Encrypted Replicated Data Types within REScala

Christian Kuessner
Technische Universität Darmstadt, Germany

Ragnar Mogk
Technische Universität Darmstadt, Germany

Anna-Katharina Wickert
Technische Universität Darmstadt, Germany

Mira Mezini
Technische Universität Darmstadt, Germany

Abstract

This paper is about programming support for local-first applications that manage private data locally, but still synchronize data between multiple devices – to allow a single user to synchronize their settings and data, and to support collaboration between multiple users. Such applications must never 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 programming model based on application-specific replicated data types with an automatic 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 communication between local-first applications while retaining coordination freedom. Our evaluation highlights that the proposed solution is practical in terms of runtime and data size 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, CRDTs, AEAD

1 Introduction

The dominating software architecture today is centralized: data is collected, managed, and processed in the cloud, while devices on the edge of the communication infrastructure serve only as interfaces to the users. Areas where this architecture is employed include a single user with multiple devices (i.e., synchronizing settings and data), a collaboration between multiple parties (e.g., shared calendars, document editors, business workflows), and building autonomous systems with remote control and interactions (e.g., home automation and autonomous vehicles).

There are many issues with the centralized architecture including lost control over data ownership and privacy, lack of offline availability, poor latency, inefficient use of communication infrastructure, and waste of powerful computing resources on the edge. Worst of all, there is not a single user facing benefit to centralization. To address these issues, local-first software [26] call for "data confidentiality and privacy by default" and "ultimate ownership and control" by the user – achieved by moving data storage and processing to the edge.

However, local-first applications are complex and developing them is challenging. They model and manage rich private data and complex interactions with the environment. Furthermore, coordination-free and secure data synchronization is needed for a range of diverse

network topologies from direct peer-to-peer connections to communication via third party (untrusted) intermediaries, which can serve as post offices that store, forward, and eventually discard old messages when direct connections are not available.

The current approach to developing local-first applications is to organize application design around the data storage layer by mapping application-specific state to existing off-the-shelf *coordination-free replicated data types (CRDTs)* [50]. CRDTs have been invented for efficient coordination-free data synchronization in geo-replicated databases. They are available as off-the-shelf modules in a conceptual data storage layer implemented as databases [48] or libraries [25, 17].

But organizing local-first application design around a CRDT-based data storage layer has deficiencies. First, mapping from application state to a fixed set of off-the-shelf database operations is known to cause application design issues [12]. Second, design choices underlying off-the-shelf CRDTs assume a geo-replicated data store within a controlled environment. Specifically, direct connections between trusted replicas are assumed and hence there is no need to address data confidentiality and integrity in the presence of untrusted intermediaries. Moreover, adding such confidentiality and integrity for customers of the data center is seen as challenging [40].

The way out of the above limitations would be application-specific solutions with encrypted synchronization. But without proper programming support this puts significant burden on application developers, who are typically not experts neither in managing and synchronizing replicated data, guaranteeing consistency properties, nor in ensuring data integrity and privacy, let alone in addressing all in concert. The risk is high that accidental complexity causes applications to be vulnerable and unreliable.

To address this risk, we propose a library-based approach for programming for *application-specific replicated data types* (ARDTs). The design of our library makes use, refines, combines, and expands two existing results – the theory of consistency as logical monotonicity (CALM theorem) [13] and pragmatic results from delta replication [3]. The library offers unified abstractions based on composing functions and data types to implement both complex application logic and custom communication protocols, thus integrating well with existing programming practice. It makes use of these same unified data and functional abstractions to offer an encryption layer for efficient synchronization of ARDTs in any network topology, including untrusted intermediaries.

We evaluate our approach on a case study and with benchmarks. The results show that (a) typical local-first applications can be implemented with reasonable performance, and (b) encryption comes with minimal computational costs and has predictable and reasonable storage overhead on intermediaries.

In summary, our contributions are:

- A critical analysis of the state-of-the-art in developing local-first applications (Section 2)
- An approach for programming application-specific replicated data types (Section 3).
- Synchronization-free authentication and encryption schemes that are themselves provided as application-specific replicated data types (Section 4). As part of designing the encryption scheme, we contribute a systematic analysis of the suitability of existing encryption primitives for decentralized synchronization protocols (Section 5).
- An implementation of our proposal as an embedding into the Scala programming language along with a systematic empirical evaluation (Section 6).

2 State of the Art and Problem Statement

There are two families of existing approaches that are relevant for our work: dedicated systems for collaborative workflows and approaches to replicated data types employed in geo-replicated data stores. In the following, we briefly introduce features of these systems that are relevant and those that are missing with respect to the development of secure local-first software. We also introduce existing building blocks for distributed systems that we use and combine to exploit their advantages in new ways.

2.1 Systems for Distributed Workflows

There are two kinds of systems for distributed workflows. The first kind features automated handling of conflicts at the price of centralized coordination, with prominent examples such as Google Docs or Firefox Sync. The second kind of systems features much more flexible replication that does not rely on centralization, with the most prominent example being Git, which allows for replication via different intermediaries including specialized ones like GitHub but also reusing general ones like Email. Similarly, systems like Syncthing and Resilio Sync enable peer-to-peer file synchronization in an arbitrary network topology by relying on encrypted intermediaries. But Git, Syncthing and Resilio Sync require manual user intervention for conflict resolution.

We aim for combining flexible and secure data synchronization à la Git with automated conflict resolution à la Google Docs. Moreover, we want to offer this combination to general-purpose programs with unconstrained types of data, unlike the above solutions, which target specific use cases and specific types of data. For example, Google Docs builds on a body of research around operational transform to enable efficient synchronization specifically for text documents using a centralized server. Such a dedicated effort is infeasible for the many potential application domains of local-first software. Adapting existing solutions would require developers to become experts and understand the underlying assumptions and data models, or otherwise risk to introduce errors into an adaptation.

2.2 Replicated Data Types in Data Stores

Another class of solutions that are relevant for our purposes are those developed to enable availability in geo-replicated data centers in the presence of network partitions – a scenario that bears some superficial similarity with local-first software. The research results from this context most relevant to the development of local-first applications are conflict-free replicated data types (CRDTs) [50]. CRDTs are data types, whose API consists of a fixed set of query and update operators, which satisfy the condition that two replicas that know of the same updates return the same result for all queries (also known as *eventual consistency*). This property is key in supporting coordination-free synchronization. CRDTs are typically built into a replicated data store with specific assumptions about the underlying network for efficiency (e.g., availability of reliable causal broadcast). The assumptions built into the design of off-the-shelf CRDTs limits their applicability to local-first software development.

First, application developers are left with not much choice but to express their application design using the fixed APIs of existing CRDTs. A similar approach – object-relational mappings in database-centric software – is known to be a leaky abstraction, requiring frequent security relevant changes, and does not work well together with language-based tooling [12].

Second, using off-the-shelf solutions means that choices regarding the underlying networking platforms are also predefined. As these requirements are hidden to the developer it

is impossible to exchange one synchronization mechanism for another. But local-first software may need to operate in varied scenarios for which there is no "one size fits all" solution. And even if there is a suitable way to disseminate messages for the target network scenario, it may not have suitable security guarantees and – with current solutions – these guarantees cannot be added without modifying the implementation of the network communication.

To recap, we need to enable application developers to design application-specific replicated data types capable of coordination-free and secure replication in any network. With current strategies to design CRDTs, developers need to satisfy the constraints required for consistency of the chosen design approach, while simultaneously considering the properties of the targeted network topology. This is too much domain specific knowledge to ask of local-first application developers.

2.3 Building Blocks for Application-specific Replicated Data Types

In the following, we briefly overview results from previous research that provide us building blocks in designing our programming support for application-specific replicated data types.

CALM and lattices. The first result we exploit is the *consistency as logical monotonicity* (CALM) theorem [22], which states that consistency is possible without coordination if and only if all replicas only add to the overall solution, but never invalidate any prior results. That is, coordination-free consistency goes hand in hand with monotonic operations. A constructive way to make use of this insight, is to restrict the available programming support to monotonic functions [13], however, such an approach may be too restrictive and does not integrate well into general-purpose languages.

Instead of relying on a specifically designed language with restricted expressivity, we turn to the concept of lattices and embed those into our design. For us, a lattice is a set of states (in other words, a data type) for which a *merge function* exists, which must be associative, commutative, and idempotent. In the abstract, these properties mean that the result of a merge is monotonic, because a merge function computes the *least upper bound* $\sqcup$ of two states: $x = s_1 \sqcup s_2$ where $s_1 \leq s$ and $s_2 \leq s$. Concretely, associativity ensures that states can be combined before transmission, commutativity tolerates disordered arrivals of messages, and idempotence enables coping with duplicated transmissions. Lattices are widely used to prove correctness of coordination-free replication schemes, because it is sufficient to show that a state forms a lattice and all changes only increment the state (according to the lattice order). However, lattices are so far not directly used during the design of applications.

Delta replication. The second result is delta replication [3], which provides an approach to separate efficient message dissemination from application logic, by expressing monotonic changes as the delta to the prior state. Delta replication is also based on lattices, but each state is separated into many small pieces (called deltas), which when merged together form the state. Delta replication is used to optimize network communication, by trying to transmit each delta only once (unreliable channels may still cause multiple transmissions) instead of having to transfer full states for replication. Moreover this scheme allows for flexible customization of message dissemination (originally called an anti-entropy algorithm) for the specific networking platform.

However, even if delta replication allows separating application logic from message dissemination efficiently, it is still unclear how to add transparent encryption without adapting all existing message dissemination implementations (of which, each application may have its own). In general, message dissemination in a local-first setting needs to consider many

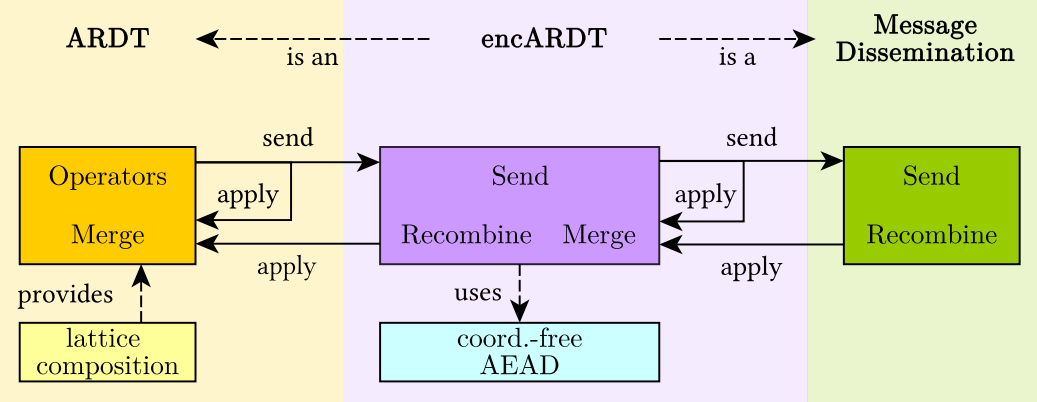

Figure 1 Architecture overview.

application- and environment-specific interwoven concerns, such as causal delivery, importance of messages to the application logic, messages that become obsolete because of newer state changes, and security. This requires a systematic approach to be able to add and customize extensions for each application.

Authenticated encryption with associated data. From a privacy and security perspective, local-first applications require confidentiality – the guarantee that application private data can not be accessed by unauthorized parties – and integrity – the guarantee that neither private data nor communication metadata (the associated data) can be tampered with by an attacker. These guarantees are provided by authenticated encryption with associated data (AEAD) [43] – cryptographic solutions based on symmetric-key cryptography, where only trusted parties (in our case replicas) have access to a single shared key. AEAD is well studied and widely used, e.g., in TLS 1.3 [42].

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, for local-first applications, an unknown number of replicas need to encrypt and decrypt data using a single shared key without coordination. Widely available AEAD functions require a globally unique number (nonce) as an input for each operation using the same key. But guaranteeing global uniqueness requires coordination, which we need to avoid.

3 Programming Local-first Applications with Application-Specific Replicated Data Types

Figure 1 shows an overview of our library for programming local-first applications. The library consists of three modules (colored areas in the figure) corresponding to three concerns of the design of application-specific replicated data types (ARDTs). The module on the left provides building blocks for modeling the application logic of ARDTs, which also includes logical synchronization of replicated state. The module on the right offers building blocks for implementing physical synchronization, which involves disseminating messages to send and receive (recombine) state changes among replicas of an ARDT over a concrete communication network. Finally, the module in the middle offers support for encrypted synchronization (encrypted ARDTs – encARDTs). This section presents the ARDT and

message dissemination modules. The presentation of the encARDT module is deferred to Section 4 and Section 5.

The goal of the library is fostering modular design of ARDTs. We achieve modularity by making consequent use of common abstraction and composition techniques for data and functions – both at the library level and at the level of individual modules.

Developers can combine predefined implementations of each module, and exchange the implementation of individual modules to address specific application needs. From the perspective of the ARDT module, any message dissemination algorithm that provides at-least-once delivery is suitable; this puts the least amount of restrictions on possible networking support, thus allowing network experts to extend the library with new message dissemination algorithms. Moreover, an encARDT is an ARDT that implements the message dissemination module. The state of an encARDT contains the encrypted messages that are managed by the send and receive (recombine) operators, which handle encryption and decryption. As a result, one can plug encARDTs between any existing combination of ARDTs and concrete message dissemination and achieve confidentiality and authenticity with no changes to the ARDT and networking components.

Modularity is key to facilitate development by fostering extensibility and reusability. More importantly, we also rely on modularity to ensure correctness. In particular, ARDTs separate operators for reading and modifying a state from the merge function responsible for encoding the logical synchronization of that state. Merge functions are used to ensure both the correctness of operators (when applying modifications locally) and consistency between replicas (when applying remote modifications). This architecture allows developers to freely add operators to existing ARDTs and to combine existing ARDTs into compound ARDTs. Grounded in the theory of lattices, our library automatically generates correct merge functions of compound ARDTs out of the merge functions of their constituents, thus developers get consistency for free. The developers need to ensure correctness manually, only (in the rare case) when they define custom merge functions that encode custom logical synchronization, and the library helps in this case with support for property-based testing.

3.1 Programming, Using, and Replicating ARDTs

The common way to design an ARDT is to use existing (immutable) types to represent its state. The only requirement on the used types is, that it is possible to define a merge function for that type. Given a suitable state representation, developers can implement operators as functions on that state, with modifications expressed as functions that return a new state.

Availability of a merge function for a type S is modeled by the type class Lattice[S] below (all code examples use Scala 3 syntax). Specifically, S is an ARDT when there is a correct instance of Lattice[S], where correct means that the merge function of that instance is associative, commutative, and idempotent.

```
1 trait Lattice[S] { def merge(left: S, right: S): S }
```

The ARDT module of our library comes with a broad range of built-in ARDTs. Moreover, it automatically generates correct instances of Lattice for standard data structures, user-defined product types, and the compositions of all of the above.

For illustration, assume that we want to implement a local-first social media application to be collaboratively used by a group of friends in a peer-to-peer network to share messages, comments, likes, and dislikes. The ARDT in Figure 2 models the state and operators of such an application with our library. The SocialMedia type (Line 2) is defined as an alias

```
2  type SocialMedia = HashMap[ID, SocialPost]
3  case class SocialPost(message: LWW[String], comments:
       Set[LWW[String]], likes: Counter, dislikes: Counter)
4  // extension allows to define methods on type aliases
5  extension(sm: SocialMedia)
6    def like(post: ID, replica: ReplicaID): SocialMedia =
7      val increment = sm(post).likes.inc(replica)
8      HashMap(post -> SocialPost(likes = increment))
```

Figure 2 Compositional design of the social media ARDTs.

```
9  type Counter = HashMap[ReplicaID, Int]
10 // object defines static methods, extension defines instance methods
11 object Counter:
12   def zero: Counter = HashMap.empty
13 extension (c: Counter)
14   def value: Int = c.values.sum
15   def inc(id: ReplicaID): Counter =
16     HashMap(id -> (c.getOrElse(id, 0) + 1))
```

Figure 3 The state and operators of a counter ARDT.

for a built-in map from IDs to values of type SocialPost (Line 3). The latter is defined as a product type with named components (a case class in Scala). Social posts consist of values of the built-in type Set, of a Counter type, and of the LWW type, representing last-writer-wins registers. The LWW register is a built-in ARDT provided by our library and Counter is another user-defined ARDT defined in Figure 3. Its state definition (Line 9) uses the built-in HashMap with replica IDs as keys and integers as values. Its automatically generated merge function keeps all entries of both maps with the maximum values at each entry (see Subsection 3.2 for the precise definition). A counter of zero is the empty map (Line 12) and the value of a counter is the sum of all values in the map (Line 14). A counter is incremented by increasing the value associated to the current replica ID (Line 15).

The SocialMedia ARDT can be processed like any other data structure. The conventional immutable design prescribes that modifications are represented by operators that return a new state. However, for ARDTs it is sufficient to only return a *delta* – the changed parts of the state – because the rest is managed automatically by applying the merge function. For example, the like operator in Line 6 (that "likes" the post with the given ID) computes the increment of likes (Line 7) and returns a new delta of the SocialMedia state, which contains only the changed ID and defines only the likes component[1] of the social post (Line 8). Using deltas allows developers to clearly express the intended change of an operator. The effective change represented by a delta is identical to that represented by a full state, because deltas returned by operators are merged back into the current state, as we discuss in the next paragraph.

To understand how ARDTs are used, consider their interaction with message dissemi-

[1] The syntax that looks like an assignment in Line 8 is a named parameter, and we assume that this constructor sets all other components to "empty" values (not shown in the example for brevity).

```
17 class SimulatedMessageDissemination[S]:
18   private var network: Set[S] = Set.empty
19   def send(delta: S): Unit = network += delta
20   def recombine(using Lattice[S]): S =
       network.reduce(Lattice[S].merge)
```

Figure 4 Example message dissemination module.

```
21 val smd = new SimulatedMessageDissemination[SocialMedia]
22 val current: SocialMedia = smd.recombine
23 val delta: SocialMedia = current.like(myPost, replicaID)
24 smd.send(delta)
25 val updated: SocialMedia = smd.recombine
```

Figure 5 Using and replicating the social media ARDT.

nation. The API of the message dissemination module consists of a send and a recombine method – Figure 4 shows a simple illustrative implementation, which simulates a network with a set of states S of the replicated type[2] (Line 18). The send method (Line 19) takes a message and simulates sending by adding it to the current set of messages. The recombine method (Line 20) simulates receiving messages by pairwise merging all states in the simulated network into a single result, i.e., a combined state. The merge function of a type S is provided by an instance of type Lattice[S] and the using keyword (Line 20) asks the compiler to provide such an instance if available.

Figure 5 shows how to use and replicate the social media platform with SimulatedMessageDissemination. Line 21 creates an instance of SimulationMessageDissemination named smd. We access the current state of social media (current) using recombine (Line 22). To like a post named myPost, we first apply the like operator on current to compute the delta state. Note that while delta is of type SocialMedia, it only contains that single like. Once we send delta (Line 24), the like is merged into the rest of social media, and we can access the full updated state by calling recombine (Line 25).

3.2 Lattice Composition

By design, the correctness of both the operators and the distributed consistency of ARDTs rely exclusively on their state forming a lattice, i.e., having a correct merge function. Moreover, the library reduces the need to explicitly define (potentially incorrect) merge functions to a minimum. This is due to built-in support for automatically generating merge functions for compound data types (such as maps, tuples, and user-defined product types) given their constituents are ARDTs (i.e., have a lattice instance).

For example, we did not explicitly define a merge function for the SocialPost type – its merge function gets automatically generated because its constituent types, Set, Counter, and LWW are also ARDTs. The generation is recursive: the recursion anchors are ARDTs with custom merge functions, either (a) those whose correctness has been proven in the literature

[2] Note that this API supports multiple ARDTs by composing them into a single type S.

```
29 case class LWW[A](time: Time, value: A)
30 given [A]: Lattice[LWW[A]] = (left, right) =>
31   if right.time < left.time then left else right
```

Figure 6 Last-writer-wins state and lattice.

(e.g., LWW), and (b) user-defined ones – no such merge function is used in the example – for which our library offers property-based correctness testing.

In the following, we elaborate on how lattice instances are defined for different ARDTs starting with the recursion anchors and ending with automatic derivation of instances for compound data types.

3.2.1 Provided and Custom Lattice Instances

We provide ready-to-use lattice instances for existing primitive data types and for common CRDTs. This makes them available as ARDTs and allows them to be used during automatic derivation of merge functions. For example, the code below implements a lattice instance for integers.

```
26 given Lattice[Int] with
27   def merge(left: Int, right: Int): Int = max(left, right)
```

The with keyword states that what follows is the concrete implementation of the Lattice methods, in this case, the implementation of merge as the application of the max function. This definition uses Scala's support for implicit values to seamlessly integrate ARDTs into the rest of the language. The given keyword defines an unnamed instance of Lattice[Int]. Methods can access this instance with the using keyword (as seen in the recombine method from Figure 4), without having the developer explicitly provide the instance when the method is called. The boilerplate of defining a given instance can be reduced by directly assigning a function. For illustration, consider the lattice for sets with an arbitrary element type A (the => defines an anonymous function in Scala).

```
28 given [A]: Lattice[Set[A]] = (left, right) => left union right
```

We have used this approach to define Lattice instances provided by our library for a range of data types such as last-writer-wins registers, multi-version registers, and lists (RGA) based on existing schemes for state-based CRDTs [49, 3].

Custom lattice instances. Application developers may implement custom merge functions by providing corresponding Lattice instances in the same way as we have provided the built-in instances. For illustrating why its important that developers can define custom merge functions, consider the LWW register in Figure 6. Its state associates a unique timestamp and a value. The merge function makes an arbitrary decision – selecting the state with the larger timestamp and ignoring the other one. Such arbitrary decisions encode application semantics of the data type into its distributed consistency.

In general, custom merge functions are more powerful than designing new ARDTs by composition, because they allow to decide which parts of the merged states are no longer necessary in the result – e.g., the LWW register removes the older value. However, allowing such custom decisions about removing parts of the state has the risk of causing inconsistency,

```
32 given [K, V](using Lattice[V]): Lattice[HashMap[K, V]] with
33   def merge(left: HashMap[K, V], right: HashMap[K, V]) =
34     left.merged(right){
35       case ((id, v1), (_, v2)) => (id, Lattice[V].merge(v1, v2))
36     }
```

Figure 7 Map lattice.

e.g., if different replicas remove different parts. To avoid introducing inconsistencies via user-defined merge functions, we can reuse any methods for checking correctness such as static verification, manual proofs, or testing. Currently, our library supports property-based testing (using ScalaCheck) for checking the correctness of merge implementations. This approach requires defining a data generator for each type of state we wish to test, but the test implementation is the same for all types and provided by the library.

3.2.2 Derived Lattice Instances

Besides explicitly defined instances of Lattice, our library is capable of automatically deriving Lattice instances for compound data types including associative maps, pairs, tuples, and user-defined case classes (generally all product types)[3]. Conceptually the generation of derived lattices works by producing lattice instances for generic types (such as a tuple of two generic components) from the lattice instances of the individual component types, depending on the structure of the produced instance. Technically, this is represented as given instances that take other instances as parameters. In the following, we present these composed lattice instances and prove their correctness by showing that the respective merge functions are commutative, associative, and idempotent.

The map lattice. We have already seen a type that uses the map lattice: the counter example. Figure 7 states how any HashMap[K, V] has a lattice instance, if its values V also have a lattice instance. Specifically, the using keyword states that to create the lattice instance for the map we require a Lattice[V] where V is the type of values stored in the map. The merge function (Line 33) for a map delegates to the built-in merged function of HashMap (Line 34). The built-in merged function does not automatically handle the case when a key is assigned to value in both the left and the right map and requires a custom function to handle such conflicts. We implement this function to delegate to the merge function of the value type provided by Lattice[V] (Line 35). Consistency for automatically generated lattices hinges on the correctness of this merge function, which we prove in A.1.

The product lattice. Merging pairs is theoretically enough to represent complex structures by nesting arbitrarily many pairs. However, using nested pairs is syntactically inconvenient and languages provide better abstractions such as tuples of arbitrary size and named data classes (e.g., case classes in Scala). More generally, these are different forms of product types, and we support automatic generation of lattices for any product type whose elements themselves have lattice instances. Consider the exemplary case class MyData in the first line

[3] This is not unlike Haskell's support for deriving instances of type classes for compound data structures.

of the code snippet below, and an explicit definition of the automatically generated lattice instance in the second line.

```
37 case class MyData(a: A, b: B)
38 given Lattice[MyData] = derived[MyData]
```

The derived method generates a lattice instance for any product type S. Its implementation is shown in Figure 15 (in the Appendix).

At a high-level of abstraction, a lattice instance for a product is generated as follows. (i) Acquire lattice instances for each component of the product. (ii) Define the merge function for two instances of the product type (a left and a right one) to (iii) take each component of the left product and merge it with the corresponding component of the right product and (iv) return the results wrapped in a new instance of the product.

We give the full technical details of the implementation in Appendix A.2. We prove correctness of the merge function for arbitrary products.

Proof. Given that K is the set of product indices (this would be the field names of a case class), x_k is a lookup of index k in product x, the syntax $\{k \to v\}_{k \in K}$ constructs a new product of correct type that associates the index k to the value v, each component type at index k has a merge function m_k, and $m(x, y) = \{k \to m_k(x_k, y_k)\}_{k \in K}$ is the implementation of the merge function for the product. We show that m is commutative, associative, and idempotent. All three proofs are calculations that first expand the definition of m, then use the respective property of the component merge functions, and finally use the reverse definition of m (except in the idempotence case which is already done).

$$\begin{aligned} \text{Commutative:} \quad m(x,y) &= \{k \to m_k(x_k, y_k)\}_{k \in K} \\ &= \{k \to m_k(y_k, x_k)\}_{k \in K} = m(x,y) \end{aligned} \tag{1}$$

$$\begin{aligned} \text{Associative:} \quad m(m(x,y),z) &= \{k \to m_k(m_k(x_k, y_k), z_k)\}_{k \in K} \\ &= \{k \to m_k(x_k, m_k(y_k, z_k)\}_{k \in K} = m(x, m(y,z)) \end{aligned} \tag{2}$$

$$\begin{aligned} \text{Idempotent:} \quad m(x,x) &= \{k \to m_k(x_k, x_k)\}_{k \in K} \\ &= \{k \to x_k\}_{k \in K} = x \end{aligned} \tag{3}$$

◀

3.3 Implementation Considerations

Use of Scala. We have implemented the library in the Scala language and make use of Scala in the code examples throughout the paper. Yet, the approach generalizes beyond Scala. In particular, the core design of ARDTs relies only on immutable state and functions, which are available in all general-purpose languages. That said, there are two specific features of Scala that we use. First, we use meta-programming to abstract over the concrete shape of products; while meta-programming is also supported in other languages by tools such as reflection and macros, technical details are language-specific. It would also be possible to provide generation of merge functions for each kind of product type individually. Second, implicit definitions and parameters (the given and using keywords) are used to reduce boilerplate. Specifically, all composed lattices and the recombine method of message dissemination

modules use implicit parameters that would need to be provided explicitly. For example, consider the version of composing lattices below that does not use implicits; we use the tupleLattice to explicitly compose the intLattice and lwwLattice, and then explicitly pass the result to the recombine method.

```
39 val composedLattice: Lattice[(Int, LWW[String])] =
       tupleLattice(intLattice, lwwLattice)
40 recombine(composedLattice)
```

This quickly becomes inconvenient for large composed data structures and is error-prone in untyped languages. Unfortunately, many widely used languages do not have satisfactory replacements, because alternatives such as code generation have their own drawbacks.

Message dissemination. The current implementation of the message dissemination module that comes with our library uses TCP-like connections (including WebRTC and Websockets) – it transfers the full current state once when a connection to a new replica is established, and then sends each delta directly on all established connections. We also have a prototypical dissemination scheme, where replicas periodically combine deltas into a snapshot. When a connection is established, the replicas exchange information about the snapshots they know of and only missing ones are transferred. As already mentioned, the library can be easily extended with other message dissemination algorithms.

3.4 Qualitative Assessment of ARDTs

We recap by highlighting key features of our proposal, their advantages and limitations.

Modular reuse in application design. Any existing data structure can be reused to design new ARDTs as long as we can define a suitable merge function. For example, while our definitions of replicated sets (Subsection 3.2) and maps (Figure 7) are deceptively simple, they do reuse the highly optimized implementations provided by the Scala standard library. Furthermore, developers are not limited to the data types available in the library. For example, there are multiple implementation strategies for sets including sorted trees and hash-based solutions. Developers are free to choose an implementation that best suits their application needs when designing ARDTs and this decision is decoupled from and does not affect consistency management.

Existing off-the-shelf CRDTs can be reused as-is by providing a suitable lattice instance. State-based CRDTs have a correct merge function, which we can and have directly used for this purpose. Operation-based CRDTs can be systematically converted to state-based CRDTs [50], thus they can also be reused. However, such an approach to reuse may result in inefficient implementations with regard to the delta replication strategy.

Unified consistency management. The CALM theorem [22] states that monotonicity is a necessary requirement for consistency. To achieve monotonicity, existing state-based CRDT implementations [50] require that operators return a state that is larger than the original one (with respect to a pre-defined order of all possible states) and operation-based CRDTs require operators to be commutative. In contrast, our approach automatically enforces monotonicity of operators by merging their delta result with the current state. As a result, application developers do not need to reason about consistency when designing operators and the potential for introducing consistency bugs is delimited to a single place in the design – the merge functions.

To illustrate the positive effects of this, consider again the counter ARDT used in the social media application. We only have to ensure that operators implement the intended application semantics, but we are always guaranteed consistent results. That is, the developer may make a mistake and the increment operator does not increment the value of the counter, as it is supposed to do. But it is guaranteed that the operator exhibits the same (erroneous) behavior on all replicas. This is in contrast to classic CRDTs [50] where an incorrect operator may lead to different states on different replicas. In other words, due to unified consistency, distributed correctness becomes equivalent to correctness of a single replica – an incorrect operator will produce incorrect results even without replication. The fact that we get along with local reasoning tremendously simplifies development and testing.

Finally, having correctness relying exclusively on the properties of the merge functions greatly simplifies reasoning about consistency and ensuring it automatically. An indication for this are the proofs for generated merge functions presented in Subsubsection 3.2.2. Not only are they of manageable size, they also can be proven independently of other merge functions and operators, because they do not rely on any global assumptions. Correctness for all composed data types then follows automatically from the individual proofs. This significantly reduces the burden of ensuring that the implementation actually reflects the proven specification. This is also the foundation of why generating merge functions is possible in the first place.

Versatile message dissemination. Local-first applications may run on diverse communication infrastructures, especially when considering various potential intermediaries ranging from a centralized server, to a shared network disk, to passing data along multiple ad-hoc Wi-Fi connections, to storing messages on a USB drive and sending the latter via physical mail. In general, networks where messages are not exchanged directly, but rather stored and forwarded until they are eventually received, are called delay-tolerant networks (DTN) [6]. They are actively developed and researched to enable resilient communication [51, 46, 6, 7], a highly relevant area for local-first software.

Even though message dissemination is not a focus of our contributions our assumptions about message dissemination are explicitly designed to admit communication over DTNs, and we thoroughly explore how ARDTs enable secure communication in such a decentralized setting in Section 4. On the other side of the spectrum, our approach also accommodates optimizations for the specific case, when applications are known to be used with a centralized server as an intermediary, as we explore in Section 6.

Limitations. By the nature of coordination-free consistent replication, ARDTs are limited to express monotonic algorithms. Our design to force monotonicity of operators by using the merge function means that developers require an understanding of how the merge works to design new operators. In this sense, our approach only ensures correctness of the resulting construction regarding consistency, leaving developers with both the freedom and the burden of designing their algorithms. Ultimately, this also means that the consistency management of our approach is limited to applications that do not require consensus. Some features of applications, however, do require consensus and in those cases developers may need to use other approaches that also provide stronger forms of consistency – at a higher cost.

```
44 type EncARDT[S] = Set[AEAD[S, Unit]]
45
46 extension [S] (c: EncARDT[S])
47   def send(data: S, key: Secret, replicaID: ReplicaID): EncARDT[S] =
48     Set(encrypt(data, (), key))
49
50   def recombine(key: Secret)(using Lattice[S]): Option[S] =
51     c.flatMap(aead => decrypt(aead, key)).map(_.data)
52      .reduceOption(Lattice[S].merge)
```

Figure 8 Naive encARDT stores all states.

4 Encrypting ARDTs

We provide special ARDTs, called *encrypting ARDTs* (encARDTs), to protect user data when untrusted intermediaries are used to facilitate communication between trusted replicas. The encARDT module is in the middle of Figure 1, and sits between the other two modules to ensure confidentiality and integrity. For disambiguation, we use the term *message* when referring to the serialized and encrypted states that are stored inside the encARDT, instead of the state of the encARDT itself. On a trusted replica, the send and recombine operators of an encARDT are used to store and retrieve messages. Then, on an untrusted intermediary, the merge function of that same encARDT uses the causality information to manage the messages, in particular, to remove those that are no longer needed.

For encryption, we rely on *authenticated encryption with associated data* (AEAD) to ensure confidentiality of the state and integrity of both the state and the metadata. There are multiple encryption primitives that provide AEAD, but careful considerations is required to ensure they can be used without coordination in our setting. We elaborate on those considerations in Section 5. For the presentation of encARDTs in this section, we assume that there exists an AEAD module with the following interface, where values of type AEAD have the required security guarantees. Intermediaries cannot use these operators as they do not know the secret key.

```
41 object AEAD:
42   def encrypt[S, A](data: S, metadata: A, key: Secret): AEAD[S, A]
43   def decrypt[S, A](aead: AEAD[S, A], key: Secret): Option[(S, A)]
```

In the following, we present three pre-defined encARDTs, each with different implementations of sending and recombining. The first one is naive in the sense that it encrypts and stores sent states indefinitely; the other two implement strategies to avoid redundancy.

4.1 Naive encARDTs

The implementation of the naive encARDT is shown in Figure 8. Line 44 defines the state as a set of AEAD values, and it uses the default merge function for sets. The send operator (Line 47) adds new messages into the encARDT, by using the encrypt method of the AEAD module and returning the encrypted result as a singleton set (which will be merged into the state as usual). The recombine operator (Line 50) reconstructs the plaintext ARDT state. The decrypt method (Line 51) returns an empty value when authentication fails, thus causing that message to be ignored due to flatMap. The decrypted messages are

```
53 type EncARDT[S] = Set[AEAD[S, Version]]
54
55 given [S]: Lattice[EncARDT[S]] with
56   def merge(left: encARDT[S], right: encARDT[S]): encARDT[S] =
57     val combined = left union right
58     combined.filterNot(
59       s => combined.exists(o => s.metadata < o.metadata))
60
61 extension [S] (c: EncARDT[S])
62   def version: Version = c.map(_.metadata)
63                           .reduceOption(Lattice.merge[Version])
64                           .getOrElse(Version.zero)
65
66   def send(data: S, key: Secret, replicaID: ReplicaID): encARDT[S] =
67     val causality = c.version merge c.version.inc(replicaID)
68     Set(encrypt(recombine(key) merge data, causality, key))
```

Figure 9 Subsuming encARDT based on version data.

merged pairwise using the lattice of the plaintext ARDT Lattice[S].

Consistency of the naive encARDT follows from the automatic construction of the merge function. In Appendix A.3, we prove that sending, receiving, and then merging any states of an ARDT in any order has the same result as merging the same states without the use of the encARDT, i.e., the transparency of the naive encARDT.

4.2 Pruning Subsumed States

The naive encARDT stores all messages forever, even if they are no longer relevant. For example, consider the counter ARDT, which stores the maximum value for each replica in its state. Older states contain only outdated information, i.e., the value of the counter prior to incrementing. We say that the old state is *subsumed* by the new state. Formally, a state s' subsumes another state s, if s' contains all updates of s, i.e., $s \sqcup s' = s'$ (where $\sqcup$ is the merge function). In the following, we present an encARDT that makes use of state subsumption to reduce the amount of data that needs to be stored, called *subsuming encARDT*.

The subsuming encARDT attaches logical timestamps [28] in the form of *version vectors* [11] to messages as associated metadata. Version vectors allow intermediaries to compute an order $\leq$ on encrypted states $e(s)$, which implies subsumption, i.e., $e(s) \leq e(s') \implies s \sqcup s' = s'$. Figure 9 shows the implementation of the subsuming encARDT. Its state type is again a set of AEAD values, but this time they contain the version vector as metadata. To implement subsumption, we define an explicit merge function (Line 59). After computing the union of the sets of encrypted states, merge keeps only the states that are not subsumed by another state; formally the kept states are $\{e(s) | \nexists e(s') : e(s) < e(s')\}$. Defining a specific merge function like this, enables a form of customization that is impossible with existing off-the-shelf CRDT designs.

The operators of the subsuming encARDT are used to add the correct metadata to messages. The helper function version (Line 62) produces a version vector that is larger or equal to all version vectors currently stored. The send operator increments the current version (Line 67), implying that the new message subsumes all existing messages. To ensure this is true, the sent state (data in Line 68) is merged with all current values in the encARDT,

```
69 type EncARDT[S] = Set[AEAD[S, (Dot, Set[Dot])]]
70
71 extension [S] (c: EncARDT[S])
72   def send(data: S, key: Secret, replicaID: ReplicaID): encARDT[S] =
73     val contained = c.flatMap(aead => decrypt(aead, key))
74     val subsumed =
75       contained.filter(s => Lattice[S].merge(s.data, data) == data)
76                .flatMap(s => dotsIn(s.metadata)).toSet
77     Set(encrypt(data, (Dots.next(replicaID), subsumed), key))
```

Figure 10 Dotted encARDT based on precise subsumption metadata.

thus producing a state that does contain all others. The recombine operator is the same as for the naive encARDT in Figure 8, hence not shown.

For an intuition to how a subsuming encARDT behaves, consider that a sent message subsumes all messages that are currently stored in the encARDT, and the merge function removes all subsumed messages. Thus, each time a replica sends a message, only that message is stored. However, when multiple untrusted intermediaries synchronize between each other, each may store multiple incomparable messages (generated by different replicas), and merging will keep all of these messages until a trusted replica decrypts and merges them.

In Appendix A.4, we prove that the subsuming encARDT is transparent, i.e., sending and recombining behaves as if we just merge states without encryption, and without removing them based on subsumption metadata. In addition, we also have to prove that the custom merge function is correct (associative, commutative, idempotent).

4.3 Pruning Encrypted Deltas

With the subsuming encARDT, we loose the advantages of delta replication, because it combines all deltas into a single state when sending a message. To address the problem, we instead store per-delta subsumption information in the metadata. Specifically, (a) a globally unique logical timestamp for the message, called a *dot* [39], and (b) the set of dots that the message subsumes. The *dotted encARDT* shown in Figure 10, implements this scheme. The implementation for dotted encARDT is similar to that of subsuming encARDT, except for sending messages, so the other methods are not shown. The send operator computes the set of messages currently contained (Line 73) in the encARDT. For each contained message, it uses the merge function (Line 75) to check if it is subsumed by the new message. Subsumption is transitive, thus the new subsumption info combines all dots in the metadata of all subsumed messages (Line 76). Finally, the message containing only the delta (data) and subsumption info is returned (Line 77).

For an intuition of how dotted encARDTs behave, consider that each trusted replica computes very precise subsumption information for each encrypted message. Thus, trusted replicas never store any duplicated information. When multiple untrusted intermediaries synchronize between each other, they have enough metadata to ensure that subsumed messages are removed during merges. Correctness proofs for the dotted encARDTs are analogous to the subsuming encARDT, because using a more precise notion of subsumption only strengthens the preconditions of the proof.

4.4 Security Properties of encARDTs

Assuming that AEAD protects data confidentiality and authenticity of the contained messages, all encARDTs prevent the following attacks. Intermediaries cannot tamper with the order of data, because recombination is order independent. Replay attacks using duplicated messages also have no effect, since merging is idempotent. Intermediaries can forge new messages using incorrect keys, but these are ignored when decrypting. The only way for intermediaries to interfere is to selectively stop disseminating messages to some or all replicas – this is not worse than the scenario where the intermediary did not exist.

Encrypted communication may still leak information such as the size of messages, and who send which message at what time. In addition, subsuming encARDTs leak the order of messages and dotted encARDT leak precise subsumption information. The order of messages can already be learned by an attacker that can observe the overall network – a common threat model. Generally, leaking metadata is considered as unproblematic when synchronizing rich data such as texts and images, because the contained data is not deducible by the order in which modifications happened. However, leak of metadata can be problematic for certain simple ARDTs, such as the Counter ARDT (Figure 3), which has a single operation (increment) – thus learning the number of messages allows deducing the current value.

However, these issues are not unique to our solution, and countermeasures exist [21, 52]. Moreover, because encARDTs do not require a central entity, it becomes easier to apply countermeasures. For example, one can split messages over multiple intermediaries (such that no single intermediary may learn all metadata), or one can use randomized routing such as TOR [14], because ARDTs are resilient against unreliable message delivery.

The remaining challenge is that AEAD primitives require that each call to the encrypt method uses a globally unique number (nonce). In general, ensuring global uniqueness of something requires coordination, which contradicts our goal to support coordination-free synchronization. We explore the available options in Section 5.

4.5 Discussion

We have presented three different encARDT strategies that cover different points in the design space. A dotted encARDT is as precise as possible, but also leaks the most amount of metadata, while the naive encARDT leaks no additional metadata, but will store unneeded messages. The subsuming encARDT is a middle ground, that only leaks what attackers may learn anyway, while still enabling to remove unneeded messages.

It is noteworthy that besides solving the practical problem of ensuring the integrity and authenticity of replicated state in the presence of untrusted replicas, encARDTs also represent the novel concept of RDT-based implementations of what would classically be seen as a (network) protocol. An encARDT addresses protocol concerns such as transparency of encrypting and decrypting transferred data, which messages are important (and must be retransmitted), and which ones have been superseded by newer messages. Yet, these concerns are separated from concrete issues in physical networks such as message losses, retries and retransmission delays, or splitting of large packages. The platform-specific concerns are handled by the concrete message dissemination module.

As a future expansion on this concept it could be possible to implement other concerns of network protocols as ARDTs. A concrete example is message delivery in causal order, which can be achieved by attaching ordering information to each message [3]. The recombine operator would then merge messages in causal order (temporarily ignoring messages that were received out of order).

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

Figure 11 Overview of supported AEAD modes in various environments.

5 Coordination-free Secure Cryptography

AEAD constructions require a globally unique number (nonce) for encryption. In our setting, the open question is how to guarantee global uniqueness while avoiding coordination. To this end, we are the first to analyze multiple stochastic methods of selecting unique numbers, where the chances for conflicts are within common security standards, and which are suitable for the local-first setting. While there is no single best solution, we investigate and implement multiple choices with specific insights on how many replicas are securely supported and how many operations they can execute without coordination. In summary, the best available options are:

AES-GCM Use a 64 bit random ID per replica and 32 bit replica specific counter as nonces. Supports up to 92,000 replicas, communicating once per second for 132 years.

XChaCha20-Poly1305 Use fully random 192 bit nonces. Supports 2^{32} replicas for communicating once per millisecond for 8900 years.

Concretely, our implementation defaults to XChaCha20-Poly1305, which allows us to hide the use of nonces from the developer altogether. In the following, we discuss the availability of basic constructions and provide the details on how we computed the recommended choices above.

5.1 Availability of Concrete AEAD Constructions

The AEAD constructions we considered are AES-GCM, AES-GCM-SIV, and (X)ChaCha20-Poly1305. Figure 11 shows their availability in the Java Cryptography Architecture[4], Web Cryptography API[5], libsodium[6], and Tink[7]. All libraries support AES-GCM due to its use in the TLS specification [42]. The more modern AEAD construction ChaCha20-Poly1305 was introduced in TLS 1.3 [42] 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]. Also, not standardized is AES-GCM-SIV [20] (implemented only in Tink), which claims resistance to nonce reuse.

5.2 Coordination-free Generation of Nonces for AEAD

To encrypt and authenticate a message, AEAD schemes generally require three inputs: the message, the encryption key, and a *nonce* [44]. A nonce is a **n**umber that must only be used

[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

once together with the same key. If a nonce is used multiple times, 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 [31]. The AES-GCM construction even allows an attacker to forge authenticated messages when a nonce is reused [23]. Nonce misuse has lead to severe real-world attacks, e.g., on TLS [10] and WPA2 [53].

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 [27, 36, 41]. This is not surprising, considering that libraries like the Web Cryptographic API do not even document that nonces should be unique.

Ensuring uniqueness is a classical coordination problem. Thus, we discuss how to select unique nonces without coordination, while staying within generally accepted levels of certainty for the provided confidentiality.

5.2.1 Selecting Nonces by Space Partitioning

A textbook approach to ensure uniqueness of nonces is using a strictly monotonic counter [15]. 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 [42]. Using a single counter for all replicas is not possible without 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 ARDT example (Figure 3) 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 [25]. 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 [29], 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 [15]. 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} [15]. Considering the birthday paradox [47], 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.

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

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 a realistic choice for local-first applications when only AES-GCM is available.

5.2.2 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 [45]. 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 [30], while the documentation of Tink recommends it for "most uses" [18]. Specifically, Tink guarantees that AES-GCM with random nonces can be used for at least $2^{32} \approx 4.3 \times 10^9$ messages, while keeping the attack probability smaller than 2^{-32} [18].

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 outbound message per second for each replica. Moreover, enforcing a limit on the number of replicas also requires coordination.

However, random nonces become practical with the very large nonce sizes supported by XChaCha20-Poly1305 [4]. The use of 192-bit nonces allows $2^{80} (\approx 10^{24})$ 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.

5.2.3 Nonce Misuse-resistant AEAD Schemes

A newer development are *nonce misuse-resistant authenticated encryption schemes*, such as AES-GCM-SIV [20]. These schemes aim to be secure even when a nonce is reused for the same key with a different message. Thus, in theory, it is a good candidate for use with shared, long-lived keys. However, these schemes also do have bounds on the number of messages that can be safely sent [24]. Moreover, they are not yet standardized and fully scrutinized, so we can not give clear recommendations. In addition, as discussed in Figure 11, an implementation of AES-GCM-SIV is not widely available.

6 Implementation and Evaluation

Our main focus so far was the flexibility of ARDTs – they can do everything that off-the-shelf CRDTs offer and more. The question that remains is at what cost this flexibility is

acquired. Specifically, we evaluate our proposal along the following research questions:

- RQ1: Is the performance overhead of ARDTs – including encryption – small enough for general use in local-first applications?
- RQ2: Is the space overhead of ARDTs using intermediaries acceptable?

We use two strategies to explore each of these questions. A concrete case study that makes specific choices about the used ARDTs, and a set of microbenchmarks that explore encARDTs more generally to uncover their behavior in multiple dimensions in particular the overhead caused by encryption and intermediaries.

Unless otherwise noted, we use the following hardware and software setup for evaluation.

CPU 2015 Intel Core i7-6700HQ (laptop CPUs are the most common for local-first software).

OS Arch Linux (Linux 5.16.16).

JRE We use the java runtime OpenJDK 17.0.3.

Microbenchmarks For performance microbenchmarks, we use JMH[9] the standard Java benchmarking tool. The time measurements we conduct have very stable runtime behavior, with a maximum relative error of 3%, thus we do not show error bars.

Libraries AEAD implementations are provided by Tink 1.6.1[10], which uses hardware acceleration for AES variants, but not for XChaCha20-Poly1305. To serialize states, we use jsoniter-scala[11] (the arguably fastest JSON serializer available on the JVM[12]).

6.1 To-do List Case Study

We implement the popular to-do list example as a JavaFX GUI application. The application manages a list of to-dos, and the user may add entries containing arbitrary text, mark to-dos as completed, change their text, or delete to-dos completely. These interactions touch on most of the complexity in the design space of local-first applications. Furthermore, the state of the to-do list – a potentially ordered set of changeable entries – has enough complexity to demonstrate the need for composed data types. Its correctness and consistency properties are: added to-dos remain until deleted, and all users see the same to-dos in the same order.

We experienced no limitations in implementing the to-do list application with ARDTs and encARDTs. The prototype makes heavy use of the ARDT composability. Concretely, the to-do list uses an add-wins last-writer-wins map for its primary state. This is a composition out of a tombstone-free add-wins set [9] and a last-writer-wins register. When two users edit the same to-do entry, then a deterministic decision keeps one of the edits and the other is discarded. Changes to the primary state are normally triggered by the UI library (e.g., a button click handler), but the UI is replaced by our benchmark infrastructure. The handlers for each change are similar to the example in Figure 5. Each handler uses a corresponding operator on the to-do entries (the add-wins-last-writer-wins map) to compute the delta of the new application state. The delta is passed to the send operator of the dotted encARDT, and the operator computes its own delta that is in turn passed to the message dissemination implementation (a custom one for benchmarking the transferred data). In addition, there is a notification API (not discussed in the paper) in the message dissemination module that executes a handler whenever a change happens (caused locally or remotely), which triggers the UI to update and show the new state.

[9] https://openjdk.java.net/projects/code-tools/jmh/
[10] https://developers.google.com/tink
[11] https://github.com/plokhotnyuk/jsoniter-scala
[12] https://plokhotnyuk.github.io/jsoniter-scala/

To answer our research questions, we run a deterministic simulation of the to-do list. Our simulation uses a single intermediary and simulates a total of one million operations that add, modify, and remove to-do entries (see Subsection 6.2 for a discussion of concurrent operations and multiple intermediaries). A million operations correspond to about 11 days of usage, given an interaction every second. We include serialization, encryption, and other application logic in the simulation, but omit physical network, storing the state on disk, or rendering the graphical UI, as their performance is irrelevant (not part of our contributions). The simulation follows a randomly generated trace of adding to-dos and marking them as completed and deletes the oldest 30 to-dos once 50 are completed; it adds and completes individual to-dos, but deletes them in batches to reflect the expected usage of the application, which has a "remove all completed to-dos" button, but no methods of batch insertion or completion. The top plot of Figure 12 shows the runtime behavior of the simulation. The x-axis represents abstract time as the number of executed interactions and the graphs show the respective state of the application, specifically, the number of open and completed to-do entries.

RQ1: Time overhead is presented in the middle plot in Figure 12. It shows the runtime per interaction (measured in batches of 100 interactions). This time includes executing the operator locally, merging it into the local state, serializing then encrypting and sending the delta, merging the encrypted delta into the encARDTs thus computing subsumption, and replicating the encARDTs to the intermediary. The spike in the beginning is due to the warm up of the JVM. Otherwise, the overall runtime is proportional to the size of the current application state, because tasks – such as merging the add-wins-map, computing subsumption for existing deltas, and the application logic – linearly depend on the number of to-do entries. We believe that staying within 3 ms per operation is a reasonable result of our prototype. While further optimizations are certainly possible there is no indication that our core architecture is prohibitive in cost for local-first applications.

RQ2: Space overhead is presented in Figure 12 (bottom). In summary, we observe that the total data stored at the intermediary has a linear relation to the actual size of the application state and grows and shrinks accordingly. We want to specifically point out that we do not observe the state, nor the causality metadata to increase over time, which seems to be a common misconception about CRDT like implementations. While encARDTs require that we store information about subsumed deltas indefinitely (the set of subsumed dots specifically), it is stored as efficient ranges that only grow with the number of replicas, concurrent operations, and current size of the data set, but not with the number of total interactions over time. The data transferred (used bandwidth) between the replica and the intermediary remains mostly constant because transfer time depends on the size of deltas, which are largely unaffected by the size of the application state. The slight increase in bandwidth is because each removal delta includes causality information in the encARDT that grows with the amount of currently non-removed entries. Note that we show the accumulated bandwidth of 100 interactions (i.e., 100 deltas), because the size would otherwise not be visible at the scale of the figure. We show the difference to using a trusted intermediary (i.e., no encARDT) in Appendix A.5, which requires less space due to the impact of encrypted deltas discussed in Subsection 6.2. In conclusion, we consider the size demand and required bandwidth of ARDTs adequate for the local-first scenario.

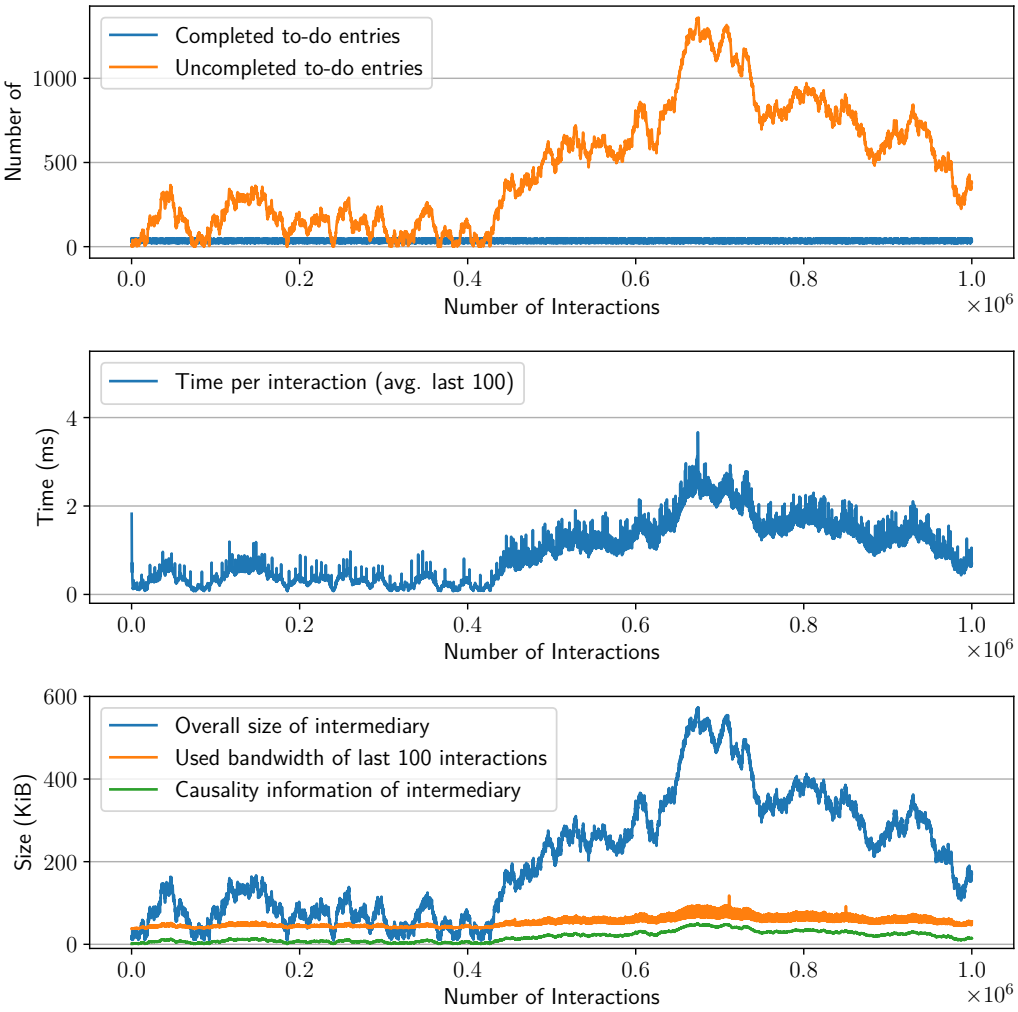

Figure 12 To-do list case study measurement results.

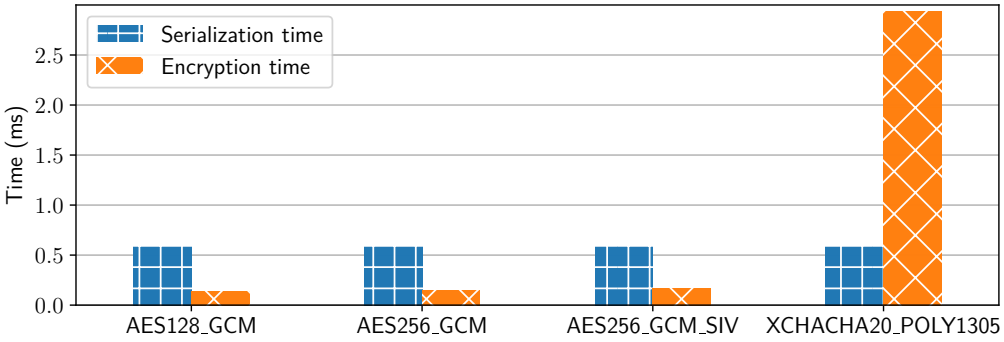

Figure 13 Encryption vs. serialization time for AWLWWMap states of 256 KB.

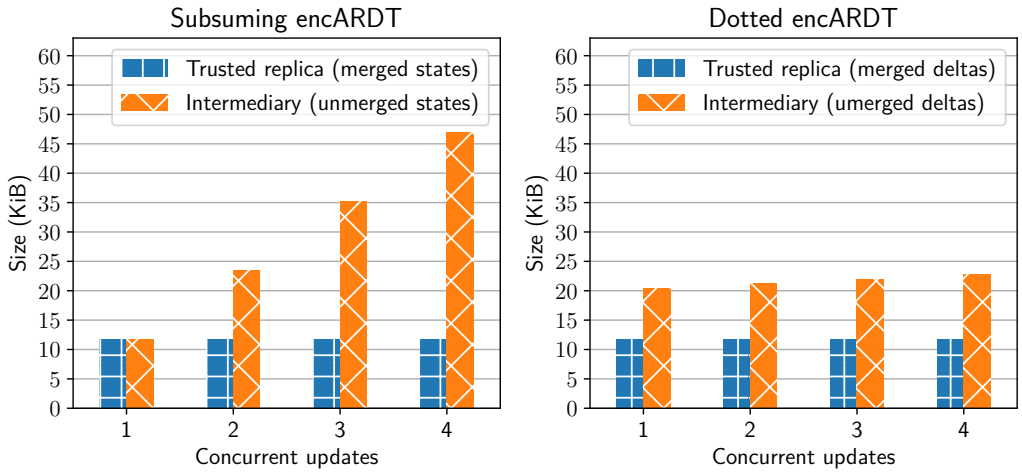

Figure 14 State size increase when storing concurrent encrypted messages.

6.2 Microbenchmarks

We provide microbenchmarks to acquire data points that the case study does not exhibit. Specifically, we investigate the isolated overhead of encryption as well as the effect of concurrent operations (e.g., due to multiple intermediaries) on the required storage size. We use the same add-wins last-writer-wins map (AWLWWMap) for the microbenchmarks that was used for the to-do list case study; however, the results are independent from the concrete choice of ARDT, because for the microbenchmarks only the serialized size of the state matters, which we give explicitly.

RQ1: Time overhead of encARDTs. We measure how long it takes to prepare the serialized bytes (which could then be sent over the network without encryption) compared to the time for encrypting those bytes via an encARDT. The results in Figure 13 show the difference (encryption time vs serialization time) for the different AEAD schemes, and for a payload of 256 KiB (1,000 to-do entries). Hardware accelerated AES has an overhead of a fraction of a millisecond. XChaCha20-Poly1305 does not benefit from hardware acceleration yet still has an overhead of less than 3 ms. However, XChaCha20-Poly1305 is designed to be efficiently implemented in software [4], thus should be considered for systems where no hardware accelerated encryption is available.

To put these numbers into context, consider the relative sizes of operations. An ARDT that serializes in 0.67 ms into a 256 KiB state, requires an additional 0.16 ms to be encrypted. This is for the full state of a to-do list with 1000 to-do entries. So, with our approach, there is an additional latency of 0.16 ms for serialization. After serialization, the data is sent over the network with typical latencies of 0.1~100 ms. After receiving and processing the data on the other replica, displaying the result adds a minimum of 7~33 ms due to typical refresh rates of monitors. To recap: we believe that the time overhead of encARDTs is small compared to all other parts of the synchronization process.

RQ2: Space overhead of encARDTs. Intermediaries cannot merge states that are created concurrently by different replicas, because the metadata does not contain enough information. The concrete overhead depends on which encARDT is used and the number concurrent

operations. Figure 14 shows the space requirement of storing 1 to 4 concurrent updates using a subsuming encARDT (left) and a dotted encARDT (right). The base size of the stored ARDT is an AWLWWMap with 96 (+1 to +4 added) entries requiring about 14 KiB. Any trusted replica (in blue) can always merge any received updates, thus the total stored size does not grow noticeably.

For the intermediary, however, we observe that the size of subsuming encARDT (left subfigure) grows linearly with each concurrent update. We expected this result as each update contains the full state to be stored, and the timestamps of the four updates are incomparable, because each full state differs in exactly one item, thus they do not subsume each other. For the dotted encARDT (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 concurrent update only marginally increases the state size because only the single additional delta is stored. Note that the number of concurrent operations is typically limited by the number intermediaries, because once a replica is connected to an intermediary the next operations of the replica will merge and subsume concurrent operations.

In general, storing only the deltas at intermediaries does not cause large storage increases due to concurrent updates, but has a fixed cost associated. Which strategy is more suitable depends on how reliable connections are, and how many intermediaries are part of the system, because both unreliability and more intermediaries introduce more concurrency. In summary, we believe that one of the presented encARDTs is suitable for most use cases. If other behavior is required, new variants of encARDTs with different subsumption strategies can be used.

7 Related Work

Local-first. Two popular general purpose CRDT implementations that can be integrated into applications are automerge[13] (loosely based on a paper by Kleppmann et al. [25]) and Yjs[14] [37]. They both run in the same process as the application (JavaScript based) and provide the application with an API that allows to update and query a single JSON document (a nested tree structure). The intended way to use the APIs is to have developers convert their application state into the JSON structure, with no further customization of available operations. Both libraries are based on the operation-based variant of CRDTs.

Moreover, REScala [34] integrates arbitrary state-based CRDTs with functional reactive programming and demonstrates that reactive applications are not limited by the restricted nature of operations on CRDTs. In particular, for local-first applications that we are targeting, the intuition is that 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 then leveraged to map user interactions to synchronous updates of related CRDTs to enable coordination-free consistency of entire applications [33, 35]. We believe that the same approach could be used while replacing state-based CRDTs with our ARDTs that offer the same functionality, but with a lower development cost and more flexibility.

[13] `https://github.com/automerge/automerge`
[14] `https://github.com/yjs/yjs`

Message dissemination. The Yjs and automerge libraries both provide a set of utility functions to convert any applied updates to messages that can be sent over the network, and multiple implementations of different message dissemination strategies for their target settings (primarily direct connections to other devices using WebRTC, or a central broadcasting server using Websockets). REScala follows a similar strategy, but uses its reactive programming abstractions to make the current state of the CRDT available as a time-changing value. REScala also provides a default message dissemination using any connections and topologies supported by ScalaLoci [54].

A framework for causally consistent delta CRDTs [39, 2] discusses how to achieve causal consistency for delta CRDTs independently of the underlying network.

Security considerations. 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 insecure systems [16, 38]. The correct usage of cryptographic components is challenging in general [36], with 84 % of Apache projects containing cryptographic misuses [41]. 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 [27]. High-level abstractions with built-in cryptographic features are considered as an effective solution to support developers with writing secure software [1, 19, 32]. However, correct use of encryption is usually considered for pairwise connections (or single writer multiple readers) but not for multiple concurrent writers. We instead opt to make fixed decisions suitable for our domain and package them into the encARDTs, thus reducing the potential for misuse.

Preguiça et al. [40] give an overview on the research and applications around off-the-shelf CRDTs in data centers including the need for future research on security in CRDT systems. They observe that while it is possible to restrict access to a CRDT based service using authentication, the replicas themselves are vulnerable to harmful operations on other replicas. They also discuss end-to-end encryption of states stored on third parties. They state that this would require pushing most of the computation to the edge (i.e., the client that uses the CRDT based service). Two alternatives they suggest are homomorphic encryption and hardware-supported trusted execution (e.g., Intel SGX and ARM TrustZone).

Barbosa et al. [5] implement the approach envisioned above of moving computation to the edge. They use a mixture of custom techniques and solutions from the space of homomorphic encryption to allow clients to interface with a distributed database (AntidoteDB [48]) without fully trusting the provider that hosts them. 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, but 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. Crucially, an adversarial provider could modify data, because operations can not be authenticated. Moreover, all of their cryptographic constructions are specific to individual CRDTs.

8 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 application-specific replicated data types (ARDTs) that provide the same guarantees as CRDTs, but as a modular and extensible library. This approach facilitates the integration of ARDTs into existing programming models and existing network runtimes. Further, our solution provides confidentiality and authenticity by design. Specifically, we presented a family of encrypting ARDTs for different network requirements. Each such encARDT provides a secure layer between the data of an ARDT and the message dissemination over untrusted connections.

Using our encARDTs the application data is authenticated and encrypted, while retaining coordination freedom and preventing the common misuse of cryptographic primitives. Our evaluation shows that we can implement typical local-first applications efficiently using our approach and that any ARDT can be securely disseminated. 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 by including more precise metadata. Moreover, the storage requirement does not increase indefinitely, as ARDTs allow removing data that is no longer needed by the application logic. 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.

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 Russell Brown. Vector clocks revisited. URL: https://riak.com/posts/technical/vector-clocks-revisited/index.html.

12 Tse-Hsun Chen, Weiyi Shang, Jinqiu Yang, Ahmed E. Hassan, Michael W. Godfrey, Mohamed N. Nasser, and Parminder Flora. An empirical study on the practice of maintaining object-relational mapping code in java systems. In Miryung Kim, Romain Robbes, and Christian Bird, editors, *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 165–176. ACM, 2016. doi:10.1145/2901739.2901758.

13 Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In Michael J. Carey and Steven Hand, editors, *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 1. ACM, 2012. doi:10.1145/2391229.2391230.

14 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.

15 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.

16 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.

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

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

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

20 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.

21 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).

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

23 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.

24 Antoine Joux. Nonce misuse-resistant authenticated encryption. URL: https://datatracker.ietf.org/doc/html/rfc8452.

25 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.

26 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.

27 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.

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

29 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.

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

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

32 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.

33 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/.

34 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.

35 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.

36 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.

37 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.

38 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.

39 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.

40 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.

41 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.

42 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.

43 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.

44 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.

45 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.

46 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.

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

48 Marc Shapiro, Annette Bieniusa, Nuno M. Preguiça, Valter Balegas, and Christopher Meiklejohn. Just-right consistency: Reconciling availability and safety. *CoRR*, abs/1801.06340, 2018. URL: http://arxiv.org/abs/1801.06340, arXiv:1801.06340.

49 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.

50 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.

51 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.

52 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.

53 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.

54 Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with scalaloci. *Proc. ACM Program. Lang.*, 2(OOPSLA):129:1–129:30, 2018. doi:10.1145/3276499.

A Appendix

A.1 Map Merge is Correct

Proof. Given that K is the set of all keys (replica ids), x_k is a lookup of key k in map x that returns 0 if the key is not present, $\{k \to v\}_{k \in K}$ constructs a new map that associates the key k to the value v, that m_0 is a correct merge function for the values stored in the map, and $m(x, y) = \{k \to m_0(x_k, y_k)\}_{k \in K}$ is the implementation of the merge function. All three proofs are calculations that first expand the definition of m, then use the respective property of the m_0 function, and finally use the reverse definition of m (except in the idempotence case which is already done).

$$\begin{aligned} \text{Commutative:} \quad m(x, y) &= \{k \to m_0(x_k, y_k)\}_{k \in K} \\ &= \{k \to m_0(y_k, x_k)\}_{k \in K} = m(y, x) \end{aligned} \tag{4}$$

$$\begin{aligned} \text{Associative:} \quad m(m(x, y), z) &= \{k \to m_0(m_0(x_k, y_k), z_k)\}_{k \in K} \\ &= \{k \to m_0(x_k, m_0(y_k, z_k)\}_{k \in K} = m(x, m(y, z)) \end{aligned} \tag{5}$$

$$\begin{aligned} \text{Idempotent:} \quad m(x, x) &= \{k \to m_0(x_k, x_k)\}_{k \in K} \\ &= \{k \to x_k\}_{k \in K} = x \end{aligned} \tag{6}$$

◀

A.2 Derived Product Merge is Correct

We elaborate on the technical details of the implementation of method derived in Figure 15. The method is marked inline to make use of compile-time meta programming, which we use to acquire lattice instances of the individual components of the product, specifically, the summonAll method used later. The using keyword asks the compiler to provide a product mirror (named pm) for the product type S to allow inspection of the components of S.

The summonAll method in Line 80, similar to the using keyword, "summons" instances provided by the given keyword based on their types. Specifically, the type we request are the component types (pm.MirroredElemTypes) of our product mapped to the Lattice type. As an example, the component types of our MyData class returns the type (A, B) and mapping that onto the lattice type results in the type (Lattice[A], Lattice[B]) which is the type for which we "summon" the instances. The result is a tuple of typed lattice instances, but we throw away all type information (Line 81) and rely on the fact that all used products have the same component type at the same structural position.

```
78 inline def derived[S<:Product](using pm:ProductOf[S]):Lattice[S]=
79   val lattices =
80     summonAll[Tuple.Map[pm.MirroredElemTypes, Lattice]]
81       .toIArray.map(_.asInstanceOf[Lattice[Any]])
82   new Lattice[S]:
83     def merge(left: S, right: S): S = pm.fromProduct(
84       new Product {
85         def productElement(i: Int): Any =
86           lattices(i).merge(left.productElement(i),
87                             right.productElement(i))
88       })
```

Figure 15 Automatic derivation of lattice instances for product types.

Having computed lattice instances of the components, we create a new instance of the lattice trait (Line 82). The merge function of that instance (defined in Line 83) uses the pm.fromProduct helper to generically create a new instance of the result product S (e.g., a new instance of our MyData class) in Line 84. The parameter to pm.fromProduct essentially assigns each component at index i (productElement in Line 84) the result of using merge function i to merge the left and right components at position i.

The technical challenges of generating merge functions for arbitrary products are mostly related to practical concerns in the programming language.

A.3 Naive encARDT is Transparent

Proof. We show that for any subset of states $c \subset S$ sending (encrypting) and recombining (decrypting and merging) the set c is equivalent to merging the set c directly. This uses the secret key k, the merge function m_S for states in S, the merge function m_e = union of the encARDT, the encrypt e_k and decrypt d_k function with $d_k(e_k(s)) = s$, the send function $\text{send}_k(s) = \{e_k(s)\}$, and the recombine function $\text{rec}_k(c) = m_S(\{d_k(s)|s \in c\})$. The proof is done by expanding the above definitions (highlighted in blue) when appropriate.

$$
\begin{aligned}
&\text{rec}_k(m_e(\{\text{send}_k(s')|s' \in c\})) \\
&= m_S(\{d_k(s)|s \in m_e(\{\text{send}_k(s')|s' \in c\})\}) && \text{def of rec} \\
&= m_S(\{d_k(s)|s \in m_e(\{\{e_k(s')\}|s' \in c\})\}) && \text{def of send} \\
&= m_S(\{d_k(s)|s \in \{e_k(s')|s' \in c\}\}) && \text{def of } m_e \\
&= m_S(\{d_k(e_k(s))|s \in c\}) && \text{simplify set ops} \\
&= m_S(\{s|s \in c\}) && \text{def of } e_k \text{ and } d_k \\
&= m_S(c)
\end{aligned}
\tag{7}
$$

◀

A.4 Subsuming encARDT is Transparent

Proof. Given a secret key k, a subset of states $c \subset S$, individual states s, x, y, z, a merge function m_S for states in S, associated data for each state a_s where $a_x \leq a_y$ if $m_S(x, y) = y$, the filter function $f(c) = \{x \in c | \nexists y \in c : a_x < a_y\}$, the merge function $m_e(c) = f(\bigcup(c))$ of the encARDT, the encrypt e_k the decrypt d_k function with $d_k(e_k(s)) = s$,

the current encrypted states c_e, the send function $\text{send}_k(c_e, s) = \{e_k(m_S(\text{rec}_k(c_e), s))\}$, and the recombine function $\text{rec}_k(c_e) = m_S(\{d_k(s)|s \in c_e\})$.

It holds that filtering distributes over union $f(p \cup q) = f(f(p) \cup q)$, because all elements of $f(p)$ are larger or equal to all elements in p, so filtering them out first does not change the result of $f(p \cup q)$.

Filter distributes over decryption, i.e., $\{d_k(s)|s \in f(c)\} = f(\{d_k(s)|s \in c\})$, because filtering is defined on associated data which is also available in the encrypted state.

It holds that filtering is subsumed by merging $m_S(f(c)) = m_S(c)$, because for each removed element $r \in p \setminus f(p)$ it is subsumed by one of the remaining elements $q \in f(p)$ thus merging it again makes no difference $m_S(r, p) = p$.

We first show that the merge function of the encARDT m_e is associative, idempotent, and commutative. Note, that up until now, we have proven a slightly stronger version of idempotence that requires less calculation, but we can not do so here, because the filtering function does not provide the stronger guarantee that $f(a) = a$, thus we only have $m(x, x) = f(x)$. Instead of strong idempotence, we prove that $m(m(x, y), y) = m(x, y)$, that is, merging y multiple times still makes no difference, but we must merge at least once.

$$\text{Commutative:} \quad m_e(x, y) = f(x \cup y) = f(y \cup x) = m_e(y, x) \tag{8}$$

$$\begin{aligned}\text{Associative:} \quad m_e(m_e(x, y), z) &= f(f(x \cup y) \cup z) = f(x \cup y \cup z) \\ &= f(x \cup f(y \cup z)) = m_e(x, m_e(y, z))\end{aligned} \tag{9}$$

$$\begin{aligned}\text{Idempotent:} \quad m_e(m_e(x, y), y) &= f(f(x \cup y) \cup y) = f(x \cup y \cup y) \\ &= f(x \cup y) = m_e(x, y)\end{aligned} \tag{10}$$

Finally transparency of the subsuming encARDT, i.e., that receiving a set of send and filtered states is equivalent to merging those states directly. The applied definitions are listed and the changes highlighted in blue.

$$\begin{aligned}
&\text{rec}_k(m_e(\{\text{send}_k(c_e, s')|s' \in c\})) \\
&= m_S(\{d_k(s)|s \in m_e(\{\text{send}_k(c_e, s')|s' \in c\})\}) && \text{def of rec} \\
&= m_S(\{d_k(s)|s \in m_e(\{\{e_k(m_S(\text{rec}_k(c_e), s'))\}|s' \in c\})\}) && \text{def of send} \\
&= m_S(\{d_k(s)|s \in f(\{e_k(m_S(\text{rec}_k(c_e), s'))|s' \in c\})\}) && \text{def of } m_e \\
&= m_S(f(\{d_k(s)|s \in \{e_k(m_S(\text{rec}_k(c_e), s'))|s' \in c\}\})) && \text{filter distributes} \\
&= m_S(f(\{d_k(e_k(m_S(\text{rec}_k(c_e), s')))|s' \in c\})) && \text{simplify set ops} \\
&= m_S(f(\{m_S(\text{rec}_k(c_e), s')|s' \in c\})) && \text{def of } e_k \text{ and } d_k \\
&= m_S(\{m_S(\text{rec}_k(c_e), s')|s' \in c\}) && \text{filter subsumed} \\
&= m_S(\text{rec}_k(c_e), m_S(c)) && \text{merge properties} \\
&= m_S(m_S(\{d_k(e_k(s))|e_k(s) \in c_e\}), m_S(c)) && \text{def of rec} \\
&= m_S(m_S(\{s|e_k(s) \in c_e\}), m_S(c)) && \text{decrypted} \\
&= m_S(\{s|e_k(s) \in c_e\} \cup c) && \text{merge properties}
\end{aligned} \tag{11}$$

◀

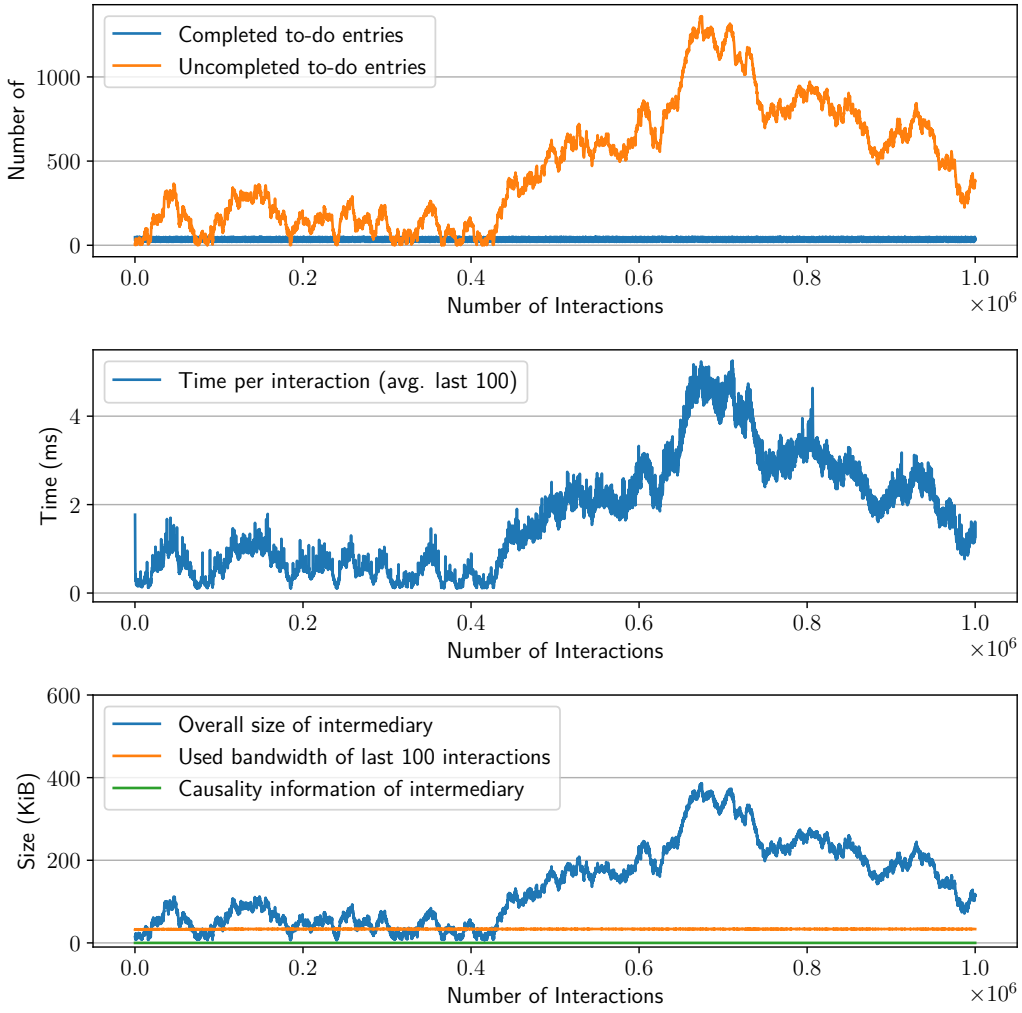

Figure 16 To-do list case study measurement results with trusted intermediary.

A.5 Case Study with Trusted Intermediary

Figure 16 shows the benchmark results for the to-do list case study when we trust the intermediaries and do not use an encARDT. The overall trends are similar, both the time per interaction and the size stored on the intermediary have a linear correlation to the current number of to-do entries. This is because those costs are inherent to the ARDT of the to-do list. There are notable differences. First, the overall runtime when using encARDTs is better (each interaction is faster), because merging the encARDT on the intermediary (i.e., pruning subsumed deltas) is faster than merging the to-do list on the intermediary (i.e., merging the two add-wins-last-writer-wins maps). The encryption overhead is negligible compared to that cost. Second, the overall size of the stored data on the trusted intermediary is smaller, because storing individual encrypted deltas requires more space as discussed in Subsection 6.2. Third, the client does not have to transmit any additional causality information and also does not create subsuming deltas that would reduce the overall size of an encARDT, but lead to larger deltas in some cases. This leads to a nearly constant bandwidth use, with small variations for the random difference between the relative amount of added, completed, and removed to-dos, as well as differences in to-do description lengths.