

Data-strophy: When Your Integrity Goes Wild, So Does Your Data!*

Ya-Nan Li[†]
The University of Sydney
Sydney, Australia
yanan.li@sydney.edu.au

Yaqing Song
University of Electronic Science and
Technology of China
Chengdu, China
yaqings@163.com

Qiang Tang[‡]
The University of Sydney
Sydney, Australia
qiang.tang@sydney.edu.au

Moti Yung
Google LLC&
Columbia University
New York, United States
moti@google.com

Yuan Zhang[‡]
University of Electronic Science and
Technology of China
Chengdu, China
zhangyuan@uestc.edu.cn

Abstract

Proton is a popular security and privacy service provider with a large user base spanning both organizations and individuals. Proton Docs/Sheets support real-time collaborative document editing while claiming to provide end-to-end security.

We analyze the cryptographic design and the collaborative editing protocol of Proton Docs/Sheets inspecting the open-source Web client and webpage code. We demonstrate three distinct “integrity” attacks against Proton Docs/Sheets that can cause history rewriting, context manipulation, and censorship, all of which can, in fact, evade detection. The first two can be launched even when the Proton server acts honestly, and the third is mounted by a corrupted Proton server. We also present the corresponding mitigation methods. Our attacks highlight the subtleties of end-to-end security in collaborative settings involving multiple users and constant updates. This state of affairs naturally calls for systematic formal treatment (i.e., design and/or analysis) of the security of such systems.

CCS Concepts

• **Security and privacy** → **Cryptanalysis and other attacks**;
Web application security; **Privacy-preserving protocols**.

Keywords

Cryptographic Attacks, E2EE

1 Introduction

End-to-end encryption (E2EE, which we will also use to denote “end-to-end encrypted” in later sections) has become one of the most important security paradigms in modern computing, aiming to ensure that user data remains both confidential and protected against unauthorized tampering, even from the service provider itself. Its appeal has made it increasingly popular across consumers and enterprise products, from its initial applications in messaging and email to file storage and sharing. These days, it is further expanding into richer collaborative applications such as online documents and spreadsheets. This trend is visible in major privacy-focused platforms, including Proton [19], DekkoCORE [7], Anytype [3],

Tresorit [22], SNote [20], and collaborative suites such as ONLYOFFICE [18], CryptPad [6], and Fileverse [10], which explicitly market E2EE document editing and spreadsheet collaboration.

However, the security of E2EE systems has long been recognized as subtle. A substantial body of research has studied attacks on end-to-end encrypted messaging systems, and more recent work has extended this scrutiny to encrypted storage platforms, revealing that even when plaintext remains protected, deployment choices can still create significant confidentiality risks through metadata leakage, misuse of cryptographic mechanisms, or flaws in system integration. Alongside these practical attacks, more formal treatments of E2EE security have also begun to emerge, aiming to capture what such systems do and do not warrant in practice [1, 4, 15].

E2EE in collaborative systems is even more challenging. Unlike messaging or simple storage, collaborative applications involve multiple users, continuously evolving shared state, and a constant stream of concurrent updates, so security depends not only on protecting individually encrypted objects but also on preserving their meaning within a larger context. This makes the notion of “integrity” particularly subtle: users must be able to trust not only the secrecy of updates, but also their history, authorship, placement, and visibility among participants. However, research has not kept up with this shift in deployment, and our understanding of the actual security provided by real-world encrypted collaboration products remains limited. Beyond recent work on end-to-end encrypted Git systems [16], there has been little study of integrity in practical E2EE collaboration platforms. Motivated by this gap, we set out to examine the security of a widely deployed end-to-end encrypted collaborative document system provided by Proton, which is a popular privacy-focused service vendor, serving over 100,000 organizations (including The United Nations) [25]. Our goal is to understand what guarantees such a system actually provides once end-to-end encryption is combined with shared, evolving state, and real-time collaboration.

1.1 Our contributions

After carefully analyzing the protocols of Proton Docs/Sheets, we identify three types of integrity attacks that could cause different ways of tampering with history, context, or content without being detected.

*This is a full version, and the preliminary version is published at ACM CCS’26.

[†]The authors are listed in alphabetical order.

[‡]Corresponding author.

History rewriting attacks. In our first attack, a single malicious user (with write permission) can manipulate the edit history of a shared document (including the latest version); thus, we call it *history rewriting attack*. For example, the attacker can remove or replace content without being detected by anyone else, including the server and other users. The attack is easy to launch, and the malicious user can just reuse the honest client code with minimum changes, e.g., changes in certain parameters.

Despite that each update is signed by the user who generates it, this attack is still possible as Proton Docs/Sheets provide a squash function,¹ run by clients to merge versions when above a threshold. A corrupted client can modify the threshold by intervening in the normal execution of the Web client and resetting the value, and then trigger this squash function and carefully insert/delete content (as updates).

To illustrate potential consequences, see the following example about reporting (more examples can be found in Section 4.1). Startups regularly prepare reports (e.g., financial reports) to their investors (or similarly, venture capital also needs to prepare performance reports to limited partners). Some of them may involve third-party consultants (e.g., financial advisory), or even auditors. When the startup team finalizes a version, a malicious team member may modify numbers or content and add some dummy changes, so that the auditor or external consultant would squash those updates again and completely eliminate the trace of the malicious member. The final report submitted would contain incorrect or even confidential information (in some historical text) and would cause confusion, reputation damage, or even compliance issues. Moreover, the malicious member will not be identified.

Context manipulation attacks. Our second attack allows a malicious user $\mathcal{U}_A$ to introduce edits that *appear to originate from another user*. Instead of directly impersonating the victim $\mathcal{U}_V$, the attacker takes a more subtle route: forwarding text (in the form of ciphertext with proper signatures) that the victim legitimately authored in one document Doc 1 into a different document Doc 2. The two documents may be shared among two groups of users, when the attacker and the victim are in both groups. Moreover, *the forwarded text passes verifications* and thus appears to other collaborators as if it were written by $\mathcal{U}_V$, without triggering any alerts. Slightly different attack techniques would be applied to different modes of Proton Docs (i.e., editing mode, or suggesting/comment mode).

This attack is possible because each update message is not properly bound to the document. Still, naively forwarding would not be effective (i.e., will not be visible to users on Doc 2): each update contains a pointer to a certain identifier/position in the document to be applied. However, the text “position” in Doc 1 may not even exist in Doc 2 (due to identifier generation rules from the underlying CRDT: the “Conflict-free Replicated Data Types” distributed data structure mechanism), thus could simply be invisible to users of Doc 2. We observe that the attacker can carefully prepare a “bridge” update on Doc 2 (through the underlying CRDT to generate proper encoding) as a stepping-stone to generate an *anchor identifier* in Doc 2 for the

forwarded update to be applicable on Doc 2, particularly to enable the forwarded update links correctly to the anchor identifier.

Let us see the consequence through the following example (and more examples can be found in Section 4.2). There are two documents in a company. Doc 1 contains a candidate list of employees that would be laid-off (e.g., the company is facing financial difficulties at the moment), which is shared and kept confidential only by a small number of executives. Doc 2 is a human resources (HR) document shared more broadly among all employees (e.g., on-boarding document). One manager $\mathcal{U}_A$ may be unhappy with the list in Doc 1 (which has employees from their department) suggested by another executive $\mathcal{U}_V$, and therefore can forward the list to Doc 2, so that all employees see the list and recognize that it is from $\mathcal{U}_V$ (e.g., Doc 2 is in the *suggesting* mode, showing that the text is from $\mathcal{U}_V$).

Censorship attacks. In our third attack, a corrupted server can selectively *block* content from some victim users without being detected. To mount the attack, the corrupted server will selectively *drop* some intermediary *update* messages before “pushing” the new edits (e.g., adding some text) from victim users to other online honest clients. This effectively filters content, but users will not notice.

Other potential issues. In addition to the above we identify two cryptographic misuses, including signing on a secret key (and revealing the signature) that may leak information about the key, and utilizing the same key pair for both encryption and signing. Both methods are not generically secure and thus require more detailed investigation. In addition, document names are directly leaked when inviting a user via email or when editing in the suggesting/comment mode.

Mitigation suggestions. We provide general mitigation methods, at least to “mute” the attacks. For history rewriting, we can let all collaborators check the squashed result, or deploy some kind of byzantine consensus among all online users, to ensure that the squash function is performed honestly. A remedy for context manipulation is to properly bind each update to the corresponding document. For censorship, we suggest triggering warnings to users if abnormal cases (e.g., receiving many unapplicable updates) occur, and deploying an updatable authenticated data structure at the client side to check the consistency of the document version. In addition, we suggest that Proton follows general security practices and prevents unnecessary leakage. Some mitigation may cause efficiency downgrades and/or only mute our attacks. A systematic study on efficient and provably secure designs for E2EE collaborations is urgently needed.

Proof of concept. We provide a proof-of-concept (PoC) to test the effectiveness of our attacks. Our experimental results show that all our (first three) attacks can be easily reproduced on the Proton’s Web client. To replicate those attacks, we devise and facilitate manual interaction with the Proton Web client in tandem with local program execution; moreover, for some attacks, we need the participation of the Proton server, which we do not have access to. Instead, we use man-in-the-middle (MitM) to *simulate* a malicious server.

Implications and reflections. Integrity, as one basic goal of E2EE security, is indeed more complicated in the collaborative setting; edit history, authorship, document views across all collaborators,

¹It is different from the usual “compression” on plaintext (e.g., gzip) that was leveraged in practical chosen ciphertext attacks (CCA). Instead, it only merges some parts of metadata, particularly digital signatures, across a sequence of updates. The squash function is temporarily disabled in Proton Sheets. When reopened, Proton Sheets would still suffer from this attack.

and others should all be taken into account. This becomes even more challenging when other functionalities, such as squash, are needed for efficiency considerations. Systematic studies with formal security modeling and analysis are needed. They not only provide a rigorous foundation for secure system design (even if with limited scope), but also inspire new vulnerability discovery in real-world systems, such as the *unforgeability* model in the recent E2EE Git service [16] that inspired our first two attacks. Moreover, we find that the design is not specific to Proton, but is also widely adopted across other E2EE collaborative systems, which implies the underlying integrity issue may be more fundamental and widespread. Furthermore, lack of “integrity” can lead to other security violations, including privacy breaches and censorship. For example, in our attacks, malicious users can leak confidential information (by modifying content) under others’ names. Our attacks not only improve our understanding of E2EE collaborative systems, but also raise multiple immediate questions, both on vulnerabilities and secure design and analysis.

What we do not consider. We do not consider that a malicious service provider plants a backdoor into the client code, which means that in our attacks, honest clients do have faithful client code running. Theoretically, no security guarantees can be ensured without a trusted client software. In practice, most end-to-end encrypted service providers release their client code as open source to enable public scrutiny. We also do not consider the vulnerability due to the low entropy of the user passwords used for authentication.

1.2 Ethical considerations

We followed the steps proposed by Coordinated Vulnerability Disclosure (CVD) programs: We disclosed our findings to Proton on 10 April, 2026, and received an official acknowledgment on 13 April, 2026, where we were asked to follow their vulnerability disclosure policy and were temporarily embargoed from making our findings public. We redacted all identifying details, including names of the provider and algorithms, and identifying features in our CCS ’26 submission. We sent a follow-up email on 4 June 2026, and they acknowledged on the same day. On 11 August 2026, the 120-calendar-day confidentiality period has expired, calculated from the first acknowledgment date. We sent the second follow-up email on 12 August 2026, notifying them of the end of the embargo, and they acknowledged it on 13 August 2026. No extension was requested at any point. Under Proton’s published disclosure policy², the coordinated disclosure requirements have been met.

For more details, including our PoC relevant ethical considerations, we defer to Appendix B.

1.3 Related work

Recent attacks on E2EE storage and messaging. For one of the most widely used E2EE cloud storage platforms, MEGA, it was shown that a malicious server can recover a user’s private key by attempting multiple logins [2, 5, 13]. Based on this vulnerability, the adversary can recover the plaintext to undermine confidentiality and further inject forged files to compromise integrity. Hofmann et al. [14] investigated several cloud storage systems and found that

they are vulnerable. These attacks were caused by fundamental flaws in cryptographic design, including the lack of user public key authentication and the use of unauthenticated encryption modes. Fábrega et al. [9] proposed injection attacks against the end-to-end encryption applications, where an adversary can “inject” chosen messages to a target user via sharing and then observe the length of the resulting ciphertext to recover the users’ files. Similar attacks that involve injecting data and then observing the target ciphertext length have also been used in other applications to breach data confidentiality, such as password managers [8], messaging [12], secure channel [11], and chunk-based storage [23].

In contrast, we examine an emerging E2EE collaborative system, Proton Docs/Sheets. Unlike E2EE storage systems, it treats a document in a fine-grained manner: a document is stored as multiple update records, and each update results in a new version (as detailed in Section 2). Those recent attacks mostly focus on confidentiality issues, while our attacks are mostly on variants of integrity, though they can still result in different consequences.

2 Overview of Proton Docs/Sheets

Roughly, in Proton Docs/Sheets, users merely send the “delta” (denoting the *updates*) induced by their edits to the server, and the server forwards them to other collaborators. Overall, Proton Docs/Sheets utilize a conflict-free replicated data type (CRDT) to encode each update and deploys cryptographic algorithms to wrap them, and the encrypted updates are broadcast among all collaborators to achieve synchronization. Such a mechanism is widely deployed by many E2EE collaborative editing platforms [3, 6]. In this section, we review the overall document update procedure of Proton Docs/Sheets as follows.

Updates generated at the client side. To achieve synchronization without conflicts, Proton Docs/Sheets employ the open-source library Yjs [26], which implements a high-performance CRDT [17]. Under Yjs, a user’s modification to a document based on their local replica is encoded using a structured representation that ensures convergence across all collaborators.

CRDT’s algorithm. When editing a shared document, each user is assigned a unique identifier named `clientID`, and each character³ is identified by its editor’s identifier and a Lamport timestamp, with the form of `id: <clientID, clock>`. There are only two types of operations, insertion and deletion, which represent all modifications. Each user’s clock starts at 0 and increments by 1 when one character is inserted. Each character is represented by an insertion operation with a unique identifier. An insertion operation also involves two pointers, one pointing to the direct predecessor and the other to the successor at the time of creation, respectively. The pointers are essentially the identifiers of the characters they refer to. For the deletion operation, the character to be deleted will be marked as “deleted” instead of being physically removed. In a nutshell, each character can be regarded as a node in a doubly linked list, satisfying the causal ordering. When collaborating on editing a document, users only need to broadcast the insertions and deletions (collectively referred to as updates) generated by their edits to all

²<https://proton.me/security/vulnerability-disclosure>

³Ref. [17] uses the term “element” to refer to the inserted character. For clarity, we utilize the term “character” in this paper.

users, rather than the entire replica. Thanks to the causal ordering, the users who receive *all* updates can recover the same document content even if the insertion and deletion operations are made on different replicas. Conflicts would occur, i.e., two users may type at the same position at the same time. The two insertions have the same origin and rightOrigin. To achieve eventual consistency, the algorithm specifies ordering rules to resolve conflicts: the update with a small clientID is placed to the left. The detailed ordering rules are referred to [17].

Update's structure. Yjs provides a compact data structure to represent an update, which is also the actual updated data broadcast by users.⁴ An update is represented by an array, which consists of multiple Items and a DeleteSet. An Item depicts consecutive insertions in the same position, which includes five parameters: the identifier of the first character in the insertions id: $\langle \text{clientID}, \text{clock} \rangle$, the length of the insertions len, the identifier of the predecessor origin, the identifier of the successor rightOrigin, and the content. The Item implies that the i -th ($1 \leq i \leq \text{len}$) character has an identifier id: $\langle \text{clientID}, \text{clock} + i - 1 \rangle$, and the i -th ($1 < i \leq \text{len}$) has a predecessor $\langle \text{clientID}, \text{clock} + i - 2 \rangle$, the i -th ($1 \leq i < \text{len}$) has a successor $\langle \text{clientID}, \text{clock} + i \rangle$, the predecessor of the first character is origin, and the successor of the last one is rightOrigin. A DeleteSet is a compact structure for describing logical deletions: it is a mapping from clientIDs to collections of DeleteItems. Each DeleteItem is a contiguous deletion interval. It can be described as a pair $\{\text{clock}, \text{length}\}$, which indicates that the characters inserted by the given clientID with clock in $[\text{clock}, \text{clock} + \text{length} - 1]$ are marked as deleted. The consecutive insertions/deletions are organized into a single Item/DeleteItem instead of generating one for each insertion/deletion, thereby reducing communication and storage costs. For clarity, we provide an editing instance in the Section 3.1.

Update generation and wrapping. Proton Docs/Sheets directly utilize Yjs, where a random 32-bit clientID is assigned to a user each time the document is opened, and the clock starts from 0 and increases monotonically. Proton Docs batches a user's document modifications into an update every 300 milliseconds when multiple users are editing collaboratively. Proton Sheets generates an update when an operation is completed (e.g., completing a cell input). In Proton Docs/Sheets, each user has an address private/public key pair $(\text{sk}_{\text{addr}}, \text{pk}_{\text{addr}})$ generated using Curve25519, and all collaborators in a document share a symmetric key ek used to encrypt the document content.⁵ Each update is cryptographically wrapped under its editor's sk_{addr} and the document symmetric key ek into an *update message* and sent to the server via WebSocket. (The wrapping algorithm, named WrapMsg, is described in Section 4.2.1.)

Update compression. To reduce storage costs, Yjs provides a merge function, denoted as MergeUpdate, that merges multiple updates into a single update. Corresponding to the merge functionality, Proton Docs/Sheets provide a *squash* algorithm, denoted as Squash,

⁴Here, we describe the basic update structure, and updates generated in the suggesting/comment mode in Proton Docs may contain additional parameters (e.g., style information), which are detailed in Section 3.2. Proton Sheets adopts a more complex structure to accommodate their cell-oriented organization, and we defer the details to Section 3.3.

⁵The details of Proton Docs/Sheets's key hierarchy can be found in Appendix D.

```

OpenDoc(IDDoc, ek)
Given: a document identifier IDDoc, the symmetric key ek
Return: the document content ContDoc
1: IDcom ← getCommitid(IDDoc)
2: {mi}i=1n ← getCommitDoc(IDDoc, IDcom)
3: for i = 1 to n
4:   Updi, adi ← UnwrapMsg(ek, mi)
5: end for
6: sta ← Yjs.Init() // Initialize the Yjs state
7: sta' ← Yjs.ApplyUpdate(sta, Upd1, Upd2, ..., Updn)
8: ContDoc ← sta'.getCont() // Retrieve from the current Yjs state
9: return ContDoc

```

Figure 1: The algorithm of opening a document.

which is run by clients (while the outcome will be accepted by the server).

The MergeUpdate function operates in two steps. *Step 1:* All Items are grouped according to the clientID contained in their id fields and ordered by decreasing clientID. Within each group sharing the same clientID, the corresponding Items are sorted by increasing clock. Items whose clock ranges are fully covered by Items already added to the merged result are discarded, while partially overlapping Items are truncated to exclude the already covered clock ranges before being added. *Step 2:* All DeleteSets are unioned into a single set and ordered by decreasing clientID. Deletion intervals with the same clientID are sorted by increasing clock and merged into a single DeleteItem. Overlapping deletion intervals are deduplicated following the same rules applied to insertion ranges.

When the number of update messages reaches a threshold, Proton squash mechanism leverages Yjs's merge function and further cryptographically wraps the merged result into a single update message. The detailed squash algorithm is provided in Section 4.1.

Update synchronization at the server side. To distinguish between versions of a document (*each new update message results in a new version*), the server assigns a unique identifier, called ID_{com}, to each version. The server stores all received update messages, which are ordered according to the sequence in which they were received by the server and form the latest document version. To synchronize updates, the server encapsulates update messages in different formats depending on the user's online status.

For users who just come back, they open a document and reconstruct its plaintext content. We denote this process as OpenDoc, which is illustrated in Fig. 1. Specifically, when opening a document identified by ID_{Doc}, the user first requests the latest ID_{com} from the server via the getCommitid interface, and then retrieves the corresponding latest version via the getCommitDoc interface. The server responds by returning all received update messages {m_i}_{i=1}ⁿ for the document via HTTPS, which are wrapped in a batch format (detailed in Section 4.3). The user then unwraps each update message by invoking the UnwrapMsg algorithm (detailed in Section 4.2.1) to recover the underlying plaintext updates. Finally, the

recovered updates are applied using the Yjs-provided ApplyUpdate interface to reconstruct the document content $Cont_{Doc}$.

For users who are already online, the server directly sends the update messages received from other online users to them via WebSocket. Upon receiving them, the users unwrap them and apply the new updates to their current replica.

Three modes provided in Proton Docs. Proton Docs provides three modes: editing mode, suggesting mode, and comment mode.⁶ Under the suggesting or comment mode, Proton Docs directly shows the authorship of document changes. For the suggesting mode, users can modify a document, and those with write permission can accept or reject the changes. But comments cannot be directly applied to document modification. The modification-s/comments made in any of the three modes will result in the corresponding update messages. But the difference is that when making changes under the suggesting/comment modes, a comment is generated, which abstracts the modification or records the comment. Subsequently, the comment is cryptographically wrapped into a *comment message* with the editor's sk_{addr} and the document symmetric key ek , and sent to the server. (The wrapping algorithm, denoted as $WrapCmt$, is described in Section 4.2.2.)

3 Detailed Update Structure

3.1 Document editing instance

Assume there are two users, U_A , U_B , collaborating to edit an empty document. U_A types "ADFG" from left to right, it will generate an update Upd_A with the specified data structure (described in Section 2), as shown in Fig. 2. Then, U_A sends Upd_A to U_B . U_B types "BC" between "A" and "D", types "E" between "D" and "F", and deletes the content "FG", generating an update Upd_B , as shown in Fig. 3. U_B sends Upd_B to U_A . U_A applies Upd_B , and both U_A and U_B see the content "ABCDE". Fig. 4 shows the doubly linked list generated from Upd_A and Upd_B .

When conflicts occur, i.e., two users may type at the same position (e.g., U_A types "a" between "A" and "B", but U_B types "b") on the same replica. The two insertions have the same origin and rightOrigin. To achieve eventual consistency, Yjs utilizes the ordering rules to resolve conflicts. In the above scenario, the update with a small clientID is placed to the left. For example, if U_A 's clientID is 1, and U_B 's clientID is 2, then they will see "AabB".

An instance of MergeUpdate. We provide an instance of the merge function. Consider three Items, denoted as $Item_1$, $Item_2$, and $Item_3$, all sharing the same clientID. Suppose $Item_1$ records clocks from 5 to 9, $Item_2$ records clocks from 0 to 6, and $Item_3$ records clocks from 3 to 4. During merging, $Item_2$, which has the smallest starting clock, is added to the merged result first. Next, $Item_3$ is discarded because its clock range 3-4 is already covered. Finally, $Item_1$ is added after removing the overlapping portion corresponding to clocks 5-6.

3.2 Updates in the suggesting/comment mode

Proton Docs provides the suggesting mode that enables users to type content and later decide to accept or reject it. In this case, Proton Docs also generates comments, which show what characters

```
Update: [
  Item {
    id: <clientID: 1, clock: 0>,
    // Assume  $U_A$ 's clientID is 1
    len: 4,
    // the length of insertion is 4
    //  $U_A$ 's clock increases from 0 to 3
    origin: null,
    rightOrigin: null,
    content: "ADFG"
  }
  ds: DeleteSet: { clients: Map(0) {} }
]
```

Figure 2: The structure of the plaintext update Upd_A .

```
Update: [
  Item {
    id: <clientID: 2, clock: 0>,
    // Assume  $U_B$ 's clientID is 2
    len: 2,
    // the length of insertion is 2
    //  $U_B$ 's clock increases from 0 to 1
    origin: <clientID: 1, clock: 0>,
    rightOrigin: <clientID: 1, clock: 1>,
    content: "BC"
  }
  Item {
    id: <clientID: 2, clock: 2>,
    len: 1,
    // the length of insertion is 1
    //  $U_B$ 's clock increases from 1 to 2
    origin: <clientID: 1, clock: 1>,
    rightOrigin: <clientID: 1, clock: 2>,
    content: "E"
  }
  ds: DeleteSet {
    clients: Map(1) {
      clientID: 1: [clock: 2, length: 2]
    }
    // delete clock 2 and 3 from  $U_A$ 
  }
]
```

Figure 3: The structure of the plaintext update Upd_B . The text in green indicates a DeleteItem.

are inserted or deleted. In addition, Proton Docs provides a comment mode that allows users to select content and leave a comment.

Under the two modes, Proton Docs creates a thread with a unique identifier for each comment (as described in Section 4.2.2), and the corresponding content is highlighted. Hence, except for the above basic update data structure, updates generated under the suggesting/comment mode include additional parameters based on

⁶Currently, Proton Sheets only supports the editing mode.

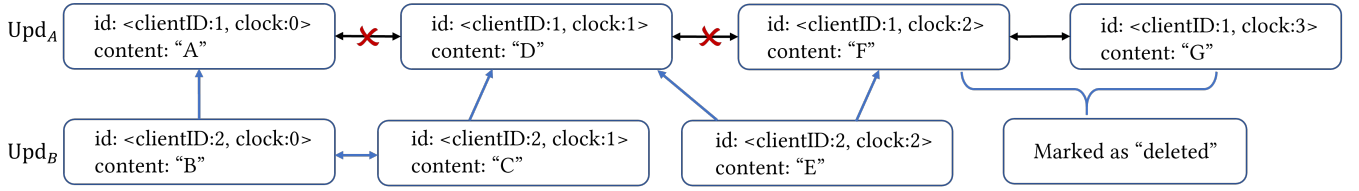


Figure 4: An example of a doubly linked list in CRDT.

the Yjs-provided shared type `YXmlText`, which is usually used to represent rich-text content.

An update generated by typing the first characters in the suggesting mode is represented by a `YXmlText`, where an `Item` initializes a `YXmlText` with `rightOrigin` pointing to the last `YXmlText`, an `Item` contains the corresponding suggesting thread identifier, an `Item` contains the inserted content, and some `Items` include the format and style information. Updates generated by typing in an existing suggesting thread (insertion/deletion immediately following a previously inserted character in the suggesting mode) have the same data structure as those in the editing mode. An updates generated in the comment mode have a similar structure to those in the suggesting mode, with one difference: the update first marks the selected content as deleted and re-inserts it.

3.3 Updates in Proton Sheets

Proton Sheets also uses Yjs to achieve conflict-free synchronization across collaborative spreadsheet editing. Yjs provides two shared types, `Y.Map` and `Y.Array`, which can be used for spreadsheet editing. A `Y.Array` is an ordered sequence of elements, internally represented as a doubly linked list of elements. Each element in the array has a unique identifier `id: {clientID, clock}` and maintains `origin` and `rightOrigin` pointers to its left and right neighbors. The first element inserted into a `Y.Array` also maintains a value named `parent` which points to the initial `Item` that initialized the `Y.Array`. A `Y.Map` is an unordered collection of unique keys and their corresponding values, which achieves key-based access. The inserted key-value pair maintains a `parent` that points to the initial `Item` that initialized the `Y.Map`.

The organization of a spreadsheet. A spreadsheet is organized as a multi-module structure. Typically, the `sheetDataV2` module maintains all cell data, the `shareStrings` module acts as a global dictionary to deduplicate and store textual content efficiently, the `cellXfs` module serves as a centralized style registry for visual and numeric formatting, and the `recalcCells` module operates as a task queue for formula recalculation.⁷ The structures of the four modules are detailed as follows.

The `sheetDataV2` module utilizes a four-layer nested hierarchy, which is structured from top to bottom as follows. At the top layer, a `Y.Map` is used to store all sheets in a spreadsheet, where a sheet’s identifier serves as the key, and its data container serves as the corresponding value. It allows for instant lookup of a specific sheet without traversing all updates. At the second layer, each sheet’s data container is stored as a `Y.Array`, in which an element represents a

row, and the physical index of the array represents the visual row order. At the third layer, each row is represented by a `Y.Map`, which contains a specific key “value” and the corresponding cell data in the row. For the bottom layer, a cell is stored as a plain object. In addition, the `shareStrings`, `cellXfs`, and `recalcCells` modules are instantiated with `Y.Array`, respectively.

Update structure. Similar to document updates, a spreadsheet update is also represented as a collection of `Items` and a `DeleteSet`. `Items` are the basic units of operations, representing the content insertion or the initialization of data types (e.g., `Y.Array/Y.Map`). Each `Item` is uniquely identified by an identifier `id`, comprising a `clientID` and a monotonic `clock`. Except for the parameters `id`, `origin`, `rightOrigin`, and `len` described in Section 2, an `Item` may contain additional parameters, `parentSub` and `parent`. `parentSub` is a structural pointer indicating whether the current operation is for `Y.Array` or `Y.Map`. If `parentSub` is a string, it indicates that the parent node is a `Y.Map`, and the string is the key in this map. If `parentSub` is undefined, it indicates that the parent node is a `Y.Array`. For `parent`, if an `Item` initializes a root-level module, `parent` is the name of the root module, e.g., `shareStrings`. If an `Item` is a child element nested within another structure, the `parent` points to the unique identifier of its parent `Item`. For example, in the relationship between the second and the third layers of a spreadsheet, a `Y.Map` representing a specific row contains a `parent` that references the unique identifier of the `Y.Array` representing the sheet. The `DeleteSet` is also a compact representation of marking logical deletions, which is the same as that of document update.

Sheet editing instance. In the following, we take the example of entering numbers and strings in a spreadsheet to illustrate the parameters in `Items`. When a user with `clientID: ID` enters a string into a cell in the first row and first column in a new spreadsheet, the corresponding update generally has nine `Items`:

- `Item0` with `id0: {ID, 0}` assigns a new `Y.Array` with `parentSub: "1"` (the default identifier for the first sheet) and `parent: sheetDataV2` to initialize a data container.
- `Item1` with `id1: {ID, 1}` inserts a null value to the `Y.Array` by setting its `parent: id0`.
- `Item2` with `id2: {ID, 2}` and `origin: id1` marks `Item1` deleted.
- `Item3` with `id3: {ID, 3}`, `origin: id1`, and `rightOrigin: id2` inserts a `Y.Map` and sets `len` to 1. `Item1` to `Item3` initialize the row structure by first inserting a temporary null, then replacing it with a `Y.Map` which is used to hold all subsequent cell data.
- `Item4` with `id4: {ID, 4}` and `parent: id3` initializes a `Y.Array` named “value” into the row container identified by `id3`.
- `Item5` with `id5: {ID, 5}` and `parent: id4` populates the cell value `[null, {ss: 0}]` to the `Y.Array` identified by `id4`, where

⁷There are other modules, and we only introduce the commonly used ones.

the null value serves as a structural offset which makes the column number equal to the array index, and the reference `{ss: 0}` points to the global string dictionary, and the clock increases by 2, which is the same as the number of inserted elements in the cell value.

- `Item6` with `id6: {ID, 7}`, `parent: sharedStrings` initializes a global string dictionary.

- `Item7` with `id7: {ID, 8}` with `rightOrigin: id6` inserts the actual text string to the global string dictionary. It only performs the insertion operation once, and the clock increases by 1. The inserted string is treated as a single unit to the `sharedStrings` array, and `Yjs` assigns it exactly one clock tick.

- `Item8` with `id8: {ID, 9}` and `parent: recalcCells` inserts a patch which indicates the author's identifier, the sheet identifier, and the row and column coordinates of the cell.

The user broadcasts the update, and we assume another user with `clientID: ID'` applies it, and then enters a number into a cell in the fifth row and fourth column, it generates an update with the following `Items`:

- `Item0` with `id0: {ID', 0}`, `origin: id3`, and `rightOrigin: id2` insert three null placeholders to fill the gap between the first row and the fifth row, and the clock increases by 3.

- `Item1` with `id1: {ID', 3}`, `origin: {ID', 2}`, and `rightOrigin: id2` insert a logical marker and immediately deletes it to offer an inserting position for the upcoming fifth row.

- `Item2` with `id2: {ID', 4}`, `origin: {ID', 2}`, and `rightOrigin: id1` inserts a `Y.Map` used as the data container for the fifth row.

- `Item3` with `id3: {ID', 5}` and `parent: id2` inserts a `Y.Array` with `parentSub: "values"` into the row container identified by `id2`.

- `Item4` with `id4: {ID', 6}` and `parent: id3` inserts four null values to fill the gap between the first column and the fourth column and make the column number equal to the array index and inserts the actual number, as well as the format reference, to the `Y.Array` identified by `id3`. The clock increases by 5.

- `Item5` with `id5: {ID', 11}` and `parent: cellXfs` registers the format (numeric pattern) with a unique style identifier.

- `Item6` with `id6: {ID', 12}` and `parent: cellXfs` registers the format (numeric pattern and alignment method) with a unique style identifier.

- `Item7` with `id7: {ID', 13}` and `origin: id9` inserts a patch which indicates the author's identifier, the sheet identifier, and the row and column coordinates of the update.

In subsequent updates, when a user enters content into a cell, and if the row where the cell is located has not been initialized, then a new `Y.Map` is initialized for this row, and a new `Y.Array` is initialized for the cell. If the content is a string and the inserted string is already in the global dictionary, it is directly referenced; otherwise, it is first inserted into the global dictionary and then referenced. If the content is a number, it generates two `Items` of format, which point to the last `Items` in the `cellXfs` module. Each update includes a patch that also points to the last `Item` stored within the `recalcCells` module.

4 Descriptions of the Attacks

4.1 History rewriting attacks

High level intuition. History rewriting attacks demonstrate that a malicious user can manipulate a shared document's edit history, for example, by removing already shared historical updates or adding new updates, *without being detected* by anyone else, including the server or any other users. To mount the attack, the attacker who corrupts one legitimate user (e.g., with one client) with write access can alter the document history arbitrarily by leveraging the "compression-like" functionality, named *squash*, that Proton Docs/Sheets provided. By tampering with the document history, the attacker can produce a document that deviates from its *latest* content, *without being caught*.

Impacted security in the context of Proton Docs/Sheets. Just like many other collaborative editing tools, Proton Docs/Sheets offer a "See view history" feature for users to inspect and restore prior versions (i.e., essentially keep the edit history). The "compression" function *squash* will regularly merge versions (particularly the digital signatures on each update), e.g., history is in the form of version 1,2,3,4, and after a regular *squash*, it becomes version 1,2,3 (the new version 3 merges the updates in previous versions 3,4). Proton Docs can show the authorship of the edits made under the suggesting mode, even if they are squashed.⁸

Our history rewriting attack would leave no record in this edit history: i.e., if certain previous records were intentionally deleted or modified by the corrupted user/client $\mathcal{U}_A$, the prior records would simply disappear in everyone's view history, without any notification. Even if in the rare case that some other honest user $\mathcal{U}$ manually keeps track of edits, notices the change, and immediately starts extracting all the digital signatures that were supposed to be associated with each edit to trace who did the edit, there is only one record of *squash* from $\mathcal{U}_A$. $\mathcal{U}$ cannot report this edit to anyone else, as the actual deletion or modification could be done by any collaborator, and $\mathcal{U}_A$ only did a regular *squash*.⁹

Examples. To concretize the attack, we consider the following examples¹⁰: policy documents. Many policy documents (e.g., for public services, large organizations, or enterprises) are long and usually the result of rounds of discussions and negotiations among various parties (e.g., several committee members). Imagine one term among many (e.g., could be 100) is with an explicit condition as a compromise from one committee member $\mathcal{U}_A$. After everyone reviewed the documentation (before releasing to the whole organization), $\mathcal{U}_A$ can simply remove the condition, or modify other terms.

Attacker model. We assume only one user with write permission gets corrupted; the server and other users are all honest. The attacker reuses the honest client with minor modifications to it.

Source of vulnerability (how Proton Docs/Sheets work). Upon the number of update messages reaches a threshold, the Squash

⁸The edits made under the suggesting mode correspond to comment messages including the authorship information, which is detailed in Section 4.2.

⁹In most of the time, even the record of squash by $\mathcal{U}_A$ may disappear, i.e., merged into other honest squash executions.

¹⁰We note that there are many other examples, since Docs/Sheets are widely used by all kinds of organizations. Besides the reporting example in the contribution, we give another example for illustration.

```

Squash( $ID_{Doc}$ ,  $ID_{com}$ ,  $\alpha$ ,  $sk_{addr}$ ,  $ek$ ,  $\{m_i\}_{i=1}^n$ )
Given: a document identifier  $ID_{Doc}$ , a version identifier  $ID_{com}$ , the
factor  $\alpha$ , the editor's address private key  $sk_{addr}$ , the symmetric key  $ek$ ,
the document update messages  $\{m_i\}_{i=1}^n$ 
Return: a merged update message  $m_{mrg}$ , a lock identifier  $ID_{LK}$ 
1:  $n \leftarrow \text{numberOfMsg}(\{m_i\}_{i=1}^n)$ 
2:  $t \leftarrow \text{GetThreshold}(\text{docType})$ 
3: if  $n \leq t$  then return
4:  $ID_{LK} \leftarrow \text{LockDoc}(ID_{Doc}, ID_{com})$ 
5: for  $i = 1$  to  $n$ 
6:    $Upd_i, ad_i \leftarrow \text{UnwrapMsg}(ek, m_i)$ 
7: end for
8:  $v \leftarrow \lfloor t\alpha \rfloor$ 
9:  $Upd_{mrg} \leftarrow Yjs.\text{MergeUpdate}(\{Upd_i\}_{i=v+1}^n)$ 
10:  $\{email, ts, vn\} \leftarrow \text{GetMetadata}()$ 
11:  $m_{mrg} \leftarrow \text{WrapMsg}(ek, sk_{addr}, Upd_{mrg}, email, ts, vn)$ 
12: return  $m_{mrg}$ ,  $ID_{LK}$ 

```

Figure 5: The squash algorithm (run by honest users/clients). The source of vulnerability is highlighted.

algorithm is invoked after running OpenDoc algorithm. Given the plaintext updates recovered from OpenDoc and stored on the user's local client, the Squash algorithm is run by the client to *compress* multiple update messages into one. At a high level, the Squash algorithm drops the signatures on *each individual* update message, and signs the merged (one single) update message. *This is exactly the source of vulnerability.*

More specifically, as illustrated in Fig. 5, the algorithm takes as input the document identifier ID_{Doc} , the current ID_{com} , the factor α , the editor's address private key sk_{addr} , the document symmetric key ek , and the set of document update messages $\{m_i\}_{i=1}^n$, where n denotes the total number of update messages. It outputs a *single* update message that wraps the merged update. The steps are as follows.

(1) It first obtains the current number of update messages n with the `numberOfMsg` interface and the threshold t with the `GetThreshold` interface that takes the document type `docType` as input. The two parameters, threshold t and factor α , determine when squash is triggered and which updates are selected for merging. Concretely, t denotes the minimum number of updates required to trigger squash and α , (< 1) is utilized to compute the number of updates that need to be kept¹¹. A reverse-order squash policy is adopted, under which the latest $n - \lfloor t\alpha \rfloor$ updates are merged into a single one, while the remaining updates are preserved.¹²

(2) The algorithm then checks whether the number of document updates exceeds the threshold t . If this condition does not hold, the algorithm terminates without performing squash. Otherwise, the

squash process proceeds. The user first invokes the LockDoc interface, taking ID_{Doc} and ID_{com} as input, to request a lock identifier ID_{LK} from the server, which informs the server which version is being squashed.

(3) Next, the algorithm computes the number of updates to be preserved. Specifically, the first $v = \lfloor t\alpha \rfloor$ updates are retained, while the remaining updates $Upd_{v+1}, \dots, Upd_n$ are merged into a single update Upd_{mrg} by invoking `MergeUpdate`. The algorithm then retrieves the current metadata via the `GetMetadata` interface and invokes the `WrapMsg` algorithm to wrap the merged update together with the metadata into a new update message m_{mrg} .

(4) After squashing, the preserved original update messages $\{m_i\}_{i=1}^v$, the merged update message m_{mrg} , and ID_{LK} are sent to the server. While a user is executing Squash, other online users may continue to generate and submit new update messages to the server. During this period, the server withholds these new updates from the user performing Squash until all existing update messages have been fully unwrapped (i.e., finishing the first seven steps shown in Fig. 5).

(5) Finally, based on the ID_{com} and ID_{LK} included in the request, the server replaces the previous version with the squashed one and forwards to only those update messages generated after that ID_{com} to the client that runs squash just now.

Suppose a document contains 600 updates and the parameters are set to $t = 500$ and $\alpha = 0.5$, then the latest $600 - 500 \times 0.5 = 350$ updates are merged during the squash process, while the remaining updates are preserved in their original form.

Concrete attack. During running Squash, a malicious/corrupted user/client with write permission can *arbitrarily insert or delete* updates, as the Squash algorithm removes individual signatures, thus destroying the source of individual updates.

Also, the OpenDoc algorithm does not check whether the content is consistent before and after squash, i.e., some other clients repeat the squash themselves.¹³ Therefore, other collaborators of the document would not notice it had been tampered with unless they manually compare with the previous content. We demonstrate the attack step-by-step below. The malicious/modified version of the Squash algorithm (we call Squash*) run by the malicious insider is illustrated in Fig. 6.

1) *Trigger the squash function arbitrarily — manipulating the squash condition at the Web client.* Under normal operations, the Squash algorithm is triggered only when the number of updates in a document exceeds the predefined threshold. To carry out the attack efficiently, the attacker can manipulate the squash trigger conditions, namely, the threshold t and factor α , at the Web client, so that the existing updates satisfy an attacker-defined triggering condition and the desired updates are included in the merged result.

We note that modifying the squash trigger condition requires some care, as the Web client code is dynamically retrieved from the server and cannot be persistently altered. Instead, the attacker $\mathcal{U}_A$ leverages the browser's inspection and debugging interface. Concretely, $\mathcal{U}_A$ locates the code responsible for retrieving the squash threshold t , sets a breakpoint immediately before this value

¹¹In Proton, the threshold t is 500 for documents and 250 for spreadsheets, and α defaults to 0.5.

¹² $\lfloor \cdot \rfloor$ is a rounding down function.

¹³This is unlikely to be practical, as it requires an honest client to be always online, and keeps tracking all squash executions; or server honestly keeps more version histories for each client, so that they can retrieve both versions to compare consistency.

Squash*($ID_{Doc}, ID_{com}, \alpha, sk_{addr}, ek, \{m_i\}_{i=1}^n$)

Given: a document identifier ID_{Doc} , a version identifier ID_{com} , the factor α , the editor's address private key sk_{addr} , the symmetric key ek , the document update messages $\{m_i\}_{i=1}^n$

Return: a merged update message m_{mrg}^* , a lock identifier ID_{LK}

```

1:  $n \leftarrow \text{numberOfMsg}(\{m_i\}_{i=1}^n)$ 
2:  $\lfloor \text{reset } t \text{ as } t^* \rfloor$ 
3: if  $n \leq t^*$  then return
4:  $\lfloor \text{reset } \alpha \text{ as } \alpha^* \rfloor$ 
5:  $ID_{LK} \leftarrow \text{LockDoc}(ID_{Doc}, ID_{com})$ 
6: for  $i = 1$  to  $n$ 
7:    $Upd_i, ad_i \leftarrow \text{UnwrapMsg}(ek, m_i)$ 
8: end for
9:  $v \leftarrow \lfloor t^* \alpha^* \rfloor$ 
10:  $\lfloor \{Upd^*\}_{i=v+1}^n \leftarrow \text{Tamper}(\{Upd\}_{i=v+1}^n) \rfloor$ 
11:  $\lfloor Upd_{mrg}^* \leftarrow Yjs.MergeUpdate(\{Upd^*\}_{i=v+1}^n) \rfloor$ 
12:  $\{email, ts, vn\} \leftarrow \text{GetMetadata}()$ 
13:  $m_{mrg}^* \leftarrow \text{WrapMsg}(ek, sk_{addr}, Upd_{mrg}^*, email, ts, vn)$ 
14: return  $m_{mrg}^*, ID_{LK}$ 

```

Figure 6: The malicious/modified squash algorithm, where we highlight the differences from Squash.

is retrieved, and redefines the corresponding function to return an attacker-chosen value. Execution is then resumed. As a result, Squash *proceeds using the attacker-specified threshold*. $\mathcal{U}_A$ can follow the similar method to modify the factor α , so that the merge process starts at an arbitrary update chosen by the attacker.

2) *Tamper with the document content — modifying the updates to be merged.* As a malicious client with write access to the target document, $\mathcal{U}_A$ can arbitrarily manipulate the updates that are fed into MergeUpdate to produce a merged update. Such manipulations include, but are not limited to, *replacing existing updates with new content, removing selected updates, and injecting additional updates*. Assume a document contains a sequence of n updates $\{Upd_i\}_{i=1}^n$, and $\mathcal{U}_A$ wants to modify one of the updates Upd_q ($1 \leq q \leq n$).

In Step 1), $\mathcal{U}_A$ adjusts the squash threshold and factor so that the update Upd_q is included in the set of updates to be squashed. In Step 2), $\mathcal{U}_A$ replaces Upd_q with a modified update Upd_q^* . As a result, $\mathcal{U}_A$ obtains a merged update $Upd_{mrg}^* \leftarrow Yjs.MergeUpdate(Upd_q^*, \dots, Upd_n)$. Finally, the merged update Upd_{mrg}^* is wrapped into a new update message m_{mrg}^* using the WrapMsg algorithm. Once this update message is accepted by the server and propagated to the others (step 3) below), the modified document state is displayed.

3) *Send the tampered document to the Proton server.* As the attacker has write access to the document, the merged update message m_{mrg}^* (that includes malicious updates), together with the remaining update messages, can be sent directly to the Proton server. When other collaborators subsequently open the document and reconstruct the plaintext content via OpenDoc, the server returns the merged update message (maybe along with some remaining unmerged update

messages). The clients then unwrap these update messages and apply the plaintext updates (without realizing the document history or content has been tampered with), as illustrated in Fig. 1.

4.2 Context manipulation attacks

High level intuition. Context manipulation attacks allow an attacker to introduce edits that *appear to originate from another user* by forwarding a victim $\mathcal{U}_V$'s text in one document Doc 1 into a different document Doc 2. The forwarded text passes both client-side and server-side verifications, and thus appears to other collaborators as if it were written by $\mathcal{U}_V$, without triggering any alerts. The consequences of such attacks can be severe in document-editing systems: some content may be confidential in one document and get leaked in another more broadly shared document, or some content would have a completely different meaning depending on the different context of the two documents, causing potential damages, under the name of the victim.

Impacted security in the context of Proton Docs/Sheets. In online collaborative document editing platforms, such as Google Docs and Overleaf, users can check the authorship of each edit. Although Proton Docs does not explicitly provide an authorship-tracking interface in the *editing* mode, it is often the case that the edited content already contains the editor's identity. For example, multiple authors write a paper, and an author may leave some comments starting with their name, e.g., "Alice: delete the above paragraph". Moreover, it also provides the *suggesting* mode and *comment* mode, both of which enable users to check the author of each edit. Slightly different tricks will be applied to different modes. For the editing mode, $\mathcal{U}_A$ only forwards $\mathcal{U}_V$ ' updates to the target document. For the suggesting/comment mode, $\mathcal{U}_A$ also needs to forward comment messages to enable the authorship to be visible. Please see details in Section 4.2.1 and 4.2.2.

Our attack demonstrates that this authorship checking provides no security guarantee in Proton Docs/Sheets: the "claimed" author of an edit in one document¹⁴ may never perform the edit on that document, which can cause a completely different interpretation depending on the context of each document.

Examples. Besides the HR documents example in introduction, we consider another example. One other scenario could be a software company testing two pre-release systems with external pilot customers (who may not have coding knowledge), where each system comes with its own collaborative README document. In the README for system 1 (Doc 1), there is a harmless maintenance instruction such as "rm -rf" on a temporary folder, because that folder is only meant for disposable test files. In the README for system 2 (Doc 2), however, customers are instructed to move valuable local data into that same folder as part of the migration workflow, for example, by copying the data there and deleting the original copy afterward. Now an attacker (e.g., an employee who knows he will be asked to leave the company) can forward the rm -rf instruction from Doc 1 into Doc 2 while preserving the appearance that it was legitimately written there, and does so immediately after

¹⁴This could either be from the suggesting or comment mode, or directly examining actual records with associated digital signatures.

the customer moves the data, the result can be catastrophic: the customer may execute the command believing it is an authorized step in the system 2 workflow, irreversibly deleting their only remaining copy of important local data. The damage here is not merely textual confusion; it is the forged provenance of the instruction inside the trusted collaborative document that makes the destructive command look like part of the official process.

4.2.1 Context manipulation attacks in the editing mode. The attacker only needs to manipulate the victim’s update messages and “forward” them to a target document Doc 2. Certain text (from some document Doc 1) will appear to originate from the victim on Doc 2.

Attacker model. The attacker is allowed to corrupt just one legitimate user $\mathcal{U}_A$ who shares (at least can read) two documents Doc 1, Doc 2 with the victim (an honest user) $\mathcal{U}_V$. $\mathcal{U}_V$ has write access to the document Doc 1, while the corrupted user $\mathcal{U}_A$ has write access to Doc 2. The server and other users (including $\mathcal{U}_V$) are all honest.¹⁵

Source of vulnerability (how Proton Docs/Sheets work). Proton Docs/Sheets apply a cryptographic protection mechanism to wrap plaintext document updates for its claimed “end-to-end security” guarantees. At a high level, when an editor generates a plaintext update, the update is wrapped following a “sign-then-encrypt” paradigm, using the editor’s signing key and the document’s symmetric encryption key. The resulting ciphertext, together with the editor’s identifier and additional metadata, as well as an identifier for the update, forms an update message that is sent to the Proton server and relayed to other collaborators. Upon receiving an update message, other users apply the corresponding verification and decryption procedures to unwrap the message and recover the plaintext update. For brevity and clarity, we refer to users who share a document as belonging to the same *group*. The detailed procedures WrapMsg and UnwrapMsg are illustrated in Fig. 7 and 8.

Concretely, each plaintext update Upd is associated with a set of metadata ad, including a generation timestamp ts , the editor’s email address $email$, and the current system version number vn (which defaults to 1).

The WrapMsg procedure first signs the concatenation of an update and its associated metadata Upd||ad using the Ed25519 signature algorithm under the editor’s private key sk_{addr} , yielding a signature $\sigma = \text{Sign}(sk_{addr}, \text{Upd}||\text{ad})$. Subsequently, the plaintext update Upd and its signature σ are encrypted using an authenticated encryption with associated data (AEAD) scheme, with associated data ad, resulting in a ciphertext $C \leftarrow \text{AEAD.Enc}(ek, \text{ad}, \text{Upd}||\sigma)$. More specifically for the AEAD.Enc algorithm, a 32-byte random salt $salt$ is first generated and used to derive an encryption key from ek via the HKDF-SHA-256 algorithm, as $k = \text{HKDF}(ek, salt)$. The plaintext Upd|| σ is then encrypted using the AES-GCM algorithm with associated data ad and a random nonce $nonce$, producing the ciphertext $salt||c$, where $c \leftarrow \text{AES-GCM.Enc}(k, nonce, \text{ad}, \text{Upd}||\sigma)$. It also generates a 128-bit random number ID_{msg} as the identifier

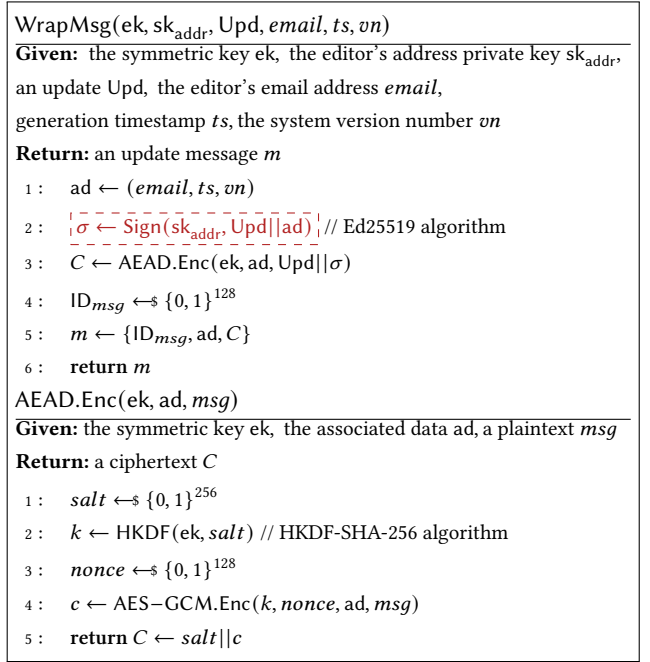


Figure 7: The WrapMsg procedure of wrapping a plaintext update. The source of vulnerability is highlighted.

for the update. Finally, the identifier ID_{msg} , the associated data ad, and the ciphertext of the update C form the update message.

Upon receiving an update message, other collaborators execute UnwrapMsg: first compute the encryption key as $k = \text{HKDF}(ek, salt)$ and decrypt to get $\text{Upd}||\sigma \leftarrow \text{AES-GCM.Dec}(k, \text{ad}, c)$, verify the embedded signature, and, if verification succeeds, accept and apply the update to recover the document state.

Fig. 7 and 8 highlight the source of this vulnerability, where we can see that the core reason is that each update is not bound with the document via its signature. We notice that such a design flaw of missing binding to context information (e.g., the sender and recipient identities) has been recognized in secure messaging [15]. However, the attack can work not only due to the lack of binding between updates and documents, but also due to the insufficient binding of the updates in Yjs.

Concrete attack. The attack begins by manipulating an *update message* that originates from the victim (say m_V) in a document Doc 1, to be legitimately put to a target document Doc 2, *exploiting a vulnerability in the WrapMsg procedure*.

To ensure that the manipulated update message is compatible with the underlying CRDT and can be applied at an arbitrary position chosen by the attacker in Doc 2, the attacker *injects one additional update message* with the wrapped plaintext update containing carefully chosen identifiers and lengths. After these update messages are forwarded by the Proton server to the collaborators of Doc 2, all collaborators see m_V as if it were authored by the victim on Doc 2. The attack is demonstrated step-by-step below.

Assume that an attacker corrupts the user $\mathcal{U}_A$ with a key pair (sk_A, pk_A) . The victim $\mathcal{U}_V$ has a key pair (sk_V, pk_V) . $\mathcal{U}_A$ and $\mathcal{U}_V$

¹⁵We describe only the necessary conditions for the attack to succeed. To make it more meaningful in practice (e.g., avoid easy deniability), $\mathcal{U}_V$ should have write access to Doc 2.

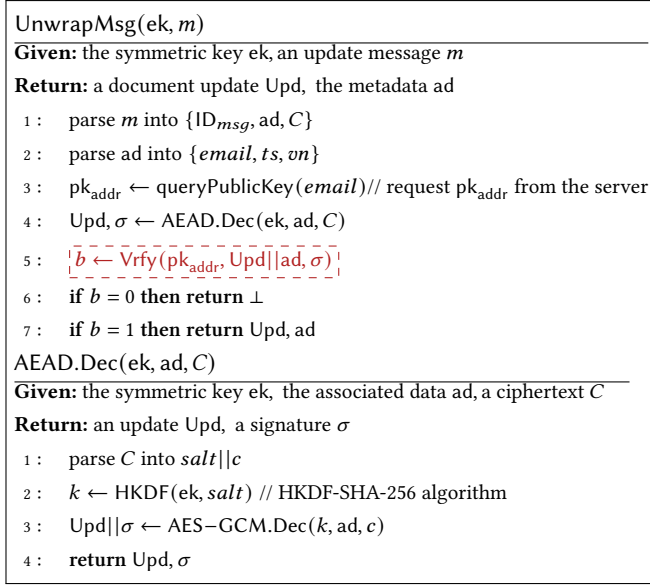


Figure 8: The UnwrapMsg procedure of unwrapping an update message. The source of vulnerability is highlighted.

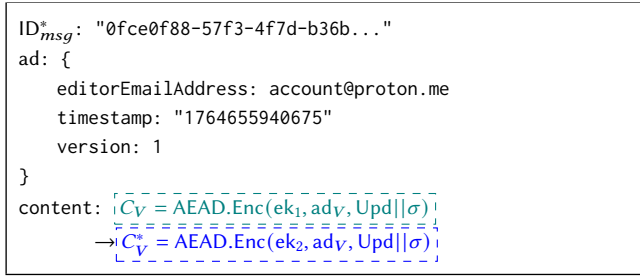


Figure 9: Manufacturing $\mathcal{U}_V$'s update message, where the boxed green part belongs to the original message from $\mathcal{U}_V$, and the boxed blue part belongs to the modified message.

share Doc 1 which is encrypted under a symmetric key ek_1 . $\mathcal{U}_A$ intends to "impersonate" $\mathcal{U}_V$ to make modifications to Doc 2 which is encrypted with ek_2 and to which $\mathcal{U}_A$ has write access.

1) *Manufacture an update message — maliciously wrapping updates for the target document.* Specifically, in Doc 1, there exist multiple update messages authored by $\mathcal{U}_V$. The attacker $\mathcal{U}_A$ selects one such update message $m_V = \{ID_{msg}, ad_V, C_V\}$ and unwraps it using ek_1 by executing $\text{UnwrapMsg}(ek_1, m_V) \rightarrow (Upd_V, ad_V)$, thereby recovering the plaintext update Upd_V , the associated metadata ad_V , as well as the internal transcript containing $\mathcal{U}_V$'s signature σ_V .

The attacker then "forges" a new update message for the target document Doc 2 by maliciously re-wrapping the recovered update Upd_V . Concretely, the metadata ad_V and the signature σ_V are kept unchanged. The attacker re-runs the AEAD encryption algorithm under the symmetric key ek_2 of Doc 2, obtaining a ciphertext $C_V^* \leftarrow \text{AEAD.Enc}(ek_2, ad_V, Upd_V || \sigma_V)$, as shown in Fig. 9.

Next, the attacker samples a fresh random identifier ID_{msg}^* and constructs a manipulated update message $m_V^* = \{ID_{msg}^*, ad_V, C_V^*\}$. According to the cryptographic unwrapping procedure, all collaborators of Doc 2 can successfully execute $\text{UnwrapMsg}(ek_2, m_V^*) \rightarrow (Upd_V, ad_V)$, thereby recovering $\mathcal{U}_V$'s original update Upd_V and accepting it as a valid update for Doc 2.

2) *Make forged updates applicable — planting bridge updates message with specified identifiers to match Proton's document update mechanism.* An update is represented by a specified data structure, described in Section 2. We observe that it is unnecessary for the attacker to replicate the entire document state; instead, it suffices to ensure that the relevant `clientId` and `clock` values of Upd_V appear in the target document's history. The attacker can locally run a Yjs instance to generate an update with manipulated `clientIDs` and then wrap it into an update message. Below, we illustrate how to make the update Upd_V wrapped in the "forged" update message m_V^* visible in the target document Doc 2.

To this end, the attacker introduces one additional update message m_A to "bridge" the current state of Doc 2 and the manipulated update message m_V^* , so that Yjs's update validation, based on the structural consistency of `clientId`, `clock`, `origin`, and `rightOrigin`, can be satisfied for Upd_V in Doc 2.

Concretely, there are two cases depending on the `origin` and `rightOrigin` fields of Upd_V .

Case 1: The `origin`: $\{clientId_V, clock_o\}$ and `rightOrigin`: $\{clientId_V, clock_r\}$ of Upd_V share the same `clientId_V`. In this case, the bridge update should contain continuous insertions each of whose identifier matches that of `clientId_V`, and the inserted content has a length exceeding the maximum of `clock_o` and `clock_r`. If `clock_o` < `clock_r`, the bridge update can be generated by typing the content from left to right with `clientId_V`. Otherwise, it can be generated by typing the content from right to left.

Case 2: The `origin` and `rightOrigin` of Upd_V have different `clientIDs`. In this case, the bridge update should contain at least two insertions: one matching Upd_V 's `origin` and the other matching its `rightOrigin` in terms of `clientId` and `clock`.

The `origin` and `rightOrigin` fields of these insertions can be chosen by the attacker based on the position he intends to put the insertion in Doc 2. The inserted content of the bridge update is likewise fully controlled by the attacker. With the plaintext update, the attacker can wrap it using the symmetric key ek_2 of Doc 2 and the corrupted private key sk_A to get the new update message m_A .

As shown in Fig. 10, the forged update Upd_V specifies <145268301, 5> as its `origin` and <145268301, 6> as its `rightOrigin`. Accordingly, the bridge update Upd_A contains a single insertion whose `clientId` is set to 145268301 and whose inserted content has length at least 7 (since the clock starts from 0).

3) *Send the manufactured update message and the bridge update to other collaborators via the Proton server.* After obtaining both the manipulated update message and the bridge update that can be applied correctly, the remaining challenge is to deliver these messages to the Proton server so that they are accepted and forwarded to the collaborators of Doc 2. As the server does not check whether the update message is submitted by its author, any client with write access can submit update messages on behalf of other users.

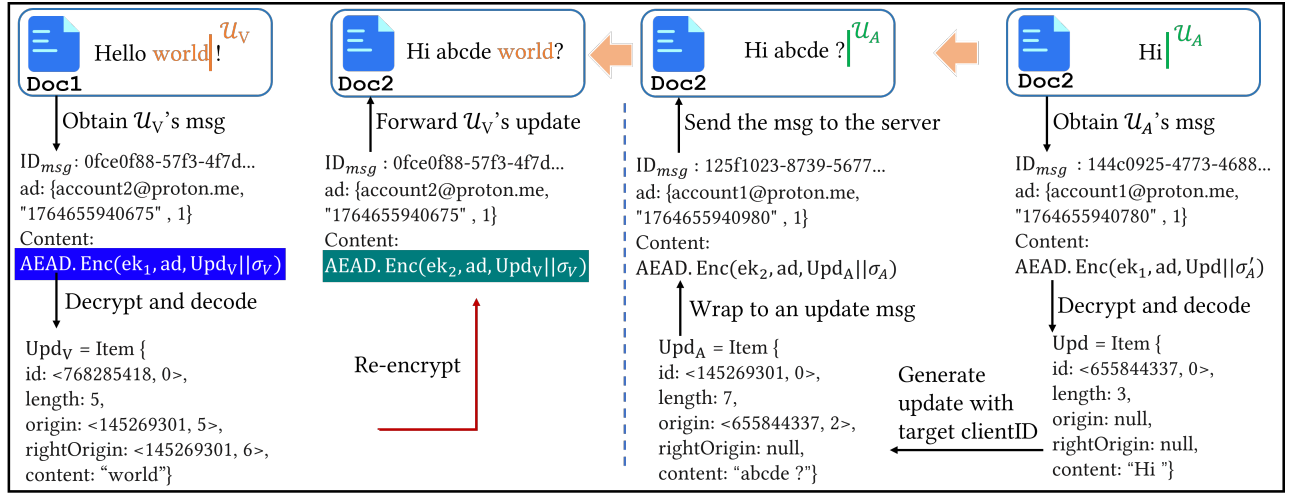


Figure 10: The procedures of the context manipulation attack in the editing mode.

Consequently, the attacker U_A can directly send both the manufactured update message m_V^* attributed to U_V and the bridge update message m_A wrapped from Upd_A through their own client. The server accepts these messages and forwards them to collaborators of Doc 2 without raising any alerts. And users of Doc 2 thought m_V^* is from U_V on Doc 2.

Extension to Proton Sheets. Context manipulation attacks can also be applied to spreadsheet editing with additional care, besides the need to plant “bridge” update messages (as for Docs) with the specified identifiers (step 2 above). The attacker not only has to facilitate origin and rightOrigin to match, but also needs to take other update parameters into account (detailed in Section 3.3). Note that an update generated by first inserting a cell in a row is difficult to be applied to other spreadsheets, since it links to the Items that initialize a row and the entire spreadsheet. To make such updates applicable, the bridge updates should include Items used to initialize the row/spreadsheet with desired identifiers, while it may cause conflicts due to the existence of previous initialization Items. The attacker can forward updates of the inserted cells in already-initialized rows to another spreadsheet. To make them applicable, the attacker only needs to plant bridge updates with specified identifiers, which enable the manufactured updates to be inserted into the doubly linked lists of the current row and some global modules. It is similar to the attacks on Proton Docs.

4.2.2 Context manipulation attacks in the suggesting/comment mode. An attacker can forward a victim’s modifications under the suggesting mode or comments from the original document to the target one, and other users in the target group would explicitly see them. Compared with the editing mode, the attacker additionally needs to compromise the server to let the forwarded comment messages be accepted by the server and users sharing the target document.

Attacker model. The attacker model is the same as above (of editing mode), except for one difference: the attacker is additionally allowed to control the server to prevent it from performing some

checks. This attack does not need a fully malicious server (e.g., a server is fully corrupted), and it only requires being “lazy”.

Source of vulnerability (how Proton Docs/Sheets work). At a high level, when a user makes a new modification/comment to a document under the suggesting/comment mode, it triggers two operations: 1) a plaintext comment with a unique identifier is generated and wrapped to an encrypted comment message; 2) a corresponding Yjs update containing the identifier is generated and wrapped to an update message using WrapMsg. Both of them are sent to the server.

Fig. 11 illustrates the procedures of WrapCmt and UnwrapCmt, corresponding to wrapping a user’s comment and processing comment messages received from the server. A plaintext comment Cmt is associated with a set of metadata, including a 128-bit randomly-chosen identifier ID_{cmt} and the editor’s email *email*.

The WrapCmt procedure first signs a comment and its associated metadata $Cmt || ad$ under the editor’s address private key. Then, a plaintext comment Cmt and the signature are encrypted with the document symmetric key with associated data ad using AEAD. When a user makes a modification/comment, a new comment thread will be created, by sending ID_{cmt} , the editor’s email address *email*, the comment ciphertext C_{Cmt} , and the document name to the server; then the server assigns a 128-bit identifier for the comment thread. When opening a document, all comment threads are retrieved from the server, and the comment ciphertext is decrypted using UnwrapCmt and then applied to the document.

In the suggesting mode, the insertions right after another character that has already been inserted, or the deletions of characters that have already been inserted, belong to the same comment thread. Modifications to an existing thread only generate update messages and do not change the original comment message. When opening a document, the comment content shown on the webpage will be updated according to the update messages in the same thread.

Concrete attack. The attack starts by manipulating a comment that originates from the victim U_V in the original document Doc 1 but

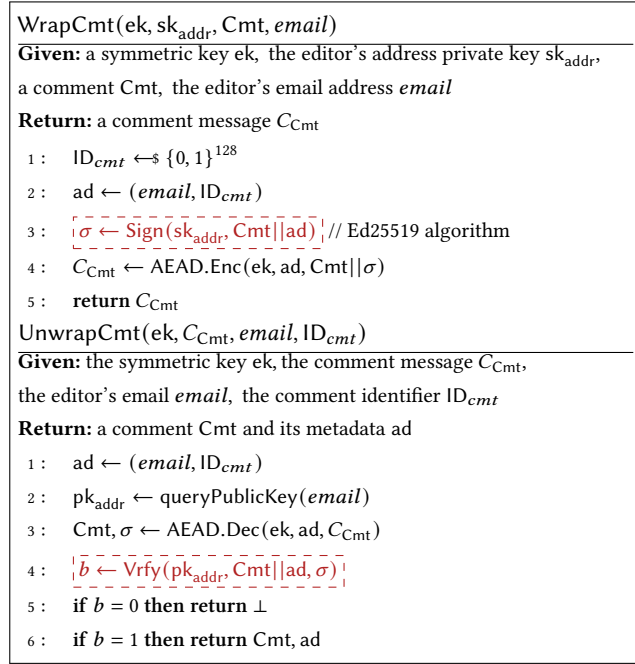


Figure 11: The WrapCmt and UnwrapCmt procedures. The source of vulnerability is highlighted.

is displayed to collaborators of a target document Doc 2. The attack leverages the vulnerability in WrapCmt (the highlighted text in Fig. 11). Since there is also an update message corresponding to the comment, the attack can be mounted using context manipulation in the editing mode. The attack is demonstrated step-by-step below.

Assume that an attacker corrupts the user $\mathcal{U}_A$ with a key pair (sk_A, pk_A) and controls the server. The victim is the user $\mathcal{U}_V$ with a key pair (sk_V, pk_V). $\mathcal{U}_A$ and $\mathcal{U}_V$ share a document, Doc 1, which is encrypted under a symmetric key ek₁. $\mathcal{U}_A$ intends to forward $\mathcal{U}_V$'s edits in the suggesting/comment mode to a target document, Doc 2 (encrypted with ek₂), to which $\mathcal{U}_A$ has write access.

1) *Manufacture a comment message - maliciously wrapping comments for the target document.* Assume that there is a comment thread {C_{Cmt}, email_V, ID_{Cmt}} in Doc 1 authored by $\mathcal{U}_V$. $\mathcal{U}_A$ first unwraps it with ek₁ by executing Cmt, ad ← UnwrapCmt(ek₁, C_{Cmt}, email_V, ID_{Cmt}). During the process, $\mathcal{U}_A$ obtains the $\mathcal{U}_V$'s signature σ_V on Cmt || ad. $\mathcal{U}_A$ then “forges” a new comment message for Doc 2 by re-wrapping {Cmt, email_V, ID_{Cmt}}. Specifically, the attacker runs the AEAD encryption algorithm with ek₂, yielding a ciphertext C_{Cmt}^{*} ← AEAD.Enc(ek₂, ad, Cmt || σ_V). According to UnwrapCmt, the collaborators of Doc 2 can successfully recover the original comment by decrypting C_{Cmt}^{*} and verifying σ_V .

2) *Create a new comment thread in the target document - sending the manufactured comment message to the server.* The attacker generates a new comment thread for Doc 2 by sending {C_{Cmt}^{*}, email_V, ID_{Cmt}} to the server on behalf of $\mathcal{U}_V$. The server is supposed to check whether email_V is the attacker's email address, but a corrupted or “lazy” server simply skips it and continues.

3) *Manipulate the update message corresponding to the forged comment - mounting the context manipulation attacks on the update message.* $\mathcal{U}_A$ can directly run the context manipulation attack described in Section 4.2.1 on the update message, such that all collaborators of Doc 2 can see that $\mathcal{U}_V$ has made such a modification on Doc 2 with explicit authorship of $\mathcal{U}_V$ shown.

The aforementioned attack shows how to manipulate context, leveraging a protocol design flaw. Additionally, there is an “easier” method to misattribute authorship in the suggesting mode. Specifically, any insertion immediately following a previously modified character in the suggesting mode is automatically attributed to the author of the adjacent character. Thus, a straightforward way for the attacker to break authorship is to edit content following the victim's editing in the suggesting mode. Then all collaborators will see that the attacker's changes are attributed to the victim, unless they manually check the authorship contained in the update messages.

4.3 Censorship attacks

High level intuition. Censorship attacks show that a malicious server can immediately launch *content*-based censorship and block content from some victim users (say $\mathcal{U}_V$) without being detected. To mount the attack, the corrupted server can adopt slightly different methods to *drop* some *update* messages before pushing the edits to users/clients, depending on their online status. The resulting document that other honest users see would differ from the version created and seen by $\mathcal{U}_V$ (e.g., with certain text removed). Moreover, neither the victim nor other honest users would be aware that their views of the document had been manipulated. No errors or warnings would be triggered by existing security checks, as the remaining updates are properly signed and encrypted. By also corrupting one user with read access, the attacker (the malicious server) can view the update plaintext and strategically choose which updates to drop for the honest users. Consequently, the remaining updates still pass all security checks and remain logically connected to the existing document history for both the victim and other honest users.

Impacted security in the context of Proton Docs/Sheets. Since Proton Docs/Sheets claim to provide E2EE security, even if the server is corrupted, the document content should not be leaked or tampered (otherwise detected). This is yet another form of attack on *integrity*¹⁶ that end-to-end encrypted Docs/Sheets would provide. While this attack will demonstrate that the Proton server could *modify document content* without generating any alert to the users, even if cautious users check digital signatures of all updates.

For the victim users, their clients would still show all the messages they put. For currently online users, they see a version with certain text removed, while they will not realize it. For a user who just comes back online (and may have closed the Proton Docs application tab), the client would need to request synchronization to retrieve all updates from the server. In this case, the server can safely drop any intermediate updates in the response (i.e., even remove some earlier text not in the latest updates).

¹⁶We would remark that this is different from the easy-to-identify denial of service. Proton Docs/Sheets enable clients to periodically ping the server to check network status.

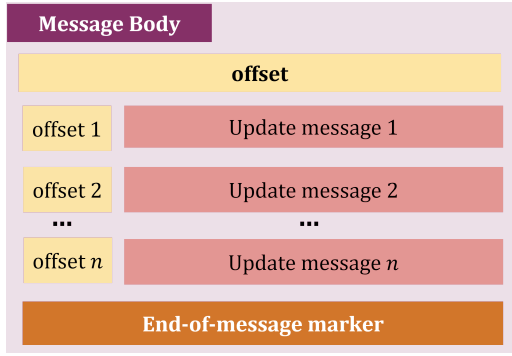


Figure 12: The format of the Proton server’s response to the users who have just come online and open documents.

Examples. The consequence of content-based censorship is obvious. For instance, suppose the users edit a document containing information, such as a report exposing negative information about the E2EE Docs/Sheets provider, in this case, Proton. The server can effectively block certain text, while the users will usually not even notice.

Attacker model. We assume the Proton server is corrupted, and users are honest. If we want censorship to be targeted, based on content, the attacker also needs to corrupt a user with read access.

Source of vulnerability (how Proton Docs/Sheets work). The Proton server acts as a central communication hub that enables collaborators to synchronize their document edits. Consequently, a user’s view of the document is entirely determined by the update messages delivered by the Proton server, which is *the main source of vulnerability*.

As described in Section 2, the Proton server encapsulates update messages in different formats depending on the user’s online status. For users who are already online, each update message is sent directly to them via WebSocket without additional reformatting. For users who were previously offline and have just come online, all existing update messages are instead wrapped in a batch format, as illustrated in Fig. 12. Concretely, multiple update messages are encapsulated into a single message body, which begins with the total length, denoted as an offset, and ends with an end-of-message marker. The individual update messages are placed sequentially in the body, each preceded by its own length. The resulting formatted message is used as the message body of an HTTPS response and is delivered to the requesting user via the HTTPS protocol.

Another source of vulnerability is that unapplied updates are allowed to be preserved for later application. Concretely, the underlying CRDT reconstructs the plaintext document by invoking the ApplyUpdate algorithm on a sequence of updates, determining whether and where each update should be applied based on its origin and rightOrigin. For updates that cannot be applied immediately, clients do not discard them. Instead, to tolerate real-world network latency and instability (e.g., packet loss or reordering), clients retain such unapplied updates locally and attempt to

apply them again when subsequent updates arrive. As a result, updates that are initially inapplicable may later become valid and be incorporated into the document state.

Concrete attack. An attacker who corrupts the Proton server can create divergent document views among collaborators by arbitrarily withholding/dropping update messages but wrapping the remaining updates under the aforementioned formats depicted in Fig. 12 to avoid detection of basic client checks on the updates. To ensure the censorship is effective and “reasonable”, the attacker can also corrupt one user to obtain document content, thereby carrying out more *targeted* censorship attacks.

Concretely, when a victim comes online and invokes OpenDoc, the server selectively drops update messages and encapsulates the remainder within the HTTPS response. For online users, the server selectively pushes new update messages via WebSocket. *This selection is strategic both for censorship and evading detection*: the server drops both the primary censored content and any *dependent* updates that cannot be applied without it. In doing so, the server precisely controls the scope of the attack, ensuring the resulting document views remain self-contained and “valid” to the victim and other honest users without triggering easy-to-spot conflicts.

As a result, victims will never receive the updates dropped by the server, regardless of whether they remain online or just come back online. This behavior arises because the document-opening procedure, shown in Fig. 1, does not verify the integrity or completeness of all update messages forwarded by the server.

4.4 Other vulnerabilities

In addition to the aforementioned attacks, we also discovered some minor issues towards Proton Docs/Sheets.

Cryptographic misuses. We discover two cryptographic misuses in the key hierarchy. (1) It uses the same public and private key pair for both encryption and signing, which is not generically secure. That said, whether it is fine for the specific schemes used in Proton also requires a more careful and rigorous analysis. (2) The signature on a secret key is directly revealed to the server, (i.e., $\text{sign}(k_1, k_2)$ for the Ed25519 signature scheme) while the signature itself does not provide any guarantee of confidentiality, thus has further risk of leaking some information of the secret key.

Document name leakage. The implementation of Proton Docs/Sheets exposes document names directly to the server in two aspects. First, Proton Docs/Sheets enable users to invite an existing user or an external user (who has not registered) to share a document/folder. A document/folder owner can send an invitation email to the targeted user via the Proton server (detailed in Appendix D.4). The owner can choose whether to write an email, which is sent to the collaborator along with the document name. We notice that the email content is sent to the server in plaintext form. If a user chooses to write an email and send it along with the invitation, then the email content and the document/folder name would be leaked to the server. Second, when editing in the suggesting mode or writing comments, document names are sent directly to the server as part of requests to create comment threads.

5 Mitigation Suggestions

Resist history rewriting attacks. One immediate option is to abandon the squash operation all together, which may hinder efficiency, particularly on the server side. To securely support squash while preserving original authorship and realizing the “unforgeability” of document history would require a more systematic study that involves both rigorous modeling and analysis, and it is one of major problems to study next.

In general, introducing a checking mechanism for each squash operation could help, to ensure that the operation is done correctly. One possible way is to ask every collaborator to check (consistency before and after squash), and the server accepts it only when *all* collaborators complete the checking. This is unlikely to be practical, as some collaborators may be offline for a long time, and the server needs to keep sufficient records till the last collaborator gets online and completes the checking. Another possible way is to deploy some kind of byzantine consensus among all *online* collaborators on the squash operation and jointly endorse each operation.

Resist context manipulation attacks in the editing mode. An immediate remediation is to let the Proton server check whether the identity of the user who submitted the update is the same in the signed update message (e.g., email). This may still weaken the overall claim of E2EE security though.

The context manipulation attacks can be more thoroughly limited through the following cryptographic method without any trust in the server. The unique document identifier should be bound to each document update, i.e., add the document identifier to the associated data of the corresponding update, and sign the full associated data and the update together to bind the document identifier to each update message.

Resist context manipulation attacks in the suggesting/comment mode. The unique document identifier should be bound with comments and signed together. Furthermore, authorship should be determined by the update messages’ signatures, rather than the comments’ signatures. In general, defining integrity models to properly capture those threats and demonstrate provable security for natural defenses is needed as a foundation for E2EE collaboration.

Resist censorship attacks. One immediate remedy is that users/clients can deploy a trigger mechanism: if the number of updates that *cannot be applied* at the client exceeds a predetermined threshold or no updates are received for a long time, a warning is triggered. The threshold setting is tricky, as it must tolerate network latency while enabling users to detect the server’s malicious behavior as soon as possible. In addition, each user needs to locally deploy an updatable authenticated data structure (e.g., a Merkle tree) to generate a tag (e.g., the root hash) for all updates they receive and periodically send to other clients to check consistency.

Fix misuse issues. For the cryptographic misuses, each public and private key pair should be only utilized for either encryption or signing, not for both of them. The signature of a secret should not be sent to the server. We suggest encrypting the signature and sending the ciphertext to the server, or first encrypting a secret and then signing on the ciphertext.

For the document name leakage in the share setting, Proton Docs/Sheets can encrypt a shared document name under an internal

sharee’s public key and send the ciphertext to the server, and the sharee decrypts it on the client side. When inviting an external user, Proton should exclude document names from invitation emails. For document name leakage in the suggesting/comment mode, the requests to create comment threads already include the document identifier, and thus, the document names are not needed.

6 Proof of Concept

To validate the practicality of our attacks, we register three test accounts on Proton and conduct proof-of-concept (PoC) tests on the latest Proton Web Client (March 30, 2026).

All experiments are performed using test accounts only, including malicious users and victims, and other involved honest users.

All data involved in the experiments were created solely for this study. We did not access or interfere with any third-party user accounts, or attempt to manipulate server-side storage, except for controlling communications between the test users and the server.

6.1 PoC setting

Under the attacker model, the history rewriting attack and context manipulation attack in the editing mode require the attacker to corrupt an honest user without controlling the server. Therefore, we implement their PoC tests by controlling a test user. The context manipulation attack in the suggesting/comment mode and the censorship attack require the participation of the server. Since the Proton server is not open-source and accessible, we simulate a malicious server by mounting MitM attacks.

Concretely, we install a bogus TLS root certificate on the test users’ clients, enabling us to deploy a proxy that impersonates the Proton server to the users while still communicating with the real server to execute its (unknown) server-side logic. To implement the proxy, we use Burp Suite, a widely used web proxy and traffic analysis tool, to intercept and manipulate communications between users and the real server. Under this setup, the proxy acts as a MitM by splitting the original TLS channel into two independent TLS connections: one between the client and the proxy, and another between the proxy and the real Proton server. During the experiments, the attacker can leverage the proxy to perform a range of malicious server behaviors, such as message dropping, blocking, relaying, modification, and selective forwarding.

6.2 PoC tests

Simulate the history rewriting attack. We simulate an attacker to tamper with a document’s history, with the following steps.

1) *Adjust the squash trigger threshold and factor.* After logging in to Proton Docs and opening the target document, the attacker can leverage the browser’s developer tools to intervene in the execution, so that arbitrarily adjust the threshold and factor. Concretely, the attacker set breakpoints at the following three code lines:

- `return this.numberOfMessages() > GetCommitDULimit (documentType)` that invokes the squash threshold retrieving function `GetCommitDULimit` and compares the number of update messages with the threshold.
- `const squashCommitResult = await this.squashThe Commit(..., dto.documentType)` that performs squash.

```

async encryptSquashResult(...):
Promise<Result<DocumentUpdate[]>> {
  ...
  let squashedContent = ...
  if (isDocumentUpdateCompressionEnabled(documentType)){
    ...
    squashedContent =
    compressDocumentUpdate(squashedContent)
    ...
  }
  const encryptedUpdate = await this.encryptMessage.execute(
    squashedContent,...)    ...
}

```

Figure 13: Source code of encrypting the squashed result.

- `const desiredNumberOfUpdates = Math.floor(config.limit * config.factor)` that takes the threshold and the factor as input to compute the number of updates to be kept.

The attacker then reloads the page, causing execution to pause at the first breakpoint.

In JavaScript, variables roughly fall into two broad categories: primitives and objects. (1) primitive types (including number, string) are stored directly, immutable, and reassigning them changes the variable; (2) object types (including function) are stored by reference, and usually can be modified by reassigning the reference before being called.

Assume `GetCommitDULimit` is stored by a reference identified by a character “X”. The attacker can redefine `GetCommitDULimit` in the Web client by running the code `X = function modifyThreshold () {return  $t^*$ }` at the developer console, which enables it to return a threshold t^* less than the number of document updates. After modifying the function, the program continues to execute and pauses at the second breakpoint, where the attacker changes the document type by running `dto.documentType = “sheet”`.¹⁷ The program continues to execute and pauses at the third breakpoint, where the attacker modifies the factor to a desired value α^* by running the code `config.factor =  $\alpha^*$` at the console.

2) *Modify the merged update.* The attacker sets a breakpoint at the code line `updatesToSquash.map((update)...)` that takes the updates to be merged as input. When the execution pauses, the attacker extracts the updates to be squashed and decodes them locally using Yjs-provided interfaces, which enables inspection of the concrete modifications they contain. Given the decoded plain updates to be squashed, the attacker runs a local program to selectively keep certain updates, insert/delete some updates, and invokes the `MergeUpdate` interface to obtain a modified merged update Upd_{mrg}^* .

¹⁷This breakpoint and step are optional. As described later, this breakpoint and step are used to inject the false merged update when its length differs from the correct one. If they have the same length, please skip the step. The attacker can first determine whether the lengths are equal. If the attacker only wants to replace some content, with no new insertions or deletions, then the lengths are equal with high probability; if the attacker wants to insert new content or delete, the lengths are not equal.

3) *Send the squashed result to the server.* Fig. 13 shows the source code of encrypting the merged update represented by squashed Content. The attacker sets a breakpoint at the code line `if (isDocumentUpdateCompressionEnabled(documentType))`, and the program continues to execute and pauses at the above breakpoint. `TypedArray`, `squashedContent` in this type, is a special object stored in a fixed-length buffer. In general, this type of variable can only modify its content without changing its length. We notice a method that can modify `squashedContent` while changing its length. As shown in Fig. 13, we can see that `squashedContent` would be compressed with `gzip` if `isDocumentUpdateCompressionEnabled` returns true. Thus, we can let `isDocumentUpdateCompressionEnabled` return true,¹⁸ so that `compressDocumentUpdate` is executed before encryption, and by assigning a new reference to `compressDocumentUpdate`, we can set `squashedContent` as the modified squashed result. In the following, we provide detailed steps for modifying `squashedContent` in the two cases.

Case 1: The length of the correct merged update is the same as that of Upd_{mrg}^* . The attacker runs the code `squashedContent.set(Upd $_{mrg}^*$ )` at the developer console.¹⁹ The Web client execution is subsequently resumed, allowing the client to cryptographically wrap the modified merged update and send the squashed result to the server.

Case 2: The length of the correct merged update is not equal to that of Upd_{mrg}^* . In this case, the attacker cannot directly modify the merged update in the console because the length has changed. The attacker can modify `squashedContent` by changing the outputs of the `isDocumentUpdateCompressionEnabled` and `compressDocumentUpdate` functions. If the document type is “sheet”, then `isDocumentUpdateCompressionEnabled` will return true. The attacker can set a breakpoint at the code line `squashedContent = compressDocumentUpdate(squashedContent)`, which performs the `gzip` compression. Assume the `compressDocumentUpdate` function is identified by “K”. The program continues to run and pauses at the breakpoint above. The attacker can redefine the function by running the following code at the console to modify `squashedContent`.

```

K = function modifyMrg () {
  const forgeMerg = new Uint8Array(Upd $_{mrg}^*$ );
  return forgeMerg;
}

```

Then, the attacker enables the program to continue executing, which sends the modified squashed result to the server via the Web client.

Results of simulated attack. When other users open the target document, they see the tampered document content and editing history without getting any warning.

Simulate the context manipulation attacks in the editing mode. We simulate an attacker who attempts to forward an update message m_V generated by a victim in Doc 1 (encrypted with ek_1) to Doc 2 (encrypted with ek_2). The attack proceeds as follows.

¹⁸If `documentType = “doc”` then `isDocumentUpdateCompressionEnabled` returns false, and it returns true when `documentType = “sheet”`.

¹⁹As described before, the attacker does not change `documentType` in this case, and the `compressDocumentUpdate` is not called.

Algorithm 1: Re-encrypt an update

```

1: Input: Two symmetric keys  $ek_1, ek_2$ , an update message  $m$ ;
2: Output: A manufactured update message  $m^*$ ;
3: function ManufactureMsg ( $ek_1, ek_2, m$ )
4:   parse  $m$  into  $\{ID_{msg}, ad, C\}$ 
5:    $Upd, \sigma \leftarrow \text{decryptData}(C, ad, ek_1)$ 
6:    $salt \leftarrow \$\text{getRandomValues}(32 \text{ bytes})$ 
7:    $k \leftarrow \text{deriveGcmKey}(ek_2, salt)$ 
8:    $C^* \leftarrow \text{gcmEncryptWith16ByteIV}(k, ad, Upd || \sigma)$ 
9:    $ID_{msg}^* \leftarrow \$\text{GenerateUUID}()$ 
10:   $m^* \leftarrow \text{CreateDocumentUpdateMessage}(ID_{msg}^*, ad, C^*)$ 
11:  return  $m^*$ 
12: end

```

1) *Manufacture an update message.* We first implement a local program by invoking the interfaces provided in the open-source Proton Web client. With the local program executions, the update message m_V is decrypted with ek_1 , and the plaintext update Upd_V , as well as the unchanged metadata and the signature, is encrypted under ek_2 . The attacker obtains a manufactured update message m_V^* . Algorithm 1 presents the pseudocode of this program. Specifically, the program performs the following steps.

- Parse an update message m to extract the message identifier ID_{msg} , the associated data ad , and the ciphertext C ;
- Utilize `decryptData` to decrypt the update ciphertext C with ek_1 and obtain the plaintext update Upd and the signature σ ;
- Randomly choose a 32-byte $salt$ value using `getRandomValues`;
- Derive a key k from ek_2 and $salt$ with `deriveGcmKey`;
- Use k to encrypt ad , Upd , and σ with `gcmEncryptWith16ByteIV` and obtain the new ciphertext C^* ;
- Generate a new ID_{msg}^* from `GenerateUUID`;
- Wrap ID_{msg}^* , ad , and C^* into a new update message m^* by calling `CreateDocumentUpdateMessage`.

2) *Generate the bridge update with specified identifiers.* To make m_V^* applicable, the attacker needs to inject a bridge update with carefully manipulated `clientId` and `clock`. To this end, the attacker can locally instantiate the Yjs library and generate the desired update Upd^* by explicitly controlling `clientId` and `clock`.

3) *Send the manufactured update message and inject the bridge update to the server.* The attacker logs in to Proton Docs using a browser and opens the target document, after which the Web client establishes a WebSocket connection with the server. The attacker leverages this established WebSocket connection to transmit forged messages to the server.

To send the manufactured message m^* , the attacker sets a breakpoint in the browser at the code line `this.socket?.send(data)`, which is the socket sending interface. The attacker then types characters in the target document to trigger the web client to run the

update message generation procedure and pause its execution at the breakpoint. When execution pauses at this point, the attacker directly invokes the following code at the developer console to send m^* to the server.

```

const forgedmsg = new Uint8Array(m*)
this.socket?.send(forgedmsg)

```

To inject a bridge update Upd_A , the attacker sets a breakpoint on the code line `const updateToPropagate = this.createSyncMessagePayload(...)`, which is a preparation step before wrapping an update. Then the attacker triggers the update message generation procedure by typing characters in the target document. When the Web client pauses execution at the breakpoint, the `TypedArray` variable named `updateToPropagate` represents the original update. A straightforward way is to modify the original update with the bridge update at the developer console. As described in the first attack, this type of variable cannot be directly reassigned. In the following, we provide detailed steps for modifying `updateToPropagate` in the two cases.

Case 1: The length of the original update is the same as that of the bridge update Upd_A . The attacker can replace the update content with Upd_A by running the code `updateToPropagate.set(Upd_A)` at the developer console. The Web client execution is subsequently resumed, allowing the client to cryptographically wrap the crafted update and transmit it to the server.

Case 2: The length of the original update is not equal to that of the bridge update Upd_A . The attacker can assign a new reference to the `this.createSyncMessagePayload` function to modify `updateToPropagate`. When execution pauses at the breakpoint, the attacker runs the following code at the console.

```

this.createSyncMessagePayload= function modifyUpd () {
  const upd = new Uint8Array(Upd_A);
  return {type: { wrapper: 'du' }, content: upd, }
}

```

Then, the attacker resumes the execution. In this way, `updateToPropagate` represents the bridge update, and then is cryptographically wrapped and sent to the server. Upon receiving the update messages, the server forwards them to other online collaborators of the target document, and then they apply these update messages to their own document replicas.

Results of simulated attack. When other collaborators open the target document Doc 2, they will see the manufactured modification.

Simulate the context manipulation attacks in the suggesting/comment mode. We simulate an attacker with an email address $email_A$ who attempts to forward a comment message $\{C_{cmt}, email_V, ID_{cmt}\}$ and the corresponding update from a victim in Doc 1 (encrypted with ek_1) to a target document Doc 2 (encrypted with ek_2). The attack proceeds as follows.

1) *Manufacture a comment message.* We implement a local program that encrypts and decrypts comment messages by invoking the interfaces provided in the open-source Proton Web client. By running the local program, the attacker obtains a manufactured comment message $\{C_{cmt}^*, email_V, ID_{cmt}\}$, which can pass the collaborators' verification in Doc 2. Algorithm 2 presents the pseudocode

Algorithm 2: Re-encrypt a comment

```

1: Input: Two symmetric keys  $ek_1, ek_2$ , a comment message  $C_{Cmt}$ ,
2: the editor's email address  $email$ , the comment's identifier  $ID_{cmt}$ ;
3: Output: A manufactured comment message  $C_{Cmt}^*$ ;
4: function ManufactureCmt ( $ek_1, ek_2, C_{Cmt}, email, ID_{cmt}$ )
5:    $ad \leftarrow (email, ID_{cmt})$ 
6:    $Cmt, \sigma \leftarrow \text{decryptData}(C, ad, ek_1)$ 
7:    $salt \leftarrow \text{getRandomValues}(32 \text{ bytes})$ 
8:    $k \leftarrow \text{deriveGcmKey}(ek_2, salt)$ 
9:    $C_{Cmt}^* \leftarrow \text{gcmEncryptWith16ByteIV}(k, ad, Cmt || \sigma)$ 
10:  return  $C_{Cmt}^*$ 
11: end

```

of this program. Specifically, the program performs the following steps.

- Obtain the editor's email and the comment's identifier ID_{cmt} as the associated data ad ;
- Utilize `decryptData` to decrypt the comment message C_{Cmt} with ek_1 and obtain the plaintext comment Cmt and signature σ ;
- Randomly choose a 32-byte $salt$ value using `getRandomValues`;
- Derive a key k from ek_2 and $salt$ with `deriveGcmKey`;
- Use k to encrypt ad , Cmt , and σ with `gcmEncryptWith16ByteIV` and obtain the new ciphertext C_{Cmt}^* .

2) *Create a comment thread at the server.* Since the server checks whether the sender's email matches the editor's email, we simulate creating a new comment thread with $(C_{Cmt}^*, email_V, ID_{cmt})$ at the server by mounting MitM attacks as follows.

The attacker first queries the server to create a new comment thread for Doc 2 by sending $(C_{Cmt}^*, email_A, ID_{cmt})$ to the server. Specifically, the attacker sets a breakpoint at `const result = await this.api.createThread(...)` that requests creating a comment thread on the server. Then the attacker types a character in the suggesting mode, and the code pauses at the breakpoint. The attacker replaces the comment information to be sent with $(C_{Cmt}^*, email_A, ID_{cmt})$ by running the following code line at the console, and the execution continues.

```

id = IDcmt, forgecmt = CCmt*
const result = await this.api.createThread({
  Id: id, encryptedMainCommentContent: forgecmt,
  type: dto.type, commentType,
  decryptedDocumentName: dto.decryptedDocumentName, },
  dto.entitlements, )

```

When collaborators of Doc 2 pull the comment thread, the attacker intercepts the server's response by replacing the attacker's email address with the victim's.

3) *Forward the corresponding update message.* The attacker can follow instructions for simulating the context manipulation attacks

in the editing mode to forward the corresponding update message to the target document.

Results of simulated attack. When other collaborators open the target document, they will see the forged modification with a corresponding comment that originates from the victim.

Simulate the censorship attacks. We simulate a malicious server via deploying the MitM attack. We explain how to carry out the attack using the proxy in the following two cases.

1) *Attacking online users.* The attacker utilizes the proxy to arbitrarily block or drop the update messages sent by the online users. As a result, different users observe different document views.

2) *Attacking newly online users.* When a user opens a document and invokes the `getCommitDoc` interface, the Proton server returns an HTTPS response containing all update messages corresponding to the latest document version. With the proxy, the attacker intercepts the response and then selectively removes some update messages and their associated offsets in the HTTPS response message body. Next, the attacker updates the message body's start offset to reflect its current length. Finally, the modified HTTPS response is returned to the user. As a result, the user does not receive the latest version of the document, but rather a version that has had certain updates removed by the server.

Results of simulated attack. Users sharing the document will see an incomplete document without getting any warning.

Remarks. Our experimental steps involve operations on the browser console and modifications to network responses. During our tests, sometimes they may fail due to network issues or other unexpected reasons, and multiple attack attempts may be required.

7 Conclusion

In this paper, we have analyzed the cryptographic collaborative editing protocol of Proton Docs/Sheets and identified three integrity attacks: history rewriting, context manipulation, and censorship. The first two attacks can work even with an honest server. We have proposed corresponding mitigation measures that can invalidate those direct attacks (at the potential cost of reduced efficiency). Furthermore, similar protocol designs are adopted by many other E2EE collaborative editing systems, implying that the impact of these attacks may extend beyond Proton Docs/Sheets. Systematically developing solutions that provide provable security and high efficiency is thus an important direction for future research.

Acknowledgments

The work of Qiang and Ya-Nan was supported in part by Google via the Digital Future Initiative, Research Awards from the Stellar Development Foundation, and Input Output (iog.io) as part of the Cardano Vision 2026 research initiative. The work of Yuan was supported in part by the National Nature Science Foundation of China under Grant 62472074.

References

- [1] Martin R. Albrecht, Benjamin Dowling, and Daniel Jones. 2025. Formal Analysis of Multi-device Group Messaging in WhatsApp. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 242–271.

- [2] Martin R. Albrecht, Miro Haller, Lenka Mareková, and Kenneth G. Paterson. 2023. Caveat Implementor! Key Recovery Attacks on MEGA. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 190–218.
- [3] Anytype. [n. d.]. <https://anytype.io/>.
- [4] Matilda Backendal, Hannah Davis, Felix Günther, Miro Haller, and Kenneth G. Paterson. 2024. A Formal Treatment of End-to-end Encrypted Cloud Storage. In *Annual International Cryptology Conference*. Springer, 40–74.
- [5] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. 2023. MEGA: Malleable Encryption Goes Awry. In *IEEE Symposium on Security and Privacy*. IEEE, 146–163.
- [6] CryptPad. [n. d.]. <https://cryptpad.fr/>.
- [7] DekkoSecure. [n. d.]. <https://www.dekkosecure.com/dekkocore-interactive-demos>.
- [8] Andrés Fábrega, Armin Namavari, Rachit Agarwal, Ben Nassi, and Thomas Ristenpart. 2024. Exploiting Leakage in Password Managers via Injection Attacks. In *USENIX Security Symposium*. 4337–4354.
- [9] Andrés Fábrega, Carolina Ortega Pérez, Armin Namavari, Ben Nassi, Rachit Agarwal, and Thomas Ristenpart. 2024. Injection Attacks against End-to-end Encrypted Applications. In *IEEE Symposium on Security and Privacy*. IEEE, 2648–2665.
- [10] Fileverse. [n. d.]. <https://fileverse.io/>.
- [11] Aviv Frenkel and Dmitry Kogan. 2025. An Attack on TON’s ADNL Secure Channel Protocol. In *IEEE Symposium on Security and Privacy*. IEEE, 2002–2017.
- [12] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. 2016. Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage. In *USENIX Security Symposium*. 655–672.
- [13] Nadia Heninger and Keegan Ryan. 2023. The Hidden Number Problem with Small Unknown Multipliers: Cryptanalyzing MEGA in Six Queries and Other Applications. In *IACR International Conference on Public-Key Cryptography*. Springer, 147–176.
- [14] Jonas Hofmann and Kien Tuong Truong. 2024. End-to-end Encrypted Cloud Storage in the Wild: A Broken Ecosystem. In *ACM SIGSAC Conference on Computer and Communications Security*. 3988–4001.
- [15] Joseph Jaeger, Akshaya Kumar, and Igors Stepanovs. 2024. Symmetric Signcryption and E2EE Group Messaging in Keybase. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 283–312.
- [16] Ya-Nan Li, Yaqing Song, Qiang Tang, and Moti Yung. 2025. End-to-End Encrypted Git Services. In *ACM SIGSAC Conference on Computer and Communications Security*. 468–482.
- [17] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klammer. 2016. Near Real-time Peer-to-peer Shared Editing on Extensible Data Types. In *ACM International Conference on Supporting Group Work*. 39–49.
- [18] ONLYOFFICE. [n. d.]. <https://www.onlyoffice.com/>.
- [19] Proton. [n. d.]. <https://proton.me/>.
- [20] SNote. [n. d.]. <https://www.snote.app/>.
- [21] David Taylor, Thomas Wu, Nikos Mavrogiannopoulos, and Trevor Perrin. 2007. Using the Secure Remote Password (SRP) protocol for TLS authentication. RFC 5054. <https://www.rfc-editor.org/rfc/rfc5054>
- [22] Tresorit. [n. d.]. <https://tresorit.com>.
- [23] Kien Tuong Truong, Simon-Philipp Merz, Matteo Scarlata, Felix Günther, and Kenneth G. Paterson. 2025. Breaking and Fixing Content-defined Chunking. In *ACM SIGSAC Conference on Computer and Communications Security*. 2294–2308.
- [24] Paul Wouters, Daniel Huigens, Justus Winter, and Yutaka Niibe. 2024. OpenPGP. RFC 9580. <https://www.rfc-editor.org/rfc/rfc9580>
- [25] Andy Yen. [n. d.]. Introducing Proton Workspace: An encrypted suite for team collaboration. <https://proton.me/business/blog/proton-workspace>
- [26] YJS. [n. d.]. <https://docs.yjs.dev/>.

A Open Science

We provide a detailed description of the vulnerabilities, their sources, and the methodology used to validate the attacks. For ethical considerations, we do not provide the source code of the local execution programs of our PoC.

B Ethical Considerations

In this paper, we examine the security of a widely deployed E2EE collaborative document system, Proton Docs/Sheets, and identify three types of attacks. We carefully evaluate the ethical implications and provide a stakeholder-based ethics analysis as follows.

Stakeholders and impacts/harms. We identify the stakeholders and the potential impacts for them, following the principles from The Menlo Report.

Users of Proton Docs/Sheets. If the vulnerabilities are exploited in reality, the data integrity and even the confidentiality of users’ data may be affected. However, we believe that users ultimately benefit from our research, once vulnerabilities are identified and mitigated.

Proton’s vendors and developers. They may face short-term risks, including reputation damage and financial/time costs for fixing the vulnerabilities. Meanwhile, they may benefit from early identification of security vulnerabilities and have the opportunity to strengthen the security guarantee before attackers exploit them.

Research community. Our work provides a new perspective on potential security challenges of E2EE collaborative document systems, which may facilitate systematic research. However, publication may also lower the barrier for attackers.

The broader society. Society may benefit from more secure E2EE collaborative document systems, once vulnerabilities are identified and mitigated. Conversely, vulnerability disclosure may increase systemic risk.

Mitigations. We take the following measures to mitigate potential harms.

Responsible disclosure. We have reported the vulnerabilities to Proton and provided PoC code, reproduction steps, and mitigation suggestions. For de-identification in the CCS submission, we only reviewed the relevant technical details of our attacks and masked function names and identifiers that might indicate to Proton. The detailed timeline of responsible disclosure is provided in Section 1.2.

Controlled experiments. We designed our experiments carefully so that they involve no private data from the server, or real users, or third-party systems; and they were conducted in our fully controlled environment (which cannot affect other users’ accounts, only affecting our own accounts). It was conducted under our institution’s Research Code of Conduct (and relevant guidelines/regulations). Moreover, there are no human participants or personal data, and it is not human-subjects research. It thus requires no IRB/REC review (according to our universities’ policies). We also acted within Proton’s “Safe Harbor Policy” and followed responsible disclosure.

Limited PoC. To minimize the risk of direct exploitation and potential harm to users and Proton, we do not release a complete PoC. Instead, we provide sufficient details to understand and reproduce the key findings without exposing a readily deployable exploit.

Decision. We carefully considered whether to conduct and publish this work. We acknowledge that publishing this work may result in negative impacts or even harm to stakeholders, and we take mitigations. We believe publishing this work is justified, as it demonstrates the need for systematic security analysis in collaboration settings and motivates the community to further investigate.

C Generative AI Usage

We utilized ChatGPT and Gemini for grammar checking. We asked ChatGPT how to write a script that automatically modifies server responses and forwards them to clients in MitM attacks to simulate a malicious server. We adopted the provided script framework and added the functionalities we needed to it. We verified the script by running the experiments.

getNode(pk, sk)
Given: a public key pk, a private key sk
Return: another key pair sk', pk' , ciphertext of sk' Key, passphrase ciphertext pp, passphrase signature ppSig, a session key ppssk
1: $rawpp \leftarrow \{0, 1\}^{256}$
2: $sk', pk', Key \leftarrow \text{generateKey}(rawpp, \text{CURVE25519})$
3: $pp, ppssk \leftarrow \text{asymEncrypt}(pk, rawpp)$
4: $ppSig \leftarrow \text{sign}(sk, rawpp)$
5: return $sk', pk', Key, pp, ppSig, ppssk$
getEncName(name, pk, sk)
Given: a folder/document name name, a public key pk, a private key sk
Return: the encrypted name EncryptName, a session key ppssk
1: $\sigma_{name} \leftarrow \text{sign}(sk, name)$
2: $\text{EncryptName}, \text{namessk} \leftarrow \text{asymEncrypt}(pk, name \sigma_{name})$
3: return EncryptName, namessk
getPrimaryKey(pwd)
Given: a password pwd
Return: a primary key pair SK, PK, the ciphertext of SK Key
1: $salt \leftarrow \{0, 1\}^{128}$
2: $h \leftarrow \text{Blowfish}(pwd, salt) // \text{use Blowfish hash algorithm}$
3: $\text{Keypwd} \leftarrow h[30:] // \text{remove the first 29 characters}$
4: $SK, PK, \text{Key}_{pri} \leftarrow \text{generateKey}(\text{Keypwd}, \text{CURVE25519})$
5: return SK, PK, $\text{Key}_{pri}, salt$

Figure 15: The algorithms of Proton Docs/Sheets.

subfolder involved for checking name collisions, and each document has a symmetric encryption key ek used to encrypt its content.

Document/folder name encryption. A node's document or folder name is encrypted by its parent public key pk_p and the address private key sk_{addr} with the getEncName algorithm, as shown in Fig. 15. To encrypt and authenticate the folder/document name, name is first signed under sk_{addr} as σ_{name} , and the concatenation of name and σ is encrypted using asymEncrypt with pk_p , resulting in a ciphertext EncryptName and a session key namessk. We notice that each node's public/private key pair is used for both encryption and signature purposes.

D.3 The key hierarchy of nodes

Address node generation. A user's address node has a parent node, which has a primary keypair "derived" from the user's password. A registered user generates the primary keypair (SK, PK) by the getPrimaryKey algorithm, as shown in Fig. 15. Taking a user's password pwd as input, a 32-byte salt is randomly generated, and a key Keypwd is derived from pwd, salt with the Blowfish hash algorithm. Then, the primary keypair from CURVE25519 is generated under the protection of Keypwd. The encrypted private key Key_{pri} and salt are sent to the server. With the primary

keypair, the getNode algorithm is invoked to generate an address keypair, as $sk_{addr}, pk_{addr}, \text{Key}_{addr}, pp_{addr}, \sigma_{addr} \leftarrow \text{getNode}(PK, SK)$, and $\{pk_{addr}, \text{Key}_{addr}, pp_{addr}, \sigma_{addr}\}$ is sent to the server, and the server assign an identifier ID_{addr} for the email address. In Proton Docs/Sheets, only the users' address public keys are sent to the server.

Volume node generation. There is a volume node above the root folder, where its node keys are also generated by invoking $sk_{vol}, pk_{vol}, \text{Key}_{vol}, pp_{vol}, \sigma_{vol} \leftarrow \text{getNode}(pk_{addr}, sk_{addr})$, and $\{\text{Key}_{vol}, pp_{vol}, \sigma_{vol}\}$ is sent to the server.

Folder/document node generation. The detailed procedures createFolder, createDocument for creating a new folder/document are illustrated in Fig. 16. The two procedures take the parent node's public key pk_p and the HMAC key HashKey_p as input, which can be retrieved according to the key hierarchy.

Create a new folder. The procedure first computes an HMAC-SHA256 value Hash on the folder name name with HashKey_p , generates the node keys by invoking $\text{getNode}(pk_p, sk_{addr})$, and folder name is encrypted with the getEncName algorithm. A 32-byte HMAC key HashKey is randomly generated for the new folder, where HashKey is signed under sk as σ_h , and $\text{HashKey} || \sigma_h$ is encrypted using asymEncrypt with pk, resulting in a ciphertext NodeHashKey. Finally, the queryCreateFolder interface is invoked to create a new folder on the server, and the server returns a unique ID for it.

Create a new document. Unlike creating a folder, Proton does not support choosing document names when creating documents; instead, a default document name is generated from the current timestamp ts. The second through fourth steps are the same as the first three steps for creating a folder, which compute an HMAC value, generate the node keys, and encrypt the document name. Then, a session key ek is randomly generated and used as the symmetric encryption key. ek is signed under sk and encrypted with pk, respectively, which results the signature ContentKeyPacketSig and the ciphertext ContentKeyPacket. A signature on an empty array under sk_{addr} is computed, since the current document is empty. Finally, the queryCreateDoc interface is called to create a document on the server, and the server returns a unique ID for the new document. When opening documents, users can obtain the document encryption key by decrypting layer by layer according to the key hierarchy.

D.4 The key hierarchy of sharing

Proton Docs/Sheets support read-write access separation, and users with write permissions can modify the document. Only the document owner can share documents/folders with others, specifying the collaborators' permissions, which are mostly enforced by the Proton server's access control.

The sketch of sharing. Before inviting collaborators, the owner first generates a share node for the target folder or document. Fig. 17 demonstrates the key hierarchy of the original folder/document node and the share node. Similar to generating node keys for a document or folder under its parent node, creating a share node requires the owner to generate a node, including a passphrase and a key pair, under the current node's public key, and the newly generated public key is used to encrypt the session keys used to encrypt the

```

createFolder(ParentID, pkp, HashKeyp, email, skaddr, name)
Given: the parent identifier ParentID, the parent public key pkp,
the parent HMAC key HashKeyp (optional input),
the creator's email and address private key email, skaddr
the folder name name
Return: the folder identifier ID
1: if (HashKeyp)
2:   Hash ← HMAC-SHA256(HashKeyp, name)
3: else if Hash = null
4:   sk, pk, NodeKey, Nodepp, NodeppSig, Nodeppssk
      ← getNode(pkp, skaddr)
5:   EncryptName, namesk ← getEncName(name, pkp, skaddr)
6:   HashKey ← {0, 1}256
7:   σh ← sign(sk, HashKey)
8:   NodeHashKey ← asymEncrypt(pk, HashKey || σh)
9:   ID ← queryCreateFolder(ParentID, NodeKey, NodeHashKey,
10:  email, Hash, Nodepp, NodeppSig, EncryptedName)
11:  return ID
createDocument(ParentID, pkp, HashKeyp, email, skaddr, ts)
Given: the parent identifier ParentID, the parent public key pkp,
the parent HMAC key HashKeyp, the creator's email address email,
the creator's address private key skaddr, and the current timestamp ts
Return: the document identifier IDDoc
1:  name ← GenerateName(ts)
2:  Hash ← HMAC-SHA256(HashKeyp, name)
3:  sk, pk, NodeKey, Nodepp, NodeppSig, Nodeppssk
      ← getNode(pkp, skaddr)
4:  EncryptName, namesk ← getEncName(name, pkp, skaddr)
5:  ek ← generateSessionKey()
6:  ContentKeyPacketSig ← sign(sk, ek)
7:  ContentKeyPacket, ppssk ← asymEncrypt(pk, ek)
8:  ManifestSig ← sign(skaddr, []) // sign an empty array
9:  IDDoc ← queryCreateDoc(EncryptName, Hash, ParentID,
10:  Nodepp, NodeppSig, email, NodeKey, ContentKeyPacket,
11:  ContentKeyPacketSig, ManifestSig, docType)
12:  return IDDoc

```

Figure 16: The procedure of creating a folder/document. The blue and green text highlights the differences.

document/folder name and the current node's passphrase. When inviting a collaborator, the owner shares the session key used to encrypt the share node's passphrase with the collaborator. With the shared session key, the collaborator can decrypt the share node to obtain the node session key and further decrypt the node. A document/folder has only one share node, which is created when first inviting a user or generating a share link. If all the collaborators are removed later, then the share node is deleted. If the user is invited again later, a new share node will be created.

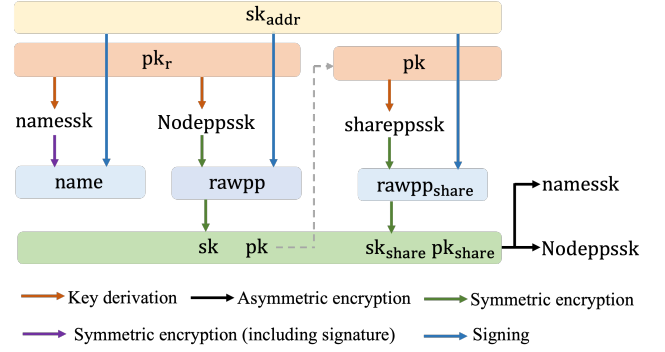


Figure 17: The key hierarchy of sharing.

```

createShare(IDaddr, ID, pk, Nodeppssk, namesk, skaddr)
Given: the owner's address identifier IDaddr,
the document/folder identifier ID,
the current document/folder node's public key pk,
the current node's passphrase session key Nodeppssk,
the current node's name session key namesk,
the creator's address private key skaddr
Return: the share identifier IDshare
1:  skshare, pkshare, ShareKey, Sharepp, ShareppSig, Shareppssk
      ← getNode(pk, skaddr)
2:  ppKeyPacket, ppssk1 ← asymEncrypt(pkshare, Nodeppssk)
3:  NameKeyPacket, ppssk2 ← asymEncrypt(pkshare, namesk)
4:  IDshare ← queryCreateFolder(IDaddr, IDDoc, ShareKey,
5:  Sharepp, ShareppSig, ppKeyPacket, NameKeyPacket)
6:  return IDshare

```

Figure 18: The procedure of creating a share node.

Share node generation. Before generating a share node for a document/folder node with Nodepp, EncryptedName, the owner first retrieves its public key pk, passphrase session key Nodeppssk, and name session key namesk as the input of createShare. The procedure of creating a share node is shown in Fig. 19. createShare invokes getNode to generate a key pair (sk_{share}, pk_{share}) for the share node, along with the ciphertext of the private key ShareKey, the passphrase ciphertext Sharepp, the passphrase signature ShareppSig, and the passphrase session key Shareppssk. Subsequently, the current node's Nodeppssk and namesk are encrypted with the share node's public key pk_{share}, resulting in the ciphertexts ppKeyPacket, NameKeyPacket, respectively. Finally, {ID_{addr}, ID_{Doc}, ShareKey, Sharepp, ShareppSig, ppKeyPacket, NameKeyPacket} is sent to the server via the queryCreateShare interface to request creating a share node on the server, and the server returns the share's identifier ID_{share}.

Two sharing methods. Proton Docs/Sheets provide two sharing methods: sending an invitation via email to the collaborator or using a sharing link.

```

inviteInternalUser(emailinviter, skaddr, emailinvitee, pkinvitee,
permission, Shareppssk, EmailDetail, ExInviteID = null)
Given: the owner's email and address private key emailinviter, skaddr,
the invitee's email and address public key emailinvitee, pkinvitee,
the access permission permission,
the share node's passphrase session key Shareppssk,
the email details EmailDetail,
an external invitation identifier ExInviteID (optional, null in default)
Return: the invitation identifier InviteID
1:  KeyPacket, ppssk ← asymEncrypt(pkinvitee, Shareppssk)
2:  KeyPacketSig ← sign(skaddr, KeyPacket)
3:  InviteID ← queryInviteInternalUser(emailinvitee, emailinviter,
4:  permission, KeyPacket, KeyPacketSig, EmailDetail, ExInviteID)
5:  return InviteID
inviteExternalUser(IDaddr, skaddr, emailinvitee, permission,
Shareppssk, EmailDetail)
Given: the owner's address identifier and private key IDaddr, skaddr,
the invitee's email emailinvitee, the access permission permission,
the share node's passphrase session key Shareppssk,
the email details EmailDetail
Return: an external invitation identifier ExInviteID
1:  ExInviteSig ← sign(skaddr, emailinvitee || Shareppssk)
2:  ExInviteID ← queryInviteExternalUser(IDaddr, emailinvitee,
3:  permission, ExInviteSig, EmailDetail)
4:  return ExInviteID
acceptInvitation(skaddr, pkinviter, KeyPacket, KeyPacketSig)
Given: the invitee's address private key skaddr,
the inviter's address public key pkinviter,
the invitation information KeyPacket, KeyPacketSig
1:  if Vrfy(pkinviter, KeyPacket, KeyPacketSig)
2:    Shareppssk, ppssk ← asymDecrypt(skaddr, KeyPacket)
3:    sskSig ← sign(skaddr, Shareppssk)
4:    queryAccept(sskSig)
5:  else return
convertExInvite(emailinviter, skaddr, pkaddr, emailinvitee, pkinvitee,
permission, Shareppssk, ExInviteSig, EmailDetail, ExInviteID)
Given: the owner's email and address keypair emailinviter, skaddr,
the invitee's email and address public key emailinvitee, pkinvitee,
the access permission permission,
the share node's passphrase session key Shareppssk,
the external invitation ExInviteSig, EmailDetail, ExInviteID
Return: an invitation identifier InviteID
1:  if Vrfy(pkinviter, emailinvitee || Shareppssk, ExInviteSig)
2:    return inviteInternalUser(emailinviter, skaddr, emailinvitee,
3:    pkinvitee, permission, Shareppssk, EmailDetail, ExInviteID)
4:  else return

```

Figure 19: The procedure of inviting users.

```

CreateShareUrl(email, pkaddr, Shareppssk, permission,
Modulus, g)
Given: the inviter's email and address public key email, pkaddr,
the share's node passphrase session key Shareppssk,
the permission permission,
the modulus and generator of SRP Modulus, g
1:  pwd ← $ RandomString(12)
2:  Keysalt ← $ {0, 1}128
3:  h1 ← Blowfish(pwd, Keysalt)
4:  rawpp ← h1[30:]
5:  KeyPacket ← symEncrypt(rawpp, Shareppssk)
6:  encpwd ← asymEncrypt(pkaddr, pwd)
7:  Urtsalt ← $ RandomString(10)
8:  h2 ← Blowfish(pwd, Urtsalt, Modulus)
9:  verifier ← gh2 mod Modulus
10: queryCreateSharedLink(permission, email, KeyPacket,
11: Modulus, verifier, Keysalt, Urtsalt, encpwd)
12: url = https://drive/urls/{Urtsalt}#{pwd}
13: return url

```

Figure 20: The procedure of creating a share URL.

Invitation email. Proton Docs/Sheets enable users to invite an existing Proton user or an external user (who has not registered with Proton) to share a document/folder. A document/folder owner can send an invitation email to the targeted user via the Proton server. The owner can choose whether to write an email, which is sent to the collaborator along with the document name. Fig. 19 shows the inviteInternalUser and inviteExternalUser algorithms, which are respectively used to invite internal and external users.

inviteInternalUser takes the owner's email address email_{inviter} and the address private key sk_{addr}, the collaborator's email address email_{invitee} and the address public key pk_{invitee}, the permission, the share node's passphrase session key Shareppssk, and EmailDetail including the email content and document name as input. Shareppssk is encrypted under pk_{invitee}, yielding the ciphertext KeyPacket, and KeyPacket is signed with sk_{addr}, resulting in a signature KeyPacketSig. Then, {email_{invitee}, email_{inviter}, permission, KeyPacket, KeyPacketSig, EmailDetail} is sent to the server. The server returns an invitation identifier and sends an invitation email containing EmailDetail to email_{invitee}.

In inviteExternalUser, Shareppssk is signed under the inviter's address private key, and the signature ExInviteSig, the invitee's email, the inviter's address identifier, the permission, and EmailDetail are sent to the server. The server returns an identifier of the external invitation and sends an email to the invitee. If the invitee wants to accept the invitation, they first register with Proton using the invited email. The inviter converts the external invitation to an internal invitation using the registered invitee's address public key by running the convertExInvite algorithm (as shown in Fig. 19). The inviter then checks the validity of the external invitation's signature and runs the inviteInternalUser algorithm.

An invitee can obtain the invitation information from the server and accept it by invoking the `acceptInvitation` algorithm (as shown in Fig. 19), where if `KeyPacketSig` is valid, the invitee decrypts `KeyPacket` with their address private key sk_{addr} , signs `Shareppssk` with sk_{addr} , and sends the signature `sskSig` to the server via the `queryAccept` interface.

We notice that EmailDetail is sent to the server in the plaintext form. If a user chooses to write an email and send it along with the invitation, then the email content and the document/folder name would be leaked to the server.

Sharing link. A document/folder owner can generate a public sharing link, and anyone who gets the link can access the document/-folder. The `CreateShareUrl` procedure is shown in Fig. 20, which

is based on the Secure Remote Password (SRP) authentication protocol [21]. In `CreateShareUrl`, a 12-character password `pwd` and a 32-byte salt `Keysalt` are randomly generated, yielding a passphrase `rawpp` using the Blowfish algorithm; the share node's passphrase session key `Shareppssk` is encrypted with `rawpp`, and the password `pwd` is encrypted with pk_{addr} . Then, an SRP verifier is generated for the password, where a 10-character salt `Urlsalt` is randomly generated. Finally, the permission, the ciphertexts of `pwd` and `Shareppssk`, the salts, and the verifier are sent to the server. A sharing link consisting of the salt used for authentication and the password is created. When creating a share link, the permission is read-only by default, while the user can later change it to write access.