Security in Data Aggregation for Eventually Consistent Systems

Pedro Miguel de Jesus Jorge

Mestrado em Segurança Informática
Departamento de Ciência de Computadores
2022

Orientador
Bernardo Luís Fernandes Portela, Professor Auxiliar, Faculdade de Ciências da Universidade do Porto

Coorientador
Manuel Bernardo Martins Barbosa, Professor Associado, Faculdade de Ciências da Universidade do Porto

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ____/____/______

Abstract

The ever growing globalization of the modern world requires online applications to maintain high levels of availability that are only achieved through replication, while applications' functionality often does not require strong levels of consistency. Conflict-free Replicated Data Types (CRDTs) are abstract data types that provide eventual consistency guarantees in distributed systems, automatically dealing with concurrent operations with no need for synchronization. Society concerns about security and privacy are also on the rise, but despite the extensive study of CRDTs in literature and their use in the industry, very little development exists in relation to privacy-preserving CRDT solutions. Existing work requires each construction to be designed specifically on a per-case basis.

I present an approach to privacy-preserving CRDTs that leverages Multi-Party Computation (MPC) in order to lift any CRDT construction to its secure variant. The proposed system maps each replica in the CRDT network to a group of MPC parties that are responsible for all computation over secret values, while being agnostic to the network topology and to the multi-party protocols inner workings.

I build on this approach by proposing designs for register, counter, maximum value and set CRDTs. These designs are experimentally validated with an implementation that uses Sharemind MPC protocols, exhibiting the impact each construction has on performance through an evaluation of latency and throughput for each operation.

Resumo

A crescente globalização do mundo moderno requer que aplicações online mantenham altos níveis de disponibilidade que apenas são atingíveis com recurso a replicação, enquanto que frequentemente a funcionalidade dessas aplicações não exige consistência forte. Conflict-free Replicated Data Types (CRDTs) são um tipo abstrato de dados que fornece garantias de consistência eventual e, sistemas distribuídos, lidando automaticamente com operações concorrentes sem necessitar de sincronização. A preocupação da sociedade com segurança e privacidade tem vindo também a crescer, mas apesar do extenso estudo de CRDTs na literatura e o seu uso na indústria, existe muito pouco desenvolvimento no que toca a soluções de CRDTs que preservem privacidade. O trabalho existente requer que cada construção seja desenhada especificamente para cada caso de uso.

Apresento uma abordagem a CRDTs privados que se baseia em Multi-Party Computation (MPC) para elevar qualquer construção de CRDT à sua variante segura. O sistema proposto mapeia cada réplcia na rede de CRDTs a um group de entidades MPC que são responsáveis por todas as computações sobre valores secretos, sendo ao mesmo tempo agnóstica à topologia da rede e às pecularidades dos protocolos de MPC usdos.

Desenvolvo esta abordagem propondo desenhos de CRDTs para registos, contadores, valor máximo e conjuntos. Estes desenhos são validados eexperimentalmente com uma implementação que utiliza os protocolos Sharemind para MPC, mostrando o impacto que cada construção tem no desempanho através da avaliação da latência e débito para cada operação

Declaração de Honra

Eu, Pedro Miguel de Jesus Jorge, inscrito no Mestrado em Segurança Informática da Faculdade de Ciências da Universidade do Porto declaro, nos termos do disposto na alínea a) do artigo 14.º do Código Ético de Conduta Académica da U.Porto, que o conteúdo da presente dissertação reflete as perspetivas, o trabalho de investigação e as minhas interpretações no momento da sua entrega.

Ao entregar esta dissertação, declaro, ainda, que a mesma é resultado do meu próprio trabalho de investigação e contém contributos que não foram utilizados previamente noutros trabalhos apresentados a esta ou outra instituição.

Mais declaro que todas as referências a outros autores respeitam escrupulosamente as regras da atribuição, encontrando-se devidamente citadas no corpo do texto e identificadas na secção de referências bibliográficas. Não são divulgados na presente dissertação quaisquer conteúdos cuja reprodução esteja vedada por direitos de autor.

Tenho consciência de que a prática de plágio e auto-plágio constitui um ilícito académico.

Pedro Miguel de Jesus Jorge

Porto, 30 de setembro de 2022

Agradecimentos

Em primeiro lugar agradeço ao Professor Bernardo Portela por todo o acompanhamento e tutela ao longo deste trabalho. A sua acessibilidade e profissionalismo foram essenciais para a criação de um ambiente de trabalho apelativo e eficiente.

Quero agradecer ao Rogério Pontes pela sua disponibilidade em partilhar comigo o seu conhecimento e experiência profissional. Esta partilha e o seu apoio, acima de tudo no desenvolvimento e validação experimental do trabalho, foram indispensáveis e contribuíram largamente para a elevação da qualidade do mesmo.

Agradeço também ao Professor Hugo Pacheco pela sua contribuição para a discussão sobre o desenho das construções propostas.

Expresso a minha gratidão para com a minha família pelo apoio e aconselhamento ao longo de toda a minha vida, e por me darem a oportunidade de prosseguir a minha educação. O seu amor e dedicação fizeram de mim quem sou hoje.

Por último quero agradecer a todos os amigos e colegas que tornaram este percurso não só produtivo mas divertido e memorável.

Este trabalho foi financiado pelo projeto THEIA - POCI-01-0247-FEDER-047264.

Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

AES	Advanced Encryption Standard
CRDT	Conflict-free Replicated Data Type
DCC	Departamento de Ciência de Computadores
FCUP	Faculdade de Ciências da Universidade do Porto
LWW	Last-Writer-Wins
MAC	Message Authentication Code
MPC	Multi-Party Computation
SHA	Secure Hash Algorithm
SMPC	Secure Multi-Party Computation

Chapter 1

Introduction

As modern technology adapts in order to better serve an ever more globalized world, large-scale distributed systems are becoming more and more prevalent [15, 16, 23]. Replication and consistency are thus essential features of any such system. One popular class of distributed applications are *replicated stores*: systems composed of several computers that act as replicas, often geo-replicated, that maintain multiple copies of shared information and routinely perform exchanges to stay synchronized, while clients can interact with data from any of these replicas at any given time. Consider, for example, online applications that may combine cloud-deployed replicas with on-device replicas in order to support offline use.

These types of systems require special attention to the balance between *consistency* (the property that says that every replica in the system has the same view of data at a given point in time) and *availability* (the property that says a system will respond to any request in an acceptable time interval). The standard approach of *strong consistency* serializes updates in a global total order. However, this approach, which requires permanent synchronization betwen all replicas, often results in availability problems and limits performance and scalability. An alternative is to provide *eventual consistency*: updates happen locally at any given replica, without synchronization, and are later sent to other replicas. Eventually, if all updates cease and enough propagation operations have been executed, every update takes effect on all replicas. As these updates do not follow a global total order, concurrent updates may generate conflicts, which often require manual arbitration and even rolling back updates. Although eventual consistency is enough for a myriad of applications, conflict resolution may prove to be troublesome. Some ad-libitum solutions such as Dynamo, Amazon's highly available key-value store [16], are prone to concurrency anomalies.

Conflict-free Replicated Data Types (CRDTs) [34] are a class of distributed data structures that provide eventual consistency. These structures leverage mathematical properties such as commutativity and monotonicity in order to guarantee that replicas that have received the same updates have the same state, automatically merging conflicting updates without synchronization. Some examples of systems built using CRDTs include collaborative text editors [26], geo-replicated databases [1] and chat systems for world-wide online video games [31].

This widespread adoption of cloud-based systems has increased awareness about privacy, and this concern extends to CRDT-based systems. European data protection policies state [3]:

> (...) the controller shall, both at the time of the determination of the means for processing and at the time of the processing itself, implement appropriate technical and organisational measures, (...), which are designed to implement data-protection principles (...).

Ensuring data protection in a CRDT-based system is not easily achieved through standard encryption techniques, as it would require computations over encrypted data on the replica's side of the system in order to perform propagation which is exactly what these techniques are designed to avoid, in order to provide integrity guarantees. *Barbosa et al.* [6] present the first theoretically sound proposal to secure CRDTs, defining tailor made constructions for registers, sets and counters. Each construction is carefully designed in order to perform the necessary computations over encrypted data.

Much of the literature on CRDTs [19, 33, 34] is focused on the design of new structures with slightly different behaviors, to suit different application requirements. The above-mentioned approach to security has no answer this constant expansion of available CRDT constructions, as each construction would have to be given special attention, which is not an efficient process. An interesting development would be the ability to directly lift any CRDT construction to its secure variant. The approach used in [6] does not allow this, as computations over encrypted data have limited expressiveness and the behavior of each CRDT would need to be manually determined.

MPC denotes a collection of cryptographic protocols that enable multiple untrusted parties to compute a function on joint input while not disclosing their private data. These techniques are well suited for privacy-preserving operations in a distributed system. The theory behind this technique is fairly well developed, and recent practical advances have prompted a myriad of MPC frameworks [18] for easily writing distributed MPC protocols over partitioned secret data.

Using MPC to enable the necessary computations in CRDT systems would be ideal. This approach was even considered in [6], but discarded:

> Intuitively, privacy-preserving CRDT operations is realisable using (...) general secure multi-party computation. However, [MPC] solutions (...) would require sharing secret data between multiple nodes, which goes against the purpose of CRDTs in the first place.

At first sight, CRDT and MPC might seem two antagonistic concepts if each CRDT replica is mapped to an MPC party. On the other hand, this is in fact not the case if each CRDT replica is mapped to a group of MPC parties that communicate between each other in order to compute its functionality. As an example, a system could be composed of several geo-replicated replicas, each of which comprised of several MPC parties deployed in separate cloud providers, at the replica's location. In this work I study this possibility.

My contributions. This dissertation presents the following contributions:

- I propose an approach to CRDT security that leverages Multi-Party Computation (MPC) to directly transpose CRDT construction to their secure variant, while not limited by functionality;
- I present detailed secure CRDT constructions for a register, grow-only counter, pn-counter, maxvalue, boundedcounter and grow-only set;
- I provide an open-source implementation of these protocols, and their experimental validation.

After this introduction, Chapter 2 starts by providing a background knowledge on several essential security concepts such as confidentiality and adversary models, MPC and CRDTs. Chapter 3 covers the related literature and details the current state of the art, concerning distributed storage, CRDTs and multi-party computation. Chapter 4 introduces the design of the proposed system, as well as developed specifications for the following CRDT construction archetypes: registers, counters, maximum value and sets. Chapter 5 details the implementation of this system and its experimental validation. It presents the used methodology, the experimental setup characteristics and the results, as well as discussion on the obtained results. Finally, chapter 6 reiterates the problem, consolidates results and personal opinions and provides potential approaches for future work.

Chapter 2

Background

This chapter presents in a concise manner all the background knowledge necessary to understand the present work. It starts with an introduction to security concepts such as confidentiality and availability. Then, it goes into Conflict-free Replicated Data Types (CRDTs), explaining what they are, what problem they solve and how they can be constructed. Finally, it addresses multi-party computation, some of its most known methods, and assesses its strengths and weaknesses.

2.1 Security

Systems and information security measures are guided by and evaluated through the CIA triad, a model composed by three factors: confidentiality, integrity and availability. A proper knowledge of these factors is of extreme importance in order to understand what the problem is and what is trying to be accomplished in any project that relates to systems and information security, and thus the present chapter begins with this topic.

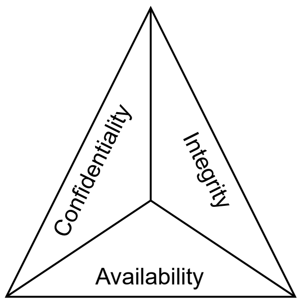


Figure 2.1: Confidentiality, integrity and availability triad

- **Confidentiality** refers to the guarantees that certain data remains secret or private. It encompasses all efforts and measures taken to inhibit data from being accessed by anyone other than the authorized entities. Taking this into account, confidentiality has two main components: making sure that unauthorized entities have access to the information, and making sure that authorized entities have that access whenever necessary. As mentioned in the introductory section of this document, guarantees of confidentiality are the main objective of the secure system proposed in this work, and so it is given the most attention out of the three factors.

This guarantee can be achieved through several methods (usually combined), such as the use of encryption, the implementation of an access control system based in authentication and authorization mechanisms or the use of secure multi-party computation protocols.

- **Integrity** involves all methods to detect unauthorized data alterations. As the main focus of this work is confidentiality, integrity is beyond its scope, it will not be developed further.
- **Availability** relates to the guarantee that the data can be accessed whenever necessary. It holds a special place in relation to the two previously explained factors, as even confidential data whose integrity has been assured is useless if it is not available. It includes not only if the access to the data is possible, but also how quickly it is done. This factor is another also important to the present work, as the main purpose of eventual consistency is to provide better availability than strong consistency.

2.1.1 Cryptographic mechanisms for security

A multitude of cryptographic mechanisms and standards were created to provide guarantee for the aforementioned security factors.

Confidentiality is achieved mainly through the use of encryption with standards such as Advanced Encryption Standard (AES), which is a way of scrambling data so that only authorized entities can understand the information. It requires the use of cryptographic keys, a set of mathematical values that both the sender and recipient of an encrypted message agree on, which can be symmetric (both users have the same key that is used for their communication) or asymmetric (each user has a pair of keys: the public key can be used by any other user to encrypt data so that it can only be decrypted by the former, with the use of its private key). Encryption can be applied directly to information or to channels, such as in the SSL standard.

Integrity is usually achieved through the use of hashing mechanisms, such as Secure Hash Algorithm (SHA), and Message Authentication Codes (MACs). Hashing mechanisms transform any data into a usually short and fixed-length value, and this is a deterministic process (a certain input will always result in the same output, if the hashing protocol and secret is constant). Given that anyone (even a potential adversary) can compute the output of a secure Hash Function, MACs are instead used: a secret key is hashed in conjunction with the data to send. This allows the recipient to verify that a message comes from the intended sender and has not been tampered with, as a potential attacker would not know the secret key, meaning he couldn't forge an authentic MAC. Combining hashes with asymmetric key cryptography results in digital signatures, which also provide non-repudiation: an entity can use its own private key to encrypt the hash of a message. Any other entity with access to the first entity's public key can use it to verify that the message was sent by the first entity.

Availability can also be amplified through cryptographic mechanisms, but it has a more direct relation with measures such as replication and redundancy.

These standard cryptographic mechanisms for confidentiality and integrity make it fundamentally impossible to perform computation over secure data. If we have two integers that are encrypted it is not trivial to get the result of their multiplication without decryption. In order to provide these desired guarantees while allowing computation over secure data, more advanced security mechanisms are necessary. One example is secure multi-party computation.

2.1.2 Secure multi-party computation

Secure Multi-Party Computation (SMPC) is a generic cryptographic primitive that enables distributed parties to jointly compute an arbitrary functionality without revealing their own private inputs and outputs [4].

Informally, this means that two or more participants (which for simplicity are referred to as "players") that hold private inputs want to perform some computation over these inputs and receive the respective output. This must be accomplished while gaining no knowledge over each other's inputs, other than what can be inferred from the received output. This definition results in a broad scope, as any cryptographic procedure that involves two or more participants may be considered an SMPC procedure. SMPC considers one or more potential corrupt players (participants that act maliciously), whose intent may be to discover other players; private information or to cause errors in the procedure. MPC protocols involve some building blocks such as secret sharing, garbled circuits, oblivious transfer, homomorphic encryption and / or zero-knowledge proofs. Concretely, we will leverage secret sharing based MPC protocols.

MPC protocols allow for the computation ensuring properties such as:

- Correctness: the computation output must be correct
- Privacy: players must only be able to obtain their own outputs and no information about other players' inputs or outputs;
- Independence of input: inputs from a player must be independent from the inputs of other players;
- Guarantee of output: corrupt players must not be able to stop honest players from receiving their outputs;
- Fairness: corrupt players must receive their outputs if and only if honest players receive their own.

2.1.3 Secret sharing

Secret sharing schemes distribute some private data among multiple players, so that the private data is only accessible if a threshold of players gather their own data. An attacker that achieves access to some of the shares must gain no information about the private data. Secret shares possess homomorphic properties such as additive homomorphism, that enable the computations serving as a base for these schemes. Figure 2.2 exemplifies how an arithmetic protocol that takes advantage of this properties in a trivial way is implemented through secure multi-party computation.

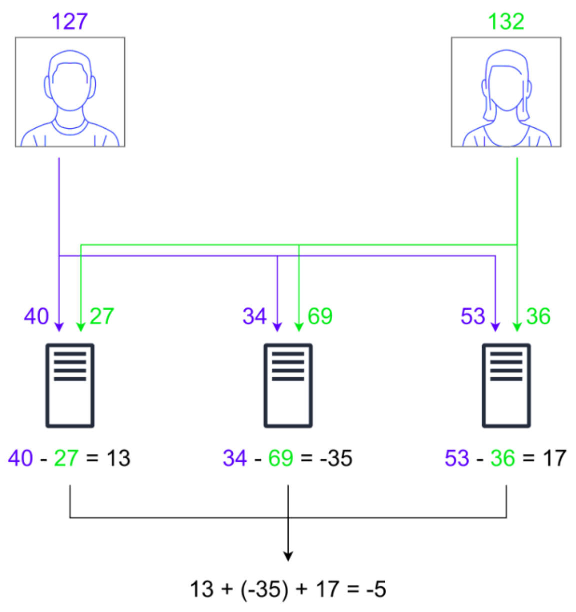


Figure 2.2: Overview of simple MPC procedure

Two individuals hold a secret number, and wish to know which one has the bigger number. Each of them divides his number in three shares, and distribute this shares in three machines. Each machine then calculates the difference between both values it received, and the subsequent sum of these differences results in the overall difference between the two initial numbers, in a way that none of the three machines has access to the initial numbers.

This addition protocol is represented in a simplified and visual way in order to facilitate a quick understanding of secret sharing, but more complex protocols exist. *Bogdanov et al.* propose a three-party multiplication protocol [11], also enabled by this additive homomorphism property, that is quite more interesting as it requires communication between parties.

Specification 1 presents the multiplication protocol, which takes two secret values $[[u]]$ and $[[v]]$ and returns a secret value $[[w]]$ such that $w = u \times v$. The protocol starts with a resharing of both inputs, a simple protocol to generate fresh shares for each player, turning $[[u]]$ and $[[v]]$ to $[[u']]$ and $[[v']]$, respectively, and each party P_i then sends their shares to the next party $(P_{(i+1)\%3})$. Now that every party has access to two shares of each secret value, they calculate their result by

Specification 1 MPC multiplication protocol, adapted from [11]

```
Input: Shared value $[[u]]$ and $[[v]]$
Output: Shared value $[[w']]$ such that $w' = u \times v$
1: $[[u']] \leftarrow reshare([[u]])$    ▷ simple protocol to generate fresh shares for each player
2: $[[v']] \leftarrow reshare([[v]])$
3: $P_1$ sends $u'_1$ and $v'_1$ to $P_2$
4: $P_2$ sends $u'_2$ and $v'_2$ to $P_3$
5: $P_3$ sends $u'_3$ and $v'_3$ to $P_1$
6: $P_1$ computes $w_1 \leftarrow u'_1 \times v'_1 + u'_1 \times v'_3 + u'_3 \times v'_1$
7: $P_2$ computes $w_2 \leftarrow u'_2 \times v'_2 + u'_2 \times v'_1 + u'_1 \times v'_2$
8: $P_3$ computes $w_3 \leftarrow u'_3 \times v'_3 + u'_3 \times v'_2 + u'_2 \times v'_3$
9: Return $[[w']] \leftarrow reshare([[w]])$
```

performing a matrix multiplication of the shares. Consider the following demonstration:

$$u = (u1 + u2 + u3)$$
$$v = (v1 + v2 + v3)$$
$$u1 \times v1 + u1 \times v2 + u1 \times v3 + u2 \times v1 + u2 \times v2 + u2 \times v3 + u3 \times v1 + u3 \times v2 + u3 \times v3 =$$
$$(u1 + u2 + u3) * (v1 + v2 + v3) = u * v$$

This result is then reshared, resulting in $[[w']]$ as the output of the protocol.

2.1.4 Adversary behavior

Threat models are representations of anything that may affects the security of a system or protocol. Its objective is to improve security by identifying potential threats and defining countermeasures, and thus the security of a system or protocol can only be discussed under a specific threat model. Although there are several topics that fall under the threat modeling scope, one of the most important for the current work is adversary behavior, which can be divided in two main types:

- *Semi-honest adversary:* Semi-honest adversaries, also known as honest-but-curious, are those that have access to complete information on the internal status of the system but may only use that knowledge according to the guidelines of the protocol. Although a weak adversary model it applies to several real world scenarios: consider a protocol that implies collaboration between companies, where these companies can not behave in a notoriously dishonest way due to the potential reputation effect but may try to collect as much private information about other participants as possible.
- *Malicious adversary:* Malicious adversaries are not limited to following the protocol as intended. In order to be secure against this type of adversaries, protocols must be able to detect manipulation of exchanged messages. As a stronger adversary model, it usually comes with potentially severe losses in the performance department.

2.2 Conflict-free Replicated Data Types

One fundamental concept in the study of distributed systems is called replication: the practice of keeping several copies of data in different places. Plenty of literature is focused on keeping a global total order, and approach known as "strong consistency" [25]. This approach, although necessary in some instances, has limited performance and scalability while also responding badly to faults [17].

On the other hand, the approach known as "eventual consistency" delivers better availability and performance in regards to networks which are tolerant to delays in the update of information. [32, 36]. In networks that implement this approach, operations are executed locally at any replica, with no synchronization mechanism, and is then propagated to other replicas in an asynchronous manner. This means that, eventually, every update is received by every replica, in no mandatory global order. Some work provides guidance on a theoretically-sound approach to eventual consistency [33, 34], providing frameworks that leverage mathematical properties such as commutativity to define some simple data types that can be used in this type of systems, also known as Conflict-free Replicated Data Types.

Conflict-free Replicated Data Types are a popular class of distributed data structures that present the desired characteristics: replicas that have received the same updates have the same state, automatically merging conflicting updates without synchronization. In [33] is presented a comprehensive portfolio of CRDT designs.

2.2.1 Operations

The environment in which these data types are employed consists of a distributed system with finite number of processes interconnected by an asynchronous network. Processes can either be replicas that comprise the CRDTs or unspecified clients that manage the input for the replicas.

CRDTs are composed by two phases that happen sequentially: the local phase, which encompasses interactions between the client and an available replica, and the downstream phase (also known as replication phase), which relates to interactions between different replicas. Each phase is comprised of two functions, these being:

- For the local phase, *update* receives some value which may or may not change the state of the replica, while *query* returns the current state of the replica. These operations are part of the CRDT's functional logic.
- For the downstream phase, *propagate* creates a copy of the replica's state and sends it to other replicas, while *merge* refers to the operation that receives an incoming state from a different replica and assimilates it. These two operations define how replicas synchronize their data.

In the local phase, a client chooses a replica to interact with, ideally based on points such as availability and closeness. The client can then execute an update operation that modifies the state of that one replica. The procedure then advances to the downstream phase. Likewise, the client can also execute a query operation, which retrieves the current state of the chosen replica. As part of an eventually consistent system, the result returned by the query operation may not be the most updated result at any given time.

Regarding the downstream phase, and based on communication and application requirements two main models for CRDTs exist, which make different network assumptions on when and how replication is executed: the state-based model and the operation-based model.

2.2.1.1 State-based replication

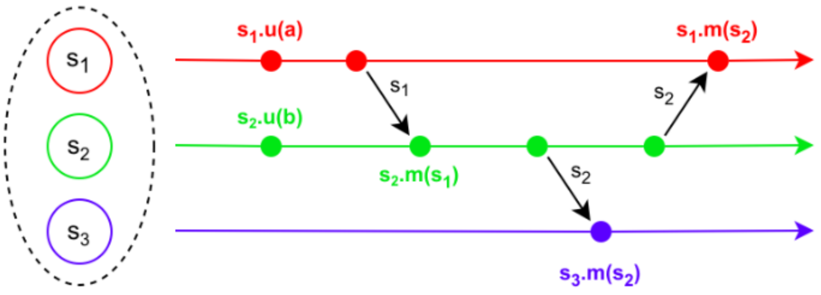


Figure 2.3: State-based CRDT, adapted from [34]

Figure 2.3 represents the overall idea behind a state-based CRDT. An update, marked as **s1.u(a)** where **a** corresponds to the update payload, modifies the state of the local replica it is applied to, as explained previously, in an atomic manner. The replica then occasionally propagates its state to other replicas, as denoted by the black arrows. The receiving replicas perform a merge operation, marked as **s2.m(s1)**, where **s2** corresponds to the replica sending its state and **s1** to the replica receiving the incoming state.

This approach is simple to understand and implement as all the required information to perform synchronization is assimilated in the state, but may become troublesome for large states, as it may require a large amount of available bandwidth. It lends itself well to container-style objects and as such the state-based approach is used, for example, in Coda, a distributed file system [20] and in Dynamo, Amazon's distributed key-value store [16].

2.2.1.2 Operation-based replication

Figure 2.4 provides an insight into how an operation-based CRDT works. In relation to the state-based approach, it disposes of the merge operation, and instead splits the update operation into a two step process: a prepare-update operation, denoted in the figure as **s1.p(a)** where **s1**

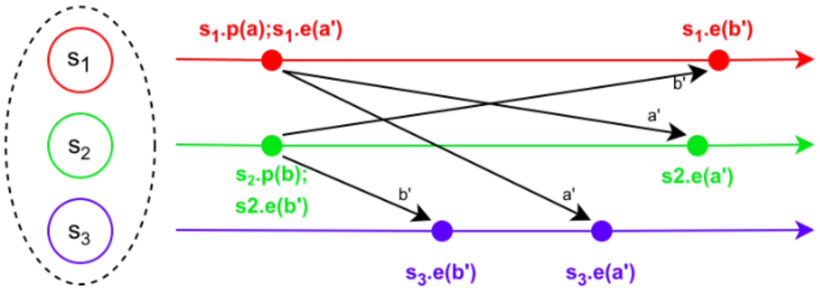


Figure 2.4: Operation-based CRDT, adapted from [34]

stands for the initial state and **a** for the update payload, and an effect-update operation, denoted in the figure as **s1.e(a')** where **a'** stands for the payload to send to other replicas. It may or may nor be the case that **a** and **a'** differ. The prepare-update operation executes locally in the source replica, immediately followed by an effect-update operation that executes at all other replicas. In literature, [34] provides proof that the operation-based CRDTs can be equivalent to the state-based CRDTs.

This approach is more complex than the previous one, as it requires replicas to maintain insight into its updates history. On the other hand, messages sent between replicas may be lighter. It has been mainly used in distributed cooperative systems, such as XRay, a collaborative text editor focused on high responsiveness for group editing [13] or TreeDoc, which achieves a similar goal [30].

2.2.2 Examples

In this section I present some of the simplest CRDT constructions, in order to make it easier for the reader to visualize what has been presented until this section of the document.

Specification 2 State-based Increment-only Counter

```
payload integer[n] P                                  ▷ n: number of replicas
    initial [0, 0, ..., 0]
update (integer v)
    $i \leftarrow replicaID()$                        ▷ i: source replica
    $P[i] \leftarrow P[i] + v$
query : integer v
    $v \leftarrow \sum_{i=0}^{n} P[i]$
    return v
propagate (X) : payload Y
    $\forall i \in [0, n-1] : Y.P[i] \leftarrow X.P[i])$
    return Y
merge (X, payload Y)
    $\forall i \in [0, n-1] : X.P[i] \leftarrow max(X.P[i], Y.P[i])$
```

Starting with a state-based construction for an increment-only counter, which can be found in Specification 2, consider the state of a replica to be an array of integers, with size equal to the number of replicas in the system and denoted as **n**. In order to perform an update operation, the source replica adds **v** to the position that corresponds to itself in the array, with **v** being the value to add to the counter. Performing a query operation means to return the sum of all the positions in the array, and thus returning the sum of the updates that were made in each replica. The propagate operation creates a payload, denoted as **Y**, with an array that is equal to the present replica's array. Lastly, a merge operation the replica receives a payload **Y** from other replica, and for each position in the array it stores the max value between its own value and the incoming value.

Specification 3 Operation-based Increment-only Counter

```
payload integer c
    initial 0
update (integer v)
    $c \leftarrow c + v$
    downstream (integer v)
query
    return c
```

On the other hand, Specification 3 corresponds to an operation-based construction for the same increment-only counter. Each replica simply stores an integer, denoted as **c** that portrays the current local state of the counter. To perform a query operation is to simply return this value. An update is split into two parts: firstly, the replica adds the incoming value, denoted as **v**, to its own current value, and then, in the downstream phase, it sends the update value to every other replica in the system.

This more concrete demonstration of the two approaches shows the main differences between both of them, demonstrating that each approach has its own merits and disadvantages. It also serves as a first approach to the language that is used in all specifications throughout this document, which is explained in chapter 4.

2.2.3 CRDT Security

Based in its characteristics of high availability and scalability, CRDTs are used in several systems for which confidentiality may be of high concern. Although the study of CRDTs has been going on for more than a decade, there is very little formal treatment of CRDT security in literature. In [7], *Barbosa et al.* propose the first notion in this field, which supports the following data types: register, set, counter and bounded counter. It defines tailor-made examples of secure CRDT constructions for these data types, which must be carefully designed to use dedicated cryptographic techniques in order to perform computations over encrypted data.

This approach to CRDT security through encryption makes use of a layer of security between clients writing and reading data, in a way that sensitive data is encrypted before entering an untrusted network and must be decrypted when exiting the network, while being able to propagate and merge between replicas in an encoded state. Figure 2.5, taken from [7], captures the stated scenario. There is a setup phase where cryptographic keys are established by the clients, to be used in the aforementioned security layer. The presented model attributes no preference to either symmetric or asymmetric keys, as it is agnostic to this setup phase. Afterwards, an encryption operation precedes every update (step 1). The server receives the encrypted data, and performs the update (step 2), followed by several propagations and merges between replicas (step 3). This step is at the core of the proposed model, as it is specific to each CRDT construction. Querying a node for data is next (step 4), followed by a decryption of this data (step 5).

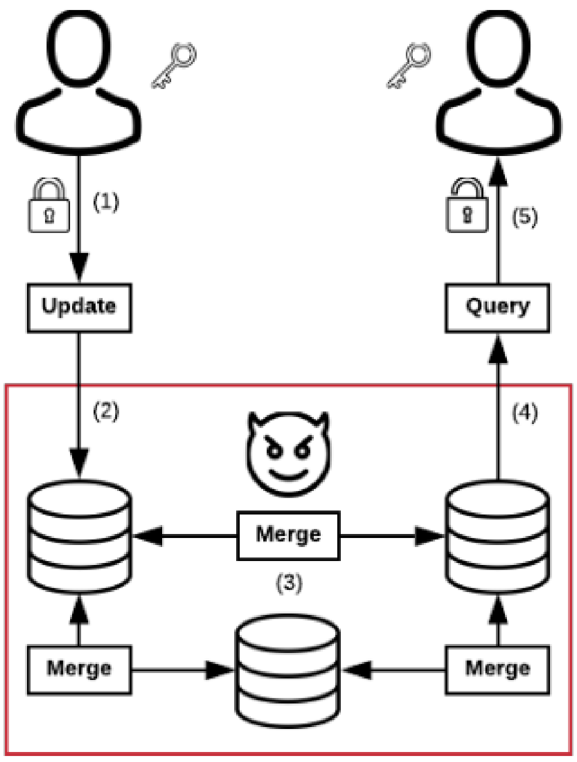


Figure 2.5: State-based CRDT, taken from [7]

In order to better understand this approach to CRDT security, let's consider the simple example of a register CRDT, a data structure holding a single value, which may be used as a building block for more complex data structures. An update replaces the current value and a query returns it. Merge operations for this CRDT are quite simple, as new data is in no way related to previous data, meaning that there is no computation over encrypted values. The update data is encrypted before being sent to a server, which simply substitutes its older value by the new, to then be returned as plaintext when a query occurs.

The cryptographic overhead of this construction is minimal, as it is only affected by a key generation progress, on encryption and one decryption. This is only the case because there is no computation to be done over encrypted data, as the consistency mechanism is simply based on metadata (the timestamps which allow the construction to decide on which data is new or old).

The set CRDT requires equality comparison over encrypted data, in order to understand if a certain value is either already in the set or is to be added. This is achieved with the use of a deterministic encryption scheme, which assures that equality comparison over encrypted values is as simple as it is over plaintext values, which maintains the low overhead seen in the previous case. Unfortunately, this is not the case for every CRDT construction: the counter CRDT, a numerical data structure which can be incremented and decremented, is proposed in this work to be implemented as an aggregate of two simple grow-only counters (one for increments and one for decrements). The model also uses a per-replica Lamport clock [24], stored in plaintext, to establish partial order of events, and thus reducing the computations over encrypted data to an addition. This is achieved with the use of additively homomorphic encryption scheme, which imposes a higher performance overhead than the two previous constructions.

Directly transposing each CRDT implementation to its secure variation is not possible based on this approach, as the computations that can be done over encrypted data are limited, and each different scheme must be custom designed. Furthermore, complex constructions were shown to incur in bigger latency without any growth in throughput, meaning that each operation took longer and no increase in operations per second was seen. In order to lift general purpose implementations of CRDTs to their secure variants, without limiting functionality, a possible approach is the use of secure Multi-Party Computation (MPC).

Chapter 3

State of the Art

This chapter's purpose is to provide an overview the current state of the art, mainly in relation to Conflict-free Replicated Data Types (CRDTs) and Multi-Party Computation (MPC). It is divided in two sections: the first one approaches the objective in a theoretic way, focusing on the literature, while the second section gives an overview on the current universe of practical tools and frameworks.

3.1 Related Work

This section succinctly describes the state of literature on several topics that are relevant to the present work:

- *Secure distributed storage:* The storage of confidential data in the cloud environment has been an extensive research topic, starting with the migration of in-premises storage to an encrypted storage infrastructure in a single cloud provider [28]. More cloud native solutions such as BlueSky [38] have emerged to provide strong consistency and availability. However, cloud systems can also be affected by loss of availability and data corruption. DepSky [9] overcame these limitations with a cloud-of-cloud system and a byzantine fault tolerance protocol to recover from a cloud failure. DepSky also uses secret sharing to split sensitive data over multiple parties. However, none of these systems provide neither confidential computation nor eventual consistency.
- *Eventual consistency and CRDTs:* Eventual consistency has long been a focal point in the research of highly-available and scalable asynchronous systems [32, 36]. *DeCandia et al.* [16] provide Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. In contrast, *Shapiro et al.* [34] study eventual consistency with a formal approach grounded on commutativity and semi lattices, following their separate technical report [33] which serves as a comprehensive portfolio of CRDT designs. *Balegas et al.* [6] present a prototype built on top of Riak

for extending eventually consistent cloud databases for enforcing numeric invariants, from where this document draws inspiration to build a maximum value CRDT. Riak is itself a key-value store inspired in Dynamo [16] which is built using Riak_core [21] as the DHT communication substrate.

- *SMPC:* Multi-party protocols have been used as a solution for private computation on several systems. SDB [39] is a relational database that uses a two-party protocol suite optimized fore relational queries. NoSQL operations can also be fully supported with an acceptable performance and security trade-off as shown by d'Artagnan [29], the first decentralized NoSQL confidential database. These protocols are also used in the commercial Sharemind platform, which provides secure data analysis [10]. More recently, SMCQL [8] and ConClave [37] have presented optimized systems for big data workloads in a honest-but-curious model and a three party setting. Senate [27] has gone one step further and presented a platform for collaborative analytics secure against an active adversary for any number of parties. Multi-party protocols have also extended into machine learning with Cerebro [40].
- *Secure CRDTs:* Secure CRDTs were first formalized by *Barbosa et al.* [7]. The authors also presented multiple constructions for secure registers, counter and set CRDTs that leverage deterministic encryption and partial homomorphic encryption schemes. *Cachin et al.* [14] have proposed Authenticated Data Types (ADTs) for authenticated data outsourcing in a singe-server/single-client setting. Snapdoc [22] presents a solution for collaborative document edition with history-privacy. This solution ensures that the join operation of two documents is authenticated and the privacy of a document's edition history is preserved. The work of *Shoker et al.* presents Byzec [35], a protocol designed to address the lack of CRDTs resistant to byzantine faults. However, only the work of *Barbosa et al.*, this document and the adjacent paper consider the problem of secure CRDTs.

3.2 Practical Solutions and Frameworks

3.2.1 For CRDTs

Several frameworks exist that leverage CRDTs in order to provide eventually consistent distributed systems with strong availability and fault-tolerance. In this subsection three relevant frameworks are presented, in order to provide an overview of the current state of the art when it comes to CRDT-based tools.

Automerge is a JavaScript library for data synchronization between mobile devices, which enables users to interact with data while offline and then when online merges changes even if made concurrently on different devices. Other similar services and applications are implemented by storing a main copy in a centralized server, and while some allow the user to interact with such data offline, others only work while online. The problem with this approach is that the centralized server may be located far away from the devices, resulting in high latency even if two devices in the system are besides each other. Automerge enables devices to swap data directly via Bluetooth, a local network or peer-to-peer networks through optional end-to-end encryption, and to synchronize the data it uses a JSON data model implemented as a CRDT. Although data may be exchanged in an encrypted state between devices, it must be decrypted before merging states, thus falling short of the goal of this work.

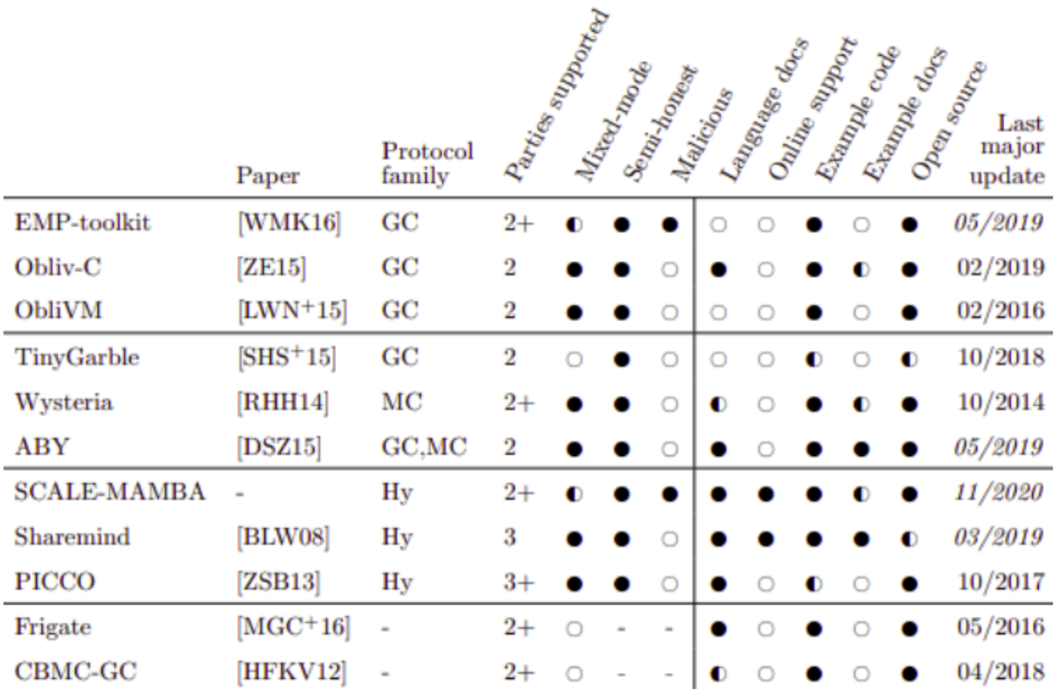
Yjs [26] is an open-source JavaScript implementation for peer-to-peer shared editing. In relation to automerge, it offers better flexibility in the form of support for mutable objects, as it was designed for rich text editing instead of fixed application states, and provides better performance for most operations. It achieves this extra functionality by using an internal linked list representation in the form of a CRDT and adding a garbage collector to reduce the number of necessary operations for synchronization. Also in contrast to Automerge, it uses an operation-based approach for CRDT propagation. As it provides no security features, it stands the case that developments from this work could potentially be used to enhance Yjs's functionality from a privacy-preserving standpoint.

AntidoteDB [1] is a highly-available geo-replicated NoSQL database that uses CRDTs as the data model. Similarly to both tools previously mentioned, it possesses no security mechanisms, being developed from a purely functional standpoint. *Barbosa et al.* used their secure CRDT constructions presented in [7] to implement a privacy-preserving version of AntidoteDB, that ultimately faced the constraints described in the previous section.

3.2.2 For MPC

Protocols for secure computation have existed for decades, and in recent years a series of compilers for executing multi-party computation on arbitrary functions have been developed. These projects are evolving and changing at such a fast pace that it isn't easy to be on par with the various capabilities of every framework. in an attempt to help solve this problem, *Hastings et al.* have surveyed several compilers for SMPC [18]. Their work considers eleven systems: EMP-toolkit, Obliv-C, ObliVM, TinyGarble, SCALE-MAMBA (formerly SPDZ), Wysteria, Sharemind, PICCO, ABY, Frigate and CBMC-GC. These systems are evaluated in language expressibility, capabilities of the cryptographic back-end and accessibility. They also provide a repository that contains a collection of sample programs for all these frameworks, set up in Docker containers.

Figure 3.1 is presented in this works as a summary of each framework's defining features and documentation types. In order to fulfill the needs for the present work's objectives, frameworks to consider should support 3 parties and provide (at a minimum) semi-honest security. Good documentation was also a highly prioritized feature, as would simplify and improve the quality of the work. Of all the these frameworks, SCALE-MAMBA and Sharemind deserve special consideration.



	Paper	Protocol family	Parties supported	Mixed-mode	Semi-honest	Malicious	Language docs	Online support	Example code	Example docs	Open source	Last major update
EMP-toolkit	[WMK16]	GC	2+	◐	●	●	○	○	●	○	●	*05/2019*
Obliv-C	[ZE15]	GC	2	●	●	○	●	○	●	◐	●	02/2019
ObliVM	[LWN+15]	GC	2	●	●	○	○	○	●	○	●	02/2016
TinyGarble	[SHS+15]	GC	2	○	●	○	○	○	◐	○	◐	10/2018
Wysteria	[RHH14]	MC	2+	●	●	○	◐	○	●	◐	●	10/2014
ABY	[DSZ15]	GC,MC	2	●	●	○	●	○	●	●	●	*05/2019*
SCALE-MAMBA	-	Hy	2+	◐	●	●	●	●	●	◐	●	*11/2020*
Sharemind	[BLW08]	Hy	3	●	●	○	●	●	●	●	◐	*03/2019*
PICCO	[ZSB13]	Hy	3+	●	●	○	●	○	◐	○	●	10/2017
Frigate	[MGC+16]	-	2+	○	-	-	●	○	●	○	●	05/2016
CBMC-GC	[HFKV12]	-	2+	○	-	-	◐	○	●	○	●	04/2018

Figure 3.1: Summary of MPC frameworks, taken from [18]

SCALE-MAMBA implements an hybrid protocol that is secure against malicious adversaries. It is composed of two distinct parts: MAMBA is a language built on top of Python that can be used to specify the MPC tasks, while SCALE implements the secure protocol. The framework has extensive documentation covering installation and running instructions, as well as more practical examples. Although it allows the definition of custom I/O classes, the framework provides malicious security through a secure channel that requires users to produce an authority certificate to run computation. This added layer of security inhibits the removal of shares in the midst of a computation, which would be required in order to share the state between replicas in a CRDT system. After a deep investigation into the framework's documentation and direct contact with its developers, it was discarded as changing its inner workings in order to accommodate the desired use case would entail a significant overhead in development, which we considered to be unjustified, given its alternatives.

Sharemind [10] is a framework for Secure Multi-Party Computation (SMPC) that uses additive secret sharing and is secure in the honest-but-curious adversary model (explained in chapter 2). Their traditional model, which is similar to the model used for each CRDT replica the present work (explained in chapter 4), consists of three server participants and a client application that orders computation, shares inputs and receives outputs. Although Sharemind is a commercial framework, its protocols are well documented and open-source and have been implemented by other projects, one of which is d'Artagnan [29].

d'Artagnan is an open-source multi-cloud NoSQL database that leverages these protocols to process queries. This database has two main components, *SafeClient* and *SafeServer*. The *SafeServer* is built on top of a high-level API of SMPC protocols (derived from Sharemind) that is designed to abstract the details of the protocol's implementation from the *SafeServer*, which enables the integration of new protocols with no relation to the concepts of a database but also the integration of the existing protocols in other projects, such as this one. The proper documentation of the protocols and extreme simplicity of the SMPC library allowed certain alterations to be made in order to facilitate the removal and introduction of shares from a state. As a result, this open-source library was the chosen tool for the multi-party computation tasks that would be necessary for the developed protocols.

Chapter 4

MPC-based Conflict-free Replicated Data Types

As a follow up, this section presents my contribution. It aims to detail and reason about the design of the proposed system, as well as developed specifications for the following CRDT construction families: registers, counters, maximum value and sets. I consider a semi-honest adversary model.

4.1 System design

My goal with this work is to propose and develop on the possibility to lift general-purpose implementations of CRDTs to their secure variants using general secure multi-party computation. As mentioned at the end of chapter 2, the idea that MPC and CRDTs are antagonistic if we map each CRDT replica to a MPC party is understandable. Nonetheless, the intention is that data is kept private while stored in the replicas and thus what is proposed is to map each CRDT replica to a group of MPC parties, which are leveraged for secure computation inside a single replica.

Being successful with this approach for a series of "building-block CRDTs" opens the door to the feasibility of achieving security for any other more complex CRDT that can use these basic constructions in its own construction. In order to achieve this goal, the designed system is modular and can be split into two parts: a CRDT-based network and a MPC service (figure 4.1).

The CRDT network is composed by multiple server replicas that routinely interact between each other to share their current state, and multiple clients that interact with a single replica at a time in order to perform updates or queries to the stored data. Each replica consists of three different parties that can and should be run in different machines so as to provide the intended level of security (if all three parties are run in the same machine and this is compromised, the adversary can easily retrieve all three shares that compose a secret and reveal its content). For a public cloud-based geo-replicated system, this would mean that CRDT replicas would be distributed around different locations of the globe, and for each location the respective replica could have its three CRDT parties running on different cloud providers (for example, replicas could have one party hosted on Google Cloud, one party hosted on Microsoft Azure and one party hosted on Amazon Web Services).

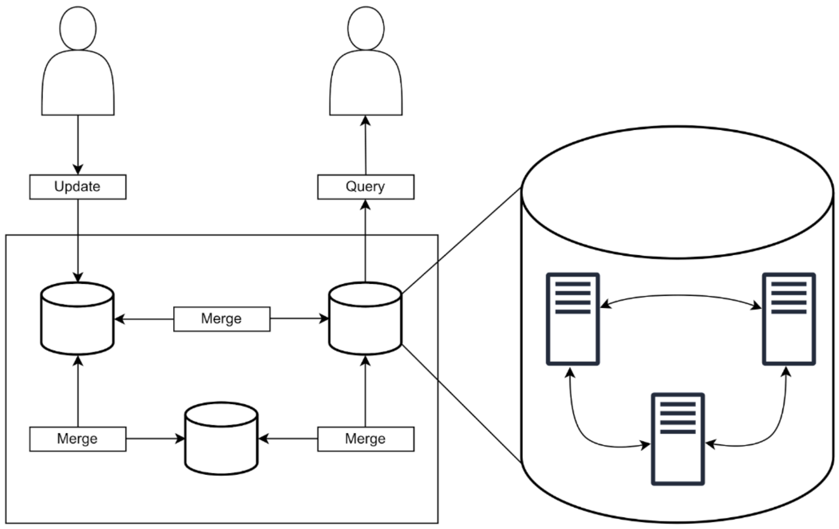


Figure 4.1: System overview

Clients would then choose one replica to connect to, a choice that can be based on several factors, mainly location proximity and latency, and perform the desired operation (update or query). For an update, the client is responsible for splitting the secret payload into three shares and sending each share to one of the replica's CRDT parties accompanied by all the necessary information to perform the update, such as operation identifiers, timestamps or other information specific to the CRDT construction. Clients are also responsible, on query operations, for receiving shares from each CRDT party and performing the necessary steps to recover the information to a readable state. Each replica would then be responsible for propagating their state to other replicas according to the implementation-specific constraints, and dealing with incoming propagations from other replicas. This happens as a direct mapping between parties of each replica (party 1 of replica A sends its state to party 1 of replica B, party 1 of replica A sends its state to party 2 of replica B and party 3 of replica A sends its state to party 3 of replica B).

Performing update and merge operations over secret shares may require CRDT parties to cooperate, which is done through the MPC service. The system may choose to have on MPC service per replica or one MPC service for all replicas. In order to maintain the high availability and low latency that is characteristic of CRDT systems, I chose to implement one MPC service per replica, which means that each CRDT party is accompanied by an MPC party. The CRDT parties communicate with their respective MPC parties through an interface, creating an abstraction layer that allows different MPC implementations to be used on a near plug-and-play basis. Figure 4.2 portrays an high-level view of the architecture of a single replica.

In order to guarantee freshness for all shares received by clients, when a query operation is performed the SMPC layer is used to create new shares for the secret information, overriding

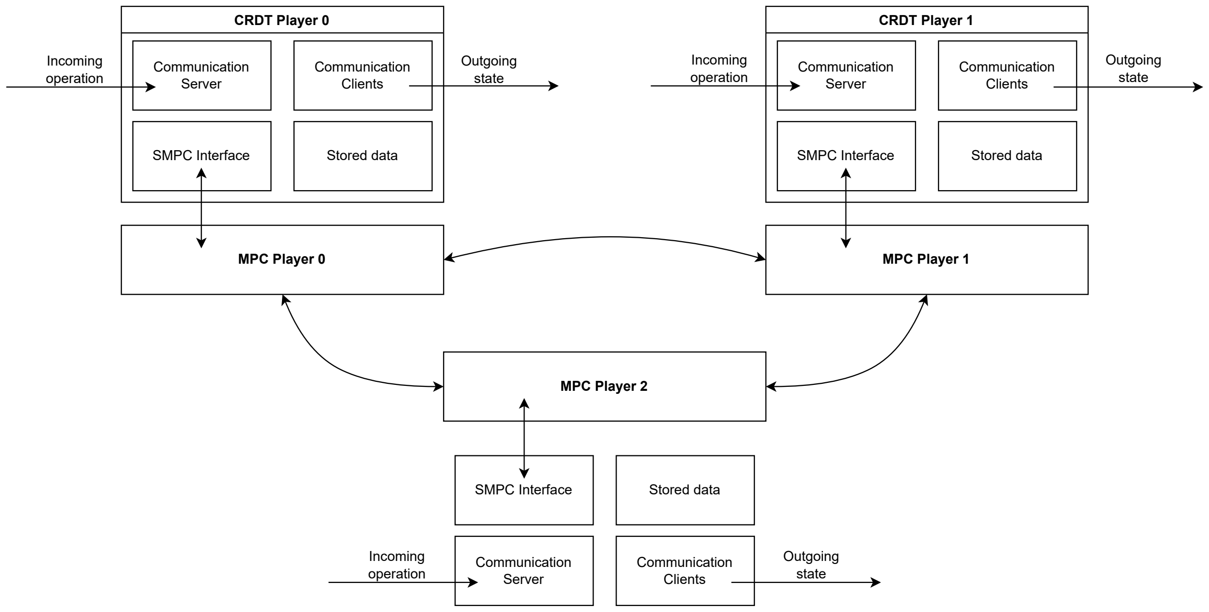


Figure 4.2: Replica architecture

existing data. This resharing process can also be set to happen periodically or at every propagation operation. This ensures that if two different CRDT parties of the same replica are compromised at different points in time, no secret information is divulged. Because this behavior is common to all the proposed CRDT constructions, this step is omitted from the presented specifications in order to simplify its reading.

4.2 Secure CRDT constructions

In alignment with the above-mentioned goal, CRDT constructions were designed in a interactive functional MPC language that makes an explicit syntactic distinction between public and secret data, and in which computations over private data are seen as calls to an underlying MPC system, while computations over public data are processed normally. For our concrete protocols, we assume that the MPC library supports operations for addition, multiplication, equality verification and greater-or-equal-than verification.

This chapter presents the designed CRDT constructions for a register, three different counters, a maximum value, and two different sets. This constructions detail the behavior of each and every party that composes a replica in a CRDT system, as all three parties have the same behavior. For each constructions I explain how it is often implemented as a normal, non-privacy-preserving CRDT, and then detail the designed specification for a secure version. Each specification is divided in 2 columns and comprised of three parts:

- The top side of the left column is where *custom data types* are defined. These types can be public integers (denoted by *Int*), secret integers (denoted by *SInt*), strings of characters (denoted by two quotation marks with the string content in between), combinations of these, or even arrays of combinations of these. As a quick example, specification 4's left column starts with the following:

$$S_r : \text{SInt, Int}$$

This line means that S_r is a custom type that is composed of a secret integer and a public integer. For another example, we can look at the second line from specification 6:

$$OP_u : \text{"Inc", SInt, Int} \mid\mid \text{"Dec", SInt, Int}$$

This means that OP_u is a custom type that is comprised of the string "Inc", a secret integer and a public integer OR comprised of the string "Dec", a secret integer and a public integer.

- The top side of the right column is where *function types* are declared in a functional manner. This section declares, for each function, what data types it takes and what is its output. Let's take, for example, the third line from the right column of specification 4:

  $$\textbf{query} :: \Pi \rightarrow OP_q \rightarrow S_r \rightarrow SInt$$

  This line declares a function called *query*, which takes as input a replica id (denoted throughout all specifications as Π), an object of the type OP_1, an object of the type S_r and returns as output a secret integer.

- Next is the pseudocode for the functions, spanning across both columns. In this section, variables that represent a secret value have a line on top, while variables that represent a public value do not (for example, $\overline{a}$ denotes a secret value while a would denote a public value). This section is explained in text along the rest of the chapter and any further detail that may not be so trivial will be mentioned in a comment at the end of the line, denoted as follows:

  $$\triangleright \text{ this is a comment}$$

4.2.1 Register

The first construction is for a Last-Writer-Wins (LWW) Register, with specification 4. Registers are data structures that maintain a general opaque value, being considered a container CRDT, and thus can be directly made to hold secure elements. Its operations perform no secure computations, because their behavior is agnostic to secret values, so the secure version of this CRDT is extremely similar to the original proposal by *Shapiro et al.* [33]. The one change has to do with the timestamps, where instead of using a *now*() operation to get the current timestamp, I established that the client is expected to generate it and present it as an argument (as the public integer that is part of OP_u).

The *new* function initializes the CRDT with secret shares that represent the value 0 and also the value 0 as timestamp. The *update* function takes an OP_u object that includes the secret shares for the new value and a public timestamp; if the incoming timestamp is more recent than the stored timestamp then the secret shares are updated. The *query* function simply returns the stored secret shares. The function *propagate* returns the state of all the parties. The function *merge* compares the timestamp of both the incoming state and the stored state, and keeps the secret shares and timestamp that correspond to the more recent timestamp of the two.

When a party is compromised, the only leaked information is the most recent timestamp.

Specification 4 Register CRDT

```
S_r : SInt, Int
OP_u : "Upd", SInt, Int
OP_q : "Get"

new :: $\Pi \rightarrow S_r$
update :: $\Pi \rightarrow OP_u \rightarrow S_r \rightarrow S_r$
query :: $\Pi \rightarrow OP_q \rightarrow S_r \rightarrow SInt$
propagate :: $(\Pi, \Pi) \rightarrow S_r \rightarrow Msg_r$
merge :: $(\Pi, \Pi) \rightarrow S_r \rightarrow Msg_r \rightarrow S_r$

new (id)                                  ▷ id: replica number
    $\overline{s} \leftarrow new\ SInt(0)$
    $A \leftarrow (\overline{s}, 0)$
    $return\ A$

update (id, op, A)
    $case\ (op = "Upd", \overline{v}, t')$
        $(\cdot, t) \leftarrow A$
        $if\ (t' > t)$
            $A \leftarrow (\overline{v}, t')$
    $return\ A$

query (id, op, A)
    $case\ (op = "Get")$
        $(\overline{s}, \cdot) \leftarrow A$
    $return\ \overline{s}$

propagate ((id_i, id_j), A)
    $return\ A$

merge ((id_i, id_j), A, A')
    $(\overline{s}, t) \leftarrow A$
    $(\overline{s}', t') \leftarrow A'$
    $if\ (t' > t)$
        $A \leftarrow (\overline{s}', t')$
    $return\ A$
```

4.2.2 Grow-only Counter

Specification 5 portrays the secure construction for a grow-only counter CRDT. A grow-only counter is a replicated integer that supports the operation to increment by an arbitrary integer. I start with this limited-functionality counter in order to simplify the approach, as it can be used as a basis for the next counter, a pn-counter. *Shapiro et al.* follow the same approach in [33] when presenting their baseline non-secure constructions for a counter CRDT. Their proposed state-based construction's payload is a vector of integers, in which each replica is assigned a position. The assumption that the number of replicas is known is quite acceptable. To increment by an arbitrary integer *a* they add *a* to the position that corresponds to the local replica that is performing the operation. Their implementation requires comparison over counter values in order to check the max value to merge two states, which is an expensive secure operation because it requires communication between parties.Given the monotonic nature of the grow-only counter, I avoid this secure operation by extending the state to include a per-replica public timestamp, which works similarly to a Lamport clock. This guarantees that a greater timestamp corresponds to a greater counter value.

The *new* function initializes the counter by creating an array of pairs (secret integer, public integer) with the same size as the number of replicas in the system, where each position contains the secret share of the value for all increments done by that replica (initialized as secret shares of the value 0) and a the timestamp of the most recent increment done by said replica (initialized as 0). The *update* function takes a secret share and a timestamp in the object OP_u; it adds the incoming secret share to the existing share in the correct position of the array, and substitutes the old timestamp with the incoming one. A *query* function iterates through the array and returns the sum of all secret shares. The function *propagate* simply returns all the state of the parties. The $mergefunction$ makes use of the timestamps in order to avoid the need for communication with other parties; it iterates through the array, and for each position it stores the pair (secret share, timestamp) with the most recent timestamp.

Leakage, or the information accessible to an attacker who might have compromised one of the parties, can be summarized as the number of update operations performed at each replica.

4.2.3 PN-Counter

Supporting decrements in a counter with the previous representation is not as straightforward as one would expect, but *Shapiro et al.* [33] provide an approach that is followed here. A counter capable of both increments and decrements no longer displays a monotonic behavior, and thus both the comparison used in [33] and the timestamps used by my design are no longer effective. In order to circumvent this, the solution is to build a pn-counter (positive/negative counter) combining two grow-only counters, where one is used to store increments, and the other is used to store decrements. The value of the pn-counter corresponds then to the difference between the two grow-only counters.

This construction is detailed in specification 6. Most pn-counter functions contain calls to the grow-only counter functions mentioned above. The *new* function initializes two grow-only counters. The *update* function now must check the string that comes in OP_u, as there are now two possible values: "Inc" for an increment operation and "Dec" for a decrement operation; according to this value an update function is called on the corresponding grow-only counter. The function *query* must perform the function of same name for each of the grow-only counters and return the difference between their resulting secret shares. The function *propagate* simply returns all the state of the parties. A *merge* function retrieves both grow-only counters from the incoming state and performs a *merge* function on both the grow-only counters stored, using the respective incoming state.

Leakage, similarly to the grow-only counter, can be summarized as the number of update

Specification 5 Grow-only Counter CRDT

```
$S_{gc}$ : [(SInt, Int)]
$OP_u$ : "Inc", SInt, Int
$OP_q$ : "Get"
Static $n$ : Int

new :: $\Pi \rightarrow S_{gc}$
update :: $\Pi \rightarrow OP_u \rightarrow S_{gc} \rightarrow S_{gc}$
query :: $\Pi \rightarrow OP_q \rightarrow S_{gc} \rightarrow SInt$
propagate :: $(\Pi, \Pi) \rightarrow S_{gc} \rightarrow Msg_{gc}$
merge :: $(\Pi, \Pi) \rightarrow S_{gc} \rightarrow Msg_{gc} \rightarrow S_{gc}$

new (id)                                  ▷ id: replica number
    $S \leftarrow [\,]$
    $for\ (i\ in\ [0..n-1])$
        $\overline{s_i} \leftarrow new\ SInt(0)$
        $A[i] \leftarrow \overline{s_i}, 0$
    $return\ A$

update (id, op, A)
    $case\ (op = "Inc", \overline{v}, t')$
        $(\overline{s_i}, t) \leftarrow A[id]$
        $A[id] \leftarrow (\overline{s_i} + \overline{v}, t')$
    $return\ A$

query (id, op A)
    $case\ (op = "Get")$
        $for\ (i\ in\ [0..n-1])$
            $(\overline{s_i}, \cdot) \leftarrow A[i]$
            $\overline{s} \leftarrow \overline{s_i} + \overline{s}$
    $return\ \overline{s}$

propagate ((id$_i$, id$_j$), A)
    $return\ A$

merge ((id$_i$, id$_j$), A, A')
    $for\ (i\ in\ [0..n-1])$
        $(\overline{s_i}, t) \leftarrow A[i]$
        $(\overline{s_i}', t') \leftarrow A'[i]$
        $if\ (t' > t)$
            $A[i] \leftarrow (\overline{s}', t')$
    $return\ A$
```

operations performed at each replica.

4.2.4 Maximum Value

Specification 7 for a CRDT that stores a maximum value. It is similar to a register CRDT, but the value it holds should be updated if and only if the new value is greater than the one stored, regardless of how recent either one is. A non-privacy-preserving construction like this is trivial to implement: there must only be one comparison between the old and new values.

Lifting such construction to a privacy-preserving one is not as simple. Although both previously mentioned counter constructions make use of timestamps as a tactic to avoid comparison over secret values, this is not always possible, as is the case in this CRDT. Calculating the maximum between two secret values requires communication between parties, which makes it an expectedly slower operation in relation to the the ones on previous constructions.

I propose performing this computation arithmetically combining multiplication, equality comparison and greater-or-equal-than comparison over secret shares. Each of these is done as a

Specification 6 PN Counter CRDT

```
$\boldsymbol{S_{pnc}}$ : Sgc, Sgc                              new :: $\Pi \to S_{pnc}$
$\boldsymbol{OP_u}$ : "Inc", SInt, Int || "Dec", SInt, Int     update :: $\Pi \to OP_u \to S_{pnc} \to S_{pnc}$
$\boldsymbol{OP_q}$ : "Get"                                    query :: $\Pi \to OP_q \to S_{pnc} \to SInt$
Static $\boldsymbol{n}$ : Int                                  propagate :: $(\Pi, \Pi) \to S_{pnc} \to Msg_{pnc}$
                                                               merge :: $(\Pi, \Pi) \to S_{pnc} \to Msg_{pnc} \to S_{pnc}$

new (id)                          ▷ id: replica number
    $A \leftarrow new\ S_{GC}(id), new\ S_{GC}(id)$
    return A
update (id, op, A)
    $(P, N) \leftarrow A$
    case $(op = "Inc", \overline{v}, t')$
        $P \leftarrow P.update(id, ("Inc", \overline{v}, t'), P)$
    case $(op = "Dec", \overline{v}, t')$
        $N \leftarrow N.update(id, ("Inc", \overline{v}, t'), N)$
    $A \leftarrow (P, N)$
    return A
query (id, op A)
    case $(op = "Get")$
        $(P, N) \leftarrow A$
        $\overline{R_p} \leftarrow P.query(id, "Get", P)$
        $\overline{R_n} \leftarrow N.query(id, "Get", N)$
    return $\overline{R_p} - \overline{R_n}$

propagate ((id_i, id_j), A)
    return A

merge ((id_i, id_j), A, A')
    $(P, N) \leftarrow A$
    $(P', N') \leftarrow A'$
    $P \leftarrow P.merge((id_i, id_j), P, P')$
    $N \leftarrow N.merge((id_i, id_j), N, N')$
    $A \leftarrow (P, N)$
    return A
```

call to the underlying Secure Multi-Party Computation (SMPC) system. In order to calculate the maximum between two values a and b using the above mentioned operations, the formula is as follows:

$$max(a, b) \leftarrow a \times (a \geq b) + b \times (b \geq a) - a \times (a = b)$$

The *new* function initializes the CRDT state with secret shares corresponding to the value 0. An *update* performs the above-mentioned formula in order to keep in store the secret share that corresponds to the maximum value. The *query* function returns the secret share currently stored in the CRDT. The function *propagate*, simply returns the current state of each party. Performing a *merge* function invokes an *update* function, as merging two states with a single secret share is the same as simply updating one state with a new value (the relevant content of both an *update* and a *merge* payloads is the same).

This CRDT construction for a maximum value has no leakage, as only one secret share is kept at all times and its value changes on every single *update* and *merge* (even if the value of the secret remains the same, its shares change).

Specification 7 Maximum Value CRDT

```
$\boldsymbol{S_{mv}}$ : SInt
$\boldsymbol{OP_u}$ : "Upd", SInt
$\boldsymbol{OP_q}$ : "Get"

new :: $\Pi \to S_{mv}$
update :: $\Pi \to OP_u \to S_{mv} \to S_{mv}$
query :: $\Pi \to OP_q \to S_{mv} \to SInt$
propagate :: $(\Pi, \Pi) \to S_{mv} \to Msg_{mv}$
merge :: $(\Pi, \Pi) \to S_{mv} \to Msg_{mv} \to S_{mv}$

new (id)                                ▷ id: replica number
    $\overline{s} \leftarrow new\ SInt(0)$
    $A \leftarrow \overline{s}$
    $return\ A$

update (id, op, A)
    $case\ (op = "Upd", \overline{v})$
        $\overline{s} \leftarrow A$
        $A \leftarrow \overline{s} \times (\overline{s} \geq \overline{v}) + \overline{v} \times (\overline{v} \geq \overline{s})$
            $- \overline{s} \times (\overline{v} = \overline{s})$
    $return\ A$

query (id, op, A)
    $case\ (op = "Get")$
        $\overline{s} \leftarrow A$
    $return\ \overline{s}$

propagate ((id_i, id_j), A)
    $return\ A$

merge ((id_i, id_j), A, A')
    $\overline{u}' \leftarrow A'$
    $A \leftarrow A.update(id_i, ("Upd", \overline{u}), A)$
    $return\ A$
```

4.2.5 Bounded Counter

It is notoriously difficult to preserve invariants while avoiding synchronization. The original literature on CRDTs [33] does not present a proper construction for a baseline non-privacy-preserving bounded counter, but it proposes the possible approach of enforcing a local invariant that implies the global invariant (i.e., a replica can not generate more decrements than it has generated increments). This approach is too strong, as it limits functionality that should be present (a replica should be able to generate decrements if enough increments have been generated by other replicas).

Balegas et al. show how to extend an eventually consistent cloud database for enforcing numeric invariants [6] by using a system of rights that manage when a certain operation can be executed; these rights may be transfered between replicas. They present a baseline, non-privacy-preserving state-based construction for a bounded counter CRDT that can be incremented and decremented while maintaining the invariant *greater or equal to k*. The state maintains the limit value k and information about the rights each replica has. Considering a system with n replicas, the state maintains a matrix R where $R[i][j]$ holds the rights transferred from replica i to replica j (which is a registry of all increment operations executed) and a vector U where $U[i]$ holds the rights consumed by replica i (which is a registry of all decrement operations executed). Lifting this construction to a privacy-preserving alternative, shown in specification 8, means to simply represent every value (the limit k, every value in the matrix R and every value in the vector U) as a pair (secret value, public value) that contains the secret share of the value and a public timestamp of the last operation that updated the corresponding value. Similarly to the grow-only counter, tiestamps are used in merge operations to maintain the most recent data, in order to avoid unnecessary comparisons, which are computationally expensive.

The *new* function accepts a secret share that represents the limit value k and initializes both the matrix R and the vector U with secret shares that represent the value 0 and timestamps also set to 0. the *update* function supports three different operations:

- an *increment* gets the position of the matrix R corresponding to the local replica ($R[id][id]$), adds the incoming share of the value to increment to the existing share of previous increments and also increments the local timestamp for that position by 1.
- a *decrement* must first check if there are enough local rights to act, which is done through the function *localRights* detailed soon; this verification, which is a comparison between the number of existing rights and the incoming value to decrement, is done through the same formula as the comparison in the maximum value CRDT - specification 7 - (in this case, a and b correspond to the number of available rights and the sum between the number of incoming decrements and the number of existing local decrements stored in $U[id]$).
- a *transfer* operation makes a similar verification to the decrement operation but applied to $R[id][to]$ instead of $U[id]$

The *query* operation iterates through the matrix R in order to get all the increments and through the vector U in order to get all the decrements; then it returns the current value of the replica which corresponds to the limit value added to the increments and subtracted the decrements A *propagate* function returns all the state of the replica. A *merge* function iterates through both the matrix R and the vector U and for each entry it keeps the one that has the most recent timestamp, be it the incoming state or the existing state. The additional function *localRights* iterates through the matrix R adding both the received rights (stored in $R[i][id]$, where i corresponds to the iteration lap) and the rights sent to other replicas (stored in $R[id][i]$); it then gets the number of available rights by adding the number of local increments to the number of received rights, subtracting the number of sent rights and subtracting the number of locally consumed rights.

This construction may be reversed as a bounded counter with an upper limit by using the matrix R to store decrements and the vector U to store increments. Similarly to creating a PN counter from two grow-only counters, it can also be used for a bounded counter with upper and lower limit by combining two of these bounded counters.

Specification 8 Minimum Bounded Counter CRDT

```
$S_{bc}$ : SInt, [[(SInt, Int)]], [(SInt, Int)]
$OP_u$ : "Inc", SInt || "Dec", SInt ||
        "Transfer", SInt, Int
$OP_q$ : "Get"
Static $n$ : Int

new :: $\Pi \to SInt \to S_{bc}$
update :: $\Pi \to OP_u \to S_{bc} \to S_{bc}$
query :: $\Pi \to OP_q \to S_{bc} \to SInt$
propagate :: $(\Pi, \Pi) \to S_{bc} \to Msg_{bc}$
merge :: $(\Pi, \Pi) \to S_{bc} \to Msg_{bc} \to S_{bc}$
localRights :: $\Pi \to S_{bc} \to SInt$

new (id, $\overline{k}$)                                ▷ id: replica number
    $R \leftarrow new\ (SInt, Int)[n][n]$
    $U \leftarrow new\ (SInt, Int)[n]$
    $for\ (i\ in\ [0..n-1])$
        $for\ (j\ in\ [0..n-1])$
            $\overline{v} \leftarrow new\ SInt(0)$
            $R[i][j] \leftarrow (\overline{v}, 0)$
        $U[i] \leftarrow (\overline{v}, 0)$
    $A \leftarrow (\overline{k}, R, U)$
    $return\ A$

update (id, op, A)
    $(\overline{k}, R, U) \leftarrow A$
    $case\ (op = "Inc", \overline{v})$
        $(\overline{s}, t) \leftarrow R[id][id]$
        $R[id][id] \leftarrow (\overline{s} + \overline{v}, t + 1)$
    $case\ (op = "Dec", \overline{v})$
        $(\overline{s}, t) \leftarrow U[id]$
        $rights \leftarrow localRights(id, A)$
        $\overline{s}' \leftarrow \overline{s} \times (rights < \overline{v}) + (\overline{s} + \overline{v}) \times$
            $(\overline{v} < rights) + (\overline{s} + \overline{v}) \times$
            $(\overline{v} = rights)$
        $U[id] \leftarrow (\overline{s}', t + 1)$
    $case\ (op = "Transfer", \overline{v}, to)$
        $(\overline{s}, t) \leftarrow R[id][to]$
        $rights \leftarrow localRights(id, A)$
        $\overline{s}' \leftarrow \overline{s} \times (rights < \overline{v}) + (\overline{s} + \overline{v}) \times$
            $(\overline{v} < rights) + (\overline{s} + \overline{v}) \times$
            $(\overline{v} = rights)$
        $R[id][to] \leftarrow (\overline{s}', t + 1)$
    $A \leftarrow (\overline{k}, R, U)$
    $return\ A$

query (id, op, A)
    $case\ (op = "Get")$
        $for\ (i\ in\ [0..n-1])$
            $(\overline{r_i}, \cdot) \leftarrow A.R[i][i]$
            $(\overline{u_i}, \cdot) \leftarrow A.U[i]$
            $\overline{r} \leftarrow \overline{r_i} + \overline{r}$
            $\overline{u} \leftarrow \overline{u_i} + \overline{u}$
        $\overline{s} \leftarrow A.\overline{k} + \overline{r} - \overline{u}$
        $return\ \overline{s}$
propagate ((id$_i$, id$_j$), A)
    $return\ A$
merge ((id$_i$, id$_j$), A, A')
    $(\overline{k}, R, U) \leftarrow A$
    $(\overline{k}', R', U') \leftarrow A'$
    $for\ (i\ in\ [0..n-1])$
        $for\ (j\ in\ [0..n-1])$
            $(\overline{r}, t) \leftarrow R[i][j]$
            $(\overline{r}', t)' \leftarrow R'[i][j]$
            $if\ (t' > t)$
                $R[i][j] \leftarrow (\overline{r}', t')$
        $(\overline{u}, t) \leftarrow U[i]$
        $(\overline{u}', t') \leftarrow U'[i]$
        $if\ (t' > t)$
            $U[i] \leftarrow (\overline{u}', t')$
    $A \leftarrow (\overline{k}, R, U)$
    $return\ A$
localRights (id, A)
    $for\ (i\ in\ [0..n-1])$
        $(\overline{r_1}, \cdot) \leftarrow A.R[i][id]$
        $(\overline{r_2}, \cdot) \leftarrow A.R[id][i]$
        $\overleftarrow{\overline{r_1}}' \leftarrow \overleftarrow{\overline{r_1}} + \overleftarrow{\overline{r_1}}'$
        $\overleftarrow{\overline{r_2}}' \leftarrow \overleftarrow{\overline{r_2}} + \overleftarrow{\overline{r_2}}'$
    $(\overline{r}, \cdot) \leftarrow A.R[id][id]$
    $(\overline{u}, \cdot) \leftarrow A.U[id]$
    $\overline{v} \leftarrow \overline{r} + \overline{r_1}' - \overline{r_2}' - \overline{u}$
    $return\ \overline{v}$
```

4.2.6 Set

Sets are one of the most basic data structures, which usually serve as building blocks for more complex structures. Although basic, these structures can not be fully implemented as a CRDT, because their two mutating operations (add and remove) do not commute. This means that a CRDT can only try to behave close to a set. The simplest way to approximate the behavior of a CRDT to a set is to consider only a grow-only set. *Shapiro et al.* [33] proposes a construction that supports *add* and *lookup* operations. Its payload is a set, thus operations are similar to any mundane set outside of the literature on CRDTs: performing an *add* operation is to calculate the union between two sets; performing a *lookup* for an element returns true if the element is in the set and false otherwise.

I propose two different constructions for a grow-only set, both of which support the *add* operation for updates and *exists* or *getall* operations for queries but deal with the *add* operation in different ways: the first construction always adds an element even if it exists, while the second constructions only adds an element if it does not exist. For simplicity, I encode the set's payload as a list, but order is not preserved.

Specification 9 represents the first construction for a grow-only set. In order to distinguish both constructions, I call this one "Ever-growing set". The *new* function initializes an empty list of secret shares. The *update* function, as discussed, always appends the incoming element to the list. The *query* function, distinctively from all previous constructions, supports two very different operations. *getall* returns all the elements in the list (which is the same as the *propagate* function) *exists* takes a secret share as input and iterates through the list, calculating the sum of the equalities between the element to lookup and the element on each entry of the list; this will result in the shares of the value 1 if the element is in the list and 0 otherwise. Performing a *merge* operation is iterating through every element of the incoming set and calling the *update* function for each.

The second specification for a grow-only set - specification 10 - differs from the first one on the way it deals with *add* operations. Instead of always adding new elements to the set CRDT, this specification makes use of a SMPC operation that I call "unshare_internally". Every function other than *update* is equal the previous specification, so I refrain from explaining them again. For an *update*, this specification starts by iterating over all elements on the list and calculating a sum of the equalities between the element to add and the elements in each entry of the list (similarly to the *query* operation *exists*); then, the SMPC operation "unshare_internally", which takes a secret value as input, decodes the value obtained with the sum so that every player knows if the incoming element is already in the list or not; if the element is not in the list, then it is appended.

While the first approach can become inefficient quite quickly, with new elements always being added and passing the duty of checking for repeats to the client, it provides no leakage. On the other hand, the second approach leaks information when any value is added, as the list only grows if the set was not previously in the list.

With the overall system design explained and all the CRDT specifications presented, the next chapter relates to the implementation of this system and subsequent experimental validation.

Specification 9 Ever-growing Set CRDT

```
$\boldsymbol{S_{egs}}$ : List <SInt>
$\boldsymbol{OP_u}$ : "Add", SInt
$\boldsymbol{OP_q}$ : "Getall" || "Exists", SInt
$\boldsymbol{R_{egs}}$ : $S_{egs}$ || SInt

new :: $\Pi \to S_{egs}$
update :: $\Pi \to OP_u \to S_{egs} \to S_{egss}$
query :: $\Pi \to OP_q \to S_{egs} \to R_{egs}$
propagate :: $(\Pi, \Pi) \to S_{egs} \to Msg_{egs}$
merge :: $(\Pi, \Pi) \to S_{egs} \to Msg_{egs} \to S_{egs}$

new (id)                              ▷ id: replica number
    $l \leftarrow new\ List < SInt >$
    $A \leftarrow l$
    $return\ A$

update (id, op, A)
    $case\ (op = "Add", \overline{v})$
        $l \leftarrow A$
        $l.append(\overline{v})$
        $A \leftarrow l$
    $return\ A$

query (id, op, A)
    $case\ (op = "Getall")$
        $return\ A$
    $case\ (op = "Exists", \overline{v})$
        $l \leftarrow A$
        $\overline{u} \leftarrow new\ SInt(0)$
        $for\ (\overline{s}\ in\ l)$
            $\overline{u} \leftarrow \overline{u} + (\overline{s} = \overline{v})$
        $return\ \overline{u}$

propagate ((id_i, id_j), A)
    $return\ A$

merge ((id_i, id_j), A, A')
    $l' \leftarrow A'$
    $for\ (\overline{v}\ in\ l')$
        $A \leftarrow A.update(id_i, ("Add", \overline{v}), A)$
    $return\ A$
```

Specification 10 Leakage Set CRDT

```
$S_{ls}$ : List <SInt>
$OP_u$ : "Add", SInt
$OP_q$ : "Getall" || "Exists", SInt
$R_{ls}$ : $S_{ls}$ || SInt
```

```
new :: $\Pi \rightarrow S_{ls}$
update :: $\Pi \rightarrow OP_u \rightarrow S_{ls} \rightarrow S_{lss}$
query :: $\Pi \rightarrow OP_q \rightarrow S_{ls} \rightarrow R_{ls}$
propagate :: $(\Pi, \Pi) \rightarrow S_{ls} \rightarrow Msg_{ls}$
merge :: $(\Pi, \Pi) \rightarrow S_{ls} \rightarrow Msg_{ls} \rightarrow S_{ls}$
```

```
new (id)                                ▷ id: replica number
    $l \leftarrow new\ List < SInt >$
    $A \leftarrow l$
    $return\ A$

update (id, op, A)
    $case\ (op = "Add", \overline{v})$
        $l \leftarrow A,\ \overline{u} \leftarrow new\ SInt(0)$
        $for\ (\overline{s}\ in\ l)$
            $\overline{u} \leftarrow \overline{u} + (\overline{s} = \overline{v})$
        $res \leftarrow unshare\_internally(id,\ \overline{u})$
        $if\ (res = 0)$
            $l.append(\overline{v})$
        $A \leftarrow l$
    $return\ A$

query (id, op, A)
    $case\ (op = "Getall")$
        $return\ A$
    $case\ (op = "Exists", \overline{v})$
        $l \leftarrow A,\ \overline{u} \leftarrow new\ SInt(0)$
        $for\ (\overline{s}\ in\ l)$
            $\overline{u} \leftarrow \overline{u} + (\overline{s} = \overline{v})$
        $return\ \overline{u}$
```

```
propagate (($id_i$, $id_j$), A)
    $return\ A$

merge (($id_i$, $id_j$), A, A')
    $l' \leftarrow A'$
    $for\ (\overline{v}\ in\ l')$
        $A \leftarrow A.update(id_i, ("Add", \overline{v}), A)$
    $return\ A$
```

Chapter 5

Implementation and Experimental Validation

This section presents the implementation of the system detailed in section 4 and conduct an experimental evaluation on the proposed constructions that demonstrates the following:

- Conflict-free Replicated Data Type (CRDT) constructions can be lifted to their privacy-preserving variants efficiently;
- Secure Multi-Party Computation (SMPC) protocols are an efficient alternative to classical privacy-preserving schemes schemes, frequently used in confidential databases;
- Secure CRDTs are inter-operable and can be reused as building blocks for other constructions with minimal performance overhead.

5.1 System Implementation

The privacy-preserving CRDT system has been implemented as an open-source Java 8 project. It is composed of two main parts: the client and replica.

In order to simplify testing and work within the constraints created by the available resources (hardware and time), the client has been implemented so that it can simulate the behavior of the CRDT network. This means that the client is not only responsible for generating payloads to send to the replica for *update* operations and requesting *query* operations, but it also manages the *propagate* and *merge* operations.

The replica can be divided in two conceptual objects:

- a CRDT Player that handles the operation requests sent from the client and follows the CRDT constructions defined in chapter 4;

- a SMPC Player that supports several multi-party protocols over secret shares and exchanges secrets with the two other SMPC Players of the replica.

My implementation is agnostic to both the underlying network topology as well as to the SMPC protocols. For the purpose of experimental evaluation, the network is implemented as a simple mesh network using TCP channels and the chosen SMPC framework is the *d'Artagnan* library, an implementation of the Sharemind protocols written in Java. These protocols are based on an additive secret sharing scheme and are optimized for a static three-party setting with an honest-but curious adversary. As such, each CRDT replica is composed by three parties (three CRDT Players and three SMPC players, one of each per party), each of one storing one of the three sets of shares that compose the value stored in the replica.

5.2 Experimental Setup

The evaluation was conducted on a cluster of 4 machines, each with two Intel(R) Xeon(R) Silver 4210 CPU processores clocked at 2.20GHz and 265GB of RAM. The cluster consisted of a single client and a single CRDT replica, composed of three independent parties. These 4 processes communicated between machines using TCP. The network has an average latency of 200 μs between hosts

In order to perform load testing on the system I used Locust [2], an easy to use, scriptable and scalable python framework. It uses simple Python code to define tasks, that are then executed by a defined number of users. It also provides a user friendly web interface that shows the progress of tests in real time, although this feature is not necessary for this work's sue case. Locust interacts with applications either through HTTP or RPC protocols, meaning that a custom HTTP interface had to be developed for the system. This tool was used with custom CRDT workloads, which were made public and are explained in the following section.

5.2.1 Procedure and Workloads

For the experimental evaluation I measured the latency and throughput of all CRDT operations defined in chapter 4 for all privacy-preserving CRDT constructions. The goal of the workloads is to measure the performance of each operation in isolation and clearly understand the impact of every different approach. As such, the workload's requests differ based on the CRDT data structure that they correspond to. I do not measure the performance of either *propagate* and *merger* operations, as the former requires absolutely no computations (neither private nor public) and the latter should have similar overhead to the *update* operation across all constructions.

- *Register and Maximum Value CRDTs:* the workloads start by initializing the replica with a default random value. The evaluation of the *update* operation consists of the insertion of a series of random values during the benchmark. The evaluation of the *query* operation is

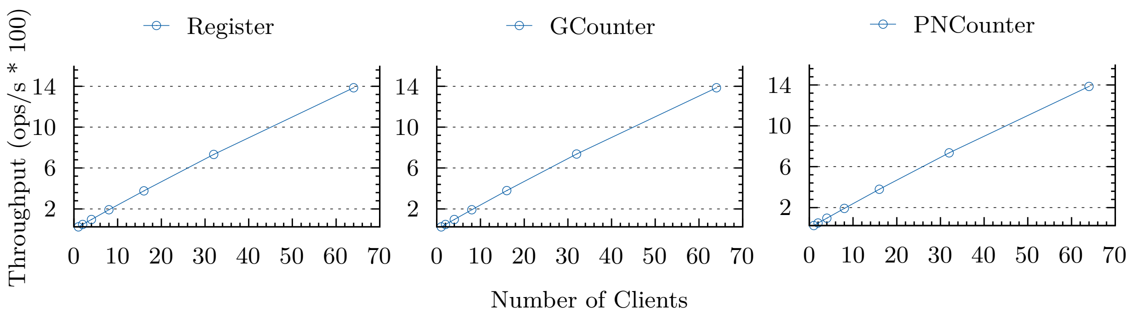


Figure 5.1: Average throughput of the update operation for the secure Register, GCounter and PNCounter CRDT.

straightforward, consisting of a series of requests from the client for the most recent value stored in the replica. As these CRDT constructions do not use multi-party computation protocols, it is expected that there is no significant performance overhead.

- *Counter CRDTs:* i implemented three workloads, one for each counter CRDT: the grow-only counter, the pn-counter and the bounded counter. The workloads are mostly similar for all operations across the three CRDTs. They all start by initializing the replica with a counter set to 1 and for each *update* operation increment the counter by 1. The pn-counter and bounded counter workloads differ slightly from the grow-only counter as they can also decrement the counter. Additionally, each of these operations uses multi-party protocols differently and as such have different expected performance overheads. To accurately measure these differences, each operation is measured in isolation and the *decrement* operation is measured by initially setting the counter to its highest possible value and then decrementing by 1 per request. The evaluation of the *query* operation is straightforward, consisting of a series of requests from the client for the most recent value stored in the replica.

- *Set CRDTs:* the performance of CRDT operations on sets depends on the set size: as the number of elements in a set increases, more multi-party computation protocols must be executed. As such, while the workload for the *update* operation simply consists on starting with an empty set and for each operation adding a random value, the *query* operation differs. For this, I measure its overhead for different set sizes in an exponential scale, from 2^3 to 2^6 elements.

I have also implemented a baseline system for each CRDT evaluated. The baseline system follows the state-of-the art definitions for CRDTs over plaintext, that were briefly mentioned in 4, without any secure multi-party protocols.

CRDT	Operation	Baseline	SMPC
Register	Query	1387	1376
	Update	1388	1385
MaxValue	Query	1388	1360
	Update	1388	9
GCounter	Query	1388	1365
	Update	1377	1385
PNCounter	Query	1388	1363
	Update	1387	1385
Bounded Counter	Query	1387	1163
	Increment	1387	1378
	Decrement	1387	9

Table 5.1: Average throughput of update and query operations for the baseline and secure CRDTs for 64 clients.

5.2.2 Results

For the benchmarks of Counter CRDTs, Register and MaxValue, I evaluated the update and query workloads for an exponentially increasing number of clients, from a single client (2^0) to 64 clients (2^6). In Table 5.1 I show how my proposed secure CRDT constructions compares to the baseline in terms of throughput. This table shows the maximum throughput reached by my constructions with an average peak of 1388 ops/s for 64 concurrent clients across all workloads. For a higher number of clients, the performance of the secure construction starts to decrease due to the overhead of sharing and resharing data between three parties for every request. Figure 5.1 show how the performance of my approach to secure CRDTs compares with the baseline implementations as the number of concurrent client increases. More specifically, this image shows how the throughput of the Register, GCounter and PNCounter update operation scales from $\sim$ 24 ops/s for 1 client up to $\sim$ 1380 ops/s for 64 clients. Overall, the majority of the CRDT operations presented in Table 5.1 have at maximum overhead of $\sim$ 2%.

It's important to highlight the overhead of the update operation in the MaxValue counter. This operation is $\sim$ 154 times slower than the baseline as shown in Table 5.1. This operation stabilizes at $\sim$ 8 ops/s for 2 concurrent clients. For any number of concurrent clients greater than 2, the request latency increases significantly. However, this overhead is expected, as the update operation of the GCounter requires multiple SMPC protocols for `equality` comparison, `greater than or equal to` comparison and integer multiplications. All of these SMPC

Figure 5.2: Average Latency of increment and decrement operations on secure MinBounded-Counter.

SET CRDT	Operation	Set Size			
		8	16	32	64
SMPC	Query	9.77	6.03	3.41	1.76
	Update	9.02	5.80	3.45	2.08
Baseline	Query	24.04	24.05	23.89	23.66
	Update	40.10	40.13	40.13	40.12

Table 5.2: Average throughput of update and query operations for the baseline and secure SET CRDT with a fixed set sizes from 8 to 64.

protocols require multiple communication rounds between the parties and results in a significant overhead. More specifically, the greater or equal than protocol has the highest communication overhead whereas the multiplication protocol requires only a single communication round.

The update operation of the MinBoundedCounter also differs in performance to the other Counters. Figure 5.2 shows that the increment operation has at most a throughput decrease of $\sim 0.6\%$ in comparison to the baseline as it does not require any SMPC protocol. However, the decrement operation uses the `equality` and the `greater than or equal to` SMPC protocols to ensure the counter does not decrease bellow it's lower bound. As such, the decrement operation has a maximum throughput of ~ 9 ops/s which is reached with two concurrent users.

For the evaluation of the secure SET CRDT, I measure the throughput of a construction that discloses some information. More specifically, it discloses when an element exists in the set during the update operation. The query operation can retrieve a value from the set without any information disclosure. This evaluation was done with a single client for different set sizes as defined in the workload specification. In Table 5.2 I show how the throughput of update and query operations of the secure SET compares to the baseline. Overall, the secure SET update operation has a maximum throughput of ~ 9 ops/s for the smallest set size, and decreases to ~ 2 ops/s for a set size with 64 elements. The baseline CRDT has a consistent throughput of ~ 40 ops/s for the update and ~ 24 ops/s for the query operations.

5.3 Discussion

Overall, this experimental results show that the majority of secure CRDT construction have a small overhead of $\sim 0,26\%$ for both update and query operations. The constructions with the highest overhead are the the MaxValue update, Bounded Counter decrement and Set operations which are at most $\sim 154\times$ lower than the baseline. This overhead results from multiple factors, including my preliminary implementation of the secure CRDTs that can be significantly optimized and the underlying SMPC protocols.

Optimizing CRDT Constructions The current implementation of the secure CRDTs constructions and the overall communication framework is only an initial prototype. These implementations have several possible optimizations that can result in increased throughput. For example, both the MaxValue update and Bounded Counter decrement operations use at least 3 SMPC protocols for secret comparison per operation. This number of operations can be reduced at least to 2 and as such removing multiple communication rounds. Similarly, optimizations can be done by designing specialized SMPC protocols for CRDTs, a topic not explored in this paper.

Multi-Party Protocols I can also improve the performance of my constructions by using a different protocol suite. For this evaluation, I leveraged the Sharemind protocols which are optimized for data analysis in the three-party setting, but my contributions as well as my implementation are abstracted from the underlying protocols. The protocols can be replaced by recent optimizations to the three-party model where they reduce the number of communication rounds of the protocols [5] or by using a different class of protocols such as function secret sharing that have a constant number of communication rounds [12].

The results of this experimental evaluation, even though preliminary, demonstrate the practical applicability of my theoretical contributions. Secure CRDTs can have throughput similar to a baseline system by minimizing the number of multiparty protocols and still ensure the confidentially of the data.

Chapter 6

Conclusions

With the ever-growing use of distributed eventually consistent systems, which use Conflict-free Replicated Data Types (CRDTs) as their core data structures, it's important to provide a secure way for data aggregation. The presented dissertation aims to prove the potential to directly transpose any CRDT construction to its privacy-preserving variant, problem for which solutions are currently inexistent in literature.

The proposed approach leverages Multi-Party Computation (MPC) to enable computations over secret data that are not possible using standard encryption methods because of their limited expressiveness. This new generalized approach allows for faster development of privacy-preserving systems, as there is no need to carefully design each construction individually. The modularity of the system with its agnosticism to both network topology and MPC protocols allows for independent developments in either fields while maintaining an almost plug-and-play level of compatibility. Another objective was to show that simple MPC-based CRDT constructions can easily be used as building blocks for other more complex constructions with minor drawbacks, which was proved by the use of two grow-only counter CRDTs to build a pn-counter CRDT.

The results from the experimental validation are overall on par with expectations, which proves the feasibility for the use of MPC in the creation of secure CRDTs.

6.1 Future Work

Being the first work that uses MPC to build secure CRDT constructions, there are several improvements that can be done. For example, the arithmetic formula used to privately compute comparisons between secret values can be optimized in order to reduce the necessary communication round-trips, something that was considered but not implemented for the lack of time. More efficient set implementations can also be implemented, specially in relation to sets with a limited universe of possible elements. The proposed approach can also easily be extended to other more complex data types, either built from scratch or using the proposed constructions as building elements.

Bibliography

[1] Antidotedb: A planet scale, highly available, transactional database.

[2] Locust: An open source load testing tool.

[3] General Data Protection Regulation, 2016.

[4] Secure multi-party computation: Theory, practice and applications. *Information Sciences*, 476:357–372, 2019. ISSN: 0020-0255.

[5] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 805–817, New York, NY, USA, 2016. Association for Computing Machinery. ISBN: 9781450341394. doi:10.1145/2976749.2978331.

[6] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 31–36, 2015.

[7] Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela, and Nuno Preguiça. Secure conflict-free replicated data types. Cryptology ePrint Archive, Paper 2020/944, 2020.

[8] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N. Kho, and Jennie Rogers. Smcql: Secure querying for federated databases. *arXiv: Databases*, 2016.

[9] Alysson Neves Bessani, Miguel Pupo Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *ACM Trans. Storage*, 9:12, 2013.

[10] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. *IACR Cryptol. ePrint Arch.*, 2008:289, 2008.

[11] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11:403–418, 2012.

[12] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. Cryptology ePrint Archive, Paper 2018/707, 2018. https://eprint.iacr.org/2018/707.

[13] Loïck Briot, Pascal Urso, and Marc Shapiro. High Responsiveness for Group Editing CRDTs. In *ACM International Conference on Supporting Group Work*, Sanibel Island, FL, United States, November 2016.

[14] Christian Cachin, Esha Ghosh, Dimitrios Papadopoulos, and Björn Tackmann. Stateful Multi-client Verifiable Computation. In *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, volume 10892 of *Lecture Notes in Computer Science*, pages 637–656. Springer, 2018.

[15] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1:1277–1288, 2008.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. volume 41, pages 205–220, 10 2007.

[17] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, 2002.

[18] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. Sok: General purpose compilers for secure multi-party computation. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1220–1237, 2019.

[19] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. 3(OOPSLA), oct 2019.

[20] James J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10:3–25, 1992.

[21] Rusty Klophaus. Riak core: building distributed applications without shared state. In *CUFP '10*, 2010.

[22] Stephan A. Kollmann, Martin Kleppmann, and Alastair R. Beresford. Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing. *Proceedings on Privacy Enhancing Technologies*, 2019:210 – 232, 2019.

[23] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.*, 44:35–40, 2010.

[24] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. 21(7): 558–565, jul 1978.

[25] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, pages 558–565, July 1978.

[26] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. In *Proceedings of the 19th International Conference on Supporting Group Work*, pages 39–49, 2016.

[27] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. Senate: A Maliciously-Secure MPC platform for collaborative analytics. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2129–2146. USENIX Association, 2021.

[28] Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and R. Oliveira. Safefs: a modular architecture for secure user-space file systems: one fuse to rule them all. *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017.

[29] Rogério Pontes, Francisco Maia, Ricardo Manuel Pereira Vilaça, and Nuno Machado. d'artagnan: A trusted nosql database on untrusted clouds. *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 61–6109, 2019.

[30] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403, 2009. doi:10.1109/ICDCS.2009.20.

[31] Michal Ptaszek. Scaling lol chat to 70 million players., 2014.

[32] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37: 42–81, March 2005.

[33] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. 2011.

[34] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS*, 2011.

[35] Ali Shoker, Houssam Yactine, and Carlos Baquero. As secure as possible eventual consistency: Work in progress. *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, 2017.

[36] Werner Vogels. Eventually consistent. *Queue*, 6:14–19, 2008.

[37] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave. *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[38] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. Bluesky: a cloud-backed file system for the enterprise. In *FAST*, 2012.

[39] W. Wong, Ben Kao, David Wai-Lok Cheung, Rongbin Li, and Siu-Ming Yiu. Secure query processing with data interoperability in a cloud database environment. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

[40] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca A. Popa, Aurojit Panda, and Ion Stoica. Cerebro: A platform for multi-party cryptographic collaborative learning. In *IACR Cryptol. ePrint Arch.*, 2021.