End-to-End Encrypted Collaborative Documents

Christian Knabenhans*
EPFL

Zayd Maradni*
MPI-SP

Carmela Troncoso
EPFL, MPI-SP

Abstract

Collaborative documents (e.g., Google Docs, Microsoft 365) often contain sensitive information such as personal or financial data. In this work, we extend the protection of E2EE encryption, currently (mostly) restricted to the use case of messaging, to collaborative documents. We elicit and formalize the security and functional requirements of End-to-End Encrypted Collaborative Documents (E2EE-CD). We then put forth a generic framework to realize E2EE-CD, by combining an end-to-end encrypted asynchronous broadcast channel with any edit reconciliation mechanism which ensures globally consistent views of a document. We give formal proofs that directly relate the security of our E2EE-CD solution to the security of the underlying end-to-end encrypted communication channel. We then elicit additional deployment requirements for E2EE-CD for investigative journalists and design SignalCD, an E2EE-CD system built on top of Signal's group messaging protocol tailored for this setting. We analyze the security guarantees of SignalCD, implement a prototype, and empirically show that our solution is efficient enough to permit real-time collaboration.

1 Introduction

End-to-end encryption (E2EE) has become a widespread security paradigm for point-to-point communication, and has contributed to the security and privacy of millions of users using secure messaging, encrypted email, and TLS. However, beyond communicating, people also routinely create, share, and collaboratively edit documents online (using tools such as Apple Notes, Google Docs, Microsoft 365, Overleaf, Canva, etc.). Such documents often contain very sensitive information, such as personal notes, financial data, health data, or intellectual property. Yet, these applications do not offer end-to-end encryption of the documents, and thus users' data remains vulnerable to surveillance and security breaches. In this work, we aim to provide a solid foundation for end-to-end encrypted collaborative documents (E2EE-CD). Our contributions are as follows.

E2EE-CD: Requirements and formal model. The security guarantees for encryption at rest, secure messaging, and transport-layer security are now well understood. How to extend these guarantees to collaborative documents, which combine aspects of both, is however non-trivial. Our first contribution is to identify functional, security, and privacy requirements of E2EE-CD solutions (§2). We also formalize these requirements in an ideal functionality (§3) amenable to provable security.

Generic construction. We show that to realize end-to-end encrypted collaborative documents, it suffices to combine an edit reconciliation mechanism with an *end-to-end encrypted asynchronous broadcast* primitive (§4). We introduce a modular E2EE-CD construction combining these two blocks in §5. The end-to-end encrypted asynchronous broadcast (E2EE-AB) can be constructed from a variety of cryptographic building blocks (encrypted cloud storage, symmetrically-encrypted broadcasts, secure messaging protocols such as Signal or MLS, etc.). We prove that our construction directly lifts any suitable E2EE-AB to a full-fledged E2EE-CD system with essentially the same security guarantees.

SignalCD: E2EE-CD for investigative journalists. To show the potential of our solution, we use the case of investigative journalists, who routinely work with sensitive documents and sources, and are often targeted by surveillance and censorship. In §6, we define journalist-specific additional deployment requirements for E2EE-CD, and we construct SignalCD by instantiating our generic framework for E2EE-CD with the Signal group messaging protocol. Finally, we implement a prototype of SignalCD and show that it is practical for real-world workloads (§7).

2 E2EE Collaborative Documents

We identify the requirements that an E2EE solution for collaborative documents, E2EE-CD, must provide, and discuss

*These authors contributed equally to this work.

how existing approaches to (secure) collaborative documents fare in terms of these requirements. We gather requirements for secure collaborative documents in two complementary ways. First, we lift existing requirements from end-to-end encrypted solutions (secure group messaging and encrypted cloud storage) that provide confidentiality but do not support updates on encrypted data. Second, we consult with investigative journalists who use collaborative documents to work with sensitive information in order to extract functional and security requirements relevant to their work (see §6).

2.1 E2EE-CD Requirements

We define a collaborative document as one that is edited by a set of users. Users can create empty documents, to which they are assigned admin permissions. Documents can contain content (such as text, images, and tables) and overlay content (such as comments, annotations, and suggestions). We consider the following (mutually exclusive) permissions: $\mathsf{read} < \mathsf{write} < \mathsf{admin}$. A user with admin permission to a document can grant admin, write, or read access to this document to other registered users in the system. All users with admin or write permission to a document can edit it. Users with read-only permissions can only access the content, but cannot modify it (R 1, see below). Further, a user with admin permission to a document can permanently archive (R 2) or delete (R 3) the document.

Users can edit a document (R 4) either asynchronously (R 5), or concurrently with other users (R 6). Edits by different users may be conflicting (e.g., one user appends a letter to a word, while another deletes the word); such conflicts are eventually resolved via a reconciliation mechanism (R 7).

In order to allow the asynchronous propagation of edits between users, we assume the existence of a *helper server* (as most collaborative document systems deployed today). We assume that the helper server is generally non-trusted (we discuss our threat model in more detail in §3.1).

Functional requirements.

R 1 (Sharing)**.** *Users should be able to share documents with other users, and be able to choose which edit permissions to give them (read-only, read-write, or admin permission).*

R 2 (Archival)**.** *Admins can archive a document, which prevents anyone from writing to this document in the future, but does not remove the ability to read this document.*

R 3 (Deletion)**.** *Admins can delete a document, which prevents anyone from reading from this document in the future. Naturally, users which possess a local copy of the document will still be able to access their local copy.*

R 4 (Collaboration)**.** *Multiple users can edit the same document. Edits include adding, removing, and modifying elements (text, tables, images, formatting, comment annotations, edit suggestions).*

R 5 (Asynchronous editing)**.** *Users must be able to edit the document even if not all collaborators are online. Any edits are propagated to users that are offline while the editing happens as soon as users go back online.*

R 6 (Concurrent editing)**.** *The system must efficiently handle users concurrently editing the same document, and allow for near-real-time collaboration.*

R 7 (Convergence)**.** *Users starting from the same document and receiving the same set of edits (not necessarily in the same order) converge to the same document (in which edits have been applied, in some implementation-defined order).*

Security requirements.

R 8 (Participant list unforgeability)**.** *Only users with admin permission to a document can add or remove users from the document.*

R 9 (Confidentiality)**.** *Only users with read permission to a document can access the content of the document.*

R 10 (Integrity)**.** *Only users with write permission to a document can modify the content of the document.*

2.2 Reconciliation mechanism

At the heart of any collaborative documents system lays a *reconciliation mechanism* Rec, which initializes empty documents and applies (sometimes conflicting) edits to a document, with well-defined merge semantics (helping to fulfill both the Concurrent editing and Convergence requirements).

A reconciliation mechanism Rec is characterized by a type of document (rich text, spreadsheet, graphic, etc.), edit operations (insertion, deletion, comment, suggested modification, etc.), and metadata (authorship, version control, etc.) it supports. $\mathsf{Rec.new}()$ creates a new empty document. $\mathsf{Rec.apply}(\mathsf{doc}, \mathsf{edits})$ takes in a set of edits edits for the document doc , and uses the set of rules of the reconciliation mechanism to figure out how to incorporate those edits and change the document accordingly. The property we will require in this work is strong convergence, which ensures that the order in which edits are applied does not matter, and that all users will end up with the same document after applying the same set of edits. This is a property that is already achieved by popular collaborative editing software such as Google Docs [14] and Overleaf [4]. Additionally, this property is also achieved by conflict-free replicated data types, such as Automerge [1] [17] and Yjs[2] [21], which is used by collaborative software such as Evernote and JupyterLab.

Definition 1 (Strong convergence [26])**.** *Parties starting from the same document* doc *and receiving the unordered set of*

[1] https://automerge.org/
[2] https://github.com/yjs/yjs

updates $\{\mathsf{edits}_i\}_{i=n}$ *have equivalent document state* doc' *after applying all updates (regardless of order), i.e., for any document* doc*, edits* $\{\mathsf{edits}_i\}_{i=n}$*, and permutation* $\pi : [n] \to [n]$*:*

$$\mathsf{Rec.apply}(\ldots \mathsf{Rec.apply}(\mathsf{doc}, \mathsf{edits}_1), \ldots, \mathsf{edits}_n) = \mathsf{Rec.apply}(\ldots \mathsf{Rec.apply}(\mathsf{doc}, \mathsf{edits}_{\pi(1)}), \ldots, \mathsf{edits}_{\pi(n)}).$$

Since the focus of this work is to securely and modularly construct E2EE-CD systems, not to improve reconciliation mechanisms, we do not describe other properties of reconciliation mechanisms.

2.3 Approaches to Secure Collaboration

Existing (secure) collaborative documents apply reconciliation mechanisms either server-side or client-side.

Server-side reconciliation. In a typical non-encrypted collaborative document system, users send their edits to a central server, which applies the edits to the document and is the source of truth for the document. A natural idea to build secure collaborative documents is to perform the same operations, but on encrypted data. Some systems for secure collaborative documents in this vein have been proposed, using homomorphic encryption [9] or secure multi-party computation [22]. However, these approaches only achieve semi-honest security or rely on the existence of many non-colluding parties. Additionally, because to their use of expensive cryptographic primitives, it is unclear if these approaches can scale to realistic document sizes and editing rates.

Client-side reconciliation. Alternatively, users can send their edits to all other users, and each user applies the reconciliation mechanism to the document themselves. In this paradigm, moving to end-to-end encryption requires end-to-end encrypting the communication between users. This is the approach taken by CryptPad [1] and Capsule [19], which use symmetric encryption primitives to realize this channel. Unfortunately, these systems either do not support fine-grained access control (e.g., read/write/admin permission, which is a crucial feature in some deployment scenarios), or they do not have a formal security treatment. Other systems such as Proton Drive [3] and Apple Notes [16] claim to support end-to-end encrypted collaborative documents, but no public description of the constructions underlying these protocols exist, and no security proofs are available.

Further end-to-end encrypted collaborative services have also been proposed, such as end-to-end encrypted Git [2, 20, 28] and end-to-end encrypted cloud storage [6]. The former require additional manual input to upload and download edits and to resolve conflicting edits, which is a hurdle for achieving near real time collaboration (R 6) in which edits are expected to appear without delay on all users' documents (like in Google Docs or Overleaf). The latter implements updates as whole-file replacements which does not straightforwardly realize real-time collaboration, additionally incurs large communication overhead.

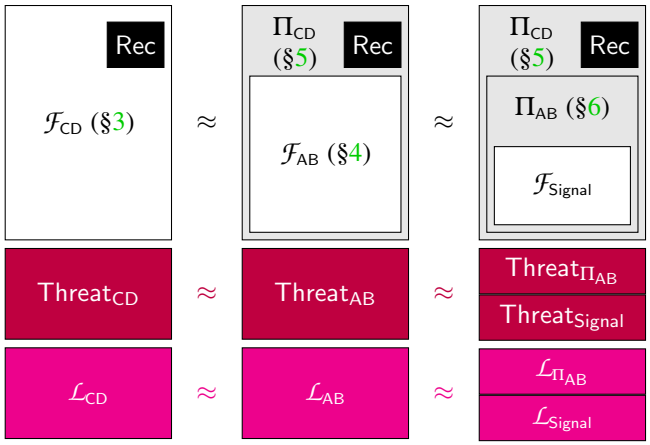

Figure 1: Overview of our generic framework, which relies on a reconciliation mechanism Rec (§2.2).
Left: $\mathcal{F}_{\mathsf{CD}}[\mathsf{Rec}, \mathcal{L}_{\mathsf{CD}}]$ (§3) is the ideal functionality for E2EE-CD, parametrized by a threat model $\mathsf{Threat}_{\mathsf{CD}}$ and leakage $\mathcal{L}_{\mathsf{CD}}$.
Center: $\mathcal{F}_{\mathsf{AB}}[\mathcal{L}_{\mathsf{AB}}]$ (§4) is the ideal functionality for End-to-End Encrypted Asynchronous Broadcast (E2EE-AB), parametrized by a threat model $\mathsf{Threat}_{\mathsf{AB}}$ and leakage $\mathcal{L}_{\mathsf{AB}}$. $\Pi_{\mathsf{CD}}[\mathsf{Rec}, \mathcal{F}_{\mathsf{AB}}]$ (§5) is a generic construction for E2EE-CD, which leverages $\mathcal{F}_{\mathsf{AB}}$.
Right: $\Pi_{\mathsf{AB}}[\mathcal{F}_{\mathsf{Signal}}]$ (§6) is a concrete instantiation of $\mathcal{F}_{\mathsf{AB}}$ based on the Signal group messaging functionality $\mathcal{F}_{\mathsf{Signal}}$.

3 A Formal Model of E2EE-CD

In this section, we introduce a modular, formal model of E2EE-CD, which can be parametrized to accommodate different threat models (§3.1). We define and discuss the parameterization dimensions (§3.2), and we formalize the E2EE-CD requirements from §2 as ideal functionalities (§3.3). We use this formal model to prove the security of SignalCD, an E2EE-CD construction based on the Signal group messaging protocol (§6). We also sketch how CryptPad can be modeled in our framework (§D), as well as an E2EE-CD system built from the Messaging Layer Security (MLS) protocol (§E).

3.1 Which Threat Model for E2EE-CD?

E2EE-CD can be realized using a variety of cryptographic building blocks, which might have different threat models and offer various security guarantees. The choice of building block thus influences the threat model and guarantees of the final E2EE-CD system. While threat models of existing building blocks share some commonalities (the server is generally untrusted, and clients are somewhat trusted), they differ in more subtle ways (e.g., with respect to rollback attacks — see the next section). These differences cannot be captured in a single, rigid formal model. Defining a rigid model would imply that instantiations that satisfy the requirements of E2EE-CD in practice could be artificially ruled out by the model because of purely technical reasons. Conversely, defining a different model and doing an ad-hoc analysis for each instantiation

not only entails extra work, but is prone to errors and hinders comparison between solutions as models might not rely on the same assumptions.

To resolve this tension between generality and complexity, we provide a *parametrized* ideal functionality (parametrized by a leakage and a threat model, see Fig. 1, left), which captures the requirements of E2EE-CD, while still being a meaningful target functionality for many instantiations. This allows us to easily compare different instantiations of E2EE-CD (in terms of threat model and security guarantees) by inspecting the difference between two parameter sets rather than between two full models. We elaborate on this parametrized threat model below. Our threat model in this work only considers a malicious server, and users are assumed to be honest. This represents a use case where users trust other users. In particular, this captures the specific use-case of journalists (6), where journalists trust each other as they would only share their documents with trusted colleagues that they know.

3.2 Model Parameters

Before describing the ideal functionalities, we define and discuss the parameters of our formal model.

Access control. We consider three permission levels. The read permission allows users to only download the document, write allows users to download and edit the document, and admin allows users to download and edit the document, as well as to manage the document permissions which we denote as T. When $\mathsf{PermSet} = \{\bot, \mathsf{admin}\}$, all users associated to the document are assigned admin. Otherwise, users can get any of the other permissions specified in PermSet.

The set PermSet specifies the access control structures supported by the E2EE-CD system. For example, whether the permissions in T are boolean (all users have administrator permissions on the document — i.e., all can read, write, and change access permissions) or finer (users can have different permissions, e.g., admin, read, or write). The functionalities in Fig. 2 need to model this set of permissions accordingly. We use $\bot$ to indicate that a user has no entry in T.

Rollback. In a rollback attack, an adversary storing some publicly available and evolving state reverts this state to a previous version. Such operation might break the security of some protocols relying on this state. Since the adversary is not forging a new state, but rather reverting it to a previously existing (and valid) state, encrypting or authenticating the state is not sufficient to protect against such attacks. Additional countermeasures that need to be built into the protocol usually impose additional constraints or slowdowns on the protocol. For this reason, some E2EE channels assume a slightly weaker adversary that does not have rollback capabilities, while other channels are designed to be secure against rollback attacks. We capture both of these threat models by parametrizing our functionalities with a boolean variable Rollback specifying whether the adversary is allowed to carry out a rollback attack.

Leakage parameters. All functionalities leak information to the adversary. Part of this leakage is *fundamental*, in that it is inherent to the functionality itself and thus independent of how E2EE-CD is implemented. We have made this leakage explicit in every functionality through the function **Leak** (). For example, we write **Leak** $(\texttt{share}, \mathsf{did})$ to indicate that when executing the functionality $\mathcal{F}_{\mathsf{CD.share}}(\mathsf{uid}, \mathsf{did}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$, the adversary learns that the document did was shared.

In a perfect instantiation, this would be the only leakage of the system (i.e., the system would have only fundamental leakage). However, in practice, implementations of E2EE-CD might leak slightly more information than the fundamental leakage, for example due to the use of legacy, but widely-used protocols, or due to the presence or absence of metadata-hiding padding. While it is important that this extra leakage does not void the security requirements of E2EE-CD, some amount of additional leakage might be acceptable in practice (or even desirable for backwards-compatibility or efficiency). To capture such additional flows of information to the adversary, we parametrize our ideal functionalities with leakage functions $\mathcal{L}_{\mathsf{CD}}$. This allows our formalization of E2EE-CD to be adapted to already existing implementations (and ideal functionalities) of E2EE asynchronous broadcast channels.

As an example, for our instantiation of E2EE-CD based on the Signal group messaging protocol (§6), when a user uid shares a document with another user uid', the protocol will leak the index of $\mathsf{uid}, \mathsf{uid}'$ in a table storing the permissions for the document. While this leakage does not prevent the final E2EE-CD system from satisfying the requirements of §2, it is not part of the fundamental leakage of the $\mathcal{F}_{\mathsf{CD.share}}(\mathsf{uid}, \mathsf{did}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$ functionality. By explicitly modeling the fundamental leakage and specifying what the leakage functions are not allowed to leak, we ensure that the requirements for E2EE-CD are properly modeled, while at the same time allowing enough flexibility to choose different channels and implementations based on different requirements, while maintaining the ability to prove security with respect to the security requirements.

3.3 Ideal Functionalities

We define ideal functionalities capturing End-to-End Encrypted Collaborative Documents (E2EE-CD) in Fig. 2, and discuss how they relate to the requirements discussed in §2. We assume that users communicate with the functionalities using authenticated channels. We abstract away specific constructions of authenticated channels in the ideal functionality; in a concrete realization of the functionality, a protocol will instantiate these secure channels (e.g., using transport-layer security or a public key infrastructure) to establish an authenticated channel to communicate with the server. The protocol will also define the leakage from setting up, authenticating, and using those channels.

Setup. The setup functionality initializes the reconciliation

$\mathcal{F}_{\text{CD.setup}}()$

Initialize empty maps Docs, T
Set PermSet, Rollback, THistory

$\mathcal{F}_{\text{CD.share}}(\text{uid}, \text{did}, \text{uid}', \text{perm}, \text{msg})$

Leak $(\texttt{auth}, \text{did}, \mathcal{L}_{\text{CD.auth}}(\text{did}, \text{uid}))$
if $\text{uid} \neq \text{uid}' \vee \text{T}[\text{did}, \text{uid}] \neq \text{admin} \vee \text{perm} \notin \text{PermSet}$:
  Out $\bot \to \mathcal{U}_{\text{uid}}$
  Leak $(\texttt{share}, \text{did}, \mathcal{L}_{\text{CD.share}}(\text{did}, \text{uid}', \text{perm}, \text{msg})), \bot$
$\text{T}[\text{did}, \text{uid}'] := \text{perm}$; $\text{THistory}[\text{did}, \text{uid}'] \mathrel{+}= \text{perm}$
Out $(\text{did}, \text{msg}) \to \mathcal{U}_{\text{uid}'}$
Out $\top \to \mathcal{U}_{\text{uid}}$
Leak $(\texttt{share}, \text{did}, \mathcal{L}_{\text{CD.share}}(\text{did}, \text{uid}, \text{uid}', \text{perm}, \text{msg})), \top$

$\mathcal{F}_{\text{CD.list\_users}}(\text{uid}, \text{did})$

Leak $(\texttt{auth}, \text{did}, \mathcal{L}_{\text{CD.auth}}(\text{did}, \text{uid}))$
if $\text{T}[\text{did}, \text{uid}] \geq \text{read}$:
  Out $\{(\text{uid}', \text{perm}) \mid \text{perm} \in \text{T}[\text{did}, \text{uid}']\} \to \mathcal{U}_{\text{uid}}$
else Out $\bot \to \mathcal{U}_{\text{uid}}$

$\mathcal{F}_{\text{CD.download}}(\text{uid}, \text{did})$

Leak $(\texttt{auth}, \text{did}, \mathcal{L}_{\text{CD.auth}}(\text{did}, \text{uid}))$
$\text{T}'[\text{did}] := \mathcal{F}_{\text{Rollback}}(\text{did})$
if $\text{T}'[\text{did}, \text{uid}] \in \{\text{read}, \text{write}, \text{admin}\}$:
  $\text{doc} := \text{Docs}[\text{did}]$
  Out $\text{doc} \to \mathcal{U}_{\text{uid}}$
  Leak $(\texttt{download}, \text{did}, \text{uid}, \mathcal{L}_{\text{CD.download}}(\text{uid}, \text{did}, \text{doc}))$
else
  Out $\bot \to \mathcal{U}_{\text{uid}}$
  Leak $(\texttt{download}, \text{did}, \text{uid}, \mathcal{L}_{\text{CD.download}}(\text{uid}, \text{did}, \bot))$

$\mathcal{F}_{\text{CD.new}}(\text{uid})$

$\text{did} \leftarrow \text{NextDocumentID}()$
$\text{Docs}[\text{did}] := \text{Rec.new}()$
$\text{T}[\text{did}, \text{uid}] := \text{admin}$
Out $\text{did} \to \mathcal{U}_{\text{uid}}$
Leak $(\texttt{new}, \text{did}, \mathcal{L}_{\text{CD.new}}(\text{uid}, \text{did}))$

$\mathcal{F}_{\text{CD.list}}(\text{uid})$

$\text{dids} := \{(\text{did}, \text{perm}) \mid \text{perm} \in \text{T}[\text{did}, \text{uid}]\}$
Out $\text{dids} \to \mathcal{U}_{\text{uid}}$
Leak $(\texttt{list}, \text{uid}, \text{dids})$

$\mathcal{F}_{\text{CD.edit}}(\text{uid}, \text{did}, \text{edits})$

Leak $(\texttt{auth}, \text{did}, \mathcal{L}_{\text{CD.auth}}(\text{did}, \text{uid}))$
$\text{T}'[\text{did}] := \mathcal{F}_{\text{Rollback}}(\text{did})$
if $\text{T}'[\text{did}, \text{uid}] \in \{\text{write}, \text{admin}\}$:
  $\text{Docs}[\text{did}] := \text{Rec.apply}(\text{Docs}[\text{did}], \text{edits})$
  Leak $(\texttt{list}, \text{did}, \mathcal{L}_{\text{CD.list}}(\text{uid}, \text{did}))$
  for uid' **with** $\text{T}'[\text{did}, \text{uid}'] \geq \text{read}$:
    Out $(\texttt{edit}, \text{uid}, \text{did}, \text{msg}) \to \mathcal{U}_{\text{uid}'}$
  Out $\top \to \mathcal{U}_{\text{uid}}$
  Leak $(\texttt{edit}, \text{did}, \text{uid}, \mathcal{L}_{\text{CD.edit}}(\text{uid}, \text{did}, \text{edits}))$
else
  Out $\bot \to \mathcal{U}_{\text{uid}}$
  Leak $(\texttt{edit}, \text{did}, \text{uid}, \mathcal{L}_{\text{CD.edit}}(\text{uid}, \text{did}, \bot))$

$\mathcal{F}_{\text{Rollback}}(\text{did})$

if $\text{Rollback} = 0$: **return** $\text{T}[\text{did}]$
In $\text{Members} \leftarrow \mathcal{S}$
return $\{(\text{uid}, \text{perm}) \mid (\text{uid}, \text{perm}) \in \text{Members} \wedge$
  $\text{perm} \in \text{THistory}[\text{did}, \text{uid}]\}$

Figure 2: Ideal functionalities for secure collaborative documents. All functionalities are implicitly parametrized by the leakage functions $\mathcal{L}_{\text{CD.new}}$, $\mathcal{L}_{\text{CD.share}}$, $\mathcal{L}_{\text{CD.edit}}$, $\mathcal{L}_{\text{CD.list}}$, $\mathcal{L}_{\text{CD.download}}$, and $\mathcal{L}_{\text{CD.auth}}$. Steps affected by the value of PermSet (respectively, Rollback) are shown in cyan (respectively, red). Users call the functionalities using secure channels authenticated to their uid.

mechanism Rec and the global state of the system, i.e., the document map Docs and the permissions table T. The choice of reconciliation mechanism determines the type of edits that can be carried out on a document, e.g., adding or removing text, images, or annotations. Docs maps a document ID did to a document $\text{Docs}[\text{did}]$ (e.g., a binary file); $\text{T}[\text{did}, \text{uid}]$ stores the permission of user uid for the document did. A user uid has access to the document did with permission perm iff $\text{T}[\text{did}, \text{uid}] = \text{perm}$. The setup functionality also defines two parameters: PermSet, and Rollback (see §3.2).

New Document. Given a user uid, the functionality $\mathcal{F}_{\text{CD.new}}(\text{uid})$ creates an empty document $\text{Docs}[\text{did}]$ with document id did and assigns the user uid the admin permission in $\text{T}[\text{did}]$.

Sharing. $\mathcal{F}_{\text{CD.share}}(\text{uid}, \text{did}, \text{uid}', \text{perm}, \text{msg})$ models the Sharing requirement by allowing users with admin permission to the document did to grant the permission perm to the document to another user uid'. This functionality also sends the new user uid' a sharing message msg to notify them that the document has been shared with them, and provide them with relevant information such as the document ID did. When called with $\text{perm} = \bot$, $\mathcal{F}_{\text{CD.share}}(\text{uid}, \text{did}, \text{uid}', \text{perm}, \text{msg})$ revokes any permissions of uid' for the document, which we use to model user removal.

Edit. $\mathcal{F}_{\text{CD.edit}}(\text{uid}, \text{did}, \text{edits})$ models the Collaboration requirement by allowing users uid to edit documents to which they have write or admin access. The functionality passes the edits edits to the reconciliation mechanism Rec, which applies the edits to the document $\text{Docs}[\text{did}]$. After implementing the edits, the functionality notifies all other users in $\text{T}[\text{did}]$,

optionally including a message msg that includes relevant information about the edits. The reconciliation mechanism Rec ensures that even if $\mathcal{F}_{\mathsf{CD.edit}}(\mathsf{uid}, \mathsf{did}, \mathsf{edits})$ is called by multiple users, concurrent edits will converge (as required for Concurrent editing and Convergence). Rec can also output editing metadata, which is sent to other users as part of msg.

Download. The functionality $\mathcal{F}_{\mathsf{CD.download}}(\mathsf{uid}, \mathsf{did})$ enables a user uid to retrieve all edits made to document did, provided that $\mathsf{T}[\mathsf{did}, \mathsf{uid}]$ is set to at least read. This functionality realizes the Asynchronous editing requirement, as doc contains all the edits users have made since the user last called $\mathcal{F}_{\mathsf{CD.download}}(\mathsf{uid}, \mathsf{did})$ and so they are propagated to the user uid, including those made while the user was offline.

Listing users and documents. We additionally define two functionalities, $\mathcal{F}_{\mathsf{CD.list\_users}}(\mathsf{uid}, \mathsf{did})$ and $\mathcal{F}_{\mathsf{CD.list}}(\mathsf{uid})$, even though they are not part of the requirements and in some cases can be done on the user side with just the previous functionalities. We chose to do this because being able to list which documents and collaborators a user is working with is a functionality that any user of a collaborative document system would expect to have. Thus, to make our system more usable and more in line with the usability of existing collaborative documents, we explicitly model these functionalities part of $\mathcal{F}_{\mathsf{CD}}$. $\mathcal{F}_{\mathsf{CD.list\_users}}(\mathsf{uid}, \mathsf{did})$ enables all readers of a document did to retrieve a list of other users with permissions for $\mathsf{T}[\mathsf{did}]$. $\mathcal{F}_{\mathsf{CD.list}}(\mathsf{uid})$ enables a user to retrieve a list of all the documents they have access to, along with their permission to these documents.

Other requirements. The above functionalities also model the Archival and Deletion requirements (as well as other advanced document lifecycle management functionalities such as soft-archival/deletion and auto-archival/deletion). For the sake of conciseness, we discuss these in §B.1.

Participant list unforgeability. Only admins can update the participant list, as $\mathcal{F}_{\mathsf{CD.share}}(\mathsf{uid}, \mathsf{did}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$ is the only functionality that allows editing of $\mathsf{T}[\mathsf{did}]$, and the adversary does not have admin permission in this permission list. If this requirement is fulfilled, then the functionality also guarantees that no users can be maliciously granted read or stronger other than through Rollback attacks.

Confidentiality. Confidentiality is captured by the fact that the adversary does not have any permissions on documents it has not created itself, together with our use of secure authenticated channels for the outputs of all functionalities. However, unless special protections are provided (e.g., padding), the size of doc and the size of edits will be leaked as they are sent; we discuss extensions to mitigate this leakage in §B.4.

Integrity. To achieve Integrity, the adversary must not be able to change $\mathsf{Docs}[\mathsf{did}]$. $\mathcal{F}_{\mathsf{CD.download}}(\mathsf{uid}, \mathsf{did})$ and $\mathcal{F}_{\mathsf{CD.edit}}(\mathsf{uid}, \mathsf{did}, \mathsf{edits})$ ensure this is the case: the former ensures that the document did cannot be changed on its way to the user; while the access control in $\mathcal{F}_{\mathsf{CD.edit}}(\mathsf{uid}, \mathsf{did}, \mathsf{edits})$ ensures that only users in $\mathsf{T}[\mathsf{did}]$ with write can modify the document. If the requirement of Participant list unforgeability is achieved, no users can be maliciously granted write or stronger, other than through Rollback attacks.

Modeling rollback attacks. In a rollback attack, the adversary reverts an object to one of its previous states. If the adversary can successfully carry out rollback attacks ($\mathsf{Rollback} = 1$), it can revert the entries of T individually, and in particular it may revert a permission change for a given user. We model this capability by tracking the evolution of T through time using the table THistory.

4 E2EE Asynchronous Broadcast

We formally define an *End-to-End Encrypted Asynchronous Broadcast* (E2EE-AB) channel as an ideal functionality $\mathcal{F}_{\mathsf{AB}}$ (Fig. 3), which we will later use to securely send and receive edits between users. $\mathcal{F}_{\mathsf{AB}}$ models adding, managing, and checking user's permissions in addition to sending and receiving messages. Any E2EE channel that securely realizes $\mathcal{F}_{\mathsf{AB}}$ can be plugged into a construction of E2EE-CD system and have the security requirements of the system be realized without the need to carry out an end-to-end analysis of the entire system.

For the same reasons as discussed in §3.1, our definition of $\mathcal{F}_{\mathsf{AB}}$ (§4) is parametrized by a threat model and a set of leakage functions. In general, the threat model of $\mathcal{F}_{\mathsf{AB}}$ includes attacks that can be carried out on the access control part of the protocol, where the adversary attempts to change users' permissions in the broadcast channel, potentially adding itself or a user it controls. This is also the component targeted by a Rollback attack. The adversary can also target the messages being sent between the users, in an attempt to tamper with existing messages, inject its own messages, or gain access to the plaintexts being sent.

In this subsection, we define the interfaces and functionalities necessary to realize an E2EE asynchronous broadcast channel and relate them to the ideal functionalities in Fig. 2. We summarize these functionalities in Fig. 3, where all **Out** $\rightarrow$ channels are modeled to be secure channels.

Setup. $\mathcal{F}_{\mathsf{AB.setup}}()$ initializes an instance of E2EE asynchronous broadcast channel. Similarly to $\mathcal{F}_{\mathsf{CD.setup}}()$, $\mathcal{F}_{\mathsf{AB.setup}}()$ it initializes a permission table T. It also initializes empty message queues $\mathsf{Queue}[\mathsf{bid}, \mathsf{uid}]$ that stores all messages sent to user uid in the broadcast group bid, but have not been delivered yet. Since the broadcast channel also handles access control, the setup functionality also determines the flags PermSet, Rollback, and initializes the table THistory.

New broadcast. The functionality $\mathcal{F}_{\mathsf{AB.new}}(\mathsf{uid})$ creates a new broadcast channel with broadcast id bid and assigns the user uid the admin permission in $\mathsf{T}[\mathsf{bid}]$.

Add member. $\mathcal{F}_{\mathsf{AB.add}}(\mathsf{uid}, \mathsf{bid}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$ allows users with admin permission to give access to new users to the broadcast channel. Concretely, a user uid with permission

$\mathcal{F}_{\mathsf{AB.setup}}()$

Initialize an empty map T
Set PermSet, Rollback, THistory

$\mathcal{F}_{\mathsf{AB.new}}(\mathsf{uid})$

$\mathsf{bid} \leftarrow \mathsf{next\_bid}()$
$\mathsf{T}[\mathsf{bid}, \mathsf{uid}] := \mathsf{admin}$
Leak $(\mathtt{new}, \mathsf{bid}, \mathcal{L}_{\mathsf{AB.new}}(\mathsf{uid}, \mathsf{bid}))$
Out $\mathsf{bid} \rightarrow \mathcal{U}_{\mathsf{uid}}$

$\mathcal{F}_{\mathsf{AB.add}}(\mathsf{uid}, \mathsf{bid}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$

Leak $(\mathtt{auth}, \mathsf{bid}, \mathcal{L}_{\mathsf{AB.auth}}(\mathsf{bid}, \mathsf{uid}))$
if $\mathsf{T}[\mathsf{bid}, \mathsf{uid}] \neq \mathsf{admin} \vee \mathsf{perm} \notin \mathsf{PermSet}$: **return** $\perp$
$\mathsf{T}[\mathsf{bid}, \mathsf{uid}'] := \mathsf{perm}$; $\mathsf{THistory}[\mathsf{bid}, \mathsf{uid}'] \mathrel{+}= \mathsf{perm}$
Leak $(\mathtt{add}, \mathsf{bid}, \mathcal{L}_{\mathsf{AB.add}}(\mathsf{bid}, \mathsf{uid}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg}))$
Out $(\mathtt{out}, \mathsf{did}, \mathsf{msg}) \rightarrow \mathcal{U}_{\mathsf{uid}'}$

$\mathcal{F}_{\mathsf{AB.send}}(\mathsf{uid}, \mathsf{bid}, \mathsf{msg})$

Leak $(\mathtt{auth}, \mathsf{bid}, \mathcal{L}_{\mathsf{AB.auth}}(\mathsf{bid}, \mathsf{uid}))$
if $\mathsf{T}[\mathsf{bid}, \mathsf{uid}] \notin \{\mathsf{write}, \mathsf{admin}\}$ **return** $\perp$
$\mathsf{T}'[\mathsf{bid}] := \mathcal{F}_{\mathsf{Rollback}}(\mathsf{bid})$
Leak $(\mathtt{list}, \mathsf{bid}, \mathcal{L}_{\mathsf{AB.list}}(\mathsf{uid}, \mathsf{bid}))$
for uid' **in** $\mathsf{T}'[\mathsf{bid}]$ **with** $\mathsf{T}'[\mathsf{bid}, \mathsf{uid}'] \neq \perp$:
  enqueue $(\mathsf{uid}, \mathsf{msg})$ **in** $\mathsf{Queue}[\mathsf{bid}, \mathsf{uid}']$
  Leak $(\mathtt{send}, \mathsf{uid}, \mathcal{L}_{\mathsf{AB.send}}(\mathsf{uid}, \mathsf{uid}', \mathsf{msg}))$

$\mathcal{F}_{\mathsf{AB.rcv}}(\mathsf{uid}, \mathsf{bid})$

Leak $(\mathtt{auth}, \mathsf{bid}, \mathcal{L}_{\mathsf{AB.auth}}(\mathsf{bid}, \mathsf{uid}))$
if $\mathsf{T}[\mathsf{bid}, \mathsf{uid}] \notin \{\mathsf{read}, \mathsf{write}, \mathsf{admin}\}$ **return** $\perp$
$\mathsf{T}'[\mathsf{bid}] := \mathcal{F}_{\mathsf{Rollback}}(\mathsf{bid})$
Leak $(\mathtt{list}, \mathsf{bid}, \mathcal{L}_{\mathsf{AB.list}}(\mathsf{uid}, \mathsf{bid}))$
for uid' **in** $\mathsf{T}'[\mathsf{bid}]$
  $\mathsf{msg} := \mathsf{Queue}[\mathsf{bid}, \mathsf{uid}].\mathbf{pop}(\mathsf{uid}')$
  Leak $(\mathtt{rcv}, \mathsf{uid}, \mathsf{uid}', \mathcal{L}_{\mathsf{AB.rcv}}(\mathsf{uid}, \mathsf{uid}', \mathsf{msg}))$
  Out $(\mathsf{uid}', \mathsf{msg}) \rightarrow \mathcal{U}_{\mathsf{uid}}$

$\mathcal{F}_{\mathsf{AB.list}}(\mathsf{uid})$

return $\{(\mathsf{bid}, \mathsf{T}[\mathsf{bid}, \mathsf{uid}]) \mid \mathsf{T}[\mathsf{bid}, \mathsf{uid}] \neq \perp\}$

Figure 3: Ideal functionalities for an end-to-end encrypted asynchronous broadcast channel. All functionalities are implicitly parametrized by the leakage functions $\mathcal{L}_{\mathsf{AB.add}}$, $\mathcal{L}_{\mathsf{AB.list}}$, $\mathcal{L}_{\mathsf{AB.new}}$, $\mathcal{L}_{\mathsf{AB.auth}}$, $\mathcal{L}_{\mathsf{AB.send}}$, and $\mathcal{L}_{\mathsf{AB.rcv}}$. Steps affected by the value of PermSet (respectively, Rollback) are shown in cyan (respectively, red). Users call the functionalities using secure channels authenticated to their uid.

admin in $\mathsf{T}[\mathsf{bid}, \mathsf{uid}]$ can create a new entry in $\mathsf{T}[\mathsf{bid}, \mathsf{uid}']$ = perm that grants uid' access to the broadcast bid with permission $\mathsf{perm} \in \{\mathsf{admin}, \mathsf{read}, \mathsf{write}\}$. This functionality also sends the new user uid' a message msg to notify them that they were given access to the broadcast and provide them with relevant information, such as the broadcast ID bid. Fine-grained permissions have the same meaning and follow the same hierarchy as in $\mathcal{F}_{\mathsf{CD.share}}(\mathsf{uid}, \mathsf{did}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$. Similarly, we model user removal from the AB by calling $\mathcal{F}_{\mathsf{CD.share}}(\mathsf{uid}, \mathsf{did}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$ with $\mathsf{perm} = \perp$ to revoke all permissions from that user.

Send. The functionality $\mathcal{F}_{\mathsf{AB.send}}(\mathsf{uid}, \mathsf{bid}, \mathsf{msg})$ allows a user uid whose entry $\mathsf{T}[\mathsf{bid}, \mathsf{uid}]$ contains write or admin to send messages msg to the broadcast bid. If the user has permission to send messages to the broadcast, the functionality retrieves the list of users uid' in the broadcast $\mathsf{T}[\mathsf{bid}]$ and appends the tuple $(\mathsf{uid}, \mathsf{msg})$ for each of those users queue $\mathsf{Queue}[\mathsf{bid}, \mathsf{uid}']$.

Receive. The functionality $\mathcal{F}_{\mathsf{AB.rcv}}(\mathsf{uid}, \mathsf{bid})$ enables a user uid to retrieve all messages sent to the broadcast channel bid, provided that $\mathsf{T}[\mathsf{bid}, \mathsf{uid}]$ is set to at least read. The $\mathcal{F}_{\mathsf{AB.rcv}}(\mathsf{uid}, \mathsf{bid})$ functionality retrieves the list of users uid' that have access to the broadcast channel bid with at least write permission in $\mathsf{T}[\mathsf{bid}, \mathsf{uid}']$. Then for each of those users uid', it outputs the first message msg (or $\perp$) stored in the user's uid message queue $\mathsf{Queue}[\mathsf{bid}, \mathsf{uid}]$ sent by uid'.

Listing broadcasts. $\mathcal{F}_{\mathsf{AB.list}}(\mathsf{uid})$ allows a user to retrieve the list of all the broadcast channels they have access to.

5 A Generic Construction of E2EE-CD

E2EE-CD can be realized using a variety of cryptographic building blocks (using symmetric versus public-key encryption, different authentication mechanisms, post-quantum security, post-compromise and forward security, etc.). As there are numerous use cases of E2EE-CD with heterogeneous deployment settings, we believe that there is no single best, one-size-fits-all E2EE-CD construction. To accommodate this diversity, we propose a *generic* construction of E2EE-CD (see Fig. 1, center). Our generic construction separates the task of building E2EE-CD into two orthogonal components: a reconciliation mechanism Rec, and an end-to-end encrypted asynchronous broadcast channel (E2EE-AB) $\mathcal{F}_{\mathsf{AB}}$. This allows us to separate the concerns of reconciliation semantics and user-facing features from security and privacy considerations. In this section, we present a generic protocol $\Pi_{\mathsf{CD}}[\mathsf{Rec}, \mathcal{F}_{\mathsf{AB}}]$ (in Fig. 4) which realizes the functionalities of E2EE-CD (Fig. 2) using an E2EE asynchronous broadcast channel $\mathcal{F}_{\mathsf{AB}}$ (Fig. 3) and a convergent reconciliation mechanism Rec (§2.2).

5.1 Construction

$\Pi_{\mathsf{CD}}[\mathsf{Rec}, \mathcal{F}_{\mathsf{AB}}]$ (Fig. 4) securely realizes the functionalities of E2EE-CD shown in Fig. 2 by using the functionalities of an E2EE asynchronous broadcast $\mathcal{F}_{\mathsf{AB}}$ (Fig. 3) and a reconciliation mechanism Rec. The construction implements each instance of E2EE-CD form Fig. 2 as an instance of an E2EE asynchronous broadcast Fig. 3, which is used to send edits to

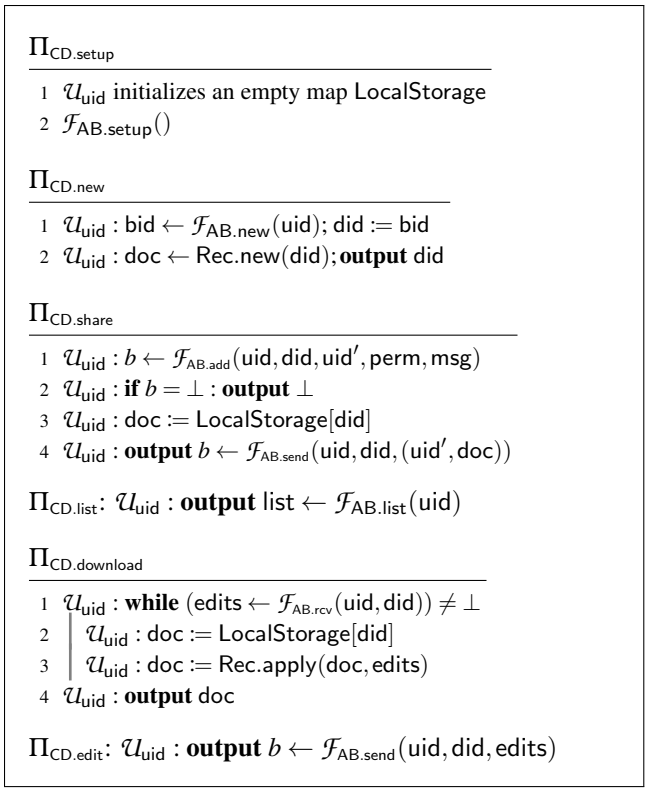

Figure 4: Protocols $\Pi_{\mathsf{CD}}[\mathsf{Rec}, \mathcal{F}_{\mathsf{AB}}]$ for E2EE-CD.

the other users, and by using Rec to apply those edits. However, in contrast to the centralized document editing in Fig. 2, the construction implements document editing using a distributed approach where each user locally applies the edits. This is because realizing a centralized E2EE system would require a server to operate on encrypted data as discussed in §2. Due to the strong convergence property of Rec , the document content each user has is the same whether the users send all their edits to a central server that applies them to the document or each user directly receives those edits from other users and locally applies them to the document. The AB functionality already offers access control (in the form of sending and receiving permissions for each user in the broadcast), which we leverage as access control for the document by controlling who is allowed to send and receive updates to the document. The E2EE-CD protocol will also inherit the efficiency and usability properties of the AB with regards to access control and edit propagation.

Setup. The setup protocol $\Pi_{\mathsf{CD.setup}}$ in Fig. 4 initializes LocalStorage[did] for each user instead of having one centralized version Docs[did] like in Fig. 2. Since the construction leverages the access control offered by the AB, it is sufficient to additionally call $\mathcal{F}_{\mathsf{AB.setup}}$ (which also internally sets the values of PermSet, Rollback).

New document. $\Pi_{\mathsf{CD.new}}$ realizes $\mathcal{F}_{\mathsf{CD.new}}(\mathsf{uid})$. A user uid creates a new asynchronous broadcast bid using $\mathcal{F}_{\mathsf{AB.new}}(\mathsf{uid})$ and the protocol sets the new document id did to be the broadcast id bid and creates a new empty document LocalStorage[did] using Rec.new

Sharing. $\Pi_{\mathsf{CD.share}}$ realizes $\mathcal{F}_{\mathsf{CD.share}}(\mathsf{uid}, \mathsf{did}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$ using the add functionality $\mathcal{F}_{\mathsf{AB.add}}(\mathsf{uid}, \mathsf{bid}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$ on the broadcast channel did when a user uid wants to grant a new user uid' access to the document did with permission perm. The functionality $\mathcal{F}_{\mathsf{AB.add}}(\mathsf{uid}, \mathsf{bid}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$ then checks the permission of uid and adds the user to the broadcast accordingly. The user also sends the current document state to the group. This is necessary, since some AB channels might prevent new users from accessing messages send before they were past of the broadcast (this is the case, for example, or the Signal group messaging protocol).

Edit. Construction realizes $\mathcal{F}_{\mathsf{CD.edit}}(\mathsf{uid}, \mathsf{did}, \mathsf{edits})$ by calling $\mathcal{F}_{\mathsf{AB.send}}(\mathsf{uid}, \mathsf{bid}, \mathsf{msg})$ when a user uid wants to perform edits on the document did. The functionality $\mathcal{F}_{\mathsf{AB.send}}(\mathsf{uid}, \mathsf{bid}, \mathsf{msg})$ then checks the user's permission and broadcasts their edits to the other users accordingly.

Download. $\Pi_{\mathsf{CD.download}}$ realizes $\mathcal{F}_{\mathsf{CD.download}}(\mathsf{uid}, \mathsf{did})$ by allowing user uid to keep calling $\mathcal{F}_{\mathsf{AB.rcv}}(\mathsf{uid}, \mathsf{bid})$ on the document did until it receives all the edits made since the last call to the protocol, provided that uid permission is set to at least read. The user then locally passes the edits they received to Rec, which applies those edits locally on the document LocalStorage[did]. The Rec mechanism ensures that edits received concurrently or out-of-order will converge, realizing the Concurrent editing and Convergence requirements.

Listing documents. Construction realizes $\mathcal{F}_{\mathsf{CD.list}}(\mathsf{uid})$ by calling $\mathcal{F}_{\mathsf{AB.list}}(\mathsf{uid})$, which retrieve a list of all did the user uid has access to and their permission in that document.

5.2 Correctness and Security

To prove that the construction in Fig. 4 achieves the security requirements of §2, we prove that the protocol $\Pi_{\mathsf{CD}}[\mathsf{Rec}, \mathcal{F}_{\mathsf{AB}}]$ securely realizes $\mathcal{F}_{\mathsf{CD}}$. We prove Lemma 1 in §A.1.

Lemma 1. *If* Rec *is a convergent reconciliation protocol (Def. 1), then* $\Pi_{\mathsf{CD}}[\mathsf{Rec}, \mathcal{F}_{\mathsf{AB}}]$ *securely realizes* $\mathcal{F}_{\mathsf{CD}}[\mathsf{Rec}, \mathsf{PermSet}, \mathsf{Rollback}, \mathcal{L}_{\mathsf{CD}}]$ *in the* $\mathcal{F}_{\mathsf{AB}}[\mathsf{PermSet}, \mathsf{Rollback}, \mathcal{L}_{\mathsf{AB}}]$*-hybrid model, with the leakage functions as shown in Table 1, access control structure* PermSet *and rollback flag* Rollback.

6 SignalCD: E2EE-CD for journalists

In this section, we introduceSignalCD. SignalCD is an E2EE-CD solution built on top of the Signal group messaging protcol and platform that fulfills deployment requirements specific to investigative journalists.

6.1 Requirements

An E2EE-CD for journalists must satisfy the following requirements (see §5 for more details).

Functional requirements. The channel must support Sharing, so journalists can add their colleagues as collaborators, as well as Asynchronous editing so groups of journalists can work on their documents being online simultaneously while being able to retrieve any edits made while they were offline.

Security requirements. The channel needs to provide Confidentiality and Integrity for all the messages sent so the server can never learn or edit the content they are working on. The channel also must provide Participant list unforgeability so that only the (admin) journalists can add new collaborators.

Deployment requirements. In conversation with investigative journalists, we identify the following requirements necessary for them to be able to deploy any E2EE-CD solution (in addition to those we describe in §2):

R 11 (Minimal infrastructure)**.** *Journalists cannot afford to set up an entire infrastructure to host and operate E2EE-CD. Thus, the E2EE-CD's asynchronous broadcast channel needs to exist already.*

R 12 (No new trust assumptions)**.** *Journalists would like to avoid new trust assumptions beyond the ones that they already have with existing platforms they use.*

R 13 (Simple key management)**.** *The channel must have a simple password or key management, and ideally use credentials that are already in use. The issuance/revocation/key discovery processes must be simple, and ideally the same as those for other parts of the journalists' workflow.*

Motivated by these requirements, we build SignalCD, an end-to-end encrypted collaborative document system which uses the Signal group messaging protocol as its anonymous broadcast channel. In the following, we recall the Signal group messaging protocols and its security properties (§6.2);we show how it can be used to implement the E2EE-AB functionality $\mathcal{F}_{\mathsf{AB}}$. In §6.4, we define and discuss the parameters (access control, threat model, and leakage functions) of this instantiation of our framework, and how the guarantees of the Signal group messaging protocol transfer to the guarantees of SignalCD.

6.2 The Signal group messaging protocol

The Signal group messaging protocol consists of two subprotocols: the Signal protocol $\mathcal{F}_{\mathsf{Signal}}$, which provides end-to-end encrypted messaging between two parties, and the Signal private group system, which extends the Signal protocol to realize secure group messaging and enables users to communicate within groups rather than just one-on-one.

Signal protocol. The Signal protocol $\mathcal{F}_{\mathsf{Signal}}$ enables to build end-to-end encrypted *channels* for two-party messaging. It provides very strong security guarantees such as forward security and post-compromise security, and its security has been extensively analyzed [5, 11, 13].

Signal private group system. The Signal private group system [12] augments the Signal protocol to support groups, by modeling a group as a set of logical two-party Signal channels between group members (to a first order, see *Sender keys* below). This ensures that the security properties of the Signal protocol carry over to the group setting. We introduce ideal functionalities of the Signal group system relevant to SignalCD in §6.3; for additional functionalities and objects (profile keys and timing-based authentication), and more details about the security properties we refer the reader to [12].

Group creation. When creating a new group gid, the creating user uid samples a group master key gmsk. This key serves to derive sets of group secret parameters gsk and of group public parameters gpp. The group creator uid sends gpp to the server. Whenever adding a new user, the creator sends the gmsk to them, which the new users can use to generate the groups secret (gsk) and public (gpp) parameters.

Encrypted group tables. For every group gid, the Signal server stores a group table T. Each table entry $\mathsf{T}[\mathsf{gid}]$ stores a list of encrypted ids of members of that group, along with their permissions. We write $\mathsf{T}[\mathsf{gid},\mathsf{ct}] = \mathsf{perm}$ to denote that the user with encrypted id ct has permission perm in group gid. ct is the encryption of the user's identity uid as $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{gsk},\mathsf{uid})$. Group table entries are encrypted using an authenticated encryption scheme $(\mathsf{Enc},\mathsf{Dec})$, which ensures that the server cannot neither read nor modify the individual ct. However, the current Signal protocol does not provide integrity for the table as a whole and thus it does not provide any protection against the server deleting or re-instating individual accounts ct in the group table. In particular, it does not prevent the server from rolling back the group table to a previous state. Additionally, Signal only supports admin level permissions for users.

Sending messages. To send a message msg to a group gid, users anonymously authenticate themselves to the server proving they are members of the group. More concretely, uid requests AuthCredResponse from the server over a mutually authenticated channel. The user then generates an anonymous credential presentation cred using gsk and AuthCredResponse, and encrypts its identity uid to obtain ct as described above. To authenticate itself to the server (and prove that they are member of a group), uid sends cred and ct to the server over an unauthenticated channel. The server can then, using gpp, verify cred (and with it, that ct was constructed correctly and that it belongs to the user it is communicating with). The unauthenticated channel ensures that the server does not know which user it is communicating with. If the verification is successful, the server sends

the group table $\mathsf{T}[\mathsf{gid}]$ to the user. The user then decrypts the group table using gsk to obtain the list of group members and their permissions. Finally, the user sends msg to each of the group members listed in the group table, using the already established pairwise Signal channels. To receive messages, the user operates analogously, receiving messages in the pairwise Signal channels instead of sending.

Sender keys. To improve efficiency in practice, the Signal group system uses *sender keys* whenever possible (with pairwise communication only used as fallback) [7], which are symmetric keys shared between all group members. Each user encrypts their messages to the group using a sender key, which is then encrypted and sent to each group member using the pairwise Signal channels. This reduces the number of messages that need to be sent when sending a message to a group from linear in the size of the group to constant.

6.3 E2EE-AB from Signal group messaging

In this section, we define a protocol $\Pi_{\mathsf{AB}}[\mathcal{F}_{\mathsf{Signal}}]$ (Fig. 5), which implements the functionality $\mathcal{F}_{\mathsf{AB}}$ in the $\mathcal{F}_{\mathsf{Signal}}$-hybrid model. Our implementation leverages the encapsulation (SendMsg) and decapsulation (RecvMsg) functionalities of $\mathcal{F}_{\mathsf{Signal}}$ from the UC-security proofs of the two-party Signal protocol [11], and the protocol for the Signal private group systems [12]. The Universally Composable (UC) security proof from [11] allows us to use SendMsg and RecvMsg as-is, without having to deal with the internal details (and thus we do not discuss any internal but refer the reader to [11] for details). Clients communicate with the server using TLS to realize secure and server-authenticated channels [24]. Since the AB uses these channels, all of the functionalities of Fig. 2 inherit these authenticated properties.

Setup. The protocol $\Pi_{\mathsf{AB.setup}}()$ implements $\mathcal{F}_{\mathsf{AB.setup}}()$. It initializes an instance of Signal Group Messaging, and a group table T, where $\mathsf{T}[\mathsf{gid},\mathsf{ct}]$ maps an encryption $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{gsk},\mathsf{uid})$ of an account uid to the permission of uid in the group gid. The setup protocol also sets the parameters $\mathsf{Rollback} = 0$ and $\mathsf{PermSet} = \{\bot, \mathsf{admin}\}$ as Signal only supports admin-level permissions, and initializes the table THistory. We discuss these choices in §6.4.

New Group. The protocol $\Pi_{\mathsf{AB.new}}(\mathsf{uid},\mathsf{gid})$ implements $\mathcal{F}_{\mathsf{AB.new}}(\mathsf{uid})$. Given a user uid, this protocol generates a group secret gmsk to be used for the encryption and MAC of the group members' ids in T. $\mathsf{KGen}()$ is a key generation algorithm used for generating gmsk and to derive $(\mathsf{gsk},\mathsf{gpp})$ as in [12]. The protocol uses gsk to encrypt uid into ct, creates a new group with id gid, and sets admin permissions in $\mathsf{T}[\mathsf{gid},\mathsf{uid}] = \mathsf{admin}$. In addition, it outputs gpp to the server.

Add Member. The protocol $\Pi_{\mathsf{AB.add}}(\mathsf{uid},\mathsf{gid},\mathsf{uid}',\mathsf{perm})$ implements $\mathcal{F}_{\mathsf{AB.add}}(\mathsf{uid},\mathsf{bid},\mathsf{uid}',\mathsf{perm},\mathsf{msg})$. It allows users with admin permission in the group gid to give access to the group to new users. Concretely, a user uid with corresponding ct that can authenticate as having admin in $\mathsf{T}[\mathsf{gid},\mathsf{ct}]$ can create a new entry $\mathsf{T}[\mathsf{gid},\mathsf{ct}'] = \mathsf{perm}$ that grants the related

$\Pi_{\mathsf{AB.setup}}()$

1 Initialize an empty table T and THistory
2 Set $\mathsf{PermSet} = \{\bot, \mathsf{admin}\}$, $\mathsf{Rollback} = 0$

$\Pi_{\mathsf{AB.new}}(\mathsf{uid},\mathsf{gid})$

1 **if** $\mathsf{T}[\mathsf{gid}] \neq \bot$ **return** $\bot$
2 $\mathcal{U}_{\mathsf{uid}} : \mathsf{gmsk}_{\mathsf{gid}} \leftarrow \mathsf{KGen}()$
3 $\mathcal{U}_{\mathsf{uid}} : (\mathsf{gsk}_{\mathsf{gid}}, \mathsf{gpp}_{\mathsf{gid}}) \leftarrow \mathsf{Derive}(\mathsf{gmsk}_{\mathsf{gid}})$
4 $\mathcal{U}_{\mathsf{uid}} : \mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{gsk}_{\mathsf{gid}}, \mathsf{uid})$
5 $\mathsf{T}[\mathsf{gid},\mathsf{ct}] := \mathsf{admin}$
6 $\mathcal{U}_{\mathsf{uid}} : \textbf{Send}\ (\mathsf{gpp}_{\mathsf{gid}}, (\mathsf{ct}, \mathsf{admin})) \rightarrow \mathcal{S}$
7 **Leak** $(\mathtt{new}, \mathsf{gid}, \mathcal{L}_{\mathsf{AB.new}}(\mathsf{gid},\mathsf{uid}))$

$\Pi_{\mathsf{AB.add}}(\mathsf{uid},\mathsf{gid},\mathsf{uid}',\mathsf{perm})$

1 $\mathcal{U}_{\mathsf{uid}} : \mathsf{ct}' \leftarrow \mathsf{Enc}(\mathsf{gsk}_{\mathsf{gid}}, \mathsf{uid}')$
2 **if** $\Pi_{\mathsf{AB.auth}}(\mathsf{uid},\mathsf{gid},\mathsf{cred}) \neq \mathsf{admin}$ **return** $\bot$
3 **if** $\mathsf{perm} \notin \mathsf{PermSet}$ **return** $\bot$
4 $\mathsf{T}[\mathsf{gid},\mathsf{ct}'] := \mathsf{perm}$; $\mathsf{THistory}[\mathsf{gid},\mathsf{ct}'] \mathrel{+}= \mathsf{perm}$
5 $\mathcal{U}_{\mathsf{uid}} : \mathsf{SendMsg}(\mathsf{uid},\mathsf{uid}',\mathsf{msg} := (\mathsf{gid},\mathsf{gmsk}_{\mathsf{gid}}))$
6 **Leak** $(\mathtt{add}, \mathsf{gid}, \mathcal{L}_{\mathsf{AB.add}}(\mathsf{gid},\mathsf{uid},\mathsf{uid}',\mathsf{perm},\mathsf{msg}))$

$\Pi_{\mathsf{AB.send}}(\mathsf{uid},\mathsf{gid},\mathsf{msg})$

1 **if** $\Pi_{\mathsf{AB.auth}}(\mathsf{uid},\mathsf{gid},\mathsf{cred}) \notin \{\mathsf{write},\mathsf{admin}\}$ **return** $\bot$
2 $\mathsf{T}'[\mathsf{gid}] \leftarrow \mathcal{F}_{\mathsf{Rollback}}(\mathsf{gid})$
3 **Leak** $\mathsf{List}(\mathsf{uid},\mathsf{gid})$
4 **for** $\mathsf{uid}' \in \{\mathsf{Dec}(\mathsf{gsk}_{\mathsf{gid}},\mathsf{ct}') \mid \mathsf{ct}' \in \mathsf{T}'[\mathsf{gid}]\}$:
5 $\quad \mathsf{SendMsg}(\mathsf{uid},\mathsf{uid}',\mathsf{msg})$
6 $\quad$ **Leak** $(\mathtt{send}, \mathsf{uid}, |\mathsf{msg}|, \mathcal{L}_{\mathsf{AB.send}}(\mathsf{gid},\mathsf{uid},\mathsf{uid}',\mathsf{msg}))$

$\Pi_{\mathsf{AB.receive}}(\mathsf{uid},\mathsf{gid})$

1 **if** $\Pi_{\mathsf{AB.auth}}(\mathsf{uid},\mathsf{gid},\mathsf{cred}) \notin \{\mathsf{read},\mathsf{write},\mathsf{admin}\}$ **return** $\bot$
2 $\mathsf{T}'[\mathsf{gid}] \leftarrow \mathcal{F}_{\mathsf{Rollback}}(\mathsf{gid})$
3 **Leak** $\mathsf{List}(\mathsf{uid},\mathsf{gid})$
4 **for** $(\mathsf{uid}',\mathsf{ct}') \in \{(\mathsf{Dec}(\mathsf{gsk}_{\mathsf{gid}},\mathsf{ct}'),\mathsf{ct}') \mid \mathsf{ct}' \in \mathsf{T}'[\mathsf{gid}]\}$:
5 $\quad$ **if** $\mathsf{T}'[\mathsf{gid},\mathsf{ct}'] \geq \mathsf{write}$:
6 $\quad\quad \mathsf{RecvMsg}(\mathsf{uid},\mathsf{uid}')$
7 $\quad\quad$ **Leak** $(\mathtt{rcv}, \mathsf{uid}, \mathsf{uid}', |\mathsf{msg}|, \mathcal{L}_{\mathsf{AB.rcv}}(\mathsf{gid},\mathsf{uid},\mathsf{uid}',\mathsf{msg}))$

$\Pi_{\mathsf{AB.auth}}(\mathsf{uid},\mathsf{gid},\mathsf{cred})$

1 **Leak** $(\mathtt{auth}, \mathsf{gid}, \mathcal{L}_{\mathsf{AB.auth}}(\mathsf{bid},\mathsf{uid}))$
2 **if** $\mathsf{Vfy}(\mathsf{cred},\mathsf{gpp}_{\mathsf{gid}}) = 0$ **return** $\bot$ **else** **return** $\mathsf{T}[\mathsf{gid},\mathsf{ct}]$

Figure 5: Implementation of asynchronous broadcast using $\mathcal{F}_{\mathsf{Signal}}$. We show in cyan color the steps that are affected by the value of PermSet and in red those affected by Rollback. Users call the functionalities using *unauthenticated* channels but in addition they always send their cred for the group gid.

$\mathcal{F}_{\mathsf{CD}}$	$\Pi_{\mathsf{CD}}[\mathcal{F}_{\mathsf{AB}}]$	$\Pi_{\mathsf{AB}}[\mathcal{F}_{\mathsf{Signal}}]$
$\mathcal{L}_{\mathsf{CD.new}}$	$\mathcal{L}_{\mathsf{AB.new}}$	$\mathsf{idx}(\mathsf{T}[\mathsf{gid},\mathsf{uid}])$
$\mathcal{L}_{\mathsf{CD.auth}}$	$\mathcal{L}_{\mathsf{AB.auth}}$	$\mathsf{idx}(\mathsf{T}[\mathsf{gid},\mathsf{ct}])$
$\mathcal{L}_{\mathsf{CD.share}}$	$\lvert\mathsf{doc}\rvert, \mathcal{L}_{\mathsf{AB.add}}$	$\lvert\mathsf{msg}\rvert, \mathsf{idx}(\mathsf{T}[\mathsf{gid},\mathsf{uid}]), \mathsf{idx}(\mathsf{T}[\mathsf{gid},\mathsf{uid}'])$
$\mathcal{L}_{\mathsf{CD.download}}$	$\mathcal{L}_{\mathsf{AB.rcv}}$	$\lvert\mathsf{edits}\rvert$
$\mathcal{L}_{\mathsf{CD.edit}}$	$\mathcal{L}_{\mathsf{AB.send}}$	$\lvert\mathsf{edits}\rvert, \mathsf{epoch\_id}, \mathsf{msg\_num}, \mathsf{N}$
$\mathcal{L}_{\mathsf{CD.list}}$	$\mathcal{L}_{\mathsf{AB.list}}$	$\mathsf{idx}(\mathsf{T}[\mathsf{gid},\mathsf{uid}])$

Table 1: Leakage functions for the model $\mathcal{F}_{\mathsf{CD}}$ (§3), the construction $\Pi_{\mathsf{CD}}[\mathcal{F}_{\mathsf{AB}}]$ (§5), and the instantiation $\Pi_{\mathsf{AB}}[\mathcal{F}_{\mathsf{Signal}}]$ (§6).

uid' access to the group with permission $\mathsf{perm} \in \{\mathsf{admin}\}$. It also sends the new user uid' a message msg that contains gid of the group, to notify them that they were added to it, and the gmsk. Having this key, the new user can derive gsk to generate its own cred and decrypt ct of other users when accessing $\mathsf{T}[\mathsf{gid}]$. They can also use this key to encrypt uid of new users when adding them to the group, if they were given admin permission. User removal is modeled by calling the protocol with $\mathsf{perm} = \perp$.

Send. The protocol $\Pi_{\mathsf{AB.send}}(\mathsf{uid},\mathsf{gid},\mathsf{msg})$ implements $\mathcal{F}_{\mathsf{AB.send}}(\mathsf{uid},\mathsf{bid},\mathsf{msg})$ using the SendMsg functionality from $\mathcal{F}_{\mathsf{Signal}}$. It allows a user uid with a corresponding ct whose entry $\mathsf{T}[\mathsf{gid},\mathsf{ct}]$ contains write or admin to send messages to the group gid. If the user has permission to send messages to the group, the protocol retrieves the list of users ct' in the group $\mathsf{T}[\mathsf{gid}]$. It decrypts each of the ct' to uid' using gsk and then sends the message msg to each of those users by using SendMsg of the corresponding two-party Signal channel.

Receive. The protocol $\Pi_{\mathsf{AB.receive}}(\mathsf{uid},\mathsf{gid})$ implements $\mathcal{F}_{\mathsf{AB.rcv}}(\mathsf{uid},\mathsf{bid})$ using the RecvMsg functionality from $\mathcal{F}_{\mathsf{Signal}}$. It enables a user uid with a corresponding ct to retrieve all messages sent to group gid, provided that $\mathsf{T}[\mathsf{gid},\mathsf{ct}]$ is set to at least read. The $\Pi_{\mathsf{AB.receive}}(\mathsf{uid},\mathsf{gid})$ protocol retrieves the list of users ct' in the group $\mathsf{T}[\mathsf{gid}]$ that have at least write in $\mathsf{T}[\mathsf{gid},\mathsf{ct}']$. It decrypts each of the ct' to uid' using gsk and then receives the message msg from each of those users by using RecvMsg of the two-party Signal channel.

Authentication. The protocol $\Pi_{\mathsf{AB.auth}}(\mathsf{uid},\mathsf{gid},\mathsf{cred})$ implements the (inlined) permissions checks in the functionalities in Fig. 3 and is used to realize the authenticated channels used in Fig. 2 and Fig. 3. It checks the permissions of a user uid to a group gid using anonymous credentials by verifying a user's anonymous credential presentation cred that it receives through an *unauthenticated* channel. cred provides a proof that the user own the plaintext uid corresponding to the ct in $\mathsf{T}[\mathsf{gid}]$, and the protocol can verify it using the group's gpp. It returns the $\mathsf{T}[\mathsf{gid},\mathsf{ct}]$ if it succeeds.

6.4 Access Control, Threat Model, Leakage

To prove the security of our instantiation, we provide a proof sketch for how it securely realizes the $\mathcal{F}_{\mathsf{CD}}$ and we fully define the leakage functions of the Signal-based instantiation. We prove that the Signal Group Messaging in Fig. 5 securely implements the functionalities of the E2EE asynchronous broadcast in Fig. 3, which implies the security of the E2EE-CD (Fig. 1, right). We model the leakage of the protocols in Fig. 5, and explicitly define the parametrized leakage functions that E2EE-CD inherits from Signal.

Threat model. To define the threat model of SignalCD we slightly modify the strong threat model from Signal to match the use case of investigative journalists. Concretely, we assume all the users to be honest-but-curious, given that they trust each other, and that they benefit from the guarantees of the E2EE-CD system. We also assume the server used for message delivery and storing the group tables to be malicious with some restrictions. Regarding the server, the Signal protocol assumes a fully malicious server that can read, inject, and change the order of any messages sent, and guarantees confidentiality and integrity of the messages under this threat model (see [11] for Universally Composable (UC) proof of security), which our protocol inherits. The Signal private group system assumes a malicious server that stores the group tables, and can read and change any of the group table entries. However, since the Signal group system does not provide integrity for the whole table, the protocol is trivially insecure against rollback attacks. Therefore, SignalCD can only be secure in a setting where Signal itself is secure. We make this explicit by setting $\mathsf{Rollback} = 0$ such that our analysis meaningfully reflects Signal's threat model. Access control is also enforced by the server. Security proofs for the security of the encryption, MAC, and the anonymous credential system are discussed in [12], and apply to our protocol.

Leakage. The leakage of SignalCD is the union of the leakage of Π_{AB}, in this instantiation the leakage of the Signal private group system, and that of $\mathcal{F}_{\mathsf{Signal}}$ (see Fig. 1). In addition to the fundamental leakage that is explicitly shown in Fig. 5, the leakage of Π_{AB} also includes the leakage functions $\mathcal{L}_{\mathsf{AB.auth}}, \mathcal{L}_{\mathsf{AB.list}}, \mathcal{L}_{\mathsf{AB.add}}, \mathcal{L}_{\mathsf{AB.new}}$ which are defined according to the leakage from [12]; and the leakage functions $\mathcal{L}_{\mathsf{AB.send}}, \mathcal{L}_{\mathsf{AB.rcv}}$ which are part of the leakage of the two-party Signal protocol functionalities SendMsg, RecvMsg from [11]. We define all of them in Table 1, which details the relation between the leakage functions of the model, construction, and instantiation.

When a user in a group gid calls any of the functionalities that contain the leakage functions $\mathcal{L}_{\mathsf{AB.auth}}, \mathcal{L}_{\mathsf{AB.list}}, \mathcal{L}_{\mathsf{AB.add}}$, these functions leak the index of that user in the group table $\mathsf{T}[\mathsf{gid}]$. This is because whenever the user interacts with the server, they send their own ct and which group gid they are calling the functionality on. Therefore, since the server has access to the group table, the server can look up where ct is stored in the table and therefore learn all the information associated with that ct (mainly the permissions of that user), even though the server never learns uid. The leakage of $\mathcal{L}_{\mathsf{AB.send}}, \mathcal{L}_{\mathsf{AB.rcv}}$ come from the leakage of the two-party Signal

channel, which sends additional information alongside the encapsulated message. These include the epoch epoch_id of the message, the index of the message in this epoch msg_num, and the number of messages N sent in the last epoch. These values are required for the receiver to correctly implement ratcheting and decapsulating the message.

Security. We prove Lemma 2 in §A.2; Corollary 1 follows directly from Lemmas 1 and 2.

Lemma 2. $\Pi_{\mathsf{AB}}[\mathcal{F}_{\mathsf{Signal}}]$ *(Fig. 5) securely implements* $\mathcal{F}_{\mathsf{AB}}[\mathsf{PermSet}, \mathsf{Rollback}, \mathcal{L}_{\mathsf{AB}}]$ *(Fig. 3) in the* $\mathcal{F}_{\mathsf{Signal}}$*-hybrid model, with leakage functions* $\mathcal{L}_{\mathsf{AB}}$ *as defined in Table 1,* $\mathsf{PermSet} = \{\bot, \mathsf{admin}\}$*, and* $\mathsf{Rollback} = 0$*.*

Corollary 1. *If* Rec *is a convergent reconciliation protocol (Def. 1),* $\Pi_{\mathsf{CD}}[\mathsf{Rec}, \Pi_{\mathsf{AB}}[\mathcal{F}_{\mathsf{Signal}}]]$ *securely implements* $\mathcal{F}_{\mathsf{CD}}[\mathsf{Rec}, \mathsf{PermSet}, \mathsf{Rollback}, \mathcal{L}_{\mathsf{AB}}]$ *for* $\mathcal{L}_{\mathsf{AB}}$ *as defined in Table 1,* $\mathsf{PermSet} = \{\bot, \mathsf{admin}\}$*, and* $\mathsf{Rollback} = 0$*.*

7 Practicality of SignalCD

We implement a prototype of SignalCD using the `signal-cli`[3] third-party client to interact with the Signal servers. As our reconciliation mechanism, we use the `automerge` library[4]. In the spirit of modularity, we use an `automerge` wrapper provided by the `crdt-benchmarks`[5] suite of benchmarks, which allows us to also easily benchmark other CRDT libraries (`yjs`, `ywasm`, and `loro`). Our prototype implementation, benchmarks, and evaluation scripts are available online at https://github.com/spring-epfl/signal-collaborative-documents.

We describe how users interact with SignalCD, which we illustrate in Fig. 6. We assume that users have already registered with Signal using an app on a phone or other device.

Setting up SignalCD. For usability, SignalCD is logically an additional device in the user's Signal account. This allows users to leverage the existing Signal infrastructure for registration, authentication, and key management, fulfilling the Minimal infrastructure, No new trust assumptions, and Simple key management requirements. At setup, the user installs SignalCD on their device (e.g., laptop); SignalCD then displays a QR code that the user scans with the device on which they have an active Signal session. This links SignalCD to the user's Signal account, and allows it to send and receive messages through the Signal servers.

New document. When a user wants to create a new document, SignalCD creates a new Signal group, with the user being the first member of that group. The user can then start editing the document. The user can also archive and mute this group in their other devices to avoid unnecessary notifications.

[3] https://github.com/AsamK/signal-cli
[4] https://automerge.org/
[5] https://github.com/dmonad/crdt-benchmarks

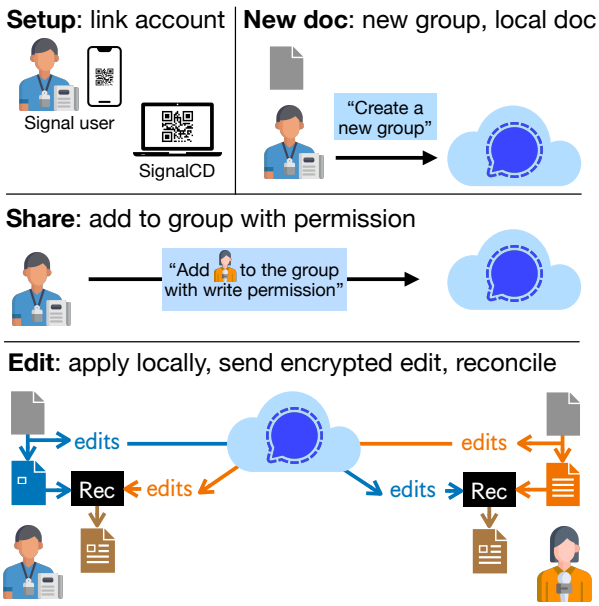

Figure 6: Overview of SignalCD, our E2EE-CD system for investigative journalists.

Editing. When a user makes an edit to the document, SignalCD invokes `automerge`'s API to apply that edit to their local copy of the document. It then serializes this edit, and sends it to the Signal group corresponding to the document. Upon receiving a message from this Signal group, SignalCD deserializes that message as an update, and uses `automerge`'s reconciliation mechanism on the local copy of the document.

Sharing. When sharing a document, the user specifies the phone number or Signal username of the user they want to share the document with. SignalCD then adds that user to the Signal group corresponding to the document.

7.1 SignalCD benchmarking

We evaluate the practicality of SignalCD by answering the following questions:

- *Is SignalCD efficient for asynchronous collaboration?*
- *Does SignalCD allow real-time synchronous collaboration?*
- *How does SignalCD scale with the number of users?*

Experimental setup. We run our experiments inside a Docker container on a MacBook Pro with an M2 Max chip with 32 GB of RAM, using the following two network configurations: `slow` (30 ms latency / 5 Mbit s^{-1} bandwidth, roughly corresponding to a 4G connection), and `fast` (5 ms latency / 60 Mbit s^{-1}, roughly corresponding to a WiFi connection). We rely on the Signal servers to deliver messages. No public data is available on the configuration of these servers, however Signal had 55 million active users (as of 2023), and its

infrastructure costs totalled 14 million US dollars per year (as of November 2023)[6]. One can thus roughly assume that a single user costs Signal a fraction of a US dollar per year and thus consider our prototype does not incur in significant cost for Signal. We use two Signal accounts (one per user), and we send and receive messages through a Signal group. In our experiments, we have not observed any message rate limiting.

While our measurements indicate SignalCD is practical, we stress that our benchmarking is very conservative. First, because we run both users on the same container on the same machine. Second, because of our use of the `signal-cli` client, sending and receiving messages is done through a websocket connection to the Java `signal-cli` local server, which issues a request to the Signal servers (for our synchronous collaboration benchmarks), or by spawning a child process which launches a new instance of the `signal-cli` binary (for our asynchronous collaboration benchmarks). Third, we send a Signal message for every character edit (rather than sending a message every few characters or per word, or adaptively based on the presence of other users). Thus, we expect our system to perform better in a more realistic setting with no system overhead, with an optimized client implementation, and with tuned edit batching.

Asynchronous collaboration. We first consider the case where collaborating users are not online at the same time, and therefore do not edit the document concurrently. In this scenario, user $\mathcal{U}_1$ makes a series of edits to the document. Later, when user $\mathcal{U}_2$ comes online, they need to receive all edits from $\mathcal{U}_1$. We benchmark the time it takes to fetch all edits and to reconcile them, varying the number of edits. From a user's perspective, this corresponds to the time-to-first-render. In our benchmark, $\mathcal{U}_1$ makes single-character edits corresponding to a real-world editing trace which we obtain from [18].

Fig. 7 shows the number of received edits as a function of the time elapsed since $\mathcal{U}_2$ came online. After a delay of roughly 2.4 s corresponding to the time it takes to launch a new instance of the `signal-cli` client, $\mathcal{U}_2$ receives edits at a rate of roughly 460 edits per second.

Synchronous collaboration. Next, we consider the case where multiple users are online at the same time, and therefore edit the document concurrently (either on overlapping or non-overlapping sections). We focus on the two-user case in this subsection. In order to simulate a realistic synchronous editing scenario, we model both users as skilled typists: $\mathcal{U}_1$ types a character every 200 ms to 300 ms (200–300 characters per minute), and $\mathcal{U}_2$ types a character every 100 ms to 400 ms (150–600 characters per minute). Every character is serialized as an update by Rec and sent to the Signal group corresponding to the document. Applying the reconciliation mechanism for a single update always takes less than 2 ms, and is negligible compared to the network latency.

For each edit, we record the *sending latency* (from the client

[6] https://signal.org/blog/signal-is-expensive/

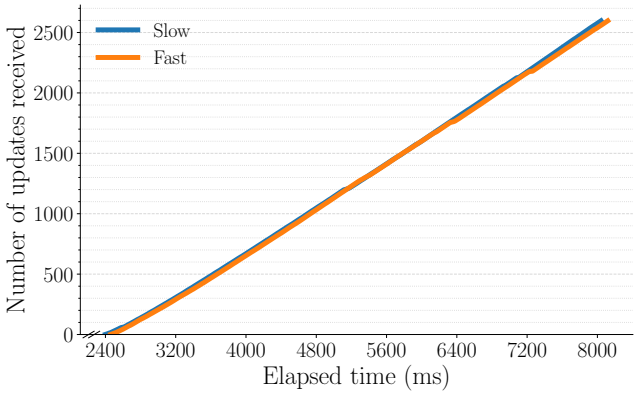

Figure 7: Received edits as a function of time since launch, for the `slow` and `fast` network configurations. After 2.4 s to launch a new instance of the `signal-cli` client, users can process roughly 460 single-character edits per second, even with a slow connection.

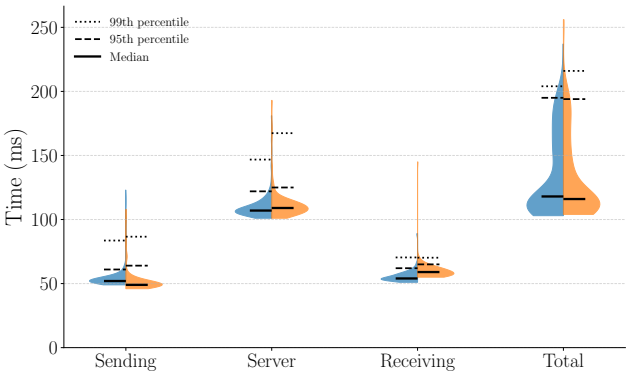

Figure 8: Violin plots of sending, server, and receiving latency for synchronous editing, measured for 100 single-character edits from each user (left/blue for the `slow` configuration, right/orange for `fast`).

encrypting and dispatching the message until its reception by the Signal server), the *server latency* (from reception by the Signal server until it is delivered to the recipient), and the *receiving latency* (from when the recipient's client receives the message until it processes it and updates the document). We show violin plots of the sending, server, and receiving latency for 100 messages sent by $\mathcal{U}_1$ and $\mathcal{U}_2$ in Fig. 8; we eschew the first 10 messages to avoid warm-up effects. The median end-to-end total latency is roughly 120 ms, with an 99$^{\text{th}}$ percentile below 220 ms. Given these delays, in almost all cases, when $\mathcal{U}_1$ types a character, $\mathcal{U}_2$ receives, decrypts, and reconciles the update corresponding to that character into its local document before $\mathcal{U}_1$ types the next character. Thus, even in our conservative setup, users can see each other's edits in real-time, and can react to them immediately. We show additional plots in §C.

Scaling (users). We are unfortunately not able to benchmark the performance of SignalCD with more than two users, as this would require multiple Signal accounts, each registered with a different phone number. However, we note that, thanks to Signal's sender keys, the sending latency of SignalCD does not grow with the number of users: only one encryption is needed regardless of the number of users. The receiving latency is also independent of the number of users, as each user processes each message independently. Only the server latency depends on the number of users, as the server needs to serve the edit ciphertext to each user in the group. The server latency grows, at worst, linearly with the number of users, but might grow sublinearly or stay constant depending on the server's capability to parallelize processing. Extrapolating from our measurements, for N users, the median server latency should thus be between 120 ms (for perfect parallelism) and $N \cdot 120$ ms (for fully serial server execution). The Signal server uses parallelization and a pool of threads to dispatch messages to multiple users, and thus we expect the median server latency in SignalCD to be close to our measurements for two when the group size grows. During our experiments, we have also observed no explicit messaging restrictions when sending messages (reconciliation updates) to Signal.

In the case of investigative journalists, our interviews reveal that most editing collaborations involve a small number of users, and that document writing in large groups is rare. Large groups are typically used to share access to leaked documents, or draft articles, to a large audience of fellow journalists, who only get read access and do not edit the document. Since in this case most members of the group will not write to the document, we believe that the efficiency of SignalCD is still sufficient for effective journalistic collaboration.

Scaling (edit sizes). We also benchmark how SignalCD scales with the size of edits. While single-character edits are common during typing and may allow for a more fluid collaboration experience, users may also make larger edits, e.g. when pasting text, or when making larger structural changes to the document. Fig. 9 shows how the time to apply the reconciliation update changes with the size of edits, for edits corresponding to the insertion of 1 up to 50'000 characters. The latter represents between 7'000 and 12'000 words (for reference, this paper has about 14'500 words). We observe that after a thousand characters the growth is superlinear. Yet, even for very large edits (e.g., inserting 50 000 characters at once), reconciliation takes less than 150 ms. We note that improving the performance of reconciliation mechanisms for large edits is an orthogonal research problem, and that our system would benefit from future improvements in this area.

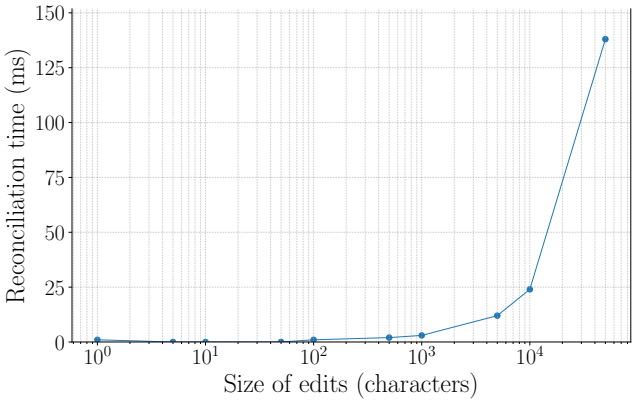

Figure 9: Time to apply reconciliation update versus edit size (number of characters inserted). Even for very large edits (e.g., inserting 50 000 characters at once), reconciliation takes less than 150 ms.

8 Final remarks

In this work, we provide a formal treatment of requirements and constructions of End-to-End Encrypted Collaborative Documents (E2EE-CD). Our generic framework shows that an end-to-end encrypted asynchronous broadcast channel is sufficient to build secure E2EE-CD systems. To prove this point, we design and implement SignalCD, an E2EE-CD system tailored for investigative journalists that instantiates our framework with the Signal's group messaging protocol. We show that SignalCD is practical for real-world workloads for investigative journalists.

Future work. In this work, we have formalized the minimal requirements to enable secure and private collaborative document editing. Yet, it would be interesting to explore how to properly formalize stronger security guarantees for E2EE-CD, such as forward security, post-compromise security, hybrid/post-quantum security, and metadata-hiding; and study whether it is still possible to achieve these stronger guarantees in a modular fashion as in our construction.

While enabling journalists to collaborate securely is an important use case for E2EE-CD, we acknowledge that SignalCD is limited in scope (relatively powerful devices, good network connectivity, limited number of users per document, small to medium-sized documents). To ensure that our black-box use of the underlying E2EE broadcast channel can support general-purpose collaborative document settings future work should investigate more constructions of E2EE-AB such that the system can accommodate more demanding settings.

Finally, while we show the practicality of SignalCD from a performance perspective, it is necessary to expand our prototype implementation into a full-fledged application with a user-friendly interface, and test its usability with journalists.

9 Ethical Considerations

Designing tools to enable secure and privacy-preserving digital interactions inherently comes with ethical considerations.

End-to-end encryption (E2EE) protects the privacy and security of users, shielding them from surveillance and censorship. However, E2EE can also be used by malicious actors to facilitate harmful activities, such as the spread of misinformation, harassment, or the coordination of criminal activities. In this paper, we take the (widely accepted in the community of security and privacy researchers) stance that the benefits of E2EE outweigh its potential harms, and that the development of secure and privacy-preserving digital tools is a net positive for society.

When choosing our stakeholders, we also take care of selecting a use case beneficial to society such as that of investigative journalists. Investigative journalism has a critical role by revealing corruption scandals and tracing infringements of laws, and critically assessing governments', companies' and other organisations' policies or actions, keeping them accountable to the public. Moreover, investigative journalists may be more at-risk of surveillance and censorship than the general population, and have a strong ethical commitment to protecting the privacy and security of their sources and collaborators; and thus are an ideal target of our work.

We also take care that our prototype does not cause any harm to users or live systems. As we detail in §7, our use of the actual Signal server causes negligible overhead and cost. When testing the system we also avoid the use of any sensitive data by taking the character-by-character, already published, editing trace of a paper provided by the authors of the automerge library.

10 Open Science

The code for our prototype implementation of SignalCD is available at https://github.com/spring-epfl/signal-collaborative-documents[7]. The artifact contains code to reproduce the benchmarks from §7. This code requires the use of registered SIM cards to send and receive messages. Our artifact contains registered configurations for two such SIM cards, although interaction with the authors might be needed in the case of a Signal update between the time of submission and the time of artifact review, which might require re-linking of the accounts (this requires access to a device where the phone number has already been registered). We note that our artifact might be bound to a specific version of the deployed Signal protocol and server code, and that future updates to Signal might break compatibility with the artifact as submitted. The artifact contains a detailed README with instructions on how to set up and run the benchmarks.

We take care to ensure that our artifacts are reproducible by providing a multi-platform Docker setup, which can be run on Linux and MacOS, on x86_64 or ARM architecture. For the sake of reproducibility, we also provide configurations to cap the network bandwidth and latency of the Docker container to match the conditions of our benchmarks. Finally, we provide scripts to automatically run the benchmarks all benchmarks, as well as re-generate the figures shown in §7.

[7] The artifact is also available at https://doi.org/10.5281/zenodo.17973971.

Acknowledgements

The authors thank investigative journalists from the International Consortium of Investigative Journalists (ICIJ), as well as Alastair Beresford, Daniel Hugenroth, and Martin Kleppmann for insightful discussions during the initial stages of this project.

References

[1] URL: https://cryptpad.org/.

[2] Keybase Book: Learn about using Keybase for Git. URL: https://book.keybase.io/git.

[3] Jul 2024. URL: https://proton.me/blog/docs-proton-drive.

[4] Collaborating in Overleaf | Overleaf docs, August 2025. URL: https://docs.overleaf.com/collaborating/collaborating-in-overleaf.

[5] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Cham, May 2019. doi:10.1007/978-3-030-17653-2_5.

[6] Matilda Backendal, Hannah Davis, Felix Günther, Miro Haller, and Kenneth G. Paterson. A Formal Treatment of End-to-End Encrypted Cloud Storage, 2024. Publication info: A major revision of an IACR publication in CRYPTO 2024. URL: https://eprint.iacr.org/2024/989.

[7] David Balbás, Daniel Collins, and Phillip Gajland. WhatsUpp with sender keys? Analysis, improvements and security proofs. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part V*, volume 14442 of *LNCS*, pages 307–341. Springer, Singapore, December 2023. doi:10.1007/978-981-99-8733-7_10.

[8] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 1253–1270. USENIX Association, August 2023.

[9] Manuel Barbosa, Bernardo Ferreira, João C. Marques, Bernardo Portela, and Nuno M. Preguiça. Secure conflict-free replicated data types. In *ICDCN '21: International Conference on Distributed Computing and Networking, Virtual Event, Nara, Japan, January 5-8, 2021*, pages 6–15. ACM, 2021. doi:10.1145/3427796.3427831.

[10] Benjamin Beurdouche, Eric Rescorla, Emad Omara, Srinivas Inguva, and Alan Duric. The Messaging Layer Security (MLS) Architecture. RFC 9750, April 2025. URL: https://www.rfc-editor.org/info/rfc9750, doi:10.17487/RFC9750.

[11] Ran Canetti, Palak Jain, Marika Swanberg, and Mayank Varia. Universally composable end-to-end secure messaging. (2022/376), 2022. Publication info: A major revision of an IACR publication in CRYPTO 2022. URL: https://eprint.iacr.org/2022/376.

[12] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. (2019/1416), 2019. URL: https://eprint.iacr.org/2019/1416.

[13] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, October 2020. doi:10.1007/s00145-020-09360-1.

[14] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying Strong Eventual Consistency in Distributed Systems. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, October 2017. arXiv:1707.01747 [cs]. URL: http://arxiv.org/abs/1707.01747, doi:10.1145/3133933.

[15] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving correctness of transformation functions in real-time groupware. In Kari Kuutti, Eija Helena Karsten, Geraldine Fitzpatrick, Paul Dourish, and Kjeld Schmidt, editors, *Proceedings of the Eighth European Conference on Computer Supported Cooperative Work, 14-18 September 2003, Helsinki, Finland*, pages 277–293. Springer, 2003. doi:10.1007/978-94-010-0068-0_15.

[16] Apple Inc. Secure features in the notes app. Accessed 2025-03-12. URL: https://support.apple.com/en-gb/guide/security/sec1782bcab1/web.

[17] Martin Kleppmann and Alastair R. Beresford. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, October 2017. arXiv:1608.03960 [cs]. URL: http://arxiv.org/abs/1608.03960, doi:10.1109/TPDS.2017.2697382.

[18] Martin Kleppmann and Alastair R. Beresford. A conflict-free replicated JSON datatype. *IEEE Trans. Parallel Distributed Syst.*, 28(10):2733–2746, 2017. doi:10.1109/TPDS.2017.2697382.

[19] Nadim Kobeissi. Capsule: A protocol for secure collaborative document editing. Cryptology ePrint Archive, Report 2018/253, 2018. URL: https://eprint.iacr.org/2018/253.

[20] Ya-Nan Li, Yaqing Song, Qiang Tang, and Moti Yung. End-to-end encrypted git services. Cryptology ePrint Archive, Paper 2025/1208, 2025. URL: https://eprint.iacr.org/2025/1208, doi:10.1145/3719027.3744815.

[21] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. In *Proceedings of the 19th International Conference on Supporting Group Work*, pages 39–49, Sanibel Island Florida USA, November 2016. ACM. URL: https://dl.acm.org/doi/10.1145/2957276.2957310, doi:10.1145/2957276.2957310.

[22] Bernardo Portela, Hugo Pacheco, Pedro Jorge, and Rogério Pontes. General-purpose secure conflict-free replicated data types. In *CSF 2023 Computer Security Foundations Symposium*, pages 521–536. IEEE Computer Society Press, July 2023. doi:10.1109/CSF57540.2023.00030.

[23] Aurel Randolph, Hanifa Boucheneb, Abdessamad Imine, and Alejandro Quintero. On consistency of operational transformation approach. In Mohamed Faouzi Atig and Ahmed Rezine, editors, *Proceedings 14th International Workshop on Verification of Infinite-State Systems, Infinity 2012, Paris, France, 27th August 2012*, volume 107 of *EPTCS*, pages 45–59, 2012. doi:10.4204/EPTCS.107.5.

[24] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in signal, whatsapp, and threema. (2017/713), 2017. Publication info: Published elsewhere. Major revision. 3rd IEEE European Symposium on Security and Privacy (EuroS&P 2018). URL: https://eprint.iacr.org/2017/713.

[25] CryptPad Team @ XWiki SAS. Cryptpad: End-to-end encrypted collaboration suite. v1.0.0. URL: https://blog.cryptpad.org/images/whitepaper.pdf.

[26] Marc Shapiro, Nuno M. 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 - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011. doi:10.1007/978-3-642-24550-3_29.

[27] Chengzheng Sun and Clarence A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In Steven E. Poltrock and Jonathan Grudin, editors, *CSCW '98, Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work, Seattle, WA, USA, November 14-18, 1998*, pages 59–68. ACM, 1998. doi:10.1145/289444.289469.

[28] Sean Whitton. spwhitton/git-remote-gcrypt, December 2025. original-date: 2016-01-05T12:11:30Z. URL: https://github.com/spwhitton/git-remote-gcrypt.

[29] Yi Xu, Chengzheng Sun, and Mo Li. Achieving convergence in operational transformation: conditions, mechanisms and systems. In Susan R. Fussell, Wayne G. Lutters, Meredith Ringel Morris, and Madhu C. Reddy, editors, *Computer Supported Cooperative Work, CSCW '14, Baltimore, MD, USA, February 15-19, 2014*, pages 505–518. ACM, 2014. doi:10.1145/2531602.2531629.

A Proofs

A.1 Proof of Lemma 1

Proof. We first note that the protocol is correct, i.e., it realizes the same functionality as Fig. 2. This follows from the convergence property of Rec (Def. 1), which ensures that all users eventually converge to the same document state after applying the same set of edits, regardless of the order in which they are applied.

To argue security, we proceed through the following sequence of hybrids, starting with the real-world $\mathcal{F}_{\mathsf{AB}}[\mathcal{L}_{\mathsf{AB}}]$-hybrid protocol $\Pi_{\mathsf{CD}}[\mathsf{Rec}, \mathcal{F}_{\mathsf{AB}}[\mathcal{L}_{\mathsf{AB}}]]$, and ending with the ideal-world $\mathcal{F}_{\mathsf{CD}}[\mathsf{Rec}, \mathcal{L}_{\mathsf{CD}}]$ functionality.

H_1: Replace $\Pi_{\mathsf{CD.new}}$ with $\mathcal{F}_{\mathsf{CD.new}}(\mathsf{uid})$. This hybrid induces the exact same view as the real-world protocol, as both hybrids have the same fundamental leakage, the same leakage function $\mathcal{L}_{\mathsf{CD.new}} = \mathcal{L}_{\mathsf{AB.new}}$, and the same identifiers $\mathsf{did} = \mathsf{bid}$.

H_2: Replace $\Pi_{\mathsf{CD.share}}$ with $\mathcal{F}_{\mathsf{CD.share}}(\mathsf{uid}, \mathsf{did}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$. This hybrid induces the exact same view as the previous hybrid, as both hybrids have the same access control structure PermSet, rollback flag Rollback, fundamental leakage, authentication leakage $\mathcal{L}_{\mathsf{AB.auth}}$, and the same leakage function $\mathcal{L}_{\mathsf{CD.share}} = \mathcal{L}_{\mathsf{AB.add}}$.

H_3: Replace Construction with $\mathcal{F}_{\mathsf{CD.edit}}(\mathsf{uid}, \mathsf{did}, \mathsf{edits})$. $\mathcal{F}_{\mathsf{AB.send}}(\mathsf{uid}, \mathsf{bid}, \mathsf{msg})$ leaks uid and allbid listed by the user. This is the same as $\mathcal{F}_{\mathsf{CD.edit}}(\mathsf{uid}, \mathsf{did}, \mathsf{edits})$ leaking the size of edits,did, and uid of the user that did the edits. Since $\mathcal{L}_{\mathsf{CD.edit}} = \mathcal{L}_{\mathsf{AB.send}}$, security follows.

H_4: Replace $\Pi_{\mathsf{CD.download}}$ with $\mathcal{F}_{\mathsf{CD.download}}(\mathsf{uid}, \mathsf{did})$. $\mathcal{F}_{\mathsf{AB.rcv}}(\mathsf{uid}, \mathsf{bid})$ leaks both uid and the bid it requested, which is the same as CD leaking the did, uid. However, AB also leaks from which uid' the user is receiving the edits. This can be simulated to the adversary in $\mathcal{F}_{\mathsf{CD.download}}(\mathsf{uid}, \mathsf{did})$ by additionally (re)leaking the set of uid', (which are known since each element of the set was already leaked during $\mathcal{F}_{\mathsf{CD.edit}}(\mathsf{uid}, \mathsf{did}, \mathsf{edits})$) that were carried out since the last time uid called $\mathcal{F}_{\mathsf{CD.download}}(\mathsf{uid}, \mathsf{did})$.

H_5: Replace Construction with $\mathcal{F}_{\mathsf{CD.list}}(\mathsf{uid})$. This hybrid induces the exact same view as the previous hybrid, as both hybrids have the same fundamental leakage, the same leakage function $\mathcal{L}_{\mathsf{CD.list}} = \mathcal{L}_{\mathsf{AB.list}}$, and the same identifiers $\mathsf{did} = \mathsf{bid}$.

Finally, Hybrid 5 is exactly the ideal functionality $\mathcal{F}_{\mathsf{CD}}[\mathsf{Rec}]$. □

A.2 Proof of Lemma 2

Proof. Correctness of $\Pi_{\mathsf{AB}}[\mathcal{F}_{\mathsf{Signal}}]$ follows directly from the correctness of the Signal protocol and the fact that the protocol correctly implements each of the functionalities in Fig. 3. In order to show security, we proceed through a series of hybrids, starting with the real-world $\mathcal{F}_{\mathsf{Signal}}$-hybrid $\Pi_{\mathsf{AB}}[\mathcal{F}_{\mathsf{Signal}}]$ protocol, and ending with the ideal functionality $\mathcal{F}_{\mathsf{AB}}$. In some hops, we reduce to the security of the Signal protocol for two-party messaging [11]. For other hops pertaining to the Signal group messaging protocol, we follow the proof of [12].

H_1: In Π_{AB}, replace all presentation proofs with simulated proofs in all the authentication steps. This hybrid is indistinguishable from the real world due to the anonymity property of the credential system.

H_2: In $\Pi_{\mathsf{AB.receive}}(\mathsf{uid}, \mathsf{gid})$, abort if any ciphertext ct' decrypts correctly ($\mathsf{Dec}(\mathsf{gsk}_{\mathsf{gid}}, \mathsf{ct}') \neq \perp$), but was not uploaded by another user through $\Pi_{\mathsf{AB.add}}(\mathsf{uid}, \mathsf{gid}, \mathsf{uid}', \mathsf{perm})$. The probability of this event is bounded by the advantage of any adversary against the ciphertext integrity (INT-CTXT) security of the authenticated encryption scheme.

H_3: In $\Pi_{\mathsf{AB.new}}(\mathsf{uid}, \mathsf{gid})$ and $\Pi_{\mathsf{AB.add}}(\mathsf{uid}, \mathsf{gid}, \mathsf{uid}', \mathsf{perm})$, generate ciphertexts using the selective opening simulator instead of encrypting the actual identities. This hybrid is indistinguishable from the previous one due to the selective opening security of the authenticated encryption scheme.

H_4: Replace SendMsg and RecvMsg from the Signal Protocol $\mathcal{F}_{\mathsf{Signal}}$ with ideal secure channels. This is indistinguishable because $\mathcal{F}_{\mathsf{Signal}}$ UC-realizes a secure messaging channel [11]

Finally, the last hybrid is indistinguishable from $\mathcal{F}_{\mathsf{AB}}$, as the leakage functions $\mathcal{L}_{\mathsf{AB.add}}$, $\mathcal{L}_{\mathsf{AB.list}}$, $\mathcal{L}_{\mathsf{AB.new}}$, $\mathcal{L}_{\mathsf{AB.auth}}$, $\mathcal{L}_{\mathsf{AB.send}}$, and $\mathcal{L}_{\mathsf{AB.rcv}}$ensure that hybrid 4 and $\mathcal{F}_{\mathsf{AB}}$ have the same leakage. □

B Extensions

We discuss possible extensions of E2EE-CD, both in terms of additional requirements and potential constructions to achieve them.

B.1 Document lifecycle

Archival. Recall that for the Archival requirement, an admin user must be able to make a document read-only, preventing any further edits. This can be implemented by having the admin user calling the sharing functionality $\mathcal{F}_{\mathsf{CD.share}}(\mathsf{uid}, \mathsf{did}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$ with read for all users, and by additionally preventing itself from writing to the file in the future. This will overwrite all existing permissions with read, making the document read-only: users will neither be able to call $\mathcal{F}_{\mathsf{CD.edit}}(\mathsf{uid}, \mathsf{did}, \mathsf{edits})$ to edit the document, nor $\mathcal{F}_{\mathsf{CD.share}}(\mathsf{uid}, \mathsf{did}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$ to recover edit or admin permissions.

Deletion. Deletion can also be implemented using the sharing functionality $\mathcal{F}_{\mathsf{CD.share}}(\mathsf{uid}, \mathsf{did}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg})$ by revoking all user's access in $\mathsf{T}[\mathsf{did}]$. Revoking all permissions implies that no user can call any of the functionalities for the document (and no user can be re-assigned administrator permissions), including $\mathcal{F}_{\mathsf{CD.download}}(\mathsf{uid}, \mathsf{did})$ to retrieve the document. Thus, the document effectively does not exist anymore.

Soft archival and deletion. When using soft deletion, documents are not deleted immediately, but rather marked as deleted. This allows the user to restore the document later, if needed. This can be achieved by having users' clients keep a list of deleted documents, and not displaying them in the list of documents. This feature can also be combined with the auto-deletion functionality below, which will delete the document after a certain period of time.

Auto-archival and deletion. When using auto-deletion, documents are deleted automatically after a certain period of time. This can be achieved by having the server keep track of the last edit time of each document, and deleting it if it has not been edited for a certain period of time. Note that this requires cooperation of the server, or periodic polling of a client.

B.2 Consensual sharing

In some cases, it is beneficial to prevent users from sharing documents with other users without their consent. This can be realized with a client-side implementation, which asks the user for consent before accepting the sharing request. If the user accepts, the client continues as before; otherwise, the client ignores the sharing request and hides the document from the user. An interesting open problem is how to cryptographically enforce this consent, preventing users from being added to a document without their consent. This would require a more involved protocol, where users can only be added to a document if they provide a cryptographic proof of consent.

B.3 Ephemeral data

In order to effectively collaborate, if two or more users are concurrently editing the same document, they may wish to have access to metadata about other user's real-time editing behavior (e.g., which users are also in an editing session for this document, or where their cursor is). This data is not be part of the document itself, nor its edit history, but might be overlayed on top of the document view in the user interface. This can be achieved by having the clients send ephemeral data (e.g., cursor position, selection, etc.) to each other using the same end-to-end encrypted asynchronous broadcast channel used for sending edits. This ephemeral data can be sent at a higher frequency than edits, and can be discarded after a certain period of time. However, this requires either additional control headers to distinguish updates from ephemeral data, or a separate broadcast channel for ephemeral data. For the sake of conciseness, we do not describe this extension in our protocol Π_{CD}, and we leave it to future work to explore the design space for end-to-end encrypted ephemeral data in collaborative editing systems.

B.4 Metadata hiding

While E2EE-CD protects the content of the document, it does not necessarily protect metadata, such as the identities of the users collaborating on a document, the timing and frequency of edits, or the communication patterns between users. Our formalism splits this leakage into two parts, which can be addressed separately: in our formalism of §3, we allow the server to learn the patterns of edits, as well as the identities of the users collaborating on a document; in SignalCD, the server also learns the size of the document when a user downloads or shares it, as well as the size of edits when a user edits the document.

In order to address the former, an E2EE-CD system can be built on top of an anonymous communication channel, such as Tor or Nym, which hides the identities of the users collaborating on a document. To address the timing and frequency of edits, the system can use cover traffic, where users send dummy edits at random times, in order to hide the real edits. This would prevent the server from learning when and how often users edit the document.

In order to address the latter, an E2EE-CD system needs to implement padding countermeasures (e.g., always padding to a fixed size, or padding to a randomized size).

Metadata-hiding is a complex topic, and the exact implementation details of such countermeasures can drastically affect the performance and usability of the system. We leave a more detailed exploration of metadata-hiding for E2EE-CD to future work. In this work, we are satisfied with a similar leakage profile as Signal's group messaging protocol, which investigative journalists already use for communication.

C Additional plots

We show staggered latencies for two users concurrently editing the same document and sending character-by-character updates in Fig. 10, for the `slow` network configuration, and for a two-second interval of a longer editing session. These results shows that SignalCD enables real-time collaborative editing: if user $\mathcal{U}_1$ types a character, $\mathcal{U}_2$ will have almost always applied the edit before $\mathcal{U}_1$ types the next character. Further, if both $\mathcal{U}_1$ and $\mathcal{U}_2$ happen to perform a character edit at the same time (as happens shortly after the 5000 ms mark in Fig. 10), both their documents will be consistent before either user types the next character.

D Modelling CryptPad

CryptPad [1] is an end-to-end-encrypted office suite. The protocol underlying CryptPad is described in a 2022 whitepaper [25]. The threat model of CryptPad assumes honest-but-curious users and an actively malicious adversary; the server is assumed to be honest-but-curious for the purpose of delivering the code which users run to interact with the protocol. In the following, we describe how CryptPad can be cast in our framework. Specifically, we discuss how CryptPad instantiates an E2EE-AB channel; the construction of E2EE-CD from E2EE-AB is then similar to our generic construction from §5.

New. When a user creates a new document, they generate a fresh symmetric encryption key $\mathsf{k}_{\mathsf{ENC}}$ and a fresh signing key pair $(\mathsf{vk}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{SIG}})$, using a dedicated key derivation procedure applied to a password and seed. The user also derives a random broadcast channel identifier. For admin permissions, additional signing-verification key pairs are generated.

Sharing. In order to grant another user read (resp. write) access to the broadcast, an existing user needs to share $\mathsf{k}_{\mathsf{ENC}}$ and $\mathsf{vk}_{\mathsf{SIG}}$ (resp., $\mathsf{k}_{\mathsf{ENC}}$, $\mathsf{vk}_{\mathsf{SIG}}$, and $\mathsf{sk}_{\mathsf{SIG}}$) with them. This is done either by sending the keys through CryptPad internal encrypted messaging system, or by using an out-of-band channel. In order to grant admin permissions, the corresponding admin signing key is also shared.

Editing. In order to send a message, a user requires access to a symmetric encryption key $\mathsf{k}_{\mathsf{ENC}}$, as well as to the signing key $\mathsf{sk}_{\mathsf{SIG}}$ corresponding to a verification key $\mathsf{vk}_{\mathsf{SIG}}$, which is known to the server. To send a message m to the broadcast, the user encrypts m under $\mathsf{k}_{\mathsf{ENC}}$ and signs the resulting ciphertext with $\mathsf{sk}_{\mathsf{SIG}}$. The user then sends the resulting ciphertext-signature pair to the server, which stores it. The server then checks whether the signature is valid using $\mathsf{vk}_{\mathsf{SIG}}$, and if so, forwards the ciphertext-signature pair to all users listed as having access to the document. Each user can then decrypt the ciphertext using $\mathsf{k}_{\mathsf{ENC}}$.[8] When a user wants to read a document, they fetch all ciphertexts from the server, verify their signatures using $\mathsf{vk}_{\mathsf{SIG}}$, and decrypt them using $\mathsf{k}_{\mathsf{ENC}}$.

D.1 Parameters

We discuss how CryptPad fits in our framework, in terms of access control, leakage, and reconciliation mechanism.

Reconciliation mechanism. CryptPad uses a custom reconciliation mechanism based on operational transformation [27]. The CryptPad whitepaper does not provide a formal description of the reconciliation mechanism, nor does it prove that it satisfies the convergence property (Def. 1). However, the whitepaper does state that "CryptPad's real-time collaborative editing is based on [OT]" and cites [27], which proves convergence for their OT algorithm. We thus assume that CryptPad's reconciliation mechanism satisfies Def. 1, although we note that many OT algorithms are notoriously difficult to prove convergent, and that many algorithms in the literature have later been shown to be incorrect [15, 23, 29].

Access control. CryptPad (respectively, its underlying broadcast channel) supports three permission levels: read, write, and admin. Admin users can add or remove users with read or write permissions; write users can edit the document; read users can only read the document. Thus, $\mathsf{PermSet} = \{\bot, \mathsf{read}, \mathsf{write}, \mathsf{admin}\}$.

Leakage. The asynchronous broadcast channel underlying CryptPad has the following leakage functions:

$$
\begin{aligned}
&\mathcal{L}_{\mathsf{AB.new}}(\mathsf{gid}, \mathsf{uid}) && = \bot \ , \\
&\mathcal{L}_{\mathsf{AB.auth}}(\mathsf{gid}, \mathsf{uid}) && = \bot \ , \\
&\mathcal{L}_{\mathsf{AB.add}}(\mathsf{gid}, \mathsf{uid}, \mathsf{uid}', \mathsf{perm}, \mathsf{msg} = \bot) && = \bot \ , \\
&\mathcal{L}_{\mathsf{AB.rcv}}(\mathsf{gid}, \mathsf{uid}, \mathsf{msg}) && = |\mathsf{msg}| \ , \\
&\mathcal{L}_{\mathsf{AB.send}}(\mathsf{gid}, \mathsf{uid}, \mathsf{msg}) && = |\mathsf{msg}| \ .
\end{aligned}
$$

Since we have no description of the listing functionality in CryptPad, we do not know what $\mathcal{L}_{\mathsf{AB.list}}$ should be. These leakage functions are very similar to those of SignalCD (§6), except that CryptPad does not leak the index of users in the access control list, nor any information related to epochs, as CryptPad has no notion of either. Further, CryptPad implements a snapshotting functionality: periodically, users will save a snapshot of the current document state (and publicly mark it as such), and store this snapshot on the server. This

[8] We note here that for security, the users should re-verify the signature themselves, rather than trusting the server to do so. However, the CryptPad whitepaper states that "The separation of encrypting and signing further allows outsourcing the validation [...] to the server as it can have $\mathsf{vk}_{\mathsf{SIG}}$, but not $\mathsf{k}_{\mathsf{ENC}}$."

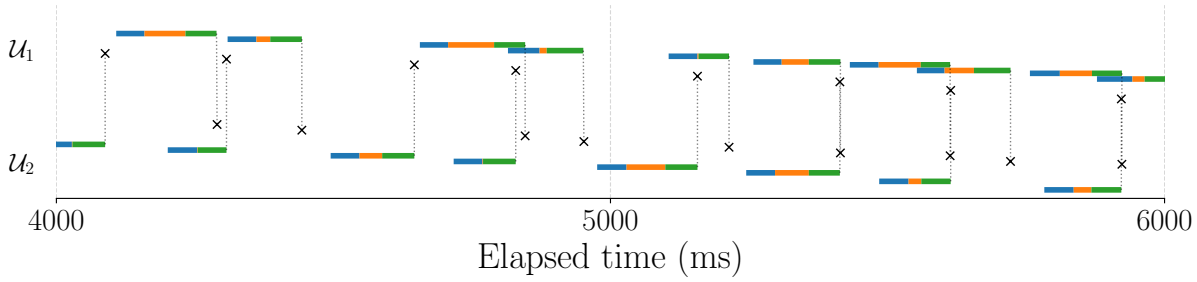

Figure 10: Staggered latencies for concurrent edits over a two-second interval (part of a longer editing session). Horizontal lines show the time from encrypting and dispatching the sending request until reception by the Signal server (blue), followed by the time until the Signal server delivers the message to the recipient (orange), followed by the time until the recipient's client application receives the message (green); the black cross marks the point at which the other user has applied the edit using Rec.

allows new users to download the latest snapshot, and then only fetch edits made since the snapshot, rather than having to fetch all edits ever made to the document. This snapshotting functionality additionally leaks the size of the document at the time of the snapshot, as well as the time at which the snapshot was taken; in our framework, this additional leakage is captured by setting $\mathcal{L}_{\mathsf{CD.edit}}$ to additionally leak the size of the document $|\mathsf{doc}|$ for edits leading to a snapshot.

Rollback. Revocation of permissions in CryptPad is done by generating a fresh encryption key $\mathsf{k}'_{\mathsf{ENC}}$ and re-sharing it with all users except the revoked user. This requires users to either have separate out-of-band channels to share the new key, or to use CryptPad's internal encrypted messaging system. Assuming that either of these channels are secure against a malicious server, the server cannot perform rollback attacks. Thus, we can set $\mathsf{Rollback} = 1$ and allow the server to attempt such rollback attacks, as they will not endanger the security of the final E2EE-CD protocol. We note that a similar technique of tearing down the group, and recreating a new one with the updated permissions could be applied in SignalCD.

E E2EE-CD from MLS

The Messaging Layer Security (MLS) protocol [10] is a recently standardized protocol for secure group messaging, relying on group key agreement. MLS provides an efficient, asynchronous, end-to-end encrypted broadcast channel with strong forward secrecy and post-compromise security properties.

Threat model. The threat model of MLS assumes an actively malicious adversary. Whereas the Signal group messaging protocol relies on the server to store the (encrypted) group membership table, MLS clients store the group membership table themselves, and the server relays messages. Similarly to our instiantion of E2EE-AB from the Signal group messaging protocol (§6), we can instantiate an E2EE-AB channel from MLS, and then use our generic construction from §5 to obtain an E2EE-CD system. We discuss the parameters of such a system below.

Access control. The base MLS protocol supports two permission levels: read and write. Write users can send messages to the group; read users can only receive messages. However, MLS extensions also supports admin permissions, which grant users the ability to manage the group membership [8].

Leakage. An E2EE-AB instantiated from an MLS protocol leaks the following information:

$$\begin{aligned}
&\mathcal{L}_{\mathsf{AB.new}}(\mathsf{gid},\mathsf{uid}) &&= \bot\ ,\\
&\mathcal{L}_{\mathsf{AB.auth}}(\mathsf{gid},\mathsf{uid}) &&= \bot\ ,\\
&\mathcal{L}_{\mathsf{AB.add}}(\mathsf{gid},\mathsf{uid},\mathsf{uid}',\mathsf{perm}) &&= \bot\ ,\\
&\mathcal{L}_{\mathsf{AB.rcv}}(\mathsf{gid},\mathsf{uid},\mathsf{msg}) &&= |\mathsf{msg}|\ ,\\
&\mathcal{L}_{\mathsf{AB.send}}(\mathsf{gid},\mathsf{uid},\mathsf{msg}) &&= |\mathsf{msg}|,\mathsf{epoch_id}\ .
\end{aligned}$$

Rollback. In contrast to the Signal group messaging protocol, MLS is designed to prevent rollback attacks by a malicious server [10]. In MLS, group membership and keys are derived using the TreeSync structure, where every update is signed and consistent across all members. Further, MLS handshake messages cryptographically bind the group state to the entire history of group operations. Therefore, any rollback attempt by the server will be detected by the clients. In our model, we can thus set $\mathsf{Rollback} = 1$ and allow the server to attempt such rollback attacks, as they will not endanger the security of the final E2EE-CD protocol.