Secure Conflict-free Replicated Data Types

ABSTRACT

Conflict-free Replicated Data Types (CRDTs) are abstract data types that support developers when designing and reasoning about distributed systems with eventual consistency guarantees. In their core they solve the problem of how to deal with concurrent operations, in a way that is transparent for developers. However in the real world, distributed systems also suffer from other relevant problems, including security and privacy issues and especially when participants can be untrusted. In this paper we present the first formal cryptographic treatment of CRDTs, as well as proposals for secure implementations. We start by presenting a security notion that is compatible with standard definitions in cryptography. Then we present several specialized constructions that leverage different security/performance trade-offs. We accompany these contributions with formal proofs in the proposed model, and we implement and integrate them in AntidoteDB, a geo-replicated NoSQL database that leverages CRDTs for implementing its operations. Experimental evaluations conducted show the tradeoffs of the different proposals and reveal that they are ready to be used in practical applications.

ACM Reference Format:

. 2020. Secure Conflict-free Replicated Data Types. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A Conflict-Free Replicated Data Type [26] (CRDT) is a recent abstraction for distributed cloud-oriented protocols that permits maintaining multiple replicas of a data value with high availability and low latency for local access. These protocols explore a tradeoff where one forsakes strong consistency in exchange for a weaker, but sufficient, notion of coherence between geographically distributed operations, called eventual consistency. Apple [10], Microsoft [18], Facebook [16], and Google [13] are some of the many organizations that have used CRDTs in one or more of their products. Example systems built on top of CRDTs include geo-replicated databases [23], collaborative text edition [12], and chat systems for massive-multiplayer online video games [22]. In this paper we initiate the rigorous treatment of CRDT security in order to enable CRDT-based privacy-preserving applications.

CRDT Concepts. A CRDT is a distributed protocol in which a set of clients interacts with a set of replica server nodes to update and query values stored under the form of complex data-structures, including registers, sets, lists, maps, and counters. Server nodes maintain the current value of the replica and additional meta information that is needed to provide the prescribed consistency semantics. Server nodes may propagate full or aggregate information about their internal states to other replicas, which leads to a notion of eventual consistency: the idea is that, if local updates cease to occur and enough propagation of replica states takes place, the whole system will converge to the same observable data value in all replicas. The rate at which replica propagation occurs is application-specific.

CRDT Application. CRDTs can be used in different scenarios. In this paper, we are particularly interested in their application to support cloud-backed geo-replicated NoSQL databases. An example implementation available in the real-world is AntidoteDB [1]. This type of technology is important, for instance, for medical hospitals that need to store large volumes of patient health records in the cloud in a highly-available and privacy-preserving way. In this scenario, each cloud server stores an AntidoteDB replica for high-availability, and clients (i.e., the medical doctors) connect to a cloud of their choice and search/update health records of their patients. Moreover, cloud servers are typically considered untrusted, following an honest-but-curious model and hence justifying the need for privacy, while medical doctors are usually considered trustworthy and only require access control mechanisms.

CRDT Security. There is no formal treatment of CRDT security in the literature, and so we begin our treatment of this topic by proposing a notion of security in the Universal Composability framework [8]. Intuitively, the notion we propose, with respect to adversaries that control the scheduling of CRDT operations, is realisable using fully homomorphic encryption and general secure multiparty computation. However, in the first case the solution is not practical and, in the second, the trust model associated with efficient protocols [9] would require sharing secret data between multiple nodes, which goes against the purpose of CRDTs in the first place.

Our Contributions. The contributions presented in this paper are as follows:

- Definitions: We give a definition of security for CRDTs in the Universal Composability framework. Our ideal functionality is general enough to capture solutions that leak no information about the stored data, as well as those that may leak partial information. We then focus on honest-but-curious adversaries and give a restricted execution model tailored for this class of adversaries that is still strong enough to capture the consistency requirements of arbitrary CRDTs.
- Black-box constructions: we give efficient constructions that introduce minimum overhead over standard CRDTs. A black-box construction overlays encryption over CRDT protocols, which permits using existing CRDT implementations without modification. The feasibility of this construction depends on the CRDT context, as we may require encryptions to leak

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

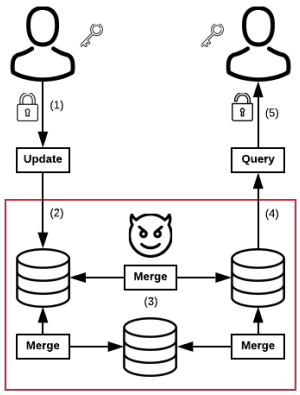

Figure 1: High-level view of our system and adversary models.

partial information about stored values. In this case, we map the protocol leakage to standard weaker notions of encryption security.

- Homomorphic transformations: For CRDT protocols that are not covered by the previous approach (e.g., counters and bounded counters) we show that partially homomorphic encryption schemes can also be used to naturally transform them into secure ones with small overhead and leakage, and minimal alterations to the implementation.

We support these results with both formal security proofs and an experimental evaluation. We analyse security/performance trade-offs for register, set, counter and bounded counter CRDTs. All secure CRDT constructions are implemented and integrated in AntidoteDB, allowing it to provide security and privacy guarantees to clients regarding their cloud-stored data.

2 TECHNICAL OVERVIEW

We start by presenting an overview of our system. In a CRDT protocol, clients interact with a group of servers nodes via update and query operations, and servers propagate their states between themselves to ensure that the view of each node will eventually become consistent with respect to all operations performed by the clients. Our setting for CRDT security is focused on providing guarantees to the clients against an untrusted network and servers. Ideally, we want a layer of security between clients writing and reading data, such that one can encode sensitive data before sending to the untrusted network and have it be decoded when it exits, while seamlessly propagating and merging encoded states.

Figure 1 captures this scenario. First, we have a setup assumption where clients establish cryptographic material beforehand to use in the security layer. This can be done with symmetric keys, by having clients perform an a-priori key-exchange protocol; or with asymmetric keys, i.e. by separating writer from readers, having a secret key shared by all writers to perform updates, and creating a public key for readers to encode queries. Our execution model is agnostic to this setup, as it can be defined by the chosen cryptographic technique.

From then onwards, every update operation is preceded by an encoding operation (e.g., encryption) to ensure security (step 1). Given the encoded data, the server can then perform the CRDT update operation (step 2). This will be followed by (potentially multiple) propagations and merges (step 3), which are processed over the encoded data. This is the core challenge of our constructions, as the chosen security mechanism must also ensure that all computations in both update and merge steps can be performed efficiently over encoded data. The nodes can then be queried for the encoded state (step 4). Finally, the obtained result must be decoded to retrieve the query response (step 5).

We assume adversaries will be honest-but-curious. This is a common adversarial model for cloud computing and secure computation solutions [5], as it captures attack vectors where, for instance, the service provider does not deviate from its service level agreements but may still observe all data state and accesses; or when an external intruder briefly gains access to the system and can access the database and execution logs. In particular, we show our solutions to be secure in a setting where all server nodes can reveal their internal executions. Given this setup, one cannot prevent however an attacker with total control over the network from delaying or shutting down the system (i.e., denial-of-service). Our goal is instead to demonstrate that the attacker is unable to extract any meaningful information from encoded messages, or to have the system deviate in any other way besides delaying updates.

3 CRDT SECURITY

Syntax. A CRDT protocol is deployed over a network composed of n server nodes, or replicas, statically defined at the beginning of the protocol and identified by $\mathsf{id}_1, \ldots, \mathsf{id}_n$. These nodes are accessible to an arbitrary number of client nodes—the entities performing read/write accesses to the data-type—which we model as two (distributed) entities. This allows for client nodes to share long-term keys, and is sufficiently flexible to capture symmetric scenarios, where both have the same keys, and asymmetric scenarios, where readers and writers play different roles and thus have access to different cryptographic material. Secure CRDT protocols have the following syntax:

- setup() is the global client setup procedure, which produces a set of private parameters prv_q, prv_u and public parameters pub.
- init(pub, id) is the server node initialization procedure, which on input the public parameters pub and the server node identifier id outputs the inital state st for that node.
- $\mathsf{query}\langle \mathsf{prv}_q, \mathsf{op} \mid \mathsf{st} \rangle$ is an interactive protocol executed between a client node and a server node. On the client-side it takes as input the private parameters for query prv_q and a query operation op. On the server-side, it takes a state st as input. There is no server-side output. The client recovers output o.
- $\mathsf{update}\langle \mathsf{prv}_u, \mathsf{op}, v \mid \mathsf{st} \rangle$ is an interactive protocol executed between a client node and a server node. On the client-side it takes as input the private parameters for update prv_u, an input operation op and an input value v. On the server side it takes as input a state st. At the end of the protocol the server gets an updated state st' and the client may recover output o, e.g., indicating the success of the operation.

- prop(st, id) is a local server node operation that takes st and a target replica identifier id and produces update data up to be sent over the network to the target replica.
- merge(up, st) is a local server node operation that takes an initial state st and an update up and produces an updated state st′.

Security. We formalize security in the Universal Composability framework [7], but we simplify presentation of the execution model as a consequence of focusing on a restricted class of adversaries, as follows:

- We consider an honest-but-curious adversary with adaptive corruptions: the attacker will attempt to break system confidentiality by observing messages passed in the system and internal server states, but it does not have full control over any of the entities of the system (e.g., causing them to send arbitrary messages), nor does it have full control over the communication channels, which we assume to be authenticated.
- To guarantee confidentiality and correctness are preserved for any possible scheduling of CRDT operations we allow the adversary to control the sequence of operations, namely the interactions between server nodes and the points at which clients provide inputs and receive outputs from the system at different server nodes. This essentially means that we adopt an asynchronous execution model and allow the attacker to control the message scheduling.
- For simplicity, we restrict the adversary's scheduling capabilities when it comes to the query and update subprotocols, and assume that they are atomic in the execution model; the attacker receives an execution trace t whenever one of them is run between a client and a server. However, all our results for concrete protocols hold in the more general execution model where the attacker could also arbitrary schedule the intermediate messages of query and update.

As in standard UC, the correctness and security of the protocol are specified via an ideal functionality which we introduce next.

Ideal Functionality. The ideal functionality $\mathcal{F}$ is shown in Figure 2. It is parametrised by an algorithm correct and a leakage function $\mathcal{L}$, and it maintains a log L of all operations carried out over the CRDT.

The correct algorithm can be used to specify arbitrary concurrency semantics [21] for the CRDT: whenever the environment provides an input via write, or reads a value via read, the functionality computes the correct client-side result that should be observed by the environment by computing correct(L) over the entire history of operations carried out over the CRDT. The history of operations is accumulated in a numbered list L (implemented here as a set) and includes both the input/output operations carried out by the environment $\mathcal{Z}$ and the scheduling of state merges between replicas specified by the adversary $\mathcal{S}$. The latter are specified via two commands: snap allows S to commit to a point in the global history where a snapshot of the state of replica i is sent to replica j, whereas the set command allows $\mathcal{S}$ to register into the global log the merge into the state of replica j of the oldest available snapshot. The restriction of using the oldest replica snapshot limits the power of the

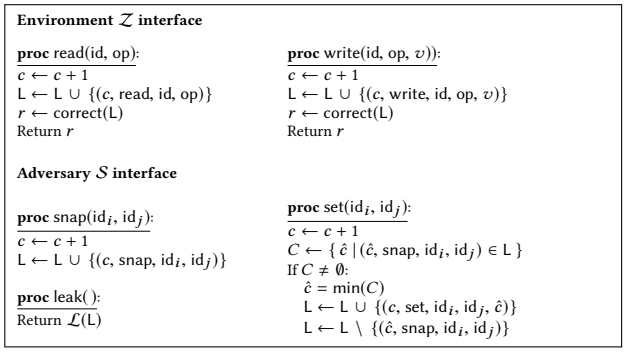

Figure 2: Ideal functionality $\mathcal{F}$. $\mathcal{F}$.init sets L to $\perp$ and c to 0.

simulator so that it is not able to instruct the functionality to forget commands to propagate the state from one replica to another: it can only delay the time at which that snapshot of the state is merged. Secure CRDTs therefore need to be consistent with this scheduling restriction, which in most cases is achieved by relying on standard authenticated channels between server nodes.

The leakage function $\mathcal{L}$ allows the adversary $\mathcal{S}$ to obtain leakage from the global trace. A fully secure CRDT will reveal only how many operations the environment carried out, plus the sizes of inputs and outputs to these operations. Weaker security definitions can be obtained by revealing to $\mathcal{S}$ more information about which operations were carried out and partial leakage of the input/output values themselves, such as equality patterns. For conciseness throughout the paper, when describing specific leakage functions, these will detail the additional leakage over this baseline, i.e. $\mathcal{L}(\mathsf{L}) = \epsilon$ reveals how many operations the environment carried out, and sizes of inputs parameters and output values.

We will later demonstrate how CRDT solutions can be constructed with this baseline leakage, as well as exemplify how specific protocols can rely on some leakage to achieve improved performance.

Note that by defining correct behaviour as a function of the whole history our functionality permits specifying the strong convergence requirements of CRDTs via the UC-security requirement. Confidentiality follows from the fact that the adversary/simulator can control the sequence of server node interactions, but it obtains no information about the client inputs other than what is specified by the leakage functions.

Security model. Figure 3 shows the simplified execution model for our UC security definition. As usual, we consider an environment $\mathcal{Z}$ that will collaborate with adversary $\mathcal{A}$ to distinguish the real world from an ideal world where it interacts with a simulator $\mathcal{S}$. In both worlds, $\mathcal{Z}$ can call oracles write and read to trigger client actions on the CRDT. In the real world these actions map to client updates and client queries to a CRDT replica. As previously mentioned, we consider these to be atomic in the execution model for simplicity: the update/query protocols are executed and then the attacker can retrieve the execution trace.

Furthermore, in the real world, environment $\mathcal{Z}$ can control the sequence of prop and merge operations between server nodes via adversary $\mathcal{A}$. Rather than requiring protocols to explicitly rely on a hybrid authenticated channel functionality, and since we are dealing with honest-but-curious adversaries, we simplify the execution

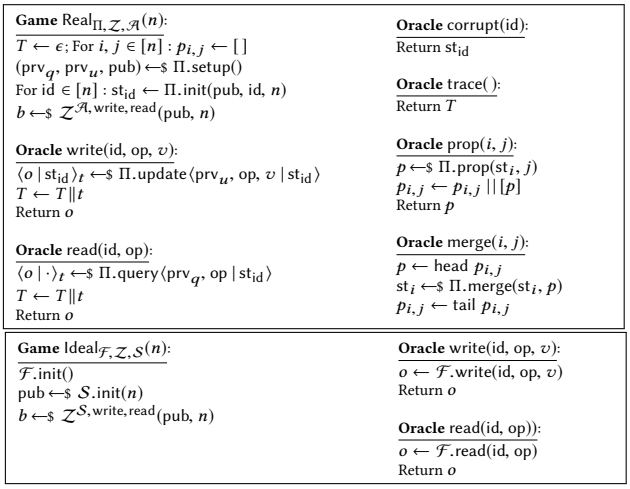

Figure 3: Real and Ideal security games. In the real world, $\mathcal{A}$ has access to oracles corrupt**,** trace**,** prop **and** merge**. In the ideal world, $\mathcal{S}$ has access to the adversarial interface of $\mathcal{F}$.**

model by exposing two oracles to $\mathcal{A}$ that directly map to these operations and impose that values output by the propagate oracle are delivered to merge in the correct order.

The fundamental difference between real and ideal world is that the real world will be executing and displaying the protocol Π on the different nodes, while the ideal world will be displaying the interface $\mathcal{F}$ of read and write. Messages exchanged in the network will instead be emulated by a simulator $\mathcal{S}$, whose only interaction with $\mathcal{F}$ is via its snap, set and leak commands. $\mathcal{S}$ is also responsible for presenting a consistent replica state upon corrupt. The concrete security goal is to show that the distribution of output bit b produced by $\mathcal{Z}$ is essentially the same in both worlds.

The intuition behind this security definition is that, for any secure CRDT protocol, a real world adversary cannot influence the system beyond refusing to transmit state transitions (denial-of-service), and gains no information other than what is concretely specified by our leakage functions. This is because the real world trace can be simulated without access to non-leaked internal values of $\mathcal{F}$, by a simulator that can only control the schedule of server node interactions.

Definition 3.1. *Let $n \in \mathbb{N}$. Let $\mathcal{F}$ be an ideal functionality, and let Π be the corresponding CRDT protocol. We say that Π realises $\mathcal{F}$ if there exists a simulator $\mathcal{S}$ such that, for any environment $\mathcal{Z}$ and adversary $\mathcal{A}$,*

$$\text{Real}_{\Pi,\mathcal{Z},\mathcal{A}}(n) \approx \text{Ideal}_{\mathcal{F},\mathcal{Z},\mathcal{A},\mathcal{S}}(n)$$

Correctness and relation to CRDT concurrency semantics. Our ideal functionality can also be used to give a crypto-style definition of correctness for CRDT protocols. Indeed, we can take correct to be the algorithm that implements the concurrency semantics for a given CRDT as in [21].

Concurrency semantics of a CRDT can be seen as a function $\text{sem} : O \to (\prec) \to V$, where:

- O is a set of client update operations on the entire CRDT that is known to a given replica. Each element in this set is of the form $o_{i,j}$, indicating that update operation o occurred in replica i and this was the j-th update operation in this node.
- $(\prec)$ is a partial order on the operations in O capturing the *happens-before* causality notion [15]: $o_{i,j} \prec o_{k,l}$ holds if and only if, when replica k processed its l-th client update, it had received information depending on the j-th update at replica i.
- V is the set of possible CRDT read values.

When client update operations may return immediate feedback to the caller, e.g., to indicate success or failure of the operation, then usually this is expressed as a predicate inv that takes the current replica state and update operation and determines the validity of the update operation.

Given sem and inv it is easy to see that one can define an algorithm correct that exactly matches these concurrency semantics: correct first computes O and $(\prec)$ from the global history recorded by $\mathcal{F}$ and then uses sem and inv to compute the value that should be observed by the client. Then, a CRDT protocol is correct in the sense of [21] if it securely emulates functionality $\mathcal{F}$ with *full* leakage.[1] To see this, suppose the protocol is incorrect. Then there exists an adversary that can drive the CRDT in the real world into a configuration that is inconsistent with the concurrency semantics, and hence with correct. However, no simulator can ever do this. It is also the case that a CRDT protocol securely emulates functionality $\mathcal{F}$ with *full* leakage if it is correct in the sense of [21], making these definitions equivalent. To see this, observe that with full leakage we can have a trivial simulator that follows the protocol specifications to produce indistinguishable states and traces. In this case, the difference is only on the outputs of Π and correct. However, correct was defined according to sem and inv, so these are also the same.

4 CONSTRUCTIONS

We now present several CRDT constructions that are demonstrably secure under the proposed security model. These designs can be seen as natural instantiations of CRDT solutions with a security layer ensuring confidentiality of stored data. The client will perform an encryption to protect sensitive information in update operations, and a decryption to successfully query the CRDT state. Replica-side states must merge efficiently, even when storing encrypted data.

Intuitively, the differences between constructions are dependent on the underlying functionality provided by the CRDT. For instance, registers perform no server-side computation over the stored values, so we can rely on a standard encryption scheme seamlessly. On the other hand, counters assume that the servers can perform arithmetic over the stored value, suggesting the value of using encryption schemes with homomorphic properties. This highlights the natural correlation between CRDT functionality and the necessary properties of the underlying security mechanism.

4.1 Register CRDT

A register CRDT is a standard data structure holding a single value. Update operation replaces the register value, and query returns it. This is often a fundamental building block to real-world applications requiring more complex data structures, such as multi-value maps.

[1] By full leakage we mean the identity leakage function, revealing everything that occurred in the environment interface to the adversary.

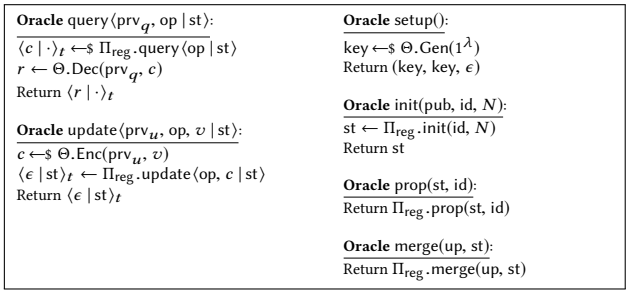

Figure 4: Construction of Secure register from standard encryption scheme Θ.

Register CRDTs are very simple data types, in which merge operations compare and maintain only the most recent version of the register. This means we have no computational requirements over the values maintained, and thus can rely on their encoded versions without disrupting their behavior. Concretelly, when a client wants to update the register, it encrypts it before sending to the server. When the client wants to read the register, it requests its encoding from the server and decrypts it to retrieve the plaintext value. Since no computations must be done by the servers on the register value, the CRDT runs seamlessly over encoded values.

Our construction is described in Figure 4 and is as follows. We rely on a black-box construction of register CRDT Π_{reg}, with operations for init, update, query, prop and merge, and on a standard encryption scheme Θ. The client generates and maintains a symmetric key. update calls Θ.Enc to instead store the ciphertext in Π_{reg}. query retrieves the ciphertext from Π_{reg} and calls Θ.Dec to obtain the original value.

Security. We argue that our construction is a CRDT register with baseline leakage

Theorem 4.1. *If Π_{reg} is a correct CRDT, and Θ is a IND-CPA encryption scheme, the construction in Figure 4 is a secure CRDT with leakage*

$$\mathcal{L}_{\text{reg}}(\mathsf{L}) = \epsilon$$

Confidentiality is ensured by an encryption scheme, and correctness comes from the underlying CRDT. The full proof can be read in Appendix A, and the sketch is as follows.

The simulator $\mathcal{S}$ will initialize its parameters (a symmetric key), a trace status counter, and the emulated state of every replica. Whenever any operation is called on $\mathcal{S}$, it will call $\mathcal{F}$.leak to obtain the list of environment operations and their length, and update its view for the operations not yet processed (accounted by the trace status counter). This will be a common behavior pattern for secure CRDT simulators, as presenting indistinguishable views for prop, merge, trace and corrupt follows the same simulation strategy: emulate environment operations according to what is given by $\mathcal{F}$.leak, then follow the protocol for the simulated states/traces. The protocol-specific nuances in simulation strategy are related to the processing of environment operations, which are as follows.

For every write operation, the simulator $\mathcal{S}$ receives the size of the update and generates a dummy ciphertext (encrypts zeros) with the same length. Since these will be the messages stored in Π_{reg}, $\mathcal{S}$ has perfect information of its global log, and can call its trivial simulator to produce the associated execution trace and states. The proof is done in three hops.

- Hop 1 - Instead of using the outputs of Π_{reg}, use an idealized structure for storing encrypted values and to present the output of read (decrypted before being returned). These games are identical given the correctness of Π_{reg}.
- Hop 2 - Instead of using a structure with encrypted values, use a structure with plaintext values. These games are identical given the assumption that the CRDT behaves seamlessly over encoded values.
- Hop 3 - Replace values on update operations by dummy encryptions. IND-CPA of Θ will ensure that the difference between these games is negligible. We are now in a simulatable ideal world.

Discussion. The cryptographic overhead of this security layer, considering widely available hardware acceleration of modern processors, is as minimal as one can expect to achieve: we require one key generation step at the start of the protocol, one encryption on update and one decryption on query.

We stress that this is only possible given that operations for managing replica states do not rely on any computation over the actual encrypted state. This suggests the potential of having highly scalable secure CRDT solutions when the protocol for ensuring consistency only relies on associated metadata (e.g. timestamps).

We can also further reduce the leakage of this CRDT by having the client pad updates to the maximum length of the register value, which is free if we are storing fixed length values.

4.2 Set CRDT

Set CRDTs present a slightly more complex problem, as internal operations require the servers to compute over stored values. Standard CRDTs merge sets by having the servers perform comparisons to maintain only unique elements. This suggests the need for equality comparison, a functionality for which cryptography presents multiple solutions. As such, our security layer will encode set values upon update, decode values when queries are performed, and replace (if necessary) server-side comparisons with equivalent operations on encoded data.

Concretely, we denote CRDT set to be a data structure enabling update operations for adding (add, v) and removing (rem, v) elements to the set. This can then be queried using (cont, v), which checks if v is in the set; and get, which retrieves the full set.

To demonstrate a feasible construction, we instantiate our comparison enabling security layer with a deterministic encryption scheme. This allows us to rely on any set CRDT in a black-box manner, as our encoded value comparison is seamless. Observe that this is for simplicity in presentation and not restrictive, as we can freely choose other implementations that enable comparisons, such as searchable encryption. This extension entails replacing instances of comparison within the CRDT with the respective secure operations, and minimal changes to the proof.

Our construction is described in Figure 5, and is very similar to the previous one. We rely on a black-box construction of set CRDT Π_{set}, with operations for update, query, prop and merge, and on a deterministic encryption scheme Ω. The client generates and maintains a symmetric key. update calls Ω.Enc to instead store

Oracle query$\langle \mathsf{prv}_q, \mathsf{op} \mid \mathsf{st} \rangle$:
If op = (cont, v):
  $\mathsf{cph} \leftarrow\$\ \Omega.\mathsf{Enc}(\mathsf{prv}_q, v)$
  $\langle r \mid \cdot \rangle_t \leftarrow \Pi_{\mathsf{set}}.\mathsf{query}\langle(\mathsf{cont}, \mathsf{cph}) \mid \mathsf{st}\rangle$
Else:
  $\langle \mathsf{cph} \mid \cdot \rangle_t \leftarrow \Pi_{\mathsf{set}}.\mathsf{query}\langle \mathsf{op} \mid \mathsf{st}\rangle$
  $r \leftarrow \Omega.\mathsf{Dec}(\mathsf{prv}_q, \mathsf{cph})$
Return $\langle r \mid \cdot \rangle_t$

Oracle update$\langle \mathsf{prv}_u, \mathsf{op}, v \mid \mathsf{st}\rangle$:
$\mathsf{cph} \leftarrow\$\ \Omega.\mathsf{Enc}(\mathsf{prv}_u, v)$
$\langle \epsilon \mid \mathsf{st}\rangle_t \leftarrow \Pi_{\mathsf{set}}.\mathsf{update}\langle \mathsf{op}, \mathsf{cph} \mid \mathsf{st}\rangle$
Return $\langle \epsilon \mid \mathsf{st}\rangle_t$

Oracle setup():
$\mathsf{key} \leftarrow\$\ \Omega.\mathsf{Gen}(1^\lambda)$
Return (key, key, ϵ)

Oracle init(pub, id, N):
$\mathsf{st} \leftarrow \Pi_{\mathsf{set}}.\mathsf{init}(\mathsf{id}, N)$
Return st

Oracle prop(st, id):
Return $\Pi_{\mathsf{set}}.\mathsf{prop}(\mathsf{st}, \mathsf{id})$

Oracle merge(up, st):
Return $\Pi_{\mathsf{set}}.\mathsf{merge}(\mathsf{up}, \mathsf{st})$

Figure 5: Construction of secure set from deterministic encryption scheme Ω.

the ciphertext in Π_{reg}. query either performs a contains operation, with inputs op = cont and v, or retrieves the ciphertext from Π_{reg} and calls Ω.Dec to obtain the set's full state.

Security. We argue that our construction is a CRDT set with additional write leakage of the operation, and of a synthetic label l_v for each stored/checked values v, allowing for the adversary to know when a duplicate is processed.

Theorem 4.2. *If Π_{set} is a secure CRDT with full leakage, and Ω is a secure deterministic encryption scheme, the construction in Figure 5 is a secure CRDT with leakage*

$$\mathcal{L}_{\mathsf{write}}(\mathsf{L}) = (\mathsf{op}, l_v)$$
$$\mathcal{L}_{\mathsf{read}}(\mathsf{L}) = (\mathsf{cont}, l_v)$$

where $\mathcal{L}_{\mathsf{write}}$ denotes the behavior of $\mathcal{L}$ for every write *operation, and $\mathcal{L}_{\mathsf{read}}$ denotes the behavior of $\mathcal{L}$ for every* read *operation for* op = cont.

Security. The security reasoning is very similar to the one of the register: confidentiality is ensured by the encryption scheme (with leakage of duplicates), and correctness comes from the underlying CRDT. The full proof is structurally identical to the previous one, modulo simple extensions of write and read operations to parametrize for the different operations and a slight tweak on the simulator. To avoid redundancy, we explain the differences.

The simulator $\mathcal{S}$ will behave exactly like the one for the register, modulo the creation of dummy encryptions. It will maintain a table from labels to dummy encryptions, initialized empty. For read/write operations, the simulator receives a label (leakage): if it is not in the map, it will encrypt it with Ω, and add it to the map, otherwise it reuses the previous one. This ensures that every different encryption in the system is replaced by a dummy encryption without breaking consistency in equalities. $\mathcal{S}$ can then have perfect information of the values stored in Π_{set}, and rely on its trivial simulator to emulate traces and states.

The proof sketch follows the same overall structure of the previous proof. The main difference is on the third hop, as the difference between these games can be established as negligible given a restricted indistinguishability security setting for deterministic encryption schemes [3]. Here, the adversary can also perform adaptive queries, but has pattern restrictions for repeated queries (to prevent trivial attacks), which are enforced by the simulator.

Discussion. Similar to the register, regarding performance the cryptographic overhead imposed is minimal. Again, this is only possible with the assumption that operations do not rely on any computation over the encrypted values other than equality comparison.

Albeit not being able to directly retrieve the plaintext values, the leakage implies that rogue replicas have full knowledge of when an element that already exists in the set is removed, and of the results of all cont queries. To reduce this to have standard indistinguishability security, we must exclude all behaviors that require this equality comparison – which has communication and computation tolls – or have an implementation that relies on techniques for homomorphic equality comparison – which cannot be done black-box, and will naturally be less efficient than standard equality comparisons.

4.3 Counter CRDT

A counter CRDT is a numerical data structure that can be either incremented or decremented by an arbitrary amount, at any server on the network. The implementation of these data structures usually involves maintaining two counters per replica, one for increments and another for decrements. Update operations increment to the respective replica counter. These are compared upon merge and added upon query, to obtain the observed value.

For this construction, we have first a transformation that removes the need for comparing counter values upon merge. This is simply the inclusion of a per-replica Lamport clock in the clear to establish partial ordering of events.[2] This allows us to restrict the necessary computations on the encoded counter to additions, which can be done over encoded values if we instantiate our security layer with an additively homomorphic scheme.

This construction now requires some changes to the underlying CRDT. To argue that the resulting CRDT does not deviate from the correct behavior of a counter, we specify the concurrency semantics of our counter. Let O denote the set of all update operations seen by the queried replica, and i integer values:

$$\sum\{\mathsf{inc}(i) \mid \mathsf{inc}(i) \in O\} - \sum\{\mathsf{dec}(i) \mid \mathsf{dec}(i) \in O\}$$

These concurrency semantics can be instantiated in our functionality syntax by defining $\mathsf{correct}_{\mathsf{ctr}}(\mathsf{L})$ as:

- Get the last identifier id_q and last operation in L. If that is $(\cdot, \mathsf{read}, \cdot, \cdot)$ return ϵ (no feedback on update).
- Construct sets $C[\mathsf{id}]$ as projections of $(c, \mathsf{write}, \mathsf{id}, \mathsf{op}, v)$ in L for all $\mathsf{id} \in N$.
- Sequentially, for every $(\cdot, \mathsf{set}, \mathsf{id}_i, \mathsf{id}_j, c) \in \mathsf{L}$, copy all entries $(\mathsf{write}, \mathsf{id}_k, \mathsf{op}, v, c')$ in $C[\mathsf{id}_i]$ to $C[\mathsf{id}_j]$ such that $c' < c$, remove duplicates.
- Sum every $(\cdot, \mathsf{write}, \mathsf{id}, \mathsf{inc}, v) \in C[\mathsf{id}_q]$, subtract the sum of all $(\cdot, \mathsf{write}, \mathsf{id}, \mathsf{dec}, v) \in C[\mathsf{id}_q]$ and produce it as output.

Here, $\mathsf{correct}_{\mathsf{ctr}}$ is simply reconstructing and computing on the local view of the replica at the time of the operation: i. collect all update operations seen by each replica; ii. complete the view with merges taken from previous snapshots; iii. sum all operations to produce the result. For concretess in our proofs, we will demonstrate correctness of the counter CRDT based on this $\mathsf{correct}_{\mathsf{ctr}}(\mathsf{L})$, which we assume to adequately capture the concurrency semantics of counters specified in [21].

Concretely, our construction is an adaptation of the state-based CRDT counter (Specification 7) in [25], generalised to allow for

[2] Observe that we assume an adversary with full control of the network, which means that no additional information is revealed.

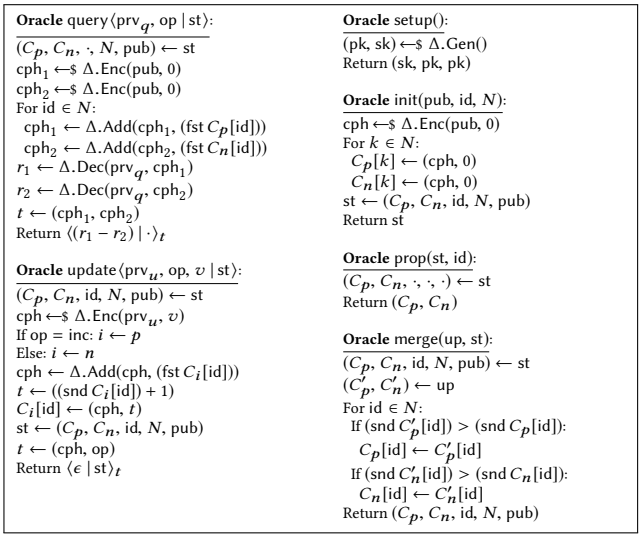

Figure 6: Counter from additively homomorphic scheme Δ

arbitrary increments and decrements. Our only functional tweak is in the behavior of merge, where we use a per-replica operation count to establish freshness in updating the counter. Afterwards, given that all replica-side operations on the counter are additions, we can again overlay security using an additively homomorphic encryption scheme Δ, encrypting inputs, decrypting outputs, and allowing replicas to perform additions. Our construction is detailed in Figure 6.

Security. We argue that our construction is a secure CRDT counter with additional leakage of the operation, i.e.

Theorem 4.3. *If Δ is an IND additively homomorphic encryption scheme, the construction of* Π_{ctr} *described in Figure 6 is a secure CRDT with leakage*

$$\mathcal{L}(\mathsf{L}) = \mathsf{op}$$

The full proof can be read in Appendix B, and the proof sketch is as follows. Security is ensured by the underlying homomorphic encryption scheme, and allows the leakage of the operation being performed at update. This will ensure confidentiality, as well as the capability to perform correct additions over ciphertexts. Per-replica timestamps will ensure the same verifications as the ones in the original specification of [21], enforcing the expected behavior defined by $\mathsf{correct}_{\mathsf{ctr}}(\mathsf{L})$.

The protocol-specific nuances of the simulator are as follows. For every write operation, the simulator $\mathcal{S}$ receives the operation and generates a dummy ciphertext (encrypts zero). These two values will be the trace of write operations. The trace of read operations is calculated according to the replica state.

- Hop 1 - All operations are stored in a global log L. Read executes $\mathsf{correct}_{\mathsf{ctr}}(\mathsf{L})$ to produce the client output. This is possible from the construction and the correctness and additively homomorphic properties of Δ: on every oracle call we are updating L of id with the same operations reproduced on their respective states.
- Hop 2 - Replace every client-side encryption with an encryption of the value 0. This can be done by the simulator (only requires the public key), and is indistinguishable from the previous hop, as the difference between these games is an instantiation of the indistinguishability game of Δ.

Discussion. Our design explicitly reveals the operation being performed, for the replica to know which structure will receive the addition. Observe that we can reduce this leakage by having all write operations produce two ciphertexts, one for increments and one for decrements, where the client will simply encrypt 0 as the operation not being performed. This requires additional client-side computation (encryption), as well as larger update messages (both the ciphertexts must be sent).

4.4 Bounded Counter

As an extension to the counter CRDT, Shapiro et. al. [24] suggest the value in enforcing numeric invariants over these distributed datatypes (e.g. $x \geq K$) for enforcing application correctness. CRDT counters enforcing such invariants are often designed following concepts from the escrow transactional model [19], where the difference between the actual value of the counter and its upper or lower bound is seen as a cumulative set of rights that enables said operations. E.g. a counter of value N with lower bound 0 can be seen as having N rights, which are consumed as the counter is decremented, and created as the counter is incremented. These CRDT counters are known as Bounded Counters, which can perform five operations:

- value(), which returns the counter value.
- inc(v), which increments v to the value.
- dec(v), which decrements v to the value.
- rights(), which returns the local rights of the replica.
- tran(v, id), which transfers v rights from the target replica to replica id.

All of these operations can fail, if the consequence of applying it breaks the underlying invariant. E.g. if a replica has 3 rights to a counter and is requested to transfer 5 to any other replica.

A natural implementation of the bounded counter is structurally similar to the counter, as the CRDT has to keep track of how many rights each replica has, and how many it has sent/received. As such, on top of using comparisons and additions, it also has the server check the invariant. We can reduce the need for comparisons by instead using per-replica Lamport clocks (similar to the previous construction), which leaves additions and invariant checks. The concurrency semantics for the bounded counter are similar to the previous counter, with an additional step for verifying the invariant. We omit these for brevity.

Now observe that merges will never break the invariant, as each individual replica never adds (or subtracts) more than what it has the rights to. This means that all operations in which the invariant must be checked involve interaction with the client, allowing these to be off-loaded to the client. Given these transformations, the only remaining computations on encoded values are additions, and again we can use an additively homomorphic encryption scheme.

Our transformation is similar to that of the counter. We build on the protocol of [2] and perform two main functional changes: i. we use per-replica operation counts to establish freshness of updates (same as before), and ii. upon updates, we delegate to the client the verification of the invariant. After this verification, the client sends an encryption of either the operation value (in case of success),

or of a neutral element to the operation (in case of failure). This allows replica-side processing of both successful and unsuccessful operations without disclosing the result of invariant validation.

The techniques for adapting the protocol follow a very similar approach as the secure counter, with an added step of straightforward client-side invariant verification. For succinctness, we detail the full construction in Appendix C.

Security. We argue that our construction is a secure bounded CRDT counter with additional leakage of the operation, and target replica for right transition (tran), i.e.

Theorem 4.4. *If Δ is an IND additively homomorphic encryption scheme, the construction of Π_{bctr} is a secure CRDT with leakage*

$$\mathcal{L}(\mathsf{L}) = (\mathsf{op}, \mathsf{id}_t)$$

where $(\cdot, \mathsf{id}_t) \leftarrow v$ for $\mathsf{op} = \mathsf{tran}$ and $\mathsf{id}_t = \epsilon$ otherwise.

The security argument is similar to that of the previous counter, as the underlying encryption scheme allows for operations to transfer encryptions indistinguishable from dummy values. The main differences are that query now leaks the operation being performed op and, if $\mathsf{tran}(v, \mathsf{id})$, the replica id to which it is transferring to. update now contains two additional ciphertexts $(\mathsf{cph}_1, \mathsf{cph}_2)$, for client-side validation of the invariant.

The initialization stage is similar to the previous proof. For every write operation, the simulator receives the operation and the replica receiving the rights (if that is the case) but does not know the actual value. It will instead produce a new encryption of 0, to emulate the updated value, and add either to the increment, decrement, or to the replica receiving the value. The values presented in the trace are $\mathsf{cph}_1, \mathsf{cph}_2$ following the specifications of update, alongside the operation and the produced dummy encryption. For every read operation, the simulator will provide the two ciphertexts within the simulated replica state as trace, as well as the received operation.

Observe that designing our CRDT to send an encryption of the neutral value when the invariant fails allows us to conceal when an operation has no effect due to this restriction. Formally, this allows the simulator to present an indistingishable dummy ciphertext that can correspond either to the value or to 0. The proof is done in three hops, two similar to those of the previous proof and one additional to enforce the invariant. The proof sketch is as follows.

- Hop 1 - All operations are stored in a global log L. Read parses over L to produce the client output, filtering operations that break the local invariant. The reasoning here is analogous to the previous proof (in the proposed construction, the operations filtered from L are also excluded by the client).
- Hop 2 - Read executes $\mathsf{correct}_{\mathsf{bctr}}(\mathsf{L})$. The only difference here is that now we are exactly following the description of $\mathcal{F}$, which means that we must also account for the global invariant. The argument for this hop is similar as the one in [2] regarding how the concurrent validations ensure not breaking the global invariant.
- Hop 3 - Replace every client-side encryption with an encryption of value 0, just like the previous proof.

Discussion. We can also reduce the leakage of both operations, but at a much steeper cost than the previous design. Hiding queries duplicates the size of replica-client messages, as the replica must now prepare and send 4 ciphertexts (two for value queries, and two for rights).

Hiding updates is possible via the same strategy as before, but if we do not want to reveal the replica receiving the rights, the client must now prepare $N + 1$ encryptions instead of 1, encrypting the neutral value for parts of the state that must remain unchanged. The benefit is that we can now perform the same update operation on the replica-side without knowing if what occured was an increment, decrement, or right transfer, and if the latter, to which replica the rights were transferred to.

5 IMPLEMENTATION AND EVALUATION

We implemented a prototype version of our secure CRDT constructions, integrating them in AntidoteDB [1], a replicated NoSQL database that uses CRDTs as the data model. AntidoteDB's core is implemented in Erlang, while there are clients in multiple programming languages. As such, we adapted a Python client to integrate our secure CRDT operations, and modified the Erlang core only when strictly necessary.

Concretely, for AntidoteDB's Register and Set CRDTs operations, we extended the client to encrypt/decrypt data before storage. Given that server-side operations are seamless over encrypted data, no adaptation of the server-side Erlang core is necessary. For the Counter and Bounded Counter CRDTs, it is necessary to modify both clients, to ensure consistency of arithmetic over encoded data. For the bounded counter, we further leveraged AntidoteDB's transactions to implement its logic, ensuring that the client maintains a consistent view of the state between read/write operations, and that there is no local concurrent operation that might compromise invariant verification.

Regarding cryptographic computations, we used AES-OFB with random IVs and a 128 bit key for standard encryption operations on the Register, AES-OFB with fixed IVs (to ensure determinism) and 128 bit key for the Set, and the Paillier cryptosystem with a 2048 bit key for both counters.

5.1 Experimental Evaluation

This experimental evaluation section aims to assess the performance and scalability overhead of our secure CRDT constructions, when compared to their non-secure versions.

Our experiments were performed in a cluster with seven machines, where two acted as servers and the others executed multiple clients in parallel. The server machines had an AMD EPYC 7281 16-core 2.1GHz CPU and 128GB of RAM each. Amongst the client machines, three had the same CPU and RAM of the server, while the other two had two Intel Xeon E5-2620 v2 6-core 2.1GHz CPU each and 64GB of RAM. With this setup we were able to saturate the servers with 128 clients in parallel. Communication between machines was done through a one gigabit network.

We ran a micro-benchmark, where clients execute operations in a closed loop for 2 minutes. For each construction, we ran multiple experiments, increasing the number of clients until the servers were saturated (from 8 to 128 clients). The size of data objects stored in maps and register was 2500 bytes. Results are presented in the form of *latency* $\times$ *throughput plots*, where the x-axis represents the throughput of the servers (i.e., the number of operations per second performed by the servers) and the y-axis exhibits the average

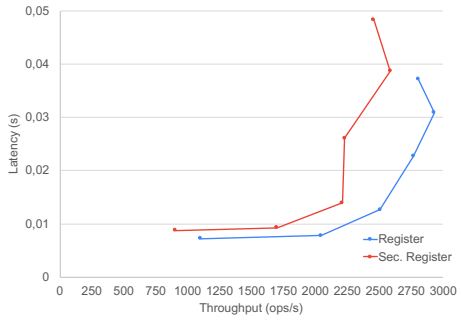

Figure 7: Throughput/latency for the plaintext and secure versions of the Register CRDT.

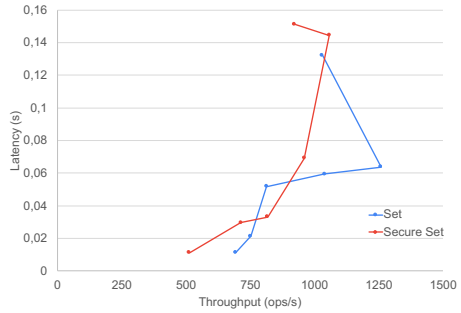

Figure 8: Throughput/latency for the plaintext and secure versions of the Set CRDT.

latency, as observed by the clients. The successive dots in a line correspond to the results of experiments with an increasing number of clients.

Register CRDT. Figure 7 compares AntidoteDB's plaintext Register CRDT and our secure construction with a workload consisting of 50% reads and 50% writes. The results show that the two versions exhibit a similar behaviour, although the secure construction has an overall higher latency and lower throughput. While the servers are not saturated, the latency of the plaintext and secure constructions is similar, with around 7-8 milliseconds per operation respectively.

The servers start becoming saturated close to 2200 ops/s for the secure construction and 2500 ops/s for the plaintext version. This can be explained by the cryptographic expansion of the data in the secure construction, which entails a larger amount of data processed and stored by the server. The small difference between the two suggests that very little overhead is imposed when considering the secure version of the Register CRDT.

Set CRDT. Figure 8 shows the results for the secure set CRDT. For these results we used a 50% gets, 35% inserts, and 15% deletes benchmark. Again, results are very similar given the small adaptations necessary for the secure version. Indeed, at some points the secure counter outperforms the plain version. This is justifiable due to small variations at the saturation point of AntidoteDB's servers, and suggests the minimal overhead of the security layer.

Counter CRDT. Figure 9 shows the results obtained for the plaintext and secure counters. For these tests we used 33% reads, 33% increments, and 33% decrements. Naturally, the counter imposes a higher performance overhead than its Register and Set counterparts. This is a consequence of relying on cryptographic schemes enabling server-side arithmetic over encoded data, which are fundamentally

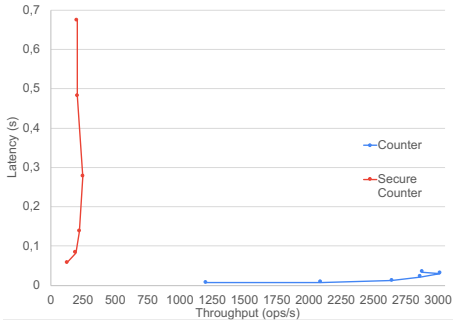

Figure 9: Throughput/latency for the plaintext, secure general construction-based and specialized versions of the Counter CRDT.

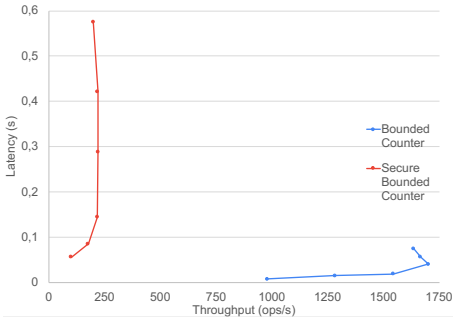

Figure 10: Throughput/latency for the plaintext and secure versions of the Bounded Counter CRDT.

richer in functionality than the previous examples. Results for the secure construction start at 127 ops/s and 57ms of latency, however latency quickly increases without any growth in throughput. The reason for this is that the server gets saturated quickly as it has to perform operations over encoded data (we note that for registers and sets, the servers only store the ciphered data, never executing operations other than comparison over that data). The plaintext version goes from 1250 to 3000 ops/s always with very small latency (from 6 to 33 ms).

Bounded Counter CRDT. Figure 10 shows the results for the plaintext and secure bounded counter. Results are very similar to the ones of the secure counter, despite the additional step for invariant preservation. Results for the secure bounded counter start with 100 ops/s and 50ms latency, then they reach a throughput cap at 220 ops/s, at which point latency starts to increase at a very steep rate. The plaintext bounded counter scales well up to 1700 ops/s.

5.2 Discussion

Our experimental results support a natural and important trade-off for designing secure CRDT solutions: the cost of security is proportional to the requirements imposed to the cryptographic scheme. For Register and Set CRDTs, we were able to rely on standard cryptographic techniques, as little interaction was necessary with the encrypted data. On the other hand, if we require server-side computations over stored data, then richer cryptographic techniques are necessary, which impose different overheads in scalability.

Comparing client/server overheads, there are several aspects that should be noted. For registers and sets, clients do all cryptographic operations and the only overhead for servers come from

the cryptographic expansion of stored data. Thus, the throughput achieved by systems storing encrypted data and plaintext data is similar, with only a small decrease of throughput on the former. For counters and bounded counters, the server has to execute operations over encrypted data (specifically, modular multiplications of large numbers, instead of normal additions), which imposes a non-negligible overhead. This has direct impact in the maximum throughput that a system with encrypted data can achieve.

6 RELATED WORK

CRDTs were originally designed for decentralized distributed systems without any security concerns [26]. In this work we propose the first formal security treatment of CRDTs, nonetheless a few previous works have studied how to protect distributed applications that leverage CRDTs for synchronization. Snapdoc [14] studied how to offer collaborative document edition with authenticated snapshots and history-privacy, where a new participant joining a document being edited can have authentication guarantees regarding the consistency of the document while simultaneously preserving the privacy of the document's edition history. Snapdoc uses CRDTs to ensure concurrent edits can be merged, however it is not a central component in their security proposal. Shoker et al. [27], in a *work in progress* report, aims to detect and tolerate malicious participants in CRDT-based systems by allowing replicas to perform operations normally and then running a Byzantine fault tolerance algorithm in the background.

CRDT security is also related with outsourcing of computations and multiparty computation [17], in the sense that clients are collaboratively performing a computation over a shared database state. Cachin et al. [6] proposed Authenticated Data Types (ADTs) for authenticated data outsourcing and computation. However their setting is restricted to a single client performing operations and single server holding the data.

Secure data storage has also been achieved through techniques other than CRDTs. However such systems typically require synchronization to detect adversarial behaviour. DepSky [4] uses Byzantine fault tolerance to ensure that replicas converge, and traditional encryption to preserve data privacy, however it does not support data computation. CryptDB [20] leverages some of the techniques we also explore in this work, including deterministic encryption and partially homomorphic encryption, however it only considers a single server. SPORC [11] supports secure group collaboration and data storage, however it only supports multiple servers in a *shared-nothing* architecture, where servers are independent.

7 CONCLUSION

We have presented the first cryptographic treatment of CRDTs, which establishes a formal framework for the validation of new secure CRDT algorithms, specifically designed to explore trade-offs between performance and information leakage.

Our results show that one can instrument correct CRDT constructions with a layer of security (e.g. standard encryption) to achieve secure CRDT solutions, provided that this encoding does not break the functional part of the CRDT. Our transformations also show that we can leverage cryptographic schemes with additively homomorphic properties to have secure CRDT solutions even when merge operations process over encoded data.

REFERENCES

[1] AntidoteDB. 2019. AntidoteDB: A planet scale, highly available, transactional database. https://www.antidotedb.eu/. (2019).

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

[3] Mihir Bellare, Alexandra Boldyreva, and Adam OâĂŹNeill. 2007. Deterministic and efficiently searchable encryption. In *Annual International Cryptology Conference*. Springer, 535–552.

[4] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)* 9, 4 (2013), 12.

[5] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*. Springer, 192–206.

[6] Christian Cachin, Esha Ghosh, Dimitrios Papadopoulos, and Björn Tackmann. 2018. Stateful multi-client verifiable computation. In *International Conference on Applied Cryptography and Network Security*. Springer, 637–656.

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

[8] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. 2002. Universally composable two-party and multi-party secure computation. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. ACM, 494–503.

[9] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*. Springer, 643–662.

[10] Steve Dunham. 2018. Notes on Notes.app. https://github.com/dunhamsteve/notesutils/blob/master/notes.md. (2018).

[11] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. 2010. SPORC: Group Collaboration using Untrusted Cloud Resources.. In *OSDI*, Vol. 10. 337–350.

[12] GitHub. 2019. Xray: An experimental next-generation Electron-based text editor. https://github.com/atom-archive/xray. (2019).

[13] Google. 2018. xi-editor: A modern editor with a backend written in Rust. https://opensource.google/projects/xi-editor. (2018).

[14] Stephan Alexander Kollmann, Martin Kleppmann, and Alastair Beresford. 2019. Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 1–23.

[15] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

[16] Sander Mak. 2014. Facebook Announces Apollo at QCon NY 2014. https://dzone.com/articles/facebook-announces-apollo-qcon. (2014).

[17] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. 2004. Fairplay-Secure Two-Party Computation System.. In *USENIX Security Symposium*, Vol. 4. San Diego, CA, USA, 9.

[18] Rimma Nehme. 2018. Azure #CosmosDB @ Build 2018: The catalyst for next generation apps. https://azure.microsoft.com/en-us/blog/azure-cosmosdb-build-2018-the-catalyst-for-next-generation-apps/?_lrsc=996193e3-e253-4a89-a50a-0b105862c2e6. (2018).

[19] Patrick E O'Neil. 1986. The escrow transactional method. *ACM Transactions on Database Systems (TODS)* 11, 4 (1986), 405–430.

[20] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 85–100.

[21] Nuno Preguiça. 2018. Conflict-free Replicated Data Types: An Overview. *arXiv preprint arXiv:1806.10254* (2018).

[22] Michal Ptaszek. 2014. Scaling LoL Chat to 70 Million Players. https://www.slideshare.net/michalptaszek/strange-loop-presentation. (2014).

[23] RIAK. 2019. RIAK Documentation: Data Types. https://docs.riak.com/riak/kv/2.2.3/learn/concepts/crdts/. (2019).

[24] Marc Shapiro, Annette Bieniusa, Nuno Preguiça, Valter Balegas, and Christopher Meiklejohn. 2018. Just-Right Consistency: reconciling availability and safety. *arXiv preprint arXiv:1801.06340* (2018).

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

[26] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[27] Ali Shoker, Houssam Yactine, and Carlos Baquero. 2017. As secure as possible eventual consistency: Work in progress. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, 5.

A PROOF OF REGISTER CONSTRUCTION

In game $G_1^{\mathcal{Z}}$, we replace the execution of Π_{reg} with its idealized version with $\mathcal{F}$ and $\mathcal{S}$. Functionality $\mathcal{F}$ will maintain global log L, updated on write and read, and by the simulator on prop and merge. $\mathcal{F}$ produces the output of read by calling correct on the global log. Since the read operations of registers are always get and write operations are always set, these are stored as such in L. Recall that correct CRDTs imply total leakage, so the trivial simulator can simply call the protocol specification to generate indistinguishable traces and states. These reconstructions are done with process(), which executes as follows:

- Call $\mathcal{L}$ to receive all the execution log.
- Update the trace and states of each replica using Π_{reg}.

Again, observe that since $\mathcal{L}$ returns the full log, $\mathcal{S}$ can simply execute the protocol to emulate it. We argue that these games are identical, given the correctness of Π_{reg}, i.e.

$$|\Pr[G_0^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}] - \Pr[G_1^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}]| = 0$$

If there exists an environment $\mathcal{Z}$ that successfully distinguishes these two games, we demonstrate that there also exists a successful environment $\mathcal{Z}'$ against the correctness game of Π_{reg}. $\mathcal{Z}'$ emulates $G_1^{\mathcal{Z}}$ as follows: with the exception of the calls to Θ, all other operations are replaced by their respective oracles in the correctness game of Π_{reg}. Indeed, if we are in the real world of the correctness game, this exactly matches $G_0^{\mathcal{Z}}$; and if we are in the ideal world of the correctness game with the trivial simulator, this exactly matches $G_1^{\mathcal{Z}}$. Given that the differences between the two games are exactly that, and that our assumption ensures that no $\mathcal{Z}'$ can break the correctness of Π_{reg}, then no $\mathcal{Z}$ can distinguish our two games.

In game $G_2^{\mathcal{Z}}$, we replace the output of read by having it compute over another idealized structure L' that stores the original values of write, instead of having them encrypted and decrypted at the end. We argue that the behavior of these games is identical given the seamless behavior of the register CRDT on encoded values.

$$|\Pr[G_2^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}] - \Pr[G_1^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}]| = 0$$

The only difference between these two games is that on $G_2^{\mathcal{Z}}$ we have the output of read being computed over L' with plaintext values, and on $G_1^{\mathcal{Z}}$ we have the output of read being computed over L with encryptions of the same values, decrypted afterwards. Under our assumption, we know that

$$\forall \mathsf{L}' : \mathsf{correct}(\mathsf{L}') = \mathsf{decode}(\mathsf{correct}(\mathsf{map}(\mathsf{encode}, \mathsf{L}')))$$

thus the outputs of read are the same.

Observe that the encrypted values of st are no longer used for read. As such, game $G_3^{\mathcal{Z}}$ we replace encrypted states $\mathsf{st}_{\mathsf{id}}$ with dummy messages of the same length. Since $\mathcal{Z}$ does not have access to key, we upper bound the distance between these two games by constructing an adversary $\mathcal{B}$ against the IND-CPA security of Θ, such that

$$|\Pr[G_2^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}] - \Pr[G_3^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}]| \leq \mathsf{Adv}^{\text{IND-CPA}}_{\Theta,\mathcal{B}}(\lambda)$$

Adversary $\mathcal{B}$ simulates the environment of $G_3^{\mathcal{Z}}$ by replacing the encryption on the write operation by $\mathsf{Encrypt}((\mathsf{op}, v, c), \{0\}^{|v|})$. $\mathcal{B}$ presents the result of $G_3^{\mathcal{Z}}$ as the guessing bit of $\text{IND-CPA}_{\Theta,\mathcal{B}}$.

Game $G0_{\Pi_{\text{reg}},\mathcal{Z},\mathcal{A}}(n)$:
$T \leftarrow \epsilon$; For $i, j \in [n] : p_{i,j} \leftarrow [\,]$
$\mathsf{prv}_u, \mathsf{prv}_q \leftarrow\$\ \Theta.\mathsf{Gen}()$; $\mathsf{pub} \leftarrow \epsilon$
For $\mathsf{id} \in [n] : \mathsf{st}_{\mathsf{id}} \leftarrow \Pi_{\text{reg}}.\mathsf{init}(\mathsf{id}, N)$
$b \leftarrow\$\ \mathcal{Z}^{\mathcal{A},\mathsf{write},\mathsf{read}}(\mathsf{pub}, n)$

Oracle write(id, op, v):
$\mathsf{cph} \leftarrow\$\ \Theta.\mathsf{Enc}(\mathsf{prv}_u, v)$
$\langle \epsilon \mid \mathsf{st}_{\mathsf{id}} \rangle_t \leftarrow \Pi_{\text{reg}}.\mathsf{update}\langle \mathsf{op}, \mathsf{cph} \mid \mathsf{st}_{\mathsf{id}} \rangle$
$T \leftarrow T \| t$
Return ϵ

Oracle prop(i, j):
$p \leftarrow \mathsf{st}_i$
$p_{i,j} \leftarrow p_{i,j} \,||\, [p]$
Return p

Oracle read(id, op):
$\langle \mathsf{cph} \mid \cdot \rangle_t \leftarrow\$\ \Pi_{\text{reg}}.\mathsf{query}\langle \mathsf{op} \mid \mathsf{st}_{\mathsf{id}} \rangle$
$r \leftarrow \Theta.\mathsf{Dec}(\mathsf{prv}_q, \mathsf{cph})$
$T \leftarrow T \| t$
Return r

Oracle corrupt(id):
Return $\mathsf{st}_{\mathsf{id}}$

Oracle trace( ):
Return T

Oracle merge(i, j):
$p \leftarrow \mathsf{head}\ p_{i,j}$
$\mathsf{st}_i \leftarrow\$\ (\mathsf{st}_i \,||\, p)$
$p_{i,j} \leftarrow \mathsf{tail}\ p_{i,j}$

Game $G1_{\Pi_{\text{reg}},\mathcal{Z},\mathcal{A}}(n)$:
$\mathsf{L} \leftarrow \{\}$; $c \leftarrow 0$
$T \leftarrow \epsilon$; For $i, j \in [n] : p_{i,j} \leftarrow [\,]$
$\mathsf{prv}_u, \mathsf{prv}_q \leftarrow\$\ \Theta.\mathsf{Gen}()$; $\mathsf{pub} \leftarrow \epsilon$
For $\mathsf{id} \in [n] : \mathsf{st}_{\mathsf{id}} \leftarrow \Pi_{\text{reg}}.\mathsf{init}(\mathsf{id}, N)$
$b \leftarrow\$\ \mathcal{Z}^{\mathcal{A},\mathsf{write},\mathsf{read}}(\mathsf{pub}, n)$

Oracle write(id, op, v):
$\mathsf{cph} \leftarrow\$\ \Theta.\mathsf{Enc}(\mathsf{prv}_u, v)$
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \mathsf{write}, \mathsf{id}, \mathsf{set}, \mathsf{cph})\}$
Return ϵ

Oracle prop(i, j):
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \mathsf{snap}, \mathsf{id}_i, \mathsf{id}_j)\}$
process()
$p \leftarrow \mathsf{st}_i$
$p_{i,j} \leftarrow p_{i,j} \,||\, [p]$
Return p

Oracle read(id, op):
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \mathsf{read}, \mathsf{id}, \mathsf{get})\}$
$\mathsf{cph} \leftarrow \mathsf{correct}(\mathsf{L})$
$r \leftarrow \Theta.\mathsf{Dec}(\mathsf{prv}_q, \mathsf{cph})$
Return r

Oracle corrupt(id):
process()
Return $\mathsf{st}_{\mathsf{id}}$

Oracle trace( ):
process()
Return T

Oracle merge(i, j):
$c \leftarrow c + 1$
$C \leftarrow \{\, \hat{c} \mid (\hat{c}, \mathsf{snap}, \mathsf{id}_i, \mathsf{id}_j) \in \mathsf{L} \,\}$
If $C \neq \emptyset$:
  $\hat{c} = \min(C)$
  $\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \mathsf{set}, \mathsf{id}_i, \mathsf{id}_j, \hat{c})\}$
  $\mathsf{L} \leftarrow \mathsf{L} \setminus \{(\hat{c}, \mathsf{snap}, \mathsf{id}_i, \mathsf{id}_j)\}$
process()
$p \leftarrow \mathsf{head}\ p_{i,j}$
$\mathsf{st}_i \leftarrow\$\ (\mathsf{st}_i \,||\, p)$
$p_{i,j} \leftarrow \mathsf{tail}\ p_{i,j}$

Figure 11: First hop, correctness of Π_{reg}.

Let the internal bit of IND-CPA be 0. Then, the environment presented to $\mathcal{Z}$ is exactly the same as that of $G_2^{\mathcal{Z}}$. Now, let the internal bit of IND-CPA be 1. The environment presented to $\mathcal{Z}$ is exactly the same as that of $G_3^{\mathcal{Z}}$. Given that the difference between the two games is exactly that, then the advantage of $\mathcal{Z}$ distinguishing between these two games is exactly that of breaking the IND-CPA security of Θ.

In our final game $G_3^{\mathcal{Z}}$, we can produce all the information presented in the adversarial interface using the leakage of $\mathcal{L}$. As such, we can now define a simulator that makes $G_3^{\mathcal{Z}}$ match the ideal world. This simulator $\mathcal{S}$ is described in Figure 14.

The simulation strategy is as follows: the states and traces shown to the environment are those emulated by the trivial simulator of Π_{reg} (now referred to as $\mathcal{S}_{\text{reg}}.\mathsf{process}$). In this register will be stored dummy encryptions known to $\mathcal{S}$. To construct these encryptions, $\mathcal{S}$ will call $\mathcal{F}.\mathsf{leak}$ to obtain the correct sizes of the register values. These will constitute all operations placed in Π_{reg}, and thus $\mathcal{S}$ has all the information necessary to perfectly emulate the functionality of Π_{reg}, by maintaining its global log L and responding to $\mathcal{L}_{\mathsf{Reg}}(\mathsf{L})$ with L, as is required by the trivial simulator $\mathcal{S}_{\text{reg}}.\mathsf{process}$.

Now, the only differences between $G_3^{\mathcal{Z}}$ and $\mathsf{Ideal}^{\mathcal{F},\mathcal{Z},\mathcal{S}}$ instantiated with the simulator of Figure 14 are when traces and states of L and L' are constructed, as in $G_3^{\mathcal{Z}}$ they are processed as soon as the operations occur, and observed afterwards, and in $\mathsf{Ideal}^{\mathcal{F},\mathcal{Z},\mathcal{S}}$ they

Game $G1_{\Pi_{reg},\mathcal{Z},\mathcal{A}}(n)$:
$L \leftarrow \{\}; c \leftarrow 0$
$T \leftarrow \epsilon$; For $i, j \in [n] : p_{i,j} \leftarrow [\,]$
$prv_u, prv_q \leftarrow\$\ \Theta.Gen(); pub \leftarrow \epsilon$
For id $\in [n]$:
  $st_{id} \leftarrow \Pi_{reg}.init(id, N)$
$b \leftarrow\$\ \mathcal{Z}^{\mathcal{A},write,read}(pub, n)$

Oracle write(id, op, v):
$cph \leftarrow\$\ \Theta.Enc(prv_u, v)$
$c \leftarrow c + 1$
$L \leftarrow L \cup \{(c, write, id, set, cph)\}$
Return ϵ

Oracle prop(i, j):
$c \leftarrow c + 1$
$L \leftarrow L \cup \{(c, snap, id_i, id_j)\}$
process()
$p \leftarrow st_i$
$p_{i,j} \leftarrow p_{i,j} \,||\, [p]$
Return p

Oracle read(id, op):
$c \leftarrow c + 1$
$L \leftarrow L \cup \{(c, read, id, get)\}$
$cph \leftarrow correct(L)$
$r \leftarrow \Theta.Dec(prv_q, cph)$
Return r

Oracle corrupt(id):
process()
Return st_{id}

Oracle trace( ):
process()
Return T

Oracle merge(i, j):
$c \leftarrow c + 1$
$C \leftarrow \{\, \hat{c} \mid (\hat{c}, snap, id_i, id_j) \in L \,\}$
If $C \neq \emptyset$:
  $\hat{c} = \min(C)$
  $L \leftarrow L \cup \{(c, set, id_i, id_j, \hat{c})\}$
  $L \leftarrow L \setminus \{(\hat{c}, snap, id_i, id_j)\}$
process()
$p \leftarrow head\ p_{i,j}$
$st_i \leftarrow\$\ (st_i \,||\, p)$
$p_{i,j} \leftarrow tail\ p_{i,j}$

Game $G2_{\Pi_{reg},\mathcal{Z},\mathcal{A}}(n)$:
$L' \leftarrow \{\}; c' \leftarrow 0$
$L \leftarrow \{\}; c \leftarrow 0$
$T \leftarrow \epsilon$; For $i, j \in [n] : p_{i,j} \leftarrow [\,]$
$prv_u \leftarrow\$\ \Theta.Gen(); pub \leftarrow \epsilon$
For id $\in [n]$:
  $st_{id} \leftarrow \Pi_{reg}.init(id, N)$
$b \leftarrow\$\ \mathcal{Z}^{\mathcal{A},write,read}(pub, n)$

Oracle write(id, op, v):
$c' \leftarrow c' + 1$
$L' \leftarrow L' \cup \{(c', write, id, set, v)\}$
$cph \leftarrow\$\ \Theta.Enc(prv_u, v)$
$c \leftarrow c + 1$
$L \leftarrow L \cup \{(c, write, id, set, cph)\}$
Return ϵ

Oracle prop(i, j):
$c' \leftarrow c' + 1$
$L \leftarrow L \cup \{(c', snap, id_i, id_j)\}$
$c \leftarrow c + 1$
$L \leftarrow L \cup \{(c, snap, id_i, id_j)\}$
process()
$p \leftarrow st_i$
$p_{i,j} \leftarrow p_{i,j} \,||\, [p]$
Return p

Oracle read(id, op):
$c' \leftarrow c' + 1$
$L' \leftarrow L' \cup \{(c', read, id, get)\}$
$r \leftarrow correct(L')$
$c \leftarrow c + 1$
$L \leftarrow L \cup \{(c, read, id, get)\}$
Return r

Oracle corrupt(id):
process()
Return st_{id}

Oracle trace( ):
process()
Return T

Oracle merge(i, j):
$c' \leftarrow c' + 1$
$C' \leftarrow \{\, \hat{c}' \mid (\hat{c}', snap, id_i, id_j) \in L' \,\}$
If $C' \neq \emptyset$:
  $\hat{c}' = \min(C')$
  $L' \leftarrow L' \cup \{(c', set, id_i, id_j, \hat{c}')\}$
  $L' \leftarrow L' \setminus \{(\hat{c}, snap, id_i, id_j)\}$
$c \leftarrow c + 1$
$C \leftarrow \{\, \hat{c} \mid (\hat{c}, snap, id_i, id_j) \in L \,\}$
If $C \neq \emptyset$:
  $\hat{c} = \min(C)$
  $L \leftarrow L \cup \{(c, set, id_i, id_j, \hat{c})\}$
  $L \leftarrow L \setminus \{(\hat{c}, snap, id_i, id_j)\}$
process()
$p \leftarrow head\ p_{i,j}$
$st_i \leftarrow\$\ (st_i \,||\, p)$
$p_{i,j} \leftarrow tail\ p_{i,j}$

Figure 12: Second hop, functionality-preserving encoding $\mathcal{E}_{gen}$.

Game $G2_{\Pi_{reg},\mathcal{Z},\mathcal{A}}(n)$:
$L' \leftarrow \{\}; c' \leftarrow 0$
$L \leftarrow \{\}; c \leftarrow 0$
$T \leftarrow \epsilon$; For $i, j \in [n] : p_{i,j} \leftarrow [\,]$
$prv_u \leftarrow\$\ \Theta.Gen(); pub \leftarrow \epsilon$
For id $\in [n]$:
  $st_{id} \leftarrow \Pi_{reg}.init(id, N)$
$b \leftarrow\$\ \mathcal{Z}^{\mathcal{A},write,read}(pub, n)$

Oracle write(id, op, v):
$c' \leftarrow c' + 1$
$L' \leftarrow L' \cup \{(c', write, id, set, v)\}$
$cph \leftarrow\$\ \Theta.Enc(prv_u, v)$
$c \leftarrow c + 1$
$L \leftarrow L \cup \{(c, write, id, set, cph)\}$
Return ϵ

Oracle prop(i, j):
$c' \leftarrow c' + 1$
$L \leftarrow L \cup \{(c', snap, id_i, id_j)\}$
$c \leftarrow c + 1$
$L \leftarrow L \cup \{(c, snap, id_i, id_j)\}$
process()
$p \leftarrow st_i$
$p_{i,j} \leftarrow p_{i,j} \,||\, [p]$
Return p

Oracle read(id, op):
$c' \leftarrow c' + 1$
$L' \leftarrow L' \cup \{(c', read, id, get)\}$
$r \leftarrow correct(L')$
$c \leftarrow c + 1$
$L \leftarrow L \cup \{(c, read, id, get)\}$
Return r

Oracle corrupt(id):
process()
Return st_{id}

Oracle trace( ):
process()
Return T

Oracle merge(i, j):
$c' \leftarrow c' + 1$
$C' \leftarrow \{\, \hat{c}' \mid (\hat{c}', snap, id_i, id_j) \in L' \,\}$
If $C' \neq \emptyset$:
  $\hat{c}' = \min(C')$
  $L' \leftarrow L' \cup \{(c', set, id_i, id_j, \hat{c}')\}$
  $L' \leftarrow L' \setminus \{(\hat{c}, snap, id_i, id_j)\}$
$c \leftarrow c + 1$
$C \leftarrow \{\, \hat{c} \mid (\hat{c}, snap, id_i, id_j) \in L \,\}$
If $C \neq \emptyset$:
  $\hat{c} = \min(C)$
  $L \leftarrow L \cup \{(c, set, id_i, id_j, \hat{c})\}$
  $L \leftarrow L \setminus \{(\hat{c}, snap, id_i, id_j)\}$
process()
$p \leftarrow head\ p_{i,j}$
$st_i \leftarrow\$\ (st_i \,||\, p)$
$p_{i,j} \leftarrow tail\ p_{i,j}$

Game $G3_{\Pi_{reg},\mathcal{Z},\mathcal{A}}(n)$:
$L' \leftarrow \{\}; c' \leftarrow 0$
$L \leftarrow \{\}; c \leftarrow 0$
$T \leftarrow \epsilon$; For $i, j \in [n] : p_{i,j} \leftarrow [\,]$
$prv_u \leftarrow\$\ \Theta.Gen(); pub \leftarrow \epsilon$
For id $\in [n]$:
  $st_{id} \leftarrow \Pi_{reg}.init(id, N)$
$b \leftarrow\$\ \mathcal{Z}^{\mathcal{A},write,read}(pub, n)$

Oracle write(id, op, v):
$c' \leftarrow c' + 1$
$L' \leftarrow L' \cup \{(c', write, id, set, v)\}$
$cph \leftarrow\$\ \Theta.Enc(prv_u, \{0\}^{|v|})$
$c \leftarrow c + 1$
$L \leftarrow L \cup \{(c, write, id, set, cph)\}$
Return ϵ

Oracle prop(i, j):
$c' \leftarrow c' + 1$
$L \leftarrow L \cup \{(c', snap, id_i, id_j)\}$
$c \leftarrow c + 1$
$L \leftarrow L \cup \{(c, snap, id_i, id_j)\}$
process()
$p \leftarrow st_i$
$p_{i,j} \leftarrow p_{i,j} \,||\, [p]$
Return p

Oracle read(id, op):
$c' \leftarrow c' + 1$
$L' \leftarrow L' \cup \{(c', read, id, get)\}$
$r \leftarrow correct(L')$
$c \leftarrow c + 1$
$L \leftarrow L \cup \{(c, read, id, get)\}$
Return r

Oracle corrupt(id):
process()
Return st_{id}

Oracle trace( ):
process()
Return T

Oracle merge(i, j):
$c' \leftarrow c' + 1$
$C' \leftarrow \{\, \hat{c}' \mid (\hat{c}', snap, id_i, id_j) \in L' \,\}$
If $C' \neq \emptyset$:
  $\hat{c}' = \min(C')$
  $L' \leftarrow L' \cup \{(c', set, id_i, id_j, \hat{c}')\}$
  $L' \leftarrow L' \setminus \{(\hat{c}, snap, id_i, id_j)\}$
$c \leftarrow c + 1$
$C \leftarrow \{\, \hat{c} \mid (\hat{c}, snap, id_i, id_j) \in L \,\}$
If $C \neq \emptyset$:
  $\hat{c} = \min(C)$
  $L \leftarrow L \cup \{(c, set, id_i, id_j, \hat{c})\}$
  $L \leftarrow L \setminus \{(\hat{c}, snap, id_i, id_j)\}$
process()
$p \leftarrow head\ p_{i,j}$
$st_i \leftarrow\$\ (st_i \,||\, p)$
$p_{i,j} \leftarrow tail\ p_{i,j}$

Figure 13: Third hop, IND-CPA of Θ.

are only produced upon calling process. This is a standard eager processing argument, as the inputs used for constructing traces at the time of write and read on $G_3^{\mathcal{Z}}$ are exactly the same as those used later on $Ideal^{\mathcal{F},\mathcal{Z},\mathcal{S}}$, computed upon request on process.

Let λ be the security parameter and $\mu(\lambda)$ a negligible function in it. To conclude, we have that

$$\begin{aligned} Adv^{Sec}_{\Pi,\mathcal{Z}} &= (|Pr[G_0^{\mathcal{Z}}(n) \Rightarrow T] - |Pr[G_1^{\mathcal{Z}}(n) \Rightarrow T])+ \\ &\quad (|Pr[G_1^{\mathcal{Z}}(n) \Rightarrow T] - Pr[G_2^{\mathcal{Z}}(n) \Rightarrow T]|)+ \\ &\quad (|Pr[G_2^{\mathcal{Z}}(n) \Rightarrow T] - Pr[G_3^{\mathcal{Z}}(n) \Rightarrow T]|) \\ &\leq Adv^{IND\text{-}CPA}_{\Theta,\mathcal{B}}(\lambda) \\ &\leq \mu(\lambda) \end{aligned}$$

Algorithm init(n):
$\mathsf{L} \leftarrow \{\}$; $c \leftarrow 0$
$T \leftarrow \epsilon$; For $i, j \in [n] : p_{i,j} \leftarrow [\,]$
key $\leftarrow\$$ Θ.Gen(); $c \leftarrow 0$
$s \leftarrow 0$
For id $\in [n] : \mathsf{st}_{\mathsf{id}} \leftarrow \epsilon$
Return ϵ

Algorithm prop(i, j):
$\mathcal{F}$.snap(i, j)
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{snap}, \mathsf{id}_i, \mathsf{id}_j)\}$
process()
$p \leftarrow \mathsf{st}_i$
$p_{i,j} \leftarrow p_{i,j} \,||\, [p]$
Return p

Algorithm merge(i, j):
$\mathcal{F}$.set(i, j)
$c \leftarrow c + 1$
$C \leftarrow \{\, \hat{c} \mid (\hat{c}, \text{snap}, \mathsf{id}_i, \mathsf{id}_j) \in \mathsf{L} \,\}$
If $C \neq \emptyset$:
  $\hat{c} = \min(C)$
  $\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{set}, \mathsf{id}_i, \mathsf{id}_j, \hat{c})\}$
  $\mathsf{L} \leftarrow \mathsf{L} \setminus \{(\hat{c}, \text{snap}, \mathsf{id}_i, \mathsf{id}_j)\}$
process()
$p \leftarrow$ head $p_{i,j}$
$\mathsf{st}_i \leftarrow\$ (\mathsf{st}_i \,||\, p)$
$p_{i,j} \leftarrow$ tail $p_{i,j}$

Algorithm trace( ):
process()
Return T

Algorithm corrupt(id):
process()
Return $\mathsf{st}_{\mathsf{id}}$

Algorithm process( ):
$\mathsf{O} \leftarrow \mathcal{F}$.leak()
For $(c', \text{op}, l) \in \mathsf{O} \wedge (s < c')$
  If op = write:
    cph $\leftarrow\$$ Θ.Enc(key, $\{0\}^l$)
    $c \leftarrow c + 1$
    $\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{write}, \text{id}, \text{set}, \text{cph})\}$
    $\mathcal{S}_{\text{reg}}$.process()
  Else:
    $c \leftarrow c + 1$
    $\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{read}, \text{id}, \text{get})\}$
    $\mathcal{S}_{\text{reg}}$.process()

Figure 14: Simulator for register construction proof. $\mathcal{F}$.leak returns the set of operation identifiers O, with corresponding input/output lengths.

and Theorem 4.1 follows.

B PROOF OF COUNTER CONSTRUCTION

Our proof is a sequence of three games, presented in Figures 15 and 16.

Game $\mathrm{G}_0^{\mathcal{Z}}$ is the real world of Figure 3, extended with the protocol of Figure 6. In game $\mathrm{G}_1^{\mathcal{Z}}$, we have an idealized data set L, containing all global operations. The only functional difference is in read, where instead of using a state that is composed of ciphertexts are are constructed via multiple additions on write, we refer to the L structure that contains all the operations seen, and instead call $\mathsf{correct}_{\mathsf{ctr}}(\mathsf{L})$. We argue that the behavior of these games is identical, given the correctness of Δ, i.e.

$$|\Pr[\mathrm{G}_0^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}] - \Pr[\mathrm{G}_1^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}]| = 0$$

To show equality of these two games we prove an invariant that shows that the sum of operations $C[\mathsf{id}]$ constructed by $\mathsf{correct}_{\mathsf{ctr}}(\mathsf{L})$ (used on $\mathrm{G}_1^{\mathcal{Z}}$) produces exactly the same result as what is encoded by $\mathsf{st}_{\mathsf{id}}$ (used on $\mathrm{G}_0^{\mathcal{Z}}$).

$$\forall \mathsf{id} : \mathsf{decryptState}(\mathsf{st}_{\mathsf{id}}) = \mathsf{sum}(C[\mathsf{id}])$$

Where $\mathsf{sum}(C[\mathsf{id}])$ computes the counter value of $C[\mathsf{id}]$ according to $\mathsf{correct}_{\mathsf{ctr}}$, and $\mathsf{decryptState}(\mathsf{st})$ denotes the operation that presents the internal value of $\mathsf{st}_{\mathsf{id}}$:

$$\begin{aligned}
&\mathsf{cph}_1 \leftarrow\$ \ \Delta.\mathsf{Enc}(\mathsf{prv}_u, 0)\\
&\mathsf{cph}_2 \leftarrow\$ \ \Delta.\mathsf{Enc}(\mathsf{prv}_u, 0)\\
&\forall \mathsf{id} \in N : \mathsf{cph}_1 \leftarrow \Delta.\mathsf{Add}(\mathsf{cph}_1, \mathsf{st}.(\mathsf{fst}\, C_p[\mathsf{id}]))\\
&\forall \mathsf{id} \in N : \mathsf{cph}_2 \leftarrow \Delta.\mathsf{Add}(\mathsf{cph}_2, \mathsf{st}.(\mathsf{fst}\, C_n[\mathsf{id}]))\\
&r_1 \leftarrow \Delta.\mathsf{Dec}(\mathsf{prv}_q, \mathsf{cph}_1)\\
&r_2 \leftarrow \Delta.\mathsf{Dec}(\mathsf{prv}_q, \mathsf{cph}_2)\\
&(r_1 - r_2)
\end{aligned}$$

At beginning, the correctness of Δ shows that the invariant holds, as we will have the encryption and decryption of 0 on the left, and 0 on the right from the empty $C[\mathsf{id}]$. We now present our arguments for the rest of the oracle calls.

- read(id, op) changes nothing on both sides.
- write(id, op, v)
  - op = inc on the left performs $\Delta.\mathsf{Add}(\mathsf{cph}_1, \Delta.\mathsf{Enc}(\mathsf{pub}, v))$ before $\Delta.\mathsf{Dec}(\mathsf{sk}, \mathsf{cph})$. From the correctness of Δ and its additively homomorphic property, the result will be the an increase of v to the total value, which exactly matches the difference on the right, which adds $(\cdot, \text{write}, \mathsf{id}, \text{inc}, v)$ to the list.
  - op = dec is the same for the analogous reasoning.
- prop($\mathsf{id}_i, \mathsf{id}_j$) changes nothing on both sides.
- merge(id, up) by construction will contain a up that matches the state of some replica id_x at time of prop(id_x, id).
  As such, on the left-hand side it will update positive and negative encrypted values of outdated replica ids (tested via timestamps: $\mathsf{snd}\, C[\mathsf{id}_x] > \mathsf{snd}\, C[\mathsf{id}]$). By the construction of ($\mathsf{snd}\, C[\mathsf{id}]$) in write we can see that this will correspond exactly to a value update corresponding to the unique operations observed in id_x at time of prop(id_x, id), that are not in id.
  On the right-hand side we have $\mathsf{correct}_{\mathsf{ctr}}(\mathsf{L})$ updating $C[\mathsf{id}]$ according to the respective $(\cdot, \text{snap}, \mathsf{id}_x, \mathsf{id})$, which will extend $C[\mathsf{id}]$ with the operations in id_x that were not yet in id. Thus, let pre denote the result before the operation and post denote the result after the operation, we can see that

$$\mathsf{decryptState}_{\mathsf{post}}[\mathsf{st}_{\mathsf{id}}] - \mathsf{decryptState}_{\mathsf{pre}}[\mathsf{st}_{\mathsf{id}}] = \mathsf{sum}(C_{\mathsf{pre}}[\mathsf{id}] \cap C[\mathsf{id}_x])$$

  and the invariant holds.

In game $\mathrm{G}_2^{\mathcal{Z}}$ we replace updates with encryptions of zero. Since $\mathcal{Z}$ does not have access to key, we upper bound the distance between these two games by constructing an adversary $\mathcal{B}$ against the indistinguishability of Δ, such that

$$|\Pr[\mathrm{G}_1^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}] - \Pr[\mathrm{G}_2^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}]| \leq \mathsf{Adv}^{\mathsf{IND}}_{\Delta,\mathcal{B}}(\lambda)$$

Adversary $\mathcal{B}$ simulates the environment of $\mathrm{G}_2^{\mathcal{Z}}$ by replacing the encryption on the write operation by $\mathsf{Encrypt}(v, 0)$. $\mathcal{B}$ presents the result of $\mathrm{G}_2^{\mathcal{Z}}$ as the guessing bit of $\mathsf{IND}_{\Delta,\mathcal{B}}$.

Let the internal bit of IND be 0. Then, the environment presented to $\mathcal{Z}$ is exactly the same as that of $\mathrm{G}_1^{\mathcal{Z}}$. Now, let the internal bit of IND be 1. The environment presented to $\mathcal{Z}$ is exactly the same as that of $\mathrm{G}_2^{\mathcal{Z}}$. Given that the difference between the two games is exactly that, then the advantage of $\mathcal{Z}$ distinguishing between these two games is exactly that of breaking the IND security of Δ.

In our final game $\mathrm{G}_2^{\mathcal{Z}}$, we can produce all the information presented in the adversarial interface using the leakage of $\mathcal{L}$. As such, we can define a simulator that makes $\mathrm{G}_2^{\mathcal{Z}}$ match the ideal world. This simulator $\mathcal{S}$ is very similar to the one in Figure 14, so for conciseness we highlight the few differences.

- Instead of encrypting zeros using Ω, it will encrypt the value 0 using Δ.

Game $G0_{\Pi,\mathcal{Z},\mathcal{A}}(n)$:
$T \leftarrow \epsilon$; For $i, j \in [n] : p_{i,j} \leftarrow [\,]$
$(\text{prv}_q, \text{prv}_u, \text{pub}) \leftarrow\$\ \Pi.\text{setup}()$
For id $\in [n]$:
  $\text{cph} \leftarrow\$\ \Delta.\text{Enc}(\text{pub}, 0)$
  For $k \in [n]$:
    $C_p[k] \leftarrow (\text{cph}, 0)$
    $C_n[k] \leftarrow (\text{cph}, 0)$
  $\text{st}_{\text{id}} \leftarrow (C_p, C_n, \text{id}, n, \text{pub})$
$b \leftarrow\$\ \mathcal{Z}^{\mathcal{A},\text{write},\text{read}}(\text{pub}, n)$

Oracle write(id, op, v):
$(C_p, C_n, \text{id}, N, \text{pub}) \leftarrow \text{st}_{\text{id}}$
$\text{cph} \leftarrow\$\ \Delta.\text{Enc}(\text{prv}_u, v)$
If op = inc: $i \leftarrow p$
Else: $i \leftarrow n$
$\text{cph} \leftarrow \Delta.\text{Add}(\text{cph}, (\text{fst}\, C_i[\text{id}]))$
$t \leftarrow ((\text{snd}\, C_i[\text{id}]) + 1)$
$C_i[\text{id}] \leftarrow (\text{cph}, t)$
$\text{st}_{\text{id}} \leftarrow (C_p, C_n, \text{id}, N, \text{pub})$
$t \leftarrow (\text{cph}, \text{op})$
$T \leftarrow T \| t$
Return ϵ

Oracle prop(i, j):
$(C_p, C_n, \cdot, \cdot, \cdot) \leftarrow \text{st}_i$
$p \leftarrow (C_p, C_n)$
$p_{i,j} \leftarrow p_{i,j} \,||\, [p]$
Return p

Oracle read(id, op):
$\text{sk} \leftarrow \text{prv}_q$
$(C_p, C_n, \cdot, N, \text{pub}) \leftarrow \text{st}_{\text{id}}$
$\text{cph}_1 \leftarrow\$\ \Delta.\text{Enc}(\text{pub}, 0)$
$\text{cph}_2 \leftarrow\$\ \Delta.\text{Enc}(\text{pub}, 0)$
For id $\in N$:
  $\text{cph}_1 \leftarrow \Delta.\text{Add}(\text{cph}_1, (\text{fst}\, C_p[\text{id}]))$
  $\text{cph}_2 \leftarrow \Delta.\text{Add}(\text{cph}_2, (\text{fst}\, C_n[\text{id}]))$
$r_1 \leftarrow \Delta.\text{Dec}(\text{sk}, \text{cph}_1)$
$r_2 \leftarrow \Delta.\text{Dec}(\text{sk}, \text{cph}_2)$
$t \leftarrow (\text{cph}_1, \text{cph}_2)$
$T \leftarrow T \| t$
$o \leftarrow (r_1 - r_2)$
Return o

Oracle corrupt(id):
Return st_{id}

Oracle trace( ):
Return T

Oracle merge(i, j):
$p \leftarrow \text{head}\ p_{i,j}$
$(C_p, C_n, \text{id}, N, \text{pub}) \leftarrow \text{st}_i$
$(C'_p, C'_n) \leftarrow p$
For id $\in N$:
  If $(\text{snd}\, C'_p[\text{id}]) > (\text{snd}\, C_p[\text{id}])$:
    $C_p[\text{id}] \leftarrow C'_p[\text{id}]$
  If $(\text{snd}\, C'_n[\text{id}]) > (\text{snd}\, C_n[\text{id}])$:
    $C_n[\text{id}] \leftarrow C'_n[\text{id}]$
$\text{st}_i \leftarrow (C_p, C_n, \text{id}, N, \text{pub})$
$p_{i,j} \leftarrow \text{tail}\ p_{i,j}$

Game $G1_{\Pi,\mathcal{Z},\mathcal{A}}(n)$:
$\mathsf{L} \leftarrow \{\}$; $c \leftarrow 0$
$T \leftarrow \epsilon$; For $i, j \in [n] : p_{i,j} \leftarrow [\,]$
$(\text{prv}_q, \text{prv}_u, \text{pub}) \leftarrow\$\ \Pi.\text{setup}()$
For id $\in [n]$:
  $\text{cph} \leftarrow\$\ \Delta.\text{Enc}(\text{pub}, 0)$
  For $k \in [n]$:
    $C_p[k] \leftarrow (\text{cph}, 0)$
    $C_n[k] \leftarrow (\text{cph}, 0)$
  $\text{st}_{\text{id}} \leftarrow (C_p, C_n, \text{id}, n, \text{pub})$
$b \leftarrow\$\ \mathcal{Z}^{\mathcal{A},\text{write},\text{read}}(\text{pub}, n)$

Oracle write(id, op, v):
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{write}, \text{id}, \text{op}, v)\}$
$(C_p, C_n, \text{id}, N, \text{pub}) \leftarrow \text{st}_{\text{id}}$
$\text{cph} \leftarrow\$\ \Delta.\text{Enc}(\text{prv}_u, v)$
If op = inc: $i \leftarrow p$
Else: $i \leftarrow n$
$\text{cph} \leftarrow \Delta.\text{Add}(\text{cph}, (\text{fst}\, C_i[\text{id}]))$
$t \leftarrow ((\text{snd}\, C_i[\text{id}]) + 1)$
$C_i[\text{id}] \leftarrow (\text{cph}, t)$
$\text{st}_{\text{id}} \leftarrow (C_p, C_n, \text{id}, N, \text{pub})$
$t \leftarrow (\text{cph}, \text{op})$
$T \leftarrow T \| t$
Return ϵ

Oracle prop(i, j):
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{snap}, \text{id}_i, \text{id}_j)\}$
$(C_p, C_n, \cdot, \cdot, \cdot) \leftarrow \text{st}_i$
$p \leftarrow (C_p, C_n)$
$p_{i,j} \leftarrow p_{i,j} \,||\, [p]$
Return p

Oracle read(id, op):
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{read}, \text{id}, \text{op})\}$
$o \leftarrow \text{correct}(\mathsf{L})$
$\text{sk} \leftarrow \text{prv}_q$
$(C_p, C_n, \cdot, N, \text{pub}) \leftarrow \text{st}_{\text{id}}$
$\text{cph}_1 \leftarrow\$\ \Delta.\text{Enc}(\text{pub}, 0)$
$\text{cph}_2 \leftarrow\$\ \Delta.\text{Enc}(\text{pub}, 0)$
For id $\in N$:
  $\text{cph}_1 \leftarrow \Delta.\text{Add}(\text{cph}_1, (\text{fst}\, C_p[\text{id}]))$
  $\text{cph}_2 \leftarrow \Delta.\text{Add}(\text{cph}_2, (\text{fst}\, C_n[\text{id}]))$
$t \leftarrow (\text{cph}_1, \text{cph}_2)$
$T \leftarrow T \| t$
Return o

Oracle corrupt(id):
Return st_{id}

Oracle trace( ):
Return T

Oracle merge(i, j):
$c \leftarrow c + 1$
$C \leftarrow \{\, \hat{c} \mid (\hat{c}, \text{snap}, \text{id}_i, \text{id}_j) \in \mathsf{L} \,\}$
If $C \neq \emptyset$:
  $\hat{c} = \min(C)$
  $\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{set}, \text{id}_i, \text{id}_j, \hat{c})\}$
  $\mathsf{L} \leftarrow \mathsf{L} \setminus \{(\hat{c}, \text{snap}, \text{id}_i, \text{id}_j)\}$
$p \leftarrow \text{head}\ p_{i,j}$
$(C_p, C_n, \text{id}, N, \text{pub}) \leftarrow \text{st}_i$
$(C'_p, C'_n) \leftarrow p$
For id $\in N$:
  If $(\text{snd}\, C'_p[\text{id}]) > (\text{snd}\, C_p[\text{id}])$:
    $C_p[\text{id}] \leftarrow C'_p[\text{id}]$
  If $(\text{snd}\, C'_n[\text{id}]) > (\text{snd}\, C_n[\text{id}])$:
    $C_n[\text{id}] \leftarrow C'_n[\text{id}]$
$\text{st}_i \leftarrow (C_p, C_n, \text{id}, N, \text{pub})$
$p_{i,j} \leftarrow \text{tail}\ p_{i,j}$

Figure 15: First hop, correctness of Δ

- Instead of having the trace be the encryption, it will set it to be the pair (op, cph). It can do this because $\mathcal{L}$ also provdides op.

Game $G1_{\Pi,\mathcal{Z},\mathcal{A}}(n)$:
$\mathsf{L} \leftarrow \{\}$; $c \leftarrow 0$
$T \leftarrow \epsilon$; For $i, j \in [n] : p_{i,j} \leftarrow [\,]$
$(\text{prv}_q, \text{prv}_u, \text{pub}) \leftarrow\$\ \Pi.\text{setup}()$
For id $\in [n]$:
  $\text{cph} \leftarrow\$\ \Delta.\text{Enc}(\text{pub}, 0)$
  For $k \in [n]$:
    $C_p[k] \leftarrow (\text{cph}, 0)$
    $C_n[k] \leftarrow (\text{cph}, 0)$
  $\text{st}_{\text{id}} \leftarrow (C_p, C_n, \text{id}, n, \text{pub})$
$b \leftarrow\$\ \mathcal{Z}^{\mathcal{A},\text{write},\text{read}}(\text{pub}, n)$

Oracle write(id, op, v):
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{write}, \text{id}, \text{op}, v)\}$
$(C_p, C_n, \text{id}, N, \text{pub}) \leftarrow \text{st}_{\text{id}}$
$\text{cph} \leftarrow\$\ \Delta.\text{Enc}(\text{prv}_u, v)$
If op = inc: $i \leftarrow p$
Else: $i \leftarrow n$
$\text{cph} \leftarrow \Delta.\text{Add}(\text{cph}, (\text{fst}\, C_i[\text{id}]))$
$t \leftarrow ((\text{snd}\, C_i[\text{id}]) + 1)$
$C_i[\text{id}] \leftarrow (\text{cph}, t)$
$\text{st}_{\text{id}} \leftarrow (C_p, C_n, \text{id}, N, \text{pub})$
$t \leftarrow (\text{cph}, \text{op})$
$T \leftarrow T \| t$
Return ϵ

Oracle prop(i, j):
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{snap}, \text{id}_i, \text{id}_j)\}$
$(C_p, C_n, \cdot, \cdot, \cdot) \leftarrow \text{st}_i$
$p \leftarrow (C_p, C_n)$
$p_{i,j} \leftarrow p_{i,j} \,||\, [p]$
Return p

Oracle read(id, op):
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{read}, \text{id}, \text{op})\}$
$o \leftarrow \text{correct}(\mathsf{L})$
$\text{sk} \leftarrow \text{prv}_q$
$(C_p, C_n, \cdot, N, \text{pub}) \leftarrow \text{st}_{\text{id}}$
$\text{cph}_1 \leftarrow\$\ \Delta.\text{Enc}(\text{pub}, 0)$
$\text{cph}_2 \leftarrow\$\ \Delta.\text{Enc}(\text{pub}, 0)$
For id $\in N$:
  $\text{cph}_1 \leftarrow \Delta.\text{Add}(\text{cph}_1, (\text{fst}\, C_p[\text{id}]))$
  $\text{cph}_2 \leftarrow \Delta.\text{Add}(\text{cph}_2, (\text{fst}\, C_n[\text{id}]))$
$t \leftarrow (\text{cph}_1, \text{cph}_2)$
$T \leftarrow T \| t$
Return o

Oracle corrupt(id):
Return st_{id}

Oracle trace( ):
Return T

Oracle merge(i, j):
$c \leftarrow c + 1$
$C \leftarrow \{\, \hat{c} \mid (\hat{c}, \text{snap}, \text{id}_i, \text{id}_j) \in \mathsf{L} \,\}$
If $C \neq \emptyset$:
  $\hat{c} = \min(C)$
  $\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{set}, \text{id}_i, \text{id}_j, \hat{c})\}$
  $\mathsf{L} \leftarrow \mathsf{L} \setminus \{(\hat{c}, \text{snap}, \text{id}_i, \text{id}_j)\}$
$p \leftarrow \text{head}\ p_{i,j}$
$(C_p, C_n, \text{id}, N, \text{pub}) \leftarrow \text{st}_i$
$(C'_p, C'_n) \leftarrow p$
For id $\in N$:
  If $(\text{snd}\, C'_p[\text{id}]) > (\text{snd}\, C_p[\text{id}])$:
    $C_p[\text{id}] \leftarrow C'_p[\text{id}]$
  If $(\text{snd}\, C'_n[\text{id}]) > (\text{snd}\, C_n[\text{id}])$:
    $C_n[\text{id}] \leftarrow C'_n[\text{id}]$
$\text{st}_i \leftarrow (C_p, C_n, \text{id}, N, \text{pub})$
$p_{i,j} \leftarrow \text{tail}\ p_{i,j}$

Game $G2_{\Pi,\mathcal{Z},\mathcal{A}}(n)$:
$\mathsf{L} \leftarrow \{\}$; $c \leftarrow 0$
$T \leftarrow \epsilon$; For $i, j \in [n] : p_{i,j} \leftarrow [\,]$
$(\text{prv}_q, \text{prv}_u, \text{pub}) \leftarrow\$\ \Pi.\text{setup}()$
For id $\in [n]$:
  $\text{cph} \leftarrow\$\ \Delta.\text{Enc}(\text{pub}, 0)$
  For $k \in [n]$:
    $C_p[k] \leftarrow (\text{cph}, 0)$
    $C_n[k] \leftarrow (\text{cph}, 0)$
  $\text{st}_{\text{id}} \leftarrow (C_p, C_n, \text{id}, n, \text{pub})$
$b \leftarrow\$\ \mathcal{Z}^{\mathcal{A},\text{write},\text{read}}(\text{pub}, n)$

Oracle write(id, op, v):
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{write}, \text{id}, \text{op}, v)\}$
$(C_p, C_n, \text{id}, N, \text{pub}) \leftarrow \text{st}_{\text{id}}$
$\text{cph} \leftarrow\$\ \Delta.\text{Enc}(\text{prv}_u, 0)$
If op = inc: $i \leftarrow p$
Else: $i \leftarrow n$
$\text{cph} \leftarrow \Delta.\text{Add}(\text{cph}, (\text{fst}\, C_i[\text{id}]))$
$t \leftarrow ((\text{snd}\, C_i[\text{id}]) + 1)$
$C_i[\text{id}] \leftarrow (\text{cph}, t)$
$\text{st}_{\text{id}} \leftarrow (C_p, C_n, \text{id}, N, \text{pub})$
$t \leftarrow (\text{cph}, \text{op})$
$T \leftarrow T \| t$
Return ϵ

Oracle prop(i, j):
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{snap}, \text{id}_i, \text{id}_j)\}$
$(C_p, C_n, \cdot, \cdot, \cdot) \leftarrow \text{st}_i$
$p \leftarrow (C_p, C_n)$
$p_{i,j} \leftarrow p_{i,j} \,||\, [p]$
Return p

Oracle read(id, op):
$c \leftarrow c + 1$
$\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{read}, \text{id}, \text{op})\}$
$o \leftarrow \text{correct}(\mathsf{L})$
$\text{sk} \leftarrow \text{prv}_q$
$(C_p, C_n, \cdot, N, \text{pub}) \leftarrow \text{st}_{\text{id}}$
$\text{cph}_1 \leftarrow\$\ \Delta.\text{Enc}(\text{pub}, 0)$
$\text{cph}_2 \leftarrow\$\ \Delta.\text{Enc}(\text{pub}, 0)$
For id $\in N$:
  $\text{cph}_1 \leftarrow \Delta.\text{Add}(\text{cph}_1, (\text{fst}\, C_p[\text{id}]))$
  $\text{cph}_2 \leftarrow \Delta.\text{Add}(\text{cph}_2, (\text{fst}\, C_n[\text{id}]))$
$t \leftarrow (\text{cph}_1, \text{cph}_2)$
$T \leftarrow T \| t$
Return o

Oracle corrupt(id):
Return st_{id}

Oracle trace( ):
Return T

Oracle merge(i, j):
$c \leftarrow c + 1$
$C \leftarrow \{\, \hat{c} \mid (\hat{c}, \text{snap}, \text{id}_i, \text{id}_j) \in \mathsf{L} \,\}$
If $C \neq \emptyset$:
  $\hat{c} = \min(C)$
  $\mathsf{L} \leftarrow \mathsf{L} \cup \{(c, \text{set}, \text{id}_i, \text{id}_j, \hat{c})\}$
  $\mathsf{L} \leftarrow \mathsf{L} \setminus \{(\hat{c}, \text{snap}, \text{id}_i, \text{id}_j)\}$
$p \leftarrow \text{head}\ p_{i,j}$
$(C_p, C_n, \text{id}, N, \text{pub}) \leftarrow \text{st}_i$
$(C'_p, C'_n) \leftarrow p$
For id $\in N$:
  If $(\text{snd}\, C'_p[\text{id}]) > (\text{snd}\, C_p[\text{id}])$:
    $C_p[\text{id}] \leftarrow C'_p[\text{id}]$
  If $(\text{snd}\, C'_n[\text{id}]) > (\text{snd}\, C_n[\text{id}])$:
    $C_n[\text{id}] \leftarrow C'_n[\text{id}]$
$\text{st}_i \leftarrow (C_p, C_n, \text{id}, N, \text{pub})$
$p_{i,j} \leftarrow \text{tail}\ p_{i,j}$

Figure 16: Second hop, indistinguishability of Δ

Oracle query $\langle \mathsf{prv}_q, \mathsf{op} \mid \mathsf{st} \rangle$:
$(M, \mathsf{sk}) \leftarrow \mathsf{prv}_q$
$(R, T_r, U, T_u, \mathsf{id}, N, \mathsf{pub}) \leftarrow \mathsf{st}$
If op = value:
  $\mathsf{cph}_1 \leftarrow\$\ \Omega.\mathsf{Enc}(\mathsf{pub}, 0)$
  $\mathsf{cph}_2 \leftarrow\$\ \Omega.\mathsf{Enc}(\mathsf{pub}, 0)$
  For $\mathsf{id}' \in N$:
    $\mathsf{cph}_1 \leftarrow \Omega.\mathsf{Add}(\mathsf{cph}_1, R[\mathsf{id}'][\mathsf{id}'])$
    $\mathsf{cph}_2 \leftarrow \Omega.\mathsf{Add}(\mathsf{cph}_2, U[\mathsf{id}'])$
Else:
  $\mathsf{cph}_1 \leftarrow R[\mathsf{id}][\mathsf{id}]$
  $\mathsf{cph}_2 \leftarrow U[\mathsf{id}])$
  For $\mathsf{id}' \in N$:
    $\mathsf{cph}_1 \leftarrow \Omega.\mathsf{Add}(\mathsf{cph}_1, R[\mathsf{id}'][\mathsf{id}])$
    $\mathsf{cph}_2 \leftarrow \Omega.\mathsf{Add}(\mathsf{cph}_2, R[\mathsf{id}][\mathsf{id}'])$
$r_1 \leftarrow \Omega.\mathsf{Dec}(\mathsf{sk}, \mathsf{cph}_1)$
$r_2 \leftarrow \Omega.\mathsf{Dec}(\mathsf{sk}, \mathsf{cph}_2)$
If op = value: $r = M + r_1 - r_2$
Else: $r = r_1 - r_2$
$t \leftarrow (\mathsf{op}, \mathsf{cph}_1, \mathsf{cph}_2)$
Return $\langle r \mid \cdot \rangle_t$

Oracle update $\langle \mathsf{prv}_u, \mathsf{op}, v \mid \mathsf{st} \rangle$:
$(M, \mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{prv}_u$
$(R, T_r, U, T_u, \mathsf{id}, N, \mathsf{pub}) \leftarrow \mathsf{st}$
valid = T
If op = tran: $(\mathsf{id}', v) \leftarrow v$
$c_1 \leftarrow R[\mathsf{id}][\mathsf{id}]$; $c_2 \leftarrow U[\mathsf{id}])$
For $\mathsf{id}' \in N$:
  $\mathsf{cph}_1 \leftarrow \Omega.\mathsf{Add}(c_1, R[\mathsf{id}'][\mathsf{id}])$
  $\mathsf{cph}_2 \leftarrow \Omega.\mathsf{Add}(c_2, R[\mathsf{id}][\mathsf{id}'])$
$r_1 \leftarrow \Omega.\mathsf{Dec}(\mathsf{sk}, \mathsf{cph}_1)$
$r_2 \leftarrow \Omega.\mathsf{Dec}(\mathsf{sk}, \mathsf{cph}_2)$
If op $\neq$ inc $\wedge$ (!inv$(r_1 - r_2 + v)$):
  $\mathsf{cph} \leftarrow\$\ \Omega.\mathsf{Enc}(\mathsf{pk}, 0)$
  valid = F
Else: $\mathsf{cph} \leftarrow\$\ \Omega.\mathsf{Enc}(\mathsf{pk}, v)$
If op = inc:
  $R[\mathsf{id}][\mathsf{id}] \leftarrow \Omega.\mathsf{Add}(\mathsf{cph}, R[\mathsf{id}][\mathsf{id}])$
If op = dec:
  $U[\mathsf{id}] \leftarrow \Omega.\mathsf{Add}(\mathsf{cph}, U[\mathsf{id}]$
If op = tran:
  $R[\mathsf{id}][\mathsf{id}'] \leftarrow \Omega.\mathsf{Add}(\mathsf{cph}, R[\mathsf{id}][\mathsf{id}'])$
$\mathsf{st} \leftarrow (R, T_r, U, T_u, \mathsf{id}, N, \mathsf{pub})$
$t \leftarrow (\mathsf{cph}_1, \mathsf{cph}_2, \mathsf{cph}, \mathsf{op})$
Return $\langle \mathsf{valid} \mid \mathsf{st} \rangle_t$

Oracle setup():
$(\mathsf{pk}, \mathsf{sk}) \leftarrow\$\ \Omega.\mathsf{Gen}()$
Return $((M, \mathsf{sk}), (M, \mathsf{pk}, \mathsf{sk}), \mathsf{pk})$

Oracle init(pub, id, N):
$\mathsf{cph} \leftarrow\$\ \Omega.\mathsf{Enc}(\mathsf{pub}, 0)$
For $k \in N$:
  For $j \in N$:
    $R[k][j] \leftarrow \mathsf{cph}$; $T_r[k][j] \leftarrow 0$
  $U[k] \leftarrow \mathsf{cph}$; $T_u[k] \leftarrow 0$
$\mathsf{st} \leftarrow (R, T_r, U, T_u, \mathsf{id}, N, \mathsf{pub})$

Oracle prop(st, id):
$(R, T_r, U, T_u, \cdot, \cdot, \cdot) \leftarrow \mathsf{st}$
Return (R, T_r, U, T_u)

Oracle merge(up, st):
$(R, T_r, U, T_u, \mathsf{id}, N, \mathsf{pub}) \leftarrow \mathsf{st}$
$(R', T'_r, U', T'_u) \leftarrow \mathsf{up}$
For $\mathsf{id}_1 \in N$:
  For $\mathsf{id}_2 \in N$:
    If $T'_r[\mathsf{id}_1][\mathsf{id}_2] > T_r[\mathsf{id}_1][\mathsf{id}_2]$:
      $R[\mathsf{id}_1][\mathsf{id}_2] \leftarrow R'[\mathsf{id}_1][\mathsf{id}_2]$
      $T_r[\mathsf{id}_1][\mathsf{id}_2] \leftarrow T'_r[\mathsf{id}_1][\mathsf{id}_2]$
  If $T'_u[\mathsf{id}_1] > T_u[\mathsf{id}_1]$:
    $U[\mathsf{id}_1] \leftarrow (U'[\mathsf{id}_1]$
    $T_u[\mathsf{id}_1] \leftarrow T'_u[\mathsf{id}_1]$
Return $(R, T_r, U, T_u, \mathsf{id}, N, \mathsf{pub})$

Figure 17: Bounded Counter from additively homomorphic scheme Ω.

Again, the differences between $\mathrm{G}_2^{\mathcal{Z}}$ and $\mathsf{Ideal}^{\mathcal{F},\mathcal{Z},\mathcal{S}}$ instantiated with this simulator are when traces and states are constructed. We can repeat the eager processing argument as before.

Let λ be the security parameter and $\mu(\lambda)$ a negligible function in it. To conclude, we have that

$$\begin{aligned}\mathsf{Adv}^{\mathsf{Sec}}_{\Pi,\mathcal{Z}} &= (|\Pr[\mathrm{G}_0^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}] - |\Pr[\mathrm{G}_1^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}])+ \\ &\quad (|\Pr[\mathrm{G}_1^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}] - \Pr[\mathrm{G}_2^{\mathcal{Z}}(n) \Rightarrow \mathsf{T}]|) \\ &\leq \mathsf{Adv}^{\mathsf{IND}}_{\Delta,\mathcal{B}}(\lambda) \\ &\leq \mu(\lambda)\end{aligned}$$

and Theorem 4.3 follows.

C BOUNDED COUNTER

The bounded counter construction is detailed in Figure 17. Its behavior is similar to the counter construction, however each replica maintains a matrix of counters. These track rights held, sent and received by all replicas (R).

Query operatins can refer to operations for value, or rights. The first is the result of adding all rights held by each replica, subtracted by all decrements. The second is the result of all rights held by a replica, adding all rights received and subtracting all rights sent and subtracted.

Update operations can be increments, decrements, or rights transferred. The first two either increment or decrement rights to a replica, depending on an invariant check. Transferring rights also requires invariant check, but inestad changes values in the matrix accordingly.