: its purpose is to illustrate the integration point between the CRDV layer and the secure runtime, rather than to present the complete implementation of a specific data type.

```

1 IF privacy_tag(object, field) = 'private' THEN
2     stored_value := received_handle;
3     PERFORM crdv.reshare_to_active_replicas(stored_value);
4 ELSE
5     stored_value := plaintext_operation(value);
6 END IF;
7
8 INSERT INTO Data (... , stored_value, ...);

```

Listing 6.1: Simplified handle-based pattern used by secure CRDV operations.

For public fields, the operation computes and stores the value directly in PostgreSQL, preserving the behaviour of the original CRDV implementation. For private fields, the operation stores the handle received from the application. If the protected value must be made available to the active replicas, resharing is invoked over that handle. The actual private value is maintained by the secure runtime as secret-shared state and is identified by the handle.

This pattern is used across the implemented data types, including counters, registers, maps, lists, and sets. The main differences between data types are the way handles are generated and the secure primitive associated with each operation. These differences are a consequence of the underlying representation used by each CRDV data type.

The integration keeps the responsibilities of the two layers separate. The CRDV layer continues to manage replication, causality metadata, and convergence, while the secure runtime manages shares and executes operations over private values. Handles form the connection between both layers: they are ordinary replicated values from the perspective

of the CRDV layer, but identifiers of distributed secret-shared state from the perspective of the secure runtime.

6.2.1 Handle Generation

Handles are the identifiers stored in the CRDV state in place of private values. A handle format is the structured textual convention used to construct such an identifier: its colon-separated fields identify the data type, logical object, originating replica, and, where required, the relevant key, position, or operation. It is not an encrypted value, a secret share, or a cryptographic key. For operations involving plaintext private inputs, these handles are obtained before the corresponding CRDV operation is invoked, as described in Section 6.3. Their format follows the workflow of the original CRDV implementation as closely as possible.

Table 6.1 summarises the concrete handle formats used by the prototype. Each format reflects the granularity at which the corresponding CRDV data type stores or updates state. Counters and registers use stable per-replica handles, maps and lists include the structural component being updated, and sets use operation-specific candidate handles because private set elements cannot be exposed as replicated keys.

Data type	Handle format	Reason
Counter	<code>c:id:site</code>	Follows the local-component representation of counters.
Register	<code>r:id:site</code>	Follows the per-replica update representation of registers.
Map	<code>m:id:site:key</code>	Associates one handle with each map entry.
List	<code>l:id:site:index</code>	Associates one handle with each list element or position.
Set	<code>s:id:site:op:count</code>	Represents a private set operation without exposing the element as a key.

Table 6.1: Concrete handle formats used by the prototype.

Counters and registers use stable per-replica handles because their original CRDV workflow updates existing entries rather than creating a new replicated entry for every operation. For counters, the implementation stores one component per object and replica, and increments or decrements update that component. For registers, updates from a replica are associated with the metadata used by the CRDV layer to determine the visible value. The secure implementation therefore uses handles of the form `c:id:site` and `r:id:site`. Later operations reuse the same handle and update the secret-shared value associated with it. If a fresh handle were generated for every counter or register operation, the secure version would change the storage behaviour of the original CRDV implementation.

Maps and lists use handles that follow the structural units already present in the corresponding CRDV objects. In maps, the relevant unit is the key: each key can be associated with an independent value, so the implementation includes the key in the handle and uses the format `m:id:site:key`. The handle therefore replaces only the value stored at that key, while the key remains part of the visible CRDV metadata used to identify and merge map entries. Lists follow the same idea, but the relevant unit is the list position. Since the CRDV layer already maintains ordering metadata, the secure implementation uses handles of the form `l:id:site:index` to replace the private payload associated with a specific element. In both cases, the handle is attached to an existing structural unit instead of changing how the object is organised by the CRDV layer.

Sets require a different handle format because their CRDV representation uses the element itself as part of the replicated key. In the public case, this allows the implementation to identify when an inserted element is equal to an existing one and to preserve set semantics directly in the CRDV state. In the secure case, the element cannot be used as the replicated key because its plaintext value must remain hidden.

For this reason, the secure implementation generates candidate handles for set operations using the format `s:id:site:op:count`. The set identifier and replica identifier preserve the origin of the operation, the operation field distinguishes additions from removals, and the local counter gives each candidate operation a distinct identifier. This still follows the workflow of the CRDV set: a handle is inserted only when the operation should produce a replicated update. However, because the CRDV layer cannot compare private elements directly, duplicate private insertions are discarded rather than represented by replacing or reusing an existing replicated key. The secure logic that decides whether a candidate handle is accepted is described in Section 5.2.5.

Overall, handle generation mirrors the granularity at which the original CRDV implementation stores and updates state. Counters and registers use stable handles because the original workflow updates existing entries. Maps and lists include keys or positions because the original workflow distinguishes individual structural units. Sets require operation-specific candidate handles because the private value cannot appear directly as the replicated key.

6.3. Secure Runtime Interface

The secure runtime interface connects the secure CRDV prototype to the Rust runtime responsible for managing shares and executing secure protocols. This interface is exposed through two coordinator-facing components. The first is the client-facing coordinator, used by application clients for operations that involve plaintext private inputs or outputs. The second is the CRDV-facing coordinator, used by the PostgreSQL extension to invoke secure operations over existing handles.

The client-facing coordinator is used for operations that should not route plaintext private values through PostgreSQL. For private input or update operations, it receives the plaintext value from the application client, creates or updates the corresponding secret-shared representation in the secure runtime, and returns a handle. For reconstruction, it receives a handle, obtains the required shares through the runtime, and returns the opened value to the requesting client.

The CRDV-facing coordinator is implemented as part of the Rust extension. After a privacy-aware CRDV operation invokes an `mpc_*` primitive over a handle, the request must leave the PostgreSQL execution context and be executed by the secure runtime. The `pgrx` extension exposes Rust functions as SQL-callable primitives, while the CRDV-facing coordinator translates those calls into requests understood by the choreographic MPC runtime.

Figure 6.2 shows these coordinator-facing interfaces. The client-facing coordinator receives plaintext private inputs or handles directly from the application client, while the CRDV-facing coordinator is reached from PostgreSQL through `pgrx`. Both coordinators submit requests to the choreographic MPC runtime through Unix Domain Sockets.

The term coordinator is used for two system-boundary components and for one runtime participant. The client-facing and CRDV-facing coordinators are system-boundary components: they receive requests from outside the secure runtime, resolve the active quorum, submit runtime requests, and return responses to their callers. The coordinator location inside the Chorus runtime is different: it participates in the choreography itself and executes the selected global protocol with the MPC parties.

Each coordinator-facing component is associated with a Chorus coordinator location. For the CRDV-facing coordinator, this association allows a request issued by a given CRDV

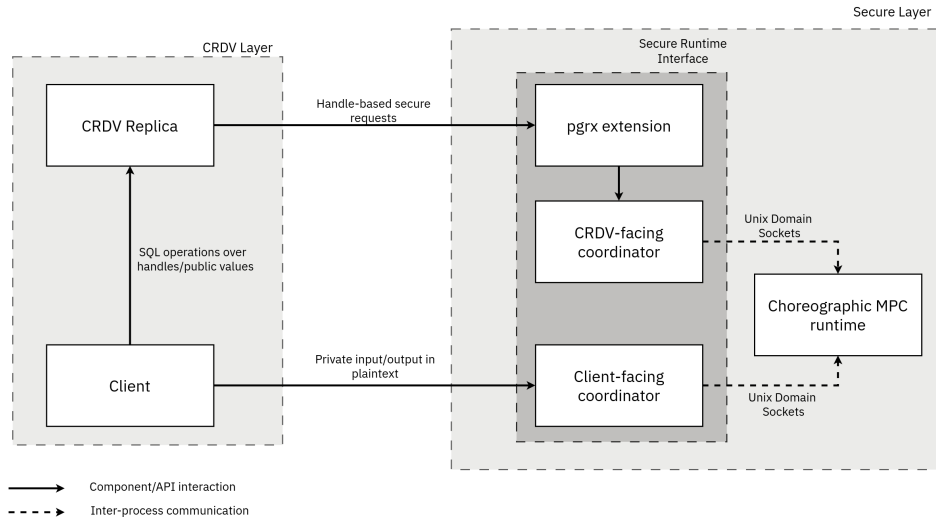


Figure 6.2: Coordinator-facing interfaces to the choreographic MPC runtime.

replica to be routed to the corresponding runtime participant. For the client-facing coordinator, it allows plaintext private input and output requests to be routed to the coordinator location responsible for the selected quorum. From that point on, the operation is executed according to the choreography selected for the current quorum.

The SQL layer does not interact directly with MPC parties. Instead, it invokes functions for handle-based secure updates, resharing, and reconfiguration. Operations that involve plaintext private input or output, such as private input creation and reconstruction, are handled by the client-facing coordinator rather than by routing plaintext values through the CRDV tables.

Listing 6.2 shows a simplified version of the request interface used between the coordinator-facing components and the secure runtime. The actual implementation contains additional variants and fields, but the listing illustrates the main categories of requests submitted to the runtime.

```
1 enum CoordinatorReq {
2     SetShares { handle, quorum, shares },
3     GetShares { handle, quorum },
4     UpdateShares { handle, quorum, operation },
5     Reshare { handle, from_quorum, to_quorum },
6     Reconfigure { target_quorum },
7 }
```

Listing 6.2: Simplified coordinator request interface.

Input operations generate shares for a private value and send them to the parties in the selected quorum. In the prototype, these operations are issued through the client-facing coordinator when they involve plaintext private values. Update operations instruct parties to modify the shares associated with an existing handle. Reconstruction operations request shares from the runtime and reconstruct the plaintext value only when an explicit client request requires it. Resharing and reconfiguration operations update the placement of shares across quorums, following the protocols described in Chapter 5.

From the perspective of PostgreSQL, requests issued through `mpc_*` primitives are synchronous. A call only returns after the CRDV-facing coordinator receives a response from the secure runtime. This is necessary because the surrounding CRDV operation may need the result immediately, for example to decide whether to insert a secure set operation or to complete a resharing request.

The coordinator-facing components do not persist secret shares. Shares are sent to the secure runtime and stored by the MPC parties. The coordinators therefore act as orchestration components: they connect application clients or PostgreSQL operations to the secure runtime, resolve the quorum used by each operation, dispatch requests, and return responses to their caller.

6.4. Choreographic MPC Implementation

The secure runtime was implemented using Chorus, following a choreographic programming model. In this model, a distributed protocol is described from a global perspective: the program specifies which participants interact and which messages are exchanged between them. From this global description, the framework derives the local behaviour executed by each participant through endpoint projection.

Figure 6.3 illustrates the endpoint projection process used by the Chorus-based runtime. The global choreography at the top represents the complete communication structure of an MPC protocol, including all messages between the coordinator location and the MPC parties. Endpoint projection then derives one local endpoint program for each participant. Each endpoint contains only the send, receive, and local actions relevant to that participant, while the set of generated endpoints preserves the communication behaviour of the original choreography.

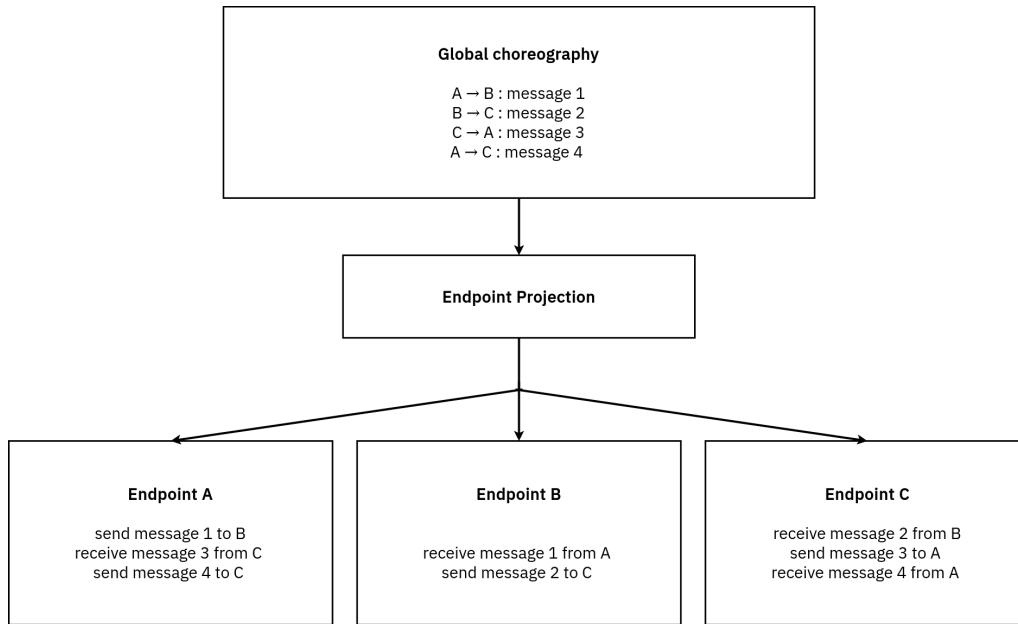


Figure 6.3: Endpoint projection overview.

This differs from a traditional implementation where each participant is programmed independently. In that setting, the developer must manually ensure that every send operation has a matching receive operation and that all participants follow compatible protocol steps. With choreographic programming, the communication structure is written once, at the global level, and the local endpoint behaviours are derived from that specification.

In this prototype, the participants of the choreography are the Chorus coordinator locations and the MPC parties. Requests reach the secure runtime either through the client-facing coordinator, when plaintext private input or output is involved, or through the CRDV-facing coordinator, when the operation is expressed over existing handles. These coordinator-facing components are described in Section 6.3 and send requests to the secure runtime through Unix Domain Sockets. Inside the runtime, the associated Chorus coordinator location receives the request and participates in the choreography that executes the corresponding secure protocol.

The runtime configuration defines the available coordinator locations, parties, quorums, and default coordinator-to-quorum assignments. This configuration is used to generate the Chorus location and quorum structures required by the runtime. As a result, each configured quorum has a corresponding communication structure known by the choreography, while the coordinator can still select among the predefined quorums at runtime.

The main choreography runs continuously and gives each Chorus coordinator location an opportunity to process pending requests. When a request is available, the coordinator

location obtains it from the runtime input queue, resolves the quorum associated with that request, and executes the choreography corresponding to the selected quorum. If no request is pending, the choreography performs no secure operation for that coordinator, but still preserves the global execution structure expected by Chorus.

The implemented protocols follow a staged communication structure. Every secure operation starts with a coordinator-to-party phase, where the Chorus coordinator location sends the request to the parties in the selected quorum. For simple operations, such as input sharing, reconstruction, and linear updates, this phase is sufficient: parties access their local state and return a response to the coordinator.

Some operations then continue with an additional party-to-party phase. Multiplication, resharing, random value generation, and declassification require parties to exchange intermediate values before the final result can be produced. This phase follows a many-to-many communication pattern between the source and target quorums: each source party prepares intermediate values for the target parties, and each target party combines the values addressed to it. Therefore, party-to-party communication is not an alternative entry point into the choreography, but an optional second phase initiated after the coordinator has started the protocol.

Figure 6.4 summarises this staged communication structure.

Each MPC party maintains its own local share state. This state maps handles to the shares held by that party. The CRDV layer never accesses this state directly: it only stores and replicates handles. Therefore, a private value is represented by a handle in the replicated CRDV state and by a set of shares distributed across the parties of the selected quorum.

Resharing illustrates the optional party-to-party phase. The coordinator location receives a request to move the shares associated with a handle from one quorum to another. The choreography then involves the parties in the source quorum, which transform their current shares into pieces addressed to the target quorum. The parties in the target quorum collect the pieces they receive and combine them to obtain the new shares associated with the same handle. The CRDV layer continues to store the same handle, while the secure runtime changes which parties hold the corresponding shares.

When resharing or reconfiguration changes the quorum associated with a private value, the handle stored in the CRDV layer remains unchanged. Only the secure runtime

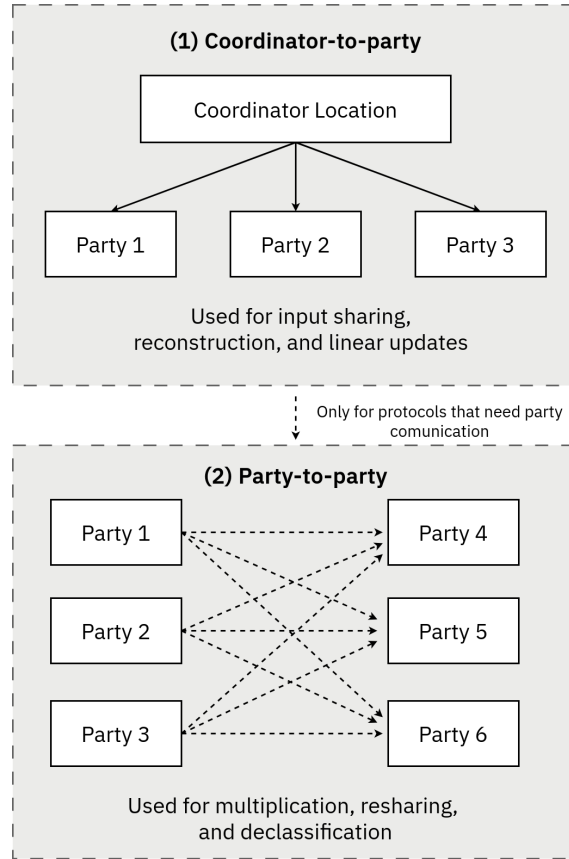


Figure 6.4: Staged communication structure used by the choreographic MPC runtime.

changes which parties hold the corresponding shares. This keeps the replicated identifier stable while allowing the secure-layer placement of private state to evolve.

Since Chorus represents locations and communication structures statically, the set of parties and quorums must be known when the runtime is built. The prototype addresses this by generating part of the Chorus code from the runtime configuration. This makes it possible to evaluate different predefined deployments by changing the configuration and rebuilding the runtime. However, it also means that arbitrary new parties or quorums cannot be added dynamically at runtime without regenerating the corresponding Chorus structures. This limitation is discussed further in Section 6.5.

6.5. Implementation Challenges and Limitations

The main implementation challenge was reconciling the reconfiguration mechanism described in Chapter 5 with the static structure of the concrete Chorus-based deployment used in the prototype. This restriction is not a fundamental limitation of the secure CRDV

architecture. At the design level, the secure layer can redistribute shares between different quorums of MPC parties. However, in the Chorus-based runtime used in this implementation, locations and communication structures are represented statically. Consequently, coordinators, parties, and quorums must be known when the runtime is built.

To address this deployment constraint, the prototype uses configuration-driven code generation. The runtime configuration specifies the available coordinator locations, parties, quorums, and default coordinator-to-quorum assignments. Build-time code generation then produces the Chorus location types, quorum representations, and dispatch logic required by the runtime. This avoids writing repetitive type-level code manually and makes it possible to evaluate different predefined deployments by changing the configuration and rebuilding the runtime.

This solution supports reconfiguration between predefined quorums. At runtime, a coordinator can move from one configured quorum to another, and subsequent secure operations use the newly selected quorum. However, the prototype does not fully support dynamic membership. Adding new parties or defining arbitrary new quorums requires updating the configuration and regenerating the runtime code. This restriction is therefore a limitation of the prototype's concrete Chorus-based deployment, rather than of the secure CRDV design.

Reconfiguration also required care during execution. Since all participants in a choreography must follow the same communication structure, changing the selected quorum while an operation is running could desynchronise the runtime. To avoid this, the implementation fixes the selected quorum for the duration of each coordinator request. Any quorum change only affects later operations, after the current request has completed.

Other limitations are engineering choices made to keep the prototype focused and evaluable. The MPC parties run as Rust threads inside the same runtime process, and party-local shares are kept in memory. Communication between the coordinator-facing components and the secure runtime uses Unix Domain Sockets. These choices simplify deployment and experimentation, but they are not fundamental to the design.

A more mature deployment would separate the components that are colocated in the prototype. Each CRDV replica would run as a PostgreSQL instance with the secure extension installed, while MPC parties would run as independent services, possibly on different machines or under different administrative domains. Application clients would

contact the client-facing coordinator through an authenticated interface when submitting plaintext private inputs or requesting reconstruction. The coordinator-facing components and the MPC parties would communicate through authenticated network channels and store their shares in durable storage, allowing protected state to survive process restarts or machine failures.

This would not require changing the database-facing CRDV interface. Applications would still issue SQL operations over public values and handles, and the CRDV layer would still store handles for private values. The difference would be in the deployment of the secure layer: instead of in-memory parties running as threads inside one runtime process, the system would use isolated MPC-party services with persistent state and authenticated communication. This structure would also fit the C-ITS-inspired deployment used later in the evaluation, where different infrastructure domains operate distinct replicas, coordinators, and MPC parties.

Chapter 7

Experimental Evaluation

This chapter evaluates the secure CRDV prototype implemented in this work. The goal is to understand the performance cost introduced by adding confidentiality-preserving execution to CRDV, and to identify which types of operations dominate the overhead of the system.

To answer these questions, the evaluation is divided into two parts. First, microbenchmarks isolate the latency of individual operations supported by the prototype, including registers, counters, sets, maps, lists, and quorum reconfiguration. These experiments make it possible to identify the cost of each secure operation independently and to observe how that cost changes with the number of previous operations or protected handles.

Second, a macrobenchmark evaluates the prototype under a mixed workload inspired by a semi-realistic C-ITS deployment. The goal of this benchmark is not only to measure aggregate performance, but also to illustrate how the proposed solution would behave in an application-oriented setting where replicated infrastructure domains manage both public metadata and confidential values. The workload combines reads, updates, insertions, removals, and reconfiguration operations over simplified DENM and IVIM objects. This makes it possible to observe the behaviour of the integrated system, including the PostgreSQL-based CRDV layer, the privacy-aware operation wrappers, the coordinator interface, and the secure MPC runtime.

All experiments compare plaintext and secure execution whenever applicable. Plaintext execution follows the original CRDV path and stores values directly in the replicated database. Secure execution marks selected fields as private and redirects the corresponding operations to the secure runtime, where values are represented by secret-shared handles. The main reported metrics are operation latency and throughput. Latency is reported in milliseconds, while throughput is reported as completed application-level operations per second in the macrobenchmark.

7.1. Environment

The experiments were executed on the machine described in Table 7.1. The CRDV replicas correspond to separate PostgreSQL databases running locally. The coordinator logic is invoked from PostgreSQL through the prototype implementation, while the secure runtime is executed as a separate process. Within this runtime, the MPC parties are represented as independent threads. Therefore, unless explicitly stated otherwise, the reported results measure the local overhead of the prototype rather than the performance of a geographically distributed deployment.

Table 7.1: Hardware used in the experimental evaluation.

Component	Configuration
Machine	ASUS ROG Zephyrus G14 GA401QH
CPU	AMD Ryzen 7 5800HS
Memory	16 GB RAM
GPU	NVIDIA GeForce GTX 1650 Mobile / Max-Q, not used by the benchmark
Operating system	Ubuntu 24.04.4 LTS, Linux 6.17.0-1032-oem

The evaluation uses the prototype described in Chapter 6. Unless stated otherwise, the experiments do not include artificial network latency. In the latency-injected macrobenchmark scenarios, Toxiproxy 2.12.0 [42] is used only as an experimental tool to emulate deterministic communication delays between CRDV replicas. It is not part of the secure CRDV prototype itself because the prototype’s replication mechanism is unchanged; the proxy is an external, reproducible fault-injection mechanism that avoids modifying that mechanism solely for evaluation. Additional delay injection between MPC parties is performed by the secure runtime, as described in Section 7.3.4.

7.2. Microbenchmarks

The microbenchmarks evaluate the cost of individual operations in the implemented secure CRDV prototype. The goal is to isolate the overhead introduced by secure execution before analysing the behaviour of the system under a mixed workload. These experiments measure the latency of selected CRDV operations when executed in plaintext and when executed over private values using the secure runtime.

The evaluated operations were chosen to cover the main replicated data types supported by the prototype: registers, counters, sets, maps, and lists. The benchmarked operations are `registerSet`, `counterInc`, `setAdd`, `mapAdd`, and `listAppend`. These operations

exercise different secure execution patterns. Registers, maps, and lists mainly require creating or replacing secret-shared values associated with handles. Counters update an existing shared value by applying a secret-shared delta. Sets are more expensive because secure insertion requires comparing the candidate private element with existing private elements without revealing their plaintext values.

In addition to regular CRDV operations, the microbenchmarks also evaluate quorum reconfiguration as a system-level operation. This operation redistributes the protected state associated with existing handles between quorums, making it possible to measure both the cost of share propagation and the cost of adapting the parties responsible for protected values.

7.2.1 Methodology and Metrics

The main metric used in the microbenchmarks is response time, measured in milliseconds. Response time is defined as the elapsed time between issuing a benchmark request and receiving its result. For regular secure operations, this includes the SQL or PL/pgSQL execution, the invocation of the secure operation through the PostgreSQL extension, communication with the secure runtime, execution of the corresponding secure protocol, and the return of the response to PostgreSQL. For reconfiguration, it corresponds to the elapsed time of the quorum-change request, including the secure-runtime work required to redistribute the affected shares.

The concrete operations, input values, and object-state sizes used in the operation microbenchmarks are described in Section 7.2.2. The quorum transitions and handle counts used in the reconfiguration benchmark are described in Section 7.2.3.

Each measured configuration was executed five times. This limited repetition count was chosen to keep the full experiment practical while covering all state sizes, modes, and quorum transitions; it is sufficient for the reported median-based exploratory comparison, but not for estimating rare tail events with high precision. Before the measured executions, warm-up runs were performed outside the timer to reduce the effect of one-time setup costs such as caches, initial runtime paths, and database extension loading. The warm-up time is not included in the reported results.

The reported values correspond to the median response time across the measured runs. Median latency is used because it is less sensitive to occasional spikes than the mean.

The benchmark scripts also collect additional statistics, including mean, standard deviation, minimum, maximum, and 95th percentile latency, which are used when discussing variability or tail latency.

7.2.2 Operation Benchmarks

The operation benchmarks compare plaintext and secure execution for individual CRDV operations: `registerSet`, `counterInc`, `setAdd`, `mapAdd`, and `listAppend`. These operations cover the main data types supported by the prototype: registers, counters, sets, maps, and lists.

The workload is deterministic rather than randomly generated. For each configuration, the benchmark creates one object and populates it with a controlled sequence of previous operations. Registers repeatedly write a fixed value, counters increment by one, sets insert generated elements, maps insert generated key-value pairs, and lists append generated values. The measured operation is then executed once over the prepared object state.

In plaintext mode, the evaluated field is marked as public and the operation follows the original CRDV execution path. In secure mode, the evaluated field is marked as private and the corresponding value is represented through the secure runtime. For sets, the private field is the key, since set elements are represented as keys in the underlying CRDV representation; for the remaining data types, the private field is the data field. Each operation is evaluated after 100, 1000, and 5000 previous operations, with five measured runs per configuration, as summarised in Table 7.2.

Table 7.2: Configuration used for the operation microbenchmarks.

Parameter	Value
Measured operations	<code>registerSet</code> , <code>counterInc</code> , <code>setAdd</code> , <code>mapAdd</code> , <code>listAppend</code>
Execution modes	Plain and secure
Previous operations	100, 1000, 5000
Runs per configuration	5
Warm-up state size	5000 operations
Metric	Median response time
Unit	Milliseconds

The results of these benchmarks allow the cost of confidentiality to be characterised at the level of individual operations. In particular, they show whether secure execution introduces a mostly constant overhead or whether the overhead increases with the number of previous operations.

7.2.3 Reconfiguration

The reconfiguration benchmark evaluates the cost of changing the quorum responsible for maintaining protected state. Unlike the operation benchmarks, which measure regular CRDV updates, this benchmark measures a system-level operation introduced by the secure layer: moving the shares of existing private values from one quorum to another.

For each configuration, the benchmark creates 100, 1000, or 5000 private handles by inserting private register values while the source quorum is active. It then measures the time required to reconfigure the protected state from a source quorum to a target quorum. Source and target quorum sizes are 3, 5, and 7 parties, producing all pairwise transitions between these sizes. The Shamir threshold is fixed at 2.

Before each transition-size configuration, the MPC party state is reset to avoid cross-test contamination. The benchmark creates a baseline state once, snapshots the CRDV tables and timestamp sequences, and restores that baseline before each warm-up and measured run. Transition-size configurations are executed in randomised order to reduce bias from machine warm-up or performance drift, while the contents of each configuration remain controlled. Each configuration is executed five times, after two warm-up runs, as summarised in Table 7.3.

Table 7.3: Configuration used for the reconfiguration microbenchmarks.

Parameter	Value
Source quorum sizes	3, 5, 7 parties
Target quorum sizes	3, 5, 7 parties
Transitions	3–3, 3–5, 3–7, 5–3, 5–5, 5–7, 7–3, 7–5, 7–7
Number of handles	100, 1000, 5000
Shamir threshold	2
Runs per configuration	5
Warm-up runs	2
Metric	Median reconfiguration time
Unit	Milliseconds

The purpose of this benchmark is twofold. First, it shows that reconfiguration is supported by the prototype. Second, it quantifies the cost of changing quorum-based trust policies as the number of protected handles grows.

7.2.4 Results

Table 7.4 reports the median latency, in milliseconds, of a single operation after a controlled number of previous operations. The table shows two distinct behaviours: stable

low-millisecond overhead for registers, counters, maps, and lists, and rapidly growing latency for secure set insertion.

Table 7.4: Latency of individual CRDV operations.

CRDT	100 previous ops		1000 previous ops		5000 previous ops	
	Plain	Secure	Plain	Secure	Plain	Secure
Register	1.26	2.28	1.27	2.46	1.24	2.50
Counter	1.35	2.42	1.33	2.37	1.37	2.39
Set	1.60	908.77	2.07	8878.23	1.30	44291.61
Map	1.23	2.61	1.23	2.45	1.26	2.63
List	1.46	2.37	1.45	2.91	1.53	2.86

The most significant result is the cost of secure set insertion. Its latency increases from 908.77 ms with 100 previous operations to 8878.23 ms with 1000 previous operations and 44291.61 ms with 5000 previous operations. This corresponds to approximately 0.9 s, 8.9 s, and 44.3 s, respectively. This increase is caused by the secure membership check required before inserting a private set element. Since the plaintext element cannot be used directly as the replicated key, the system must compare the candidate value against existing private elements represented by handles. As the number of previous set operations grows, the number of secure comparisons also grows.

For registers, counters, maps, and lists, secure execution introduces a much smaller and mostly stable overhead. Plain operations remain around 1.2–1.5 ms, while the corresponding secure operations are usually between 2.3 and 2.9 ms. This indicates that, for these data types, the secure overhead is mostly independent of the object history size.

Figures 7.1 and 7.2 visualise the contrast observed in Table 7.4. The operations with stable secure latency are grouped separately from set insertion because the latter has a different order of magnitude.

Table 7.5 reports the median reconfiguration latency, in milliseconds, for different quorum transitions and numbers of protected handles.

The results show that reconfiguration cost increases with the number of handles and, to a lesser extent, with the number of parties involved in the source and target quorums. For 100 handles, all transitions complete in approximately 18–19 ms. For 1000 handles, the cost ranges from 35.27 ms to 46.37 ms. For 5000 handles, the cost ranges from 118.47 ms to 213.53 ms. The dominant factor is the number of handles that must be reshared. Larger target quorums tend to increase the cost because more destination parties must

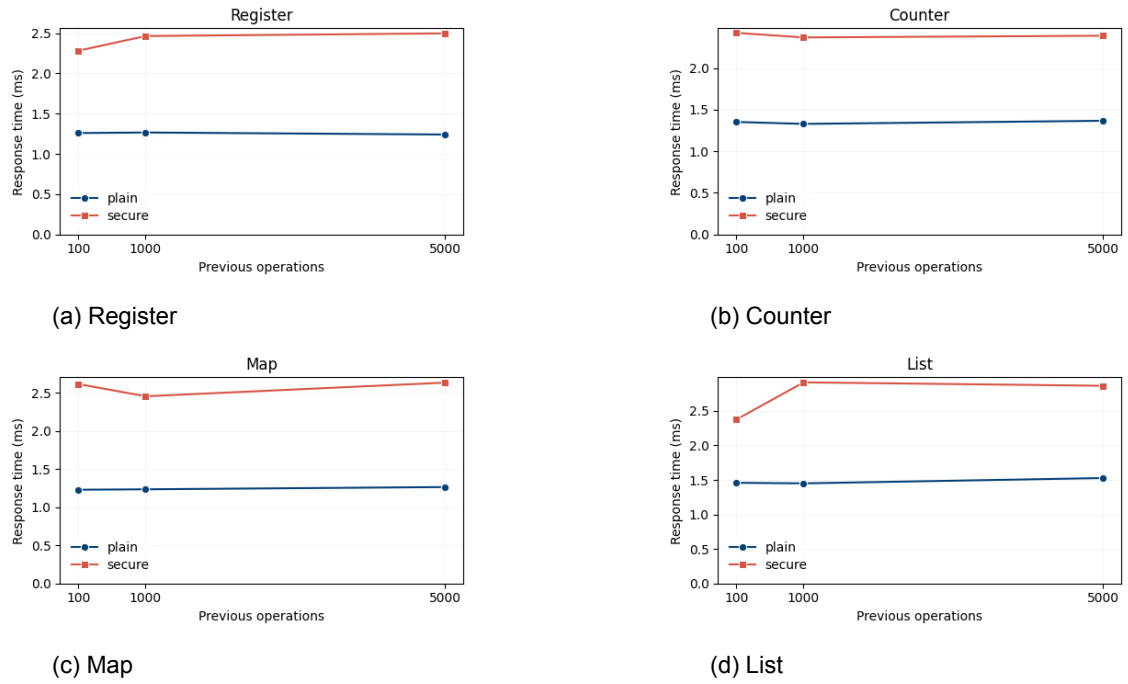


Figure 7.1: Latency of register, counter, map, and list operations.

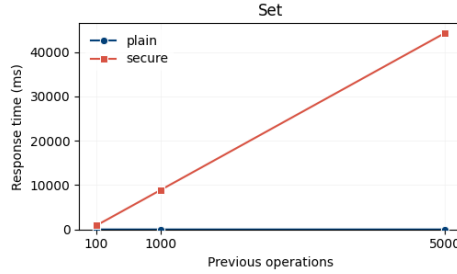


Figure 7.2: Latency of set insertion.

Table 7.5: Reconfiguration latency for different quorum transitions.

Transition	100 handles	1000 handles	5000 handles
3-3	18.03	35.27	118.47
3-5	18.30	37.80	140.02
3-7	18.76	41.15	158.03
5-3	18.20	37.10	131.48
5-5	18.55	40.12	151.43
5-7	19.04	43.64	179.53
7-3	18.32	38.01	138.58
7-5	18.77	42.53	169.91
7-7	19.24	46.37	213.53

receive shares, but the difference between quorum sizes is smaller than the difference caused by increasing the number of handles. Figure 7.3 illustrates the same trend across the evaluated quorum transitions.

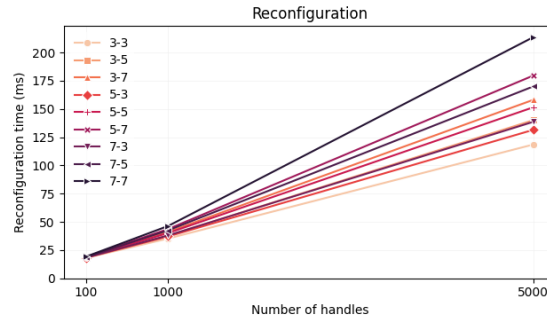


Figure 7.3: Reconfiguration latency for different quorum transitions.

7.2.5 Discussion

The microbenchmark results show that the overhead introduced by secure execution depends strongly on the type of operation. This behaviour matches the general intuition behind secret sharing-based MPC: operations that can be implemented with little or no interaction between parties remain relatively cheap, while operations that require repeated secure comparisons or communication rounds become significantly more expensive.

For registers, counters, maps, and lists, the secure version is approximately two times slower than the plaintext version, but the absolute latency remains low, typically below 3 ms. These operations either create a private value, replace a private value, or update an existing shared value directly. They do not require searching over a growing private collection, which explains why their overhead remains small and mostly stable.

Secure set insertion illustrates the opposite case. Although it is exposed as a single logical CRDV operation, its secure implementation requires more expressive computation over private state. In particular, the system must decide whether a private candidate element is already represented without using the plaintext element as a database key. This turns a simple replicated set operation into a sequence of secure private comparisons, making the operation significantly more expensive as the set grows.

This highlights an important design lesson for secure replicated data management. The cost of secure execution is not determined only by the number of application-level operations, but by the amount of private computation required inside each operation. The more

expressive the system needs to be over private replicated values, especially when merge, search, or comparison logic depends on those values, the higher the expected cost.

The reconfiguration benchmark shows that changing quorum-based trust policies is feasible in the prototype, but its cost depends on the amount of protected state being moved. Reconfiguration is more expensive than ordinary secure updates because it must reshare all affected handles to the target quorum and remove obsolete shares from parties that leave the previous quorum. However, this cost is incurred only when the trust policy changes, rather than on every regular CRDV operation.

The results should also be interpreted in light of the controlled experimental environment used in this thesis. The microbenchmarks execute the prototype in a local deployment, which makes it possible to isolate operation-level costs but does not capture the full effects of a geographically distributed setting. In a deployment where CRDV replicas and MPC parties run on separate machines or in different regions, communication latency, network variability, and operational isolation may have a stronger impact, especially on interactive MPC operations. Evaluating the system in such a geographically distributed deployment remains an important direction for future work.

Overall, the microbenchmarks provide two main conclusions. First, for most data types, secure execution adds a moderate and stable per-operation overhead. Second, operations whose semantics require private search or comparison, such as secure sets, are significantly more expensive because their cost grows with the number of private elements that must be compared. These results provide the baseline for interpreting the macrobenchmark presented in the next section, where active message enumeration builds on secure set operations and further amplifies this cost.

7.3. Macrobenchmarks

The macrobenchmark evaluates the behaviour of the implemented secure CRDV prototype under a mixed application-level workload. While the microbenchmarks isolate the cost of individual operations, this experiment measures how those costs combine when reads, updates, insertions, removals, and quorum reconfigurations are executed together. The goal is to evaluate the prototype as an integrated system, including the PostgreSQL-based CRDV layer, the privacy-aware operation wrappers, and the secure MPC runtime.

The workload is based on a simplified C-ITS scenario. Cooperative Intelligent Transport

Systems (C-ITS) connect vehicles, roadside infrastructure, and traffic-management services so that they can exchange safety and traffic information. The scenario is inspired by the role of cooperative intelligent transport systems in connecting vehicles, road infrastructure, and traffic management services [43]. This scenario was chosen because it combines replicated data management with confidentiality requirements. Multiple infrastructure domains may need to share information about road events and traffic conditions, while some message fields and identifiers may contain sensitive information. The benchmark uses simplified DENM and IVIM objects to exercise the main secure CRDV abstractions implemented in the prototype.

The macrobenchmark evaluation is divided into four parts. First, the case-study scenario defines the deployment model used in the experiment, including replicas, coordinators, MPC parties, and regional quorums. Second, the data model describes the simplified DENM and IVIM objects used by the workload. Third, the mapping to secure CRDV explains how those objects are represented using sets, registers, and maps. Finally, the workload and results show how the complete system behaves under plaintext and secure execution, both with and without injected latency.

7.3.1 Case Study Scenario

The case study models a smart-city and cooperative mobility scenario in which several road infrastructure domains maintain a shared world model of vehicles, road events, and infrastructure messages. In such a deployment, different city operators may need to collaborate to provide services such as traffic monitoring, hazard notification, platooning support, and infrastructure-to-vehicle information. However, these operators should not necessarily have to trust a single server, or each other, with all sensitive application data in plaintext.

In this setting, replicated data management allows each infrastructure domain to keep a local view of the shared state and continue operating independently, while CRDT-based convergence ensures that updates can later be reconciled. The secure layer provides confidentiality for selected values by storing only handles in the replicated CRDV state and distributing the corresponding shares across MPC parties. This models a deployment where private information can be managed collaboratively without requiring any individual infrastructure server to learn all protected values.

The concrete macrobenchmark uses a three-site C-ITS-inspired deployment. Each site represents a road infrastructure domain associated with a different city: Lisbon, Madrid, and Paris. Each city hosts one CRDV replica and one local coordinator. The replicas exchange CRDV updates with each other, while each coordinator is responsible for invoking secure operations over the private values managed by its local replica.

Each city is also associated with a regional group of five MPC parties.¹ The Lisbon region is associated with parties $P_1, \dots, P_5$, representing Lisbon, Oeiras, Amadora, Loures, and Setúbal. The Madrid region is associated with parties $P_6, \dots, P_{10}$, representing Madrid, Getafe, Leganés, Alcobendas, and Toledo. The Paris region is associated with parties $P_{11}, \dots, P_{15}$, representing Paris, Saint-Denis, Boulogne, Versailles, and Créteil. These groups model the availability of multiple secure computation parties near each road infrastructure domain.

For each coordinator, the benchmark defines a regional quorum policy specifying the quorums that are authorised to store and process that coordinator's private values. At any point, the coordinator uses one active quorum selected from this authorised set. The active quorum stores the shares of that city's private values and participates in secure computations requested by the coordinator.

As described later in Section 7.3.4, the macrobenchmark also includes reconfiguration operations. These operations change the active quorum to another quorum authorised by the same regional policy, by resharing the protected state to the new quorum. This models a local trust-policy change, such as replacing or reconfiguring the secure-computation provider used by one infrastructure domain, rather than a global restructuring of all servers in the deployment. Therefore, reconfiguration changes which parties hold and process shares, but it does not move the CRDV replica to another city or change the set of replicas.

Figure 7.4 illustrates a simplified version of this deployment model. The map is used to make the case-study scenario concrete: each city represents one infrastructure domain with a CRDV replica, a local coordinator, and a regional group of MPC parties. For readability, the figure shows only a subset of the parties in each region and places them schematically rather than at their exact geographic locations. The figure highlights the selected quorum and reconfiguration links, while the complete party groups and quorum configurations are described in the text and in Table 7.12.

¹The city and party names used in the macrobenchmark are arbitrary and are used only to make the deployment scenario easier to describe.

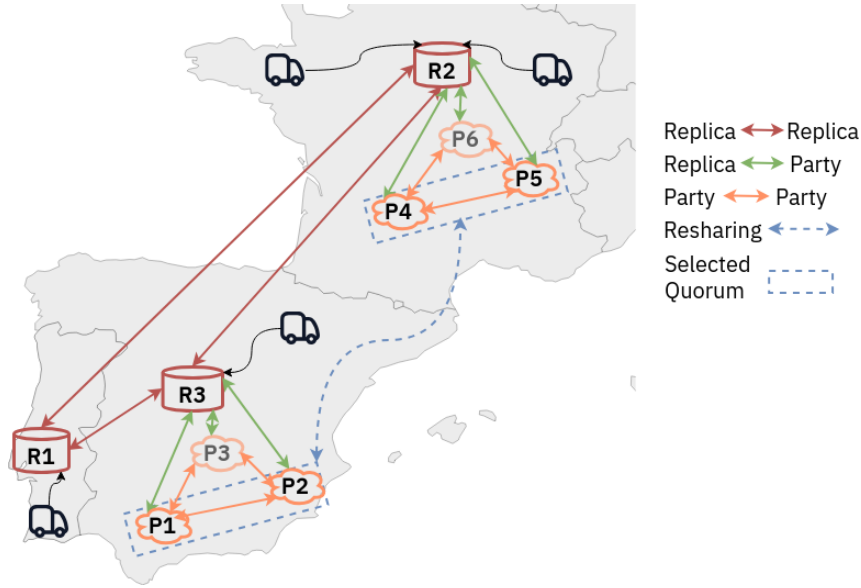
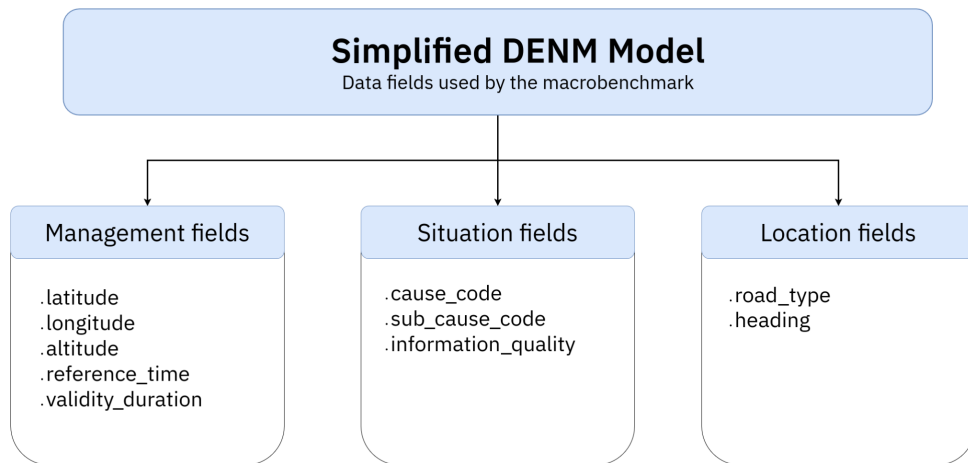


Figure 7.4: Case-study deployment topology used in the macrobenchmark.

7.3.2 DENM and IVIM Data Model

The macrobenchmark uses simplified data models inspired by two C-ITS message types defined by European Telecommunications Standards Institute (ETSI): DENMs and IVIMs [44, 45]. The selected fields and grouping are also informed by existing descriptions of DENM and IVIM message structures [46, 47]. These message types were selected because they represent two complementary categories of cooperative transport information: event-driven hazard notifications and infrastructure-provided traffic information. The goal is not to implement complete ETSI-compliant message encodings, but to derive representative application-level data structures with fields that combine public metadata and private values. For this reason, Figures 7.5 and 7.6 show the simplified data models used in the benchmark, rather than the complete ETSI message structures.

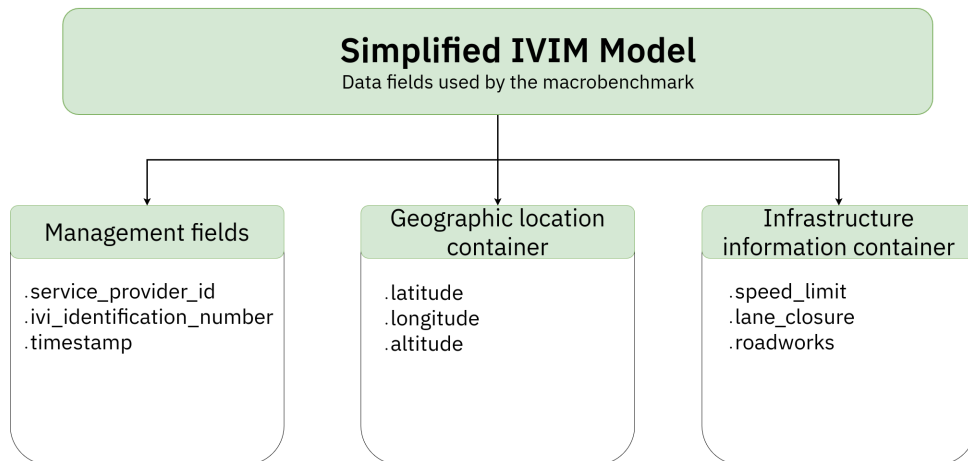
A DENM is an event-driven message used to describe a road event or abnormal traffic condition, such as an accident, roadworks, or wrong-way driving. The simplified DENM model used in this benchmark is shown in Figure 7.5. It groups the selected fields into management, situation, and location fields. The management fields describe the event position and temporal validity, including latitude, longitude, altitude, reference time, and validity duration. The situation fields describe the event itself, including the cause code, sub-cause code, and information quality. The location fields describe road context, such as the road type and the event position heading.



Inspired by ETSI DENM, but not a complete ETSI message encoding

Figure 7.5: Simplified DENM data model used by the macrobenchmark.

An IVIM is an infrastructure-to-vehicle message used to provide information from infrastructure services to vehicles, such as speed limits, lane restrictions, roadworks warnings, or other mandatory and advisory road signage. The simplified IVIM model used in this benchmark is shown in Figure 7.6. It includes management fields, such as the service provider identifier, IVI identification number, and timestamp; one geographic location container, including latitude, longitude, and altitude; and one infrastructure information container, representing information such as a speed limit, lane closure, or roadworks indication.



Inspired by ETSI IVIM, but not a complete ETSI message encoding

Figure 7.6: Simplified IVIM data model used by the macrobenchmark.

7.3.3 Mapping to Secure CRDV

The simplified DENM and IVIM objects are represented using the secure CRDV abstractions implemented in the prototype. Table 7.6 summarises this mapping. The remainder of this section refines each entry and explains how the case-study objects are decomposed into secure sets, registers, and maps.

Table 7.6: Mapping between the case-study model and secure CRDV abstractions.

Case-study concept	Secure CRDV representation
Active DENM identifiers	Secure 2P-Set D_s with private keys
Active IVIM identifiers	Secure 2P-Set I_s with private keys
DENM fields	Private registers
IVIM fields	Private registers
DENM field index	Map from field name to register identifier
IVIM field index	Map from field name to register identifier
DENM/IVIM deactivation	Insertion into the remove component of D_s or I_s

Active message identifiers are represented using a secure 2P-Set abstraction. The set D_s stores the identifiers of active DENM objects, while the set I_s stores the identifiers of active IVIM objects. Since the presence of a message identifier in the active set may reveal information about existing road events or infrastructure messages, these identifiers are stored as private set keys. Activating a DENM or IVIM corresponds to adding its identifier to the add component, while deactivating it corresponds to adding the identifier to the remove component.

The secure 2P-Set is implemented at the CRDV view level, rather than as a new primitive in the secure runtime. Internally, the add and remove components are obtained from the underlying secure set operations, and membership between private identifiers is checked using secure set containment. The resulting active set contains the private identifiers that appear in the add component and not in the remove component. This design reuses the existing secure set functionality and illustrates that higher-level secure CRDV abstractions can be built compositionally on top of the lower-level operations provided by the prototype.

The contents of each message are decomposed into individual private registers. Each register stores one field, and the register identifier encodes the message identifier, the logical container, and the field name. For example, the simplified model may use identifiers following the pattern shown in Listing 7.1.

```
DENM D1:
management.eventPosition.latitude  -> D1:management:eventPosition:latitude
management.validityDuration        -> D1:management:validityDuration
```

<code>situation.causeCode</code>	-> <code>D1:situation:causeCode</code>
<code>location.roadType</code>	-> <code>D1:location:roadType</code>
 IVIM I1:	
<code>management.serviceProviderId</code>	-> <code>I1:management:serviceProviderId</code>
<code>management.iviIdentificationNumber</code>	-> <code>I1:management:iviIdentificationNumber</code>
<code>geographicLocation.latitude</code>	-> <code>I1:geographicLocation:latitude</code>
<code>information.speedLimit</code>	-> <code>I1:information:speedLimit</code>

Listing 7.1: Example register identifiers used for message fields.

The registers that store these fields are marked as private. Consequently, the replicated CRDV state stores handles for the corresponding values rather than plaintext values. Updating a message field therefore corresponds to updating the private register associated with that field.

Maps are used as per-message field indexes. For each DENM or IVIM identifier, a map links application-level field names to the register identifiers that contain the corresponding values. This indirection keeps the message structure explicit at the CRDV level while allowing each field to be managed as an independent private register.

7.3.4 Workload and Configuration

The macrobenchmark workload is composed of application-level operations over the DENM and IVIM objects described in the previous sections. Each operation represents an action that an application may perform over the replicated C-ITS state, such as listing active messages, reading message contents, creating new messages, updating existing messages, removing messages from the active set, or changing the quorum responsible for secure execution.

Each application-level operation expands into one or more CRDV operations, as summarised in Table 7.7. Listing active messages reads the corresponding active set. Reading a message accesses the per-message map and then reads the registers referenced by that map. Creating a message inserts a new identifier into the active set, creates multiple private registers for the message fields, and adds the corresponding map entries. Updating a message modifies one or more existing private registers. Removing a message inserts the message identifier into the remove component of the corresponding active set.

Table 7.7: Application-level operations used in the macrobenchmark.

Operation	Message type	Internal CRDV operations
getActiveDENMs	DENM	Read active DENM 2P-Set D_s
getDENM	DENM	Read DENM field map and associated registers
createDENM	DENM	setAdd on D_s , multiple registerSet, multiple mapAdd
updateDENM	DENM	One or more registerSet operations
removeDENM	DENM	Add DENM identifier to the remove component of D_s
getActiveIVIMs	IVIM	Read active IVIM 2P-Set I_s
getIVIM	IVIM	Read IVIM field map and associated registers
createIVIM	IVIM	setAdd on I_s , multiple registerSet, multiple mapAdd
updateIVIM	IVIM	One or more registerSet operations
removeIVIM	IVIM	Add IVIM identifier to the remove component of I_s
reconfigure	System	Change the active quorum of the selected coordinator

The workload is defined using weighted random selection. At each step, one operation is selected according to the weights shown in Table 7.8. The same application-level operation weights are used in plaintext and secure modes. Secure execution additionally includes a `reconfigure` operation with weight 2, because quorum reconfiguration is specific to the secure runtime and has no equivalent in plaintext CRDV execution.

Table 7.8: Macrobenchmark workload distribution.

Operation	Plain weight	Secure weight
getActiveDENMs	1	1
getDENM	23	23
createDENM	1	1
updateDENM	19	19
removeDENM	1	1
getActiveIVIMs	1	1
getIVIM	23	23
createIVIM	1	1
updateIVIM	25	25
removeIVIM	1	1
reconfigure	—	2

The weights are intended to model a deployment where most operations inspect or refresh existing information rather than create new messages. Accordingly, the most frequent operations are `getDENM`, `getIVIM`, `updateDENM`, and `updateIVIM`. Creation and removal

operations have lower weights, reflecting that new DENM and IVIM objects are created or deactivated less frequently than they are read or updated. This distribution also prevents secure set insertion from dominating the entire macrobenchmark, while still including set operations in the workload.

These weights are heuristic and should not be interpreted as a validated traffic model for a real C-ITS deployment. A more mature evaluation should refine these probabilities using measurements or traces from realistic deployments, so that the workload distribution more closely matches operational conditions.

The benchmark was executed in four scenarios. First, plaintext and secure execution were evaluated without injected latency. Then, the same two modes were executed with injected latency. In the plaintext latency scenario, latency is injected only between CRDV replicas. In the secure latency scenario, latency is injected both between CRDV replicas and between MPC parties.

For the latency-injected scenarios, two forms of deterministic latency are used. First, *Toxiproxy* is used to emulate communication delays between the CRDV replicas, following the site topology defined in the case-study scenario. This affects both plaintext and secure executions, since both modes rely on replication between the PostgreSQL databases. Second, in the secure latency scenario, the secure runtime applies deterministic per-link delays between MPC parties before delivering each message.

The latency model reflects the topology of the case-study scenario. The CRDV replicas are treated as geographically distributed sites, corresponding to the Lisbon, Madrid, and Paris infrastructure domains. In contrast, the MPC parties associated with each coordinator are treated as relatively nearby parties within the same regional group. This distinction motivates using larger round-trip delays between CRDV replicas than between MPC parties. Table 7.9 summarises the round-trip latencies injected between CRDV replicas.

Table 7.9: Injected replication latency.

Replica pair	RTT latency
1–2	12 ms
1–3	24 ms
2–3	18 ms

The MPC party delay model follows the regional party placement defined in the case-study scenario. Since the parties in each group are intended to model nearby secure-computation providers within the same region, their pairwise round-trip times are smaller

than the delays between CRDV replicas. Within each region, pairwise round-trip times are assigned according to a fixed matrix. These values are deterministic and use zero jitter. The transport applies half of the configured round-trip time to each one-way send. Table 7.10 summarises the injected MPC party latency model.

Table 7.10: Injected MPC party latency model.

Party group	RTT range
P_1-P_5	2–7 ms
P_6-P_{10}	2–7 ms
$P_{11}-P_{15}$	2–6 ms
Fallback inter-party latency	9 ms

Each macrobenchmark run starts from a freshly created PostgreSQL cluster. Before each run, the previous cluster state is dropped, a new cluster is created, and the benchmark waits for PostgreSQL subscriptions to settle. For secure runs, the secure runtime is started before the benchmark begins. For latency-injected runs, `Toxiproxy` is started before creating the cluster, so that the replication links are created through the proxy.

The benchmark is executed for a fixed duration rather than for a fixed number of operations. In the configuration used in this evaluation, each run lasts two hours. During this period, operations are selected according to the workload distribution and executed against one of the configured replicas. The benchmark records per-operation latency, total throughput, and time-series samples of throughput and latency during the measured phase.

The workload is executed over the three-site deployment described in Section 7.3.1. Each site hosts one CRDV replica and is associated with one coordinator. In secure mode, each coordinator selects its active quorum from a set of predefined regional quorums. The `reconfigure` operation changes the active quorum of the coordinator receiving the operation to another predefined quorum from the same regional group. The Shamir threshold is fixed at 2. This permits reconstruction by three parties and preserves confidentiality against up to two colluding parties. Holding the threshold fixed isolates the effect of quorum size and handle count; increasing it would generally increase the number of parties required for reconstruction and the communication and computation cost of interactive protocols, so these results should not be generalised to other thresholds without further measurement. The resulting macrobenchmark configuration is summarised in Table 7.11.

The quorum policy referenced in Table 7.11 is shown in Table 7.12. Each coordinator has

Table 7.11: Macrobenchmark configuration.

Parameter	Value
Execution modes	Plain and secure
CRDV replicas	3
Coordinators	3
MPC parties	15
Quorums per coordinator	3
MPC quorum configuration	Table 7.12
Shamir threshold	2
Benchmark duration	2 hours per run
Latency injection	Tables 7.9 and 7.10
Initial DENM objects	3
Initial IVIM objects	3
Initial active DENM objects	3
Initial active IVIM objects	3
Reported metrics	Throughput and latency

three predefined authorised regional quorums and starts from the default quorum listed in the last column. Reconfiguration is restricted to the authorised quorums defined for that coordinator.

Table 7.12: Authorised regional MPC quorum policy used in the macrobenchmark.

Coordinator	Authorised quorums	Default quorum
1	$\{P_1, P_2, P_3\}, \{P_1, P_2, P_4, P_5\}, \{P_3, P_4, P_5\}$	$\{P_1, P_2, P_3\}$
2	$\{P_6, P_7, P_8\}, \{P_6, P_7, P_9, P_{10}\}, \{P_8, P_9, P_{10}\}$	$\{P_6, P_7, P_8\}$
3	$\{P_{11}, P_{12}, P_{13}\}, \{P_{11}, P_{12}, P_{14}, P_{15}\}, \{P_{13}, P_{14}, P_{15}\}$	$\{P_{11}, P_{12}, P_{13}\}$

Throughput is measured as the number of completed application-level operations per second. Latency is measured as the response time observed by the benchmark client for each selected operation. Since application-level operations may expand into multiple internal CRDV operations, the measured latency captures the cost of the complete operation as seen by the application.

7.3.5 Results

This section presents the macrobenchmark results for the four execution scenarios described in Section 7.3.4. Since each run is executed for a fixed duration, scenarios with higher throughput also execute more operations and therefore evolve the replicated state further during the same time window. As a result, the latency distributions are not measured over identical object states. This is especially relevant for operations over private active sets, whose cost depends on the number of elements accumulated during the run.

For this reason, throughput is used as the primary metric for comparing complete scenarios. Per-operation latency is used mainly as a diagnostic metric to identify which operation classes dominate the workload cost.

Table 7.13 reports the total number of executed operations and the achieved throughput in each scenario. The results show a substantial throughput reduction when secure execution is enabled. In the baseline scenario, plaintext execution reaches 443.01 operations per second, while secure execution reaches 0.84 operations per second. With injected latency, plaintext execution reaches 421.91 operations per second, while secure execution reaches 0.19 operations per second.

Table 7.13: Macrobenchmark throughput.

Scenario	Mode	Executed operations	Throughput (ops/s)
Baseline	Plain	3,189,679	443.01
Baseline	Secure	6,087	0.84
Injected latency	Plain	3,037,784	421.91
Injected latency	Secure	1,363	0.19

This reduction reflects the additional cost of invoking the secure runtime and executing MPC protocols. However, the duration-based design of the benchmark must be considered when interpreting the remaining results. Higher-throughput scenarios execute more operations and therefore accumulate more updates during the same two-hour window, while lower-throughput scenarios evolve the replicated state more slowly. Consequently, per-operation latency is not used as the main cross-scenario metric; instead, it is used to explain which operation classes are responsible for the observed throughput reduction.

Table 7.14 reports the number of completed operations of each type in each scenario. These counts provide context for interpreting the per-operation latency results, since operations with fewer samples produce less stable latency estimates.

The operation counts follow the configured workload distribution, but the absolute number of samples differs substantially between scenarios because the benchmark is duration-based. This is particularly important for the secure latency-injected scenario, where the lower throughput reduces the number of observations available for each operation type. Therefore, latency values for less frequent operations should be interpreted with more caution than latency values for frequently sampled reads and updates.

Table 7.15 reports the median latency of each macrobenchmark operation across the plaintext, secure, plaintext latency-injected, and secure latency-injected scenarios.

Table 7.14: Executed operations per macrobenchmark scenario.

Operation	Plain	Secure	Plain + latency	Secure + latency
getActiveDENMs	33531	71	31719	19
getDENM	762698	1348	727935	299
createDENM	33401	74	31799	18
updateDENM	628569	1080	601078	209
removeDENM	33152	72	31564	21
getActiveIVIMs	33784	93	32110	20
getIVIM	766032	1505	727783	326
createIVIM	33435	71	31654	25
updateIVIM	832096	1562	790616	366
removeIVIM	32981	67	31526	22
reconfigure	-	144	-	38

Table 7.15: Median latency of macrobenchmark operations.

Operation	Plain	Secure	Plain + latency	Secure + latency
getActiveDENMs	20.82	15909.40	19.85	17796.32
getDENM	1.14	12.85	1.27	12.80
createDENM	16.40	152.16	15.59	543.51
updateDENM	1.10	10.12	1.12	70.20
removeDENM	12.00	83.78	11.46	120.88
getActiveIVIMs	19.34	27809.97	18.77	23530.90
getIVIM	1.15	12.86	1.27	12.97
createIVIM	15.84	146.88	15.31	566.95
updateIVIM	0.97	8.67	0.99	40.85
removeIVIM	11.58	74.26	11.22	147.54
reconfigure	–	54.54	–	35.86

The plaintext baseline and plaintext latency-injected scenarios remain close in both operation counts and median latency. This is expected because the injected latency affects asynchronous CRDV replication between replicas, whereas the benchmark measures the completion time of each operation at the replica receiving the client request. Therefore, replication delay affects how quickly updates become visible at other replicas, but it has only a limited effect on the latency of the local operation observed by the benchmark. Small differences between plaintext values across the two scenarios are expected because the benchmark is duration-based and each run reaches a slightly different replicated state.

The table shows that secure operation costs are highly uneven. Operations that read or update known message fields, such as `getDENM`, `getIVIM`, `updateDENM`, and `updateIVIM`,

remain in the millisecond-to-tens-of-milliseconds range. In contrast, operations that enumerate active private identifiers, namely `getActiveDENMs` and `getActiveIVIMs`, are several orders of magnitude more expensive in secure mode.

This behaviour is consistent with the microbenchmark results in Section 7.2.4 and with the expected cost of MPC operations that require repeated private comparisons. The active message sets use private keys, and therefore reading them requires secure containment checks over secret-shared identifiers rather than direct database-key lookups. As the number of private identifiers increases, the number of secure comparisons also increases, making these operations substantially more expensive than direct reads of known message fields.

This gap becomes clearer when operations are grouped by the type of secure work they perform. In the baseline secure run, operations that read or update known message fields, namely `getDENM`, `getIVIM`, `updateDENM`, and `updateIVIM`, have an average median latency of 11.13 ms. In contrast, operations that enumerate or modify private active-message sets, namely `getActiveDENMs`, `getActiveIVIMs`, `createDENM`, `createIVIM`, `removeDENM`, and `removeIVIM`, have an average median latency of 7362.74 ms. Thus, private set-oriented operations are approximately 661.82 times more expensive than field-oriented operations in the secure baseline. This aggregate is dominated by active-set enumeration, whose median latencies are orders of magnitude higher than creation and removal operations.

The same pattern appears in the latency-injected secure run. Field-oriented operations have an average median latency of 34.21 ms, while private set-oriented operations have an average median latency of 7117.68 ms. In this case, private set-oriented operations remain approximately 208.09 times more expensive. This confirms that the main cost difference is driven by the structure of the secure operations themselves, with injected latency further amplifying the cost of interactive protocols.

The reconfiguration operation has a median latency of 54.54 ms in the baseline scenario and 35.86 ms with injected latency. These values should be interpreted as indicative rather than as a complete characterisation of reconfiguration performance in the macrobenchmark, because reconfiguration is sampled less frequently than the other operations and each run reaches a different amount of protected state. Nevertheless, the results show that quorum changes can be executed within the mixed workload and do not dominate the total execution time compared with private active-set operations.

Figure 7.7 visualises the same latency results. The top plot corresponds to the baseline run, while the bottom plot corresponds to the latency-injected run. The y-axis uses a logarithmic scale because the latency gap between plaintext and secure execution is large. Error bars represent p95 latency.

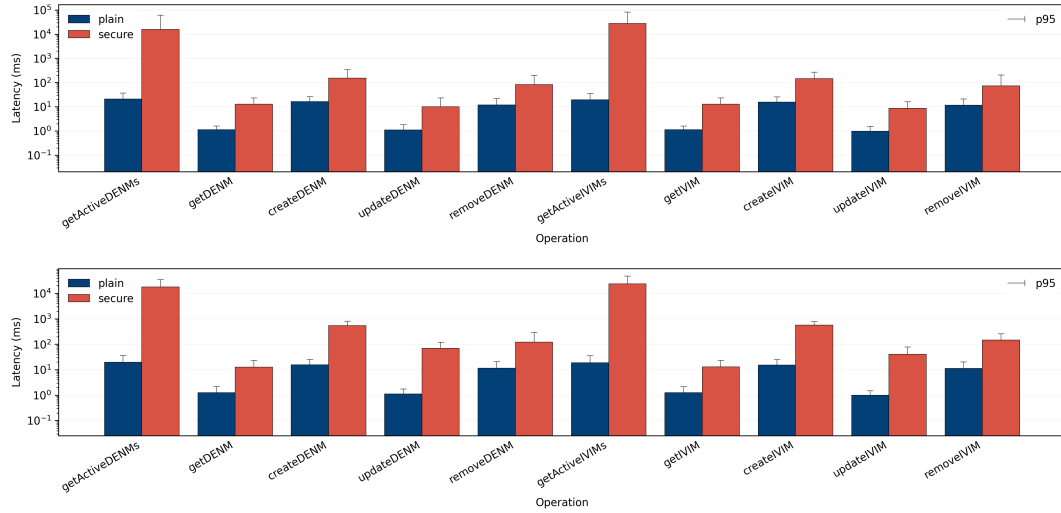


Figure 7.7: Median latency per macrobenchmark operation in the baseline scenario (top) and in the injected-latency scenario (bottom).

The plots reinforce the conclusions drawn from Table 7.15. In both scenarios, the dominant factor is the type of secure operation being executed, rather than the injected network latency alone. Operations over known message fields remain comparatively cheap, while private active-set enumeration dominates the secure latency.

The visual comparison also shows why the latency-injected run should not be interpreted only operation by operation. Some active-set operations show lower median latency in the latency-injected run, even though communication costs are higher. This is a consequence of the duration-based workload: the latency-injected scenario completes fewer operations and can therefore accumulate fewer active private identifiers before enumeration operations are sampled. Since the cost of active-set enumeration depends on the number of private identifiers to compare, these operations may appear faster in that run. The figure should therefore be interpreted together with the throughput and operation-count results.

Additional time-series plots showing the evolution of p95 latency and throughput during the macrobenchmark are reported in Appendix A.2. The main text focuses on aggregate throughput and median per-operation latency, since these metrics are sufficient to explain the dominant performance trends observed in the benchmark.

Overall, the macrobenchmark confirms the trends observed in the microbenchmarks. Secure register- and map-based operations introduce measurable overhead, but remain significantly cheaper than operations over private sets. The main performance bottleneck is the use of private active sets for DENM and IVIM identifiers. Because the benchmark is duration-based, throughput is the clearest metric for comparing complete scenarios, while per-operation latency is used to identify where the cost comes from. The results show that secure set handling dominates the cost of the mixed workload.

7.3.6 Discussion

The macrobenchmark results show that the performance of the prototype is strongly influenced by the composition of the application-level workload. While the microbenchmarks isolate individual secure CRDV operations, the macrobenchmark shows how these operations interact in a complete workload involving message reads, updates, insertions, removals, active-set enumeration, and quorum reconfiguration.

The main result is that secure execution introduces a substantial throughput reduction when compared with plaintext execution. This reduction is not uniform across all operation types. Operations that access or update known private values, such as `getDENM`, `getIVIM`, `updateDENM`, and `updateIVIM`, remain comparatively cheaper because they are mostly based on private registers and map lookups. These operations still pay the cost of invoking the secure runtime and executing secure protocols, but they do not require searching over large private collections.

In contrast, operations over private active sets dominate the cost of the secure workload. The `getActiveDENMs` and `getActiveIVIMs` operations are significantly more expensive than the remaining application-level operations because the active message identifiers are stored as private set keys. Enumerating these sets requires secure computation over the set contents, including comparisons over private values.

This cost is further amplified by the use of a secure 2P-Set to represent active message identifiers. As described in Section 7.3.3, the active set is derived from two internal sets: an add-set and a remove-set. In plaintext execution, computing this difference is inexpensive because identifiers can be compared directly. In secure execution, however, identifiers are private and equality must be evaluated through secure comparisons. Consequently, computing the active set may require comparing elements from the add-set

with elements from the remove-set. If the add-set contains a elements and the remove-set contains r elements, this step can require work proportional to $a \cdot r$. Therefore, unlike a simple secure set operation whose cost grows mainly with the number of stored elements, active-set enumeration in the secure 2P-Set can exhibit quadratic growth as both internal sets grow.

This explains why `getActiveDENMs` and `getActiveIVIMs` are the dominant operations in the secure macrobenchmark. These operations do not only access private identifiers; they also compute the active view of a private 2P-Set. As the workload creates and removes messages, both internal sets grow, increasing the number of private comparisons required to reconstruct the active view. This suggests that secure CRDT and CRDV designs should take into account not only the sequential semantics of a data type, but also whether its internal representation is friendly to secure computation. In particular, representations that require repeated private search or comparison over growing sets may be less suitable than alternatives that reduce the amount of value-dependent secure computation.

The duration-based nature of the benchmark is also important when interpreting the results. Higher-throughput scenarios execute many more operations during the same two-hour window and therefore evolve the replicated state further. Lower-throughput scenarios, especially secure executions, complete fewer operations and reach different state sizes. For this reason, throughput is the most appropriate metric for comparing complete scenarios, while per-operation latency is best interpreted as a diagnostic tool for identifying bottlenecks.

The latency-injected scenarios show that communication delay has a stronger effect on secure execution than on plaintext execution. In plaintext execution, injected CRDV replication delay is mostly outside the critical path of local operation completion because replication is asynchronous. In secure execution, however, several operations require communication between MPC parties before returning to the client, making MPC communication part of the critical path. This explains why injected latency has a limited effect on plaintext execution but a more visible effect on secure operations. However, for operations dominated by private set size, the observed latency is affected both by communication delay and by the amount of private state accumulated during the run.

Reconfiguration behaves differently from regular application operations. Although it requires resharing protected state between quorums, it is executed less frequently than

message reads and updates. In the macrobenchmark, reconfiguration does not dominate the overall cost when compared with private active-set operations. This suggests that quorum reconfiguration is feasible as a management operation, provided that it is not executed too frequently and that the amount of protected state remains controlled.

Overall, the macrobenchmark highlights an important design trade-off for secure CRDV systems. Hiding active DENM and IVIM identifiers strengthens confidentiality, since the replicated database does not reveal which messages are active. However, this confidentiality comes at a significant performance cost, because reconstructing the active view requires repeated secure comparisons over private identifiers. This suggests that future versions of the system should explore more efficient support for private set operations, including batching private comparisons, using pre-processing for comparison protocols, caching or indexing metadata that does not reveal private identifiers, and redesigning active-set enumeration to avoid repeatedly scanning all private elements [31, 48].

Chapter 8

Conclusion and Future Work

This thesis explored how confidentiality can be added to CRDV without removing their replicated and eventually consistent nature. The starting point of this work was the observation that traditional CRDTs and CRDV replicate application state in plaintext, which is unsuitable for scenarios where replicas may belong to different administrative domains or where some values must remain confidential.

To address this problem, this thesis proposed a secure CRDV design in which selected values are represented by handles in the replicated state, while the corresponding secret-shared values are maintained by a secure MPC layer. This separation allows the CRDV layer to continue replicating metadata and resolving conflicts, while private data is processed through secure operations executed by MPC parties.

The work also introduced mechanisms for propagating shares to the parties associated with active replicas and for reconfiguring the quorum responsible for maintaining protected values. These mechanisms make it possible to adapt the secure layer when the set of parties trusted by a coordinator changes, without exposing the underlying plaintext values.

A prototype was implemented on top of a PostgreSQL-based CRDV system. The implementation supports secure operations over registers, counters, sets, maps, and lists, and integrates these operations with the existing CRDV execution model. The prototype also includes support for quorum reconfiguration and was evaluated using both microbenchmarks and a macrobenchmark based on a simplified Cooperative Intelligent Transport Systems scenario.

The evaluation showed that the cost of secure execution depends strongly on the type of operation. For registers, counters, maps, and lists, secure execution introduces a moderate and mostly stable overhead. In contrast, secure set operations are significantly more expensive because they require private comparisons over existing elements. The macrobenchmark confirmed this trend at the application level: operations over known private

values remain comparatively cheaper, while active-set enumeration dominates the cost of the secure workload.

Overall, the results show that secure CRDV are feasible for workloads where most operations access or update known private values. However, they also show that operations requiring private search or enumeration over growing collections are the main performance bottleneck.

8.1. Future Work

The results of this thesis open several directions for future work. The most important direction is the optimisation of secure operations over private collections. Both the microbenchmarks and the macrobenchmark show that secure set operations are the main performance bottleneck of the current prototype. Future work should therefore investigate techniques for reducing the number of private comparisons required by membership checks and active-set enumeration. Possible approaches include batching secure comparisons, using more efficient private indexing structures, caching authorised views, or redesigning application-level representations to avoid frequent enumeration of large private sets [31, 48].

Another important direction is to strengthen the security model by integrating Verifiable Secret Sharing (VSS). The current prototype focuses on confidentiality under the assumed behaviour of the participating MPC parties. However, in deployments where parties may behave maliciously, secret shares should also be accompanied by mechanisms that allow parties to verify the consistency and correctness of the shared values [49, 50]. Integrating VSS would make it possible to detect invalid or inconsistent shares, increasing robustness against active adversaries. This would be particularly relevant for long-lived private CRDV state, quorum reconfiguration, and resharing, where incorrect shares could otherwise compromise later secure computations.

Closely related to verifiability, another direction is to improve the communication efficiency of the secure runtime. The current prototype uses relatively direct secret-sharing protocols and focuses on integration with CRDV, rather than on optimising the underlying MPC communication. More communication-efficient MPC techniques, such as batched or bivariate secret sharing protocols for dynamic groups, could reduce the cost of operations that involve resharing or communication among many MPC parties. For example, recent work on proactive MPC for dynamic groups shows that batching and bivariate sharing

can reduce the amortized communication cost of maintaining and computing over secret-shared values in dynamic settings [51]. These ideas could be useful for reducing the cost of share propagation, quorum reconfiguration, and secure operations over long-lived private CRDV state.

A further direction is improving support for dynamic deployments in the choreographic runtime. As discussed in Section 6.5, the current Chorus-based implementation requires coordinators, parties, locations, and quorums to be known when the runtime is built. The prototype mitigates this limitation through configuration-driven code generation, which allows reconfiguration between predefined quorums. However, adding new parties or defining arbitrary new quorums still requires regenerating the runtime. Extensions to the choreography library, or alternative runtime representations, could support more dynamic membership while preserving the safety guarantees provided by choreographic programming.

Another possible direction is to explore asynchronous MPC as an alternative execution model for the secure runtime. The current prototype assumes that secure operations follow predefined communication patterns and that all involved parties progress through each protocol consistently. This model simplifies the implementation, but it can be restrictive when parties are deployed across networks with variable delays or temporary unavailability. Asynchronous MPC protocols, as introduced by Ben-Or et al. [52], could make the secure layer more tolerant to delayed parties, network jitter, and partial failures by reducing the dependency on tightly coordinated protocol execution. This would be particularly relevant for long-lived secure CRDV deployments in which MPC parties may be operated by different administrative domains.

Finally, future work could extend the privacy and policy model of the system. In the current prototype, private fields are explicitly marked by the application or schema configuration. More expressive policies could define which replicas, coordinators, users, or quorums are allowed to operate over specific private values. Integrating such policies with the secure CRDV interface would make it possible to support richer application-level confidentiality requirements while preserving the replicated programming model.

References

- [1] M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed. distributed-systems.net, 2023. [Cited on page 1.]
- [2] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” in *Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400. [Cited on pages 1, 6, and 10.]
- [3] A. Rahdari, E. Keshavarz, E. Nowroozi, R. Taheri, M. Hajizadeh, M. Mohammadi, S. Sinaei, and T. Bauschert, “A survey on privacy and security in distributed cloud computing: Exploring federated learning and beyond,” *IEEE Open Journal of the Communications Society*, vol. 6, pp. 3710–3744, 2025. [Cited on page 1.]
- [4] M. Barbosa, B. Ferreira, J. Marques, B. Portela, and N. Preguiça, “Secure conflict-free replicated data types,” Cryptology ePrint Archive, Paper 2020/944, 2020. [Online]. Available: <https://eprint.iacr.org/2020/944> [Cited on pages 2, 23, and 26.]
- [5] B. Portela, H. Pacheco, P. Jorge, and R. Pontes, “General-purpose secure conflict-free replicated data types,” Cryptology ePrint Archive, Paper 2023/584, 2023. [Online]. Available: <https://eprint.iacr.org/2023/584> [Cited on pages 2, 25, and 26.]
- [6] L. Cruz-Filipe and F. Montesi, “Choreographies, computationally,” *CoRR*, vol. abs/1510.03271, 2015. [Online]. Available: <http://arxiv.org/abs/1510.03271> [Cited on page 3.]
- [7] W. Vogels, “Eventually consistent,” *Commun. ACM*, vol. 52, no. 1, p. 40–44, Jan. 2009. [Online]. Available: <https://doi.org/10.1145/1435417.1435432> [Cited on page 5.]
- [8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “A comprehensive study of Convergent and Commutative Replicated Data Types,” Inria – Centre Paris-Rocquencourt ; INRIA, Research Report RR-7506, Jan. 2011. [Online]. Available: <https://inria.hal.science/inria-00555588> [Cited on pages 6 and 8.]

- [9] N. M. Preguiça, “Conflict-free replicated data types: An overview,” *CoRR*, vol. abs/1806.10254, 2018. [Online]. Available: <http://arxiv.org/abs/1806.10254> [Cited on pages 6, 8, and 26.]
- [10] P. S. Almeida, A. Shoker, and C. Baquero, “Delta state replicated data types,” *Journal of Parallel and Distributed Computing*, vol. 111, p. 162–173, Jan. 2018. [Online]. Available: <http://dx.doi.org/10.1016/j.jpdc.2017.08.003> [Cited on page 6.]
- [11] D. Evans, V. Kolesnikov, and M. Rosulek, “A pragmatic introduction to secure multi-party computation,” *Found. Trends Priv. Secur.*, vol. 2, no. 2–3, p. 70–246, Dec. 2018. [Online]. Available: <https://doi.org/10.1561/33000000019> [Cited on pages 10, 11, and 33.]
- [12] O. Goldreich, “Secure multi-party computation,” *Manuscript. Preliminary Version*, 03 1999. [Cited on pages 10, 11, and 33.]
- [13] M. Ben-Or, S. Goldwasser, and A. Wigderson, “Completeness theorems for non-cryptographic fault-tolerant distributed computation,” in *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, ser. STOC ’88. New York, NY, USA: Association for Computing Machinery, 1988, p. 1–10. [Online]. Available: <https://doi.org/10.1145/62212.62213> [Cited on page 11.]
- [14] A. Shamir, “How to share a secret,” *Commun. ACM*, vol. 22, no. 11, p. 612–613, Nov. 1979. [Online]. Available: <https://doi.org/10.1145/359168.359176> [Cited on pages 12, 13, and 57.]
- [15] D. Feng and K. Yang, “Concretely efficient secure multi-party computation protocols: Survey and more,” *Security and Safety*, vol. 1, 2022. [Cited on page 13.]
- [16] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 3rd ed. CRC Press, 2020. [Cited on page 13.]
- [17] R. Cramer, I. B. Damgård, and J. B. Nielsen, *Secure Multi-Party Computation and Secret Sharing*. Cambridge University Press, 2015. [Cited on page 14.]
- [18] N. Faria and J. Pereira, “Crdv: Conflict-free replicated data views,” *Proc. ACM Manag. Data*, vol. 3, no. 1, Feb. 2025. [Online]. Available: <https://doi.org/10.1145/3709675> [Cited on pages xv, 17, 18, and 29.]

- [19] C. Gentry, “A fully homomorphic encryption scheme,” Ph.D. dissertation, Stanford University, Stanford, CA, USA, 2009. [Online]. Available: <https://crypto.stanford.edu/craig/craig-thesis.pdf> [Cited on page 23.]
- [20] P.-A. Rault, C.-L. Ignat, and O. Perrin, “Distributed access control for collaborative applications using crdts,” in *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 33–38. [Online]. Available: <https://doi.org/10.1145/3517209.3524826> [Cited on pages 24 and 26.]
- [21] T. Renaux, S. Van den Vonder, and W. De Meuter, “Secure rdts: Enforcing access control policies for offline available json data,” *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, Oct. 2023. [Online]. Available: <https://doi.org/10.1145/3622802> [Cited on pages 24 and 26.]
- [22] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott, “Riak dt map: a composable, convergent replicated dictionary,” in *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, ser. PaPEC '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: <https://doi.org/10.1145/2596631.2596633> [Cited on page 26.]
- [23] A. Bieniusa, V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, “Antidote: The highly-available geo-replicated database with strongest guarantees,” SyncFree Project, Tech. Rep., 2016. [Online]. Available: <https://syncfree.lip6.fr/attachments/article/59/antidote-white-paper.pdf> [Cited on page 26.]
- [24] C. Meiklejohn and P. Van Roy, “Lasp: a language for distributed, coordination-free programming,” in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 184–195. [Online]. Available: <https://doi.org/10.1145/2790449.2790525> [Cited on page 26.]
- [25] O. Goldreich and R. Ostrovsky, “Software protection and simulation on oblivious rams,” *J. ACM*, vol. 43, no. 3, p. 431–473, May 1996. [Online]. Available: <https://doi.org/10.1145/233551.233553> [Cited on page 31.]
- [26] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang, “Oblivious data structures,” in *Proceedings of the 2014 ACM SIGSAC Conference*

- on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 215–226. [Online]. Available: <https://doi.org/10.1145/2660267.2660314> [Cited on page 31.]
- [27] I. Damgård, Y. Desmedt, M. Fitzi, and J. B. Nielsen, “Secure protocols with asymmetric trust,” in *Advances in Cryptology – ASIACRYPT 2007*, K. Kurosawa, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 357–375. [Cited on page 35.]
- [28] O. Alpos, C. Cachin, B. Tackmann, and L. Zanolini, “Asymmetric distributed trust,” *Distributed Computing*, vol. 37, no. 3, pp. 247–277, Sep. 2024. [Online]. Available: <https://link.springer.com/10.1007/s00446-024-00469-1> [Cited on page 35.]
- [29] T. M. Wong, C. Wang, and J. M. Wing, “Verifiable secret redistribution for threshold sharing schemes,” Carnegie Mellon University, Tech. Rep. CMU-CS-02-114, 2002. [Cited on page 41.]
- [30] R. Gennaro, M. O. Rabin, and T. Rabin, “Simplified vss and fast-track multiparty computations with applications to threshold cryptography,” in *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '98. ACM, 1998, pp. 101–111. [Cited on page 42.]
- [31] M. J. Freedman, K. Nissim, and B. Pinkas, “Efficient private matching and set intersection,” in *Advances in Cryptology – EUROCRYPT 2004*. Springer, 2004, pp. 1–19. [Cited on pages 49, 94, and 96.]
- [32] Á. Kiss, J. Liu, T. Schneider, N. Asokan, and B. Pinkas, “Private set intersection for unequal set sizes with mobile applications,” in *Proceedings on Privacy Enhancing Technologies*, vol. 2017, no. 4, 2017, pp. 177–197. [Cited on page 49.]
- [33] T. Turban, “A secure multi-party computation protocol suite inspired by shamir’s secret sharing scheme,” Master’s thesis, Norwegian University of Science and Technology, 2014. [Cited on page 49.]
- [34] “Postgresql,” <https://www.postgresql.org/>, accessed: 2026-06-22. [Cited on page 56.]
- [35] “The rust programming language,” <https://www.rust-lang.org/>, accessed: 2026-06-22. [Cited on page 56.]
- [36] “Cargo: The rust package manager,” <https://github.com/rust-lang/cargo>, accessed: 2026-06-22. [Cited on page 56.]

- [37] “pgrx: Build postgres extensions with rust,” <https://github.com/pgcentralfoundation/pgrx>, accessed: 2026-06-22. [Cited on page 56.]
- [38] “Chorus,” <https://github.com/lsc-ucsc/Chorus>, accessed: 2026-06-22. [Cited on page 57.]
- [39] “Serde,” <https://serde.rs/>, accessed: 2026-06-22. [Cited on page 57.]
- [40] “bincode,” <https://sr.ht/~stygianentity/bincode/>, accessed: 2026-06-22. [Cited on page 57.]
- [41] D. Kales, “Secret sharing,” https://www.isec.tugraz.at/wp-content/uploads/teaching/mfc/secret_sharing.pdf, accessed: 2026-06-23. [Cited on page 57.]
- [42] Shopify, “Toxiproxy: A TCP proxy to simulate network conditions,” <https://github.com/Shopify/toxiproxy>, 2026, accessed: 2026-06-24. [Cited on page 70.]
- [43] C-Roads Platform, “Cooperative intelligent transport systems (c-its),” 2020, brochure. [Online]. Available: <https://www.itsstandards.eu/app/uploads/sites/14/2020/10/C-ITS-Brochure-2020-FINAL.pdf> [Cited on page 78.]
- [44] European Telecommunications Standards Institute, “Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 3: Specifications of Decentralized Environmental Notification Basic Service,” ETSI, ETSI TS 103 831 V2.1 V2.1.1, 2022. [Cited on page 80.]
- [45] —, “Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; In-Vehicle Information Basic Service,” ETSI, ETSI TS 103 301 V2.2.1, 2024. [Cited on page 80.]
- [46] A. Ulbel, “Analysis of v2x performance and rollout status with a special focus on austria,” Master’s thesis, Graz University of Technology, 2021. [Cited on page 80.]
- [47] V. Battista, “Development of infrastructure-to-vehicle module for vehicular communication emulation through the ms-van3t framework,” Master’s thesis, Politecnico di Torino, 2021. [Online]. Available: <https://webthesis.biblio.polito.it/21175/1/tesi.pdf> [Cited on page 80.]
- [48] D. Beaver, “Efficient multiparty protocols using circuit randomization,” in *Advances in Cryptology — CRYPTO ’91*, J. Feigenbaum, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 420–432. [Cited on pages 94 and 96.]

- [49] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch, “Verifiable secret sharing and achieving simultaneity in the presence of faults,” in *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*. IEEE, 1985, pp. 383–395. [Cited on page 96.]
- [50] P. Feldman, “A practical scheme for non-interactive verifiable secret sharing,” in *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*. IEEE, 1987, pp. 427–438. [Cited on page 96.]
- [51] K. Eldefrawy, T. Lepoint, and A. Leroux, “Communication-efficient proactive MPC for dynamic groups with dishonest majorities,” Cryptology ePrint Archive, Paper 2021/1564, 2021. [Online]. Available: <https://eprint.iacr.org/2021/1564> [Cited on page 97.]
- [52] M. Ben-Or, R. Canetti, and O. Goldreich, “Asynchronous secure computation,” in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, ser. STOC '93. New York, NY, USA: Association for Computing Machinery, 1993, pp. 52–61. [Online]. Available: <https://doi.org/10.1145/167088.167109> [Cited on page 97.]

Chapter A

Detailed Macrobenchmark Results

A.1. Time-Binned Operation Latency

Tables A.1 and A.2 report the evolution of median operation latency over the two-hour macrobenchmark runs. Values are in milliseconds; the number of observations is shown in parentheses. Columns P and S denote plaintext and secure execution, respectively.

Table A.1: Baseline macrobenchmark latency evolution by operation.

Operation	0.0–0.5h		0.5–1.0h		1.0–1.5h		1.5–2.0h	
	P	S	P	S	P	S	P	S
Get active DENMs	7.38 (11391)	3582.57 (23)	19.58 (8464)	14245.16 (19)	28.27 (7220)	35837.41 (17)	35.55 (6455)	50516.64 (12)
Get DENM	1.09 (257240)	12.78 (368)	1.15 (192530)	12.84 (451)	1.17 (166541)	13.00 (284)	1.18 (146387)	12.87 (245)
Create DENM	8.51 (11261)	124.31 (22)	15.62 (8604)	121.42 (24)	20.61 (7246)	153.83 (16)	24.68 (6290)	182.94 (12)
Update DENM	1.05 (211249)	10.99 (265)	1.12 (159903)	9.77 (369)	1.14 (137321)	10.31 (261)	1.15 (120096)	9.98 (185)
Remove DENM	4.50 (11237)	32.90 (24)	11.29 (8447)	89.40 (21)	16.12 (7088)	98.73 (15)	20.18 (6380)	143.29 (12)
Get active IVIMs	7.25 (11710)	4157.57 (24)	18.39 (8568)	19514.46 (26)	27.06 (7152)	40779.87 (23)	34.32 (6354)	50277.21 (19)
Get IVIM	1.11 (260139)	12.76 (441)	1.16 (194009)	12.86 (521)	1.17 (165904)	12.96 (295)	1.18 (145980)	13.04 (248)
Create IVIM	8.43 (11431)	99.96 (30)	15.15 (8521)	148.36 (21)	20.13 (7179)	186.65 (13)	24.22 (6304)	198.42 (7)
Update IVIM	0.92 (282278)	8.42 (473)	0.99 (210643)	8.71 (495)	1.00 (180834)	8.81 (325)	1.01 (158341)	8.72 (269)
Remove IVIM	4.49 (11264)	46.03 (29)	10.81 (8236)	85.27 (18)	15.73 (7169)	109.16 (15)	19.75 (6312)	107.28 (5)

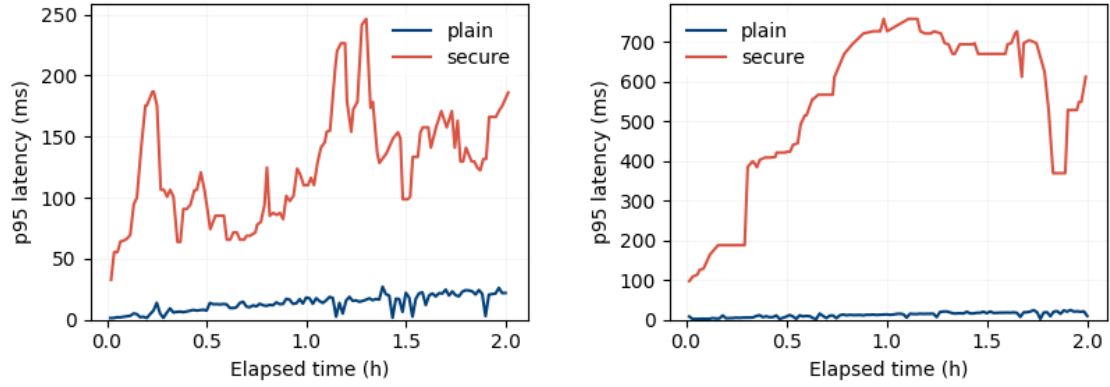
Table A.2: Latency-injected macrobenchmark latency evolution by operation.

Operation	0.0–0.5h		0.5–1.0h		1.0–1.5h		1.5–2.0h	
	P	S	P	S	P	S	P	S
Get active DENMs	7.01 (10301)	9770.70 (7)	18.28 (8307)	17796.32 (5)	26.94 (6913)	28990.57 (1)	34.44 (6198)	31884.52 (6)
Get DENM	1.23 (234692)	13.17 (117)	1.27 (190829)	12.02 (23)	1.29 (160250)	11.94 (62)	1.30 (142164)	12.01 (97)
Create DENM	8.69 (10289)	480.54 (10)	14.64 (8419)	757.94 (3)	19.89 (6928)	674.23 (2)	23.90 (6163)	624.56 (3)
Update DENM	1.09 (193652)	69.94 (93)	1.10 (158258)	68.93 (19)	1.16 (132253)	71.11 (44)	1.17 (116915)	76.42 (53)
Remove DENM	4.52 (10171)	71.44 (13)	10.50 (8318)	260.70 (2)	15.31 (6877)	294.21 (1)	19.39 (6198)	196.62 (5)
Get active IVIMs	6.87 (10780)	8515.92 (6)	17.53 (8332)	21851.94 (7)	26.38 (6869)	27682.44 (1)	33.73 (6129)	40616.34 (6)
Get IVIM	1.24 (236311)	13.18 (156)	1.27 (190636)	12.44 (59)	1.29 (159657)	11.12 (12)	1.29 (141179)	11.99 (99)
Create IVIM	8.54 (10339)	423.14 (11)	14.43 (8387)	526.24 (5)	19.66 (6924)	696.72 (1)	23.68 (6004)	746.06 (8)
Update IVIM	0.97 (255823)	40.05 (195)	0.97 (207023)	41.00 (61)	1.02 (173841)	38.44 (11)	1.03 (153929)	60.67 (99)
Remove IVIM	4.42 (10228)	102.03 (12)	10.26 (8270)	199.72 (7)	15.18 (6816)	369.37 (1)	19.13 (6211)	217.58 (2)

A.2. Time-Series Results

Figures A.1 and A.2 show the evolution of tail latency and throughput over time. Tail latency is reported using the 95th percentile, or p95. A p95 latency value means that 95% of the observed operations completed at or below that latency, while the remaining 5% took longer.

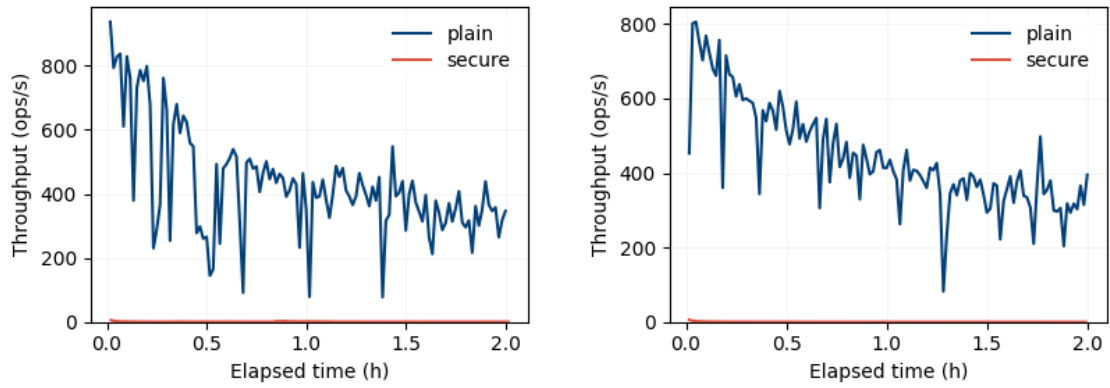
The p95 latency plots show the tail behaviour of the workload, while the throughput plots show how many operations are completed per second over time. Plaintext execution remains comparatively stable in both metrics. Secure execution presents substantially



(a) Baseline

(b) Injected latency

Figure A.1: Evolution of p95 latency during the macrobenchmark.



(a) Baseline

(b) Injected latency

Figure A.2: Evolution of throughput during the macrobenchmark.

higher tail latency and much lower throughput. In the secure runs, p95 latency is consistently above plaintext execution and tends to grow as the replicated state evolves, although the exact shape of the curve depends on the operations sampled during each time window. The throughput plots show the practical impact of this cost: secure execution completes far fewer operations per second, especially when latency is injected.