CryptPad: End-to-End Encrypted Collaboration Suite

CryptPad Team @ XWiki SAS*

Version: 1.1.0 (Changelog in Appendix A)

Abstract. Collaborative document editing has grown popularity over the last decades. This has been driven by the ease of use of online platforms compared to workflows requiring participants to mail versions back and forth and reconcile changes. However, common tools such as Google Docs or Microsoft's Office 365 come with an impact to their users privacy. The server hosting these services can access all stored documents and actively modify or passively read their content.

In this paper, we present the cryptographic design of CryptPad, a web-based, end-to-end encrypted, collaborative real-time editor for a variety of document types. We show how we use cryptography to protect against attacks in an honest-but-curious threat model. We present multiple dedicated schemes that use a mix of asymmetric and symmetric encryption, as well as signing, to allow fine-graded access control, private messaging, and team collaboration. Despite the common perception that cryptography is too complicated for the average user, CryptPad remains simple to use and has a large and growing user base.

1 Introduction

CryptPad is an end-to-end encrypted real-time collaboration suite that is used by hundreds of thousands of people per month. It is accessible through a web interface which ensures that all data is encrypted in the browser with no readable user data leaving the local device. Even the service administrators can therefore not see the content of documents or user data.

Since CryptPad's initial release in 2014, the feature set has grown from a simple editor to a full-blown set of multiple applications including forms, spreadsheets, presentation slides, kanban boards, and whiteboards. Nowadays, CryptPad also features additional collaboration utilities such as calendars, teams and simple chats.

CryptPad is an open source project with both client and server code available and licensed under the GNU Affero General Public License version 3.0 (AGPL).[1] This means that anyone with the ability to do so is free to use, host, and modify the software as long as any modifications are made available to their users under the same terms. CryptPad is developed by XWiki SAS, a company based in Paris, France that has been making open source software since 2004. The development has been supported since 2015 by French and European research funding bodies such as BPI France, NLNet Foundation, NGI Trust, and Mozilla Open Source Support.

Due to the limited information which is exposed to operators of CryptPad servers, we cannot know exactly which type of users rely on CryptPad and for what kind of activity it is used. As several blog posts suggest, among CryptPad's users are journalists [1], people working in the health sector [2], and activists [3], [4]. There is also a public instance

*contact@cryptpad.fr

[1] Source code: `https://github.com/xwiki-labs/cryptpad`

hosted by Germany's Pirate Party [5]. This instance was recently seized by police as a consequence of the publication of sensitive material [3]. Altogether, this shows that higher-risk users rely on the security established by CryptPad. The users especially do not want to trust the server as it might be corrupt or seized due to legal enforcement.

In previous work [6], [7] we described CryptPad's general architecture and user interface and compared it to other collaboration tools. In this paper, however, we focus on the cryptographic design that is used to secure user-generated information such as documents and messages. We show the desired security properties and how we establish them cryptographically. We update this paper to align it with CryptPad's newest improvements. This paper here reflects 5.1.0 (c.f. the changelog in Appendix A).

The remainder of this paper is structured as follows: Section 2 summarizes CryptPad's underlying threat model. We then explain the communication between the server and the client in Section 3. After introducing the notation in Section 4, we present in Section 5 the encryption of pads as the core functionality of CryptPad. We next show in Section 6 how the login mechanism makes the documents easily accessible across multiple devices, but keeps them secure. Section 7 explains the establishment of secure communication between different users. Finally, Section 8 shows how we enable communication and access control within a team.

2 Threat Model

We describe adversarial capabilities and goals in detail in the threat model published on our website. To summarize, we consider an honest-but-curious server that has additional active network capabilities. The server has therefore access to encrypted data, but cannot actively alter, delete or copy data.

We address most of the resulting threats directly in the platform's cryptographic design. However, to defend against Machine-in-the-middle (MITM), and partly against impersonation attacks (server authentication) we rely on the security guarantees given by the Transport Layer Security (TLS) protocol.

3 Client-Server Communication

In this section, we show how we use the NetFlux protocol [8] to enable communication between web clients and the server. We discuss the usage of TLS that helps to defend against MITM and impersonation.

Each CryptPad instance makes use of a central server: while not being able to read any content, the server still acts as an intermediary between different clients. This is not only to forward the messages, but also to store them and to guarantee a highly available persistence layer that would not be possible in a purely peer-to-peer solution.

Communication between the client and the server is mostly done using a Websocket connection based on the NetFlux protocol. Exceptions are static files (images, videos, PDF, etc.) that are stored in an encrypted format and are retrieved by users with XMLHttpRequests (XHRs).

With the NetFlux protocol, users can join *channels* and send messages to them. All users subscribed to a channel will then receive the messages.

Each document is represented on the server by a channel with a unique 32-character hexadecimal identifier called chanID. The chanID associated with a document is derived from its URL, so that users knowing the URL can subscribe to the channel.

While a channel provides a way to communicate messages or changes to a document, channels do not provide confidentiality or authenticity. We therefore use TLS to encrypt the messages between the server and the client. This allows to authenticate the server and therefore rules out MITM attacks. It further restricts any network adversary from accessing the encrypted content.

However, TLS on its own is not enough as the encryption is only between the server and the client, but not between two clients. TLS can moreover not prevent clients from subscribing to channels they should not have access to. To prevent against this, we make use of dedicated encryption schemes hand tailored to CryptPad. We present these schemes in the Sections 5 to 8.

4 Notation

In this section we introduce the used libraries and notation.

For the key derivation function (KDF) we use `scrypt` [9]. We define $\mathsf{KDF}(\mathsf{pwd}, \mathsf{salt})$ to be the `scrypt` algorithm using the password pwd and salt as an input.

All other cryptographic operations are done using the TweetNaCl.js library [10], [11]. We also use TweetNaCl.js to sample random bytes, however, TweetNaCl.js itself relies on the browsers' sources of secure randomness.

Symmetric encryption is authenticated and uses XSalsa20-Poly1305. We let $\mathsf{SymEnc}(\mathsf{K}, m)$ be the symmetric encryption of the message m under the secret key K.

Public-key encryption is authenticated and uses x25519-XSalsa20-Poly1305. We let $\mathsf{KGen}_E(\mathsf{seed})$ be the derivation of an asymmetric key pair $(\mathsf{PK}, \mathsf{SK})$ from seed. Here, PK denotes the public key and SK denotes the private key. We use $\mathsf{Nacl.box}(m, N, \mathsf{PK}_B, \mathsf{SK}_A)$ to explicitly refer to the asymmetric encryption of a message m under a nonce N, Bob's public key PK_B, and Alice' private key SK_A.

Likewise, we use Ed25519 for signatures and let $\mathsf{KGen}_S(\mathsf{seed})$ denote the derivation of an asymmetric signing key pair $(\mathsf{PK}, \mathsf{SK})$ from seed. The signing of a message m under a private key PK is written as $\mathsf{Sign}(m, \mathsf{PK})$.

Finally, we use SHA-512 for hashes and denote the hash of a string x as $\mathsf{H}(x)$. To mark a splitting of a string x in diagrams, we annotate the arrows with $i..j$ to denote all *bytes* from the i-th to the j-th (both included). We further let $x||y$ be the concatenation of two strings x and y. In diagrams, multiple arrows pointing to the same box (e.g., a hash) may implicitly denote string concatenation.

5 Documents

In this section, we present *documents* as one of the core concepts of CryptPad. Historically implemented as encrypted collaboratively editable text documents, the concept of documents has expanded. Nowadays, other types of data such as folders, polls and calendars are internally represented as documents on CryptPad.

We first show the basic idea of how we use encryption to control access rights. Next, we present the consensus algorithm that ensures that changes to a document are propagated to the users in nearly real-time. We then explain in Section 5.2 the key derivation in multiple different scenarios that enables end-to-end encryption, prevents user abuse attacks, and is easy to use. We further explain CryptPad's ownership model in Section 5.3 to achieve. Finally, we discuss in Section 5.4 how we build self-destructing documents depending on either the time or on the opening of the sharing link.

5.1 Consensus Protocol

The main difficulty for near real-time collaboration is in reconciling the fact that it is impossible for two events separated by some distance to interact instantaneously. We therefore introduce in this section a protocol that can handle simultaneous edits of documents.

CryptPad uses its own protocol based on Operational Transformation (OT) [12] and directed acyclic graphs. The basic idea is to generate *patches* from one state to the next and to collectively decide the order in which the patches need to be applied.

The user stores a local copy known as the *authoritative document* which is the last known state of the document that is agreed upon by all the users. The authoritative document can only be changed as a result of an incoming patch from the server.

The difference between modified document and the authoritative document is represented by a *patch* known as the *uncommitted work*. A patch further references the SHA-256 hash of the authoritative document. The patches therefore build a directed acyclic graph where we call the longest path to be the *chain*.

As the user adds and removes data, this uncommitted work grows. Periodically the user transmits the uncommitted work in the form of patches to the server. The server will then broadcast these patches to all users listening to the corresponding NetFlux channel. The user can afterwards update the authoritative document and reset the uncommitted work.

When receiving a patch from the server, the user first examines the validity and discards the patch if the cryptographic integrity and authenticity checks do not pass. If the patch references the current authoritative document, the user applies the patch to the authoritative document and transforms the uncommitted work by that patch.

Otherwise, the user stores it in case that other intermediate patches have not yet been received. It could be that a patch references a previous state of the document which is not the authoritative document. The user stores the patch in this case as it might be part of a fork of the chain which proves longer than the chain which the engine currently is aware of.

In the event that a fork of the chain becomes longer than the currently accepted chain, a "reorganization" will occur which will cause the authoritative document to be rolled back to a previous state and then rolled forward along the newly accepted chain. During reorganization, users will also revert their own committed work and re-add it to their uncommitted work. Conflicts are resolved with a dedicated scheme that depends on the type of changes, e.g., deletions have precedence over replacements. It might therefore be the case that some changes are lost during conflict resolution. However, due to the short period between two consecutive patches and the fast conversion of chains, this is not a problem in practice.

A special type of patch, known as a *checkpoint*, always removes and re-adds all content to the document. The server can detect checkpoint patches because they are specifically marked on the wire. In order to improve performance of new users joining the document and "syncing" the chain, the server sends only the second most recent checkpoint and all patches newer than that.

5.2 Encryption

In this section, we present the encryption scheme for documents. We first show how we use symmetric encryption of messages and then how we derive the keys for different types of documents: encrypted blobs (Section 5.2.1), editable documents (Section 5.2.2), and forms (Section 5.2.3).

For every document, we want to distinguish between at least two access rights: reading and

writing. To enforce these access rights cryptographically, we use symmetric encryption to restrict read access and asymmetric signatures to restrict write access.

More specifically: in order to write data m to a document, a user must have a symmetric encryption key K and a signing key SK that is part of an asymmetric key pair $(\mathsf{PK}, \mathsf{SK})$. The user first symmetrically encrypts m to a ciphertext c using K. This ciphertext effectively hides its underlying content from anyone not having access to K (including the server). Then, to prove the write access right, c is further asymmetrically signed using SK resulting in a signature sig. Finally, the signed ciphertext (c, sig) is sent over a NetFlux channel to the server which checks the signature. If this check succeeds, then the server stores the message and forwards it to all users listening to the same channel. Otherwise, the message is neither stored nor forwarded. When users receive a ciphertext, they can decrypt it using SK. A user who does not have SK, but only K and PK may read new incoming ciphertexts, but cannot draft new ones. The separation of encrypting and signing further allows outsourcing the validation of (c, sig) to the server as it can have PK, but not K.

To collaboratively work on a document, users must share the keys with their collaborators. Our key derivation scheme is specifically designed to make the sharing of the keys easy. We therefore first outline how keys can be shared and then show how we actually derive the keys to enable this simple sharing mechanism for different use cases, i.e., for encrypted blobs, editable documents, and forms.

There are two ways to share a document and its keys: via CryptPad's internal communication mechanisms (c.f Section 7) or via sharing a URL. For the latter, we use the fact that the URL part after # is never sent to the server [13]. Users can therefore safely put the information required to derive keys in a URL after #.[2]

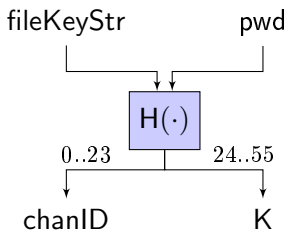

Figure 1: Key derivation for encrypted blobs

Since users may opt for an additional password required to have access to the document, we do not directly put the keys into the URL, but derive them from a seed concatenated to a (possibly empty) password. This feature is especially useful in the case that there is no confidential channel to securely share the link: If there are two distinct unconfidential channels (e.g., email and SMS), the users can share the URL over one channel and the password over the other channel. While not resulting in a truly secure sharing, the probability for an adversary to intercept both components is reduced.

5.2.1 Encrypted Blobs

Encrypted blobs such as uploaded PDFs, images or videos are encrypted once and stored on the server. There is no need for more fine-grained access control as editing the static document is by definition not possible. It is therefore enough to only derive a symmetric key K, but not a signing key pair. The key derivation is depicted in Fig. 1.

We first concatenate the (possibly empty) password pwd with the seed $\mathsf{fileKeyStr}$ and hash it. We then split the hash into 24 bytes for the chanID and 32 bytes for the K.

[2] An example of such a URL looks as follows: `https://cryptpad.fr/pad/#/2/pad/view/GcNjAWmK6YDB3E02IipRZ0fUe89j43Ryqeb4fjkjehE/`

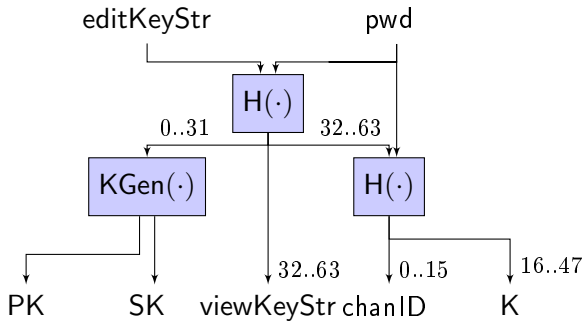

Figure 2: Key derivation for editable documents

In case that the fileKeyStr is empty, it is initialized to 18 random bytes and returned as an additional output. This will allow anyone in its possession and knowing pwd (if it exists) to derive the same chanID and K and to thus download and decrypt the file.

5.2.2 Editable Documents

Most of the editable documents are only modifiable by selected users. These users are able to not only decrypt the document, but also to sign patches and send them to the server. We therefore need to derive both, a symmetric encryption key K and a signing key pair $(\mathsf{PK}, \mathsf{SK})$.

To create a new document with all the above capabilities, we derive the keys as depicted in Fig. 2. The inputs are again a (possibly empty) password pwd and a seed editKeyStr. In case that the latter is empty, we initialize it with 18 random bytes and return it as an additional output. We then hash the concatenated pwd and editKeyStr and split the resulting hash into two parts. From the first part we generate the signing key pair $(\mathsf{PK}, \mathsf{SK})$. The second part of the hash forms the viewKeyStr and is fed together with pwd into another hash from which we derive the chanID and the symmetric key K.

Users may want to publish a document (e.g., a blog post) so that others can only read them, but not change them. To achieve this, users can publish the viewKeyStr since it allows – together with the knowledge of pwd– to derive the symmetric key K as well as the chanID. Moreover, viewKeyStr is independent of the input bits to $\mathsf{KGen}(\cdot)$ that produced $(\mathsf{PK}, \mathsf{SK})$. It is therefore not possible to deduce the signing key SK required to edit the document from viewKeyStr.

5.2.3 Forms

There are more complex use cases which require even more fine-grained access control and therefore also more encryption and signing keys to differentiate the access rights. One such example is a form having multiple roles: the *authors* should be able to write, view and answer the questions, as well as to view all responses to the form. The *participants* should be able to view the questions and to answer them. However, participants should not be able to read the responses. Furthermore, there are *auditors* with the capability to view all responses, but without the capability to answer the form themselves. The auditor role can be used to incorporate answers from a privileged set of form respondents in real time.[3]

This illustrates the need for two different sets of keys. First, we need a symmetric key K_1 that allows to encrypt/decrypt the questions, as well as a key pair $(\mathsf{PK}_1, \mathsf{SK}_1)$ to change the questions. Second, we need an asymmetric key pair $(\mathsf{PK}_E, \mathsf{SK}_E)$ to encrypt/decrypt the answers and a signing key pair $(\mathsf{PK}_2, \mathsf{SK}_2)$ to prove the answer capability. We distribute the keys as follows:

- We derive all keys from a seed editKeyStr and give this seed to the authors so that they can perform any action they want.

[3] You can, e.g., publish results of an ongoing vote in real time.

They will encrypt the questions with K_1 and sign them with $(\mathsf{PK}_1, \mathsf{SK}_1)$ to prove write access. Furthermore, they derive a public encryption key pair $(\mathsf{PK}_E, \mathsf{SK}_E)$ used to encrypt/decrypt form replies. Finally, they use the signing key pair $(\mathsf{PK}_2, \mathsf{SK}_2)$ to prove/verify the ability to reply.

- The participants get a seed viewKeyStr that allows them to derive K_1 and thus to read the questions. They further get $(\mathsf{PK}_1, \mathsf{SK}_1)$ to sign their answers and PK_E to encrypt their answers. However, they cannot decrypt the answers of others.

- Finally, the auditors get viewKeyStr to derive K_1 and thus to read the questions; $(\mathsf{PK}_E, \mathsf{SK}_E)$ to decrypt the replies; and PK_1 to verify their signature.

The key derivation is depicted in Fig. 3 and extends the derivation of editable documents. The asymmetric signing key pair $(\mathsf{PK}_1, \mathsf{SK}_1)$, respectively K, is derived in the same way as $(\mathsf{PK}, \mathsf{SK})$, respectively K, in editable documents; and viewKeyStr and chanID are also identical to their counterparts in editable documents.

However, we derive some more keys: We hash SK_1 to generate $(\mathsf{PK}_E, \mathsf{SK}_E)$ used for asymmetric encryption of replies. We further use the bytes 32 to 63 of the hash of viewKeyStr to derive signing key pair $(\mathsf{PK}_2, \mathsf{SK}_2)$.[4] This key derivation scheme ensures that the possession of viewKeyStr does not allow deducing SK_1 or SK_E. Also, it is not possible to deduce SK_1 from SK_E.

[4]There is a bit overlap between K and the input to $\mathsf{KGen}_S(\cdot)$. This overlapping poses no effective threat as there are still 16 independent bytes. However, we will fix this in a future version.

5.3 Ownership

CryptPad has a concept of document ownership to restrict some actions such as document deletion and password enabling to *owners*. Ownership is not limited to single persons, but can be held by a team, or it can be shared, i.e., an existing owner can add another. We implement ownership by relying on public keys, the server can therefore not associate usernames to documents.

When creating a document (or uploading an encrypted blob), users also submit their long term public signing key to mark themselves as owners. The server then associates the public key to the document. To perform an ownership-restricted action to a document, the owners send a request signed with their public key to the server. If the signature is correct and the public key is associated with the document match, the server performs the requested action to the document.

5.4 Self-destructing Documents

CryptPad includes a feature to create self-destructing documents that will automatically be deleted. This feature therefore helps to ensure that confidential data will not be leaked and that sensitive data is not accessible forever.

An exemplary use case is password sharing where the password owner shares it via a URL sent to a peer. Fearing that the peer's laptop might be accessed by unauthorized third-parties at a later point in time, the password owner wants to ensure that the document can only be opened once and will be destroyed automatically afterwards.

The self-destruction can be based on two mechanisms: either on an expiration time or on the event of opening a shared link. For the first mechanism, the expiration date has to be set during the document creation and cannot be changed afterwards. The expiration date

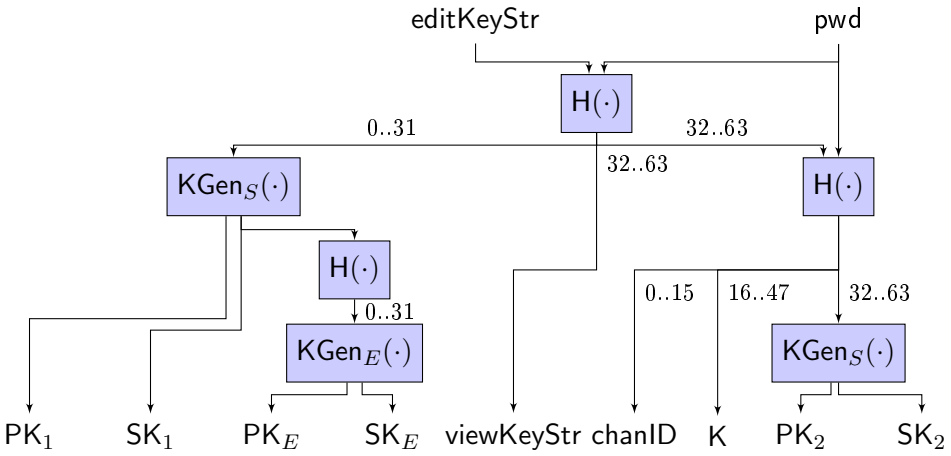

Figure 3: Key derivation for a form

is written to the document's metadata which can be read by the server. When fetching the document, the server first checks whether the expiration time has elapsed. If this is the case, the server deletes the document beforehand to prevent the users from fetching it. In the case where the expiration time elapsed while the document is opened by a user, it will be nevertheless deleted and the user will be disconnected. However, the user is still able to read the document until closing the corresponding browser tab.

The second mechanism, which we call "view-once-and-self-destruct", changes the way a shared link is created and opened. To create a view-once-and-self-destruct link, the document owner creates an ephemeral signing key pair and adds the private signing key to the list of owners of the document. This signing key is then together with a label "view-once" appended to the view-only link and sent to the receiver.

When the receiver opens the link, the receiver first fetches the content of the document. Then, immediately afterwards, the receiver sends the deletion command to the server. To prove the deletion capability, the receiver signs the command with the attached ephemeral signing key.[5]

6 CryptDrive

In this section, we introduce *CryptDrive* which provides users an interface to store and manage all their documents. We further allow the users to store their long term keys used to, e.g., encrypt messages (c.f. Section 7) or to prove ownership of a document (c.f. Section 6.2). However, all this information is encrypted and thus unreadable for the server. We first show in Section 6.1 the registration and login mechanism during which the server sees neither the username nor the password. Second, we show in Section 6.2 that the server is nevertheless able to do storage management and, e.g., impose storage limits.

6.1 Registration and Login

On a high level, we use the username and the password to generate login keys with which we

[5]We assume the receiver to be honest and to correctly issue the deletion command. This assumption is implicit in any such scheme, as a malicious user could always take screenshots and thus circumvent deletion.

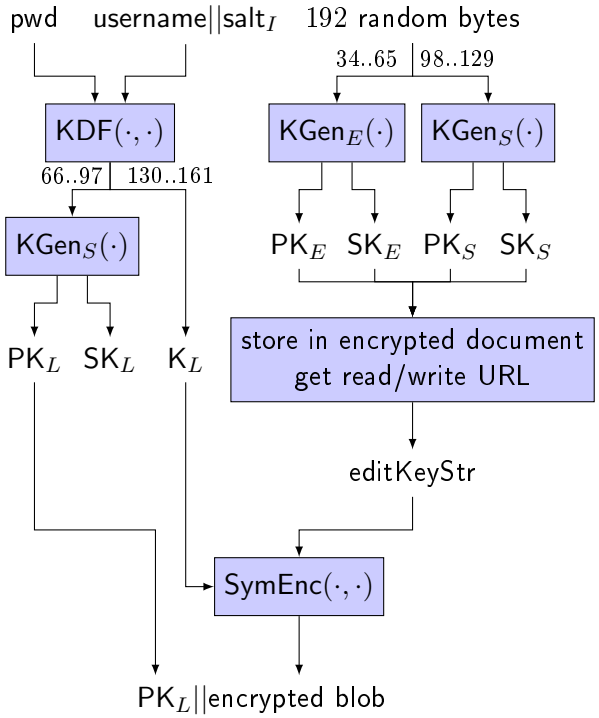

Figure 4: Derivation and storage of the login and long term keys.

access an encrypted file containing the user's long term keys as well as pointers to the user's documents and folders. The mechanism is depicted in Fig. 4 and is explained below.

During registration and login, we locally derive login keys using the KDF `scrypt` [9] which is designed to be memory expensive and therefore renders brute force attacks impractical as they are not feasible in a reasonable amount of time. More specifically, we derive from the password a symmetric encryption key K_L and an asymmetric signing key pair $(\mathsf{PK}_L, \mathsf{SK}_L)$. We let the salt for this key derivation be the concatenation of the username and the per-instance string salt_I. The usage of a per-instance salt prevents the attacker from precomputing rainbow tables for all instances and cracking passwords easily after a potential database compromise. Note that the selection of the used bytes (see Fig. 4) is not continuous due to legacy reasons.

A password change will result in a change of the login keys. Since we want some keys, such as the user's long term signing keys, to be invariant, we cannot rely on the login key to derive all further keys. Instead, we randomly generate long term encryption and signing keys $(\mathsf{PK}_E, \mathsf{SK}_E)$ and $(\mathsf{PK}_S, \mathsf{SK}_S)$ during registration to store them in an encrypted data structure. This data structure is the same that we use for encrypting documents (c.f. Section 5.2), hence we refer to it as an encrypted "document". We then read/write URL providing access to this document using the login key K_L and store them in a symmetrically encrypted blob on the server. We further add the verification key PK_L to the metadata of this blob to mark the ownership to the server.

To log in, the user derives the exact same keys $\mathsf{K}_L, (\mathsf{PK}_L, \mathsf{SK}_L)$ and submits then the public key to the server. Since the user has derived the symmetric encryption key K_L, the user can decrypt the blob and obtain the read-write URL of the encrypted document containing the long term keys.

This layer of indirection allows changing the password since we only need to re-encrypt the access to the encrypted document under new login keys K'_L and $(\mathsf{PK}'_L, \mathsf{SK}'_L)$. To change the password, the user first signs the public key PK_L with SK_L to prove the ownership. The user then sends this signature together with the re-encrypted blob under K'_L and the attached PK'_L to the server. Next, the server uses PK_L stored in the metadata to verify the correctness of the signature. If the verification succeeds, the server can safely replace the encrypted blob and store PK'_L as the new verification key.

To let the user access all their folders and documents, we store pointers to these in the same encrypted document that contains the long term keys. The access to the user's folders and documents will therefore also be safely

migrated during a password change.

Note that this login mechanism does not require the server to store the username, neither in plaintext nor in any hashed form. The server can therefore not check whether a given username has an account or not. Another consequence is that multiple users may register with the same username – as long as their passwords are different.

6.2 Storage Management

So far, we have shown how the data owned by or related to a user is encrypted. Nevertheless, we want the server to be able to manage the data storage.

First, the server should be able to impose storage limits so that users cannot allocate more storage than they are supposed to do. We therefore create a list of *pins* for each registered user where each list is identified by the user's long term public signing key. Every pin contains a document/blob identifier, a list of owners (represented with their public keys), and a creation date. The list of pins of a specific user therefore contains all the documents (co-)owned by this user. To compute the disk usage of a specific user, the server simply sums up the size of all documents and blobs contained in the pins. If the maximal storage quota for a user is reached, then the server hinders the user from pinning newly created documents and blobs.

Second, we also want the server to be able to identify unused documents created by guests. This allows the server to delete documents which have not been opened during certain time period and to thus free disk space. We therefore add to each (encrypted) file a metadata file not only an owner list (c.f. Section 5.3) but also the creation time. When the server periodically iterates over all documents, it can first check whether the document is owned by a CryptDrive user. If this is not the case, then the server can delete documents that have not been accessed during, e.g., the last 3 months.

7 Messaging

In this section we present CryptPad's own end-to-end encrypted messaging system that allows users to exchange arbitrary messages and metadata with other users. The messaging system is further used for, e.g., instance support, team messaging (c.f. Section 8) and forms. An important property of the used encryption scheme is anonymity: A user eavesdropping on another user's mailbox can not infer the sender of the message.

The basic block of the message encryption is to use public-key authenticated encryption with $\mathsf{Nacl.box}(\cdot)$ which internally derives a shared key between the receiver's and the sender's keys.

We build our encryption $\mathsf{ASymEnc}(\cdot)$ on this by encrypting a plaintext under the sender's public/private keys $(\mathsf{PK}_A, \mathsf{SK}_A)$ and the receiver's public key PK_B as follows:

$\underline{\mathsf{ASymEnc}(\mathsf{PK}_A, \mathsf{SK}_A, \mathsf{PK}_B, m)}$

$N \leftarrow\!\!\$\ \{0,1\}^{192}$

$c \leftarrow \mathsf{Nacl.box}(m, N, \mathsf{PK}_B, \mathsf{SK}_A)$

return $N||c||\mathsf{PK}_A$

The 24-bytes-sized nonce N is sampled uniformly at random and is prepended to the authenticated ciphertext c. The sender's public key PK_A is further appended to indicate the sender's identity to the receiver.

The receiver can then split the received message into N, c, and PK_A to decrypt it using $\mathsf{Nacl.box.open}(c, N, \mathsf{PK}_A, \mathsf{SK}_B)$. In case that the sender wants to decrypt this message, the server must use $(\mathsf{PK}_B, \mathsf{SK}_A)$ instead of $(\mathsf{PK}_A, \mathsf{SK}_B)$, since the sender does generally not have access to SK_B.

However, we do not directly encrypt messages using $\mathsf{ASymEnc}(\cdot)$ as this would leak the

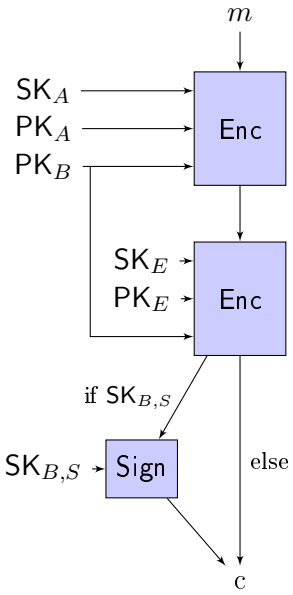

Figure 5: Sealing of a message m to return a ciphertext c.

sender of the message. Instead, we apply a second layer of encryption using *ephemeral* keys which are freshly generated for each message and thus are not linkable to the sender. Lastly, we may additionally sign the double encrypted messages using a signing key to prove write access.

An exemplary use case for this is the signing of a form answer: the users will receive it from the author to prove that they actually are allowed to send messages answering the form. For other cases the signing key may be obtained through the accounts' profile pages, or as a point of contact in documents.

This sophisticated sealing scheme is depicted in Fig. 5. It takes as an input a message m, the long term encryption key pair $(\mathsf{PK}_A, \mathsf{SK}_A)$, the ephemeral keys $(\mathsf{PK}_E, \mathsf{SK}_E)$, and – optionally – the signing key $\mathsf{SK}_{B,S}$ belonging to the receiver's signing/verification key pair $(\mathsf{PK}_{B,S}, \mathsf{SK}_{B,S})$. We first encrypt using the combination of the sender's and the receiver's keys. We then apply the second layer encryption using ephemeral keys $(\mathsf{PK}_E, \mathsf{SK}_E)$. Finally, we check whether a signing key $\mathsf{SK}_{A,S}$ was passed as input. If this is the case then we additionally sign the ciphertext before returning it. Otherwise, we just return the ciphertext without signing it.

Every user shares its verification key $\mathsf{PK}_{B,S}$ with the server such that the server can check the signatures of incoming messages. Therefore, the receiver does not have to check it. The receiver, however, needs to decrypt the outer layer using the private key SK_B and the supplied ephemeral key PK_E. Then the receiver can extract the sender's public key PK_A and use it together with the private key SK_B to decrypt the inner layer.

8 Teams

In this section, we present CryptPad's team encryption which is similar to the general messaging encryption, but suitable for the use case where we want to have fine-grained control over reading and writing access. We first show in Section 8.1 how we derive the team's keys and how we use them for message encryption and the control of the team's drive. Next, we present in Section 8.2 the different roles and permissions in team and how we enforce them.

8.1 Key derivation

We want that all users of a team can read messages, but not necessarily all of them to be able to write to the mailbox. We therefore generate not only an encryption key pair, but also a signing key pair which allows proving writing capabilities.

The key generation for a team is depicted in Fig. 6. We sample 18 Bytes uniformly at random for seed and split it into two halves. We hash the first half and build from this hash the chanID and an encryption/decryption key

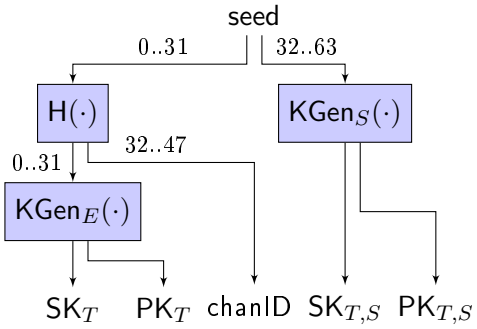

Figure 6: Derivation of team keys

pair $(\mathsf{PK}_T, \mathsf{SK}_T)$. The second half forms the input to $\mathsf{KGen}_S(\cdot)$ which generates a signing key pair $(\mathsf{PK}_{T,S}, \mathsf{SK}_{T,S})$.

To send a message to a team, the user encrypts it with the same mechanism as described in Section 7. However, the user must set the team's PK_T as the public key of the receiver and use the team's $\mathsf{SK}_{T,S}$ as the signing key.

The public signing key is further used to control a team's drive which works the same way as one of a single user. For example, we can use the signing key to pin a document, i.e., to indicate to the server that a specific document is owned. In the perspective of the server, the public key of this pin is, however, undistinguishable from the one of a single user. Based on an offline view of the database (i.e., after a seizure of law enforcement), the server can thus neither know which entities are teams, nor can the server match users to teams.

8.2 Roles and Permissions

As shown in Table 1, there are four different roles in a team: viewers, members, admins, and owners. The permissions are the following:

View: read-only access to folders and documents.

Edit: create, modify, and delete folders and documents.

Table 1: Different roles and their permissions

Role	View	Edit	Manage Members	Manage Team
Viewer	✓			
Member	✓	✓		
Admin	✓	✓	✓	
Owner	✓	✓	✓	✓

Manage users: invite and revoke users, change user roles up to admin.

Manage team: change team name and avatar, add or remove owners, change team subscription, delete team.

The number of permissions is strictly increasing, i.e., a role inherits all access rights from a weaker one, but has one additional permission.

All users keep a local copy of their teams including their rosters. To manage members and the team itself, admins (respectively owners) send the corresponding control message to all users of the team. They then check the authenticity of the message and whether the sender has sufficient rights to perform the desired action. If this is the case, then they update their local view of the team.

In order to, e.g., add a new document to the team's drive, the user sends the viewing keys to the viewers, and the editing keys to the teams' members, admins, and owners.

9 Conclusion

In this paper, we have presented the cryptographic design of CryptPad, a web-based, end-to-end encrypted, collaborative real-time editor for a variety of documents. We have shown realistic security threats for people with sensitive data and how we address them. We have explained how we use cryptography to enable various end-to-end encrypted applications,

such as documents, messaging and team communication.

Since its initial release 2015, CryptPad's user base has steadily grown. This shows, that the cryptographic design choices are also well-received by the platform's users. All in all, CryptPad has proved to be a long-lasting, sustainable open source project striving privacy-focused collaboration to the next level.

References

[1] P. Jha, *Russia's Invasion of Ukraine: How to Circumvent Censorship*, 2022. [Online]. Available: `https://vpnoverview.com/news/russias-invasion-of-ukraine-circumvent-censorship`.

[2] InterHop, *Our e-health software*, 2021. [Online]. Available: `https://interhop.org/en/projets/esante`.

[3] The Limited Times, *Data confiscated from Pirate Party servers*, 2022. [Online]. Available: `https://newsrnd.com/tech/2022-06-24-data-confiscated-from-pirate-party-servers.SJxeH5I79q.html`.

[4] N. Paris, *Tools for More Secure Activism*, 2019. [Online]. Available: `https://commonslibrary.org/tools-for-more-secure-activism`.

[5] Pirate Party of Germany, *Cryptpad.piratenpartei.de*. [Online]. Available: `https://cryptpad.piratenpartei.de`.

[6] A. MacSween, C. J. Delisle, P. Libbrecht, and Y. Flory, "Private Document Editing with Some Trust," in *Proceedings of the ACM Symposium on Document Engineering 2018*, 2018.

[7] A. MacSween and Y. Flory, "Behind the façade," in *HCI for Cybersecurity, Privacy and Trust*, 2019.

[8] C. J. DeLisle, *NetFlux Protocol 2*, 2016. [Online]. Available: `https://github.com/openpaas-ng/netflux-spec2/blob/master/specification.md`.

[9] C. Percival, "Stronger Key Derivation via Sequential Memory-Hard Functions," *BSDCan*, 2009.

[10] D. J. Bernstein, B. van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers, "TweetNaCl: A Crypto Library in 100 Tweets," in *Progress in Cryptology - LATINCRYPT*, 2014.

[11] D. Chestnykhm, D. Mandiri, and AndS-Dev, *Tweetnacl.js*, 2016-. [Online]. Available: `https://github.com/dchest/tweetnacl-js`.

[12] C. A. Ellis and S. J. Gibbs, "Concurrency Control in Groupware Systems," in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, 1989.

[13] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," RFC 3986, 2005. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc3986.txt`.

A Changelog

We keep this white paper up-to-date to reflect CryptPad's newest version.

1.1.0 Theo von Arx (2023-03-31) — Move threat model to dedicated webpage

1.0.0 Theo von Arx, Aaron MacSween (2022-11-29) — Initial white paper based on CryptPad 5.1.0