Secure Conflict-free Replicated Data Types

Manuel Barbosa
DCC-FC-UP & INESC TEC
mbb@dcc.fc.up

Bernardo Ferreira
DI-FC-UL & LASIGE
blferreira@ciencias.ulisboa.pt

João Marques
DI-FCT-UNL & NOVA LINCS
jb.marques@campus.fct.unl.pt

Bernardo Portela
DCC-FC-UP & NOVA LINCS
b.portela@fct.unl.pt

Nuno Preguiça
DI-FCT-UNL & NOVA LINCS
nmp@fct.unl.pt

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 new privacy-preserving CRDT protocols that can be used to help secure distributed cloud-backed applications, including NoSQL geo-replicated databases. Our proposals are based on standard CRDTs, such as sets and counters, augmented with cryptographic mechanisms that allow their operations to be performed on encrypted data. We accompany our proposals with formal security proofs and implement and integrate them in AntidoteDB, a geo-replicated NoSQL database that leverages CRDTs for its operations. Experimental evaluations based on the Danish Shared Medication Record dataset (FMK) exhibit the tradeoffs that our different proposals make and show that they are ready to be used in practical applications.

CCS CONCEPTS

• Computing methodologies → Distributed algorithms; **• Security and privacy → Cryptography**.

KEYWORDS

Distributed Systems, Cloud Computing, Security

ACM Reference Format:
Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela, and Nuno Preguiça. 2021. Secure Conflict-free Replicated Data Types. In *International Conference on Distributed Computing and Networking 2021 (ICDCN '21), January 5–8, 2021, Nara, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3427796.3427831

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICDCN '21, January 5–8, 2021, Nara, Japan
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8933-4/21/01…$15.00
https://doi.org/10.1145/3427796.3427831

1 INTRODUCTION

A Conflict-Free Replicated Data Type [25] (CRDT) is a recent abstraction for distributed cloud-backed protocols that allows 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 [17], Facebook [15], 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 [22], collaborative text edition [12], and chat systems for massive-multiplayer online video games [21].

Our work is the first to address the problem of creating privacy-preserving CRDTs, while allowing servers to continue performing the normal operation to maintain the CRDT state.

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 require, at most, access control mechanisms.

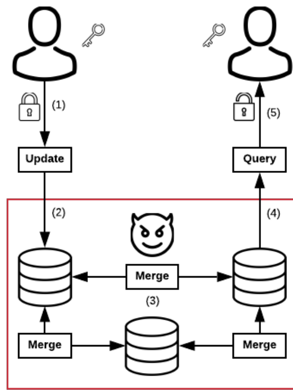

Figure 1: High-level view of our system.

CRDT Security. As far as we know, this is the first attempt at building secure CRDT protocols that can be leveraged by cloud-backed applications. Intuitively, privacy-preserving CRDT operations could be realised through fully homomorphic encryption or general secure multiparty computation. However, such solutions would either be unpractical or require sharing secret data between multiple nodes [9], which goes against the purpose of CRDTs in the first place.

Our Contributions. The contributions presented are as follows:

- Secure CRDT Protocols: We propose new secure CRDT protocols for Registers, Sets, Counters, and Bounded Counters, which are some of the most popular CRDTs used in distributed databases. Our proposals follow one of two approaches: (*i*) black-box constructions, which means that the protocols overlay encryption over standard CRDT implementations, allowing their use without modification (examples of this are the Register and Set CRDTs); and (*ii*) homomorphic constructions, which means that for CRDTs not covered by the previous approach, partially homomorphic encryption schemes can also be used to naturally transform them into secure ones with small overhead and leakage, and minimal alterations to their plaintext implementation.
- Integration with AntidoteDB: Using our new secure CRDTs, we implement a privacy-preserving version of AntidoteDB, which is a NoSQL key-value store that leverages CRDTs to perform its operations. By integrating our secure CRDT constructions with AntidoteDB, its clients can have security guarantees of the data stored in it.
- Experimental Evaluation: To experimentally access the efficiency and scalability of our proposals, we leverage a benchmark based on the Danish Shared Medication Record (FMK) and conduct experiments on our secure version of AntidoteDB, measuring the latency and throughput of operations over the different CRDTs proposed. Results obtained show that our proposals achieve practical performance and scalability, with different CRDTs making different trade-offs between efficiency and security.

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 encrypt sensitive data before sending to the untrusted network and have it be decrypted when it exits, while seamlessly propagating and merging encoded states.

Figure 1 captures this scenario. First, we have a setup phase 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 private 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 techniques.

From then onwards, every update operation is preceded by an encryption operation to ensure security (step 1). Given the encrypted 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 encrypted 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 encrypted data. The nodes can then be queried for the encrypted state (step 4). Finally, the obtained result must be decrypted to retrieve the query response (step 5).

Adversary Model. We assume adversaries will be honest-but-curious. This is a standard adversarial model for cloud computing and secure computation solutions [6], as it captures attack vectors where the service provider is not expected to deviate from its service level agreements but might still benefit from retrieving the stored data; 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 encrypted messages, or to have the system deviate in any other way besides delaying updates.

3 DEFINITIONS AND SYNTAX

Before describing our proposals in detail, we start by providing some definitions that will be common among all of our CRDTs.

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:

- setupC() is the global client setup procedure, which produces a set of cryptographic keys: prv_q for queries and prv_u for updates (in some implementations, they may be equal).
- setupS(pub, id) is the server node setup procedure, which on input the server node identifier id, outputs the initial state st for that node.
- query(prv_q, op; st) is an interactive protocol executed between a client node and a server node. On the client-side it takes as input key 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.
- update(prv_u, op, v; st) is an interactive protocol executed between a client node and a server node. On the client-side it takes as input key prv_u, an operation op and a value v. On the server side it takes as input 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 state 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 [8], 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, containing the messages exchanged between the participants. 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.

Figure 2 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.

Game $\mathrm{Real}_{\Pi,\mathcal{Z},\mathcal{A}}(n)$:
$T \leftarrow \epsilon$; For $i, j \in [n] : p_{i,j} \leftarrow [\,]$
$(\mathsf{prv}_q, \mathsf{prv}_u, \mathsf{pub}) \leftarrow\!\$\ \Pi.\mathsf{setupC}()$
For $\mathsf{id} \in [n] : \mathsf{st}_{\mathsf{id}} \leftarrow \Pi.\mathsf{setupS}(\mathsf{pub}, \mathsf{id}, n)$
$b \leftarrow\!\$\ \mathcal{Z}^{\mathcal{A},\mathsf{write},\mathsf{read}}(\mathsf{pub}, n)$

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

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

Oracle $\mathsf{write}(\mathsf{id}, \mathsf{op}, v)$:
$\langle o \mid \mathsf{st}_{\mathsf{id}} \rangle_t \leftarrow\!\$\ \Pi.\mathsf{update}\langle \mathsf{prv}_u, \mathsf{op}, v \mid \mathsf{st}_{\mathsf{id}} \rangle$
$T \leftarrow T \| t$
Return o

Oracle $\mathsf{read}(\mathsf{id}, \mathsf{op})$:
$\langle o \mid \cdot \rangle_t \leftarrow\!\$\ \Pi.\mathsf{query}\langle \mathsf{prv}_q, \mathsf{op} \mid \mathsf{st}_{\mathsf{id}} \rangle$
$T \leftarrow T \| t$
Return o

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

Oracle $\mathsf{trace}()$:
Return T

Game $\mathrm{Ideal}_{\mathcal{F},\mathcal{Z},\mathcal{S}}(n)$:
$\mathcal{F}.\mathsf{setupS}()$
$\mathsf{pub} \leftarrow\!\$\ \mathcal{S}.\mathsf{setup}(n)$
$b \leftarrow\!\$\ \mathcal{Z}^{\mathcal{S},\mathsf{write},\mathsf{read}}(\mathsf{pub}, n)$

Oracle $\mathsf{write}(\mathsf{id}, \mathsf{op}, v)$:
$o \leftarrow \mathcal{F}.\mathsf{write}(\mathsf{id}, \mathsf{op}, v)$
Return o

Oracle $\mathsf{read}(\mathsf{id}, \mathsf{op}))$:
$o \leftarrow \mathcal{F}.\mathsf{read}(\mathsf{id}, \mathsf{op})$
Return o

Figure 2: Real and Ideal security games. In the real world (left), $\mathcal{A}$ has access to oracles corrupt, trace, prop **and** merge**. In the ideal world (right), $\mathcal{S}$ has access to the adversarial interface of $\mathcal{F}$.**

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 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. $\mathcal{F}$ is a functionality that captures the idealized behavior of a CRDT, globally maintaining a log of all operations, and responding to requests accordingly. Messages exchanged in the network will instead be emulated by a simulator $\mathcal{S}$, with restricted access to $\mathcal{F}$. $\mathcal{S}$ 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. A more formal description of the ideal functionality and security model can be found in our companion technical report [3].

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 $\mathcal{A}$,

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

4 SECURE CRDT PROTOCOLS

We now present several CRDT proposals that can help build secure distributed databases. 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. The chosen CRDTs are core to the two main approaches for the development of CRDTs: CRDTs that manage data collections; and CRDTs that enable intrinsically commutative operations. In turn, these allow us to support all CRDTs made available in AntidoteDB.

Intuitively, the differences between proposals 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 more complex data structures, such as multi-value maps.

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. Concretely, when a client wants to update the register, it encrypts the value 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 on the register value, the CRDT runs seamlessly over encoded values.

Our protocol is described in Figure 3 and is as follows. We rely on a black-box standard register CRDT Π_{reg}, with operations for setupS, update, query, prop and merge, and on a standard encryption scheme Θ. The client generates and maintains symmetric key 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 if Π_{reg} is a correct CRDT, and Θ is a IND-CPA encryption scheme (meaning that it provides indistinguishability under chosen plaintext attacks), then the protocol in Figure 3 is a secure CRDT according to Definition 3.1 with baseline leakage, i.e., it only reveals how many operations were carried out and the size of inputs and outputs.

The proof is done in three hops. Hop 1 relies on the correctness of Π_{reg} to rely on an idealized structure for storing and presenting encrypted values; Hop 2 instead uses plaintext values in the idealized structure; and Hop 3 replaces all encrypted operations with dummy ciphertexts, which can be done given the IND-CPA property of Θ. A full proof of security can be found in our technical report [3].

```
query(key, op; st):
  Client:
  Send query() to Server
  Server:
  cph ←$ Π_reg.query(op; st)
  Send cph to Client
  Client:
  r ← Θ.Dec(key, cph)
  Return r

update(key, op, v; st):
  Client:
  cph ←$ Θ.Enc(key, v)
  Send update(cph) to Server
  Server:
  st ← Π_reg.update(op, cph; st)
  Return st

setupC():
  key ←$ Θ.Gen(1^λ)
  Return key

setupS(id, N):
  st ← Π_reg.setupS(id, N)
  Return st

prop(st, id):
  Return Π_reg.prop(st, id)

merge(up, st):
  Return Π_reg.merge(up, st)
```

Figure 3: Protocol of secure register from standard encryption scheme Θ and standard Register CRDT Π_{reg}.

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 computations 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 non-sensitive 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 set CRDTs merge values 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 encrypt set values upon update, decrypt values when queries are performed, and replace (if necessary) server-side comparisons with equivalent operations on encrypted 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 implementation, 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 [4]. This extension entails replacing instances of comparison within the CRDT with the respective secure operations.

Our protocol is described in Figure 4, and is very similar to the previous one. We rely on a black-box protocol of set CRDT

```
query(key, op; st):
If op = (cont, v):
   Client:
   cph ←$ Ω.Enc(key, v)
   Send query(cph) to Server
   Server:
   r ← Π_set.query(cont, cph; st)
   Send r to Client
Else:
   Client:
   Send query() to Server
   Server:
   cph ← Π_set.query(op; st)
   Send cph to Client
   r ← Ω.Dec(key, cph)
Return r

update(key, op, v; st):
Client:
cph ←$ Ω.Enc(key, v)
Send update(cph) to Server
Server:
st ← Π_set.update(op, cph; st)
Return st

setupC():
key ←$ Ω.Gen(1^λ)
Return key

setupS(id, N):
st ← Π_set.setupS(id, N)
Return st

prop(st, id):
Return Π_set.prop(st, id)

merge(up, st):
Return Π_set.merge(up, st)
```

Figure 4: Protocol of secure set from deterministic encryption scheme Ω and standard Set CRDT Π_{set}.

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 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 if Π_{set} is a correct CRDT, and Ω is a deterministic encryption scheme, then the protocol in Figure 3 is a secure CRDT according to Definition 3.1 with baseline leakage plus leakage of duplicate input values, allowing for the adversary to know when a repeated value is processed in the update and query operations.

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. A relevant nuance in the proof is that now replacing encryptions with dummy values must be done accounting for duplicates. A full proof can be found in our companion technical report [3].

Discussion. Similarly to the register, the cryptographic overhead imposed in terms of performance 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 added, and of the results of all cont queries. It is possible to reduce this leakage to standard indistinguishability security, however to achieve that 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 secure CRDT protocol, we propose a transformation that removes the need for comparing counter values upon merge. This consists in the inclusion of a per-replica Lamport clock, stored in plaintext since it is not sensitive data, to establish partial ordering of events[1]. This allows us to restrict the necessary computations on the encrypted counter to additions, which can be done over encrypted values if we instantiate our security layer with an additively homomorphic scheme.

Compared to the previous black-box proposals, this protocol 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 the baseline leakage 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. In the appendix section, 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 [20].

Concretely, our protocol is an adaptation of the state-based CRDT counter (Specification 7) in [24], generalised to allow for 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 protocol is detailed in Figure 5, where C_p and C_n respectively store the positive and negative increments observed in each replica, as well as the counter for each increment (e.g., $C_p[\mathsf{id}].\mathsf{cph}$ represents the encrypted positive increment of replica id, while $C_p[\mathsf{id}].\mathsf{ts}$ represents its counter).

[1] Considering our assumption that adversaries have full control over the network and scheduling of operations, this approach reveals no additional information.

```
query(sk; st):
Client:
Send query() to Server
Server:
(C_p, C_n, ·, N, pk) ← st
cph_1 ←$ Δ.Enc(pk, 0)
cph_2 ←$ Δ.Enc(pk, 0)
For id ∈ N:
  cph_1 ← Δ.Add(cph_1, (C_p[id].cph))
  cph_2 ← Δ.Add(cph_2, (C_n[id].cph))
Send (cph_1, cph_2) to Client
Client:
r_1 ← Δ.Dec(sk, cph_1)
r_2 ← Δ.Dec(sk, cph_2)
Return r_1 − r_2

update(sk, op, v; st):
Client:
cph ←$ Δ.Enc(sk, v)
Send update(cph, op) to Server
Server:
(C_p, C_n, id, N, pk) ← st
If op = inc: i ← p
Else: i ← n
C_i[id].cph ← Δ.Add(cph, C_i[id].cph)
C_i[id].ts ← C_i[id].ts + 1
st ← (C_p, C_n, id, N, pk)
Return st

setupC():
(pk, sk) ←$ Δ.Gen()
Return (sk, pk)

setupS(pk, id, N):
cph ←$ Δ.Enc(pk, 0)
For k ∈ N:
  C_p[k] ← (cph, 0)
  C_n[k] ← (cph, 0)
st ← (C_p, C_n, id, N, pk)
Return st

prop(st, id):
(C_p, C_n, ·, ·, ·) ← st
Return (C_p, C_n)

merge(up, st):
(C_p, C_n, id, N, pk) ← st
(C'_p, C'_n) ← up
For id ∈ N:
  If (C'_p[id].ts) > (C_p[id].ts):
    C_p[id] ← C'_p[id]
  If (C'_n[id].ts) > (C_n[id].ts):
    C_n[id] ← C'_n[id]
Return (C_p, C_n, id, N, pk)
```

Figure 5: Secure Counter CRDT from additively homomorphic scheme Δ.

SECURITY. We argue that, if Δ is an IND-CPA additively homomorphic encryption scheme, then the protocol described in Figure 5 is a secure CRDT according to Definition 3.1 with operation leakage, i.e., it reveals if the client is performing an increment, decrement, or query. A full proof can be found in our technical report [3].

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. However, this will require 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. [23] 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 [18], 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. For example, 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, and can perform:

- 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 (similarly to the previous protocol), 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 encrypted 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 counters 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. A full description of the protocol can be found in Figure 6.

SECURITY. We argue that if Δ is an IND-CPA additively homomorphic encryption scheme, our protocol is a secure bounded CRDT counter according to Definition 3.1 with additional leakage of the operation, and target replica for right transition. The security reasoning is similar to the one of the counter, but we need an additional hop to show that concurrent validations ensure that the invariant is never broken. A full proof can be found in our technical report [3].

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 occurred was an increment, decrement, or right transfer, and if the latter, to which replica the rights were transferred to.

5 INTEGRATION WITH ANTIDOTEDB

We implemented a prototype version of our secure CRDT protocols, 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 both an Erlang and Python clients

```
query(M, sk, op; st):
Client:
Send query(op) to Server
Server:
(R, T_r, U, T_u, id, N, pk) ← st
If op = value:
  cph_1 ←$ Ω.Enc(pk, 0)
  cph_2 ←$ Ω.Enc(pk, 0)
  For id' ∈ N:
    cph_1 ← Ω.Add(cph_1, R[id'][id'])
    cph_2 ← Ω.Add(cph_2, U[id']
Else:
  cph_1 ← R[id][id]
  cph_2 ← U[id])
  For id' ∈ N:
    cph_1 ← Ω.Add(cph_1, R[id'][id])
    cph_2 ← Ω.Add(cph_2, R[id][id'])
Send (cph_1, cph_2) to Client
Client:
r_1 ← Ω.Dec(sk, cph_1)
r_2 ← Ω.Dec(sk, cph_2)
If op = value: r = M + r_1 − r_2
Else: r = r_1 − r_2
Return r

setupC():
(pk, sk) ←$ Ω.Gen()
Return (M, sk, pk)

setupS(pk, id, N):
cph ←$ Ω.Enc(pk, 0)
For k ∈ N:
  For j ∈ N:
    R[k][j] ← cph; T_r[k][j] ← 0
  U[k] ← cph; T_u[k] ← 0
st ← (R, T_r, U, T_u, id, N, pk)

prop(st, id):
(R, T_r, U, T_u, ·, ·, ·) ← st
Return (R, T_r, U, T_u)

update(M, pk, sk, op, id', v; st):
Client:
valid = T
Send update() to Server
Server:
(R, T_r, U, T_u, id, N, pk) ← st
c_1 ← R[id][id]; c_2 ← U[id]
For id* ∈ N:
  cph_1 ← Ω.Add(c_1, R[id*][id])
  cph_2 ← Ω.Add(c_2, R[id][id*])
Send (cph_1, cph_2) to Client
Client:
r_1 ← Ω.Dec(sk, cph_1)
r_2 ← Ω.Dec(sk, cph_2)
If op ≠ inc ∧ (!inv(r_1 − r_2 + v)):
  cph ←$ Ω.Enc(pk, 0)
  valid = F
Else: cph ←$ Ω.Enc(pk, v)
Send (id', cph) to Server
Server:
If op = inc:
  R[id][id] ← Ω.Add(cph, R[id][id])
If op = dec:
  U[id] ← Ω.Add(cph, U[id])
If op = tran:
  R[id][id'] ← Ω.Add(cph, R[id][id'])
st ← (R, T_r, U, T_u, id, N, pk)
Return (valid; st)

merge(up, st):
(R, T_r, U, T_u, id, N, pk) ← st
(R', T'_r, U', T'_u) ← up
For id_1 ∈ N:
  For id_2 ∈ N:
    If T'_r[id_1][id_2] > T_r[id_1][id_2]:
      R[id_1][id_2] ← R'[id_1][id_2]
      T_r[id_1][id_2] ← T'_r[id_1][id_2]
  If T'_u[id_1] > T_u[id_1]:
    U[id_1] ← (U'[id_1]
    T_u[id_1] ← T'_u[id_1]
Return (R, T_r, U, T_u, id, N, pk)
```

Figure 6: Bounded Counter from additively homomorphic scheme Ω.

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 operations, and that no local concurrent operation that can 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.

6 EXPERIMENTAL EVALUATION

This experimental evaluation section aims to assess the performance and scalability overhead of our secure CRDT protocols, when compared to their non-secure versions. We performed two sets of experiments, micro-benchmarks that measure the latency and throughput of the operations of the different CRDTs designed, and macro-benchmarks that show how our secure version of AntidoteDB behaves with a realistic benchmark and in comparison with its original plaintext version.

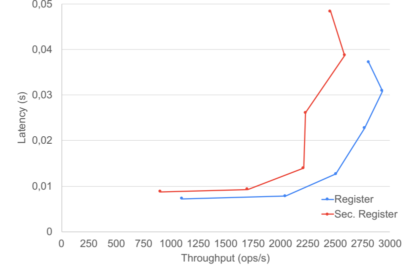

Figure 7: Throughput/latency for the plaintext and secure versions of the Register CRDT.

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.

6.1 Micro Experiments

To demonstrate the performance and scalability of our different secure CRDTs, we ran a micro-benchmark with our Python client, where clients execute operations in a closed loop for 2 minutes. For each CRDT, 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* × *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 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 version with a workload consisting of 50% reads and 50% writes. The results show that the two versions exhibit a similar behaviour, although the secure version has an overall higher latency and lower throughput. While the servers are not saturated, the latency of the plaintext and secure CRDT is similar, with around 7-8 milliseconds per operation respectively.

The servers start becoming saturated close to 2200 ops/s for the secure CRDT and 2500 ops/s for the plaintext version. This can be explained by the cryptographic expansion of the data in the secure version, 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, considering the secure version.

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 set 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%

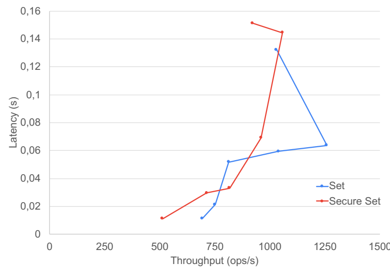

Figure 8: Throughput/latency for the plaintext and secure versions of the Set CRDT.

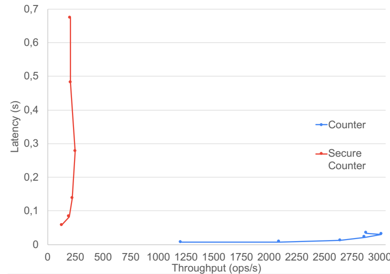

Figure 9: Throughput/latency for the plaintext and secure versions of the Counter CRDT.

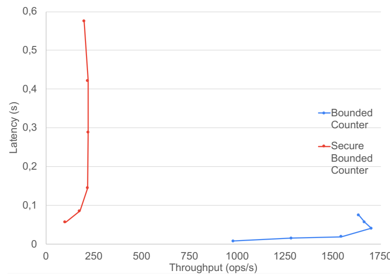

Figure 10: Throughput/latency for the plaintext and secure versions of the Bounded Counter CRDT.

increments, and 33% decrements. Naturally, the counter entails 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 richer in functionality than the previous examples. Results for the secure CRDT 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.

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.

6.2 Macro Experiments

To demonstrate the performance and scalability of our secure CRDTs when supporting real-world applications, we performed additional experiments with FMKe [27], a medical benchmark based on the Danish National Joint Medicine Card and specifically designed for NoSQL databases. FMKe populates AntidoteDB with over one million patients records, five thousand prescriptions, ten thousand medical staff, three hundred pharmacies and fifty healthcare facilities, being that the majority of the operations are carried over register and set CRDTs. FMKe includes different update and get operations over these records, including create-prescription, get-prescription-medication, among others. As such, this will rely on our CRDT implementations for secure register and secure set. To conduct these experiments we used our Erlang client, as FMKe was developed in Erlang, and the same experimental test-bench that was used in the micro-benchmarks, measuring latency and throughput with increasing number of clients. The benchmark was executed for 600 seconds, and unless stated otherwise, results presented next represent average values.

Throughput. Figure 11 shows how AntidoteDB throughput behaves as we increase the number of clients, comparing both our secure AntidoteDB and its original plaintext version.

On one hand, results obtained reveal that throughput increases steadily up to 32 clients in both secure and plaintext versions, decreasing as we add more clients. This suggests a breaking point at which the server saturates (around 32 clients), as adding more clients actually has a negative impact on the number of operations processed per second. On the other hand, results also show that the security overhead imposed by our CRDTs is minimal (around 350 ops/s) and that it increases/decreases in a similar trend as the plaintext version, suggesting that adding security mechanisms does not impose additional restrictions on system scalability.

Latency. To further understand the practicality of relying on secure CRDTs in the development of real-world systems, we now increase the granularity of our analysis to operation latency. Results are presented in Table 1.

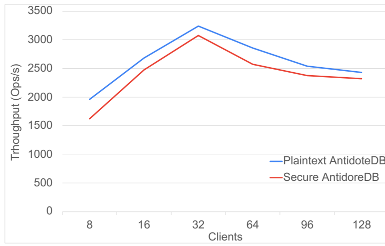

Figure 11: Throughput for the plaintext and secure versions of AntidoteDB when processing the FMKe benchmark.

All get operations exhibit similar behavior, consisting in simple retrieval of values from the database, with further value decryption in the secure version. We can observe that adding security requires an overhead of 1.62ms to 4.28ms over its plaintext counterpart, corresponding to an average latency increase of 24% to 51%. In contrast, Update Prescription and Update Prescription Medication are write operations and require the storage (and encryption in the secure version) of values in the database. This also shows a low impact of roughly 2.7ms when compared to the plaintext version, corresponding to an average latency increase of roughly 20%. Create prescription is the most computationally intensive operation of the group, as it requires multiple reads and writes from the dataset to construct the prescription data structure, which in its secure instantiation also require multiple calls to the cryptographic library. Here we can observe an average latency increase of roughly 12.7ms, corresponding to a 46% increase. A thorough description of each operation and associated pseudo-code can be found in [27].

Table 1 also presents values corresponding to percentiles 95 and 99. The increased latency (in both versions) in comparison to average values is due to the amount of data accessed by operations - for instance, the amount of data returned by the Get Patient operation depends on how many prescriptions the patient has. This means that the values shown are not outliers but a natural consequence of how FMKe data is distributed. Moreover, the differences between plaintext and secure versions of the system are reduced when considering these percentiles. This is observable in all operations, with get operations being reduced to 9%-30% increases in P95 and 6%-9% in P99; write operations to roughly 7% in P95 and 6% in P99, and create prescription having an increase of roughly 11% in both P95 and P99. These results demonstrate that security mechanisms do not provoke additional outliers and thus are not expected to produce unexpected system delays.

Stability. Another important issue regarding the overhead of security mechanisms is related to system stability. Specifically, we are interested in understanding the performance of our security mechanisms as our system develops. Figure 12 presents the average latency variation for Create Prescription, Get Patient and Update Prescription operations. Similar values were observed in other operations for respective get and update processes, which we omit for succinctness. In both plain and secure system modes, latency variation is relatively small. We can also highlight that neither version has any noticeable deterioration as the benchmark progresses, suggesting that using cryptography to enhance security is not imposing an increasing overhead on the system. This close margin between plaintext and secure version of operations

		Average	P95	P99
Get Patient	Plain	8.22	61.10	67.00
	Secure	10.25	66.81	70.80
Get Pharmacy Presc.	Plain	8.21	61.63	67.03
	Secure	12.29	69.22	73.37
Get Prescription	Plain	6.94	48.17	65.61
	Secure	8.72	63.84	69.72
Get Presc. Medication	Plain	7.01	59.24	66.03
	Secure	8.63	65.35	69.99
Get Processed Presc.	Plain	8.21	61.50	67.00
	Secure	12.44	69.30	73.21
Get Staff Presc.	Plain	8.16	61.33	66.98
	Secure	10.49	67.16	71.32
Update Prescription	Plain	13.65	67.45	71.98
	Secure	16.39	72.50	76.47
Update Presc. Med.	Plain	13.46	67.23	71.98
	Secure	16.30	72.59	76.51
Create Prescription	Plain	27.43	75.75	79.55
	Secure	40.14	84.20	89.81

Table 1: Operation latency, considering average and percentiles 95 and 99. Values are in milliseconds.

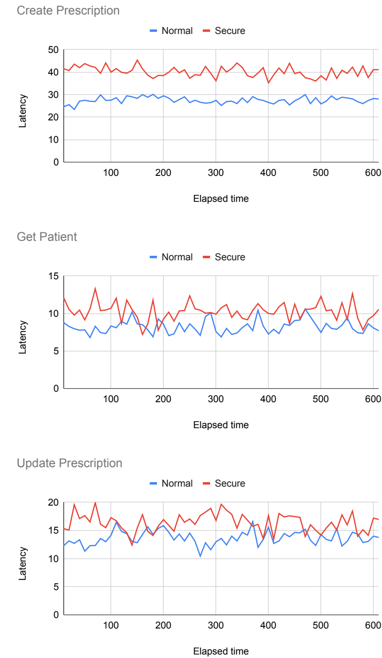

Figure 12: Average latency variation for the FMKe benchmark. Elapsed time is in seconds, Latency is in milliseconds.

is clearly observable in Get Patient, with baseline small latency values, and to a lesser extent in Update Prescription, where the variation in measurements sometimes presents the processing of requests in the secure system as having lower latency than those of the plaintext system.

Discussion. Our macro experiments present the feasibility of our design for developing privacy-critical applications relying on secure CRDTs. The latency overhead can be directly mapped to the cost of the underlying cryptographic operations, as exemplified in the micro experiments, which is a standard overhead for enhancing

real-world systems with confidentiality guarantees. Furthermore, we have shown that, when applied to a benchmark emulating a real-world application, the cryptographic overhead on throughput and latency is similar to the cost of enforcing standard security guarantees in key-value databases: encrypting and decrypting values when writing and reading from the database, respectively. These experiments also suggest that security measures impose no additional impact on the system, as the benchmark maintained consistent performance values throughout its lifecycle in both settings.

7 RELATED WORK

CRDTs were originally designed for decentralized distributed systems without any security concerns [25]. 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. [26], in a *work in progress* report, presents an approach that complements our work, focused in detecting and tolerating 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 [16], in the sense that clients are collaboratively performing a computation over a shared database state. Cachin et al. [7] 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 [5] uses Byzantine fault tolerance to ensure that replicas converge, and traditional encryption to preserve data privacy, however it does not support data computation. CryptDB [19] 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.

8 CONCLUSION

To the best of our knowledge, this work presents the first set of cryptographically validated protocols for secure CRDT solutions. Our results show that one can instrument correct CRDT protocols with a layer of security mechanisms to achieve secure CRDT solutions, provided that this encoding does not break the functional part of the CRDT necessary for operations such as state merging.

Experimental validation suggests the performance overhead for secure CRDT solutions is proportional to the underlying security mechanism. When considering a benchmark for real-world CRDT applications we can observe that the security cost is relatively low, with an average latency increase of 37.5% and 20% for get and put operations, and a throughput reduction of 5%. These results suggest the feasibility of considering secure CRDT solutions for applications managing sensitive data.

ACKNOWLEDGMENTS

This work was supported by FCT/MCTES through project HADES (PTDC/CCI-INF/31698/2017) and the NOVA LINCS (UIDB/04516/2020) and LASIGE Research Units (UIDB/00408/2020 & UIDP/00408/2020).

REFERENCES

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

[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 *SRDS'15*. IEEE, 31–36.

[3] Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela, and Nuno Preguiça. 2020. Secure Conflict-free Replicated Data Types. Cryptology ePrint Archive, Report 2020/944. https://eprint.iacr.org/2020/944.

[4] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. 2007. Deterministic and efficiently searchable encryption. In *Crypto'07*. IACR, 535–552.

[5] 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.

[6] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS'08*. Springer, 192–206.

[7] Christian Cachin, Esha Ghosh, Dimitrios Papadopoulos, and Björn Tackmann. 2018. Stateful multi-client verifiable computation. In *ACNS'18*. Springer, 637–656.

[8] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS'01*. IEEE, 136–145.

[9] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *Crypto'12*. Springer.

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

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

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

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

[14] Stephan Alexander Kollmann, Martin Kleppmann, and Alastair Beresford. 2019. Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing. *PETS'19* 2019, 3 (2019), 1–23.

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

[16] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. 2004. Fairplay-Secure Two-Party Computation System.. In *Security'04*, Vol. 4. USENIX, 9.

[17] Rimma Nehme. 2018. Azure #CosmosDB @ Build 2018: The catalyst for next generation apps. https://tinyurl.com/yxlqjm2m.

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

[19] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP'11*. ACM, 85–100.

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

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

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

[23] 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).

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

[25] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *SSS'11*. Springer, 386–400.

[26] Ali Shoker, Houssam Yactine, and Carlos Baquero. 2017. As secure as possible eventual consistency: Work in progress. In *PPCDD'17*. ACM, 5.

[27] Gonçalo Tomás, Peter Zeller, Valter Balegas, Deepthi Akkoorath, Annette Bieniusa, João Leitão, and Nuno Preguiça. 2017. FMKe: A Real-World Benchmark for Key-Value Data Stores. In *PaPoC'17*.