# General-Purpose Secure Conflict-free Replicated Data Types

Bernardo Portela[1], Hugo Pacheco[1], Pedro Jorge[1], and Rogério Pontes[2]

[1] University of Porto (FCUP) and INESC TEC
{bernardo.portela,hpacheco,up201706520}@fc.up.pt
[2] INESC TEC
{rogerio.a.pontes}@inesctec.pt

**Abstract.** Conflict-free Replicated Data Types (CRDTs) are a very popular class of distributed data structures that strike a compromise between strong and eventual consistency. Ensuring the protection of data stored within a CRDT, however, cannot be done trivially using standard encryption techniques, as secure CRDT protocols would require replica-side computation. This paper proposes an approach to lift general-purpose implementations of CRDTs to secure variants using secure multiparty computation (MPC). Each replica within the system is realized by a group of MPC parties that compute its functionality. Our results include: i) an extension of current formal models used for reasoning over the security of CRDT solutions to the MPC setting; ii) a MPC language and type system to enable the construction of secure versions of CRDTs and; iii) a proof of security that relates the security of CRDT constructions designed under said semantics to the underlying MPC library. We provide an open-source system implementation with an extensive evaluation, which compares different designs with their baseline throughput and latency.

## 1 Introduction

Large-scale distributed software is ever more a reality. One popular class of distributed applications are *replicated stores* [21], in which a number of computers – or replicas – maintain multiple copies of shared data and exchange updates regularly to stay synchronized, while clients can fetch data from any replica. For example, large-scale Internet services may use geographically distributed replicas, or online applications may combine cloud and local replicas to support usage during offline periods.

Nonetheless, designing replicated stores requires balancing *consistency* and *availability*. They can provide *strong consistency*, behaving as if a centralized entity is handling all operations. However, this usually requires synchronization among replicas, and can decrease availability in large-scale geo-replicated systems, due to high-latency networks or network outages. For this reason, many stores often provide weaker *eventual consistency* guarantees, where replicas eventually reach the same shared state if clients stop submitting updates.

*Conflict-free Replicated Data Types* (CRDTs) [44] are a very popular class of distributed data structures that strike a compromise between strong and eventual consistency known as *strong eventual consistency*, i.e., replicas that have received the same updates have the same state, automatically merging conflicting updates without synchronization. CRDTs present a sequential-style interface: a standard object (counter, set, etc) with operations to query and update its state; the CRDT then encapsulates operations for propagating effects between replicas that are hidden from the application logic. CRDTs have a wide range of applications due to their low latency and high scalability. They are used in distributed NoSQL databases like Redis and Azure Cosmos DB, as well as in collaborative text editing or messaging applications, and financial services like PayPal [25].

Similar to other systems for distributed storage and processing, CRDTs raise important security concerns, as remarked by their authors [40]. One is that a malicious replica may interfere with the other replicas to undermine global convergence or consistency, what can be mitigated with standard authentication techniques [23]. Another challenge, further accentuated with the decentralized and geodistributed nature of cloud-based deployments, is related to the privacy of the data stored within

the CRDT. This cannot be done trivially using standard encryption techniques, as secure CRDT protocols would require replica-side computation – on encrypted data – to propagate operations. The work from [8] pioneers a security model for CRDTs, and defines a few tailor-made examples of secure CRDT constructions. Each construction must be carefully designed to use dedicated cryptographic techniques, so that the CRDT computations between replicas can be performed over encrypted data.

On the other hand, much of the CRDT literature [3, 24, 29, 44] is focused on the design of new CRDTs with tailored consistency restoration behaviors, to suit varied application requirements. For instance, even for the simple CRDT whose shared object is a boolean flag with enable/disable updates, there are already several possible behaviors for handling concurrent updates, such as enable-wins, disable-wins or last-writer-wins [29].

Ideally, we would like to be able to directly transpose each such CRDT implementation to its secure variant. This is not possible with the approach from [8], as there is limited expressiveness for computations over encrypted data and the CRDT semantics would need to be manually customized. Moreover, there is usually a security tradeoff, which is also fixed and hard to customize in the constructions from [8], between the data that is kept secure and the data that is revealed to allow non-encrypted computation.

The approach advocated in this paper is to show that it is possible to lift general-purpose implementations of CRDTs to secure variants using *secure multiparty computation* (MPC) [10]. MPC denotes a collection of cryptographic protocols that enable a number of untrusting parties to compute a function on joint input without disclosing their secure data. Recent advances [22] have prompted a plethora of MPC languages inviting programmers to write sequential-style ideal functionalities that are automatically translated to distributed MPC protocols over partitioned secret data. This possibility had been considered in [8], but promptly discarded:

> Intuitively, privacy-preserving CRDT operations could be realised through [...] general MPC. However, such solutions would [...] require sharing secret data between multiple nodes, which goes against the purpose of CRDTs in the first place.

Indeed, combining CRDTs and MPC is antagonistic if we map each CRDT replica to a MPC party. This paper reconciles these concepts by proposing an orthogonal approach: to consider that each CRDT replica is realized by a group of MPC parties that compute its functionality.

We first present an overview of our approach in Section 2. Section 3 reviews CRDT concepts, specification and implementation. Section 4 revisits the security model for abstract CRDTs. Section 5 introduces our MPC framework for concrete secure CRDTs and provides a formal security proof. Our contributions are as follows:

- An extension of current CRDT security formal models to the MPC setting and semantics.
- A high-level MPC language and type system to enable the construction of secure versions of CRDTs.
- A proof that relates the security of CRDT constructions designed under said semantics to the underlying MPC library used.

We provide an open-source[3] system implementation of a sample set of secure CRDT constructions with an extensive evaluation, which compares different designs with their baseline throughput and latency.

## 2 Technical Overview

This paper advocates the use of MPC to allow CRDT replicas to perform general-purpose privacy-preserving computation over encrypted data. Figure 1 captures our approach. In MPC (left column),

[3] `https://github.com/SecureCRDT/SecureCRDT`

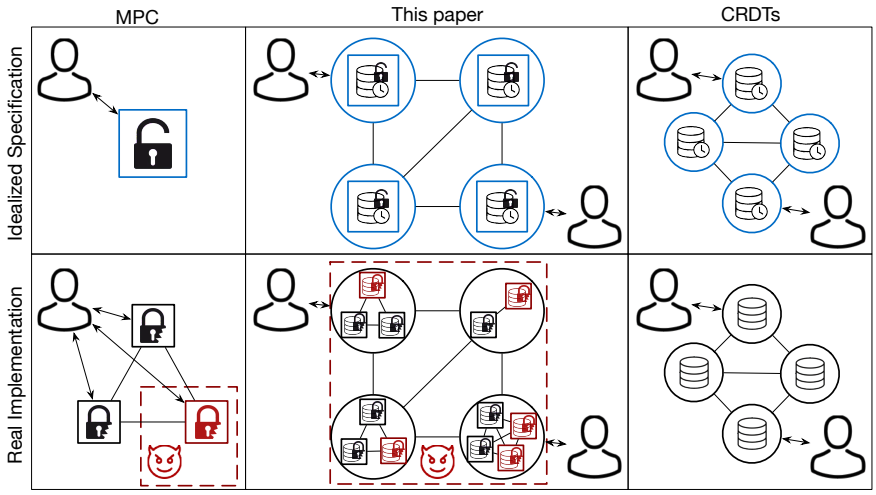


Fig. 1: Overview of our approach. Legend: client; MPC party; CRDT replica; – specification; – adversary control; plain secret value; encrypted secret share; replica state; history of events.

a client interacts with a set of parties that embody different trust domains and store partitions of a secret, to collaboratively execute a privacy-preserving protocol. In CRDTs (right column), a number of clients may interact with any out of a set of replicas that synchronize among them to increase the availability of a distributed store. In our proposed approach (middle column), each CRDT replica is composed by a set of MPC parties. The intuition is that the CRDT client interface remains the same, while the store is distributed across two orthogonal axes: replicas duplicate the store and ensure eventual consistency, and parties split each replica's data across trust domains to ensure data privacy.

*Applications:* One natural instantiation of our approach is a secure cloud service at scale, where replicas are geo-distributed, with parties running in separate cloud providers at the replica's location. E.g., each replica can be emulated by parties of different cloud providers, such that no single cloud provider controls a threshold of parties that allows recovering the secretly-stored data. This, assuming non-collusion of cloud providers over the given threshold, ensures privacy of stored data. Another concrete use case are secret vaults [45] shared among groups of users; they may use secure CRDTs where a coalition of users , e.g. a team within a company, forms a replica that synchronizes with other replicas to manage a global secret vault within the company. They may use secure CRDTs formed by coalitions of users behaving in standard MPC fashion: simultaneously playing the role of an MPC participant (party in a replica) and the role of a client of the service. This enables a decentralized global secret vault within the company, even in the presence of adversarial users.

More generally, our approach can be seen as a path to scale existing MPC use cases, typically deployed in a restricted setting and where trust domains are already well-established; examples include confidential databases, satellite-collision prevention, tax fraud detection and cross-market or societal studies [5]. Our approach can also be seen as a way to bring security to existing CRDT use cases [27], but doing so requires a sensible partitioning into trust domains. In many CRDT applications, users run local replicas that synchronize user data such as favorites or friend lists with decentralized application servers; server-side replicas could be partitioned across different infrastructures to ensure secure data processing.

In our approach, the mapping between MPC parties and CRDT replicas is statically fixed. Some efforts towards scaling MPC for large numbers of parties such as the secure polling application from [7] consider secure computation among dynamic groups of parties and data replication through the use of $n$-out-of-$m$ secret-sharing schemes. Exploring a more general setting where MPC parties can be dynamically selected to enable the different CRDT replicas is interesting future work, but falls outside the scope of the formal model developed in this paper. Indeed, this would be useful to strengthen the aforementioned secret vaults use case, allowing for users to be dynamically added/removed from the coalitions.

*Security model:* Both for MPC and CRDTs, formal proofs are defined by relating idealized specifications to real implementations, respectively the top and bottom rows of Figure 1. Standard security results for MPC (left) entail that, against some threshold of adversary-controlled parties, protocols over secret shares behave like idealized functionalities that process the secret data in a black-box manner. Standard correctness results for CRDTs (right) entail that replicas behave like abstract specifications that know the global history of events.

Observe that replicas are virtual entities, composed of sets of MPC parties. For simplicity, we establish that they communicate directly with each other, or rather, designated parties of different replicas communicate directly with each other. We assume standard point-to-point secure authenticated channels between MPC parties, which can be instantiated with standard TLS-secured channels. This will ensure that adversaries cannot eavesdrop in-transit shares, and trivially break data privacy. This paper gives a proof that, against arbitrary thresholds of adversary-controlled parties per replica, security of our approach can be demonstrated upon standard results.

*Deployment:* Despite the main contribution of this paper is a formal model for MPC-based secure CRDTs, it also makes some preliminary steps towards a language-based framework for the design and execution of MPC-based secure CRDTs. In particular, we exemplify how our abstract general model can be instantiated with an `Haskell` embedded domain-specific language for the specification of secure CRDTs, allowing programmers to reason about security tradeoffs while assuming idealized MPC functionalities; this language would allow connecting to real MPC protocols, but this has not been exercised. To support our experimental evaluation, we have instead implemented a set of selected secure CRDT constructions as an independent `Java` library with real MPC protocols. A complete framework would integrate with existing MPC frameworks [2, 22] to ensure the secure compilation of general secure CRDT designs to actual implementations. Further interesting future work would be to extend existing CRDT frameworks such as [30, 18] with security guarantees, in order to allow users to describe more simply the data to be securely replicated.

## 3 CRDTs

Despite more than a decade of research, there is no universal formal definition for CRDTs [20, 31]. This requires formalising not only the replication algorithms, but also their communication patterns. Careful balance of communication and application requirements has also given rise to different families of state- [44], operation- [44] or delta-based [3] CRDTs, with different assumptions on when and how replicas synchronize updates.

### 3.1 Basic notions

We assume that $n$ replicas are statically defined at the beginning of the protocol, and can be identified by $i, j \in \mathbb{I}$. These nodes are accessible to an arbitrary number of clients, which will perform

```
type State
data Query r
data Update
type Message
new :: 𝕀 -> State
query :: 𝕀 -> Query r -> State -> r
update :: 𝕀 -> Update -> State -> State
propagate :: 𝕀 -> 𝕀 -> State -> Message
merge :: 𝕀 -> 𝕀 -> Message -> State -> State
```

Fig. 2: Abstract interface for a CRDT.

query/update operations. Each update operation performed on a CRDT replica will be represented by an *event* $e \in E$. Query operations will not be recorded as events, as they do not change the state for further operations. We will also refer to sets recording the history of events as $E \in \mathbb{E}$, where $\mathbb{E}$ denotes the set of sets of events. Following [29], we model an event as $e = (uid, op, deps)$, having a unique identifier $uid(e)$, an operation $op(e)$ and a set of dependencies $deps(e)$ (all the events known at the origin replica where the event was initially applied). Each unique event identifier $uid = (i, c)$ is a tuple of replica $i$ and a unique replica-local counter $c$. $E(uid)$ will denote the unique event in set $E$ referred by $uid$. We also define the dependencies of sets of events as $deps(E) = (\bigcup_{e \in E} deps(e)) \setminus E$.

In our representation [29], each event also keeps all its causal past, to allow for the expression of the "happens-before" causality relation. $e_1 \prec e_2$ means that $e_1$ happened before $e_2$, i.e. the effects of $e_1$ had been applied in the replica where $e_2$ was executed. We define $e_1 \prec e_2 = deps(e_1) \subset deps(e_2)$. $c_i(E)$ denotes the filtering of events in $i$ before $c$ ($\{e \mid e \in E \wedge e \prec E((i, c))\}$).

### 3.2 Implementation

A CRDT can be simply seen as a regular abstract data type that supports query and update operations. If all the operations on the data type are commutative (i.e., independent of the order in which they are applied), then it can be easily replicated. Unfortunately, this is not the case for most data types, and several concurrent behaviors are possible for non-commutative updates, requiring the CRDT to explicitly handle concurrency.

Figure 2 presents an abstract CRDT interface in `Haskell`. Each replica keeps an internal `State`, that can be initialized with a `new` operation. The types `Query` and `Update` define supported user operations, that can be respectively executed using the `query` and `update` functions. To simplify, queries only read state and updates only change state. Updates that return output could be modeled as updates plus queries. Note that `Query` is a generalized algebraic data type, parameterized by a type variable `r` which denotes the result type for each query that is seen by the user. The additional `propagate` and `merge` functions define the concurrent behavior of the CRDT, namely, how a replica $i$ shall produce a `Message` to be sent to another replica $j$ based on its `State`, and how replica $j$ shall merge a `Message` received from $i$ into its `State`. Figure 3 exemplifies the `Haskell` implementation of a delta-based grow-only counter CRDT from [3].

### 3.3 Specification

Two of the first proposals to unify the formalization of CRDTs were given in [15, 50]: the main idea, followed by many other works such as [21, 29, 31, 39], is to separate a CRDT implementation from its abstract specification, defined not as a function on states but as a function over the history of events (together with the relationships between them). We extend each CRDT with an abstract *specification function* $\mathsf{F} : \mathbb{I} \to \texttt{Query r} \to \mathbb{E} \to \texttt{r}$ that declaratively defines the correct query behavior from the

```
type State_GC = Map 𝕀 Int
data Query_GC r where Get_GC :: Query_GC Int
data Update_GC = Inc_GC { inc_GC :: Int }
type Message_GC = Map 𝕀 Int
new_GC i = Map.empty
query_GC i Get_GC st = Map.foldr (+) 0 st
update_GC i (Inc_GC n) st = Map.alter (Just . maybe n (+n)) i st
propagate_GC i j st_i = maybe empty (Map.singleton i) (Map.lookup i st_i)
merge_GC i j st_i m_j = Map.unionWith max st_i m_j
```

Fig. 3: Grow-only counter CRDT [3].

viewpoint of a single replica after a given history of update events. For instance, we can define the specification for the grow-only counter CRDT (Figure 3) as $\mathsf{F}_{\mathtt{GC}}(i, \mathtt{Get}_{\mathtt{GC}}, E) = \sum_{e \in E} \mathtt{inc}_{\mathtt{GC}}(op(e))$.

A natural, albeit challenging way to prove properties for CRDTs is to demonstrate *functional correctness*, i.e., guaranteeing that the concrete implementation respects an abstract specification. The main challenge lies in formalizing the relationship between the state of a replica as a concrete sequence of operations is applied (denoting a total ordering) and corresponding abstract histories of events (that only capture a partial ordering) [15, 31, 50]. Following [50], we can define a *consistency relation* $E \approx \mathtt{st}$ that relates a history of events $E$ with a concrete state $\mathtt{st}$. The intuition is that this relation shall hold for the empty history and the `new` state, and be inductively preserved as CRDT operations are applied to the state. Concretely, we can inductively define this relation as in [20]:

$$E \approx \mathtt{new\ i} \tag{1}$$

$$E \approx st \wedge (\nexists e \in E.\ e_2 \prec e) \rightarrow E \cup \{e_2\} \approx \mathtt{apply}(op(e_2), \mathtt{st}) \tag{2}$$

Here, `apply` denotes the application of an operation to a state. We make further standard assumptions that all domains are finite and that all CRDT operations terminate on all inputs [44]. We can define functional correctness as follows:

**Definition 1 (Functional Correctness).** *If* $E \approx \mathtt{st}$*, then:*

$$\mathtt{query}\ i\ \mathtt{op}\ \mathtt{st} = \mathsf{F}(i, \mathtt{op}, E)$$

As domain-specific constructions, CRDTs have other desired properties beyond functional correctness. The most important are *strong eventual consistency* [44], which entails eventual update delivery (and depends on the network [20]) and strong convergence (replicas with same updates have equivalent state). As made clear below, all additional properties will be orthogonal to our notion of CRDT security, which is postulated even for CRDTs that do not meet such properties.

## 4 Secure CRDTs

Following [8], we formalize CRDT security in the Universal Composability (UC) framework [17]. The intuition is depicted in Figure 4. At the center, we have environment $\mathcal{Z}$, whose role is to distinguish the real world, where the CRDT protocol is executed, from an ideal world, where it instead interacts with a trusted party behaving as an ideal functionality. The role of $\mathcal{Z}$ is to select inputs, using query and update, and then use the adversary to schedule how the communication channels behave, using propagate and merge.

To capture scenarios where an attacker breaches the security of a participant of our secure CRDT protocol, our model also considers *corruptions*. These will present an additional challenge

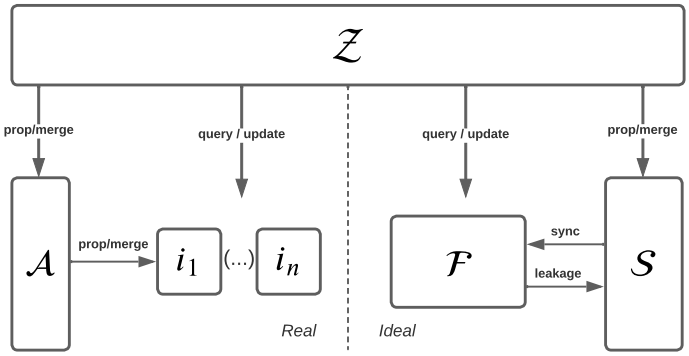


Fig. 4: Real versus Ideal world model for secure CRDTs.

in the real-world, by having participants $i$ revealing additional information to $\mathcal{A}$, related to their internal state. For generality, our model for secure CRDTs does not impose restrictions on participant corruptions. Some protocols – with examples in [8] – can be shown to be secure even if some participants are entirely corrupted, while others – as is the case with our MPC-based instantiation – require that corruptions on participants do not exceed a specific threshold. We consider two common relaxations of the model, related to participant corruptions: corruptions are *static*, meaning that the corrupt parties are statically defined at the beginning of the execution; and adversaries are *semi-honest*, meaning that they observe the internal state and communication messages of corrupt parties, which we denote as the aggregated *trace* of operations. The latter is a common assumption when outsourcing to cloud providers [11, 19], as data breaches allow for an adversary to observe an execution trace, but not to act arbitrarily during the actual protocol execution.

We adopt a network-agnostic view of CRDTs, similar to modern CRDT libraries such as automerge [26] or yjs [33]. As such, replicas execute asynchronously, and the distinguisher is given full control over the scheduling of operations in each replica. We will assume directional channels between replicas, modeled as queues, which ensures causal delivery. For simplicity, we assume that secure operations execute atomically.

### 4.1 Real World vs Ideal World

In the *real world*, $\mathcal{Z}$ interacts with adversary $\mathcal{A}$ and replicas $i_1, \ldots, i_n$, using two interfaces: query and update trigger input/output operations in replicas, producing an execution trace; propagate and merge are used by $\mathcal{A}$ to animate the network propagation, also producing an execution trace. This execution trace will contain publicly observable information regarding the protocol execution (e.g. exchanged messages), as well as additional data from corrupt parties (e.g. the state stored in a specific participant). In the *ideal world*, $\mathcal{Z}$ instead interacts with a simulator $\mathcal{S}$ and an ideal functionality $\mathcal{F}$: query and update operations denote CRDT events on $\mathcal{F}$, which emulates the behavior of $n$ replicas; propagate and merge are handled entirely by $\mathcal{S}$. We say that $\mathcal{F}$ plays the role of a trusted black-box, receiving inputs in a per-replica basis and following the CRDT *specification function* F to provide outputs. The role of $\mathcal{S}$ is to emulate the execution traces to $\mathcal{Z}$: it receives some leakage from $\mathcal{F}$, and is given access to a sync procedure of the ideal functionality, which idealizes the passing of events from one emulated replica to another within $\mathcal{F}$.

Observe that the behavior displayed by the *ideal world* entails both correctness and security, as $\mathcal{F}$ behaves according to F, and allows for nothing but the concretely specified leakage to exit its controlled environment. Leakage is kept abstract in our security model, and will be made concrete

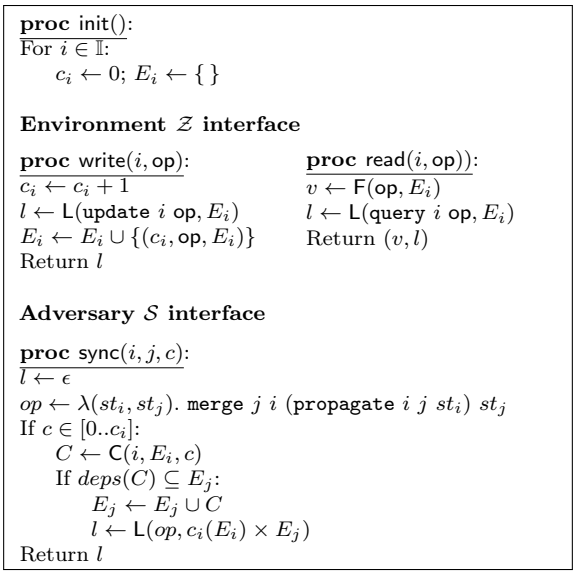


Fig. 5: Ideal functionality $\mathcal{F}$.

for particular instantiations, e.g. revealing the type of operations, or the message length. A protocol is said to UC-realize a given ideal functionality $\mathcal{F}$ if, for any *real world* adversary $\mathcal{A}$ interacting with a protocol, there exists an *ideal world* simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ can distinguish if it is interacting with $\mathcal{A}$ and actual replicas running the protocol, or with an $\mathcal{S}$ with very little information about the sensitive data in the system, and an idealized black-box. The security result entails that interactions observed by $\mathcal{Z}$ in the *real world* are indistinguishable from those observed in the *ideal world*.

### 4.2 Ideal Functionality

The ideal functionality that captures the expected behavior of the CRDT (Figure 5) follows the same structure as in [8], refined to have a concrete initialization procedure, and allowing for a more straightforward synchronization process. For each replica $i \in \mathbb{I}$, it keeps track of its views, defined as a set of of events $E_i$. Local counters $c_i$ identify replica-local events uniquely and allow reconstructing an event dependency graph. The environment interface either writes an update operation to the event history, or reads the result of a query operation over the current event history. For both query and update operations, their level of security is defined by a *leakage specification function* $\mathsf{L} : O \to \mathbb{E} \to L$. It receives an operation $op \in O$, a history of events $E \in \mathbb{E}$, and produces a leakage trace in the domain $L$. Here, $O$ denotes the set of CRDT operations that result from invoking the interfaces write, read or sync of the ideal functionality $\mathcal{F}$.

The ideal functionality allows $\mathcal{S}$ to control communication, using sync, which emulates the sending of updates from a replica to another up to a certain local counter. This counter is used by $\mathcal{S}$ to simulate the propagation of older updates. The CRDT communication pattern is abstracted by a *communication specification* function $\mathsf{C} : \mathbb{I} \to \mathbb{E} \to \mathbb{N} \to \mathbb{E}$, where $C = \mathsf{C}(i, E, c)$ must obey certain rules: sent updates must be a subset of source events before the counter; sync only synchronizes source updates with applied dependencies in the target. For instance, propagating a state in a state-based CRDT equals sending all history of locally applied updates ($\mathsf{C}_{\mathtt{st}}(i, E, c) = c_i(E)$). Delta-based CRDTs only send the history of locally-originated updates (($\mathsf{C}_{\mathtt{dt}}(i, E, c) = \{e | e \in E \wedge uid(e) = (i, c_e) \wedge c_e < c\}$)). In operation-based CRDTs, each update is typically broadcast to all other replicas

after having been locally applied ($\mathsf{C}_{\mathsf{op}}(i, E, c) = \{e | e \in E \wedge uid(e) = (i, c_e) \wedge c_e = c - 1\}$). We further define communication correctness as follows:

**Definition 2 (Communication Correctness).** *If $c_i(E_i) \approx \mathtt{st}_i$ and $E_j \approx \mathtt{st}_j$, and given $C = \mathsf{C}(i, E_i, c)$ such that $deps(C) \subseteq E_j$, then:*

$$E_j \cup C \approx \mathtt{merge}\ j\ i\ (\mathtt{propagate}\ i\ j\ \mathtt{st}_i)\ \mathtt{st}_j$$

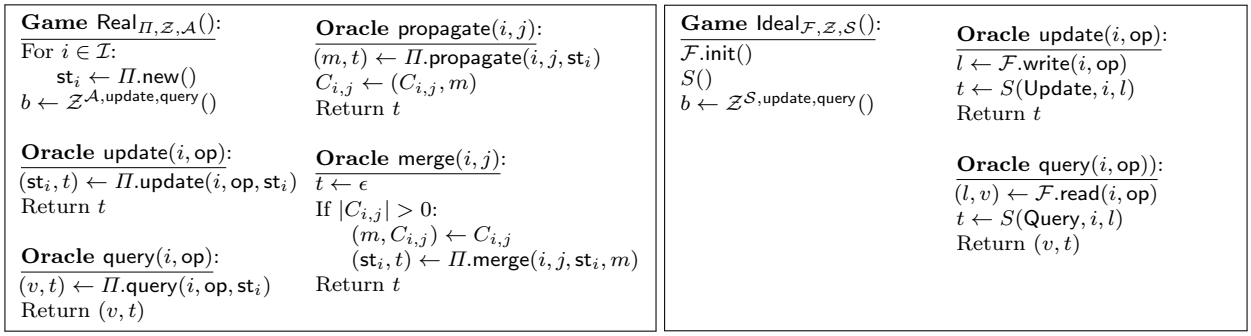


Fig. 6: Security games for secure CRDTs. In the real world (left), $\mathcal{A}$ has access to oracles propagate and merge. In the ideal world (right), $\mathcal{S}$ has access to the adversarial interface of $\mathcal{F}$.

### 4.3 CRDT Security

The concrete execution model considered for the real-versus-ideal-world model is presented in Figure 6. In the real world, we begin by initializing all replicas in $\mathbb{I}$. Afterwards, $\mathcal{Z}$ is allowed free interaction with the system. This is done either using update and query, which calls $\Pi$ over a replica state, or using the adversarial interface $\mathcal{A}$. This essentially consists in scheduling the propagation of operations according to the specifications of $\Pi$, calling propagate to put a message in a communication channel, or merge to fetch a message from a channel and apply it to a replica. The execution of $\Pi$ produces a trace $t$ which can be observed by $\mathcal{A}$.

In the ideal world, we instead initialize the functionality $\mathcal{F}$ and simulator $\mathcal{S}$. Every call to update and query will trigger the specified behavior in $\mathcal{F}$. This will produce the result (for query) and a leakage $l$, which will be used by $\mathcal{S}$ to emulate the real trace $t$. The adversarial interface here is fully controlled by $\mathcal{S}$, which must also emulate traces for propagate and merge, using its interface on $\mathcal{F}$ to trigger replica synchronization.

Our definition for secure CRDTs is as follows:

**Definition 3 (CRDT security).** *Let $\mathcal{F}$ be an ideal functionality and let $\Pi$ be the corresponding CRDT protocol. We say that $\Pi$ securely realizes $\mathcal{F}$ if there exists a simulator $\mathcal{S}$ such that, for any behavior of environment $\mathcal{Z}$ and adversary $\mathcal{A}$, the following is true.*

$$\mathsf{Real}_{\Pi,\mathcal{Z},\mathcal{A}} \approx \mathsf{Ideal}_{\mathcal{F},\mathcal{Z},\mathcal{S}}$$

Note that Definition 3, as usual for cryptographic security definitions, implies functional correctness (Definition 1). This will become particularly clear in Section 5, when we resort to Definition 1 to prove Definition 3.

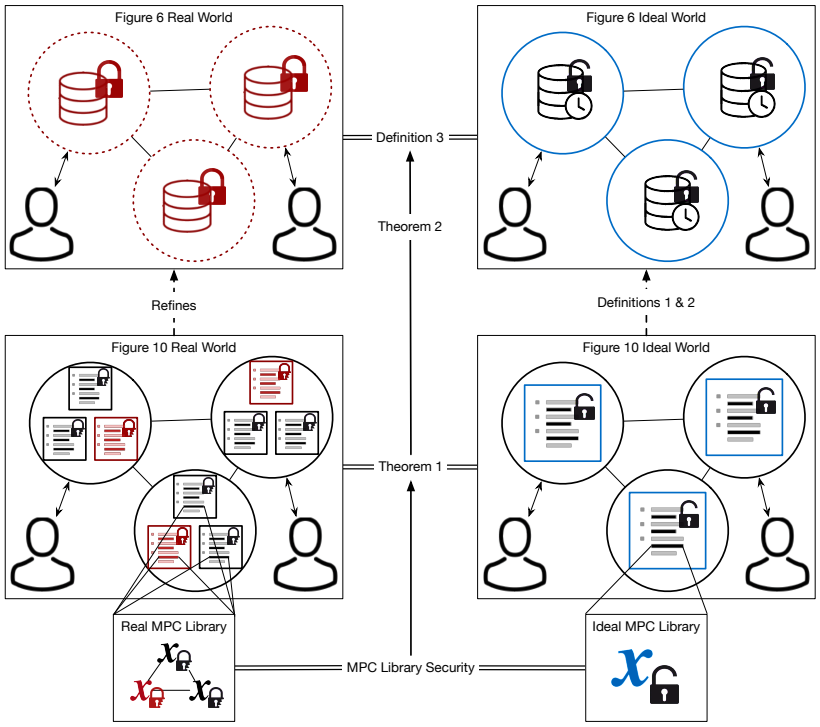


Fig. 7: Outline of our formal rationale for MPC-based secure CRDTs. Legend: ◌ virtual replica; 🔒 encrypted secret value; ≣ MPC program; $\boldsymbol{x}$ MPC variable.

## 5 MPC-based Secure CRDTs

Section 4 presented a general model for the security of abstract CRDTs from Section 3. Since many proposals of CRDTs come with pseudo-code descriptions, as exemplified in Section 3.2, we would like to lift them to secure variants while preserving the same interface. The path proposed in this section (Figure 7) is to:

1. reason about the MPC security of a program that interacts with the user to execute secure CRDT operations written in a MPC language (Theorem 1);
2. combine security of MPC programs implementation with their functional correctness w.r.t. a specification to obtain CRDT security (Theorem 2).

We emphasize the distinction between CRDT descriptions (Section 3) and their interface in the security model (Section 4) by decoupling programs according to two semantics: a non-interactive semantics that animates CRDT operations, and an interactive semantics that simply articulates calls to CRDT operations. This also connects elegantly to secure execution using MPC: the non-interactive semantics can be instantiated with any language that supports secure computation, relying on assumptions that primitive MPC operations offered by a MPC library are secure, while the interactive semantics captures the articulation between different replicas running MPC programs and clients.

Interestingly, in the MPC setting CRDT operations are no longer atomic: they receive input, execute a sequence of public and secret computations over the replica's state, and (possibly) return output. Indeed, the execution of such computations will produce a trace, which must be emulated by $\mathcal{S}$ in the ideal world. To accommodate theses notions into our security model for MPC-based CRDTs, we will adapt the program-based MPC security framework from [2], where the adversary

can interactively control the step-wise execution of MPC programs and observe their intermediate traces. Note that such MPC traces will provide the finer granularity of all program steps necessary to execute a CRDT operation.

### 5.1 MPC Security

Each replica $i$ (in the real world) is emulated by a set of computing parties $\mathcal{P}_i = p_{m_1}, \ldots, p_{m_i}$ that collaboratively compute a distributed protocol over secret-shared data, where $m_i$ is the number of participants in the replica's MPC protocol. For concreteness, we consider a linear secret sharing scheme where a value $s$ is shared as $s_{m_1}, \ldots, s_{m_i}$, denoted as $\overline{s}$. We assume authenticated communication channels among MPC players. Per replica, we allow for a threshold of parties $\mathcal{C}_i \subset \mathcal{P}_i$ to be corrupt, sharing their internal state and messages sent/received to the adversary $\mathcal{A}$. We assume *static* corruptions, which means that sets $\mathcal{C}_i$ are fixed at the start of the protocol.

### 5.2 MPC Programming Language

Even though the MPC security framework from [2] is postulated generically, for concreteness we will follow Section 3.2 and consider the design of an embedded domain-specific MPC language in `Haskell`.

*Type System:* To make our language security-aware, we define a new abstract type **S** `a` to denote secure data of type `a`. Secure types and operations will be highlighted in red. The rationale is to rely on the type system to enforce a strict separation between secret and public data: host programs do not get to know the value of **S**-tagged data. As such, all expressions over secure data *need to be* delegated to an external MPC library[4]. We will always assume that programs are well-typed.

*MPC library:* Figure 8 defines the MPC library operations that will be later used in our examples. Note that this list is not exhaustive, and many other MPC operations may be supported. Syntax may overload standard Haskell functions. We denote MPC secure operations as `sop`. They will have two interpretations: an ideal-world specification, given by a function $\mathscr{f}_{\texttt{sop}}(\vec{v}) = (v, l)$ over values that ignores security labels and returns leakage; and a real-world protocol $\pi_{\texttt{sop}}(\vec{\overline{s}}) = (\overline{s}, \tau)$, that captures its distributed secure execution over a vector of secret-shared values, returning a new secret-shared value and a trace. The two special $\mathscr{f}_{\texttt{classify}}(a) = (a, \epsilon)$ and $\mathscr{f}_{\texttt{declassify}}(a) = (a, a)$ operations are the only ones that translate between public and secret data. Allowing for secret data to be made explicitly public is important when designing MPC protocol, as it allows for exploring efficiency/security trade-offs, e.g. avoiding branching on secret values by declassifying the conditional value. In our MPC library, no operation besides `declassify` has leakage. The following sub-section details the security definition expected from the MPC library.

**Semantics** As common for MPC [1, 2, 41], the execution of our language follows a small-step operational semantics with ideal- and real-world interpretations.

*Ideal:* The ideal-world semantics is modeled as if a single trusted party was executing the program and defined as a binary relation on expressions:

$$\frac{\Gamma \vdash e \underset{\mathsf{hs}}{\longrightarrow} e'}{e \xrightarrow{\epsilon} e'} \qquad \frac{\mathscr{f}_{\texttt{sop}}(\vec{v}) = (v, l)}{e[\texttt{sop}(\vec{v})] \xrightarrow{l} e[v]}$$

[4] An actual `Haskell` embedding, with a concrete type for **S**, could use pure values as references to secure data, as can be found in [2].

```
class MPC a where
  classify   :: a -> S a
  declassify :: S a -> a
class SNum a where
  (+) :: S a -> S a -> S a
  (-) :: S a -> S a -> S a
  if' :: S Bool -> S a -> S a -> S a
class SEq a where
  (==) :: S a -> S a -> S Bool
class SOrd a where
  (>=) :: S a -> S a -> S Bool
  max  :: S a -> S a -> S a
(||) :: S Bool -> S Bool -> S Bool
(&&) :: S Bool -> S Bool -> S Bool
not  :: S Bool -> S Bool
```

Fig. 8: Secure MPC library.

The intuition is that the ideal semantics describes the step-wise reduction of expressions $e$, until it produces a final value $v$ or diverges (no rule applies). For non-MPC expressions, we rely on the semantics of the host language, where $\Gamma$ is a fixed global context with definitions for CRDT functions and constants: $\Gamma \vdash e \underset{\mathsf{hs}}{\longrightarrow} e'$ denotes a small-step reduction of expression $e$ into expression $e'$ under context $\Gamma$. We use a call-by-value convention for the semantics of MPC operations, where $e[e']$ is used to denote the selection/substitution of sub-expression $e'$ in $e$[5]. We keep the syntax and semantics of values and expressions largely abstract, as it is orthogonal to our formalization; an example of a concrete semantics for the subset of `Haskell` that we are using, which is rather standard, may be found in [46]. We say that the big-step evaluation of expression $e$ terminates in value $v$ with leakage $l$, written $e \Downarrow_l v$, if $e \xrightarrow{l}{}^* v$, where $\rightarrow^*$ forms the reflexive transitive closure of $\rightarrow$ and leakage is concatenated into a leakage trace.

*Real:* In the real-world semantics, a set of parties $\mathcal{P}$ jointly computes the functionality; we reuse the secret sharing notation for multiparty expressions. Each party locally executes its own copy of the original expression, cooperating through the execution of protocols triggered by calls to MPC operations:

$$\frac{p \in \mathcal{P} \quad \Gamma \vdash e \underset{\mathsf{hs}}{\longrightarrow} e'}{\overline{e}[p \mapsto e] \overset{\epsilon}{\Rightarrow} \overline{e}[p \mapsto e']} \qquad \frac{\pi_{\mathtt{sop}}(\overline{\vec{v}}) = (\overline{s}, \tau)}{\overline{e[\mathtt{sop}(\vec{v})]} \overset{\tau}{\Rightarrow} \overline{e[s]}}$$

For non-MPC expressions, parties may progress independently. Evaluation of MPC operations enforces synchronization by requiring that all parties are evaluating the same expression (modulo sharings of secret data). To guarantee that all parties go through the same steps when executing the same expression, we rely on two main assumptions on the semantics of the host language [2]: deterministic local reduction, which is reasonable for an actual language implementation; and independence from secret values, which is guaranteed by the type system.

### 5.3 Security of the MPC library

In our framework, we adopt the notion of secure MPC library from [2]. The intuition is that the environment and the adversary only inspect the corrupted views of intermediate computations, but

[5] An actual `Haskell` embedding could use a call-by-need semantics, but additional technical care – such as the use of monads to perform secure operations – would be necessary to ensure party synchronization.

$\boxed{\langle \overline{s}, \overline{st} \rangle \dashrightarrow_i \langle m, \overline{e} \rangle}$ (**Input processing** for replica $i$ (used in the ideal and real worlds): Given shared value $\overline{s}$ and shared replica state $\overline{st}$, enters mode $m$ for executing secret-shared expression $\overline{e}$. Values overloaded as shares for one party.)

$$\textsc{InputQuery}\ \frac{}{\langle \overline{\mathsf{Query}\ s}, \overline{st} \rangle \dashrightarrow_i \langle \mathsf{Query}, \overline{\mathtt{query}\ i\ s\ st} \rangle} \qquad \textsc{InputUpdate}\ \frac{}{\langle \overline{\mathsf{Update}\ s}, \overline{st} \rangle \dashrightarrow_i \langle \mathsf{Update}, \overline{\mathtt{update}\ i\ s\ st} \rangle}$$

$$\textsc{InputPropagate}\ \frac{\overline{e} = \overline{\mathtt{propagate}\ i\ j\ st}}{\langle \overline{\mathsf{Propagate}\ j}, \overline{st} \rangle \dashrightarrow_i \langle \mathsf{Propagate}\ j, \overline{e} \rangle} \qquad \textsc{InputMerge}\ \frac{}{\langle \overline{\mathsf{Merge}\ j\ s}, \overline{st} \rangle \dashrightarrow_i \langle \mathsf{Merge}\ j, \overline{\mathtt{merge}\ i\ j\ st\ s} \rangle}$$

$\boxed{\langle o, \overline{s} \rangle \dashleftarrow_i \langle m, \overline{v} \rangle}$ (**Output processing** for replica $i$ (used in the ideal and real worlds): Given mode $m$ and secret-shared value $v$, produces secret-shared output $\overline{s}$ under output mode $o$. Values overloaded as shares for one party.)

$$\textsc{OutputQuery}\ \frac{}{\langle \mathsf{Out}, \overline{\mathsf{Query}\ v} \rangle \dashleftarrow_i \langle \mathsf{Query}, \overline{v} \rangle} \qquad \textsc{OutputUpdate}\ \frac{}{\langle \mathsf{St}, \overline{v} \rangle \dashleftarrow_i \langle \mathsf{Update}, \overline{v} \rangle}$$

$$\textsc{OutputNew}\ \frac{}{\langle \mathsf{St}, \overline{v} \rangle \dashleftarrow_i \langle \mathsf{New}, \overline{v} \rangle}$$

$$\textsc{OutputPropagate}\ \frac{}{\langle \mathsf{Out}, \overline{\mathsf{Propagate}\ j\ v} \rangle \dashleftarrow_i \langle \mathsf{Propagate}\ j, \overline{v} \rangle} \qquad \textsc{OutputMerge}\ \frac{}{\langle \mathsf{St}, \overline{v} \rangle \dashleftarrow_i \langle \mathsf{Merge}\ j, \overline{v} \rangle}$$

$\boxed{\langle \sigma, \mathcal{I}, \mathcal{O} \rangle \xrightarrow{l}_i \langle \sigma', \mathcal{I}', \mathcal{O}' \rangle}$ (**Ideal-world semantics** for replica $i$: Under local configuration $\sigma$, input buffer $\mathcal{I}$ and output buffer $\mathcal{O}$, yields $\sigma'$, $\mathcal{I}'$ and $\mathcal{O}'$ and returns leakage $l$.)

$$\textsc{IdealInput}\ \frac{\mathcal{I} \neq \emptyset \quad f_{\mathtt{input}}(\mathcal{I}) = (s, \epsilon) \quad \langle s, st \rangle \dashrightarrow_i \langle m, e \rangle}{\langle \langle \mathsf{Idle}, st \rangle, \mathcal{I}, \mathcal{O} \rangle \xrightarrow{\epsilon}_i \langle \langle \mathsf{Run}\ m\ e, st \rangle, \emptyset, \mathcal{O} \rangle} \qquad \textsc{IdealStep}\ \frac{e \xrightarrow{l} e'}{\langle \langle \mathsf{Run}\ m\ e, st \rangle, \mathcal{I}, \mathcal{O} \rangle \xrightarrow{l}_i \langle \langle \mathsf{Run}\ m\ e', st \rangle, \mathcal{I}, \mathcal{O} \rangle}$$

$$\textsc{IdealOutput}\ \frac{\langle \mathsf{Out}, v' \rangle \dashleftarrow_i \langle m, v \rangle \quad f_{\mathtt{output}}(v') = (\mathcal{O}, \epsilon)}{\langle \langle \mathsf{Run}\ m\ v, st \rangle, \mathcal{I}, \emptyset \rangle \xrightarrow{\epsilon}_i \langle \langle \mathsf{Idle}, st \rangle, \mathcal{I}, \mathcal{O} \rangle} \qquad \textsc{IdealExit}\ \frac{\langle \mathsf{St}, st' \rangle \dashleftarrow_i \langle m, v \rangle}{\langle \langle \mathsf{Run}\ m\ v, st \rangle, \mathcal{I}, \mathcal{O} \rangle \xrightarrow{\epsilon}_i \langle \langle \mathsf{Idle}, st' \rangle, \mathcal{I}, \mathcal{O} \rangle}$$

$\boxed{\langle \Sigma, \mathcal{I}, \mathcal{O} \rangle \xRightarrow{\tau}_i \langle \Sigma', \mathcal{I}', \mathcal{O}' \rangle}$ (**Real-world semantics** for replica $i$: Under global configuration $\Sigma$, input buffer $\mathcal{I}$ and output buffer $\mathcal{O}$, yields $\Sigma'$, $\mathcal{I}'$ and $\mathcal{O}'$ and returns trace $\tau$.)

$$\textsc{RealInput}\ \frac{\mathcal{I} \neq \emptyset \quad \pi_{\mathtt{input}}(\mathcal{I}) = (\overline{s}, \epsilon) \quad \langle \overline{s}, \overline{st} \rangle \dashrightarrow_i \langle m, \overline{e} \rangle}{\left\langle \overline{\langle \mathsf{Idle}, st \rangle}, \mathcal{I}, \mathcal{O} \right\rangle \xRightarrow{\epsilon}_i \left\langle \overline{\langle \mathsf{Run}\ m\ e, st \rangle}, \emptyset, \mathcal{O} \right\rangle} \qquad \textsc{RealStep}\ \frac{\overline{e} \xRightarrow{\tau} \overline{e'}}{\left\langle \overline{\langle \mathsf{Run}\ m\ e, st \rangle}, \mathcal{I}, \mathcal{O} \right\rangle \xRightarrow{\tau}_i \left\langle \overline{\langle \mathsf{Run}\ m\ e', st \rangle}, \mathcal{I}, \mathcal{O} \right\rangle}$$

$$\textsc{RealOutput}\ \frac{\langle \mathsf{Out}, \overline{v'} \rangle \dashleftarrow_i \langle m, \overline{v} \rangle \quad \pi_{\mathtt{output}}(\overline{v'}) = (\mathcal{O}, \tau)}{\left\langle \overline{\langle \mathsf{Run}\ m\ v, st \rangle}, \mathcal{I}, \emptyset \right\rangle \xRightarrow{\tau}_i \left\langle \overline{\langle \mathsf{Idle}, st \rangle}, \mathcal{I}, \mathcal{O} \right\rangle} \qquad \textsc{RealExit}\ \frac{\langle \mathsf{St}, \overline{st'} \rangle \dashleftarrow_i \langle m, \overline{v} \rangle}{\left\langle \overline{\langle \mathsf{Run}\ m\ v, st \rangle}, \mathcal{I}, \mathcal{O} \right\rangle \xRightarrow{\tau}_i \left\langle \overline{\langle \mathsf{Idle}, st' \rangle}, \mathcal{I}, \mathcal{O} \right\rangle}$$

interactiveinteractiveinteractive

Fig. 9: Ideal and real interactive MPC semantics.

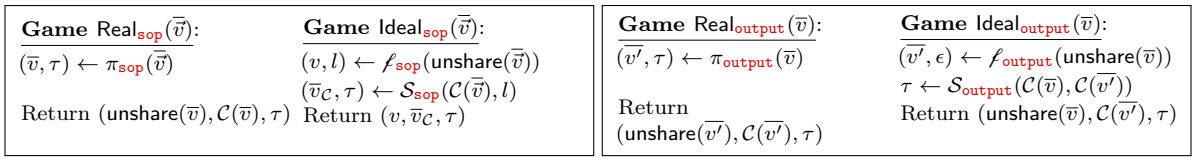


Fig. 10: Real and Ideal security games for MPC library security.

control the full secret-shared inputs and outputs. This allows for intermediate computations to have a slightly more relaxed notion of security, which facilitates reasoning over the security of composed secure operations.

We assume a linear secret-sharing scheme and a randomised procedure $\mathsf{share}(s) = \overline{s}$ that converts a value $s$ in its shared form $\overline{s}$; and its left-inverse $\mathsf{unshare}(\overline{s}) = s$ that converts $\overline{s}$ into the original value $s$. We denote $\mathcal{C}(\overline{s})$ as the subset of shares controlled by corrupted parties. In particular, we define input operations with no leakage: in the ideal world, $\mathscr{f}_{\mathtt{input}}(\overline{s}) = (\mathsf{unshare}(\overline{s}), \epsilon)$ simply computes the unshared value; in the real world, $\pi_{\mathtt{input}}(\overline{s}) = (\overline{s}, \epsilon)$ corresponds to the identity protocol. For the case of output operations: in the ideal world, $\mathscr{f}_{\mathtt{output}}(s) = (\mathsf{share}(s), \epsilon)$ simply shares the value; in the real world, $\pi_{\mathtt{output}}(\overline{s})$ runs a standard *resharing* protocol that re-randomizes a secret-shared value. Other secret operations, generically called `sop`, are left abstract.

Following [2], we define the security of operations (input is trivial since it has an empty trace) as follows:

**Definition 4 (MPC library security).** *The real MPC library is said to be a secure realisation of the ideal MPC library if there exist simulators $\mathcal{S}_{\mathtt{output}}$, and $\mathcal{S}_{\mathtt{sop}}$ (for any other secret operation), such that the experiments on the left- and right-hand side of Figure 10 are indistinguishable.*

### 5.4 Interactive MPC Programs

In order to reason about the security of MPC-based CRDT implementations, we adapt the security model from [2] to consider the parallel execution of multiple MPC programs that animate multiple CRDT replicas.

*Interactive MPC Semantics:* In order to reflect the CRDT security model (Section 4.3), we extend our MPC programming language and its semantics to support the specific input/output behavior of CRDTs. Figure 9 presents a small-step semantics that captures the interactive behavior of a replica $i$. Following [2], the small-step semantics has access to input ($\mathcal{I}$) and output ($\mathcal{O}$) channels that interact with the external environment and hold optional secret-shared values. An input is a specific CRDT operation ($\mathsf{query}/\mathsf{update}/\mathsf{propagate}/\mathsf{merge}$) and its additional arguments; only $\mathsf{query}/\mathsf{propagate}$ produce output (other CRDT operations do not need to block the output buffer). The environment is also responsible for passing messages (a propagate output to a merge input) among replicas. A replica-local configuration $\sigma = \langle \zeta, st \rangle$ is comprised of a program state $st$ and a call state $\zeta$ that mediates the execution of CRDT operations. A replica is either $\mathsf{Idle}$ or evaluating an expression $e$ in CRDT operation mode $m$ ($\mathsf{Run}\, m\, e$). A global configuration $\Sigma$ holds $\mathcal{P}_i$ local configurations for each MPC party executing the replica $i$ in the real world. Particular CRDT operations that interact with the user execute special `input` and `output` operations that convert a secret-shared value to a secret value and vice-versa.

Our richer structure on inputs and outputs (to control the scheduling of CRDT operations) is another difference to [2]. We abuse notation and assume that all inputs or outputs are represented in secret-shared form. We replicate public data across parties using zipping and unzipping operators:

an operation $\texttt{Op}\, p\, \overline{s}$ with public argument $p$ and secret-shared argument $\overline{s}$ can be "unzipped" to $\overline{\texttt{Op}\, p\, s}$ and "zipped" back to $\texttt{Op}\, p\, \overline{s}$.

### 5.5 MPC Security Model

We now propose a tailored security model for MPC-based CRDTs in Figure 11. Its main characteristic is that the experiment maintains a configuration $\Sigma_i$ to record the program state of each replica, which is defined as the multiple oracles are called, and processed via the step trigger. As such, in the real world, setInput writes a specific CRDT operation and its secret-shared arguments to the input buffer $\mathcal{I}$. getOutput retrieves the output of query/propagate operations, by checking output buffer $\mathcal{O}$. Oracle step animates the program for a particular replica. Following the interactive MPC semantics, it may read CRDT operations from $\mathcal{I}$ and set to evaluate the corresponding MPC program; perform a reduction in the MPC program; or exit evaluation, returning to idle mode or producing an output. Note that the communication between replicas is controlled by the environment, which is responsible for receiving the secret-shared result of propagate on one replica and forward a secret-shared merge request to the respective replica.

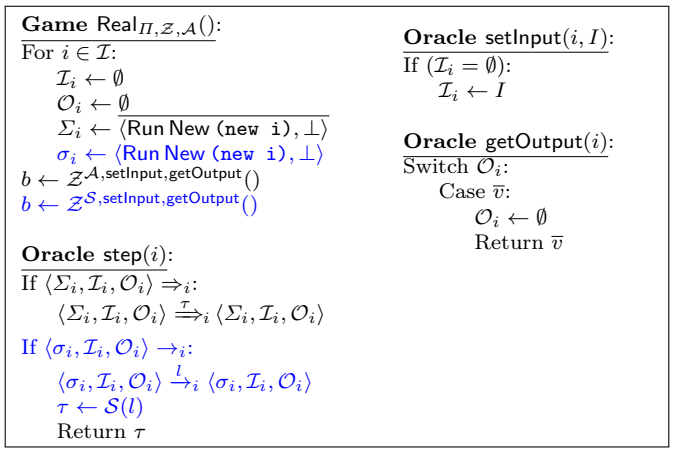


Fig. 11: Real and Ideal security games for MPC-based CRDTs. $\mathcal{A}$ has access to oracle step. Changes in blue reflect the Ideal world.

The ideal world is very similar to the real world; Figure 11 highlights the differences in blue. We maintain values $\sigma_i$ for every emulated replica, which are updated according to input $I$. Here, the simulator will observe the leakage of each operation, and must produce a simulated trace $t$ corresponding to each step.

Intuitively, CRDT operations are now described as sequences of operations where private data is processed according to the MPC semantics, and thus by an underlying MPC library. This allows us to borrow the security argument of [2], and state that the security of any CRDT construction written under such semantics is as secure as its underlying MPC library.

**Theorem 1.** *Under the assumption that the real MPC library is a secure realisation of the ideal MPC library, for any environment $\mathcal{Z}$ in Figure 11, the following is true*

$$\mathsf{Real}_{\mathcal{Z},\mathcal{A}}() \approx \mathsf{Ideal}_{\mathcal{Z},\mathcal{S}}()$$

The full proof can be found in Appendix A.1. The intuition is as follows. For a single replica instance, any CRDT implementation written under the MPC semantics will behave as a specific program in the language of [2]. As such, any advantage that allows $\mathcal{Z}$ to distinguish the experiment of Figure 11 can be used to build a distinguisher $\mathcal{Z}'$ against Theorem 1 of [2]. Since all $n$ replicas receive input shares and produce uniformly-distributed output shares, this is equivalent to executing the described experiment $n$ times in parallel.

### 5.6 MPC-based CRDT Security

We are now ready to prove the CRDT security of our MPC-based instantiation. This is not immediate for two main reasons. Going from Figure 11 to Figure 6 requires determining a specific simulation strategy, which establishes how the simulator handles propagate traces only having access to the sync operation at $\mathcal{F}$. Additionally, we have to argue that the *Ideal* small-step semantics behavior of Figure 11 is equivalent to the *Ideal* behavior over histories of events displayed by $\mathcal{F}$.

In terms of leakage, we give the most general definition for which our instantiation is secure. In practice, other higher-level leakage specifications could be considered, as exemplified in [1]. We define a leakage specification $\mathsf{L}(op, E)$ as $\mathsf{Pub}(op) \cup \{\mathsf{Pub}(st) \cup \mathsf{Pub}(res) \cup l \mid \forall st.\ E \approx st \wedge (\mathtt{apply}\ op\ st) \Downarrow_l res\}$. The intuition is that leakage should be defined directly over the history of events and the same for any state consistent with the history of events. Pub denotes the public parts as defined by the security type system. Note that, for sync, the leakage will be that of the operation defined in Definition 2.

Let $\Pi$ be a CRDT protocol defined by algorithms query, update, propagate and merge, constructed as a sequence of operations following the small-step semantics of Figure 9, and let $\mathcal{F}$ be its ideal counterpart.

**Theorem 2.** *Under the assumption that* propagate *has no leakage, if the behavior of $\Pi$ ensures functional correctness and communication correctness w.r.t. $\mathcal{F}$, then $\Pi$ securely realizes $\mathcal{F}$ according to Definition 3.*

The full proof can be read in Appendix A.2. We now detail the intuition. Given that our $\Pi$ is constructed as a sequence of small-step operations in the MPC language, the real world of Figure 6 is a strictly weaker version of the real world of Figure 11. This allows us to hop to a hybrid game, in which we have a functionality that follows an equivalent implementation of the CRDT following its ideal-world semantics. We now rely on the assumed functional and communication correctness properties to replace said functionality with the functionality following the idealized behavior of the CRDT (Figure 5). Concretely, $\mathcal{S}$ has to remember when propagates occur to trigger the correct sync queries at $\mathcal{F}$, and then rely on the small-step simulators to produce the sequence of traces $t$ corresponding to each CRDT operation.

### 5.7 Alternative Corruption Model

Observe that we are considering that a threshold of corrupt parties $\mathcal{C}_i$ per replica $i$. As such, security of our MPC library (Definition 4) requires that all states propagated from replica $i$ to replica $j$ have to be uniformly distributed, and thus produced by $\pi_{\texttt{output}}$. This prevents a trivial attack, in which a combination of corrupt parties in different replicas would allow an adversary to reconstruct shared data. Let the MPC at replica $i$ and $j$ allow for 1-out-of-3 corruptions, one can corrupt party $P_0$ in $i$ and $P_1$ in $j$, which gives him 2-out-of-3 shares when either one sends their shares to the other. Consequently, this requires the replica to *reshare* its data before all propagate operations, which entails an additional communication round.

```
type State_GC1 = Map 𝕀 (S Int)
data Query_GC1 r where Get_GC1 :: Query_GC1 (S Int)
data Update_GC1 = Inc_GC1 { inc_GC1 :: (S Int) }
type Message_GC1 = State_GC1
new_GC1 i = Map.empty
query_GC1 i Get_GC1 st = Map.foldr (+) (classify 0) st
update_GC1 i (Inc_GC1 n) st = Map.alter (Just . maybe n (n+)) i st
propagate_GC1 i j st_i = maybe Map.empty (Map.singleton i) (Map.lookup i st_i)
merge_GC1 i j st_i m_j = Map.unionWith max st_i m_j
```

```
type State_GC2 = Map 𝕀 (S Int,T)
data Query_GC2 r where Get_GC2 :: Query_GC2 (S Int})
data Update_GC2 = Inc_GC2 { inc_GC2 :: (S Int) }
type Message_GC2 = State_GC2
new_GC2 i = Map.empty
query_GC2 i Get_GC2 st = Map.foldr (\(n,_) -> n+) (classify 0) st
update_GC2 i (Inc_GC2 n) st = Map.alter (Just . maybe (n,startT) (\(n',t') -> (n + n',nextT  t'))) i st
propagate_GC2 i j st_i = maybe Map.empty (Map.singleton i) (Map.lookup i st_i)
merge_GC2 i j st_i m_j = Map.unionWith (maxOn snd) st_i m_j
```

Fig. 12: Two secure grow-only counter CRDTs, following [3].

This overhead can be avoided if one assumes the same corruption threshold $\mathcal{C}$ over all replicas, i.e. a corruption of computing party 0 in replica $i$ entails the corruption of party 0 in all other replicas. This also makes the proof of Theorem 2 easier, as performing operations on all replicas is now equivalent to interacting with a single MPC library that maintains the state of all replicas. This corruption model is consistent with a deployment where all replicas are emulated by servers in the same trust domain, e.g., cloud providers.

## 6 MPC-based CRDT implementations

To demonstrate our approach, this section discusses secure implementations of CRDTs from the literature.

### 6.1 Counters

*Grow-only counter:* Figure 12 presents two secure implementations of the grow-only counter CRDT. The first variant (`GC1`) is a direct implementation of the delta-based description from [3]. The only difference to the original non-secure version (Figure 3) is that we treat the local counter kept by each replica (and the global counter computed from those) as secure. Consequently, all operations over counter values, namely initialization, increment and comparison, need to be performed securely. The most expensive secure operation is `max`, which requires communication among MPC parties.

The second variant (`GC2`) is an optimization. Given the monotonic nature of the grow-only counter, we can avoid this secure computation by extending the state to include a per-replica public timestamp, known as a Lamport clock. The implementation guarantees that a greater timestamp corresponds to a greater counter value. Since timestamps can be inferred from the history of events, both variants have the same leakage, that can be succinctly characterized as the number of update operations performed at each replica.

*Positive-Negative counter:* Using two grow-only counters, we can construct a counter that supports both increment operations as shown in [43], with specification $\mathsf{F}_{\mathtt{PNC}}(i, \mathtt{Get}_{\mathtt{PNC}}, E) = \mathsf{F}_{\mathtt{GC2}}(i, \mathtt{Get}_{\mathtt{GC2}}, E) - \mathsf{F}_{\mathtt{GC2}}(i, \mathtt{Get}_{\mathtt{GC2}}, E)$. This is demonstrated in Figure 13.

```
type State_PNC = (State_GC2,State_GC2)
data Query_PNC r where Get_PNC :: Query_PNC (S Int)
data Update_PNC = Inc_PNC (S Int}) | Dec_PNC (S Int)
type Message_PNC = State_PNC
new_PNC i = (new_GC2 i,new_GC2 i)
query_PNC i Get_PNC (pst,nst) = p - n
  where p = query_GC2 i Get_GC2 pst
        q = query_GC2 i Get_GC2 nst
update_PNC i (Inc_PNC n) (pst,nst) = (update_GC2 i (Inc_GC2 n) pst,nst)
update_PNC i (Dec_PNC n) (pst,nst) = (pst,update_GC2 i (Inc_GC2 n) nst)
propagate_PNC i j (pst_i,nst_i) = (propagate_GC2 i j pst_i,propagate_GC2 i j nst_i)
merge_PNC i j (pst_i,nst_i) (pm_j,nm_j) = (merge_GC2 i j pst_i pm_j,merge_GC2 i j nst_i nm_j)
```

Fig. 13: Secure Positive-Negative Counter CRDT, following [43].

*Bounded counter:* In CRDTs, it is notoriously hard to preserve application invariants while avoiding synchronization [30]. One classical example of an application invariant is to guarantee that a counter never goes below some minimum value. Figure 14 demonstrates a secure bounded counter CRDT adapted from [6], where parties can transfer "decrementing rights" among them. The only difference is that our version, likewise our secure counter, avoids comparisons over secure values. Note that the CRDT initialization procedure receives the minimal value $k$. The specification is $\mathsf{F}_{\mathtt{BC}}(i, \mathtt{Get}_{\mathtt{BC}}, E) = \sum\{n \mid e \in E \wedge op(e) = \mathtt{Inc}_{\mathtt{BC}}\ n \wedge id(e) = i\} - \sum\{n \mid e \in E \wedge op(e) = \mathtt{Dec}_{\mathtt{BC}}\ n \wedge id(e) = i\} + \sum\{n \mid e \in E \wedge op(e) = \mathtt{Transf}_{\mathtt{BC}}\ n\ i\} + k$.

### 6.2 Maximum

The CRDT from Figure 15 keeps the maximum value, where `minBound` denotes the smallest possible value. Its specification is defined as $\mathsf{F}_{\mathtt{MAX}}(i, \mathtt{Get}_{\mathtt{MAX}}, E) = \mathsf{max}(\mathtt{minBound} \cup \{\mathtt{set}_{\mathtt{MAX}}(op(e)) | e \in E\})$. When a value is received, each replica must ensure that its state is updated if and only if the new value is greater than the one stored, regardless of how recent it is.

### 6.3 Polymorphic CRDTs

Container CRDTs, which are polymorphic over the type of elements, can be directly made to hold secure elements, keeping the container structure public.

*Registers:* Registers are classical CRDTs that maintain a general opaque value. Two different register specifications have been proposed [43]: the multi-value register (`MVR`) and the last-writer-wins (`LWW`) register. Figure 16 presents a secure implementation for the second; the first can be found in Appendix B.1. Note that their CRDT operations perform no secure computations, since their behavior is polymorphic over the secret values (of type `a`) stored by users.

For the `LWW` register, the only adaptation from the original proposal [43] has to do with the timestamps: instead of including a *now*() operation, we establish that the client is expected to generate timestamps. We make no assumption on timestamp uniqueness/causality; the interface may hide this to the user and simply provide, e.g., its wall clock. Its specification consists of an optional most recent value (using a global ordering on replica ids to guarantee uniqueness) seen as a set $(\mathsf{F}_{\mathtt{LWW}}(i, \mathtt{Get}_{\mathtt{LWW}}, E) = \{v | e \in E \wedge op(e) = \mathtt{Upd}\, v\, t \wedge (\nexists e' \in E.op(e') = \mathtt{Upd}\, v'\, t' \wedge (t, id(e)) < (t', id(e')))\})$.

For the `MVR` register, each replica keeps a multiset of values, each bound to a vector clock (a vector of Lamport clocks) as in [43]. Queries return a a multiset, implemented as a list sorted by vector clocks. Its specification can be defined as a multiset of most recent values $(\mathsf{F}_{\mathtt{MVR}}(i, \mathtt{Get}_{\mathtt{MVR}}, E) = \{\{v | e \in E \wedge \nexists e' \in E.e \prec e'\}\})$.

```
type State_BC = (S Int,Map (𝕀,𝕀) (S Int,T),Map 𝕀 (S Int,T))
data Update_BC = Inc_BC (S Int) | Dec_BC (S Int) | Transf_BC (S Int) 𝕀
data Query_BC r where Get_BC :: Query_BC (S Int)
type Message_BC = State_BC
new_BC i k = (classify k,Map.empty,Map.empty)
query_BC i Get_BC (k,r,u) = n where
    p = Map.foldrWithKey (\(k1,k2) (n,_) -> if k1==k2 then (n+) else id) k r
    n = Map.foldr (\(n,_) b -> (b - n) p u
update_BC i (Inc_BC n) (k,r,u) = (k,r',u) where
    r' = Map.alter (Just . (\(n',t') -> (n+n',nextT t')) . fromMaybe (classify 0,startT)) (i,i) r
update_BC i (Dec_BC n) st@(k,r,u) = (k,r,u') where
    lr = localRights i st
    u' = Map.alter (Just . (\(n',t') -> (if' (lr >= n) (n+n') n'},nextT t')) . fromMaybe (classify 0,startT) )
↪  i u
update_BC i (Transf_BC n j) st@(k,r,u) = (k,r',u) where
    lr = localRights i st
    r' = Map.alter (Just . (\(n',t') -> (if' (lr >= n) (n+n') n'},nextT t')) . fromMaybe (classify 0,startT) )
↪  (i,j) r
propagate_BC i j st_i = st_i
merge_BC i j (k,r_i,u_i) (_,r_j,u_j) = (k,r_i',u_i') where
    r_i' = Map.unionWith (maxOn snd) r_i r_j
    u_i' = Map.unionWith (maxOn snd) u_i u_j
localRights i (k,r,u) = p + rr - rs - n where
    rr = Map.foldrWithKey (\(k1,k2) (n,_) -> if k1==i then (n+) else id) (classify 0) r
    rs = Map.foldrWithKey (\(k1,k2) (n,_) -> if k2==i then (n+) else id) (classify 0) r
    p = maybe (classify 0) fst (Map.lookup (i,i) r)
    n = maybe (classify 0) fst (Map.lookup i u)
```

Fig. 14: Bounded counter specification, adapted from [6].

```
type State_MAX = S Int
data Query_MAX r where
  Get_MAX :: Query_MAX (S Int)
data Update_MAX = Set_MAX { set_MAX :: (S Int) }
type Message_MAX = State_MAX
new_MAX i = classify minBound
query_MAX i Get_MAX st = st
update_MAX i (Set_MAX n) st = max n st
propagate_MAX i j st = st
merge_MAX i j st_i m_j = max st\_i m\_j
```

Fig. 15: Secure maximum value CRDT.

*Other container CRDTs:* Many other container CRDTs, such as lists, maps and trees have been proposed in the literature [3, 29, 42]. Their MPC-based instantiation is similar to registers. A CRDT array implementation is shown in Appendix B.1.

### 6.4 Sets

*Grow-only set:* Sets are other classical container CRDTs. They are traditionally simpler than lists, since the ordering of elements does not need to be preserved; [29, 43] derive set constructions by extending the logic of counters. However, secure set CRDTs are harder to implement than secure lists, since they are no longer fully polymorphic and require comparison of elements, whose result affects the structure of the set.

Figure 17 presents a standard implementation of a state-based grow-only set [43], where the type of elements is secret. For simplicity, we encode the set as a list (regular Haskell sets requires ordering for efficient intermediate data structure), but order is not preserved. Instead, we only require equality on secret values, which implies that each set operation must perform a linear

```
type State_LWW a = Maybe (a,LC)
data Query_LWW a r where
  Get_LWW :: Query_LWW a (Maybe a)
data Update_LWW a = Upd_LWW a T
type Message_LWW a = State_LWW a
new_LWW i = Nothing
query_LWW i Get_LWW st = fmap fst st
update_LWW i (Upd_LWW v t) st = let st' = Just (v,LC t i)
  in maxOn (fmap snd) st st'
propagate_LWW i j st_i = st_i
merge_LWW i j st_i m_j = maxOn (fmap snd) st_i m_j
```

Fig. 16: Secure last-writer-wins register CRDT, following [43].

```
type State_GSet a = [S a]
data Update_GSet a = Add_GSet { add_GSet :: S a }
data Query_GSet a r where
    GetAll_GSet :: Query_GSet a [S a]
    Exists_GSet :: S a -> Query_GSet a (S Bool)
type Message_GSet a = State_GSet a
new_GSet i = []
query_GSet i GetAll_GSet st = st
query_GSet i (Exists_GSet x) ys = foldr (\y b -> (|| (x == y) b)}) (classify False) ys
update_GSet i (Add_GSet x) st = insert_GSet x st
propagate_GSet i j st = st
merge_GSet i j st_i m_j = foldr insert st_i m_j
insert x ys = if declassify b then x : ys else ys
    where b = foldr (\y b -> (|| (x == y) b)) (classify False) ys
```

Fig. 17: Secure state-based grow-only set, following [43].

traversal. We exemplify two operations on the set: one returns the whole set, and the other tests if an element exists in the set. The most relevant detail is the `insert` auxiliary function: in order to decide if the new element shall be added to the set or not, we need to declassify the result of the secure equality comparisons. Note that, in this setting, we cannot perform a secure conditional because the result of the comparison affects the public list structure. To minimize leakage, we avoid declassifying the result of all equality comparisons, and reveal only if each inserted element already exists in the set. The specification is defined as $\mathsf{F}_{\mathtt{GSet}}(i, \mathtt{GetAll}_{\mathtt{GSet}}, E) = \{\mathtt{add}_{\mathtt{GSet}}(op(e)) | e \in E\}$ and $\mathsf{F}_{\mathtt{GSet}}(i, \mathtt{Exists}_{\mathtt{GSet}}\, x, E) = \exists v. v \in \mathsf{F}_{\mathtt{GSet}}(i, \mathtt{GetAll}_{\mathtt{GSet}}, E) \wedge v = x$.

*Other set CRDTs:* Similar to counters, our grow-only set construction can be generalized to build more general sets. Nonetheless, it is not fully secure, as merging leaks element comparisons. Balancing such leakage with efficiency is precisely the craft of MPC, and the tradeoffs have been vastly explored in the context of implementing efficient versions of classical algorithms using MPC [1, 49]. The included technical report discusses some possible set constructions with no leakage.

## 7 Experimental Evaluation

We now present the results of an experimental evaluation of the CRDT constructions defined in Section 6.

### 7.1 System Implementation

For the evaluation, we implemented the secure CRDTs as a `Java` Library. This library is deployed as a CRDT replica, that consists of independent MPC parties to store secrets and evaluate queries.

This deployment follows the execution model outlined in Section 5.1. For the computation of MPC protocols, we used an implementation of the Sharemind protocols [13]. These protocols use additive secret sharing scheme and are optimized for a static three-party setting with an honest-but-curious adversary. As such, each of our CRDT replicas will consist of three independent parties. To illustrate the associated performance cost, we also implemented the same CRDTs without any security measures. We denote these as *baseline* implementations.

The Sharemind protocol suite provides an interface similar to the abstract interface defined in Section 5.2. More specifically, it provides methods to `share`, `unshare` and `declassify` secrets; protocols to multiply secrets, and protocols for equality and ordering comparison. Most of these protocols require multiple communication rounds, with the $\geq$ protocol having the highest number of rounds and secret multiplication requiring only a single communication round. The addition of secrets is the only operation that can be done without any communication.

### 7.2 Experimental Setup

The evaluation was conducted on four machines, each with two Intel(R) Xeon(R) Silver 4210 CPU @ 2.20 GHz and 256 GB of RAM. The cluster consisted of a single CRDT replica, split into three separate parties connected via Ethernet 10 Gigabit, communicating via TCP. We used the Python Locust framework to evaluate the performance of our system. Using this framework, we implemented workloads for CRDTs and made them publicly available. The benchmark client was deployed in the cluster on a dedicated host. The network latency between all hosts was $\sim 200\mu$s.

### 7.3 Experimental Workloads

We measure the throughput and latency of the `update` and `query` operations for the secure CRDTs defined in Section 4. To isolate the overhead of MPC protocols, we used dedicated workloads for each CRDT construction. As a reference, we also implemented a baseline system for each CRDT that stores the data as plaintext.

*`LWW` and `Max` CRDTs:* The workloads for these CRDTs start by initializing a replica with a random value. The evaluation of the `update` operation consists on inserting random values, and the evaluation of the `query` operation consists on the client retrieving the latest value from a replica.

*Counter CRDTs:* We implemented three distinct workloads for the `GC2`, `PNC`, and `BC` CRDTs. The workloads have the same setup, and initialize the counter at 1. The `update` operation increments the counter by a single value for all CRDTs. However, the `PNC` and `BC` workloads differ as the `update` operation can also decrement the counter. This is relevant for `BC`, as the operation uses several MPC protocols to decrement a counter, only if it has enough "decrement rights".

*GSet:* The performance of the secure `GSet` CRDT depends on the set size. The workload of the `update` operation consists in starting with an empty set and measuring the overhead of adding a single random value. The workload of the `query` operation initializes a set with a fixed size, and measures the overhead of testing set membership of a random value. For both operations, we evaluate sets sizes from $2^3$ to $2^6$ elements.

We also compare the performance of our MPC constructions with the specialized secure CRDT constructions presented by *Barbosa et al.* [8]. They leverage standard encryption and deterministic encryption techniques to implement `LWW` and `Set`, respectively, and Paillier homomorphic encryption [34] to implement `GC2` and `PNC`. Given that none of these cryptographic mechanisms allow for replica-side comparison over ciphertexts, they have no similar implementation of `MAX` or `BC`. They

include a bounded counter construction, but since Paillier does not allow for ciphertext comparison, they delegate invariant verification to the client, which is functionally different from our `BC`.

| CRDT | Operation | Baseline | *[8]* | MPC |
|---|---|---|---|---|
| LWW | Query | 1517 | 1516 | 1505 |
| | Update | 1513 | 1512 | 1514 |
| MAX | Query | 1518 | *n/a* | 1503 |
| | Update | 1514 | *n/a* | 9 |
| GC2 | Query | 1519 | 1519 | 1499 |
| | Update | 1515 | 1515 | 1516 |

| CRDT | Operation | Baseline | *[8]* | MPC |
|---|---|---|---|---|
| PNC | Query | 1517 | 1518 | 1497 |
| | Update | 1513 | 1513 | 1516 |
| BC | Query | 1516 | *n/a* | 1233 |
| | Increment | 1516 | *n/a* | 1378 |
| | Decrement | 1516 | *n/a* | 9 |

Table 1: Average throughput of `update` and `query` operations for the baseline and two versions of secure CRDTs.

### 7.4 Benchmark Results

Table 1 shows the maximum throughput achieved by our MPC constructions, the plaintext baseline and the constructions of [8]. We evaluated the performance of all CRDTs by measuring their `update` and `query` workloads for an increasing number of clients, from 1 to 64. Across all MPC constructions presented in this paper, the peak troughput was achieved by `GC2`, with 1519 ops/s for 64 concurrent clients. As the number of clients increases, the performance of the MPC constructions begins to decline due to the overhead of data resharing among parties. Overall, the MPC CRDT operations presented in Table 1 have an overhead of less than $\sim 2\%$ when compared to the baseline. Detailed results for the different number of clients are presented in Appendix D.

It's important to highlight the overhead of the `update` operation in the `MAX` CRDT. This operation is $\sim 168$ times slower than the baseline as shown in Table 1 and stabilizes at $\sim 9$ ops/s for 2 concurrent clients. For any number of concurrent clients greater than 2, the request latency increases significantly. This overhead is to be expected, as the `update` operation uses MPC protocols for `equality` comparison, `greater than or equal to` comparison, and secret multiplication.

The `update` operation of the `BC` CRDT also differs in performance to the other counters. As shown in Table 1 the secure `increment` operation throughput decreases by $\sim 9\%$ in comparison to the baseline as it does not require any MPC protocol. However, the secure `decrement` operation uses the `equality` and the `greater than or equal to` MPC protocols to ensure that the counter does not decrease bellow its lower bound. As such, the maximum throughput is $\sim 9$ ops/s.

Across all CRDTs, the MPC constructions have comparable throughput to the constructions in [8]. The largest difference is present in the `PNC` *update* operation, with a throughput decrease of $\sim 1.4\%$. Due to the limitations of the underlying cryptographic mechanisms, `MAX` and `BC` CRDTs are not made available in the work of [8].

Finally, Table 2 presents the throughput of the `update` and `query` operations of `GSet`. The evaluation was done with a single client. The secure `update` operation has a maximum throughput of $\sim 9$ ops/s for the smallest set size, and decreases to $\sim 2$ ops/s for a set with 64 elements. In contrast, the baseline `update` operation has a consistent throughput of $\sim 40$ ops/s and the baseline `query` operation has $\sim 24$ ops/s. Since [8] is using deterministic encryption, their performance is predictably similar to that of the baseline. On the other hand, it has significantly more leakage, as

`update` operations reveal which elements are duplicate, and `query` operations reveal which element was retrieved. Our solutions prevent this leakage via replica-side comparisons of private data via MPC.

| SET CRDT Operation | | Set Size | | | |
|---|---|---|---|---|---|
| | | 8 | 16 | 32 | 64 |
| MPC | 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 |
| [8] | Query | 23.80 | 23.80 | 23.78 | 23.73 |
| | Update | 24.11 | 24.09 | 24.08 | 24.05 |

Table 2: Average throughput of update and query operations for baseline and secure `GSet` CRDT with fixed set sizes.

### 7.5 Discussion

When compared to the established baseline, all experimental results show that the majority of secure MPC construction have an overhead of 2% for both `update` and `query` operations. The slowest constructions are the `MAX update`, `GSET`, and the `BC decrement`. Notably, the `BC decrement` operation is $\sim 168\times$ slower than the baseline. This overhead results from multiple factors, including our preliminary implementation of the secure CRDTs that can be optimized and the underlying MPC protocols. The feasibility of our MPC-based approach is further supported by our comparison to the secure CRDT protocols of [8], as our design allows for complex replica-side computations over private data without meaningfully sacrificing performance.

*Optimizing CRDT Constructions:* The current implementation of the secure CRDT constructions is a prototype. It can be optimized to reduce the number of MPC protocols per operation. For example, the `MAX update` operation uses at least 3 MPC protocols that have a high number of communication rounds. This can be reduced at least to 2, thus reducing network bandwidth usage. Additional optimizations can be achieved by designing specialized MPC protocols for CRDTs.

*Multiparty Protocols:* We can improve the performance of our constructions by using different MPC protocols. We used the Sharemind protocols which are optimized for data analysis in the three-party setting, but our contributions are independent from the underlying protocols. The protocols can be replaced by optimized versions of said protocols [4], or by using a different class of protocols such as function secret sharing that have a constant number of communication rounds [14].

The results of this preliminary experimental evaluation, suggest the practical applicability of our theoretical contributions. Secure CRDTs can have throughput similar to a baseline system by minimizing the number of MPC protocols, while ensuring data confidentiality.

## 8 Related Work

*Programming Languages:* There is a long line of research on language-based secure compilation [35], and on the particular compilation of regular programs to MPC protocols [2], with the proposal of a plethora of high-level languages and compilation frameworks [22]. The essential concepts are summarized in our MPC language. On the other side, there is a growing interest in language-based approaches to the design and analysis of CRDTs, including the formal reasoning about specifications and implementations [20, 31] adopted in this paper. A few recent approaches have proposed to simplify the design of replicated data types, allowing some synchronization [30] or finer control over consistency restoration [18]. However, none of these approaches considers the privacy of the data shared among the replicas.

*Secure CRDTs:* Secure CRDTs were first formalized by *Barbosa et al.* [8]. The authors present constructions for secure registers, counters and set CRDTs that leverage deterministic encryption and partial homophobic encryption schemes. Most of the existing work in this area is adjacent to *Barbosa et al.* seminal paper. Recent work by *Jannes et al.* [23] uses standard cryptographic techniques to ensure confidentiality and authentication of CRDTs, but their approach is also restricted by not allowing any operations for merging encrypted data. *Cachin et al.* [16] proposed Authenticated Data Types (ADTs) for authenticated data outsourcing in a single-server/single-client setting. Snapdoc [28] presents a solution for collaborative document edition with history-privacy.

*Decentralized Confidential Computation:* Multi-party protocols have been used as a solution for confidential computation on several systems. SDB [48] is a relational database that uses a two-party protocol suite optimized for relational queries and operation interoperability. This system is capable of supporting a wide-range of complex analytical queries. More recently, SMCQL [9] and ConClave [47] have presented optimized systems for big data workloads in a honest-but-curious model and a three party setting. Senate [36] has gone one step further and presented a platform for collaborative analytics secure against active adversaries for any number of parties. Multi-party protocols have also extended into machine learning with Cerebro [51], a platform for multi-party collaborative learning. NoSQL operations can also be fully supported with an acceptable performance and security trade-off as shown by d'Artangnan [38], the first decentralized NoSQL confidential database. This solution is based on the Sharemind protocol suite. These protocols are also used in the commercial Sharemind platform that provides secure data analysis [13]. MPSaaS (MPC system-as-a-service) [7] proposes a system for deploying large-scale MPC in a decentralized manner, supported by optimized MPC protocols that are efficient for a large number of participants and allowing parties to dynamically enroll in MPC computations. However, none of their synchronization and coordination among parties makes use of CRDTs.

*Secure Cloud 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 [37]. More cloud native solutions such has BlueSky [32] have emerged to provide strong consistency and availability. However, cloud systems can also be affected by loss of availability and data corruption. DepSky [12] 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 system provides confidential computation or eventual consistency.

## 9 Conclusion and Future Work

This paper proposes the first approach to general-purpose implementations of secure CRDTs using MPC. We propose an MPC language to facilitate the development of such protocols, a proof that attests to the security of our constructions under such language, and several MPC-based CRDT constructions.

Our work includes an open-source implementation, and our experimental results suggest practical feasibility of this approach, with considerable room for future improvements and optimizations.

# Bibliography


[1] J. B. Almeida, M. Barbosa, G. Barthe, H. Pacheco, V. Pereira, and B. Portela. Enforcing ideal-world leakage bounds in real-world secret sharing MPC frameworks. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 132–146. IEEE, 2018.

[2] J. C. B. Almeida, M. Barbosa, G. Barthe, H. Pacheco, V. Pereira, and B. Portela. A formal treatment of the role of verified compilers in secure computation. *Journal of Logical and Algebraic Methods in Programming*, 125:100736, 2022.

[3] P. S. Almeida, A. Shoker, and C. Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173, 2018.

[4] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. 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.

[5] D. W. Archer, D. Bogdanov, Y. Lindell, L. Kamm, K. Nielsen, J. I. Pagter, N. P. Smart, and R. N. Wright. From keys to databases—real-world applications of secure multi-party computation. *The Computer Journal*, 61(12):1749–1771, 2018.

[6] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 31–36. IEEE, 2015.

[7] A. Barak, M. Hirt, L. Koskas, and Y. Lindell. An end-to-end system for large scale p2p mpc-as-a-service and low-bandwidth mpc for weak participants. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 695–712, 2018.

[8] M. Barbosa, B. Ferreira, J. Marques, B. Portela, and N. Preguiça. Secure Conflict-free Replicated Data Types. In *International Conference on Distributed Computing and Networking 2021*, pages 6–15, 2021.

[9] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers. SMCQL: Secure Querying for Federated Databases, 2016.

[10] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513, 1990.

[11] F. Benhamouda and H. Lin. k-round MPC from k-round OT via garbled interactive circuits. *Cryptology ePrint Archive*, 2017.

[12] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Trans. Storage*, 9(4), nov 2013.

[13] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In S. Jajodia and J. Lopez, editors, *Computer Security - ESORICS 2008*, pages 192–206, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[14] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. Cryptology ePrint Archive, Paper 2018/707, 2018. `https://eprint.iacr.org/2018/707`.

[15] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–284, 2014.

[16] C. Cachin, E. Ghosh, D. Papadopoulos, and B. Tackmann. Stateful Multi-client Verifiable Computation. In B. Preneel and F. Vercauteren, editors, *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.

[17] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[18] K. De Porre, C. Ferreira, N. Preguiça, and E. Gonzalez Boix. Ecros: building global scale systems from sequential code. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[19] G. Elkoumy, S. A. Fahrenkrog-Petersen, M. Dumas, P. Laud, A. Pankova, and M. Weidlich. Secure multi-party computation for inter-organizational process mining. In *Enterprise, Business-Process and Information Systems Modeling*, pages 166–181. Springer, 2020.

[20] V. B. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[21] A. Gotsman and H. Yang. Composite replicated data types. In *European Symposium on Programming Languages and Systems*, pages 585–609. Springer, 2015.

[22] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. Sok: General purpose compilers for secure multi-party computation. In *2019 IEEE symposium on security and privacy (SP)*, pages 1220–1237. IEEE, 2019.

[23] K. Jannes, B. Lagaisse, and W. Joosen. Secure replication for client-centric data stores. In *Proceedings of the 3rd International Workshop on Distributed Infrastructure for the Common Good*, pages 31–36, 2022.

[24] G. Kaki, S. Priya, K. Sivaramakrishnan, and S. Jagannathan. Mergeable replicated data types. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[25] M. Kleppmann and P. Alvaro. Research for practice: Convergence. *Commun. ACM*, 65(11):104–106, oct 2022.

[26] M. Kleppmann and A. R. Beresford. Automerge: Real-time data sync between edge devices. In *1st UK Mobile, Wearable and Ubiquitous Systems Research Symposium (MobiUK 2018)*, pages 101–105, 2018.

[27] M. Kleppmann, A. Bieniusa, and M. Shapiro. CRDT Tech Implementations. `https://crdt.tech/implementations`.

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

[29] A. Leijnse, P. S. Almeida, and C. Baquero. Higher-order patterns in replicated data types. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–6, 2019.

[30] N. V. Lewchenko, A. Radhakrishna, A. Gaonkar, and P. Černỳ. Sequential programming for replicated data stores. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–28, 2019.

[31] H. Liang and X. Feng. Abstraction for conflict-free replicated data types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 636–650, 2021.

[32] Michael Vrable and Stefan Savage and Geoffrey M. Voelker. BlueSky: A Cloud-Backed File System for the Enterprise. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, San Jose, CA, Feb. 2012. USENIX Association.

[33] P. Nicolaescu, K. Jahns, M. Derntl, and R. 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.

[34] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[35] M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys (CSUR)*, 51(6):1–36, 2019.

[36] R. Poddar, S. Kalra, A. Yanai, R. Deng, R. A. Popa, and J. 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.

[37] R. Pontes, D. Burihabwa, F. Maia, J. Paulo, V. Schiavoni, P. Felber, H. 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.

[38] R. Pontes, F. Maia, R. Vilaça, and N. Machado. d'Artagnan: A Trusted NoSQL Database on Untrusted Clouds. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 61–6109, 2019.

[39] N. Preguiça. Conflict-free replicated data types: An overview. *arXiv preprint arXiv:1806.10254*, 2018.

[40] N. Preguiça, C. Baquero, and M. Shapiro. Conflict-free replicated data types (crdts). In S. Sakr and A. Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer International Publishing, 2019.

[41] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy*, pages 655–670. IEEE, 2014.

[42] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[43] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical report, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[44] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[45] M. Sporny, D. Buchner, and O. Steele. Confidential storage 0.1. Unofficial draft, W3C, Aug. 2021. https://identity.foundation/confidential-storage.

[46] M. Sulzmann, M. M. Chakravarty, S. P. Jones, and K. Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, 2007.

[47] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros. Conclave: Secure Multi-Party Computation on Big Data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[48] W. K. Wong, B. Kao, D. W. L. Cheung, R. Li, and S. M. Yiu. Secure Query Processing with Data Interoperability in a Cloud Database Environment. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1395–1406, New York, NY, USA, 2014. Association for Computing Machinery.

[49] Q. Ye and B. Delaware. Oblivious algebraic data types. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022.

[50] P. Zeller, A. Bieniusa, and A. Poetzsch-Heffter. Formal specification and verification of crdts. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 33–48. Springer, 2014.

[51] W. Zheng, R. Deng, W. Chen, R. A. Popa, A. Panda, and I. Stoica. Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2723–2740. USENIX Association, 2021.

# A Security of MPC-based CRDTs

## A.1 Proof of Theorem 1

*Proof.* Let $N$ denote the cardinality of $\mathcal{I}$. We assume that replicas are identified as values in $[1 \dots N]$. Let $\mathsf{Adv}^{\mathsf{mpc}}_{\mathcal{B}}$ denote the advantage of $\mathcal{B}$ to distinguish the behavior of the MPC library according to Theorem 1 in [2]. Consider a fixed, reactive program $P_\pi$ that conditionally executes query, update, propagate or merge in a loop, written using the small-step operations of the MPC library. Our reasoning follows a hybrid argument, where we will iteratively replace calls to replicas $i \in [1 \dots N]$ using *Real* MPC semantics, by calls to replicas using *Ideal* MPC semantics.

First consider $\mathsf{G1}_{\mathcal{Z},\mathcal{A},\mathcal{S}}$, which we define to be exactly the same as the real world of Figure 11, but where the behavior of all calls to replica 1 follows their ideal counterparts. We argue the similarity of these games by constructing an adversary $\mathcal{B}$ against Theorem 1 in [2].

Adversary $\mathcal{B}$ simulates the environment of $\mathsf{G}_0^{\mathcal{Z},\mathcal{A},\mathcal{S}}()$ as follows. For Theorem 1, it selects program $P_\pi$, which will be used to process all calls to replica 0. Specifically, $\mathcal{B}$ has to rewrite calls to specific CRDT operations to their functionally equivalent counterparts in $P_\pi$. This essentially entails receiving setInput calls from $\mathcal{Z}$, and depending on the concrete CRDT operation, triggering its execution on $P_\pi$ over the same input. For all calls to other replicas, $\mathcal{B}$ follows exactly the real-world execution of the protocol. Observe that, depending on if we are in the Real or Ideal world of Theorem 1, this exactly matches either $\mathsf{Real}^{\mathcal{Z},\mathcal{A}}()$, or $\mathsf{G}_0^{\mathcal{Z},\mathcal{A},\mathcal{S}}()$, and as such

$$|\Pr[\mathsf{Real}^{\mathcal{Z},\mathcal{A}}() \Rightarrow \mathsf{T}] - \Pr[\mathsf{G}_0^{\mathcal{Z},\mathcal{A},\mathcal{S}}() \Rightarrow \mathsf{T}]| \leq \mathsf{Adv}^{\mathsf{mpc}}_{\mathcal{B}}$$

We can now repeat this argument $N$ times for $\mathsf{G}_{i-1}^{\mathcal{Z},\mathcal{A},\mathcal{S}}()$ to $\mathsf{G}_{i}^{\mathcal{Z},\mathcal{A},\mathcal{S}}()$. All replicas [1..$i$-1] exactly match the ideal behavior, all replicas [$i$+1..$N$] exactly match the real behavior, and the role of $\mathcal{B}$ is to replace the $i$-th instance of replica calls to either their real or ideal version. Given that $\mathsf{G}_{N}^{\mathcal{Z},\mathcal{A},\mathcal{S}}()$ exactly matches $\mathsf{Ideal}_{\mathcal{Z},\mathcal{S}}()$, for negligible function $\mu()$ we have that

$$\begin{aligned}|\Pr[\mathsf{Real}^{\mathcal{Z},\mathcal{A}}() \Rightarrow \mathsf{T}] - \Pr[\mathsf{Ideal}^{\mathcal{Z},\mathcal{S}}() \Rightarrow \mathsf{T}]| \leq & N \cdot \mathsf{Adv}^{\mathsf{mpc}}_{\mathcal{B}} \\ |\Pr[\mathsf{Real}^{\mathcal{Z},\mathcal{A}}() \Rightarrow \mathsf{T}] - \Pr[\mathsf{Ideal}^{\mathcal{Z},\mathcal{S}}() \Rightarrow \mathsf{T}]| \leq & \mu()\end{aligned}$$

and Theorem 1 follows.

## A.2 Proof of Theorem 2

*Proof (Proof of Theorem 2).* Our proof is a sequence of four games, described as follows.

First, we consider $\mathsf{G}_0^{\mathcal{Z},\mathcal{A}}()$, which is a rewrite of $\mathsf{Real}^{\mathcal{Z},\mathcal{A}}()$, but where each instance of query, update, propagate and merge is instead a sequence of small step executions, written with our real MPC semantics. This game displays exactly the same behavior as $\mathsf{Real}^{\mathcal{Z},\mathcal{A}}()$ by construction, hence

$$|\Pr[\mathsf{Real}^{\mathcal{Z},\mathcal{A}}() \Rightarrow \mathsf{T}] - \Pr[\mathsf{G}_0^{\mathcal{Z},\mathcal{A},\mathcal{S}}() \Rightarrow \mathsf{T}]| = 0$$

We now describe $\mathsf{G}_1^{\mathcal{Z},\mathcal{A}}()$ which strictly replaces each of these operations by ideals ones of the MPC library. We argue the similarity of these games by constructing a distinguisher $\mathcal{B}$ against

Theorem 1. Let $\mathsf{Adv}^{\mathsf{cmpc}}_{\mathcal{B}}$ denote the advantage of $\mathcal{B}$ to distinguish the behavior of the MPC semantics according to Theorem 1.

The behavior of $\mathcal{B}$ is simply to select inputs in Figure 11 according to the CRDT operation being performed, calling the exact number of steps associated with said operations until the MPC semantics reaches an $\mathsf{Idle}$ state, and then either returning the outputs (for $\mathsf{query}$ operations), or storing them in the communication channel $C_{i,j}$ (for $\mathsf{propagate}$ operations). Observe that, depending on if we are in the Real or Ideal world of Theorem 1, this exactly matches either $\mathsf{G}_1^{\mathcal{Z},\mathcal{A}}()$, or $\mathsf{G}_0^{\mathcal{Z},\mathcal{A}}()$, and as such

$$|\Pr[\mathsf{G}_0^{\mathcal{Z},\mathcal{A}}() \Rightarrow \mathsf{T}] - \Pr[\mathsf{G}_1^{\mathcal{Z},\mathcal{A}}() \Rightarrow \mathsf{T}]| \leq \mathsf{Adv}^{\mathsf{cmpc}}_{\mathcal{B}}$$

In $\mathsf{G}_2^{\mathcal{Z},\mathcal{S},\mathcal{F}}()$ we restructure our setting, but now considering functionality $\mathcal{F}_{\mathsf{mpc}}$ of Figure 18 and simulator $\mathcal{S}_{\mathsf{mpc}}$ of Figure 19. For $\mathcal{F}_{\mathsf{mpc}}$, we slightly abuse the notation of $\not{\ell}$ to correspond to the sequence of ideal semantics operations corresponding to that specific operation. Our simulator makes use of $\mathcal{S}$, which given leakage $l$ sequentially calls the simulators for the ideal semantics, producing an aggregated trace $t$. The only caveat is at $\mathsf{propagate}$, where the same simulator must produce a trace without any input leakage. $\mathcal{S}_{\mathsf{mpc}}$ is simply maintaining a counter for the state of each replica in $c_i$, stored upon $\mathsf{propagate}$ and used for $\mathsf{merge}$, and calling $\mathcal{S}$ whenever it needs to present some execution trace. As such, each $\mathsf{update}$ and $\mathsf{query}$ operations corresponds to a call to $\mathcal{F}_{\mathsf{mpc}}$, and a call to the same $\mathcal{S}$ as in $\mathsf{G}_1^{\mathcal{Z},\mathcal{A}}()$ to produce $t$. Calls to $\mathsf{propagate}$ and $\mathsf{merge}$ work slightly different, as instead of storing the result of $\mathsf{propagate}$ in $C_{i,j}$, instead just a counter is stored, and $\mathsf{propagate}$ is done when $\mathsf{merge}$ happens, over the exact same state (identified by $c_i$). This is functionally equivalent, as $\mathcal{Z}$ does not have access to $C_{i,j}$. As such, the only difference between $\mathsf{G}_2^{\mathcal{Z},\mathcal{S},\mathcal{F}}()$ and $\mathsf{G}_1^{\mathcal{Z},\mathcal{A}}()$ is that the trace of $\mathsf{propagate}$ must be simulated without any leakage, which is the case by assumption, and as such

$$|\Pr[\mathsf{G}_1^{\mathcal{Z},\mathcal{A}}() \Rightarrow \mathsf{T}] - \Pr[\mathsf{G}_2^{\mathcal{Z},\mathcal{S},\mathcal{F}}() \Rightarrow \mathsf{T}]| = 0$$

**proc** $\mathsf{init}()$:
For $i \in \mathbb{I}$:
  $c_i \leftarrow 0$; $\mathsf{st}_i[0] \leftarrow \epsilon$

**Environment $\mathcal{Z}$ interface**

**proc** $\mathsf{write}(i, \mathsf{op})$:
$c \leftarrow c_i$
$c_i \leftarrow c_i + 1$
$(\mathsf{st}_i[c_i], l) \leftarrow \not{\ell}(\mathsf{update}, i, \mathsf{op}, \mathsf{st}_i[c])$
Return $l$

**proc** $\mathsf{read}(i, \mathsf{op}))$:
$(v, l) \leftarrow \not{\ell}(\mathsf{query}, i, \mathsf{op}, \mathsf{st}_i[c_i])$
Return $(v, l)$

**Adversary $\mathcal{S}$ interface**

**proc** $\mathsf{sync}(i, j, c)$:
$(v, \epsilon) \leftarrow \not{\ell}(\mathsf{propagate}, i, j, \mathsf{st}_i[c])$
$(\mathsf{st}[c_j], l) \leftarrow \not{\ell}(\mathsf{merge}, i, j, v)$
Return $l$

Fig. 18: Functionality $\mathcal{F}_{\mathsf{mpc}}$, working with ideal MPC semantics.

In $\mathsf{G}_3^{\mathcal{Z},\mathcal{S},\mathcal{F}}()$ we finally change $\mathcal{F}_{\mathsf{mpc}}$ to behave just like $\mathcal{F}$ in Figure 5. $\mathsf{update}$ stores the event within $\mathcal{F}$, $\mathsf{query}$ executes $\mathsf{F}$ over the set of events, and the $\mathsf{merge} \cdot \mathsf{propagate}$ is replaced with the corresponding communication specification.

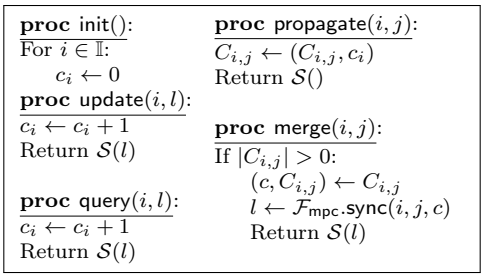


Fig. 19: Simulator $\mathcal{S}_{\mathsf{mpc}}$. Procedures refer to its behavior upon being called on the respective inputs.

```
type State_MVR a = Map VC a
data Query_MVR a r where
  Get_MVR :: Query_MVR [a]
data Update_MVR a = Upd_MVR a
type Message_MVR a = State_MVR a
new_MVR i = Map.empty
query_MVR i Get_MVR st = Map.elems st
update_MVR i (Upd_MVR v) st = merge' st (Map.singleton (incVC i (maxVCs (Map.keys st))) v)
propagate_MVR i j st_i = st_i
merge_MVR i j st_i m_j = merge' st_i m_j
merge' st1 st2 = Map.union st1' st2' where
    st1' = Map.filterWithKey (\vc v -> all (not . (leVC vc)) (Map.keys st2)) st1
    st2' = Map.filterWithKey (\vc v -> all (not . (leVC vc)) (Map.keys st1)) st2
```

Fig. 20: Secure multi-value register CRDT, following [43].

The change in behavior from $\mathsf{G}_3^{\mathcal{Z},\mathcal{S},\mathcal{F}}()$ to $\mathsf{G}_2^{\mathcal{Z},\mathcal{S},\mathcal{F}}()$ exactly matches the relation described in the Communication and Functional correctness properties of Definitions 2 and 1 respectively. The first ensures that states are equivalent at all times to events stored and synchronized, while the latter ensures that the output from write operations over events is also equivalent to its query operation over equivalent states. Hence

$$|\Pr[\mathsf{G}_2^{\mathcal{Z},\mathcal{S},\mathcal{F}}() \Rightarrow \mathsf{T}] - \Pr[\mathsf{G}_3^{\mathcal{Z},\mathcal{S},\mathcal{F}}() \Rightarrow \mathsf{T}]| = 0$$

Finally, $\mathsf{G}_3^{\mathcal{Z},\mathcal{S},\mathcal{F}}()$ matches the $\mathsf{Ideal}^{\mathcal{Z},\mathcal{S},\mathcal{F}}()$ game of Figure 6 with a concrete simulation strategy. As such, for negligible function $\mu$, we have that

$$|\Pr[\mathsf{Real}^{\mathcal{Z},\mathcal{A}}() \Rightarrow \mathsf{T}] - \Pr[\mathsf{Ideal}^{\mathcal{Z},\mathcal{S},\mathcal{F}}() \Rightarrow \mathsf{T}]| \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{cmpc}}$$
$$|\Pr[\mathsf{Real}^{\mathcal{Z},\mathcal{A}}() \Rightarrow \mathsf{T}] - \Pr[\mathsf{Ideal}^{\mathcal{Z},\mathcal{S},\mathcal{F}}() \Rightarrow \mathsf{T}]| \leq \mu()$$

and Theorem 2 follows.

# B Additional CRDTs

## B.1 Other polymorphic CRDTs

*Multi-value register:* Figure 20 provides an implementation of a secure multi-value register, directly adapted from [43]. In comparison to the lat-writer-wins register (Figure 16), the multi-value register keeps a vector clock per stored value. Remark how the consistency restoration semantics remains completely independent from the secure data held by the CRDT, which can be attested by the inexistence of secure MPC operations.

```
type State_RGA a = ([(LC,Secret a,Maybe LC)],MC,Maybe (Update_RGA a))
data Update_RGA a = Ins_RGA (Maybe LC) (Secret a) | Del_RGA LC
data Query_RGA a r where
    Get_RGA :: Query_RGA a [(LC,Secret a)]
type Message_RGA a = Maybe (VC,Update_RGA a)
new_RGA i = ([],emptyMC,Nothing)
query_RGA i Get_RGA (xs,_,_) = foldr (\(c,v,b) ys -> if isJust b then ys else (c,v):ys) [] xs
update_RGA i u st = update Nothing i u st
propagate_RGA i j (st,mc,u) = fmap (lookupMC i mc,) u
merge_RGA i j st_i Nothing = st_i
merge_RGA i j (xs_i,mc_i,u_i) (Just (vc_j,u_j)) = update (Just (LC (lookupVC j vc_j) j)) j u_j (xs_i,mergeMC j
↪  vc_j mc_i,u_i)

update mbuid i u st@(xs,mc,_) = maybe st (,mc',Just u) $ apply mc uid u xs
    where
    vc' = incVC i (lookupMC i mc)
    mc' = Map.insert i vc' mc
    uid = maybe (LC (lookupVC i vc') i) id mbuid
apply mc c (Ins_RGA j e) xs = insert mc j (c,e) xs
apply mc c (Del_RGA j) xs = delete mc c j xs
insert mc Nothing e xs = Just $ insertBody mc e xs
insert mc (Just j) e [] = Nothing
insert mc (Just j) e (x@(i_x,v_x,b_x):xs) = if j == i_x
    then Just (purgeCons mc x $ insertBody mc e xs)
    else fmap (purgeCons mc x) (insert mc (Just j) e xs)
insertBody mc (i,v) [] = [(i,v,Nothing)]
insertBody mc e@(i,v) (x:xs) = if i > fst3 x
    then (i,v,Nothing) : purgeCons mc x xs
    else purgeCons mc x (insertBody mc e xs)
delete mc k j [] = Nothing
delete mc k j (x@(i,v,b):xs) = if i == j
    then Just ((i,v,Just k) : xs)
    else fmap (purgeCons mc x) (delete mc k j xs)
purgeCons mc (i,v,Just k) xs | leMC k mc = xs
purgeCons vc x xs = x : xs
```

Fig. 21: Operation-based RGA CRDT from [20], extended with tombstone purging as in [42].

*Replicated growable array:* The replicated growable array (RGA) [42] is a CRDT that generalizes registers to sequences of elements. Likewise registers, they are polymorphic over the type of elements, meaning that they can be directly made to hold secret values. Figure 21 presents a functional RGA implementation whose convergence properties have been verified in [20]. Each element has a causally consistent unique identifier (a Lamport clock `C`), and a tombstone. when a value is deleted, it is marked with a tombstone with the event's unique identifier. A user interface may hide identifiers from the user, and internally map list indexes to unique identifiers. This operation-based CRDT assumes an underlying reliable causally-ordered broadcast communication protocol, i.e., one that delivers every message to every recipient exactly once and in an order consistent with happened-before. In our interface, the state keeps the last successfully performed update to be propagated. To purge tombstones, each replica shall keep $n$ vector clocks, as a matrix clock. Our implementation follows [42]; instead of a separate garbage collection process, we remove tombstones as we traverse the list in insert/delete operations.

### B.2 Other set CRDTs

*Shuffle-based grow-only set:* One advanced optimization trick often used in MPC, dubbed *leakage cancelling*, consists e.g., in shuffling elements of an array before declassifying comparison results; if the comparisons are performed over completely arbitrary elements of the array, their relative leakage avoids revealing control-flow information of the program [1].

```
type State_SGSet a = (State_GSet a,Bool)
data Update_SGSet a = Add_SGSet (S a) | Compress_SGSet
data Query_SGSet a r where
    GetAll_SGSet :: Query_SGSet a (Maybe [S a])
    Exists_SGSet :: S a -> Query_SGSet a (S Bool)
type Message_SGSet a = State_GSet a
new_SGSet i = ([],True)
query_SGSet i GetAll_SGSet (st,False) = Nothing
query_SGSet i GetAll_SGSet (st,True) = Just st
query_SGSet i (Exists_SGSet x) (xs,_) = query_GSet i (Exists_GSet x) xs
update_SGSet i (Add_SGSet x) (xs,b) = (x:xs,False)
update_SGSet i Compress_SGSet (xs,b) = (nubBy (\x y -> declassify (x == y)) $ shuffle xs,True)
propagate_SGSet i j (xs,_) = xs
merge_SGSet i j (xs_i,b) m_j = (xs_i++m_j,False)
```

Fig. 22: Secure state-based shuffle grow-only set.

```
type State_OGSet a = Map a (S Bool)
data Update_OGSet a = Add_OGSet (S a)
data Query_OGSet a r where
    GetAll_OGSet :: Query_OGSet a (Map a (S Bool))
    Exists_OGSet :: S a -> Query_OGSet a (S Bool)
type Message_OGSet a = State_OGSet a
new_OGSet i = Map.fromDistinctAscList (map (,classify False) [minBound..maxBound])
query_OGSet i GetAll_OGSet st = st
query_OGSet i (Exists_OGSet x) st = Map.foldrWithKey (\v b acc -> acc || (x == (classify v))) (classify False) st
update_OGSet i (Add_OGSet x) st = Map.mapWithKey (\v b -> b || (x == (classify v))) st
propagate_OGSet i j st = st
merge_OGSet i j st_i m_j = Map.unionWith (||) st_i m_j
```

Fig. 23: Secure state-based oblivious grow-only set.

Based on this idea, Figure 22 proposes a grow-only set variant that inserts duplicate elements into the list, and shuffles the elements in the list before pruning duplicates, which leaks equality comparisons. We assume the existence of an efficient special randomized operation that randomly shuffles a list of elements:

```
shuffle :: [S a] -> [S a]
```

To cancel such leakage, we devise a special compress operations, that needs to be performed before each `GetAll_SGSet` (but not before each `Exists_SGSet`). The resulting high-level leakage of this set is simply the size of the set; together with the history of events, this can be used to infer how many equal values exist in the set.

*Oblivious grow-only set:* In MPC, public conditionals can be avoided by obliviously computing the result as an arithmetic computation, at the expense of having to evaluate all possible cases. This is precisely the rationale behind our secure conditional, which computes `if b then a else c` as `a*(fromEnum b) + c*(fromEnum (not b))`, for boolean `b` and numerical values `a` and `c`, where `fromEnum` converts a boolean to a numerical value. This technique may inclusively be used to inefficiently execute general computations expressed as boolean or arithmetic circuits.

Figure 23 presents our proposal of an oblivious grow-only set. Each replica's state allocates space proportional to the maximum number of elements of type `a`, and is conceptually equivalent to an array of booleans, where each value in type `a` is identified by a unique index. This way, all CRDT operations can be performed as point-wise operations over boolean arrays. `GetAll_OGSet` returns a mapping of values to booleans; client-side code could convert it into the isomorphic representation of the set. `Exists_OGSet` obliviously tests all elements. This set has no leakage, and may be suitable for small enumerations.

A long line of work is focused on designing *oblivious ram*, a set of algorithmic techniques to hide the access patterns of arbitrary programs, that can be used construct oblivious data structures for secure computation [49]. We leave a deeper investigation on the trade-offs of using oblivious ram in the design of more complex oblivious CRDTs for future work.

## C Auxiliary `Haskell` Definitions

This section provides the `Haskell` definitions of common auxiliary data structures used for CRDTs. Standard `Haskell` definitions can be consulted at `https://hoogle.haskell.org/`.

```
maxOn :: Ord b => (a -> b) -> a -> a -> a
maxOn f x y = if f x >= f y then x else y

-- unique replica identifier
type 𝕀 = Int

-- replica-local time
newtype T = T Integer deriving (Eq,Ord)
startT = T 0
nextT (T t) = T $ succ t

-- lamport clock
data LC = LC T 𝕀 deriving (Eq,Ord)

-- vector clock
type VC = Map 𝕀 T

leVC :: VC -> VC -> Bool
leVC vv1 vv2 = all id $ Map.mergeWithKey (\k i1 i2 -> Just (i1 <= i2)) (Map.map (const False)) (Map.map (const
↪  True)) vv1 vv2

incVC :: 𝕀 -> VC -> VC
incVC i = Map.alter (Just . nextT . fromMaybe startT) i

lookupVC :: 𝕀 -> VC -> T
lookupVC i vc = maybe startT id (Map.lookup i vc)

maxVC :: VC -> VC -> VC
maxVC vc1 vc2 = Map.unionWith max vc1 vc2

maxVCs :: [VC] -> VC
maxVCs = Map.unionsWith max

-- matrix clock
type MC = Map 𝕀 VC

mergeMC :: 𝕀 -> VC -> MC -> MC
mergeMC j vc_j mc = Map.alter (Just . maybe vc_j (maxVC vc_j)) j mc

lookupMC :: 𝕀 -> MC -> VC
lookupMC i mc = maybe Map.empty id (Map.lookup i mc)

leMC :: LC -> MC -> Bool
leMC (LC c i) mc = Map.foldrWithKey (\j vc_j b -> b && c <= lookupVC i vc_j) True mc

emptyMC :: MC
emptyMC = Map.empty
```

## D Auxiliary Experimental Evaluation Results

This section presents the results of the experimental evaluation for the secure CRDTs with an increasing number of concurrent clients.

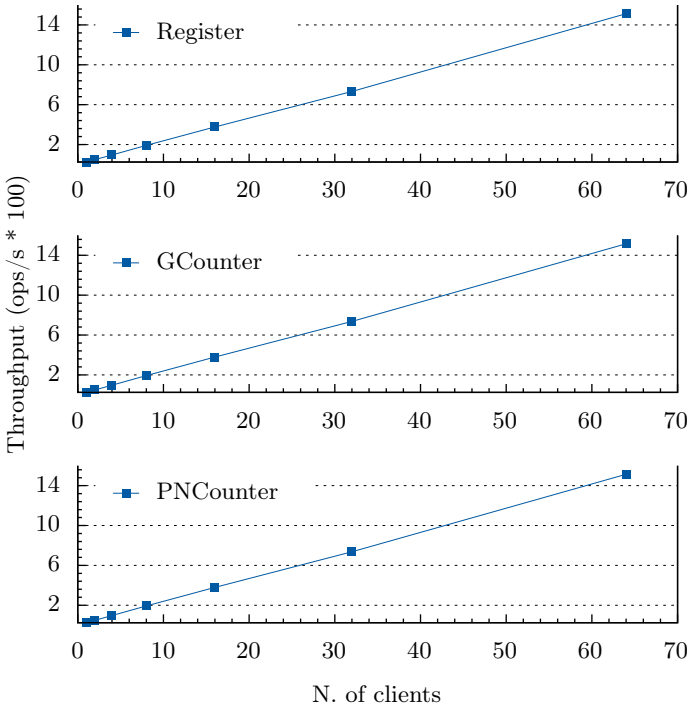


Fig. 24: Average throughput of the `update` operation for the `LWW` Register, `GC2` Counter and `PNC` Counter.

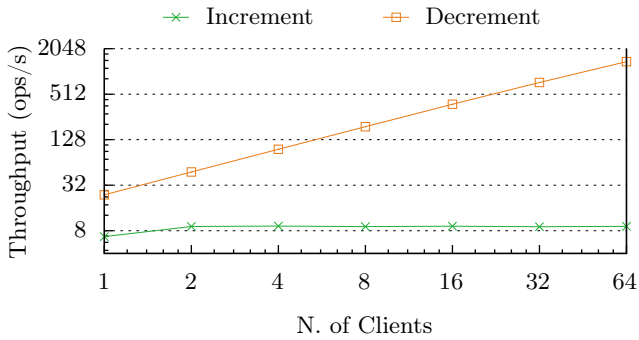


Fig. 25: Average throughput of increment and decrement operations on Bounded Counter CRDT with an increasing number of concurrent clients.