Stephan A. Kollmann*, Martin Kleppmann, and Alastair R. Beresford

Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing

Abstract: Document collaboration applications, such as Google Docs or Microsoft Office Online, need to ensure that all collaborators have a consistent view of the shared document, and usually achieve this by relying on a trusted server. Other existing approaches that do not rely on a trusted third party assume that all collaborating devices are trusted. In particular, when inviting a new collaborator to a group, one needs to choose between a) keeping past edits private and sending only the latest state (a snapshot) of the document; or b) allowing the new collaborator to verify her view of the document is consistent with other honest devices by sending the full history of (signed) edits. We present a new protocol which allows an authenticated snapshot to be sent to new collaborators while both hiding the past editing history, and allowing them to verify consistency. We evaluate the costs of the protocol by emulating the editing history of 270 Wikipedia pages; 99% of insert operations were processed within 11.0 ms; 64.9 ms for delete operations. An additional benefit of authenticated snapshots is a median 84% reduction in the amount of data sent to a new collaborator compared to a basic protocol that transfers a full edit history.

Keywords: collaborative editing, authenticated data structures, snapshots, privacy, security, integrity

DOI Editor to enter DOI
Received ..; revised ..; accepted ...

1 Introduction

Document collaboration applications, such as Google Docs, allow several users, each using one or more devices, to concurrently edit a shared document or dataset without merge conflicts. Such systems maintain a local copy of the document on every collaborating user's device. Whenever the user makes an edit on a device, these changes are first applied to the local copy of the document, and then an edit message containing the change is sent to a central server. When the server receives document edit messages, it modifies these messages to assist devices in resolving potential conflicting concurrent edits, and propagates these updated messages to the devices used by all collaborating users [8, 26].

While the communication between user devices and the server can be encrypted in such systems, e.g. using TLS, the server itself must be able to read and modify messages, and therefore it must be trusted to maintain the confidentiality and integrity of a cleartext copy of the document. For this reason, the current generation of collaborative document editing systems do not support end-to-end encryption between user devices.

Such systems require that users fully trust the service provider and its employees; this assumption is problematic in countries where service providers are obliged to hand over data to law enforcement and intelligence agencies without appropriate judicial oversight. In addition, a data breach at the service provider will reveal the contents of all documents. For certain groups of people who deal with sensitive data, such as medical professionals, lawyers, journalists, diplomats, engineers, activists, and others, this trust model is undesirable for legal, ethical, business, or personal safety reasons.

In this paper we describe how to build a document collaboration system that supports concurrent edits by several user devices, and that does not require a server to receive and centrally process all edits.

Without a trusted server to authenticate clients, user devices need a means of verifying the integrity and authenticity of operations they receive from other users. For example, consider a scenario where lawyers from different companies are negotiating a contract using a shared document editor. It is essential that everyone has a consistent view of the document and that changes can be traced back to the author. In §4 we present a simple protocol that uses digital signatures and cryptographic hashes to verify the origin of an operation, and to efficiently check if all devices have seen the same set of

***Corresponding Author: Stephan A. Kollmann:** University of Cambridge, E-mail: sak70@cl.cam.ac.uk
Martin Kleppmann: University of Cambridge, E-mail: mk428@cl.cam.ac.uk
Alastair R. Beresford: University of Cambridge, E-mail: arb33@cl.cam.ac.uk

document edits. This protocol is sufficient if the set of participants in a collaboration group is fixed.

1.1 Adding a new collaborator

Further challenges arise if new collaborators may be invited to join the editing session for an existing document. In this case, the new collaborator must be given a copy of the document at the time she is invited, and then be sent any subsequent edits to the document. We identify three approaches to inviting new collaborators:

1. If the existing collaborators keep a log of all editing operations that have occurred since the creation of the document, they can send a copy of that log to the new collaborator, who can then reconstruct the current state of the document from the edits. The new collaborator can also use the hashes and signatures on the operations to verify the integrity of the edit log. This approach is used by the protocol in §4, but it has significant disadvantages. In particular, storing, transmitting, replaying, and checking the integrity of the edit log incurs substantial costs in storage, network bandwidth, and processing time; and the edit log contains all past versions of the document, including any text that has been deleted in the current version.
2. To reduce the cost and improve the privacy properties of the first approach, the new collaborator could be sent only a *snapshot* of the current state of the document, not including its past editing history. To ensure consistency, each existing participant can be asked to confirm the validity of the snapshot. However, if any existing participants are offline, the new participant must either wait (potentially indefinitely) until they are next online, or go ahead and accept the risk that its snapshot is inconsistent with other participants' view of the document.
3. To overcome the downsides of the first two approaches, we develop a new protocol in §5. In this protocol, new collaborators are only sent a snapshot of the current state of the document, plus a cryptographic proof of the integrity of the snapshot. The new collaborator can then use this proof to verify the integrity of the snapshot, without having to wait for any communication with other participants.

1.2 Contributions of this paper

The contributions of this paper are as follows:

- We propose a scheme for cryptographically verifying the consistency of a shared text document between collaborators without relying on a trusted server. Furthermore, the scheme allows devices to be invited to a group of collaborators by sending new devices a snapshot of the shared document; this snapshot does not contain any deleted text or the editing history, and therefore has better privacy and scalability properties than a naive solution.
- In §5 we propose a scalable implementation of the scheme based on RSA accumulators and Merkle trees, and we prove (in the appendix) that our proposed protocol satisfies the required security and consistency properties.
- We evaluate the practicality of a prototype implementation using the editing history of Wikipedia articles. In our experiments, 99% of insert operations were processed within 11.0 milliseconds, and within 64.9 milliseconds for delete operations. We further achieved a median 84% reduction in the amount of data that needs to be transferred to a new collaborator by using authenticated snapshots (§5) compared to a basic protocol (§4) that transfers the full editing history.
- We propose a number of optimizations to reduce the constant factor overhead of both computation and communication in our prototype implementation.

2 Background

2.1 Conflict-free Replicated Data Types

We make use of operation-based Conflict-free Replicated Data Types (CRDTs) [33] in order to ensure that concurrent edits to a document can be merged by user devices without conflicts. We have chosen CRDTs because they do not need a central server – data can flow directly between devices. CRDTs do not need conflict resolution or transformation of operations, because operations are designed such that concurrent operations can be applied conflict-free in any order. In the context of CRDTs we call each device a *replica*, since it has a copy of the shared document. Updates at a replica are applied locally immediately without any synchronization, and are broadcast asynchronously to other replicas. All replicas converge to the same state provided

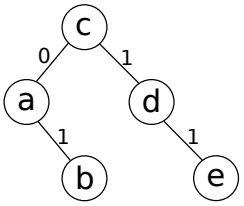


Fig. 1. Possible tree representation of the string "abcde" as used in Treedoc.

that they receive all updates eventually, a consistency model known as *strong eventual consistency* [33]. Unlike consensus protocols, CRDTs remain available under arbitrary network partitions (in the sense of the CAP theorem [3, 14]), so they are able to support offline editing.

Treedoc

Treedoc [28] is an example of an operation-based CRDT for collaborative editing of text documents. The protocols we present in §4 and §5 use Treedoc for its simplicity, but would also work with other CRDTs.

Treedoc models a shared document as a sequence of *atoms*. An atom is the smallest unit of content supported by the editor, e.g. a character of text. The basic idea of Treedoc is to assign a unique position identifier *pos* to every atom. Position identifiers are totally ordered, such that the total order is consistent with the order of the atoms in the document. Additionally, the space of position identifiers is dense, i.e. for any position identifiers pos_1 and pos_2, one can create a new position identifier pos_{new} with the property $pos_1 < pos_{\mathsf{new}} < pos_2$. Treedoc allows two operations:

- insert(pos, $atom$): Inserts the new atom $atom$ into the document at position pos.
- delete(pos): Removes the atom at position pos. For the operation to be valid, such an atom must exist in the state of the device that initiates the operation.

In Treedoc, position identifiers are defined to be paths in a binary tree. For example, Figure 1 shows a possible representation of the string "abcde", in which the path for character 'e' is "11". The order of atoms in the document is given by an infix-order depth-first traversal of the tree. When inserting a new atom, a new position identifier is generated by creating a suitable descendant of the node to the left or right of the desired insertion position. However, this alone is not enough to guarantee uniqueness of the identifier, since more than one user can perform insertions concurrently. To solve this, Treedoc attaches a *disambiguator* to each node. We define disambiguators to be (ctr, $replicaID$) pairs, where $replicaID$ is the unique identifier of a replica, and ctr is a per-replica counter.

2.2 RSA Accumulators

A cryptographic *accumulator* allows a finite set $\mathcal{X}$ to be accumulated and represented by a single, constant-sized value, $acc_{\mathcal{X}}$. For every element $s \in \mathcal{X}$, one can efficiently compute a *witness* $w_{s \in \mathcal{X}}$ that can be presented to prove the *membership* of s with regards to $acc_{\mathcal{X}}$, i.e. proving that s is part of the set accumulated in $acc_{\mathcal{X}}$. However, it is computationally infeasible to compute a witness for an element $x \notin \mathcal{X}$ (*collision-freedom*).

RSA accumulators [1, 2] are based on the hardness of the RSA problem. An RSA accumulator requires an RSA secret key consisting of two safe primes p and q, and a base value x that is drawn randomly from the cyclic group of quadratic residues modulo N, where $N = pq$ is the RSA modulus [9]. In the elementary form of the accumulator, the accumulator value is calculated as:

$$acc_{\mathcal{X}} = x^{\prod_{a \in \mathcal{X}} a} \mod N. \tag{1}$$

Due to the multiplications in the exponent, the elements to be accumulated are restricted to prime numbers for collision-freedom.

To remove the restriction to prime numbers, Barić and Pfitzmann proposed a variant of the accumulator that uses *prime representatives* [1]. They construct prime representatives as follows. The prime representative $h(a)$ for an element a is computed as $h(a) = 2^l \Omega(a) + d$, where $\Omega(a)$ is a random oracle (in practice replaced by a secure hash function), l is suitably large, and d is an l-bit number chosen such that $h(a)$ is a prime. In other words, one appends l low-order bits to $\Omega(a)$ such that the result becomes prime. A suitable d can be found by using a standard primality test [29] and trying all odd d starting from 1.

To show that an element $b \in \mathcal{X}$ is accumulated in $acc_{\mathcal{X}}$, one presents a witness $w_{b \in \mathcal{X}}$ computed as:

$$w_{b \in \mathcal{X}} = x^{\prod_{a \in \mathcal{X} - b} a} \mod N. \tag{2}$$

Thus, a witness for an element is equal to the accumulator value for all the remaining accumulated elements. To verify the correctness of the witness, one checks:

$$w_{b \in \mathcal{X}}^{b} = acc_{\mathcal{X}} \mod N. \tag{3}$$

In addition to memberships proofs for individual elements, RSA accumulators allow us to prove a subset

relationship $S \subseteq \mathcal{X}$ with a single witness. This witness is computed by accumulating all remaining elements, and verification works accordingly:

$$w_{S\subseteq\mathcal{X}} = x^{\prod_{a\in\mathcal{X}\setminus S} a} \mod N, \tag{4}$$

$$w_{S\subseteq\mathcal{X}}^{\prod_{a\in S} a} = acc_{\mathcal{X}} \mod N. \tag{5}$$

Note that computing a witness does not require knowledge of the secret key; it requires only x, N, and the accumulated set $\mathcal{X}$.

3 System architecture

We envision a collaborative document editing system with an arbitrary number of users, each of whom may own one or more devices. Document editing software is installed on each device, allowing the user to create a new document, invite others to collaborate on a document, and join an existing document. The software allows users to edit any document regardless of whether they are currently connected to a network or not; if no network connection is available, then document changes are applied locally and sent to peers when network connectivity returns. Modelling typical mobile devices, we assume that devices may frequently be offline, and that devices may suffer a permanent failure without warning, e.g. if dropped in water.

3.1 Threat model and design goals

We assume the adversary is able to control network communication and can read, modify and delay any traffic, including partitioning the network for arbitrary periods of time. Further, we assume the adversary can create an arbitrary number of fake users with devices that may participate in group collaboration; these devices may deviate arbitrarily from the protocol.

We assume that an existing key exchange and encryption protocol protects the confidentiality of messages sent via the network. In addition, we assume a public key infrastructure through which collaborators are able to find each others' public keys. The adversary cannot compromise the public-key infrastructure and does not have access to secret keys of honest participants; therefore, the adversary cannot forge messages or signatures created by honest participants.

On top of this infrastructure, our protocol provides the following properties in the face of the adversary:

Edit integrity. The shared document can only be modified by a group member.

Attributability. All edits are attributable to the honest device that made the modification. Group members can identify who added a certain part of a document, even if it was added before they joined.

Consistency. Devices have consistent views of the document. When an honest device processes an edit operation, it must have previously processed exactly those edits that happened before this operation, and possibly some concurrent edits.

Snapshot consistency. On joining a group, a new member can check the integrity of the document, i.e. they can verify that the state is consistent with states seen by other collaborators. In particular, they can verify that all modifications made or seen by collaborators up to a certain point are represented in the snapshot, and that no modifications were falsely attributed to a collaborator.

Edit history privacy. A new group member cannot see edits made before she joined the group, other than what can be inferred from the document state when she joins; in particular, she cannot see parts of the document that were deleted before she joined.

Convergence. When honest group members communicate, their local copies of the shared data converge towards a consistent state, even if arbitrarily many group members are malicious.

Availability. Any two participants can collaborate on a document, even if all other collaborators are offline; in particular, the protocol does not require any quorum of devices to be reachable.

Scalability. Assuming a bounded number of collaborators, protocol messages add only a constant size overhead compared to a simple protocol that does not allow authenticated snapshots; communication and computational overhead for inviting a new member, sending and processing a snapshot is practically linear in the number of atoms in the document at the time of the snapshot.

We prove in the appendix that the protocol described in §5 satisfies these properties. The properties protect against different kinds of attacks an adversary might attempt. For example, an estate agent selling a house could try to present different views of a contract to different parties, showing different sale prices and keeping the difference. In a collaborative code editor, an attacker may want to insert malicious code and attribute it to someone else.

Edit history privacy allows, for example, lawyers to collaborate on a contract and later share it with a third party, while ensuring that the third party is unable to see potentially sensitive contents of any previous versions of the contract. Edit history privacy is also useful when researchers working on a paper want to share a draft with a colleague, but would prefer not to reveal previous unpolished versions of the paper.

4 Basic protocol

In this section, we propose a basic protocol for collaborative editing of a text document that relies on all collaborators having a copy of the full editing history of the document. In §5 we will show how to improve the protocol's privacy properties so that new collaborators can be given a snapshot containing only the current document state, and not the past editing history.

The document is initially created on one particular device, and any existing device can add a new collaborating device using an addDevice operation (see §4.3). We assume that each device is identified by a unique device identifier, *deviceID*, which may for example be a hash of its public key.

Following the Treedoc algorithm [28], we represent a collaboratively editable text document by a set of *atoms*. Each atom represents an editable unit of text, for example, a character, a word, a line, or a sentence, and the metadata associated with it. The granularity of atoms can be chosen depending on the application, and does not affect the operation of the protocol.

An atom is a 4-tuple (pos, src, ctr, txt):

- *pos* is a variable-length bit string that identifies a position in the document as in Treedoc (§2.1).
- *src* is the *deviceID* of the source (the device on which the atom was originally created).
- *ctr* is a sequence number that is incremented by the sender as described in §4.1.
- *txt* is a text fragment (character, word, or line).

Note that an atom can be uniquely identified both by its position identifier, and by the tuple (src, ctr).

The text of the document is obtained by sorting the set of atoms in lexicographical order of the position identifier, and concatenating the associated text fragments in that order. We allow the text to be edited through two types of operation: inserting an atom, and deleting an atom. Replacement of text is expressed as deletion and subsequent insertion.

4.1 Sending messages

Collaborators communicate by sending and receiving messages. Each collaborator maintains a set of messages it has sent and received; for example, $msgs_A$ is the set of messages sent or received by A.

Each message is a 5-tuple $(src, ctr, op, deps, sig)$, constructed as follows:

- *src* is the *deviceID* of the source (the device that created the message).
- *ctr* is a sequence number that is 1 for the first message sent by a particular *src*, and incremented for each subsequent message from *src*.
- *op* is an operation: either $\mathsf{insert}(pos, text)$ to represent the insertion of a new atom, or $\mathsf{delete}(src', ctr')$ to represent the deletion of an existing atom, or $\mathsf{addDevice}(deviceID, publicKey)$ to announce the addition of a collaborator device, or noop if the document has not been changed. The noop operation is useful so a device can acknowledge that it has seen a certain state without performing any changes.
- *deps* is the set of *dependencies* of this message, that is, a reference to the most recent prior message from each device; more precisely it is a set of triples consisting of the source *deviceID*, the sequence number of the most recent message seen from that source, and the hash of that message:[1]

$$\begin{aligned} deps = \big\{ (s, c, \mathrm{h}(m)) \mid \quad & (6) \\ & m \in msgs_{src} \wedge m = (s, c, \_, \_, \_) \wedge \\ & \nexists\, c'.\ ((s, c', \_, \_, \_) \in msgs_{src} \wedge c < c') \big\}. \end{aligned}$$

The hash $\mathrm{h}(m)$ of message $m = (src, ctr, op, deps, \_)$, is computed as a cryptographic hash of the message contents (excluding the signature), and is used to check that all collaborators have received the same message contents:

$$\mathrm{h}(m) = \mathrm{H}(src \parallel ctr \parallel op \parallel deps). \quad (7)$$

$\mathrm{H}(\cdots)$ can be any secure hash function, such as SHA-256. Note that this creates a directed acyclic

1 We use the underscore as placeholder for a fresh, existentially quantified variable. For example, $(x, \_, \_) \in A$ is shorthand for $\exists\, y, z.\ (x, y, z) \in A$, and $\nexists\, (x, \_, \_) \in A$ is shorthand for $\nexists\, y, z.\ (x, y, z) \in A$.

graph of hashes, where each message references the previous message from the same device and any messages received from other devices. This hash-DAG is similar to the commit history in the Git version control system.

– *sig* is a digital signature of the preceding elements of the message tuple, using the private key of the sender *src*:

$$sig = \text{sign}_{src}(docID \parallel src \parallel ctr \parallel op \parallel deps), \tag{8}$$

where *docID* is a document identifier that uniquely identifies the document. We assume that the document identifier is known to all participants, e.g. through the messaging protocol.

When the source device *src* sends a message m, it adds the message to its message set:

$$msgs'_{src} = msgs_{src} \cup \{m\}. \tag{9}$$

m is sent to the other collaborators using a secure messaging protocol, which we elide in this description. Any protocol that protects the confidentiality and integrity of the message against network attackers can be used.

4.2 Receiving messages

When a message $m = (src, ctr, op, deps, sig)$ is received by a destination device *dst*, the destination device performs the following checks:

1. There is no existing message from the same *src* with a counter value greater than or equal to the incoming message:

$$\forall c.\ (src, c, \ ., \ ., \ .) \in msgs_{dst} \implies c < ctr. \tag{10}$$

2. The dependencies are satisfied:

$$deps \subseteq \big\{(s, c, \text{h}(m')) \mid m' \in msgs_{dst} \wedge m' = (s, c, \ ., \ ., \ .)\big\} \tag{11}$$

If $msgs_{dst}$ does not contain the dependencies because they have not yet been delivered, the message m can be buffered locally, and the destination device can request retransmission of the missing messages. Then the delivery of m can be retried after other messages have arrived. However, if the check fails because the hashes are mismatched, m must be rejected.

3. *sig* is a valid signature of $docID \parallel src \parallel ctr \parallel op \parallel deps$, checked with *src*'s public key.

If all of these checks succeed, m is added to the destination device's message set:

$$msgs'_{dst} = msgs_{dst} \cup \{m\}. \tag{12}$$

Assuming second preimage resistance of the hash function and unforgeability of the signatures, the destination device knows that $msgs_{dst} \supseteq msgs_{src}$ if the above checks succeed, since the hashes in *deps* transitively include all messages in $msgs_{src}$ at the time the message was sent.

Finally, on any device A, the set of atoms $S(msgs_A)$ that make up the document is the set of atoms that have been inserted but not deleted:

$$S(msgs) = \big\{(pos, src, ctr, txt) \mid (src, ctr, \mathsf{insert}(pos, txt), \ ., \ .) \in msgs \wedge \nexists\, (\ ., \ ., \mathsf{delete}(src, ctr), \ ., \ .) \in msgs\big\} \tag{13}$$

The text of the document is obtained by sorting this set of atoms as described in §2.1.

4.3 Adding a new collaborator

When an existing collaborator wants to add a new device as a collaborator, it first broadcasts a message containing an $\mathsf{addDevice}(deviceID, publicKey)$ operation to announce to other devices that a certain device has been added. Moreover, the device A that invites the new collaborator must send the entire set $msgs_A$ to the new device. The new device can then check the integrity of these messages by performing the same checks as in §4.2.

If A is malicious, it may try to make the new device's document diverge from the rest of the group. However, A is limited to two attacks: it can give the new device an old version of the document (corresponding to a subset of $msgs_A$), and it can give the new device a document containing edits that have not yet been sent to other collaborators. In either case, when the new collaborator communicates with other group members, they will exchange the missing operations.

5 Privacy-enhanced protocol

The protocol described in §4 has the problem that the full editing history, including any deleted past content of the document, is exposed to a new collaborator when she joins. In this section we present a revised protocol that avoids this problem.

Specifically, we want to be able to send a new collaborator a *snapshot* containing only the current set of

Variable	Description	§
$docID$	Unique identifier of shared document	4.1
src	Unique $deviceID$ of source device	4.1
ctr	Per-device message sequence number	4.1
op	Operation	4.1
pos	Position identifier	2.1
txt	Atomic text fragment	4
$deps$	List of dependencies of a message	4.1
sig	Signature of message signed by source	4.1
$\mathrm{h}(m)$	Hash of message m	4.1
$S(msgs)$	Atoms inserted but not deleted in $msgs$	4.2
r	Per-message random nonce	5.1
acc	Accumulator value of current set of atoms	5.1
T_{src}	Merkle tree of messages from device src	5.1
$T_{src}[c]$	T_{src} up to message with $ctr = c$	5.1
mh	Hash of Merkle tree roots for all devices	5.1
$msgs_d$	Set of messages processed by device d	4.1
S_d	Set of atoms in device d's document view	5.1
$sdesc$	Set of state descriptors for all devices	5.3.1
$mproofs$	Merkle consistency proofs	5.3
$mnodes$	Merkle tree nodes sent to new device	5.3
wit	Witness for atoms present in device's view	5.3.1

Table 1. Variables used in the description of the privacy-enhanced protocol.

atoms, rather than the full set of operations that led to it. However, simply changing the protocol of §4.3 to send $S(msgs_A)$ instead of $msgs_A$ removes any ability for the new collaborator to check the integrity and consistency of the document, since she cannot use the checks in §4.2. Thus, a malicious device could send the new collaborator an arbitrarily corrupted set of atoms.

To allow the new collaborator to verify that the snapshot is consistent with the sets of atoms on other devices, we use RSA accumulators [1, 2] as described in §2.2. Each device generates an RSA key $N = pq$ and makes the modulus N public. The accumulator base x can be fixed.

Devices use those accumulators to attest to their current state (set of atoms) with every message they send. When a new device is added, the device sending the invitation provides a snapshot of the document containing the latest signed accumulator from each device, and cryptographic proofs that the accumulated sets of atoms are consistent with each other. To ensure that the signed accumulators from different devices correspond to states in a consistent edit history, the snapshot also includes a set of appropriate Merkle tree consistency proofs. Each message additionally contains a hash over all messages processed by the sender so far. The remainder of this section will explain these constructions in more detail.

Since the following protocol description contains a considerable number of variables, for reference, Table 1 contains a list of the variables used, with a short description and the section where the variable is defined.

5.1 Sending messages

We update the definition of a message in §4.1 by adding three additional elements: a nonce r, an accumulator acc, and a hash mh. In our revised definition, a message is an 8-tuple $(src, ctr, op, deps, r, acc, mh, sig)$:

- src, ctr, op, and $deps$ are defined as in §4.1.
- r is a 128-bit random prime.
- acc is the value of an RSA accumulator over the current set of atoms $S_{src} = S(msgs_{src})$, which is derived from $msgs_{src}$ as shown in (13), and r:

$$acc(S_{src}, r) = x_{src}^{P(S_{src})r} \mod N_{src}, \tag{14}$$

$$\text{where} \quad P(S) = \prod_{a \in S} \text{prime}(a). \tag{15}$$

  The function $\text{prime}(a)$ is a hash function that returns only prime numbers, as described in §2.2. We accumulate r in addition to the set of atoms to make the accumulator indistinguishable, i.e. to prevent a new collaborator guessing the accumulated set based on the accumulator value and therefore learning about deleted atoms [9]. The nonce is sent to current collaborators, since it is required for the witness calculation in (22), but it is omitted from snapshots sent to new collaborators, as described in §5.3. The accumulator can be maintained incrementally, so it does not need to be recalculated from scratch for every message sent.
- mh is the hash of a set of Merkle trees, defined as follows. Let T_s be a Merkle tree [23] containing all message hashes received from device s in order of their sequence number (including the current message if s is the sending device src), and let $\text{MTH}(T_s)$ be the Merkle Tree Hash of T_s. Then

$$mh = \mathrm{H}\Big(\big\{\text{MTH}(T_s) \mid s \text{ is a } deviceID\big\}\Big). \tag{16}$$

  In §5.3.3 we use this construction to prove that the sequence of messages from a particular sending device is an append-only sequence, following the approach of Certificate Transparency [10, 20]. To ensure that mh is unique, the elements of the set are hashed in a fixed order, e.g. in lexicographic order of $deviceID$s.

– *sig* is extended to also cover the accumulator and the Merkle Tree Hash. Moreover, instead of *op*, we include $\mathrm{h}(m)$ in the data to be signed:

$$sig = \mathrm{sign}_{src}(docID \,\|\, src \,\|\, ctr \,\|\, \mathrm{h}(m) \,\|\, deps \,\|\, acc \,\|\, mh). \tag{17}$$

This construction allows a new collaborator to verify the signature of a partial message without necessarily knowing the operation contained in the message. The hash of the message, $\mathrm{h}(m)$, is extended by the nonce and accumulator:

$$\begin{aligned} &\mathrm{h}(m) = \mathrm{H}(src \parallel ctr \parallel op \parallel deps \parallel r \parallel acc), \\ &\text{where} \quad m = (src, ctr, op, deps, r, acc, \ .). \end{aligned} \tag{18}$$

5.2 Receiving messages

When a message $m = (src, ctr, op, deps, r, acc, mh, sig)$ is received by a device *dst*, it first performs the same checks as in §4.2.

Next, to validate the accumulator *acc*, *dst* computes the set of atoms that existed on the source device *src* at the time m was sent. To this end, we first find the subset of messages in $msgs_{dst}$ that are referenced in the message dependencies *deps*:

$$\begin{aligned} \mathrm{msgsIn}(deps) = \big\{ &(s, c, \ ., \ ., \ ., \ ., \ ., \ .) \in msgs_{dst} \mid \\ &\exists\, c'.\ (s, c', \ .) \in deps \wedge c \le c' \big\} \end{aligned} \tag{19}$$

As defined in (13), the set of atoms at the time m was sent is the set of atoms that were inserted but not deleted in the set of messages $\mathrm{msgsIn}(deps) \cup \{m\}$. Thus, *dst* can check that the accumulator satisfies:

$$acc \stackrel{?}{=} x_{src}^{P(S(\mathrm{msgsIn}(deps) \cup \{m\}))r} \mod N_{src}. \tag{20}$$

If *dst* has already verified the message hash $\mathrm{h}(m)$, this check is redundant and only serves to verify that *src* has calculated *acc* correctly. However, *dst* can only verify the hash if it can compute the hashes of all dependencies, which may not be the case if it does not know the full operation history because it has joined from a snapshot (as described in §5.3). If any dependencies of a message predate (or happened concurrently to) the snapshot from which *dst* was initialised, verifying the accumulator allows *dst* to check that the sender's state is consistent with its own.

If *dst* has already verified an earlier accumulator acc_{old} (with corresponding nonce r_{old}) from *src*, it can compute the new accumulator incrementally.

Lastly, the destination device verifies that *mh* has been computed correctly by recomputing the value based on its own operation hash trees.

5.3 Adding a new collaborator

Similarly to the process in §4.3, the device sending the invitation first broadcasts a message containing an addDevice(*deviceID*, *publicKey*) operation to the existing collaborators, where the public key now also contains the accumulator RSA modulus of the device. Next, the collaborator A who invites the new device sends a *snapshot* to the new device. The snapshot is a 4-tuple $(S_A, sdesc, mproofs, mnodes)$, where $S_A = S(msgs_A)$ is A's current set of atoms, as defined in (13). We show in §5.3.1 how *sdesc* is constructed and checked, and we discuss *mproofs* and *mnodes* in §5.3.3. Using the snapshot, a new device B can start collaborating from the current state, but does not learn contents that were added to the document earlier but deleted since then. Note that to ensure this privacy property, devices must not forward a message to devices that were added later (in the dependency graph) than the message. After B has received a snapshot, it immediately broadcasts a message containing a noop operation. The accumulator value of this message allows other devices to verify that B has received a set of atoms consistent with their own. Because B cannot verify the hashes of dependent messages that happened before or concurrently to a snapshot, it must only accept messages that happened after its noop message. If any messages happened concurrently to the snapshot, B must request a new snapshot that also contains the effects of these concurrent messages. Whether a message was created logically before, after, or concurrently to another message, can be determined straightforwardly based on the sequence numbers in the message and its dependencies.

5.3.1 State descriptors

sdesc is the set of *state descriptors*, one for each of the existing collaborators. A state descriptor is an 8-tuple $(src, ctr, hash, deps, acc, mh, sig, wit)$, where the first seven elements are taken from the most recent message sent by *src*:

$$\begin{aligned} sdesc = \big\{ &(src, \ ctr, \mathrm{h}(m), deps, acc, mh, sig, wit) \mid \\ &m \in msgs_A \wedge \\ &m = (src, ctr, op, deps, r, acc, mh, sig) \wedge \\ &wit = \mathrm{witness}(S(\mathrm{msgsIn}(deps) \cup \{m\}), r) \wedge \\ &\nexists\, c'.\ ctr < c' \wedge (src, c', \ ., \ ., \ ., \ ., \ ., \ .) \in msgs_A \big\} \end{aligned} \tag{21}$$

The last element, *wit*, is a *witness* that cryptographically proves the relationship between *acc* (the accumu-

lator from *src*) and S_A (the current set of atoms):

$$\text{witness}(S_{src}, r) = x_{src}^{P(S_{src} - S_A) \cdot r} \mod N_{src} \tag{22}$$

Using (19), let $S_{src} = S(\text{msgsIn}(deps) \cup \{m\})$ be the set of atoms in the document at the time when m, the most recent message from *src*, was sent. S_{src} may reflect the state of the document at some point arbitrarily far in the past, depending on the time when *src* was last active. If A knows the full message history, S_{src} is known to A. We will consider the case where A has only seen a partial history in §5.3.2. The newly invited collaborator, however, does not know S_{src} for any $src \neq A$, since the snapshot contains only S_A, the current set of atoms on device A.

In the intervening time between state S_{src} and the current state S_A, atoms may have been added or removed. The set $S_{src} - S_A$ in the exponent of (22) contains exactly those atoms that have been removed.

When device B receives a snapshot from device A, it performs the following steps to verify S_A and *sdesc*:

1. For each atom $(pos, src, ctr, txt) \in S_A$, verify that:
   (a) The pair (src, ctr) is unique:

$$\forall p, t.\ (p, src, ctr, t) \in S_A \implies p = pos \wedge t = txt. \tag{23}$$

   (b) The atom's *ctr* is contained in the state descriptor for device *src*:

$$\exists\, c'.\ (src, c',\ .,\ .,\ .,\ .,\ .,\ .) \in sdesc \wedge\ ctr \leq c'. \tag{24}$$

   (c) The atom's *ctr* is contained in *deps* in A's own state descriptor:

$$\forall deps.\ (A,\ .,\ .,\ deps,\ .,\ .,\ .) \in sdesc \implies \exists\, c'.\ (src, c',\ .) \in deps \wedge\ ctr \leq c'. \tag{25}$$

2. For each state descriptor tuple in the set *sdesc*, i.e. for $(src, ctr, hash, deps, acc, mh, sig, wit) \in sdesc$:
   (a) Verify that *sig* is a valid signature of $docID \parallel src \parallel ctr \parallel hash \parallel deps \parallel acc \parallel mh$, checked with *src*'s public key.
   (b) Find the subset of atoms in S_A that already existed in S_{src}. Although the set S_{src} is not known to the newly invited device B, the intersection $S_{src} \cap S_A$ can be computed from *src*'s state descriptor:

$$S_{src} \cap S_A = \big\{(p, s, c, t) \in S_A \mid (s = src \wedge c \leq ctr) \vee (s \neq src \wedge \exists\, c'.\ (s, c',\ .) \in deps \wedge\ c \leq c')\big\} \tag{26}$$

   We can then use *wit* to verify that the computed set $S_{src} \cap S_A$ is indeed a subset of S_{src}:

$$wit^{P(S_{src} \cap S_A)} \stackrel{?}{=} acc \mod N_{src}. \tag{27}$$

   If the snapshot is correct, the exponent from (22), $P(S_{src} - S_A) \cdot r$, is multiplied with the exponent $P(S_{src} \cap S_A)$ from (27), yielding $P(S_{src}) \cdot r$ as in the accumulator definition (14).
   (c) Check that *ctr* is the most recent sequence number seen from *src*:

$$\forall d.\ (\ .,\ .,\ .,d,\ .,\ .,\ .) \in sdesc \implies \forall c.\ (src, c,\ .) \in d \implies c \leq ctr. \tag{28}$$

   (d) Ensure that there is a state descriptor for every device in *deps*:

$$\forall s.\ (s,\ .,\ .) \in deps \implies (s,\ .,\ .,\ .,\ .,\ .,\ .) \in sdesc. \tag{29}$$

If any of the above checks fail, the snapshot must be rejected.

5.3.2 Computing witnesses incrementally

The above discussion, especially (21) and (22), assumes that the device A that sends the snapshot has access to the full message history since the creation of the document. In general, this may not be the case, since A might itself be a device that was invited by snapshot.

However, the approach above easily generalises to the case where A starts from a snapshot. In particular, the witness computation in (22) can be performed incrementally without knowledge of S_{src}. Due to space constraints we omit a detailed discussion of the iterative witness computation.

5.3.3 Merkle tree consistency proofs

The third element of the snapshot, *mproofs*, serves as a cryptographic proof that there has not been a *fork* in the editing history of the document. A fork occurs if a device presents different and contradictory edits with the same sequence number to its collaborators.

In the basic protocol of §4, the message hashes in *deps* serve the purpose of detecting forks. In the privacy-enhanced protocol of §5, the full message history is not available to a newly invited collaborator, so we instead use Merkle trees to prove that there is no fork among the state descriptors in *sdesc*.

As described in §5.1, each device keeps track of the sequence of messages it has received from each other device – an append-only log per source device. Following the approach of Certificate Transparency [7, 20], we encode that log in a Merkle tree. If no fork has occurred, each device will see the same sequence of messages from each source device. However, since devices may be offline, some devices may have an incomplete view of other devices' message logs. In those cases, we expect the message log on one device to be a prefix of the corresponding logs on other devices.

We use Merkle consistency proofs to show that the per-source message sequence on one device is a prefix of the corresponding message sequence on another device, without revealing the actual messages. For each originating device *src*, *mproofs* contains a set of Merkle consistency proofs as follows. Let $T_{src}[c]$ be the Merkle tree containing the first c messages from *src*. Let c_d be the sequence number of the last message from *src* seen by d at the time of the message corresponding to d's state descriptor. Then $T_{src}[c_d]$ contains exactly the messages from *src* received at device d at that time.

Now consider the Merkle trees $T_{src}[c_d]$ for all devices d, sorted in increasing order by c_d, and omitting duplicate c_d. For each adjacent pair of Merkle trees in this set, *mproofs* contains a consistency proof showing that the larger tree is the same as the smaller one with some additional leaves appended. By transitivity, those proofs show the consistency of all trees for each originating device.

To check the consistency proofs, we proceed as follows. From each state descriptor, extract the sequence number of the last received message from each device *src*. Group the sequence numbers by originating device *src* and sort them in increasing order, omitting duplicates. Now, for each two adjacent sequence numbers c_1, c_2 in this list, check that *mproofs* contains a valid consistency proof between Merkle trees $T_{src}[c_1]$ and $T_{src}[c_2]$, i.e. a proof that $T_{src}[c_1]$ is a prefix of $T_{src}[c_2]$. The Merkle tree roots do not need to be included with the proofs if they can be computed from a matching consistency proof. Compute the Merkle tree roots of the trees $T_{src}[c]$ for each counter c, and using those, verify the hash over the Merkle tree roots *mh* within each state descriptor.

Lastly, *mnodes* contains a partial Merkle tree for each device, containing all nodes of the latest tree T_{src} that are required for the newly invited device to be able to extend the tree by appending leaves. For this it is sufficient to include the root of every maximal complete subtree.

6 Evaluation

In this section, we evaluate the costs of the privacy-enhanced protocol described in §5. Significant costs arise for the creation and processing of messages, for inviting a new collaborator, and for joining as a collaborator. We consider the computational costs of these actions, the communication costs for different types of messages, and the memory and storage requirements. The security and consistency properties are discussed in the appendix.

We implemented a prototype of the privacy-enhanced protocol in Java based on the Treedoc CRDT with unique disambiguators [28], without optimizations. The instrumented prototype simulates all devices within a single thread of execution and measures execution times of relevant operations as well as the volume of network communication. We use a 2048-bit RSA modulus, and SHA-256 as the secure hash function. We calculate prime representatives as described in §2.2. We use SHA-256 as an approximation of a random oracle, and the Miller-Rabin primality test [29] with 50 iterations. We further chose $t = 16$, since assuming Firoozbakht's conjecture [30, p. 185] (which implies that the gap between primes p_k and p_{k+1} is less than $\ln^2 p_k - \ln p_k$ for all $k > 4$), this should always allow a suitable d to be found.

To evaluate the costs of the scheme based on realistic data, we replayed edits from Wikipedia editing histories. We randomly[2] selected 300 pages from Wikipedia. We excluded seven pages with only a single edit, and to ensure a reasonable emulation time, we excluded 23 pages which had either more than 250 edits, or more than 25,000 characters in the latest version; our results in this section demonstrate clear trends which will not significantly change for larger edit histories or pages. For the sake of estimating the communication costs, we assumed that *deviceID*s are 128-bit random numbers (to achieve uniqueness with high probability in a decentralized setting). For simplicity, we assume that all devices are always online, devices do not batch multiple operations together into a single message, and that devices only send messages when they edit the document. When replaying the editing history, we assumed that a Wikipedia user or IP address corresponds to a device, and that new collaborators get invited by and receive a snapshot of the document from the last person

2 Using https://en.wikipedia.org/wiki/Special:Random

who edited the document before them. We assume each line is represented as an atom, as commonly done in the evaluation of CRDT algorithms [24, 36, 37] (the original Treedoc paper used paragraph granularity [28]).

We did not consider the time taken for encrypting or decrypting messages, since the choice of the encryption scheme is independent of our protocol, and modern encryption algorithms are fast compared to the RSA accumulator operations. For signing, we used ECDSA and the NIST P-256 curve.

We measured execution times on a 2013 desktop-class 3.20GHz i5-4570 CPU with 32 GiB RAM running Oracle JRE 1.8.0_172 with a heap size of 8 GiB. We chose a 8 GiB heap size to reduce the number of garbage collection cycles and their impact on the measurements, and because we simulated all devices within a single process. The heap size of 8 GiB was enough to comfortably simulate up to 141 devices, therefore a single device can run the protocol with substantially less memory.

Discussion of simplifying assumptions

In a practical implementation, devices may want to batch edits, and periodically broadcast noop messages to other clients to confirm the latest seen document state. Devices may also be offline temporarily or permanently, delaying message delivery and processing until such devices comes back online again, but this merely defers when costs are incurred.

Periodically broadcasting noop messages would cause additional network traffic and devices would need to process additional messages. The dominant cost for processing noop messages is the verification of the accumulator of the sending device (see §6.1.1). This cost grows linearly with the number of atoms added and deleted since the last message from the device. Therefore, processing noop messages would reduce the computational cost for the accumulator verification for individual operations, however it is likely to increase the cumulative cost if a large number of atoms are added, and the same ones deleted, between edit operations from a device. Some additional costs may also be caused by devices that regularly send noop messages, but do not make any (more) edits. On the other hand, for devices that regularly send noop messages, other devices can skip the iterative witness computation, as the witness is simply the accumulator base if a device is up-to-date. Since we do not have reliable data on the network status of devices editing Wikipedia, we defer evaluation of these trade-offs to future work.

6.1 Computation costs

6.1.1 Basic editing operations

Processing any message requires checking the correctness of the hashes and the accumulator. Of those, verifying the accumulator tends to be the most costly, as it requires a modular exponentiation for every atom added or deleted since the last message from the source device.

In addition to the above, the dominant costs for inserting an atom are calculating a prime representative, and updating the device's accumulator, which requires one modular exponentiation. The median processing time for a message containing an insert operation from another device in our experiments was 5.6 ms, and 99% were processed within 11.0 ms. We observed outliers of up to 1.0 seconds, which were caused by the cost for verification of the accumulator when a relatively large number of changes have happened since the last message from the device that created the insert operation. Note that these numbers are only for insert operations created on a different device. Processing locally generated operations is faster, since they do not require the source device's accumulator to be verified.

For a delete operation, the additional costs are dominated by the cost of updating the device's accumulator, which requires a modular k-th root computation, and by the cost for the iterative witness computation. Figure 2 shows the time it took to process delete operations from other devices, with iterative witness computation enabled after every operation. Overall, the median cost for processing a delete operation was 12.4 ms, and 99% were processed within 64.9 ms. Outliers take up to 1.19 seconds and were due to the accumulator verification. As for the insert operation above, these numbers do not consider delete operations that were created on the same device, which are faster to process.

6.1.2 Adding a new collaborator

Adding a new collaborator requires four steps:

1. generating a snapshot on the inviting device,
2. verifying the snapshot and the new device,
3. initialising the local state on the new device, and
4. verifying the first message from every other device on the new device, and vice versa.

Let c be the current number of collaborating devices, and n be the current number of atoms in the doc-

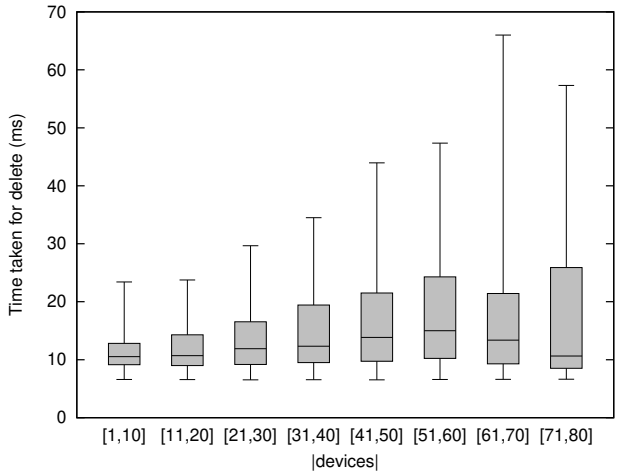


Fig. 2. Measured processing times for a delete operation from another device. The worst case execution time generally grows with the number of devices, as with every delete operation, we incrementally update the witnesses for all other devices, unless the deleted atom has been inserted after the last message from a device. Therefore, costs vary depending on the editing behaviour of users, and how often they communicate. Roughly speaking, deleting older parts of a document is more expensive, and more frequent synchronisation between devices also makes deletions more expensive. The boxes show the first, second, and third quartile. The whiskers show the range containing 99% of data points. We omit data for more than 80 devices where we have limited data. Overall, the median processing time is 12.4 ms, and the 99^{th} percentile is 64.9 ms.

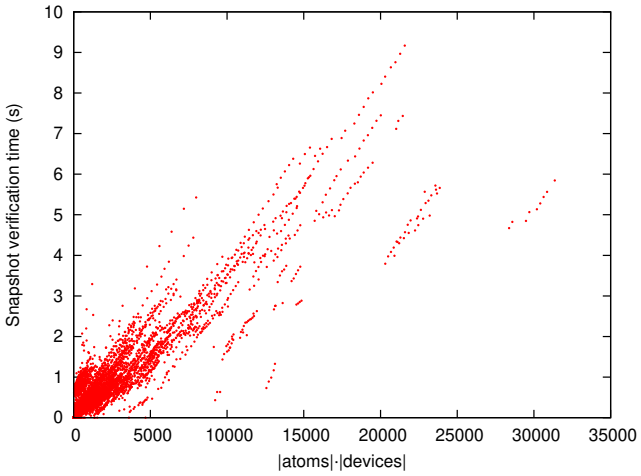


Fig. 3. Measured snapshot verification times. Verification time is at most linear in the product of the number of devices and atoms, however it can be significantly lower if many atoms have been added since the last message from collaborating devices.

ument. Then creating a snapshot requires $\mathcal{O}(n+c)$ operations, computing $\mathcal{O}(c^2)$ Merkle consistency proofs, plus computing a witness per device. Computing the consistency proofs is fast for a moderate number of devices. Computing the witness for a device requires a modular exponentiation for every atom that has been deleted since the last accumulator seen from that device (but was already present then). However, we iteratively compute witnesses with every message to minimize snapshot generation time, as described in §5.3.2. Therefore, the time taken to generate a snapshot is negligible compared to other costs such as its verification.

Verifying the Merkle consistency proofs can take $\Theta(c^2 \log m)$ time, where m is the total number of distinct messages broadcast since the document was created. However, in practice, the cost for verifying a snapshot is dominated by the costs for verifying that the set of atoms matches the accumulators, unless the number of collaborators becomes large compared to the number of atoms in the document, and many of them have sent their last message at different points in the history. For each device, this requires one modular exponentiation per atom that is present both in the latest document and in the accumulator from the device. Thus, for moderately large groups of collaborators, the worst case cost is $\mathcal{O}(c \cdot n)$. The actual cost is significantly smaller if a large number of atoms have been added since the last message from other devices. Figure 3 shows how long each snapshot verification took in our experiments, and its relationship to the product of the number of atoms and devices.

After a snapshot is verified, the remaining cost for initialising a new device is dominated by the cost for calculating the device's current accumulator value based on the current set of atoms. The cost is $n-1$ modular multiplications and a single modular exponentiation.

When a device receives the first message from another device, it needs to compute the current set of atoms at that device based on the counters within the message and the operation history, and based on that verify the accumulator value by re-computing it. This requires $O(n)$ modular exponentiations. We empirically verified this linear relationship; we observed a verification time of about 0.65 milliseconds per atom.

6.2 Communication costs

Table 3 compares the amount of data transferred for different message types for the basic and the privacy-enhanced protocols. The privacy-enhanced protocol requires additional data for individual messages (for nonce and accumulator), but snapshot sizes are smaller if the number of users is small compared to the number of

Variable	Description	Typical value	Empirical values min	median	max
d	Number of collaborating devices	variable	2	16	141
s_{devID}	Size of device identifier	16 B			
s_{hash}	Size of hash	32 B			
s_{sig}	Size of signature	72 B			
s_{nonce}	Size of nonce	16 B			
s_{RSA}	Size of accumulator	256 B			
s_{pos}	Size of position identifier	variable	1 B	3 B	117 B
s_{content}	Size of atom text fragment	variable	1 B	34 B	5.0 KiB
s_{pubkey}	Size of public key (accumulator + signing)	256+32 B			
s_{history}	Size of message history (excl. signatures)	variable	4.9 KiB	123 KiB	7.2 MiB
s_{doc}	Size of document including metadata	variable	699 B	4.8 KiB	77 KiB

Table 2. Description of different variables used in Table 3, and typical values. For the ones where typical values are highly variable, minimum, median, and maximum values from our simulations with Wikipedia edit histories are included.

	Basic protocol	Privacy-enhanced protocol
Message	$s_{\text{devID}} + d(s_{\text{devID}} + s_{\text{hash}}) + s_{\text{sig}}$+ **Op**	**Basic** + $s_{\text{nonce}} + s_{\text{RSA}} + s_{\text{hash}}$
Op_insert	**Message** + $s_{\text{pos}} + s_{\text{content}}$	**Basic**
Op_delete	**Message** + s_{devID}	**Basic**
Op_noop	**Message**	**Basic**
Op_addDevice	**Message** + s_{devID}	**Basic** + s_{pubkey}
Snapshot	$s_{\text{history}} + d \cdot s_{\text{sig}}$	$s_{\text{doc}} + d \cdot (s_{\text{devID}} + s_{\text{hash}} + 2s_{\text{RSA}} + \mathcal{O}(d \cdot \log s_{\text{history}}) + s_{\text{sig}})$

Table 3. Communication costs for different types of messages and operations. Small constants are omitted. Note that in the basic protocol, for a snapshot it is sufficient to include the most recent signature from each device.

atoms, since deleted atoms do not need to be transferred. Using the Wikipedia data, we looked at the amount of data that would need to be transferred to invite the user that has most recently made her first contribution. Figure 4 shows a comparison of the amount of data transferred in the basic scheme and the privacy-enhanced scheme. We observed a median 84% reduction in data transferred for the privacy-enhanced scheme compared to the basic scheme. The reduction was always more than 30%, and 98.2% in the best case.

6.3 Storage and memory requirements

A device needs to keep the atoms currently in the document in memory. In addition it must store, for each collaborator, the most recent message, the current witness, and additional metadata. The device needs to store the message history to be able to relay messages to other devices, and to calculate earlier states of the document which can be needed to verify an accumulator or to calculate a witness. The memory requirements for storing current atoms in a document corresponds to the y-axis in Figure 4, and the past history corresponds to the x-axis. Therefore, the overall memory and storage requirements are typically less than 10 MiB. A device may also keep an in-memory or disk cache of the prime representatives of all atoms (34 bytes per atom in our prototype) as computing those is costly. If memory/storage is scarce and the prime representative generator described in §2.2 is used, it can also memorize only the last 2 bytes, and recompute the remaining ones when needed.

7 Discussion

The privacy-preserving variant of our protocol has a significant computational and metadata overhead. The costs seem reasonable for text editing with line-granularity atoms, especially since most of the expensive operations can be parallelized and typically can be run in the background without interrupting the editing process. However, for character-level granularity or similar, the costs seem prohibitive, in particular if the document is large and collaborators get added frequently, or when edits are performed at a high frequency. The protocol may be well suited for other types of collaborative applications such as shared calendars or to-do lists [18].

While the protocol has a relatively large overhead, it scales well with the size of the document. Assuming a bounded number of devices and not considering costs for

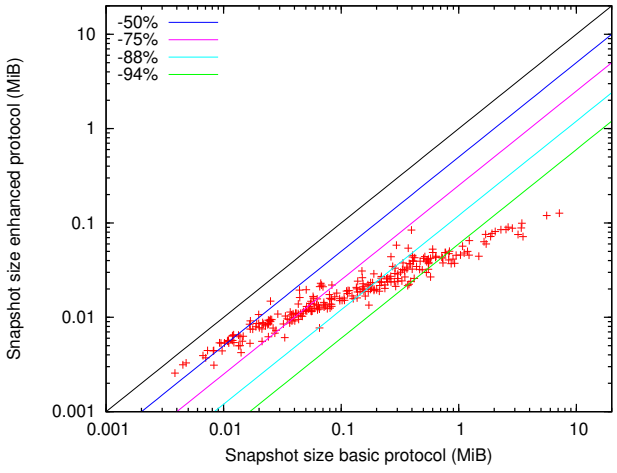


Fig. 4. Amount of data transferred to a new collaborator when invited by another device, for the most recently added collaborator in each of the pages from the Wikipedia dataset for the privacy-enhanced protocol in comparison to the basic protocol. The plot is log-log scale. A point below the black diagonal line indicates that the privacy-enhanced protocol transfers less data than the basic protocol. The privacy-enhanced scheme needs to transfer less data because it does not transfer deleted atoms.

the CRDT metadata, communication and computation costs for editing operations are constant or amortized constant, except when a new collaborator is added, in which case the cost is (practically) linear in the number of atoms.

However, communication costs also grow with the number of operations due to the CRDT metadata. We used the Treedoc CRDT without optimizations, which generates a relatively large communication overhead for CRDT metadata because the tree is not balanced and every tree node stores a device identifier. To counter this, one can use a CRDT more optimized for the application, e.g. LSEQ [24] for text editing, and introduce device identifier compression. Furthermore, metadata overhead can be reduced by allowing a list of operations to be sent within a message instead of only a single operation per message.

To reduce the cost for insert operations and snapshot verification, if the prime representative is generated as described in §2.2, the device inserting the atom can include the last 2 bytes of the prime representative with the atom metadata. Other devices only need to verify its validity, but do not need to recompute it. Note that in this case it is not necessary that the smallest d is chosen, as long as the result is a prime and every device uses the same d.

Some information about the history can still be inferred from the metadata found in the privacy-enhanced scheme, in particular how many operations have been performed on each device, and the position identifiers and counters may allow some inferences about the positions where text fragments were deleted and how much was deleted. On the other hand, it may be desirable to know at which positions parts of the document have been deleted, as the device creating a snapshot can omit arbitrary atoms and therefore potentially completely change the meaning of the content. Metadata does not, in general, allow someone in possession of only a snapshot to infer positions where atoms have been deleted.

Our protocol relies on CRDTs where atoms have totally ordered position identifiers. More research is needed to add support for other operation-based CRDTs that do not have this property, such as RGA [31].

Lastly, while our protocol detects any forks that may arise (as discussed in §5.3.3), a fork-resolution protocol is required to resolve forks caused by misbehaving devices. Such a protocol is out of scope of this paper.

8 Related work

Traditional **collaborative editing** applications rely on Operational Transformation (OT) algorithms [11, 26] to synchronize changes between devices. OT algorithms work by transforming concurrent operations so they can be applied in a different order. They tend to be relatively complex, as evidenced by the fact that several peer-reviewed OT algorithms have later been proved to be incorrect [15, 16, 27]. To the best of our knowledge, all widely deployed OT algorithms rely on a central server to totally order operations. For example, Google/Apache Wave is based on the Jupiter algorithm [26], which requires such a total ordering and needs to be able to perform server-side transformations on operations. It is therefore neither suited for peer-to-peer communication, nor for end-to-end encryption, as the server needs access to the plaintext.

More recently, (operation-based) Conflict-free Replicated Data Types (CRDTs) [32, 33] have been proposed to ensure convergence without requiring consensus between devices, providing strong eventual consistency. In contrast to OT, updates do not require any synchronization and all concurrent operations are designed to be commutative. At the time of writing, there are a number of projects actively working on collaborative editors or libraries based on CRDTs that allow devices to communicate peer-to-peer (using WebRTC), including Teletype for Atom, Conclave, and Automerge.

Several server-based collaborative editing systems with end-to-end encryption have been proposed, including SPORC [12], SECRET [13], and Capsule [19]. However, to the best of our knowledge, our protocol is the first one that provides authenticated snapshots and allows devices to verify that their view is consistent with other devices, even when other devices are offline.

Version control systems such as Git, Mercurial, or Subversion are another popular kind of tool for collaboration. They are not designed for real-time editing and require manual merging if a possible conflict is detected. Authenticated snapshots could also be implemented for version control systems; however we are not aware of any existing system that supports them.

Cryptographic accumulators have been proposed based on RSA [1, 2], bilinear maps [5, 25], Merkle trees [4], and vector commitments [6]. Variants of our protocol could also be designed based on other accumulator schemes with different trade-offs. For example, it seems possible to use Merkle trees instead of RSA accumulators to substantially reduce the constant factor of the computational overhead. However, Merkle trees do not support batch membership proofs, therefore substantially increasing the communication overhead for sending a snapshot. We chose to use RSA accumulators because they provide constant-size public keys, witnesses, and batch membership proofs.

In a three-party **authenticated data structure** (ADS) [35], a *source* replicates some data to one or more *servers*, and the servers answer queries on the data from *clients*, including a proof that allows clients to verify the authenticity of the response using a *digest* provided by the source (e.g. a hash). Our proposed scheme can be seen as an ADS for CRDTs, where the collaborators are sources, the inviter is the server, and a newly invited device is the client.

A **redactable signature** scheme [17, 34] allows a third party without knowledge of the secret signing key to remove parts of a signed message while still retaining a valid signature. Our protocol essentially uses redactable signatures – the signature within a message signs the current set of atoms, and the state descriptors within a snapshots contain a signature of a possibly redacted state of that set.

9 Conclusions and future work

We propose a protocol for peer-to-peer collaborative editing that allows new devices to be added as collaborators by sending a snapshot that only contains the latest state of a document. Such a snapshot reduces the amount of data that needs to be transferred to a new device and additionally hides the editing history of the document, while still allowing the new device to verify its integrity. This is achieved without requiring a consensus between collaborating devices and is therefore also suitable for devices that are frequently offline.

We evaluated the performance of the protocol based on editing histories of 270 Wikipedia pages, and showed that while it has a significant computational overhead due to the use of RSA accumulators, its performance is reasonable if applied to small documents or using a coarse granularity (e.g. line-based instead of character-based). 99% of insert operations were processed within 11.0 ms, and 99% of delete operations within 64.9 ms. We also measured a median 84% reduction in the data transferred to a new collaborator by using authenticated snapshots compared to a basic protocol that transfers the full editing history. Therefore it may be well suited for applications such as shared calendars and to-do lists, where users tend to make relatively few edits, and a coarser granularity of edits may be acceptable. Further research is needed to make the protocol more practical for real-time editing with character-level granularity.

Future research might also look at protocols that preserve information about the positions where text fragments have been deleted, or alternatively, completely hide this information. Another interesting research direction is developing CRDTs specifically designed for authenticated snapshots and history privacy, with a reduced overhead. It would also be interesting to design a protocol that does not only hide deleted parts from a new user, but also hides the author of a piece of text, either from new users or from all collaborators.

Acknowledgements

Stephan A. Kollmann is supported by Microsoft Research through its PhD Scholarship Programme, and by The Boeing Company. Martin Kleppmann and Alastair R. Beresford are fully and partially supported by The Boeing Company respectively. The opinions, findings, and conclusions or recommendations expressed are those of the authors and do not necessarily reflect those of the funders.

We also thank Ricardo Mendes, Laurent Simon, Tom Sutcliffe, Daniel R. Thomas, Diana Vasile, and Jiexin Zhang for helpful discussions and insight.

References

[1] Niko Barić and Birgit Pfitzmann. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In *Advances in Cryptology – EUROCRYPT '97*, pages 480–494. Springer, 1997.

[2] Josh Benaloh and Michael De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Advances in Cryptology – EUROCRYPT '93*, pages 274–285. Springer, 1993.

[3] Eric A. Brewer. Towards Robust Distributed Systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC 2000, page 7. ACM, 2000.

[4] Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. *International Journal of Information Security*, 11(5):349–363, 2012.

[5] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials. In *Public Key Cryptography – PKC 2009*, pages 481–500. Springer, 2009.

[6] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public-Key Cryptography – PKC 2013*, pages 55–72. Springer, 2013.

[7] Scott A. Crosby and Dan S. Wallach. Efficient Data Structures for Tamper-evident Logging. In *Proceedings of the 18th USENIX Security Symposium*, pages 317–334. USENIX Association, 2009.

[8] John Day-Richter. What's different about the new Google Docs: Making collaboration fast, 2010.

[9] David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In *Topics in Cryptology – CT-RSA 2015*, pages 127–144. Springer, 2015.

[10] Benjamin Dowling, Felix Günther, Udyani Herath, and Douglas Stebila. Secure Logging Schemes and Certificate Transparency. In *Computer Security – ESORICS 2016*, pages 140–158. Springer, 2016.

[11] Clarence A. Ellis and Simon J. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, volume 18, pages 399–407. ACM, 1989.

[12] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI 2010, pages 337–350. USENIX Association, 2010.

[13] Dennis Felsch, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. SECRET: On the Feasibility of a Secure, Efficient, and Collaborative Real-Time Web Editor. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*, AsiaCCS 2017, pages 835–848. ACM, 2017.

[14] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News*, 33(2):51–59, 2002.

[15] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving Correctness of Transformation Functions in Real-Time Groupware. In *ECSCW 2003*, pages 277–293. Springer, 2003.

[16] Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, and Pascal Molli. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science*, 351(2):167–183, 2006.

[17] Robert Johnson, David Molnar, Dawn Song, and David Wagner. Homomorphic signature schemes. In *Topics in Cryptology – CT-RSA 2002*, pages 244–262. Springer, 2002.

[18] Martin Kleppmann and Alastair R. Beresford. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.

[19] Nadim Kobeissi. Capsule: A protocol for secure collaborative document editing. IACR Cryptology ePrint 2018/253, 2018.

[20] Ben Laurie, Adam Langley, and Emilia Kasper. RFC 6962: Certificate Transparency. IETF, 2013.

[21] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, Availability, and Convergence. Technical Report UTCS TR-11-22, Department of Computer Science, The University of Texas at Austin, 2011.

[22] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud Storage with Minimal Trust. *ACM Transactions on Computer Systems*, 29(4):12:1–12:38, 2011.

[23] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology – CRYPTO '87*, pages 369–378. Springer, 1988.

[24] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In *Proceedings of the 2013 ACM Symposium on Document Engineering*, DocEng 2013, pages 37–46. ACM, 2013.

[25] Lan Nguyen. Accumulators from Bilinear Pairings and Applications. In *Topics in Cryptology – CT-RSA 2005*, pages 275–292. Springer, 2005.

[26] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*, UIST 1995, pages 111–120. ACM, 1995.

[27] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Proving correctness of transformation functions in collaborative editing systems. Technical Report RR-5795, INRIA, 2005.

[28] Nuno Preguiça, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A Commutative Replicated Data Type for Cooperative Editing. In *29th International Conference on Distributed Computing Systems*, ICDCS 2009, pages 395–403. IEEE, 2009.

[29] Michael O. Rabin. Probabilistic Algorithm for Testing Primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[30] Paulo Ribenboim. *The Little Book of Bigger Primes*. Springer, 2004.

[31] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[32] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report RR-7506, INRIA, 2011.

[33] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, pages 386–400. Springer, 2011.

[34] Ron Steinfeld, Laurence Bull, and Yuliang Zheng. Content Extraction Signatures. In *Information Security and Cryptology – ICISC 2001*, pages 285–304. Springer, 2001.

[35] Roberto Tamassia. Authenticated data structures. In *Algorithms – ESA 2003*, pages 2–5. Springer, 2003.

[36] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *29th IEEE International Conference on Distributed Computing Systems*, ICDCS 2009, pages 404–412. IEEE, 2009.

[37] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1162–1174, 2010.

Appendix

We show that the privacy-enhanced protocol described in §5 satisfies the properties from §3.1.

Edit integrity and attributability

Cryptographic signatures attached to each message ensure that only group members can modify the document. For the basic protocol of §4, the signatures also provide attributability of all modifications. When using the privacy-enhanced protocol of §5, a device can similarly use the signatures to attribute any changes made to the document after it joined. Parts of the document that were added earlier – before or concurrently to when a device joined – cannot be attributed directly using the signatures of the messages containing the insert operations, since the new collaborator does not receive those messages. However, attributability in this case is ensured by the signed accumulators from each collaborator that are part of each snapshot, since the set of atoms certified by each device in this way must also contain all atoms inserted by the device itself.

Edit history privacy

A new device, when joining, only receives the current set of atoms, the set of devices collaborating, several sequence numbers and cryptographic hashes, and a set of RSA accumulators. Assuming preimage resistance of the hash function and due to including a 128-bit random nonce into every message, it is infeasible to infer anything about previous contents from the hashes. RSA accumulator and witness values each contain an accumulated 128-bit random nonce; since the new device never learns this nonce, the accumulators are cfw-indistinguishable [9], making it infeasible to infer contents from the accumulator values. For efficiency reasons, we use the same nonces to calculate message hashes and accumulators to improve efficiency; we believe this does not introduce any weaknesses.

While the scheme hides the contents of all text deleted before a device joins, it does not perfectly hide the editing history. Since a snapshot also includes metadata such as position identifiers and sequence numbers, a new device can infer some information about the history, such as the number of messages sent by each device. Moreover, gaps between position identifiers can leak the fact that atoms have been deleted at a certain position (but not the values of those atoms).

Consistency and snapshot consistency

We show that our protocol satisfies a variant of *fork-join-causal consistency*, as introduced by Mahajan et al. [21, 22]. Stated informally, this consistency model requires that honest[3] devices always observe the system in a state that is consistent with a global execution graph, and that this execution graph correctly reflects the dependencies and operations performed by devices.

To prove that our protocol satisfies this consistency model, we first show how to construct the *happens-before* graph G (representing the global execution). For each honest device n we also define a graph G_n representing n's view of the execution. We then prove that G and G_n are consistent with each other: that is, reading the document at any vertex of G_n returns the same result as reading it at the corresponding vertex of G.

3 We use the word "honest" to refer to devices that correctly follow the protocol (in the distributed systems literature, the term "correct" is more common). A device that does not correctly follow the protocol, regardless whether by accident or by malice, is called "faulty".

Definition 1. Let G be a directed acyclic graph. We then define the partial order $\prec_G$ to be equal to the transitive closure of the graph. That is, for vertices a and b in G, we have $a \prec_G b$ if there is an edge $a \to b$ in G, or if there exists a vertex c such that $a \prec_G c$ and $c \prec_G b$. Similarly, the partial order $\prec_{G_n}$ is defined as the transitive closure of the graph G_n.

Definition 2. An *operation message* is a message containing an operation (insert, delete, noop, or addDevice, but not a snapshot) sent as part of the protocol.

Definition 3. For any vertex m in the graph G we define $read(G, m)$ to be the set of atoms in the document at the time immediately after m has been processed, i.e. the set of atoms a such that there exists a vertex $m_{I,a}$ containing an insert operation for a, with $m_{I,a} \preceq_G m$, and there exists no vertex $m_{D,a}$ containing a delete operation for a with $m_{D,a} \preceq_G m$:

$$read(G, m) = S(\{m' \mid m' \preceq_G m\}), \tag{30}$$

where the function $S(\dots)$ is defined in (13).

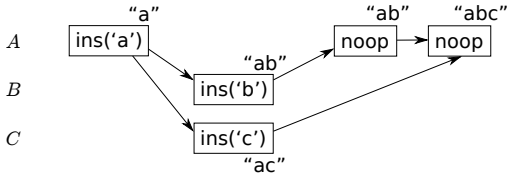
We can now formally define the fork-join-causal consistency model as follows.

Definition 4. An execution is fork-join-causally consistent if there exists a directed acyclic graph G (the *happens-before graph*) that satisfies the following three properties:

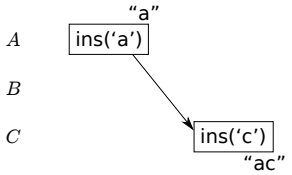
FJC0. G contains a vertex for every operation message sent by an honest device, and also a vertex for every operation message that is sent by a faulty device and processed by at least one honest device.

FJC1. The operations of an honest device are totally ordered in G. This total ordering must be consistent with the actual execution order of the operations at that device. Specifically, if v and v' are operations by n, then $v.startTime < v'.startTime \iff v \prec_G v'$.

FJC2. For each honest device n there exists a directed acyclic graph G_n in which there is a vertex for every operation message sent or received by n, and edges corresponding to the dependencies between those messages. By FJC0, for each vertex m in G_n there is a corresponding vertex m in G. We then require that for each vertex m in G_n, the document state is the same as the document state at the corresponding vertex in G: $read(G_n, m) = read(G, m)$.



(a) Happens-before graph of execution (G)



(b) Happens-before graph of C's view of the execution (G_C)

Fig. 5. Happens-before graphs for an execution with three devices where devices B and C perform concurrent inserts.

Basic protocol

For the basic protocol described in §4, G can simply be defined as follows: G contains a vertex for each message m sent or observed by an honest device, and a directed edge $a \to b$ between vertices a and b if a is one of the dependencies of b.

Figure 5a shows an example of a happens-before graph for an execution where device A inserts the atom 'a', followed by devices B and C concurrently adding atoms 'b' and 'c', respectively. Device A then performs noop operations in order to acknowledge the receipt of the edits from B and C. Figure 5b shows device C's view of the execution. For clarity, we omit addDevice operations.

This definition of G trivially satisfies property FJC0. Moreover, from the protocol definition it is relatively easy to see that property FJC1 is also satisfied. Every honest device increments its sequence number with every message it sends, and an honest device would not process a message from a device a that depends on a message from a with a higher or equal sequence number. Hence, the messages sent by an honest device are totally ordered in G.

For the basic protocol, it is also easy to see that property FJC2 is fulfilled. If a device n processes a message m, it needs to have processed all messages that happened before m in G_n. The use of cryptographic hashes within the message dependencies ensures that the set of messages preceding m in G_n is the same as the set of messages preceding m in G.

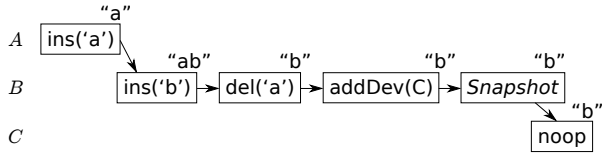


Fig. 6. Happens-before graph of an execution where device B adds device C by sending it a snapshot of the current state.

Privacy-enhanced protocol

In the privacy-enhanced protocol of §5, the above argument no longer works, since it relies on devices having received all messages that happened before a message m before processing m, which is not necessarily the case: devices do not receive messages that happened before they were added as a collaborator.

We illustrate the challenge by giving an example before proceeding to the formalisation. Consider the execution visualised in Figure 6.

Suppose that honest device A fails permanently after sending the insert operation for 'a', and therefore it never receives the later operations. Further assume that device B is faulty. Thus, there is no honest device that has observed the insert operation for 'b' or the delete operation for 'a'. The FJC0 property only requires that G contains operations observed by honest devices. Therefore it is not immediately clear how FJC2 can still be preserved for the noop operation by C (and any later operations).

One option would be to add a vertex containing an insert operation for each atom that C receives as part of the snapshot. However, this would allow too many executions. We only want to allow executions where snapshots are consistent with earlier messages seen by honest devices.

Since the messages containing the insertion of 'b' and the deletion of 'c' have not been observed by any honest device, it is not relevant for G whether they actually happened. It is only important whether it is possible to add a set of edit operations by faulty devices directly before the message adding a new device (or sending an updated snapshot) such that FJC1 and FJC2 are preserved. Thus we adapt the definition of G to allow the addition of vertices containing insert and delete operations by faulty devices between the vertices corresponding to messages observed by honest devices, and the vertex corresponding to the addDevice message for a new device, or a message containing an updated snapshot.

For each honest device n, Definition 5 describes the graph G_n that represents n's view of the execution. In summary, it contains insert operations for all atoms received by n in its initial snapshot, all messages processed by n, and edges for the dependencies between them. A device may receive more than one snapshot if another device performed operations concurrently to the first snapshot (as described in §5.3); if this is the case, n also contains insert and/or delete operations for atoms that were added/removed in subsequent snapshots.

Definition 5. For the privacy-enhanced protocol, we define G_n such that it contains:

1. A vertex for each operation message sent or processed by n.
2. An edge $a \to b$ between two messages processed by n if a is a dependency of b.
3. For each snapshot processed by n, a vertex r_i ($i = 1, ..., k$). If n has sent any messages after the snapshot, add an edge $r_i \to t_i$ to the vertex t_i corresponding to the first such message.
4. If n joined as a collaborator from a snapshot, for each atom a that was part of this first snapshot, a vertex u_a with an operation insert(a),
5. For each subsequent snapshot received by n, a vertex u_a with an operation insert(a) for each atom present in the snapshot if there is no *previous* vertex with an insert operation for a in G_n. In this context, *previous* means preceding a vertex t_i corresponding to a vertex corresponding to n's first message after the snapshot.
6. Similarly for each subsequent snapshot received by n, a vertex w_a containing an operation delete(a) for any atom a with an insert operation, but no delete operation, previously present in G_n that is not present in the snapshot.
7. For each such vertex u_a or w_a, an edge to the vertex r_i corresponding to the snapshot.

Proof overview

We construct a suitable happens-before graph G for an arbitrary execution, showing that G is a directed acyclic graph (Lemma 1), and hence that FJC0 and FJC1 are satisfied. Next, we show that deleted atoms cannot be re-added (Corollary 2.1), a property that is useful for Lemma 3, which shows that FJC2 holds for every message m, as long as it holds for every preceding snapshot. Finally, Lemma 4 shows that it holds for every snapshot, from which Corollary 4.1 deduces that FJC2 holds for G. We therefore conclude that the privacy-enhanced protocol is fork-join-causally consistent.

Definition 6. Let $v(m)$ be the version vector for a message $m = (src, ctr, op, deps, r, acc, mh, sig)$, containing the set of pairs (src_v, ctr_v) with the counter from the latest message from each device included in m, i.e. $ctr_v = ctr$ for $src_v = src$, and otherwise ctr_v is the counter of the entry for src_v in $deps$. The version vector of a snapshot is defined equivalently using the state descriptor of the creator of the snapshot.

Definition 7. We construct the happens-before graph G for an execution as follows.

1. Add a vertex for each operation message sent or processed by at least one honest device.
2. Add a directed edge $a \to b$ if a is a dependency of b and at least one honest device has sent or processed both a and b.
3. For each snapshot received by an honest device n, add a vertex r that represents the read of the snapshot by n. If n has sent any messages after the snapshot, add an edge $r \to t$ to the vertex t corresponding to the first such message.
4. For each snapshot received by an honest device n that was created by an honest device n', add an edge $m_s \to r$, where m_s is the message by n' defining the document state contained in the snapshot.
5. For each snapshot s received by an honest device n that was created by a faulty device, we add insert and delete operations by faulty devices as required to make the graph consistent as follows.
   Let src be the *deviceID* of the device that created the snapshot $s = (A_s, sdesc, ., .)$.
   Let $(src, ., ., deps, ., ., ., .) \in sdesc$ be the state descriptor for src in s.
   For every honest device n' let $(n', ctr_{n'}, .) \in deps$ be the snapshot's dependency on n', and $dep_{s,n'}$ be the vertex in $G_{n'}$ corresponding to the message from n' with sequence number $ctr_{n'}$.
   For each device n' we now find the set of operation messages that n' has observed by the time it produced $dep_{s,n'}$, and define the union of all these messages to be $ops(s)$:
   $$ops(s) = \bigcup_{n'} \{m' \mid m' \preceq_{G_{n'}} dep_{s,n'}\} \tag{31}$$
   Let $S(msgs)$ be the set of atoms that have been inserted but not deleted within a set of operation messages $msgs$, as defined in (13).
   Let A_s be the set of atoms received as part of the snapshot.
   $M_n = \{(., src_a, ., .) \in A_s \setminus S(ops(s)) \mid src_a \text{ is faulty}\}$ is defined to be the set of atoms by faulty devices that are part of the snapshot but have not been seen by any honest device before the snapshot.
   Now for each such snapshot s, do the following:
   (a) For each message $dep_{s,n'}$ from an honest device referred to in $deps$, add an edge $dep_{s,n'} \to r$, where r is the snapshot read vertex added in point 3.
   (b) For each atom $a \in M_n$, add a vertex i_a with an operation insert(a), an edge $dep_{s,n'} \to i_a$ for each honest device n', and an edge $i_a \to r$.
   (c) For each atom $a \in S(ops(s)) \setminus A_s$, add a vertex d_a with an operation delete(a), an edge $dep_{s,n'} \to d_a$ for each honest device n', and an edge $d_a \to r$.

Lemma 1. G is a directed acyclic graph.

Proof. First observe that for any edge $a \to b$ where a is a dependency of b, a's version vector must be smaller than b's. For a snapshot created by an honest device, point 4 of the construction of G (Definition 7) adds a path between vertex m_s and the corresponding vertex r. For a snapshot created by a faulty device, point 5 adds a number of paths between vertices for dependencies $dep_{s,n'}$, and r. Since each r does not have outgoing edges except to the corresponding t (as defined in point 3), and $v(m_s) < v(t)$ and $v(dep_{s,n'}) < v(t)$, the invariant $v(a) < v(b)$ is preserved for all edges $a \to b$ between actual messages, and additional edges do not add cycles. □

Since two consecutive messages m_i, m_{i+1} sent by the same honest device always have a dependency relation between them, and $v(m_i) < v(m_{i+1})$, from the above proof it also follows that G is consistent with their real-time ordering. Thus, the FJC1 property holds for G.

Lemma 2. Let n be an honest device, let m be a vertex corresponding to a message in G_n, and let $m_{D,a}$ be a vertex containing a delete operation for an atom a. If $m_{D,a} \preceq_G m$, and if n has processed m, then all insert operations for a processed by n precede m in G_n.

Proof. Let (src_a, ctr_a) be the source and counter of a, and let c_a be the entry for src_a in the version vector of m. Since the insertion of a must have happened before $m_{D,a}$ and therefore also before m, $ctr_a < c_a$. Due to the checks performed on sequence numbers, n does not accept an insert operation with a ctr less than or equal to the ctr of the latest message from src_a included in n's state. Thus, if n has already processed m, it will not

accept an insert operation for a that happened either after or concurrent to m. □

Corollary 2.1. If $m_{D,a} \preceq_G m$, $m \in G_n$, and $a \notin read(G_n, m)$, then for any vertex m' that succeeds m in G_n ($m \prec_{G_n} m'$), $a \notin read(G_n, m')$.

Lemma 3. Let m be a vertex in G corresponding to a message, and let n be an honest device such that $m \in G_n$. Assume that for every honest device $\tilde{n}$ and every vertex r_n corresponding to a snapshot received by $\tilde{n}$, $read(G, r_{\tilde{n}}) = read(G_{\tilde{n}}, r_{\tilde{n}})$ (note that this assumption will be proved in Lemma 4). Then for every such vertex m we have $read(G, m) = read(G_n, m)$.

Proof. By well-founded induction on m using the order relation $\prec_G$. That is, for any vertex m in G we assume the inductive hypothesis:

$$\forall m'.\ m' \prec_G m \implies read(G, m') = read(G_n, m') \quad (32)$$

and hence prove $read(G, m) = read(G_n, m)$. We break this down into two subgoals, $read(G_n, m) \subseteq read(G, m)$ and $read(G, m) \subseteq read(G_n, m)$.

***read*$(G_n, m) \subseteq$ *read*(G, m):**
Let a be an atom in $read(G_n, m)$. We start by showing that there must be an insert operation for a at or before m in G. Let $m_{I,a}$ be the message containing the insert operation for a observed by n. If $m_{I,a} = m$, $m_{I,a} \preceq_G m$ is trivially true. Otherwise, there exists at least one edge $d \to m$ in G_n such that $m_{I,a} \preceq_{G_n} d$, and there is no delete operation for a that precedes m in G_n. Thus, $a \in read(G_n, d)$. If d corresponds to an actual message, by the induction hypothesis, $a \in read(G_n, d) = read(G, d)$. Otherwise, d must be a vertex corresponding to a snapshot received by n, and we can apply the assumption $read(G, r_{\tilde{n}}) = read(G_{\tilde{n}}, r_{\tilde{n}})$ to conclude $a \in read(G_n, d) = read(G, d)$. Therefore, there must be an insert operation for a in G that precedes d, $m'_{I,a} \preceq_G d \prec_G m$.

To show that $a \in read(G, m)$, it remains to be shown that there is no delete operation for a at or before m in G. Suppose there was a vertex $m_{D,a} \in G$ containing such a delete operation for a, with $m_{D,a} \preceq_G m$. We show that this contradicts $a \in read(G_n, m)$. If $m_{D,a} = m$, this directly contradicts $a \in read(G_n, m)$. Otherwise, at least one vertex d with a edge $d \to m$ must contain or succeed the delete operation in G, $m_{D,a} \preceq_G d$. Let $d_1, d_2, ..., d_k$ be all such vertices. We consider two cases, whether any such d_i is in G_n, or not.

Case $d_i \in G_n$ for some i. Let d be any such d_i. Since $m_{D,a} \preceq_G d$, $a \notin read(G, d)$. We now show that $d \to m$ also exists in G_n and that $a \notin read(G_n, d)$. If d is an actual message, d must be a dependency of m, and by definition of G_n, $d \to m$ must be present in G_n. By the induction hypothesis, $a \notin read(G, d) = read(G_n, d)$. Otherwise, d must be a vertex corresponding to a snapshot received by n. Since in this case, d does not exist in any other device's view, the edge $d \to m$ can only exist in G if it exists in G_n. Thus, $d \prec_{G_n} m$. By the assumption $read(G, r_{\tilde{n}}) = read(G_{\tilde{n}}, r_{\tilde{n}})$ we have $a \notin read(G, d) = read(G_n, d)$.
Since $m_{D,a} \preceq_G d$, and we can apply Corollary 2.1, which implies that $a \notin read(G_n, m)$.

Case $d_i \notin G_n$ for all i. Let d be any such d_i. We consider two cases: whether d corresponds to an actual message, or whether it is a vertex added in our construction of G (Definition 7).
Consider first the case where d corresponds to an actual message received by an honest device n'. Since $d \to m \in G$, d must correspond to one of m's dependencies. Thus, n must have processed d before processing m, unless m is the first message by n after a snapshot. Since $d \notin G_n$, the latter must be true. Since $a \notin read(G, d)$, by the induction hypothesis, $a \notin read(G_{n'}, d)$. Since $m_{D,a} \preceq_G d$, by Corollary 2.1, in n''s view, the document does not contain a at message m, i.e. $a \notin read(G_{n'}, m)$. Since m was created by an honest device, and both n and n' have processed it and compared the accumulator value to its view of the set of atoms, they agree on the set of atoms at m. Thus, $a \notin read(G_{n'}, m) = read(G_n, m)$.
Now consider the other case, where d is one of the vertices we added when constructing G. Since m is an actual message, and there exists an edge $d \to m$ in G, and $d \notin G_n$, d must be a vertex added for a snapshot processed by a different device n', and m must be the first message by n' after that snapshot. Since $m_{D,a} \preceq_G d$, by the construction of G, $m_{D,a}$ must either be included in one of the state descriptors for an honest device $\bar{n}$ contained in the snapshot (i.e. $m_{D,a} \preceq_G s_{\bar{n}}$, where $s_{\bar{n}}$ is the message corresponding to $\bar{n}$'s state descriptor), or $m_{D,a}$ must be an additional vertex added to G in our construction (Definition 7, point 5(c)). In both cases, a is not part of the snapshot received by n', and the insert operation for a must have happened before the snapshot. Let (src_a, ctr_a) be the source and counter of a, and let c_a be the entry for src_a in the version vector associated with the snapshot. Again

in both cases, $ctr_a \leq c_a$, and therefore, n' would not accept an insert operation for a since its ctr would conflict with the snapshot it has received. Therefore, in n''s view, the document does not contain a at message m, i.e. $a \notin read(G_{n'}, m)$. Since m was created by an honest device, both n and n' have processed m, and both devices behave correctly, m's accumulator value must match the set of atoms at both devices, i.e. $read(G_{n'}, m) = read(G_n, m)$. Thus, $a \notin read(G_n, m)$.

***read*$(G, m) \subseteq$ *read*(G_n, m):**

Let $a \in read(G, m)$. Thus, there exists a vertex $m_{I,a}$ with an insert operation for a such that $m_{I,a} \preceq_G m$, and no delete operation before or within m. If $m_{I,a} = m$, since a delete operation cannot have happened before the insertion, $a \in read(G_n, m)$. Otherwise, there exists at least one edge $d \to m$ such that $m_{I,a} \preceq_G d$, and thus $a \in read(G, d)$. We consider two cases, whether $d \in G_n$, or not.

Case $d \in G_n$**.** Depending on whether d corresponds to an actual message or to a snapshot, we can apply the induction hypothesis or the assumption for snapshot vertices ($read(G, r_{\hat{n}}) = read(G_{\hat{n}}, r_{\hat{n}})$), to conclude that $a \in read(G_n, d)$. Hence, there exists an insert operation for a in G_n preceding m.

Next we show that there is no delete operation for a before m in G_n. Assume, for the sake of contradiction that there was a message $m_{D,a} \prec_{G_n} m$ containing a delete operation for a. We show that this contradicts $a \in read(G, m)$. We first consider the case that there exists a set of vertices $m_1, m_2, ..., m_k$ corresponding to actual messages such that $m_{D,a} \to m_1 \to ... \to m_k \to m$. By construction of G, the same set of messages and edges has to exist in G too, contradicting $a \in read(G, m)$. Otherwise, if no such set of vertices exist, $m_{D,a}$ must be a vertex of the type added for a snapshot in point 6 of the construction of G_n (Definition 5). Let r_i be the vertex corresponding to the snapshot directly after $m_{D,a}$. The existence of the delete vertex implies that a is not part of the set of atoms in the snapshot corresponding to r_i, but there is a vertex $\tilde{m}$ corresponding to n's entry in the dependencies of the snapshot in G_n, where a was still present: $a \in read(G_n, \tilde{m})$. Thus, there is an edge $\tilde{m} \prec_G r_i$ in G. Since m has happened after r_i, it must have also happened after the first message $m_{n,i}$ by n after r_i, and we get $\tilde{m} \prec_{G_n} r_i \prec_{G_n} m_{n,i} \preceq_{G_n} m$. Furthermore, since $m_{n,i} \preceq_{G_n} m$, there is a path of vertices corresponding to messages by n, $m_{dev,i} \to \hat{m}_1 \to ... \to \hat{m}_l$ in G_n such that $\hat{m}_1$ is a dependency of m. Since the vertices correspond to messages processed by an honest device, the same path has to exist in G, and we get $r_i \prec_G m_{n,i} \preceq_G m$. By the assumption $read(G, r_{\tilde{n}}) = read(G_{\tilde{n}}, r_{\tilde{n}})$, $a \notin read(G_n, r_i)$ implies that $a \notin read(G, r_i)$, and since $a \in read(G_n, \tilde{m}) \overset{\text{IH}}{=} read(G, \tilde{m})$ and $\tilde{m} \prec_G r_i \prec_G m$, there must be an insert operation for a preceding r_i in G. Because $\tilde{m} \prec_G r_i$ and $a \notin read(G, r_i)$, there must be a delete operation for a preceding r_i in G. Since $r_i \prec_G m$, $a \notin read(G, m)$, reaching the desired contradiction. Therefore, there is no delete operation for a before or at m in G_n, and since the edge $d \to m$ is present in G_n by construction, a must still be present at m in n's view. Thus, $a \in read(G_n, m)$.

Case $d \notin G_n$**.** As before, there are two cases. Either m is the first message by n after a snapshot (since n must have received all direct dependencies of any other message before processing it), or m is the first message by another honest device $\hat{n}$, and d is the vertex corresponding to the snapshot received by $\hat{n}$. In the first case, since $d \in G$ and $d \to m \in G$, there must be at least one honest device n' that has processed d and m. By the induction hypothesis, $read(G_n, d) = read(G_{n'}, d)$, and therefore $a \in read(G_{n'}, d)$. For the same reasons as above, there cannot be a delete operation for a before or within m in G_n, and therefore a must still be present at m in n''s view. Since both n and n' are honest devices, and n' must have verified the accumulator of m, they must have seen the same set of atoms at m, $a \in read(G_{n'}, m) = read(G_n, m)$.

In the latter case, by assumption, $a \in read(G_{\hat{n}}, d)$, implying that $m_{I,a} \preceq_{G_{\hat{n}}} d$. Again, there cannot be a delete operation for a before or at m in G, and thus a must still be present in $\hat{n}$'s view at m. Therefore, $a \in read(G_{\hat{n}}, m)$, and since n has agreed on the accumulator value of m and thus on the set of atoms, $a \in read(G, m)$.

□

Lemma 4. Let r_n be a vertex associated with a snapshot s received by n. Then the read corresponding to this vertex (which returns the atoms that were part of the snapshot) fulfils property FJC2 with regard to G, i.e. $read(G_n, r_n) = read(G, r_n)$.

Proof. By well-founded induction on r_n (using the happened-before ordering induced by G).

If n has created the document, the statement is trivially true. Otherwise, let $A_n = read(G_n, r_n)$, and

$A = read(G, r_n)$. Let $\hat{n}$ be the device creating the snapshot. If $\hat{n}$ behaves correctly, this implies that the snapshot corresponds to a message m_s created by $\hat{n}$. By induction and Lemma 3, $read(G_{\hat{n}}, m_s) = read(G, m_s)$, and since the only incoming edge for r_n in both G and G_n is from m_s, $read(G_n, r_n) = read(G_{\hat{n}}, m_s) = read(G, m_s) = read(G, r_n)$.

If $\hat{n}$ is faulty, we first show that $A_n \subseteq A$. Let $a = (src, ctr_a, ., .) \in A_n$. Thus, a was part of the set of atoms presented as part of the snapshot. We define s_h to be the message by honest device h corresponding to the h's state descriptor. We consider three cases:

src **is an honest device.**
Let $(src, ctr_{src}, ., ., acc, mh, sig, wit)$ be the state descriptor for *src* presented in the snapshot, let m_{src} be the corresponding message from *src* with sequence number ctr_{src}, and let $m_{I,a}$ be the message containing the insert operation for a. The snapshot is only accepted by n if $ctr_a < ctr_{src}$. Using *sig*, *acc*, and *wit*, n has verified that $a \in read(G_{src}, m_{src}) \overset{\text{IH, Lemma 3}}{=} read(G, m_{src})$, and $m_{I,a} \prec_G m_{src} \prec_G m$. It remains to be shown that there is no delete operation for a in G before m.
For each honest device h that was part of the snapshot where s_h happened after $m_{I,a}$, n has verified using the Merkle consistency proofs that $m_{I,a} \prec_{G_h} s_h$, and it has verified using h's witness that $a \in read(G_h, s_h) \overset{\text{IH, Lemma 3}}{=} read(G, s_h)$, and thus there exists no delete operation for a before any s_h. Lastly, since a is part of the snapshot, no delete operation for a is added in point 5 of the construction of G (Definition 7).

src **is faulty.** We further consider two sub-cases: whether at least on one honest device h has observed an insert operation for a before s_{src}.
If yes, based on the dependencies in h's state descriptor, n can infer that the insertion has happened before s_h. Again this means that n has verified using h's witness that $a \in read(G_h, m_h)$. The rest of the argument is as in the previous case.
If not, since a is part of the snapshot, the construction of G, in particular point 5, ensures that there exists an insert operation for a before r_n, and no delete operation.

Now we show that $A \subseteq A_n$. Let $a \in A$. For the sake of contradiction, assume $a \notin A_n$. We consider two cases:

$a \in S(ops(s))$ **(as defined in Definition 7).** This implies that in G, an operation delete(a) was added before r_n, contradicting $a \in A$.

$a \notin S(ops(s))$**.** This implies that either no honest device has seen an insert operation for a before r_n, or at least one has seen a delete operation for a. Either way, $a \notin A$.

□

Corollary 4.1. For a device n and a message $m \in G_n$, n's view of the document at m is equal to the state according to G, $read(G, m) = read(G_n, m)$. Thus, the privacy-enhanced protocol preserves FJC2.

Convergence and availability

For the basic protocol, convergence and availability directly follow from the properties of the CRDT and from the use of cryptographic hashes for dependencies. However, a fork-resolution protocol is required to resolve forks caused by misbehaving devices. Such a protocol is out of scope of this paper.

For the privacy-enhanced protocol, fork-join-causal consistency ensures that the views of honest group members converge to a consistent state. This again requires a fork-resolution protocol in case a misbehaving devices causes the views of honest devices to be forked. Any two participants can generally communicate even if other collaborators are offline; however, if multiple devices join concurrently, they require the help of an existing collaborator to reach a state where they can collaborate directly, since neither of them has seen all required dependencies of the others at the time of joining.