Secure Coordination-free Intermediaries for Local-first Software

Anonymous author
Anonymous affiliation

Abstract

This paper is about programming support for applications that synchronize their data between multiple devices – to allow a single user to use an application on multiple devices or to support collaboration between multiple users. Examples are (shared) calendars, document editors, task lists, finance management, collaborative workflows, and more. Such applications must not impede or interrupt the user's normal workflow – even when the device is offline or has a flaky network connection – and preserve the privacy and integrity of the user's data.

From the programming perspective, synchronization and availability along with privacy and security concerns add significant challenges. This work aims to relieve developers from this complexity so that they can focus on the business logic of the application. To this end, we design an easy-to-use CRDT-based programming model with a built-in encryption and authentication scheme that allows coordination-free synchronization of data using untrusted intermediaries without sacrificing data privacy and integrity. We show that our approach is suitable to encrypt state-based and delta-state-based CRDTs transparently while retaining coordination freedom. Our evaluation highlights that the proposed solution is practical in terms of runtime and memory overhead.

2012 ACM Subject Classification Information systems → Data management systems; Computer systems organization → Dependable and fault-tolerant systems and networks; Security and privacy → Cryptography

Keywords and phrases local first, data privacy, coordination freedom, Scala 3, CRDTs, AEAD

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.0

1 Introduction

This paper is about programming support for applications that synchronize their data between multiple devices – to allow a single user to use an application on multiple devices or to support collaboration between multiple users. Examples are (shared) calendars, document editors, task lists, finance management, collaborative workflows, and more. Especially the collaborative case has exploded in importance due to the COVID-19 pandemic.

Such applications must not impede or interrupt the user's normal workflow – even when the device is offline or has a flaky network connection – and they must preserve the privacy and integrity of the user's data. No one likes to be told: "please connect to the internet to keep working so that we can observe everything you do." From the programming perspective, the availability requirement along with privacy and integrity concerns add significant challenges. This work aims to relieve developers from this complexity so that they can focus on the business logic of the application. To this end, we design an easy-to-use CRDT-based programming model with a built-in encryption and authentication scheme that allows coordination-free synchronization of data using untrusted intermediaries without sacrificing privacy and data confidentiality.


© Anonymous author(s);
licensed under Creative Commons License CC-BY 4.0

1.1 Existing Approaches for Collaborative Data Management

Consider two examples for common approaches to enable collaborating on shared data. In Git[1], all operations happen on a user's device, and changes are synchronized by explicit user requests (i.e., pull/push a Git repository) to handle conflicts that require the user's attention. The approach taken by Google Docs[2] provides synchronization automatically, but Google's central servers need to know about the data to achieve this. Centralized coordination is undesirable for two general reasons. First, a central instance may not be available – e.g., when devices are in the same LAN or DTN [6], but not connected to the internet. Second, having data be known to Google raises privacy concerns for users and companies that want to keep their secrets.

The need to address the above dilemma between transparent synchronization on the one side and loss of privacy and centralized control on the other side led to the proposal of *local-first software*, "a set of principles for software that enables both collaboration and ownership for users" [24]. With local-first software, data ownership remains on local devices. In addition, there is the expectation that data is synchronized and kept consistent between multiple replicas without requiring coordination or manual user intervention. Practically, these goals of local-first software are often achieved by using *conflict-free replicated data types* (CRDTs) [47] to store and manage the data of each device.

A CRDT stores user's data together with sufficient metadata to enable consistent and coordination-free replication in arbitrary network topologies in a transparent way. To enable coordination-free replication, CRDTs (and any other coordination-free distributed scheme) require that the stored data is processed via "monotonic operations" [21]. In practice, this restriction is often circumvented by appropriate designs of data types. In particular, for end-user applications that we are targeting, the restriction is not an issue, given that intuitively, users always interact with their device in a monotonic fashion, because they only can press, click, and touch keys and buttons and not "unpress" a prior action. This intuition is leveraged by recent reactive programming approaches that automatically map user interactions to synchronous updates of related CRDTs to enable coordination-free consistency of entire applications [31, 32, 33].

CRDTs are not a silver bullet to replace client-server with peer-to-peer architectures because central servers can help with connectivity issues in common scenarios. Consider Figure 1 for illustration. Alice (top of Figure 1) uses a local-first calendar application with direct peer-to-peer connections on her office computer. While the application is capable of synchronizing her calendar with anyone she connects to, her options are limited. She cannot synchronize with her laptop at home because it is always off when she is at work, and when she is at home, the computer in the office is off. Also, she can synchronize with neither Bob nor Charlie. In the former case, because her office is behind a firewall that restricts incoming connections and Bob has not configured his NAT to forward the correct ports – thus a connection attempt fails in either direction. In the latter case, because Charlie uses the application in a Web browser, which does not allow incoming connections at all.

1.2 Our Proposal

Connectivity is improved by using intermediaries – third parties which serve as post offices that store, forward, eventually discard old messages, and otherwise do not modify them.

[1] `https://git-scm.com/`
[2] `https://about.google/intl/docs/`

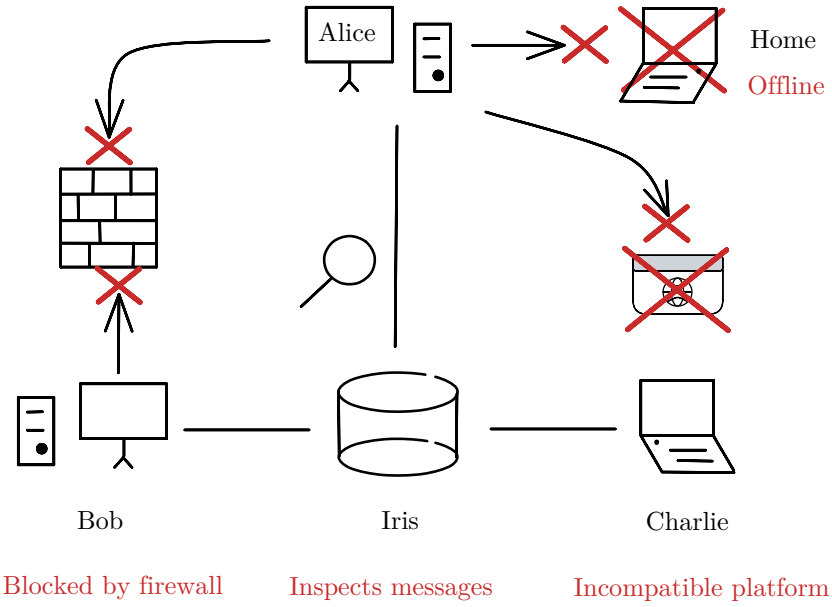

Figure 1 Example illustrating issues with peer-to-peer and intermediary-based synchronization.

Crucially, adding intermediaries combines the advantages of local-first software with those of a centralized data store, without relying on coordination by a central entity to ensure consistency. Widely known examples of intermediaries are code-hosting platforms such as GitHub[3]. In Figure 1, Iris represents an intermediary between Alice, Charlie, and Bob.

Because intermediaries never produce changes by themselves and do not have to participate in any conflict resolution of their own, it is possible to use many existing solutions as intermediaries. GitHub could readily be replaced with another code hoster supporting Git or even by direct connections between users if available. Examples of possible intermediaries in our setting include storing the replicated data into Dropbox, keeping it on a USB drive, on a shared network disk, or using a delay tolerant network (DTN). The last example is of particular interest, because DTN is actively developed to make infrastructure more resilient [48, 44] and has a wider range of applicable areas from web browsers [6] to low energy wireless communication [7].

However, with intermediaries the privacy concerns of centralized data stores still persist. Iris from Figure 1 can inspect and modify the data. Whenever Alice puts a vacation into her calendar, Iris sends her marketing material for trips, flights, and hotels. Iris also puts convenient reminders for Alice's birthday into Bob's and Charlie's calendar – directly with links to Iris' own boutique with ideas for presents. In general, without further measures, intermediaries violate the data privacy principles of local-first software, in particular, that there should be "data confidentiality and privacy by default" and that the user should "retain ultimate ownership and control" [24].

To establish "data confidentiality and privacy by default" in the presence of untrusted intermediaries, we introduce a programming model that integrates a new kind of CRDTs, called *encrypting CRDTs (or enCRDTs for short)*. An enCRDT adds an abstraction layer

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

around a normal CRDT to protect the data stored in the inner CRDT from inspection or modification while still providing coordination-free synchronization of data. To this end, we introduce an *authenticated encryption with associated data* (AEAD) scheme, which we argue is secure for coordination-free encryption of data at rest. A major issue of AEAD components that we must address is the reuse of parameters accross devices that must be globally unique. Ensuring uniqueness is a classical coordination problem – we want to minimize the need for synchronization. We discuss how to address this issue with several strategies that provide different tradeoffs. Our scheme does not require coordination beyond exchanging a secret key once and thus is suitable for decentralized use.

With enCRDTs in place, data is automatically encrypted and authenticated before messages are disseminated, thus intermediaries can forward and replicate data without privacy considerations. While primarily motivated by the need to protect stored data from malicious intermediaries, our solution also protects the data during transport, e.g., for direct communication between Alice and Bob. While securing direct communication channels may seem like a solved problem – i.e., by using TLS – it has been shown that leaving this task to application developers often leads to an insecure system [15, 36].

High-level abstractions with built-in cryptographic features are considered as an effective solution to support developers with writing secure software [1, 18, 30]. The correct direct usage of cryptographic components is challenging in general [34], with 84 % of Apache projects containing cryptographic misuses [39]. Especially developers of end-user applications seem to have a hard time, with more than 95 % of android applications that use a cryptographic API using it incorrectly [25]. Moreover, local-first software is expected to work on many different devices, increasing chances for implementation bugs, and teams developing such applications are often too small to hire security experts. Our approach addresses the challenges by automatically providing suitable guarantees.

In summary, the **contributions of this paper** are:

- A programming model for systematically using CRDTs for local-first software, which clearly separates the core state of a CRDT, which represents the user's data and the metadata for synchronization, from both its programming interface and the communication layer (Section 2). The separation allows our approach to be easily embedded into existing programming models, such as reactive programming [33], the actor model, object-oriented programming, functional programming, or even more special-purpose models. The programming model builds the foundation for the rest of our contributions.
- A family of enCRDT implementations for different network requirements (Section 3). The enCRDTs reuse the same implementation strategies as other CRDTs and provide secure communication as their set of operations. In particular, this allows us to separate reasoning about security concerns, like data confidentiality and authenticity, from reasoning about ordination-free consistent synchronization of data.
- A secure scheme to encrypt messages inside enCRDTs suitable for our target scenario of local-first software, which precludes the misuse of any of the cryptographic components by developers (Section 4). In the same manner that the data model of CRDTs enables automatic coordination-free synchronization, our enCRDTs enable correct automatic choice of cryptographic primitives and parameters.

An evaluation of a full implementation of our solution (Section 5) shows that the encryption overhead is small – with less than 3 ms in the worst case of encrypting large states without hardware-accelerated encryption, and only fractions of milliseconds when hardware acceleration is available.

```
// state definition
type Counter = HashMap[ReplicaID, Int]

// state interpreted as data in the application
object Counter:
  def zero: Counter = HashMap.empty

extension (c: Counter)
  def value: Int = c.values.sum
  def inc(id: ReplicaID): Counter =
    HashMap(id -> (c.getOrElse(id, 0) + 1))

// state is a lattice for replication
given Lattice[Counter] with
  def merge(left: Counter, right: Counter): Counter =
    left.merged(right){
      case ((id, v1), (_, v2)) => (id, (v1 max v2))
    }
```

Figure 2 Replicated counter split into the state, interpretation as data, and lattice definnition.

2 Systematic and Modular Design of Custom CRDTs

Conflict-free replicated data types (CRDTs) [47] are a special class of abstract data types that enable replication of data across multiple devices with strong eventual consistency guarantees. Many data types can be and have been implemented as CRDTs [46] ranging all the way to full JSON structures [23] and efficient representations for text [16]. While existing work has presented a range of off-the shelf CRDTs implementations, a systematic approach to support developers in designing new application-specific CRDTs is still missing. The model presented in this section aims to fill this gap.

The proposed model enables a systematic design of custom CRDTs without requiring developers to have expertise in managing distributed systems or data confidentiality and authenticity. It combines strategies from state-based CRDTs [47] and delta state replication [3] to provide coordination-free replication (using a strong eventual consistent implementation [47]), while minimizing the burden on developers. The only expectation is that developers provide a correct merge function, whereby for structured data types merge functions are automatically derived (see Subsection 2.2). In particular, we use the merge function to automatically ensure monotonicity of operators, instead of leaving the responsibility to correctly implement those to developers.

2.1 State, Operators, and API

In our model, the state of a CRDT is always an immutable, but otherwise arbitrary value. Its design is driven by the desired operations that must be supported, and the ability to define a correct merge function for the state.

Each CRDT state has associated query operators and update operators (also called mutators). Query operators are functions over the state that return some query-specific value. Mutators transform one state into another state. Note, that state-based CRDT implementations usually require that mutators return a state that is larger than the original state

```
19 class CounterClass(replicaID: ReplicaID):
20   private var current = Counter.zero
21
22   def inc(): Unit = current = current merge current.inc(replicaID)
23   def value: Int = current.value
```

Figure 3 Embedding of our CRDT design into an object oriented API.

according to a fixed order of all possible states. However, this requirement can be automatically satisfied by merging the current state with the result of a mutator to change a CRDT. Thus, using this approach developers cannot design mutators that produce inconsistencies.

As a running example, consider the Scala 3 implementation of a replicated counter in Figure 2. Line 2 shows the definition of the state, where a replicated counter is represented by the built in HashMap associating replica IDs as keys with integers as values. Lines 5-11 define operators that interpret this state as a counter data type, which has a value and can be increased. A counter of zero is the empty map (Line 6), the value of a counter is the sum of the values in the map (Line 9), and a counter is incremented by incrementing the value associated to the current replica ID. Note, that the incrementing operator returns a counter state that only contains the value of the incremented replica, not any other replicas. Finally, as the foundation of all ensured guarantees, Line 14 states that a counter is a lattice. We discuss lattices in detail in Subsection 2.2, but practically a counter being a lattice requires a merge function in Line 15, which must be associative, commutative, and idempotent. Most notably, a correct merge function is the only requirement for consistency, while incorrect operations only manifest as local application bugs.

Operators define the data that a CRDT represents. Different state implementations may define the same conceptual data, and the same state may represent different data depending on the operators. For example, the counter state in Figure 2 can also be used as a version vector by providing the suitable comparison operators.

Our design is non-classical in the sense that mutators do not actually change any state, because up until now we have only given the building blocks out of which an abstraction that is classically called a CRDT is constructed. The advantage of this "naked" encoding is that it can be easily integrated into most programming models, thus making our extensions for intermediaries and data confidentiality and authencitiy available to developers no matter which programming style they prefer. For example Figure 3 shows an embedding into an object-oriented API, where the current state is stored in a private field, which is modified by the increment method and accessed by the value method. We believe that readers familiar with the actor model will be able to adapt the example on the fly, and Mogk et al. [33] have shown how to embed CRDTs into functional reactive programming.

2.2 Lattice-based Coordination-free Consistency

The best achievable form of message delivery in a distributed system is at-least-once delivery. Consistency without coordination is achieved in such a setting by using a merge function m which must be associative, commutative, and idempotent to deal with the shortcomings of at-least-once message delivery. Concretely, the merge function must be associative $m(m(x, y), z) = m(x, m(y, z))$ to ensure that states can be combined before transmission, commutative $m(x, y) = m(y, x)$ because messages may not arrive in order, and idempotent: $m(x, x) = x$ to deal with duplicated transmissions. Mathematically speaking such a merge

```
24 given[A: Lattice, B: Lattice]: Lattice[(A, B)] with
25   def merge(left: (A, B), right: (A, B)): (A, B) =
26     (left._1 merge right._1, left._2 merge right._2)
27
28 type PNCounter = (Counter, Counter)
29
30 extension (c: PosNegCounter)
31   def value: Int = c._1.value - c._2.value
```

Figure 4 Composing merge functions to implement a positive-negative counter

function computes the *least upper bound* $\sqcup$ of two states: $s = s_1 \sqcup s_2$. Thus the state forms a *join semilattice*, or simply a lattice, which explains the naming in Figure 2.

We encourage readers to convince themselves that the merge function in Figure 2 has the three properties. The lattice it operates on are two mappings $x, y : id \mapsto \mathbb{N}$ which map replica IDs to natural numbers (with a default of 0) and merges them such that $m(x, y)(id) = \max(x(id), y(id))$, where max computes the maximum of two natural numbers.

A major advantage of the approach to represent CRDTs as immutable states with associated merge functions is composability. As an example, consider that our counter can only be incremented, not decremented. A common strategy to address this using CRDTs – where we can only increase the state – is to expand the state to represent decrements as an addition in another dimension. Figure 4 defines the state of a counter that can be incremented and decremented as a pair of two normal counters (Line 28). One counter for storing increments, and another for decrements. This construction is commonly referred to as a positive-negative counter (PNCounter). The value of this PNCounter, as an example for an operation, is the value of the first counter minus the value of the second (Line 31).

Crucially, this PNCounter is automatically a lattice, without the developer needing to write a custom merge function. The reason is that a counter is a lattice, and a pair is also a lattice given its components are lattices. To illustrate the latter, the merge function for the pair lattice is given in Line 24, which syntactically states that given A and B are lattices, the pair (A, B) is also a lattice. The remaining construction of the composed merge function for the PNCounter is then so schematic, that the Scala compiler can do it on its own.

Moreover, consider the implementation of the tuple merge function (Line 26), it simply deconstructs the pair into its components, merges the components individually, and recombines the result into a pair. The same strategy can be used for all non-recursive data types built out of components, including maps (the counter in Figure 2 is a special case of the map lattice, using the max function on integers to merge the components of the map), sets, tuples of arbitrary sizes, and any non-recursive user-defined case class. For example, if we want build a new a social media site – that has posts with likes, dislikes, and comments – we could define our data type like below (given that LWW is a last-writer-wins lattice), and immediately have a coordination-free consistently distributed private space for us and our friends. The required merge function is automatically (and correctly) constructed by the compiler (composing existing merge functions and generating one for the case class), and used by the rest of our approach.

```
252 case class SocialPost(message: LWW[String], likes: PNCounter,
253     dislikes: PNCounter, comments: Set[LWW[String]]) derive Lattice
254 type SocialMedia = Set[SocialPost]
```

While composability of confluent functions (the class of functions that merge belongs to) is well known [21], it has only been used to construct lattices in special-purpose languages where every construct is confluent. We are not aware of prior work that embeds the use of automatic derivation of merge functions (by automatically composing them) into a general-purpose language. We achieve this by cleanly separating state definition, operators, and merge functions to benefit from the inherent composability of each of the parts and let developers recombine them as needed. In contrast, the object-oriented API from Figure 3, e.g., is not suitable for automatic composition.

2.3 Anti-Entropy and Delta Replication

While the merge function of a lattice addresses the challenges of at-least-once delivery, our approach also enables efficient dissemination of messages. Similar to prior work [3], we rely on an explicit anti-entropy algorithm to ensure that every update in one replica reaches every other replica. As an example for an anti-entropy algorithm consider a simple broadcast mechanism and assume that all devices are connected to each other. Periodically each replica takes its current full state of a CRDT, serializes the state to a message, and sends the message to all other replicas. When such a message is received, it is deserialized, and the resulting state is merged into the local state.

The choice of anti-entropy algorithm determines whether the system is causally consistent, i.e., whether replicas see changes in the order they happened on a remote replica. The broadcast algorithm mentioned above is causally consistent, because it always sends the full state – which includes all changes – thus there can be no gaps in the causal history. However, always transmitting the entire state is a waste of network resources. Consider the counter example, where, every time we increment the counter, only one entry is changed, but the complete state will be transmitted. To improve efficiency, we use *delta replication* [3]. Instead of transmitting the full states, we only transmit a delta state, which has redundancies to already transmitted states removed. The delta between states s_1 and a larger state s_2 is the smallest state s_δ such that $merge(s_1, s_\delta) = s_2$. Where small and large are according to the lattice order. As we briefly mentioned in Section 2, the mutator of our counter in Figure 2 already produces delta states in the form of the singleton map from the replica ID to its new value. Specifically, deltas are normal states as far as the lattice is concerned, and can be merged into larger groups.

An anti-entropy algorithm has to merge deltas in causal order to preserve causal consistency. This can be done by relying on an ordered network protocol (e.g., TCP), if available, and only two devices are involved, or, more generally, by including causality information together with the transferred deltas. We will revisit reasoning about causality of deltas in the next section, where we use it for efficient dissemination of encrypted messages. Depending on the use case, causal consistency might not be required, thus enabling a more efficient anti-entropy algorithm. However, we recommend using causal consistency as a baseline for local-first software, because it is the least confusing form of weak-consistency for end-users (e.g., imagine you receive the answer to an email before seeing the original message).

3 Encrypting CRDTs

Our goal is to transparently protect user data in a setting where untrusted intermediaries are used to facilitate communication. There are two elements of our solution for this problem: (a) a cryptographic system that provides authenticated encryption and decryption capabilities, and (b) a scheme to efficiently replicate encrypted data. While these two are designed

```
34 type EnCRDT[S] = Set[AEAD[S]]
35
36 given [S]: Lattice[EnCRDT[S]] with
37   def merge(left: EnCRDT[S], right: EnCRDT[S]): EnCRDT[S] =
38     left union right
39
40 extension [S] (c: EnCRDT[S])
41   def send(data: S, key: Secret, replicaID: ReplicaID): EnCRDT[S] =
42     Set(encrypt(data, AssociatedData.empty, key))
43
44   def recombine(key: Secret)(using Lattice[S]): Option[S] =
45     c.flatMap(decrypt(key)).reduceOption(Lattice.merge[S])
```

Figure 5 Naive enCRDT that stores all states.

together, we will first present how the cryptographic system is used for efficient replication of encrypted data, assuming that it is safe for this use-case, and then present the cryptographic system itself in Section 4.

We provide four transparently encrypting CRDTs (enCRDTs) that implement variants of our encryption scheme. Each of the four enCRDTs provides operators that constitute the interface of an anti-entropy algorithm – sending updated states and recombining received states into a full CRDT. They differ in how they prevent duplicated encrypted data from overloading intermediaries. An enCRDT stores serialized state, which we refer to as messages, and the causality information that is required for pruning and efficient dissemination of messages (as we elaborate below). An enCRDT is used as a layer between a normal CRDT (that uses the enCRDTs to send and receive updates) and an existing anti-entropy algorithm (that treats the enCRDT as any other CRDT).

With an enCRDTs in place, the intermediaries become replicas that run the usual anti-entropy algorithm – synchronizing the state of the enCRDT. However, they do not know the required secret to use operators for inspecting or modifying enCRDTs; they are *untrusted replicas*. For disambiguation, we call replicas that do know the secret *trusted replicas*. While messages of enCRDTs are confidential, i.e., only trusted replicas should be able to access them, causality information is not. Since only trusted replicas should be able to change both messages and causality information, the authenticity of both is required.

In the following, we elaborate on the four variants of our encryption scheme.

3.1 Naive Approach

The trivial solution is an enCRDT that simply stores all messages (i.e., every serialized version of every state they receive). An implementation is shown in Figure 5. Line 34 defines the state of the naive enCRDT as a set of *authenticated encryption with associated data* (AEAD) values. AEAD allows intermediaries to merge their state based on the associated data (Line 38). However, the naive enCRDT scheme does not make use of this popssibility and simply unions the sets of AEAD values when merging two enCRDTs. The send operator (Line 38) inserts new messages into the enCRDT, ensuring they are encrypted and authenticated together with their associated data using a secret key. Intermediaries can forge new messages using incorrect keys, but this will be detected when a trusted replica decrypts the AEAD value. The recombine operator (Line 44) reconstructs the plaintext CRDT state by decrypting every message (filtering out those where decryption or authentication fails) and

merging them together according to the lattice of the plaintext CRDT.

Assuming that AEAD protects data confidentiality and authenticity of the contained messages, even the naive enCRDT prevents the following potential attacks. Recombining is based on the underlying lattice, and thus the resulting plaintext state is independent of the order in which intermediaries forward the messages. Since merging is idempotent, replay attacks using duplicated messages also have no effect. The only way for intermediaries to interfere with the replication is to selectively stop disseminating messages to some or all replicas. In the case that replicas can only disseminate messages through intermediaries, this would give the intermediaries control over whether replicas can participate in the replication, i.e., send and receive updates. This is bad but not worse than the scenario where the intermediary did not exist.

The naive approach shows that it is possible to reconcile privacy and integrity of data with coordination-free consistency in the presence of untrusted intermediaries. However, it is not an efficient solution, as intermediaries have to store the ciphertext of every state. We present solutions for this issue next.

3.2 Pruning of Subsumed States

Pruning addresses the common case where most replicas are connected and most changes are merged at every replica. Each state in a state-based CRDT contains the changes that causally happened before it, thus, in the best case, only the most recent message containing that fully merged state is necessary. Even in less optimal cases, intermediaries could discard all states that are *subsumed* by another state without loss of data.

A state s is said to subsume another state s', if s contains all updates of s', that is $s' \sqsubseteq s$. Containing all updates is essentially equivalent to being the least upper bound in the join semilattice of states: $s' \sqsubseteq s \iff (s = s' \sqcup s)$. Since intermediaries cannot obtain subsumption information from the ciphertext, they need to rely on logical timestamps [26] in the form of *version vectors* [12] attached to the messages as associated data. Intuitively only the latest states according to the logical times are kept as they subsume all earlier states. More formally only the states in K are kept with $K \coloneqq \{s | \nexists s' : s < s'\}$, since $s < s' \implies s \sqsubseteq s'$.

Figure 6 shows the Scala implementation of the subsuming enCRDT. The state type of all enCRDTs is the same, but this one stores associated data in the AEAD. In particular, the version operator (Line 60) produces a *version vector* reflecting the combined latest version of the enCRDT. The version vector implementation is, practically speaking, a counter CRDT – reusing the implementation from Figure 2, but renamed to reflect its use. The latest version is computed by merging all associated versions in the enCRDT.

Whenever one replica sends a new message it increments its counter in the associated version (Line 60). The causality information now states that the new message subsumes all prior states, thus Line 61 ensures that this is always the case. Finally, the merge function removes subsumed states by computing the set K described earlier (Line 52).

All latest states must be stored by the intermediaries since they cannot merge ciphertext and each such state contains at least some unique information. However, any trusted replica can merge the encrypted messages and the result will subsume all other states in the intermediary, thus state on intermediaries is minimized as soon as they are connected to a trusted replica. The subsuming enCRDT state has a behavior similar to a multi-value register – another well-known CRDT type – and indeed our actual implementation of subsuming enCRDTs is based on a multi-value register.

```
type EnCRDT[S] = Set[AEAD[S]]

given [S]: Lattice[EnCRDT[S]] with
  def merge(left: EnCRDT[S], right: EnCRDT[S]): EnCRDT[S] =
    val combined = left union right
    combined.filterNot(
      s => combined.exists(o => s.metadata < o.metadata))

extension [S] (c: EnCRDT[S])
  def version: Version = c.map(_.metadata)
                          .reduceOption(Lattice.merge[Version])
                          .getOrElse(Version.zero)

  def send(data: S, key: Secret, replicaID: ReplicaID): EnCRDT[S] =
    val causality = c.version merge c.version.inc(replicaID)
    Set(encrypt(recombine(key) merge data, causality, key))

  def recombine(key: Secret)(using Lattice[S]): Option[S] =
    c.flatMap(decrypt(key)).reduceOption(Lattice.merge[S])
```

Figure 6 Subsuming enCRDT based on version data.

3.3 Delta enCRDTs

Another approach to reduce the storage size of enCRDTs on intermediaries is to combine them with delta replication and only store deltas as encrypted messages. The implementation for this delta enCRDTs is identical to the naive enCRDT. The difference is that trusted replicas do not send full states but just deltas. If causality must be ensured, messages should include the full version as associated data to detect cases when intermediaries forward messages incorrectly. This is achieved by adapting the recombine operator to ignore any messages where no full causal chain to the initial version is available.

For some CRDTs, such as an add-wins set where we assume that all insertions are unique, delta enCRDTs are optimal in the sense that no two encrypted states contain redundant information. Generally, delta enCRDTs are optimal for delta CRDTs, where the deltas never subsume any prior state. For the add-wins set, this is the case, because all prior additions to the set are still present when a new element is added. However, for other CRDTs, many operators create subsuming deltas. For example, every increment operator of the counter/version subsumes all prior deltas created by the same replica.

3.4 Dotted Delta enCRDTs

We can combine the subsuming enCRDT and delta enCRDT techniques to enable subsumption of delta messages. Consider again the case of subsuming enCRDTs. The associated data is a version vector that describes both the version of the message and the set of subsumed messages (all messages with a smaller version vectors). But for a delta its version and the set of subsumed versions are no longer identical. Thus, we split the metadata into the contained versions and the subsumed versions.

A single version is referred to as a *dot* [37] – dots use the same implementation as a counter/version lattice, but are interpreted to represent a different use and renamed accordingly. Version and subsumption information is contained in a *dotted version vector* [37, 2], which

is simply a set of dots, but with the implication that the implementation is optimized to store contiguous ranges of dots efficiently.

The result is a dotted enCRDT, which stores – as associated data – two dotted version vectors, one for contained dots and one for subsumed dots. The implementation for dotted enCRDT is similar to subsuming enCRDTs, only the check for subsumption uses the subset test between the contained respectively subsumed dotted version vectors, instead of testing which version vector is smaller. As an additional note of clarification, subsumption information can be computed automatically using the merge function, thus there is no opportunity for developers to introduce causality errors.

A potential disadvantage of the dotted enCRDT is that the system must compute and manage the dotted version vectors. However, the version metadata can also be used for more efficient implementations of CRDTs and to ensure causality for delta replication [3], thus amortizing the complexity cost. A remaining disadvantage is the amount of metadata available as plaintext to the intermediaries. We discuss this issue in Subsection 4.3.

4 Coordination-free Secure Cryptography

This section presents our cryptographic system that provides authenticated encryption and decryption capabilities via *authenticated encryption with associated data* (AEAD) [41]. AEAD is a form of encryption that provides the confidentiality and authenticity required by Section 3. It relies on symmetric-key cryptography where only trusted parties (replicas) have access to a single shared key. AEAD works on a plaintext s and associated data c (we use s for state and c for causality information respectively). The *Message Authentication Code* (MAC) m ensures authenticity of (s, c), and symmetric encryption secures plaintext s. If the AEAD system is used correctly (a) only trusted parties (replicas) can decrypt messages, and (b) modified or forged encrypted messages – including associated data – are refused by all trusted parties (replicas).

AEAD systems are well studied and widely used, e.g., in TLS 1.3 [40]. However, each use of a cryptographic construction in a new field requires to carefully select concrete implementations of cryptographic functions and ensuring that they are executed with suitable parameters. In particular, all commonly available cryptographic functions suitable to our use case require a certain amount of coordination to ensure correct use of parameters. A key challenge we solve is to systematically minimize the amount of the needed coordination.

In the following, we first introduce our selection of AEAD implementations and their availability in common scenarios. We then discuss the best strategies to ensure correct use of parameters with a minimal amount of coordination. While there is no single best solution, we implement multiple choices with concrete insights on how many replicas are securely supported and how many operations they can execute without coordination. Lower bounds start at 92,000 replicas coordinating once every 130 years, ranging to any number of replicas without coordination for thousands of years. Finally, we discuss what information our system leaks in its metadata and potential ways to minimize this.

4.1 Availability of Concrete AEAD Constructions

Our supported AEAD constructions are *AES-GCM*, *AES-GCM-SIV*, and *XChaCha20-Poly1305*. Figure 7 shows the availability of the constructions in the *Java Cryptography Architecture*

	Java	Web	libsodium	Tink
AES-GCM	•	•	•	•
AES-GCM-SIV				•
ChaCha20-Poly1305	•		•	•
XChaCha20-Poly1305			•	•

Figure 7 Overview of supported AEAD modes in various environments.

(JCA)[4], *Web Cryptography API*[5], *libsodium*[6], and *Tink*[7]. All libraries support AES-GCM due to its use in the TLS specification [40]. The more modern AEAD construction *ChaCha20-Poly1305* was introduced in TLS 1.3 [40] and is currently also supported by all libraries except *Web Cryptography API*. XChaCha20-Poly1305 [4] is an adaption of ChaCha20-Poly1305 with a larger nonce-size and proven to be at least as secure [8]. While not yet standardized by IETF, it is supported by *libsodium* and *Tink* [4]. Another AEAD construction that is currently implemented only in *Tink* but might be adopted by many libraries in the future is AES-GCM-SIV [19]; it introduces a highly interesting new security characteristic, namely that its construction is resistant to parameter reuse, which simplifies coordination-free parameter selection.

4.2 Coordination-free Generation of Nonces for AEAD

To encrypt and authenticate a message, all considered authenticated symmetric encryption schemes require three inputs. The message, the encryption key, and a *nonce* [42]. A nonce is a **n**umber that must only used **once** together with the same key. If a nonce is used multiple times with the same key, then encryption schemes leak information about the plaintext. For example, in AES, an attacker learns the bitwise exclusive-or of messages with the same nonce [29]. The AES-GCM construction even allows an attacker to forge authenticated messages when a nonce is reused [22]. Nonce misuse is not a theoretical problem, but one that leads to severe real-world attacks, e.g., on TLS [10] and WPA2 [50].

The issue is that the decision on how to choose nonces is left to the developer, and, unfortunately, previous research on crypto misuses has shown that developers struggle with secure choices for crypto APIs [25, 34, 39]. This is not surprising, considering that libraries like the *Web Cryptographic API* do not even document that nonces should be unique.

In our system, the uniqueness is ensured transparently. But ensuring uniqueness is a classical coordination problem. Next, we discuss how to select unique nonces without coordination, while staying within generally accepted levels of certainty for the provided confidentiality.

4.2.1 Selecting Nonces by Space Partitioning

A textbook approach to ensure that a nonce is only used once is to employ a strictly monotonic counter that provides a unique nonce for each message [14]. This is the case for AEAD algorithms in TLS 1.3, where the specification mandates the use of the TLS sequence number to compute the nonce [40]. Using a single counter for all replicas is not possible with-

[4] https://docs.oracle.com/en/java/javase/16/security/
[5] https://www.w3.org/TR/WebCryptoAPI/
[6] https://libsodium.org/
[7] https://developers.google.com/tink

out coordination, since this is a prime example of mutual exclusion. An adaption of the counter approach is to partition the nonce space into multiple ranges, each exclusive to a single replica. The resulting counter consists of a constant replica-specific number and the incrementally increasing replica-specific counter. This strategy requires coordination only once, when the replica is initialized, and is generally a good choice for a set of devices provided by a single instance (i.e., devices of a single user or company). In large groups of loosely cooperating devices, however, short unique replica-specific numbers are not generally available deterministically and instead, *cryptographically secure pseudorandom number generation* (CSPRNG) can be used.

Using replica IDs for partitioning. As seen with our counter-CRDT example (Figure 2) many applications already require replica-specific IDs for their behavior. Typical examples for replica-IDs are randomly generated UUIDs, as seen in the *automerge*[8] library, or a hash of a replica-specific public-key [23]. Therefore it seems intuitive to reuse the replica ID to partition the space of nonces. In the case that the chance of collisions of any two replica IDs is small enough to be negligible, this is a secure choice. However, to ensure uniqueness, the size of such identifiers is usually 128 bits [27], which is too large for use with popular AEAD constructions. The NIST specification for AES-GCM, e.g., recommends that implementations should restrict their support of nonce lengths in AES-GCM to 96 bits [14]. Thus, at least for AES-GCM, direct use of such replica IDs is not possible.

Using small random replica-specific numbers for partitioning. Instead of using the replica ID, we can generate short replica-specific numbers using a CSPRNG, but this leaves us with a probability of collisions of replica-specific numbers, thus a collision of nonces. According to the NIST specification, the probability that a nonce is reused for a given key must be less or equal to 2^{-32} [14]. Considering the birthday paradox [45], there is a surprisingly high probability that two replicas choose the same replica-specific number. For example, when choosing a 64-bit long replica-specific number, we can have 92,000 replicas before the collision probability reaches over 2^{-32} and thus the NIST specification is violated.

Assuming 92,000 replicas are sufficient, and given the explicit 96-bit nonces of AES-GCM, a 64-bit replica-specific number leaves room for 32-bit replica-specific counters. This provides $2^{32} \approx 4.3 \times 10^9$ messages to each replica. Assuming that a replica would encrypt one message every second, the counter could be used for over 136 years, before requiring coordination to select a new shared secret. This is the best choice when only AES-GCM is available.

4.2.2 Selecting Fully Random Nonces

A fully coordination-free approach to nonce generation is to rely on a CSPRNG to generate a new random nonce for each message. Literature warns against random nonces in some cases [10]. For example, nonces in TLS (using AES-GCM) consist of 32-bit part specific to the sender and connection, and a 64-bit part to ensure uniqueness [43]. With 64 bit random nonces the collision probability after encrypting $2^{28} \approx 2.7 \times 10^8$ messages would be around 0.2 % and for $2^{32} \approx 4.3 \times 10^9$ messages around 39 % [10].

For using 96-bit random nonces with AES-GCM, the *libsodium* documentation recommends against it [28], while the documentation of Google's cryptography library *Tink* recommends it for "most uses" [17]. Specifically, Tink guarantees that their AES-GCM construction with random nonces can be used for encryption of at least $2^{32} \approx 4.3 \times 10^9$ messages,

[8] https://github.com/automerge/automerge

while keeping the attack probability smaller than 2^{-32} [17].

This, however, is a global message limit, i.e., counting all messages encrypted by all replicas using the same key. The only way to enforce this limit without coordination is to restrict the number of distinct messages to $\frac{2^{32}}{n}$, where n is the maximum number of replicas that can use a single key. Thus, further limiting the number of encrypted messages. Assuming 1024 as an upper bound on the number of replicas, this leaves $\frac{2^{32}}{1024} = 2^{22} \approx 4.2 \times 10^6$ messages to each replica. Or, in other words, 7 weeks of coordination-free operation using one message per second. Moreover, enforcing a limit on the number of replicas also requires coordination.

However, random nonces become practical with very large nonces supported by XChaCha20-Poly1305 [4]. The use of 192-bit nonces allows $2^{80}(\ 10^{24})$ messages to be encrypted with a nonce collision probability of 2^{-32} [4]. To put this in context, if every possible of the 2^{32} IPv4 devices is encrypting messages at the rate of one message per *milli*second, this leaves us with over 8900 years before we must rotate keys. Therefore, XChaCha20-Poly1305 should be strongly preferred over AES-GCM, if it is (efficiently) available on the target platform.

4.2.3 Nonce Misuse-resistant AEAD Schemes

A newer development are *full nonce misuse-resistant authenticated encryption schemes*, such as AES-GCM-SIV [19]. In contrast to the previously discussed AEAD schemes, it is secure to reuse a nonce for the same key with a different message. Thus it is, in theory, a good candidate for use with shared, long-lived keys. However, as discussed in Figure 7, an implementation of AES-GCM-SIV is not widely available, and the scheme has not been scrutinized as much as other presented approaches.

4.3 Information Leaks

If used correctly and securely, AEAD constructions provide confidentiality of the plaintext. This, however, does not necessarily mean that all sensitive information remains confidential. One scenario where this becomes quite clear is our counter example (Figure 2). The counter only supports one mutator: incrementing by one. Thus the state of the counter is equivalent to the version vector associated with the ciphertext. Recall that our example uses the same implementation for the counter and the version vector. There are also other types of information leaks to consider, like the size of states in transit and who disseminates states at what time, etc. However, these issues are not unique to our solution, and countermeasures exist [20, 49]. Moreover, because enCRDTs do not require a central entity, it becomes easier to apply countermeasures. An example countermeasure is to split messages over multiple intermediaries (such that no single intermediary may learn all metadata), or use randomized routing such as TOR [13]. Both are feasible because our solution is resistant to any form of delay and message reordering.

5 Implementation and Evaluation

We evaluate our proposal along the following research questions:

- RQ1: Is our approach suitable for designing common CRDTs and for developing applications using them?
- RQ2: Is the performance overhead for encryption small enough for general use?
- RQ3: Is the space overhead of enCRDTs on intermediaries acceptable?

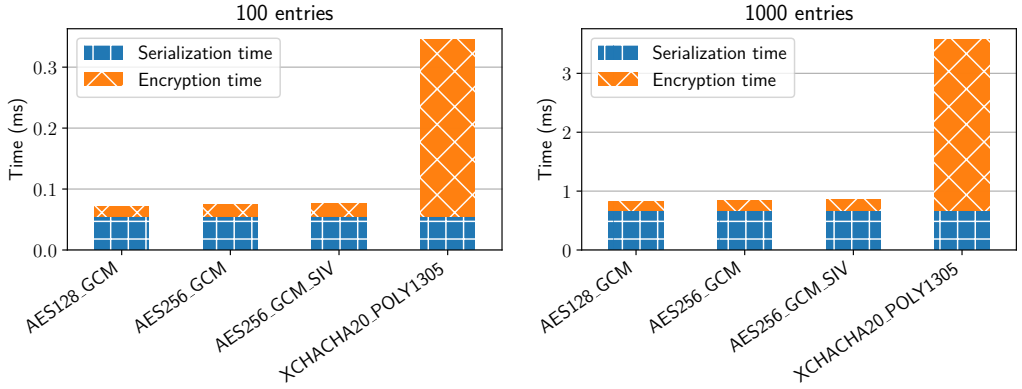

Figure 8 Encryption vs. serialization time for AWLWWMap states containing many entries.

5.1 RQ1: Suitability for Designing CRDTs and Applications

While we believe that enCRDTs have the most value when adding them to an existing system, because existing implementations for CRDTs and for the anti-entropy layer can be reused, we implemented a self-contained prototype of our overall approach. The prototype contains implementations for common CRDTs [46] and delta state CRDTs [3], our four enCRDTs, and an anti-entropy implementation using WebSockets based on Eclipse Jetty[9]. It serves three purposes: as a reference for implementations in other systems, to test our approach to designing CRDTs, and to provide data about the cost of enCRDTs.

The prototype makes heavy use of the CRDT composability introduced in our architecture. For example, the tombstone-free map is composed out of a tombstone-free add-wins set [9] and a last-writer-wins register. If the add-wins-last-writer-wins semantic does not fit a use case the inner last-writer-wins register can be substituted, e.g., with a multi-value register to keep multiple concurrent values in the map. The same applies to the add-wins set, which could be replaced with remove-wins, grow-only, or a two-phase set.

We have found no limitations on implementing CRDTs using our approach and all of them can be securely disseminated using our enCRDTs. We also implement the popular to-do list example as a JavaFX GUI application which uses the add-wins-last-writer-wins map (AWLWWMap) for its primary state – the data structure we will use primarily for the evaluation of the overhead of enCRDTs. Our implementation is publicly available, link removed for double-blind review.

5.2 RQ2: Time Overhead of enCRDTs

For the benchmark, we use JMH[10] the standard Java benchmarking tool executed on a 2015 Intel Core i7-6700HQ Laptop CPU. The main overhead of using enCRDTs for a trusted replica is the cost of AEAD, which is in our case provided by Tink[11], which uses hardware acceleration for AES-GCM and AES-GCM-SIV. To quantify the overhead and put it into perspective, we serialize and then encrypt a CRDT state. To serialize states, we use jsoniter-

[9] https://www.eclipse.org/jetty/
[10] https://openjdk.java.net/projects/code-tools/jmh/
[11] https://developers.google.com/tink

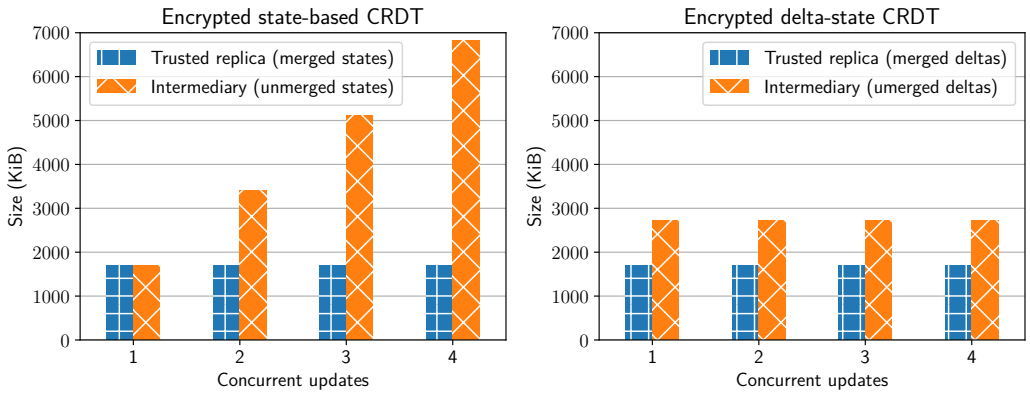

Figure 9 The size difference of the state size for trusted and untrusted replicas between encrypted state-based CRDTs (left) and encrypted delta-based CRDTS(right).

scala[12] (the arguably fastest JSON serializer available on the JVM[13]).

We compare the encryption overhead on top of the serialization cost (paid even without enCRDTs) between the different AEAD schemes, using the add-wins-last-writer-wins map with 100 and 1,000 entries as test cases. The results in Figure 8 reveal that the computation overhead of the different AEAD implementations is usually small in absolute terms. In the worst case of sending the full state of a set with 1,000 entries, the overhead is only a fraction of a millisecond due to hardware acceleration. XChaCha20-Poly1305 does not benefit from hardware acceleration yet still has an overhead of less than 3 ms in the worst case. XChaCha20-Poly1305 is an especially suitable choice on systems where hardware acceleration is also unavailable for AES, because is designed to be efficiently implemented in software [4].

To answer our research question, the overhead of introducing enCRDTs is a fraction of a millisecond, as it can reuse common and efficient encryption schemes. Thus, we believe that enCRDTs are a suitable and secure solution for most use cases.

5.3 RQ3: Space Overhead of enCRDTs

The Intermediaries of enCRDTS cannot merge states due to the encryption and need to store all concurrent encrypted states. Thus the space requirements of enCRDTs on intermediaries is higher than of the wrapped CRDT. While the absolute size of an enCRDTs is dependent on the underlying CRDT, the AWLWWMap we present here shows the general trends of the enCRDTs independently of the wrapped CRDT. The naive enCRDT and dotted enCRDT are not considered, because the first has the same behavior as the shown worst case of the subsuming enCRDT, and the latter is somewhere in between the shown cases depending on the quality of the causality information. Encrypted sizes depend on the number of concurrent updates, that is, how many updates the intermediary received from different sources, without being connected to a trusted replica that merged the encrypted state. In general, concurrent updates are either quickly merged by active replicas, or no new updates are produced because there are no connected trusted replicas – in both cases, the number

[12] `https://github.com/plokhotnyuk/jsoniter-scala`
[13] `https://plokhotnyuk.github.io/jsoniter-scala/`

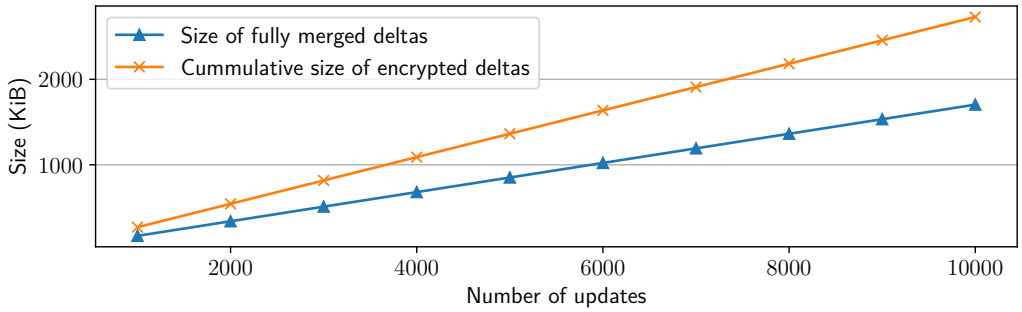

Figure 10 Cumulative size of all encrypted deltas in AWLWWMap compared to merged state.

of concurrent updates remains small.

Figure 9 shows the space requirement of storing a AWLWWMap with 10,000 entries using a subsuming enCRDT (left) and a delta enCRDT (right) on a trusted replica versus intermediary when there are 1 to 4 concurrent updates. The state size of the trusted replica is the same in all cases (technically the size increases slightly as a new entry is stored, however, the effect of this is negligible), as it always merges all received updates. For the intermediary, however, we observe that the size of subsuming enCRDT (left subfigure) grows linearly with each concurrent update. We expected this result as each update contains the full state which must be stored. For the delta enCRDT (right subfigure), the state for a single concurrent update is larger than the state of the trusted replica because each delta is stored separately, which introduces a constant overhead per delta. However, each further concurrent update only marginally increases the state size because only the single additional delta is stored.

Figure 10 quantifies the size difference of the AWLWWMap when stored as a merged state versus being stored as the set of its constituent deltas. We can see that the storage as deltas has a fixed relative overhead of 60 %. This is typical behavior for most CRDT state implementations because a delta is simply a singleton instance of the CRDT state, and there is a fixed overhead to represent this state. Thus each additional entry makes the set of deltas grow at a constant but faster rate than the merged state, which also grows at a constant rate.

In general, storing only the deltas at intermediaries does not cause the state to increase due to concurrent updates but has a fixed cost associated. Which strategy is more suitable depends on how reliable the connection between trusted replicas and intermediaries is, and how many different intermediaries are part of the system. In summary, we believe that one of the presented enCRDTs is suitable for most use cases. If other behavior is required, new variants of enCRDTs with different subsumption strategies can be used.

6 Related Work

6.1 Conclict-Free Replicated Data Types

Shapiro et al. [47] formalize CmRDTs (operation-based) and CvRDTs (state-based) as well as strong-eventual consistency as a solution to the problems of the CAP theorem. In the accompanying technical report [46] they additionally describe several concrete state-based and operation-based CRDTs. The foundation of the strong eventual consistency guarantees

of CRDTs is the CALM theorem [21]. While CAP [11] describes an impossibility of coordination freedom and consistency for arbitrary algorithms, the CALM theorem proposes a programming model restricted to monotonic operations where conflict-free consistency can be achieved. In the case of state-based CRDTs, this monotonicity comes from the "monotonically non-decreasing join semilattices" [47].

CRDTs originate from technology surrounding highly-available databases, which has commonalities with "local-first software", albeit the ostensible differences. Two popular general purpose CRDT implementations that are not part of databases are automerge[14] and Yjs[15] [35]. Both of them are JavaScript frameworks and work in ad-hoc peer-to-peer environments. They both rely on operation-based CRDT protocols. Automerge is loosely based on a paper by Kleppmann et al. [23] that describes a CRDT that replicates a JSON-like data model. This paper describes a formal semantics of concurrently editing arbitrary JSON-structures and showcases problems with concurrent modifications on replicated, nested structures. Compared to both database approaches and the approaches above, our solution relies on an "open" approach to a CRDT-based architecture, where developers can add their own data types instead of being restricted to a pre-defined set of types.

Almeida et al. [3] introduce the concept of delta state replicated delta types (delta-CRDTs). They claim that their delta-CRDTs can achieve small message sizes that are prevalent in operation-based CRDTs but not necessarily in state-based CRDTs. Contrary to operation-based CRDTs, delta-CRDTs rely on the constructs of state-based CRDTs, allowing the dissemination of updates over unreliable communication channels. They introduce two anti-entropy algorithms with one of them assuring causal consistency. In their paper they also introduce several concrete delta-CRDTs and a framework for causally consistent delta-CRDTs. The described framework was used in production as part of Riak DT[16] and relies on composition of *dot-stores* that share a common causal-context. These dot-stores are closely related to dotted version vectors [37, 2]. Our enCRDT approach makes heavy use of the concepts from dot-stores, but we put a much higher focus on composability of merge functions, and make an exlicit split between public metadata for causality, and the private state of CRDTs.

6.2 Security in CRDT systems

Preguiça et al. [38] give a broad overview on the research and applications around CRDTs and also have remarks on future directions of research. One of these directions is the area around security in CRDT systems. They observe that while it is possible to restrict access to a CRDT based interface using authentication, the replicas themselves are vulnerable to harmful operations on any other replica. Furthermore they also describe a similar idea to what is discussed in this paper: end-to-end encryption of states stored on third parties. This end-to-end encryption would remove the need to trust the provider of the third party. They state that this would require pushing most of the computation to the edge (i.e., the client). Two alternatives that they suggest are homomorphic encryption and hardware-supported trusted execution (e.g., Intel SGX and ARM TrustZone).

Barbosa et al. [5] introduce "privacy-preserving CRDT protocols". They use a mixture of custom techniques and solutions from the space of homomorphic encryption to allow clients

[14] `https://github.com/automerge/automerge`
[15] `https://github.com/yjs/yjs`
[16] `https://github.com/basho/riak_dt`

to interface with a distributed database (represented as a CRDT) without fully trusting the provider that hosts them. Specifically, they introduce an extension to AntidoteDB[17], which is a "geo-replicated NoSQL database that leverages CRDTs". They assume *honest-but-curious* adversaries, which means that the attacker (i.e., cloud service provider) is bound to service level agreements, and only interested in secretly extracting information. In their mode clients are not replicas themselves, thus require an intermediary to execute operations. The clients then use custom cryptographic methods to issue operations to the CRDTs hosted on the providers in a way that the provider may not read the data. This is presumably what Preguiça et al. [38] referred to with moving computations to the edge, as Preguiça is a co-author of both publications. Crucially, an adversarial provider could modify data and stop the client from executing any operations, because operations can not be authenticated. Moreover, all of their cryptographic constructions are specific to individual CRDTs.

7 Conclusion

Enabling users to continue working when they are offline and consistently synchronizing their data when they are online requires a coordination-free synchronization mechanism. While CRDTs address this issue, they were not designed with security in mind: Every replica is inherently trusted and no measures are taken to ensure data confidentiality and authenticity. We explained that this is especially problematic when untrusted intermediaries are used to cope with the realities of connectivity in the open internet.

The foundation of our solution is an approach for systematic, modular, and extensible design of custom CRDTs. This approach facilitates the integration of CRDTs into existing programming models and existing network runtimes. Further, provides confidentiality and authenticity by design, i.e., transparently for the application developers. Specifically, we presented a family of four encrypting CRDTs (enCRDTs) for different network requirements. Each such enCRDT provides a secure layer between the data of a CRDT and the network runtime that disseminates the data over untrusted connections.

Using our enCRDTs the application data is transparently encrypted while retaining coordination freedom. Our solution abstracts the complexity of encryption from the developer to avoid misuses of cryptographic primitives which occur very often in practice. Our evaluation shows that we can implement any CRDT with our approach and any of them can be securely disseminated using our enCRDTs. The performance overhead is only a small fraction of the existing dissemination cost. The additional storage requirement is limited by the amount of concurrent changes in the worst case and can be minimized further for CRDTs with an efficient delta decomposition. Together, the results of the experiments show that it is feasible to use the proposed solution in practice.

A remaining issue – common to all encrypted synchronization techniques – is that it needs to leak metadata to enable efficient dissemination of messages. However, because our approach is resilient to poor network conditions including reordering, delay, and duplication of messages, we believe that many common mitigation techniques can be applied without impeding normal operations. Such mitigations include sending fake data to make metadata less usable or routing data on multiple intermediaries such that no single one has a full view of the system. We may also be able to apply concepts from homomorphic encryption or secure enclaves to enable intermediaries to learn which states subsume each other, without gaining any further insight into the exact metadata of each message.

[17] `https://www.antidotedb.eu/`

References

1. Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171, 2017. doi:10.1109/SP.2017.52.
2. Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno Preguiça, and Victor Fonte. Scalable and accurate causality tracking for eventually consistent stores. In Kostas Magoutis and Peter Pietzuch, editors, *Distributed Applications and Interoperable Systems*, pages 67–81. Springer Berlin Heidelberg. URL: http://haslab.uminho.pt/tome/files/dvvset-dais.pdf.
3. Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. 111:162–173. URL: https://www.sciencedirect.com/science/article/pii/S0743731517302332, doi:https://doi.org/10.1016/j.jpdc.2017.08.003.
4. Scott Arciszewski. Xchacha: extended-nonce chacha and aead_xchacha20_poly1305. Internet-Draft draft-irtf-cfrg-xchacha-03, Internet Engineering Task Force. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-xchacha-03.
5. Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela, and Nuno Preguiça. Secure conflict-free replicated data types. In *International Conference on Distributed Computing and Networking 2021*, ICDCN '21, page 615, New York, NY, USA. Association for Computing Machinery. doi:10.1145/3427796.3427831.
6. Lars Baumgärtner, Jonas Höchst, and Tobias Meuser. B-dtn7: Browser-based disruption-tolerant networking via bundle protocol 7. In *2019 International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)*, pages 1–8, 2019. doi:10.1109/ICT-DM47966.2019.9032944.
7. Lars Baumgärtner, Patrick Lieser, Julian Zobel, Bastian Bloessl, Ralf Steinmetz, and Mira Mezini. Loragent: A dtn-based location-aware communication system using lora. In *2020 IEEE Global Humanitarian Technology Conference (GHTC)*, pages 1–8, 2020. doi:10.1109/GHTC46280.2020.9342886.
8. Daniel J. Bernstein. Extending the salsa20 nonce. In *Workshop Record of Symmetric Key Encryption Workshop 2011*. URL: https://cr.yp.to/snuffle/xsalsa-20110204.pdf.
9. Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. arXiv:1210.3368.
10. Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. USENIX Association. URL: https://www.usenix.org/conference/woot16/workshop-program/presentation/bock.
11. Eric Brewer. Cap twelve years later: How the "rules" have changed. 45:23–29. doi:10.1109/MC.2012.37.
12. Russell Brown. Vector clocks revisited. URL: https://riak.com/posts/technical/vector-clocks-revisited/index.html.
13. Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association. URL: https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router.
14. Morris J. Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. Technical report. doi:10.6028/nist.sp.800-38d.
15. Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 5061, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2382196.2382205.

16 Seph Gentle. 5000x faster crdts: An adventure in optimization. https://josephg.com/blog/crdts-go-brrr/, 2021. Accessed 2021-11-01.

17 Google. Authenticated encryption with associated data (aead). URL: https://developers.google.com/tink/aead.

18 Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

19 Shay Gueron and Yehuda Lindell. Gcm-siv: Full nonce misuse-resistant authenticated encryption at under one cycle per byte. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 109119, New York, NY, USA, 10 2015. Association for Computing Machinery. doi:10.1145/2810103.2813613.

20 Christoph Hagen, Christian Weinert, Christoph Sendner, Alexandra Dmitrienko, and Thomas Schneider. All the numbers are us: Large-scale abuse of contact discovery in mobile messengers. Internet Society, 2021. The papaer has been accepted for publication at the conference NDSS 2021. The conference will take place from February 21-24, 2021 in San Diego, California. Event Title: 28. Annual Network and Distributed System Security Symposium (NDSS'21).

21 Joseph M. Hellerstein and Peter Alvaro. Keeping calm: When distributed consistency is easy. 63(9):72–81. doi:10.1145/3369736.

22 Antoine Joux. Authentication failures in nist version of gcm. URL: https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/joux_comments.pdf.

23 Martin Kleppmann and Alastair R. Beresford. A conflict-free replicated json datatype. 28(10):27332746. URL: http://dx.doi.org/10.1109/TPDS.2017.2697382, doi:10.1109/tpds.2017.2697382.

24 Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 154178, New York, NY, USA. Association for Computing Machinery. doi:10.1145/3359591.3359737.

25 Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An extensible approach to validating the correct usage of cryptographic APIs. *IEEE Transactions on Software Engineering*, 47(11):2382–2400, 2019. doi:10.1109/TSE.2019.2948910.

26 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. 21(7):558–565. doi:10.1145/359545.359563.

27 Paul J. Leach, Rich Salz, and Michael H. Mealling. A universally unique identifier (uuid) urn namespace. RFC 4122. URL: https://rfc-editor.org/rfc/rfc4122.txt, doi:10.17487/RFC4122.

28 Libsodium Project. Aes256-gcm. URL: https://libsodium.gitbook.io/doc/secret-key_cryptography/aead/aes-256-gcm.

29 David McGrew. An interface and algorithms for authenticated encryption. RFC 5116. URL: https://rfc-editor.org/rfc/rfc5116.txt, doi:10.17487/RFC5116.

30 Kai Mindermann, Philipp Keck, and Stefan Wagner. How usable are rust cryptography apis? In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 143–154, 2018. doi:10.1109/QRS.2018.00028.

31 Ragnar Mogk. *A Programming Paradigm for Reliable Applications in a Decentralized Setting*. PhD thesis, Technische Universität Darmstadt, Darmstadt, March 2021. URL: http://tuprints.ulb.tu-darmstadt.de/194035/.

32 Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant distributed reactive programming. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 1:1–1:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ECOOP.2018.1.

33 Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A fault-tolerant programming model for distributed interactive applications. *Proc. ACM Program. Lang.*, 3(OOPSLA):144:1–144:29, 2019. doi:10.1145/3360570.

34 Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: why do java developers struggle with cryptography APIs? In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 935–946. ACM, 2016. doi:10.1145/2884781.2884790.

35 Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In Philipp Cimiano, Flavius Frasincar, Geert-Jan Houben, and Daniel Schwabe, editors, *Engineering the Web in the Big Data Era*, pages 675–678. Springer International Publishing.

36 Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. Why eve and mallory still love android: Revisiting TLS (In)Security in android applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4347–4364. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/oltrogge.

37 Nuno Preguiça, Carlos Bauqero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Brief announcement: Efficient causality tracking in distributed storage systems with dotted version vectors. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, page 335336, New York, NY, USA. Association for Computing Machinery. URL: http://gsd.di.uminho.pt/members/vff/dotted-version-vectors-2012.pdf, doi:10.1145/2332432.2332497.

38 Nuno Preguiça, Carlos Baquero, and Marc Shapiro. Conflict-free replicated data types (crdts). arXiv:1805.06358, doi:10.1007/978-3-319-63962-8_185-1.

39 Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 24552472, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319535.3345659.

40 Eric Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446. URL: https://rfc-editor.org/rfc/rfc8446.txt, doi:10.17487/RFC8446.

41 Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, page 98107, New York, NY, USA, 11 2002. Association for Computing Machinery. doi:10.1145/586110.586125.

42 Phillip Rogaway. Nonce-based symmetric encryption. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2004. doi:10.1007/978-3-540-25937-4_22.

43 Joseph A. Salowey, David McGrew, and Abhijit Choudhury. Aes galois counter mode (gcm) cipher suites for tls. RFC 5288. URL: https://rfc-editor.org/rfc/rfc5288.txt, doi:10.17487/RFC5288.

44 Sebastian Schildt, Tim Lüdtke, Klaus Reinprecht, and Lars Wolf. User study on the feasibility of incentive systems for smartphone-based dtns in smart cities. In *Proceedings of the 2014 ACM International Workshop on Wireless and Mobile Technologies for Smart Cities*, WiMobCity '14, page 6776, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2633661.2633662.

45 Bruce Schneier. *Applied cryptography - protocols, algorithms, and source code in C, 2nd Edition.* Wiley, 1996.

46 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA. URL: https://hal.inria.fr/inria-00555588.

47 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer Berlin Heidelberg.

48 Milan Stute, Florian Kohnhauser, Lars Baumgartner, Lars Almon, Matthias Hollick, Stefan Katzenbeisser, and Bernd Freisleben. RESCUE: A resilient and secure device-to-device communication framework for emergencies. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2020. doi:10.1109/TDSC.2020.3036224.

49 Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 137152, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2815400.2815417.

50 Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 13131328, New York, NY, USA, 10 2017. Association for Computing Machinery. doi:10.1145/3133956.3134027.