Cloud Data Sharing and Device-Loss Recovery with Hardware-Bound Keys

Felix Hörandner(✉) and Franco Nieddu

Graz University of Technology
{felix.hoerandner, franco.nieddu}@iaik.tugraz.at

Abstract. Cloud-based storage services, such as Dropbox, Google Drive, or NextCloud, are broadly used to share data with others or between the individual devices of one user due to their convenience. Various end-to-end encryption mechanisms can be applied to protect the confidentiality of sensitive data in a not fully trusted cloud environment. As all such encryption mechanisms require to store keys on the client's device, losing a device (and key) might lead to catastrophic consequences: Losing access to all outsourced data. Strategies to recover from key-loss have various trade-offs. For example, storing the key on a flash drive burdens the user to keep it secure and available, while encrypting the key with a password before uploading it to the cloud requires users to remember a complex password. These strategies also require that the key can be extracted from the device's hardware, which risks the confidentiality of the key and data once a curious person finds a lost device or a thief steals it.
In this paper, we propose and implement a cloud-based data sharing system that supports recovery after key-loss while binding the keys to the devices' hardware. By using multi-use proxy re-encryption, we build a network of re-encryption keys that enables users to use any of their devices to access data or share it with other users. In case of device-loss, we amend this network of re-encryption keys – potentially with the help of one or more user-selected recovery users – to restore data access to the user's new device. Our implementation highlights the system's feasibility and underlines its practical performance.

Keywords: Cloud Data Sharing · Key-Loss Recovery · Hardware-Protected Keys.

1 Introduction

Cloud storage services have seen broad adoption due to their convenience: These services enable users to store their data in the cloud, to access these data from any of their multiple devices (e.g., laptop, phone, tablet, etc.), and to share them with others. To also support sensitive data (e.g., medical records or company secrets), such services employ cryptographic mechanisms to ensure end-to-end confidentiality, for example, in the form of hybrid encryption or more elaborate lockbox-constructions [9,18,10]. With such cryptography, sharing access to data boils down to managing and distributing keys.

Employing cryptography to achieve secure data sharing requires to store the involved key material on the users' devices. If these devices (and their keys) are lost, broken or stolen, users face dire consequences, i.e., losing access to their important data. Therefore, applications that rely on client-side keys need to employ a strategy to recover from device- or key-loss. In data sharing scenarios where users own multiple devices, the loss of one device can be compensated as the user's other devices still have access. However, not all users own multiple devices, and, thus, a single-device recovery option has to be offered as well.

Approaches for recovery from key-loss have different trade-offs: Key material can be stored on a flash drive or printed as a QR code, which requires a secure location that stays available as well as confidential. Password-based encryption [17,21] or biometric encryption [8,16] can protect the key before storing it on a cloud service. Such approaches rely on limited entropy, which could be brute-forced by powerful attackers given sufficient time. Alternatively, secret sharing [23] enables to split the key into parts, distribute these parts across various semi-trusted entities, and reconstruct the key from a subset. This splitting requires a trust decision but enables a user-defined trade-off between confidentiality and availability.

However, all of these approaches for key distribution and recovery require that the key can be extracted, which leaves users vulnerable if one device is lost or stolen. Instead, we aim for a system which enables to bind the per-device keys to the devices' hardware with technologies such as Intel's SGX [15] or ARM's TrustZone [2] so that they cannot be extracted by attackers.

Our Proposed System. In this work, we build a cloud-based data sharing system, which supports multiple devices per user with keys bound to the hardware but also offers recovery from device- and key-loss for users with only one device. Highlighted features of our system are: 1) recovery with a threshold of users that enables a better trade-off between availability to confidentiality for single-device users, 2) improved key security through hardware protection, and 3) consequently immediate full access after recovery without the need for manual re-keying.

As a *basis*, we leverage proxy re-encryption (PRE) [5,3] and in particular its multi-use property. When using PRE instead of traditional public key encryption, the user generates re-encryption keys, which enable a proxy (i.e., the cloud storage service) to transform ciphertexts encrypted for the data owner into ciphertexts encrypted for an intended receiver. Multi-use PRE (MU-PRE) [6] allows to further re-encrypt ciphertexts that were already re-encrypted.

To *support multiple devices per user*, we propose to build and maintain a set of re-encryption keys, which enables to re-encrypt a user's ciphertexts for any of her (authorized) devices. With the multi-use property, any connected device of the user may generate re-encryption keys to other users' devices, which effectively shares access to her ciphertexts, as these ciphertexts can be successively re-encrypted. Building and maintaining such a network of re-encryption keys enables full access and sharing capabilities on each device.

Recovery from key-loss not only needs to be supported for the trivial case where the user owns a second device with access to all data, but also for users who only own a single device: Single-device users select a sufficiently trusted recovery

user who is willing to assist in case recovery becomes necessary. This recovery user is responsible for identifying and authenticating users who request recovery to ensure that only new devices of legit users get access. Users need to convince their recovery users to generate a re-encryption key for their new devices. With that re-encryption key, the cloud storage service re-encrypts the user's data for the recovery user and further for the user's new device, making it accessible again. As the cloud storage service does not expose intermediate ciphertexts, recovery users do not get access to the data. Recovery requires little effort on the user's side to make all data immediately available (i.e., generate re-encryption keys).

Our approach also enables us to *bind the keys to the devices' hardware*. The data sharing and recovery processes are designed so that private key material does not have to be extracted from the devices' hardware. Therefore, we are able to bind the keys to the device's hardware and only unlock them for authorized users. As attackers (e.g., thieves) are not able to extract and steal the keys, time-consuming re-keying and re-distribution of keys is not necessary.

Implementation and Discussion. Additionally, we evaluate the feasibility and performance of our system through a proof-of-concept implementation. In particular, we give details on the implementation of the used MU-PRE scheme and estimate costs of deployment on Amazon Web Services (AWS). Finally, we discuss the recovery effort from the user's perspective, elaborate on the performance results in the data sharing setting, and argue the benefits of hardware-bound keys as such approaches require no re-keying on the users' devices.

2 Background and Related Work

Cloud Data Storage. Cloud storage services employ cryptography to achieve end-to-end confidentiality for data that are handled in a not fully trusted cloud environment. With hybrid encryption, the data is symmetrically encrypted, while the used symmetric key is encrypted with a public key encryption scheme for one or more intended receivers. Fu's Cepheus [9] expands on hybrid encryption and introduces the concept of a lockbox: A lockbox contains the key to access a user's data. As the lockbox is protected (e.g., public key encrypted), it can be stored in public, and only authorized people are able to open it. Plutus [18] and SiRiUS [10] further expand on this idea.

Encryption for Data Sharing. With hybrid encryption or lockbox-constructions, *public key encryption* can be used to share access to a file by distributing access to the symmetric content encryption key.

Proxy Re-Encryption (PRE) [5,3] can be used instead of public key encryption to share access to the symmetric keys or lockboxes. PRE extends asymmetric encryption by enabling a semi-trusted proxy to transform ciphertext encrypted for one user into ciphertext encrypted for another user, without learning the underlying plaintext in an intermediate step. The user controls the sharing process by generating re-encryption keys towards other users, which the proxy requires in the transformation process. Ateniese et al. [3] applied PRE to data sharing: Users encrypt their files for themselves and generate re-encryption keys

for intended receivers. Given those re-encryption keys, the cloud storage service (i.e., proxy) transforms the user's data for authorized receivers, who are then able to decrypt the ciphertext with their own key material. Previous work (e.g., [12]) has focused on single-use PRE schemes, where ciphertexts can be re-encrypted once, but already re-encrypted ciphertext cannot be further transformed.

Attribute-Based Encryption (ABE) [11] represents another alternative to share access to data. With ABE, data is encrypted for attributes rather than for specific public keys (i.e., identities). Everyone with key material matching the ciphertext's attributes is able to decrypt. Such keys are issued by a trusted third party, which entails high trust requirements, as this party can decrypt any ciphertext.

Recovery from Key-Loss. The above-described encryption mechanisms either rely on per-user master keys or individual keys per device. We summarize approaches to recovery in case the device holding such keys breaks or is lost.

Password-based encryption can be used to wrap the user's key before uploading it to a cloud storage. This wrapped key is retrieved either by the user's other devices or on a new device once recovery becomes necessary. Such encryption relies on keys generated from the user's password through a key derivation function (e.g., PKDF2 [17], scrypt [21]). Increasing the derivation costs propagates directly to attackers. Consequently, this approach offers little protection against cloud attackers with plenty of resources and sufficient time to brute-force the password.

When employing *biometric cryptosystems* (BCSs), e.g., based on fuzzy extractors [8] or biohashing [16], the users' biometric templates (e.g., fingerprints) are used to protect the keys. After storing these protected keys at a cloud service, they can be downloaded and decrypted on any device where the user inputs her biometric template. Unfortunately, BCSs require additional data to generate stable, high-entropy keys [19], which again need to be kept available and confidential.

With *secret sharing* [23], users split their keys into multiple shares, hand these shares to different trusted parties, and are later able to reconstruct the key from a threshold of shares. For example, Huang et al. [14] propose to apply secret sharing on keys in a cloud data sharing setting. However, they do not suggest any authentication mechanisms to ensure that only authorized parties obtain shares.

Password-Protected Secret Sharing [4,7] introducing password-based authentication for secret sharing where entities holding the shares are able to verify the user's password but do not learn it. For example, Hörandner et al. [13] have applied this concept to split the keys over a hierarchy of trusted services. However, secret sharing, as well as the previous key recovery mechanisms, require that the users' keys can be extracted from the devices' hardware.

Hardware-Protection for Keys. We are interested in technologies that bind the keys to the devices' hardware and only unlock them for authorized users.

On phones with ARM's TrustZone [2] technology, the CPU switches between a less trusted and a more trusted state (so-called world), while preventing unintended information leakage between them. The more trusted world has access to secrets and is typically used to execute security code, while the less trusted world runs the operating system and applications. Such technologies can be used to establish a secure boot chain, which ensures that a valid version of the operat-

ing system is loaded on the device. Such a valid operating system only unlocks the protected key material after the user has been authenticated. Devices with hardware support for key protection are widely deployed: All devices shipped with Android Nougat or newer are required to have such hardware protections. That are >38%[1] of 2.5 Billion devices[2] as of Juli 2019, when conservatively only counting devices with Oreo and Pie, as devices rarely receive more than one major-version upgrade. iPhone 5S and later also support hardware-bound keys.

On PCs and servers, Intel's SGX [15] introduces new instruction codes that enable to deploy code in a private memory region, a so-called enclave. The memory contents cannot be read by any other process even if the operating system is malicious. SGX employs on-the-fly encryption and integrity verification in the CPU. This technology has been introduced with Intel's Skylake CPUs in 2015.

Building Block. *Multi-Use Proxy Re-Encryption* (MU-PRE) [6] not only allows to re-encrypt once but multiple times in succession, i.e., to re-encrypt ciphertexts that have already been re-encrypted. We focus on unidirectional, non-interactive MU-PRE schemes as a fundamental building block of our concept.

Definition 1 (MU-PRE). *A multi-use proxy re-encryption (MU-PRE) scheme with message space $\mathcal{M}$ consists of the following PPT algorithms:*

$\mathsf{KeyGen}(1^\kappa) \to (\mathsf{sk}, \mathsf{pk})$**:** *On input of a security parameter κ, the algorithm outputs a secret and public key $(\mathsf{sk}, \mathsf{pk})$.*

$\mathsf{Enc}(\mathsf{pk}, M) \to C^1$**:** *On input of a public key pk and a message $M \in \mathcal{M}$, the algorithm outputs a level-1 ciphertext C^1.*

$\mathsf{Dec}(\mathsf{sk}, C^l) \to M$**:** *On input of a secret key sk and level-l ciphertext C^l, the algorithm outputs $M \in \mathcal{M}$ or $\{\perp\}$.*

$\mathsf{RKGen}(\mathsf{sk}_A, \mathsf{pk}_B) \to \mathsf{rk}_{A\to B}$**:** *On input of a secret key sk_A of user A and a public key pk_B of user B, the algorithm outputs a re-encryption key $\mathsf{rk}_{A\to B}$.*

$\mathsf{ReEnc}(\mathsf{rk}_{A\to B}, C_A^l) \to C_B^{l+1}$**:** *Given a re-encryption key $\mathsf{rk}_{A\to B}$ and a level-l ciphertext C_A^l for A, the algorithm returns a l+1-level ciphertext C_B^{l+1} for B.*

3 System Model

This section introduces the actors of our system, their main interactions, and trust assumptions. Figure 1 illustrates our data sharing setting.

Actors and Data Flow. *Multiple users* want to store and share their data securely. Every user has *one primary device* and possibly *multiple secondary devices*. Each of those devices has its own key pair, where the private key is stored in a secure environment (e.g., hardware-protected). The users instruct their devices to encrypt the data before uploading it to the *cloud storage service*. Of course, they may download that data again and decrypt it with appropriate key material. To also share data with other users, the data owner generates a re-encryption key towards the data receiver and hands that key to the cloud storage service. Then, the cloud storage service can transform ciphertext of the data

[1] `https://developer.android.com/about/dashboards/`

[2] announced at Google I/O 2019

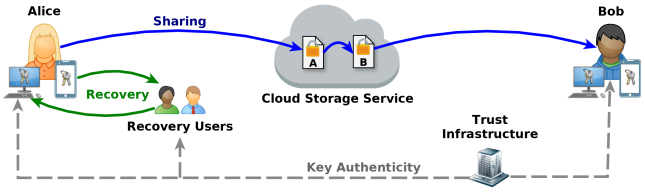

Fig. 1: System Model

owner into data encrypted for the intended receiver on demand. Our system is based on multi-use proxy re-encryption to set up a network of proxy re-encryption keys between the devices owned by a user as well as devices of other users.

Such a network of re-encryption keys not only enables our system to support convenient data sharing between different users and different devices of one user, but also offers user-friendly strategies to recover from device- and key-loss. Strategies to recover from key-loss might also require one (or more) trusted *recovery users*, which support the recovering user and thus accept further responsibility.

Additionally, a *trust infrastructure* can be used to ensure the authenticity of key material, which simplifies establishing the identity of data receivers.

Trust Assumptions. The user trusts the *cloud storage service* to operate honestly, while it might be curious to learn about the data it handles (e.g., curious insiders or cloud platform operators). To protect the data's confidentiality, the system employs end-to-end encryption (in our case, MU-PRE).

The user *trusts her devices' hardware* to protect her keys and only unlock them after successful authentication. Such hardware-protection for keys thwarts two attacks: Neither malicious applications nor unauthorized people with physical access to the device (e.g., thieves) are able to steal the user's keys and data. The user also trusts the sharing system's client application running on her devices.

Users with access permissions must not collude with the cloud storage service. Otherwise, the service re-encrypts the user's data and the corrupted receivers can decrypt it. Data owners select receivers with this trust requirement in mind.

Likewise, *recovery users* must not collude with the cloud storage service. While they might be curious, recovery users alone are not able to read another user's plain data, as honest cloud storage services do not expose the data. Recovery users are required to be willing and able to cooperate when recovery becomes necessary. As the development of users is hard to predict when selecting these recovery users, we also enable to distribute the risk over a threshold of users.

4 Concept Employing Multi-Use Proxy Re-Encryption

In this section, we introduce our concept for cloud-based data sharing that employs multi-use proxy re-encryption. First, we elaborate on how our concept builds a network of re-encryption keys between the devices of a user and – to enable data sharing – the devices of other users. Next, we give details on how to amend the network of re-encryption keys to recover from key-loss if devices are not available anymore, e.g., when they break. Also, this section considers the authenticity of keys, outlines operations per process, and suggests performance improvements.

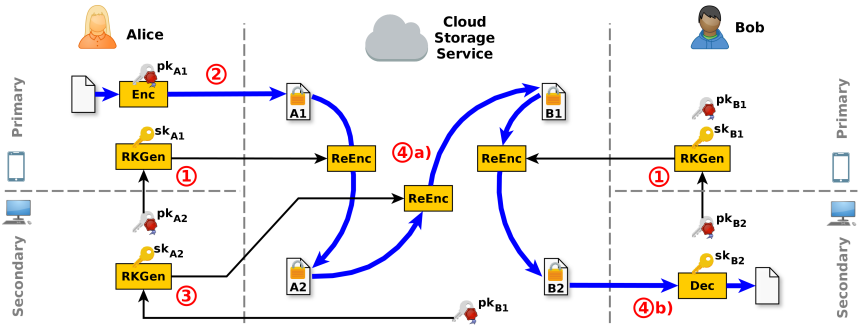

Fig. 2: Concept for Data Sharing

4.1 Setup and Multi-Device Data Sharing

Figure 2 illustrates the processes to store data at the cloud storage service, to share the data with other devices and users, and, finally, to download and decrypt the data. We describe the processes below and summarize them in Protocol 1.

⓪ **Setup.** When setting up an account, the user generates a key pair on her primary device and registers the public key at the cloud storage service. Additional steps to ensure the authenticity of her key are discussed in Section 4.3.

① **Register Secondary Device.** To add a new secondary device, in addition to performing the above-described setup, the user approves this device by generating a new re-encryption key from her primary device $\mathsf{A1}$ to this secondary device $\mathsf{A2}$: The secondary device sends its public key to the primary device, e.g., through a push notification or by showing a QR code at one device and scanning it at the other. After the user reviewed and accepted the request on her primary device, the primary device generates a re-encryption key $\mathsf{rk}_{\mathsf{A1}\to\mathsf{A2}} \leftarrow \mathsf{RKGen}(\mathsf{sk}_{\mathsf{A1}}, \mathsf{pk}_{\mathsf{A2}})$ and registers this key at the cloud storage service.

② **Upload Data (and Download for Data Owner).** All devices of a user, as well as other users, always encrypt the data for the primary device of a user, i.e. for the primary's public key.

Besides the primary device, the user's secondary devices are also able to access the encrypted data: Upon request, the cloud storage service uses the re-encryption key $\mathsf{rk}_{\mathsf{A1}\to\mathsf{A2}}$, which was generated during registration, to transform the encrypted data C_{A1} for the secondary device, resulting in C_{A2}. This ciphertext C_{A2} can then be decrypted at the secondary device with its private key $\mathsf{sk}_{\mathsf{A2}}$.

③ **Grant Access.** Data sharing with other users can be initiated by the data owner A, or requested by a receiver B, e.g., through a push notification. To grant access, the public key of the receiver's primary device $\mathsf{pk}_{\mathsf{B1}}$ is required. This key is registered at the cloud storage service or can be sent with the receiver's request. After verifying the key's authenticity, A's device generates a re-encryption key for B's primary device. Let us assume the more complex case, where A is using her secondary device, giving $\mathsf{rk}_{\mathsf{A2}\to\mathsf{B1}} \leftarrow \mathsf{RKGen}(\mathsf{sk}_{\mathsf{A2}}, \mathsf{pk}_{\mathsf{B1}})$. Next, A's device installs this re-encryption key at the cloud storage service along with an access control policy.

⓪ **Setup:** **on primary device** A1
1. generate key pair: $(\mathsf{sk}_{\mathsf{A1}}, \mathsf{pk}_{\mathsf{A1}}) \leftarrow \mathsf{KeyGen}(1^\kappa)$
2. certify authenticity of public key $\mathsf{pk}_{\mathsf{A1}}$
3. install public key $\mathsf{pk}_{\mathsf{A1}}$ at cloud storage service

① **Register Secondary Device:**
on secondary device A2
1. generate key pair: $(\mathsf{sk}_{\mathsf{A2}}, \mathsf{pk}_{\mathsf{A2}}) \leftarrow \mathsf{KeyGen}(1^\kappa)$
2. certify authenticity of public key $\mathsf{pk}_{\mathsf{A2}}$
3. install public key $\mathsf{pk}_{\mathsf{A2}}$ at cloud storage service
4. send public key $\mathsf{pk}_{\mathsf{A2}}$ to primary device

on primary device A1
5. verify authenticity of public key $\mathsf{pk}_{\mathsf{A2}}$
6. if user accepts, generate re-encryption key towards secondary device: $\mathsf{rk}_{\mathsf{A1}\to\mathsf{A2}} \leftarrow \mathsf{RKGen}(\mathsf{sk}_{\mathsf{A1}}, \mathsf{pk}_{\mathsf{A2}})$
7. install re-encryption key at cloud storage service

② **Upload Data:** **on any device**
1. encrypt data for primary device A1: $C^1_{\mathsf{A1}} \leftarrow \mathsf{Enc}(\mathsf{pk}_{\mathsf{A1}}, M)$
2. upload ciphertext C to cloud storage service

③ **Grant Access:**
on any device B$*$ **of requester (User B)**
1. send request for access to data owner with $\mathsf{pk}_{\mathsf{B1}}$

on any device A$*$ **of data owner (User A)**
2. verify authenticity of the public key of B's primary device $\mathsf{pk}_{\mathsf{B1}}$
3. let data owner review and accept request
4. generate re-encryption key from current device to B's primary device: $\mathsf{rk}_{\mathsf{A}*\to\mathsf{B1}} \leftarrow \mathsf{RKGen}(\mathsf{sk}_{\mathsf{A}*}, \mathsf{pk}_{\mathsf{B1}})$
5. install re-encryption key at cloud storage service

④ **Download Data:**
on any device B$*$ **of requester**
1. request download for ciphertext C at cloud storage service

on cloud storage service
2. find chain of re-encryption keys from data owner's primary device to requesting device: $(\mathsf{rk}_{0\to1}, ..., \mathsf{rk}_{n-1\to n})$
3. re-encrypt ciphertext along this chain (usually 0-3 re-encryptions): $C^{i+1} \leftarrow \mathsf{ReEnc}(\mathsf{rk}_{i\to i+1}, C^i)$

on device B$*$ **of requester**
4. decrypt ciphertext with device's private key $\mathsf{sk}_{\mathsf{B}*}$: $M \leftarrow \mathsf{Dec}(\mathsf{sk}_{\mathsf{B}*}, C)$

Protocol 1: Setup and Data Sharing

④ **Download Shared Data.** On request from B, the cloud storage service re-encrypts A's ciphertext C_{A1} up to three times in the default case as shown in Figure 2: These re-encryption operations use 1) the re-encryption key towards A's secondary device ($C_{\mathsf{A2}} \leftarrow \mathsf{ReEnc}(\mathsf{rk}_{\mathsf{A1}\to\mathsf{A2}}, C_{\mathsf{A1}})$), 2) the cross-user re-encryption key towards B's primary device ($C_{\mathsf{B1}} \leftarrow \mathsf{ReEnc}(\mathsf{rk}_{\mathsf{A2}\to\mathsf{B1}}, C_{\mathsf{B1}})$), and 3) the re-encryption key towards B's secondary device ($C_{\mathsf{B2}} \leftarrow \mathsf{ReEnc}(\mathsf{rk}_{\mathsf{B1}\to\mathsf{B2}}, C_{\mathsf{B1}})$). Finally, the receiver B decrypts C_{B2} by $M \leftarrow \mathsf{Dec}(\mathsf{sk}_{\mathsf{B2}}, C_{\mathsf{B2}})$. Thus, users can initiate sharing from any of their devices, while receivers can access that data from any owned device. Recovery may increase the re-encryption chain's length.

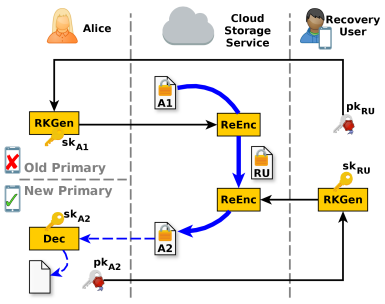

Fig. 3: Recovery with Rec. User

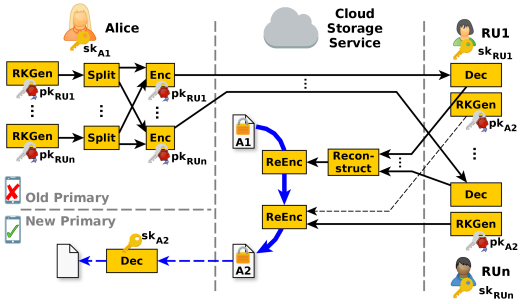

Fig. 4: Recovery with Threshold of Rec. Users

4.2 Recovery from Key-Loss

If a user's device is not available anymore (e.g., lost or broken), the user aims to regain access to her data and sharing capabilities. In our heterogeneous environment, some users own secondary devices, while others do not, and users transition between these groups over time when buying or losing devices. Thus, our recovery mechanisms also need to be seamlessly compatible with each other. Protocol 2 details three recovery mechanisms: 1) We show recovery with a secondary device as the simplest and most convenient solution. 2) Users without a secondary device may recover with the help of one recovery user. 3) We wish to highlight our third mechanism, where users recover with the support of a threshold of recovery users, and thereby improve their trade-off between availability and confidentiality.

(R1) Recovery with Secondary Device. With one or more registered secondary devices, the user simply selects one of those secondary devices as her new primary device. As new data will be encrypted for this new primary device, old secondary devices or receivers would not have access to this data through the existing network of re-encryption keys. To amend the network, the new primary device re-generates all outgoing re-encryption keys from the old device, i.e., keys that can transform ciphertexts from the primary device to other devices or users.

(R2) Recovery with Recovery User. As some users might not own a secondary device, they may rely on the support of a trusted recovery user (e.g., a family member) in the recovery process, as shown in Figure 3. During registration, the user selects another trusted user (i.e., recovery user), generates a re-encryption key from her primary device to that recovery user ($rk_{\mathsf{A1}\rightarrow\mathsf{RU}}$), and stores this key at the cloud storage service. Once the primary device's key material is not available anymore, the user convinces the recovery user to generate a new re-encryption key to the user's new primary device ($rk_{\mathsf{RU}\rightarrow\mathsf{A1}'}$). With this key, the cloud storage service can re-encrypt the user's data (encrypted for her old primary device) for the recovery user and then re-encrypt it again for her new primary device. Trusted (commercial) services might also offer to act as recovery user. Relying on a single recovery user poses an availability risk in case that user is no longer willing or able to help in recovery. This risk can be reduced by using multiple independent recovery users, which, however, increases the risk that one of these users colludes with the cloud storage service to access data.

R1 **Recover with Secondary Device:** **on secondary device** A2

1. select a secondary device A2 as new primary and inform cloud storage service
2. re-generate and replace all outgoing re-encryption keys $\mathsf{rk}_{\mathsf{A1}\to *}$ of old device: $\mathsf{rk}_{\mathsf{A2}\to *} \leftarrow \mathsf{RKGen}(\mathsf{sk}_{\mathsf{A2}}, \mathsf{pk}_{*})$

R2 **Recover with Recovery User:**

on old primary device A1**, during setup/registration**

1. select a recovery user RU
2. generate re-encryption key to recovery user: $\mathsf{rk}_{\mathsf{A1}\to\mathsf{RU}} \leftarrow \mathsf{RKGen}(\mathsf{sk}_{\mathsf{A1}}, \mathsf{pk}_{\mathsf{RU}})$
3. install this re-encryption key at the cloud storage service for recovery purposes

on new primary device A1'**, during recovery**

4. send recovery request to recovery user via cloud storage service with $\mathsf{pk}_{\mathsf{A1}'}$

on any device of recovery user RU

5. perform out-of-band authentication of the device requesting recovery
6. generate a re-encryption key to new device: $\mathsf{rk}_{\mathsf{RU}\to\mathsf{A1}'} \leftarrow \mathsf{RKGen}(\mathsf{sk}_{\mathsf{RU}}, \mathsf{pk}_{\mathsf{A1}'})$
7. install this key at the cloud storage service

on new primary device A1'

8. re-generate and replace all outgoing re-encryption keys of the old device

R3 **Recover with Threshold of Recovery Users:**

on old primary device A1**, during setup/registration**

1. select a list of recovery users $\{\mathsf{RU}_i\}$
2. for each recovery user RU_i
   (a) generate a re-encryption key: $\mathsf{rk}_{\mathsf{A1}\to\mathsf{RU}_i}$
   (b) split re-encryption key: $(s_{i,1}, ..., s_{i,n}) \leftarrow \mathsf{Split}(t, n, \mathsf{rk}_{\mathsf{A1}\to\mathsf{RU}_i})$
   (c) encrypt i^{th} shares of different keys: $\mathsf{RInfo}_i \leftarrow \mathsf{Enc}(\mathsf{pk}_{\mathsf{RU}_i}, \{s_{1,i}, ..., s_{n,i}\})$
   (d) upload RInfo_i to cloud storage service

on new primary device A1'**, during recovery**

3. send recovery request to recovery users via cloud storage service with $\mathsf{pk}_{\mathsf{A1}'}$

on any device of each recovery user RU

4. perform out-of-band authentication of the device requesting recovery
5. generate re-encryption key to new device: $\mathsf{rk}_{\mathsf{RU}\to\mathsf{A1}'} \leftarrow \mathsf{RKGen}(\mathsf{sk}_{\mathsf{RU}}, \mathsf{pk}_{\mathsf{A1}'})$
6. decrypt RInfo_i: $\{s_{1,i}, ..., s_{n,i}\} \leftarrow \mathsf{Dec}(\mathsf{sk}_{\mathsf{RU}}, \mathsf{RInfo}_i)$
7. send re-encryption key and decrypted RInfo_i to cloud storage service

on cloud storage service

8. reconstruct one $\mathsf{rk}_{\mathsf{A1}\to\mathsf{RU}_i} \leftarrow \mathsf{Reconstruct}(\{s_{i,1}, ..., s_{i,t}\})$
9. take the corresponding $\mathsf{rk}_{\mathsf{RU}_i\to\mathsf{A1}'}$ to close the re-encryption chain

on new primary device A1'

10. re-generate and replace all outgoing re-encryption keys of the old device

Protocol 2: Recovery from Key-Loss

R3 Recovery with Threshold of Recovery Users. We propose to rely on a threshold of recovery users by employing secret sharing mechanisms [23], as shown in Figure 4. This process allows users to select a more favorable trade-off between availability and confidentiality risks. The user initially selects a list of n recovery users RU_i and generates a re-encryption key towards each of them. Instead of handing these keys directly to the cloud storage service, the user splits each of these keys into n shares, takes the i^{th} share of different keys, and encrypts these i^{th} shares for the corresponding i^{th} recovery user. To complete the setup, the user uploads the resulting n ciphertexts to the cloud storage service.

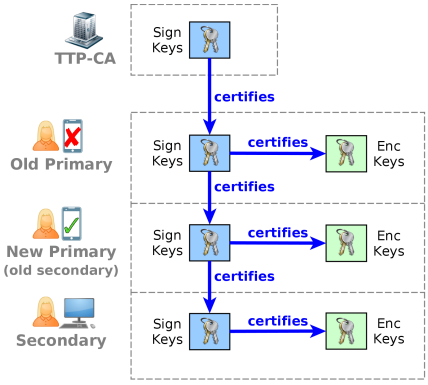

Fig. 5: Certification Chain

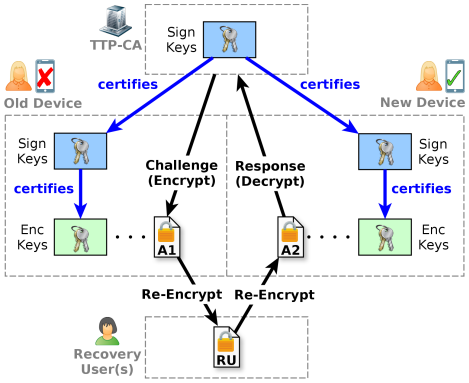

Fig. 6: Certification with Recovery Users

Once recovery becomes necessary, the user convinces a threshold of recovery users to download their ciphertexts, decrypt them, and return the decrypted shares to the cloud storage service. With those shares, the cloud storage service can reconstruct at least one re-encryption key towards one of the recovery users ($\mathsf{rk}_{\mathsf{A1}\to\mathsf{RU}_i}$). When also given a re-encryption from that recovery user to the user's new primary device ($\mathsf{rk}_{\mathsf{RU}_i\to\mathsf{A2}}$), the cloud storage service is again able to transform the user's data for her new primary device.

4.3 Key Authenticity

When users operate on keys from other devices or users (e.g., registering a device, encrypting, or sharing data), they rely on the authenticity of these keys. A key's authenticity can be established manually by comparing the received key's fingerprint with the key's owner, e.g., via a call, or shown on the owner's screen. The process to establish can be simplified by relying on trusted infrastructure. In this section, we give an example solution based on public key infrastructure (PKI), but approaches employing decentralized PKI mechanisms can also be conceived.

General Use. As shown in Figure 5, we build a certificate hierarchy, where the primary device and secondary devices act as intermediate certificate authorities (CAs). We introduce an additional signing key pair per device, which certifies the device's encryption pk. During account setup, the primary device's signing key is certified by a trusted third party CA (TTP-CA) for the user's human-readable identifier. This CA must ensure that it issues a certificate for a given identifier only once and that it does not accept similar identifiers, e.g., prevents homoglyph attacks. When registering a secondary device, the secondary device sends its signing pk to the primary device, which – acting as intermediate CA – certifies the secondary's signing pk given the user's consent. If the signing pk was transmitted by showing and scanning a QR code, the user's consent is given implicitly in this process. Other less direct communication mechanisms (e.g., push notifications) might require additional verification, for example showing the pk's fingerprint on both devices to be compared by the user. Before operating

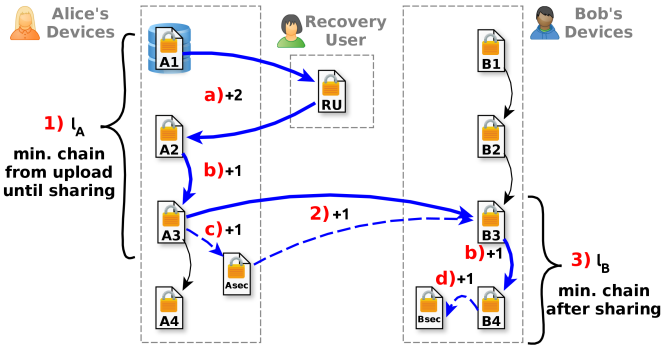

Fig. 7: Re-Encryption Chain for Data Sharing after Recovery

on foreign keys (e.g., as a reaction to a sharing request), the receiving device traverses the certificate chain to find and display the identifier of the owner. This identifier serves as a basis for users to accept or deny the operation.

When Recovering with Secondary Devices. These authenticity measures are directly compatible with our recovery process relying on a secondary device: Any secondary device that has become the new primary device can act again as intermediate CA to certify the keys of new secondary devices. Consequently, the length of the certification chain increases by one.

When Recovering with Recovery Users. In case the user did not register a secondary device, the problem becomes more interesting, as no such device can be used as a link in a chain to ensure authenticity. Naively, the signing keys of the old primary device could be extracted from hardware and backed up to allow for certifying the new devices key, which however would lead us back to the initial problem of having keys that are not bound to hardware. Instead, we propose a process to convince the TTP-CA to issue another certificate for the new device of the same user, as shown in Figure 6. The TTP-CA uses the same trust mechanism as for decryption rights and outsources the authorization check to the user's selected recovery user(s): First, the TTP-CA picks a challenge and encrypts this challenge for the (certified) encryption key of the user's old primary device. If the recovery user(s) were convinced that the recovery request and new device are legitimate, they help the new device to regain decryption capabilities for the user. Thus, the new device can decrypt the challenge and present the response to the TTP-CA, which then issues a certificate for the new signing keys.

4.4 Operations

To elaborate on the performance of our concept, Table 1 and 2 list the underlying operations required in the processes of Protocol 1 and 2. We focus on MU-PRE and secret sharing operations as they reflect the novelty of our concept and – depending on the scheme – might consume the most time. Pairing this list of operations with times of used schemes allows estimating the overall times for the processes. The tables refer to n users, t as threshold, o outgoing re-encryption keys from the old device, as well as l_A and l_B re-encryption steps (c.f. Figure 7).

Table 1: Operations for Setup and Data Sharing

Setup and Data Sharing	Client	Cloud Service
⓪ Setup (once)	1 KeyGen	-
① Register Secondary Device (rarely)	1 KeyGen + 1 RKGen	-
② Grant Access (sometimes)	1 RKGen	-
③ Upload Data (frequently)	1 Enc	-
④ Download Data (frequently)		
requester owns data	1 Dec^{l_A+1}	l_A ReEnc
requester does not own data	1 $\mathsf{Dec}^{l_A+l_B+2}$	(l_A+l_B+1) ReEnc

Table 2: Operations for Recovery

Recovery (rarely)	Registration	During Recovery		
	Client	Client	Other Client	Cloud Service
Ⓡ1 Secondary Device	-	o RKGen	-	-
Ⓡ2 Recovery User	1 RKGen	o RKGen	1 RKGen	-
Ⓡ3 Threshold t of n Users	n RKGen + n Split + n Enc	o RKGen	1 RKGen + t Dec	1 Reconstruct

Re-Encryption Chain Length. A chain of multiple re-encryption operations might be necessary to download data, as shown in Figure 7. While there might be multiple re-encryption chains for a requested file, we are interested in the shortest chain, i.e. shortest path in a graph of re-encryption keys. Such a chain consists of 1) l_A re-encryption steps between the devices of the data owner, and – in case of sharing – 2) 1 re-encryption step between the owner and the requester, as well as 3) l_B steps on the side of another user. In the default case, the requested data were not subject to a prior recovery operation. If the data owner requests such data from her primary device, no re-encryption is necessary ($l_A = 0$). The sub-chains increase in length a) by +2 each time the data was recovered with the help of recovery users, b) by +1 each time the data was involved in recovery via a secondary device, c) by +1 if the data is shared by a secondary device, and d) by +1 if the data is accessed by a secondary device.

4.5 Performance Improvements

After recovery, the added re-encryption steps have an impact on performance: a) the cloud storage service has to perform more re-encryption operations, which lead to b) ciphertexts at a higher re-encryption level that – depending on the use MU-PRE scheme – usually take longer to decrypt. Therefore, in this section, we discuss performance improvements, namely pre-computing and caching cryptographic operations, updating the network of re-encryption keys, and reducing the ciphertexts' re-encryption level. Updating keys and data trades an initial effort for a reduced subsequent effort. The actors may adaptively decide on when and where it is beneficial to apply refresh operations (e.g., on frequently used data).

Caching and Pre-Computing. Instead of re-encrypting data for the requester on demand, the cloud storage service may pre-compute the re-encryption operations to reduce the response time. Such pre-computation entails a higher initial computational effort and requires more storage space for the different ciphertext versions of the same data. Thus, pre-computation pays off for data that is frequently requested by the same users. This can be achieved by caching re-encryptions and deleting the least used when the cache grows too large.

The same approach may be applied for decryption: Clients pre-compute the decryption operation in the background or cache results for frequently used data to hide the computation costs from the users. We only apply this pre-computation or caching for the symmetric content encryption keys as the symmetric cryptography on the content is very efficient anyway. By wrapping the content encryption keys with a hardware-protected key, they can be securely stored on the user's device.

Re-Generating Re-Encryption Keys. Next, we consider re-generating re-encryption keys that were connected to the old device of a user who went through recovery. These keys can be a) outgoing, i.e., they transform ciphertext from the recovered user to other users, or b) incoming, i.e., they transform ciphertext from other users to the recovered user. Outgoing re-encryption keys of an old device are re-generated during recovery by the new device with its sk to also make new data that are encrypted for the new device accessible for the original receivers.

The problem is more interesting for incoming re-encryption keys to the old device: We need to convince the other user's device (i.e., source) to use its sk to generate a new re-encryption key towards the recovered user's new device. One possibility would be to prompt the source user to manually check if a re-encryption key (and thus decryption rights) may be generated for the requesting user. Instead, we suggest that the source user's device relies on the key authenticity mechanisms described in Section 4.3 to identify the key's owner, which enables to authorize requests without user interaction by comparing with previous decisions. Such re-generated incoming keys reduce the re-encryption chain's length, as data can be transformed directly rather than via links generated for recovery.

Refreshing Ciphertext Level. Added re-encryption steps due to recovery also increase ciphertext levels. MU-PRE schemes where the decryption time grows with the ciphertext level benefit from resetting this ciphertext level. When using hybrid encryption, the user's device decrypts the wrapped key and newly encrypts it as first level ciphertext, before uploading it. While the transmitted data size of wrapped keys is rather small, the MU-PRE operations on the client side might be more significant. However, these operations can be performed in the background by multiple threads invisible to the user. Nevertheless, the cloud storage service and user's device might wish to negotiate for which ciphertexts a refresh pays off.

5 Implementation

To validate and evaluate our proposed system, we first describe our proof of concept instantiation. Next, we discuss the MU-PRE scheme's implementation details and benchmark its performance. Finally, we estimate the costs of deploying our system in the cloud and elaborate on the binding of keys to the hardware.

Table 3: Execution Times of Our MU-PRE Implementation (in Milliseconds)

	level l	KeyGen	RKGen	Enc $M \to C^l$	ReEnc $C^l \to C^{l+1}$	Dec $C^l \to M$
Cloud Server (AWS c5.xlarge)	1	0.10	4.52	3.31	11.42	4.58
	2	-	-	-	15.84	7.53
	3	-	-	-	16.15	10.72
	4	-	-	-	16.45	13.91
	5	-	-	-	16.76	17.10
	6	-	-	-	-	20.29
Mobile Phone (OnePlus 6T)	1	1.10	47.22	34.35	118.53	47.85
	2	-	-	-	164.27	78.30
	3	-	-	-	168.24	111.97
	4	-	-	-	170.20	146.01
	5	-	-	-	171.99	181.09
	6	-	-	-	-	211.37

5.1 Instantiation

We have developed the cloud storage service as a Java web server, which we deployed on a cloud server (AWS c5.xlarge). An Android app acts as the client for both primary and secondary devices to enable users to conveniently interact with the data sharing and recovery capabilities. This app runs on a OnePlus 6T. During registration of secondary devices, these apps communicate through displaying and scanning QR codes. The apps employ push notifications via Google's Firebase for further communication between devices of one or different users. Finally, we extended and deployed a CAPSO[3] server as certification authority.

5.2 Cryptography Implementation

In this section, we describe our selection criteria for a MU-PRE scheme, give implementation details, and present performance measurements.

Scheme Selection. We selected the MU-PRE scheme by Cai and Liu [6], as it satisfies our requirements: Their scheme is unidirectional, non-interactive, collusion-safe, and CCA-secure. Unidirectional means that a re-encryption key from A to B can only be used to transform ciphertexts in that direction but not vice versa, which – as opposed to bidirectional schemes – does not force users to share decryption right in order to get access to the other users' data. In non-interactive schemes, a user can generate re-encryption keys towards another user by herself with her sk and the other user's pk. Collusion safeness captures that even if the proxy and receiver collude, they are not able to derive the sender's sk from the proxy's re-encryption key and receiver's key pair. Finally, their scheme is CCA-secure, which also covers processes where others encrypt data for a user and write it to her account with appropriate permissions.

[3] `https://ca.iaik.tugraz.at/`

Implementation. We implemented the MU-PRE scheme by Cai and Liu [6] with parameters chosen according to NIST's recommendations [20] for 128bit security. While Cai and Liu have defined their scheme for type-1 pairings, we have rewritten the scheme for more complex but also more efficient type-3 pairings. Our C implementation of the rewritten scheme builds on the RELIC toolkit [1] for their bilinear pairings on elliptic curves. We use the Java Native Interface (JNI) to integrate the compiled binaries into the server's and phone's Java environment.

Performance. Table 3 presents the benchmark results of our MU-PRE implementation, where a 128bit AES key is encrypted, repeatedly re-encrypted, and finally decrypted. Without previous recovery, ciphertexts are re-encrypted up to 3 times resulting in ciphertexts for level 1 to 4. The benchmark has been performed on both a cloud server (AWS c5.xlarge) as well as a mobile phone (OnePlus 6T). The presented times are an average of 100 runs on the phone and 10k runs on the server. Note that the currently single-threaded implementation only makes use of one core during the benchmark. For operations on different ciphertext levels, we observe the following: ReEnc on first-level ciphertexts is faster than on higher levels. The ReEnc time remains almost constant from second-level ciphertexts onward. The time to perform Dec grows linearly for each level. Using the least square method to fit a line to our measurements for Dec, we arrive at $1.31 + 3.16l$ milliseconds on the server and $13.43 + 33.14l$ milliseconds on the phone.

5.3 Deployment Cost Estimation

In Table 4, we evaluate the additional costs required to integrate our advanced cryptography into existing storage services, which includes storing/retrieving ciphertexts and re-encryption keys as well as re-encrypting the ciphertexts.

Costs Factors. When employing hybrid encryption, the symmetric encryption of the payload usually introduces little space overhead. Thus, we ignore the payload and focus on the symmetric key which is encrypted and transformed by MU-PRE as C^l. To store at AWS DynamoDB, we consider \$1.525/1M requests and \$0.306/1GB-month for first-level ciphertexts (3x384bit EC points = 144B) and re-encryption keys (5x384bit EC points = 240B), while incoming traffic is free. To get data, we consider three aspects: Firstly, we have \$0.305 per 1M requests and per 1kB block to obtain the ciphertext and l-1 relevant re-encryption keys. Secondly, to transform the ciphertext multiple times, we add \$0.194/h to run AWS EC2 c5.xlarge machines. Running multiple of our single-threaded re-encryption operations in parallel (on 10k ciphertexts across all ciphertext levels) shows that 2.15 times more operations can be performed by utilizing both cores and Intel's Hyperthreading technology on the AWS machine. We use this scaling factor on the measurements presented in Table 3. Finally, we add \$0.09/1GB to return the re-encrypted ciphertexts, which have a size of $144 + (l-1) \cdot 336B$. Our estimation is based on AWS prices for the EU-Frankfurt region in Oct 2019.

Example Scenario. We estimate the added costs of 1M users, who each upload 100 files and share their data using 10 re-encryption keys (100M C^1, 10M rk). Further, we consider that of these 100M files, 50% are downloaded by the same primary device (l=1), and 25% by a secondary device or another user's primary

Table 4: Deployment Costs on Amazon Web Services (for 100M items in $)

	DynamoDB		**EC2** (c5.xlarge)		**Traffic**	**Example Scenario**		
	C^1	rk	l	$C^1 \to C^l$	C^l	$\#C$	$\#$rk	Costs
Store	156.91	159.84	-	-	*free*	*100M*	*10M*	172.89
Get	30.50	30.50	*1*	-	1.30	*50.0M*	-	15.90
	30.50	30.50	*2*	14.28	4.32	*25.0M*	*25.0M*	19.90
	30.50	30.50	*3*	34.11	7.34	*12.5M*	*25.0M*	16.62
	30.50	30.50	*4*	54.32	10.37	*12.5M*	*37.5M*	23.34
								$248.64

device (l=2), while the remaining downloads use an even longer re-encryption chain (12.5% with l=3 and with l=4). Table 4 estimates a total of $248.64.

5.4 Hardware-Binding of Keys

In this section, we give details on how to bind MU-PRE keys to a device's hardware.

Mobile phones with technologies such as ARM's TrustZone [2] set up a trusted computing base by establishing a secure boot chain and eventually verifying the operating system's validity. In Android and iOS, the hardware-based protection for keys is integrated with the operating system's key chain, which ensures that the keys are only unlocked to be used in cryptographic operations for authenticated and authorized users. As these operating systems do not allow to run custom cryptographic code directly in the trusted execution environment, we follow a hybrid approach: We use hardware-protected keys for traditional cryptographic schemes to wrap our MU-PRE key material before storing them in the phone's file system. This MU-PRE key material is temporarily unwrapped for individual operations of our MU-PRE implementation. This approach is sufficient for our use case: Unauthorized people with physical access (e.g., thieves) are not able to authenticate to unlock or extract the keys needed for the unwrapping procedure. Further, a valid operating system sufficiently separates user-installed apps, so that malicious apps do not learn the user's data.

On PCs or servers supporting Intel's SGX [15], the cryptographic code can be deployed in a secure enclave, which protects (i.e., seals) the key material. We further need to perform user authentication inside the enclave before offering cryptographic operations on the protected key to prevent unauthorized access. Enclaves do not protect against malware on the operating system's level, as SGX was not designed to establish a secure boot chain that leads to a valid system. Therefore, malware could try to sniff and replay authentication data, e.g., by corrupting the drivers for the keyboard or fingerprint reader. Ruan [22] describes a mechanism to prevent such attacks by achieving secure input via secure output: The user is presented with a randomized keypad on her screen (via secure output). After clicking, the click coordinates can be passed without any further protection to the enclave for verification, as eavesdroppers cannot learn which keys were pressed due to their randomized position.

6 Discussion

Recovery with Threshold. Strategies to recovery from key-loss entail a trade-off between availability and confidentiality. When employing a strategy that relies on one piece of information, there is a risk that this information is not available when recovery becomes necessary. Distributing the recovery information to multiple locations increases the chances that it will be available for recovery, but leads to a higher risk that one location becomes corrupted. The same argument applies for trusted recovery users, as they might no longer be willing or able to help, or might even collude with the cloud storage service to decrypt data.

We presented a recovery mechanism that enables users to choose a more favorable trade-off for themselves: The user selects a set of recovery users and specifies a threshold of how many recovery users need to cooperate in order to recover successfully. This threshold also defines how many recovery users need to betray the user by colluding with the cloud storage service to decrypt her data.

Recovery Effort for Users. The effort to recover from key-loss depends on the users' recovery mechanism: If the user owns two or more devices, she can recover from the loss of one device by herself. A user who only owns a single device needs support from at least one other (recovery) user. After device- or key-loss, the user authenticates out-of-band to the recovery user (e.g., via a phone call) and convinces her to generate a re-encryption key for the user's new device. To reduce availability requirements, a user can also recruit multiple recovery users so that only a subset of these users need to be convinced to cooperate for recovery.

Data Sharing Performance. Our implementation of the MU-PRE scheme demonstrates the practical efficiency of the proposed system.

Cloud storage services perform re-encryption operations to make data accessible. Re-encrypting 1^{st} to 4^{th}-level ciphertexts takes an accumulated ≈44ms on our benchmark server, while significantly more powerful machines are available in cloud environments. Response times can be further reduced by adaptively caching or pre-computing re-encryptions.

For *clients* on both PCs and phones, our implementation is sufficiently fast for all operations. On our phone, re-encryption keys can be generated in ≈50ms, while encryption needs ≈35ms. To decrypt ciphertexts with levels ranging from 1^{st} to 4^{th}, our phone takes between ≈50ms and ≈150ms. These ciphertext levels depend on which devices are used to share and access the data. Besides optimizing the MU-PRE implementation or using more powerful phones, these times can be further improved by a) pre-computing and caching the decryption of the small symmetric content encryption keys and storing the results securely on the device given hardware-protection for keys, or b) resetting the ciphertext levels of ciphertexts that went through recovery in the background.

Improved Key Security. The data sharing and recovery mechanisms in our concept were designed so that the involved client-side key material does not have to be extracted from the client devices' hardware at any point. Therefore, we are able to employ technologies that bind the keys to the devices' hardware and only unlock the keys for authorized users. Consequently, attackers and malicious apps are also not to obtain the user's key material.

Fast Recovery without Re-Keying. Without hardware-bound keys, device-loss also implies potential key-compromise, e.g., if the device storing the key was stolen or has been lost and found. As unprotected keys can be extracted from the device, re-keying of all related ciphertexts becomes necessary. For a "naive" storage system using hybrid encryption, re-keying requires that wrapped symmetric keys for all ciphertexts are downloaded, decrypted, encrypted anew, and again uploaded. This causes substantial computation and communication effort for both the client as well as the cloud storage service. Until this costly process is completed, affected user might not be able to fully use the system and access their data. As our solution relies on hardware-binding, the keys cannot easily be extracted, and, therefore, no re-keying is necessary. Instead of the high re-keying effort, only re-encryption keys need to be generated, after which all access and data sharing capabilities are immediately restored.

7 Conclusion

In this paper, we proposed and implemented a cloud-based data sharing system, which 1) supports multiple devices per user, 2) offers three mechanisms to recover from device- and key-loss, and 3) enables to bind the keys to the device's hardware.

Our concept is based on multi-use proxy re-encryption to achieve end-to-end confidentiality. Users are not only able to store sensitive data at a semi-trusted storage service that is operated in the cloud, but can also share these data with authorized receivers via the cloud service. As users may own multiple devices, our concept enables them to access data and initiate sharing from each device.

The proposed strategies to recover from device-loss protect users from losing access to their encrypted data. While recovery is simple given a second device with full access, we also present mechanisms to recover with the help of one or more user-selected recovery users. We wish to highlight recovery with a threshold of multiple recovery users, as it reduces the risk of recovery users who – over time – become no longer willing or able to participate in the recovery process. The recovery mechanisms require little effort from the users: A user initially selects her recovery user(s) and later convinces them to aid her in the recovery process.

As our concept does not require to extract the devices' keys, these keys can be bound to the hardware. Hardware security features prevent thieves and malicious apps from extracting keys and form using them to access the users' data. This leads to another benefit: If a device is lost, it is no longer necessary to suspend access to the user's data while performing time-consuming and communication-intense re-keying of all ciphertexts on the user's client.

The concrete instantiation of our concept demonstrates its feasibility. Benchmarks of our MU-PRE implementation based on Cai and Liu's scheme [6] as well as our deployment cost estimation underline the concept's practical efficiency.

References

1. Aranha, D.F., Gouvêa, C.P.L.: RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`, accessed: 2019-07-04

2. ARM: TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone, accessed: 2019-06-28
3. Ateniese, G., Fu, K., Green, M., Hohenberger, S.: Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage. In: NDSS. The Internet Society (2005)
4. Bagherzandi, A., Jarecki, S., Saxena, N., Lu, Y.: Password-Protected Secret Sharing. In: ACM Conference on Computer and Com. Security. pp. 433–444. ACM (2011)
5. Blaze, M., Bleumer, G., Strauss, M.: Divertible Protocols and Atomic Proxy Cryptography. In: EUROCRYPT. LNCS, vol. 1403, pp. 127–144. Springer (1998)
6. Cai, Y., Liu, X.: A Multi-Use CCA-Secure Proxy Re-Encryption Scheme. In: DASC. pp. 39–44. IEEE Computer Society (2014)
7. Camenisch, J., Lysyanskaya, A., Neven, G.: Practical yet Universally Composable Two-Server Password-Authenticated Secret Sharing. In: ACM Conference on Computer and Com. Security. pp. 525–536. ACM (2012)
8. Dodis, Y., Reyzin, L., Smith, A.D.: Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. In: EUROCRYPT. LNCS, vol. 3027, pp. 523–540. Springer (2004)
9. Fu, K.E.: Group Sharing and Random Access in Cryptographic Storage File Systems. Ph.D. thesis, Massachusetts Institute of Technology (1999)
10. Goh, E., Shacham, H., Modadugu, N., Boneh, D.: SiRiUS: Securing Remote Untrusted Storage. In: NDSS. The Internet Society (2003)
11. Goyal, V., Pandey, O., Sahai, A., Waters, B.: Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data. In: ACM Conference on Computer and Communications Security. pp. 89–98. ACM (2006)
12. Hörandner, F., Krenn, S., Migliavacca, A., Thiemer, F., Zwattendorfer, B.: CREDENTIAL: A Framework for Privacy-Preserving Cloud-Based Data Sharing. In: ARES. pp. 742–749. IEEE Computer Society (2016)
13. Hörandner, F., Rabensteiner, C.: Horcruxes for Everyone - A Framework for Key-Loss Recovery by Splitting Trust. In: TrustCom/BigDataSE. IEEE (2019)
14. Huang, Z., Li, Q., Zheng, D., Chen, K., Li, X.: YI Cloud: Improving User Privacy with Secret Key Recovery in Cloud Storage. In: SOSE. pp. 268–272. IEEE Computer Society (2011)
15. Intel: Software Guard Extensions. https://software.intel.com/en-us/sgx, accessed: 2019-06-28
16. Jin, A.T.B., Ling, D.N.C., Goh, A.: Biohashing: Two Factor Authentication Featuring Fingerprint Data and Tokenised Random Number. Pattern Recognition **37**(11) (2004)
17. Kaliski, B.: PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC **2898**, 1–34 (2000)
18. Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., Fu, K.: Plutus: Scalable Secure File Sharing on Untrusted Storage. In: FAST. USENIX (2003)
19. Nagar, A., Nandakumar, K., Jain, A.K.: Biometric Template Transformation: A Security Analysis. In: Media Forensics and Security. SPIE Proceedings, vol. 7541, p. 75410O. SPIE (2010)
20. National Institute of Standards & Technology: SP 800-57. Recommendation for Key Management, Part 1: General (Rev 4). Tech. rep., NIST (2016)
21. Percival, C.: Stronger Key Derivation via Sequential Memory-Hard Functions (2009)
22. Ruan, X.: Intel Identity Protection Technology: the Robust, Convenient, and Cost-Effective Way to Deter Identity Theft, pp. 211–226. Apress, Berkeley, CA (2014)
23. Shamir, A.: How to Share a Secret. Commun. ACM **22**(11), 612–613 (1979)