A Decentralized Data Sharing Framework based on a Key-Redistribution method*

Fadi Barbàra[1,*,†], Mirko Zichichi[2,†], Stefano Ferretti[3] and Claudio Schifanella[1]

[1]Department of Computer Science, University of Turin, Turin, Italy Via Pessinetto 12, 10149, Turin, Italy
[2]Ontology Engineering Group, Universidad Politécnica de Madrid, Madrid, Spain
[4]Department of Pure and Applied Sciences, University of Urbino Carlo Bo, Urbino, Italy

Abstract

One of the problems of cloud-based data services is the trust involved in its management, since service managers can easily access the data on their servers. The problem is exacerbated in decentralized data services, where managers and operators are pseudo-anonymous by default, to the point where these systems are not compliant with data protection regulations such as GDPR. These problems have historically been dealt with data encryption, but this inhibits data sharing. To enable data-sharing for a encrypted decentralized file storage, we propose Key-Redistribution Proxy Re-Encryption (KeRePRE). KeRePRE is a decentralized and encrypted data-service where managers in the form of authorization servers are part of a threshold proxy re-encryption scheme. In particular, to solve the problem of malicious nodes, we extend the work in Umbral with a system based on a key-redistribution mechanism to add and remove managers in a decentralized and trustless way, and we provide a proof of concept implementation. Data access control is based on an access control list stored on a DLT which can be read-only accessed by the authorization servers.

Keywords

Proxy re-encryption, Threshold scheme, GDPR, Data Sharing, Decentralized File System

1. Introduction

Data have become valuable assets for individuals, businesses, and governments. The abundance of data generated by various sources, such as social media, sensors, and mobile devices, has led to the rise of big data and data-driven decision-making. However, the sheer volume and complexity of data have also created significant challenges in managing, processing, analyzing, and, most importantly, protecting them. This last point is where the role of a data intermediator becomes crucial. A data intermediator acts as a mediator between data holders and recipients, helping to manage the flow of data and ensure its quality and security. Data intermediation can be approached taking into consideration the (not so) recent surge of decentralized systems, such as Distributed Ledger Technologies (DLTs). These pave the way toward an intermediation

5th Distributed Ledger Technology Workshop, May 25-26, 2023, Bologna, Italy
*Corresponding author.
†These authors contributed equally.
fadi.barbara@unito.it (F. Barbàra); mirko.zichichi@upm.me (M. Zichichi); stefano.ferretti@uniurb.it (S. Ferretti); claudio.schifanella@unito.it (C. Schifanella)
https://fadibarbara.it (F. Barbàra); mirkozichichi.me (M. Zichichi); http://conceptbase.sourceforge.net/mjf/ (S. Ferretti); http://conceptbase.sourceforge.net/mjf/ (C. Schifanella)

where data recipients and processors are *constrained* to act exactly as the holder instructed, e.g., through smart contracts[1]. Decentralization, however, inevitably necessitates some forms of cryptography to render decentralized systems secure and protect data while sharing it. Comprehensive security is vital, in fact, if a user is put at the center of an environment where no one can be trusted, e.g., in a trustless or semi-trusted decentralized system. Different approaches involve the use of (t, n)-threshold cryptosystems in such semi-trusted decentralized environments [2, 3, 4]. These approaches involve multiple parties performing a cryptographic operation together, e.g., using a "share" of a secret in a secret-sharing scheme. An example is the Threshold Proxy Re-Encryption (TPRE) scheme that enables a data holder to encrypt some data and delegate some data intermediators (i.e., proxies) to re-encrypt the encryption key in favor of a data receiver [3, 4].

Most threshold cryptosystems, however, assume data intermediators will take good care of their secret share. In this paper, we argue that in a “real world” scenario, a key redistribution mechanism is needed to deal with parties that lose their share, are corrupted, or are faulty. By designing and implementing such a mechanism in a semi-trusted decentralized environment, it is possible to perform key rotation, addition, or deletion [5] and cope with situations where one/more keys get leaked/compromised, or some parties go rogue, or even when new parties need to get added and old parties get dismissed. Therefore we create a Key-Redistribution Proxy Re-Encryption (KeRePRE) scheme.

Furthermore, we place the employment of this mechanism in a specific strand of literature that focuses on data access control managed in a decentralized way through advanced cryptographic techniques. In particular, the works presented in the literature are often concerned with the types of data that are personal [2, 6, 7, 1]. The reason for this is that personal data almost always require strong data protection and security mechanisms, because they identify or render identifiable a data subject. Indeed, dedicated decentralized technologies can help companies comply with data protection regulations, such as the General Data Protection Regulation (GDPR) [8], and individuals exercise their rights. Our second contribution is, therefore, the design of a Personal Data Store (PDS) that allows data subjects to decide how/where to store the data and handle data encryption and key distribution using a TPRE with the key redistribution mechanism. Two main components are involved: Decentralized File Storage (DFS) and a DLT. The DFS embodies the data storage: it contains the data to encrypt or decrypt. DLTs allow for avoiding all the typical drawbacks of server-based approaches, such as censorship or single-point-of-failure, and offer features such as data traceability, verifiability, and, most importantly, smart contracts execution.

Our Contribution In summary our contributions are:

- We extend the work of Nunez in [4] proposing KeRePRE: a threshold based proxy re-encryption scheme with support of key-redistribution.
- We provide an implementation of its feasibility[1] tailored to the Umbral system, which can be independently used.
- We show how it is possible to use KeRePRE to create a decentralized PDS linked to a DLT to manage the access to personal data in a GDPR compliant way

[1]The implementation can be found at https://github.com/disnocen/umbral-rs

Outline The remainder of this paper is so composed. In Section 2 we provide an overview of threshold proxy re-encryption schemes and on personal data sharing mechanisms that involve a DLT. In Section 3.1 we introduce the actors, the architecture model of the system and the other building blocks that compose the KeRePRE system. In Section 4 we explain how key redistribution works within the Umbral system. In Section 5 we show how KeRePRE works and we analyze its security and data protection improvements in Section 6. Finally we conclude in 7

2. Related Works

2.1. Threshold Proxy Re-Encryption in DLTs

The first application of TPRE into a DLT environment was performed by the authors in [3]. They apply TPRE to the access permission mechanism of a consortium DLT. Authors in [9], on the other hand, propose a decentralized network for the management of a service that provides encryption and cryptographic access control. Their TPRE reference software is Umbral [4], that is the base implementation that we are going to refer throughout this work. The weak point that we argue is present in Umbral is the lack of a key redistribution mechanism.

More recently, authors in [2] proposed an architecture that converges TPRE and DLT consensus algorithm for the creation of a decentralized key management system in the Internet of Things. This system too lacks a key redistribution mechanism. [10] propose a GDPR-compliant data storage and sharing framework using blockchain for smart healthcare systems where a PRE network is used to share the encrypted data. Also in their case, there is no use of key redistribution mechanisms. Furthermore, their PRE network solution does not involve a threshold scheme and can lead to single-point-of-failures.

2.2. DLT and personal data sharing

Various works in the literature aim to propose DLTs-based architectures for data management to build novel smart services and to promote social good [11, 12]. These solutions usually store data outside the ledger, i.e., off-chain, and involve the DLT to provide transparency in the process of access to data while simultaneously enabling users to control their data. Systems based on DLTs and smart contracts can be leveraged in access control mechanisms to solve problems related to centralization and privacy leakage [13] and to store, share and transmit data securely. Many researchers have attempted to envision data management systems where the user retains control over data, and in some cases, GDPR compliance is taken into account. [6] provide a system where users have control over their personal data collection and where transaction history is recorded in a blockchain for data provenance. Their approach is GDPR compliant, but there is no specific discussion around their cryptographic solution. [14] propose an architecture based on distributed technologies to exchange health data, while the work of [7] enables healthcare data exchange through the exploitation of smart contracts for consent. Users keep a digital copy of their medical data in a personal data account that can be hosted on any cloud-based data management service. Users customize the dynamic consent preferences through smart contracts according to the type of data requested, by whom, and for what purpose. DeepLinQ [15] is a multi-blockchain architecture similar to our proposal. It aims to support

privacy-preserving data sharing in the healthcare sector through granular access control and smart contracts. Finally, authors in [16] present a PDS that allows users to collect, store and give third parties fine-grained access to their data using a Secret Sharing scheme. However their approach is not GDPR compliant as personal data are kept on-chain.

3. Background

In this section we first identify the actors involved in the system we propose, and we will give an overview of the system's architectural components (Section 3.1). We use the introduced terminology to explain how the building blocks of KeRePRE work in Sections 3.2 to 3.4.

3.1. Actors and architectural components

3.1.1. Actors

We define different actors that have one or more roles in the system. In detail, we identify the following actors:

- ***Data subject*** (DS) - The natural person that uses a personal device that in turn generates personal data.
- ***Data holder*** (DH) - The legal or natural person who has the right or obligation or the ability to make available specific data (both personal and non).
- ***Data intermediary*** (DI) - The legal or natural person who mediates between those holders who wish to make their data available and data recipients. We have two specializations of data intermediary:
  - ***DFS provider*** (SP) - The one that provides the access to the DFS. This actor provides functionalities attributed of storing and serving (encrypted) personal data.
  - ***Authorization Server*** (AS) - The one that provides the access to the DLT to the authorization service, i.e., takes part to the cryptosystem.
- ***Data recipient*** (DR) - The legal or natural person to whom the data holder makes data available.

Since the names of the actors involved in KeRePRE are different form the usual names due to the specific use case involved, we provide a comparison of the names for easy access in Appendix A.

3.1.2. Architecture Model

In the following, we use a model to refer to the elements managed in the system.

- The data holder actor controls a set of personal data that have not been encrypted, i.e., $\mathcal{D} = \{pd_l \mid 1 \leq l \leq o\}$ where o is the amount of pieces of data DH has.
- Furthermore, $\mathcal{K} = \{k_{pd_l} \mid \mathsf{Enc}_{k_{pd_l}}(pd_l), 1 \leq l \leq o\}$ is the data holder's set of keys used to encrypt personal data and $\mathcal{E} = \{epd_l \mid epd_l = \mathsf{Enc}_{k_{pd_l}}(pd_l), 1 \leq l \leq o\}$ is the set of encrypted personal data.

- Data holders and authorization servers control a set of capsules $\mathcal{C} = \{\gamma_{k_{pd_l}} \mid \gamma_{k_{pd_l}} = \mathsf{Enc}_{pk_{DH}}(k_{pd_l}), 1 \leq l \leq o\}$, where pk_{DH} is the public key of the data holder (see Definition 1, that contain a key used to encrypt a piece of personal data.
- We consider that all DFS providers SP store the data holders' set of encrypted personal data $edp \in \mathcal{E}$ and the associated set of decentralized identifiers used to identify the epd. In this case the decentralized identifier is equal to an hash pointer obtained by hashing the epd, i.e., $HP = \{hp_{epd_l} \mid hp_{epd_l} = \mathsf{Hash}(epd_l), 1 \leq l \leq o\}$ where Hash is a predetermined hash function (e.g., in the IPFS DFS these hash pointers are CIDs). Thus hp_{epd_x} is both the identifier of the epd_x datum in the DFS and the on-chain hash pointer, i.e., that will be stored in the DLT.

3.2. Proxy re-encryption schemes

A well-known problem faced by data holders (DHs) in a decentralized PDS is the lack of direct control over the outsourced data in $\mathcal{E}$ [17, 18] which raises security concerns especially in (pseudo) anonymous settings.

One of the most effective ways to deal with this issue is for the DH to encrypt the data before uploading it to the PDS [19]. This naive solution, though, inhibits the delegation of access (i.e. "sharing") to a data receiver DR of a piece of data pd_i, since the process requires DH to download, re-encrypt pd_i for DR and re-upload pd_i. To solve this issue Blaze *et al.* introduced the Proxy Re-Encryption (PRE) scheme in [20]. A PRE is a semi-trusted proxy that transforms a cyphertext encrypted for DH to a cyphertext encrypted for DR, without decrypting the cyphertext or leaking the related plaintext. Specifically, with a PRE, DH can encrypt pd_i under its own public key before uploading it to the PDS. After receiving the request of data sharing from DR, DH can generate a *proxy re-encryption key* and send it to the PRE. The PRE is then able to re-encrypt pd_i into a cyphertext under the public key of DR.

In the following we adapt the definition 3.1 of [4] to the case of a PDS:

Definition 1. *A Proxy Re-Encryption (PRE) scheme is a tuple of algorithms* $(\mathsf{KeyGen}, \mathsf{ReKeyGen}, \mathsf{Enc}, \mathsf{ReEnc}, \mathsf{Dec})$*:*

- $(sk_A, pk_A) \leftarrow \mathsf{KeyGen}(1^\lambda)$ *On input security parameter λ, the key generation algorithm* KeyGen *outputs a pair of secret and public keys (sk_A, pk_A) for user A.*
- $rk_{A\to B} \leftarrow \mathsf{ReKeyGen}(sk_A, pk_A, sk_B, pk_B)$ *On input the pair[2] of secret and public keys (sk_A, pk_A) for user A and the pair of secret and public keys (sk_B, pk_B) for user B the re-encryption key generation algorithm* $\mathsf{ReKeyGen}$ *outputs a re-encryption key $rk_{A\to B}$.*
- $c_A \leftarrow \mathsf{Enc}(pk_A, pd)$ *On input the public key pk_A and a piece of data $pd \in PD$, the encryption algorithm* Enc *outputs a cyphertext $c_A \in \mathcal{C}$*
- $c_B \leftarrow \mathsf{ReEnc}(rk_{A\to B}, c_A)$ *On input the re-encryption key $rk_{A\to B}$ and a cyphertext c_A, the re-encryption algorithm* ReEnc *outputs a second cyphertext $c_B \in \mathcal{C}$ or the error symbol $\perp$ indicating that c_A is invalid.*

[2]Note that this definition is not suitable for a data-sharing setting since both secrete keys are required. This is a general definition: a more appropriate definition used in our setting will be given in Section 3.4

- $pd \leftarrow \mathsf{Dec}(sk_A, c_A)$ *On input the secret key* sk_A *and a cyphertext* $c_A \in \mathcal{C}$*, the decryption algorithm* Dec *outputs the piece of data* pd *or the error symbol* $\perp$ *indicating that* c_A *is invalid.*

From Definition 1 it is easy to see that a PRE is an extension of a public key encryption scheme (PKE). Therefore a PRE must follow the security models of PKEs which presents an interesting challenge. On the one hand PREs have to guarantee confidentiality and validity of the cyphertexts as any PKE. On the other PREs have to allow re-encryption of cyphertexts. A thorough overview of how different schemes deal with the challenge is presented by Nunez *et al.* in [4].

3.3. Threshold cryptosystems

Intuitively a (t, n)-threshold cryptosystem is a system involving multiple parties in a set $\mathcal{P} = \{P_i\}_{i \in \mathcal{I}}$, where $\mathcal{I}$ is a set of indexes, that perform a cryptographic operation together. The distinguishing property is that only some of these parties, at least a *threshold* t, are required in order for the cryptosystem to be successful. In other words, at least t parties out of n have to be honest and follow the cryptosystem protocol. Since generally a cryptosystem is used to give access to some information, we say that $\Sigma = (t, n)$ is an *access structure.*

Algorithm 1 LsssPrep for a (t, n) secret sharing

Require: $\mathcal{I}$: set of identities of parties, t: threshold, s: secret

1: $n = \operatorname{len} \mathcal{I}$
2: **for** $i = 1 \dots t-1$ **do** ▷ Initialize a_i for $i = 0, \dots, t-1$
3: $a_i \leftarrow_{\$} \mathbb{F}$
4: **end for**
5: $a_0 = s$ ▷ $q(0) = s$
6: Initialize $q(\cdot) = a_0 + \sum_{i=1}^{t-1} a_i \cdot^i$ ▷ Polynomial initialization
7: **for** i in $\mathcal{I}$ **do** ▷ The values of $\mathcal{I}$ must be numbers in the field $\mathbb{F}$
8: Send $(i, q(i))$ to party i
9: **end for**

Two research strands involve threshold cryptosystems: threshold signing and secret sharing. On the one hand, (t, n)-threshold signing, considered to have been introduced by Desmedt in 1987 [21], is a process where t-of-n parties are involved into signing a message on behalf of all n participants. On the other hand, in (t, n) secret sharing a secret s is split into n different parts called *shares* (or *fragments*) such that t of them are necessary to reconstruct the original s. Among many secret sharing schemes, we focus on the Shamir Secret Sharing (SSS) one since it is the one used in both Umbral and the redistribution mechanism our work is based on.

The scheme is based on polynomial interpolation over a field. In a field $\mathbb{F}$, it is well known that given t points in the 2-dimensional plane $\{(x_i, y_i)\}_{i=1}^{t}$ there is one and only one polynomial $q(x)$ of degree $t-1$ such that $q(x_i) = y_i$ for each $i = 1, \dots, t$. Assume a *secret* number s. As mentioned, this secret can be split and shared to n parties in such a way that t of those shares are needed to reconstruct s: see LsssPrep in Algorithm 1.

Since the shares are distinct points on a plane for polynomial $q(\cdot)$, the SSS scheme uses Lagrange Polynomials applied to the shares, as presented in Algorithm 2, to reconstruct the secret.

Algorithm 2 LsssRec for a (t, n) secret sharing

Require: $\mathcal{I}$: set of identities of parties, t: threshold

1: $n = \text{len}\,\mathcal{I}$
2: Wait for t shares $i_{j_1}, q(i_{j_1})$ from parties $i_{j_1}, \ldots, i_{j_t} \in \mathcal{I}$
3: $L = 0$
4: **for** $k = 1 \ldots t$ **do**
5: $\quad \lambda_k = \prod_{w=1, w\neq k}^{t} \frac{i_{j_w}}{i_{j_w} - i_{j_k}}$ ▷ Create a Lagrange basis
6: $\quad L = L + \lambda_k q(i_{j_k})$ ▷ Use Lagrange polinomials to create $q(0) = s$
7: **end for**
8: **return** L ▷ $L = q(0) = s$

3.4. Threshold PRE

One of the goals of this paper is to prove that it is possible to manage decentralized PDSs in a way that is dynamically secure against malicious nodes. By dynamically secure we mean that the PDS is able to maintain the properties insured during its instantiation even if a part of the operator nodes becomes malicious in the future (e.g. it get compromised by an external actor) and/or if new nodes join the managerial part of the PDS.

For these reasons we develop from the work of Nunez [4], which is called Umbral. Umbral is a threshold PRE which uses a Key Encapsulation Mechanism (KEM) to obtain a Data Encryption Method (DEM). More explicitly, in Umbral, each file $pd \in \mathcal{D}$ from a DH is encrypted with a symmetric key $K \in \mathcal{K}$. The encrypted file is a couple $(\mathsf{Enc}(pd, K), \mathsf{Enc}_{pk_{DH}}(K))$. The Umbral threshold PRE leverages the ReKeyGen procedure to output multiple shares of the re-encryption key via a SSS scheme. These share are called fragments or more concisely $kFrag$, in [4], and are distributed to the node operators as part of the ReKeyGen routine.

More formally a (t, n) instance of the threshold PRE Umbral for data sharing is different from an ordinary PRE (see Section 3.2) in the following algorithms:

- $kFrag_1, \ldots, kFrag_n \leftarrow \mathsf{ReKeyGen}(sk_A, pk_B, n, t)$: On input the secret key sk_A of user A (generally the DH), the public key pk_B of user B (generally the DR), a number of shares n and a threshold t, the re-encryption key generation algorithm ReKeyGen computes the re-encryption key $rk_{A\to B}$ and then uses SSS scheme to share if in n different $kFrag$s, where $kFrag_i = (id_i, rk_i, opt)$, with id_i the identity of node i, rk_i its share of the re-encryption key and opt optional arguments depending on the implementation.

Moreover, the KEM part of Umbral is so composed:

- $(K, \gamma_K) \leftarrow \mathsf{Encapsulate}(pk_A)$: On input the public key pk_A of user A, the algorithm Encapsulate outputs a symmetric key $K \in \mathcal{K}$ used to encrypt the data and a capsule $\gamma_K = \mathsf{Enc}(K)$.

- $cFrag_i \leftarrow \mathsf{ReEncapsulate}(kFrag_i, \gamma_K)$: On input a key share $kFrag_i$ and a capsule γ_K, algorithm ReEncapsulate outputs a share (or fragment) of the capsule $cFrag_i$ of the capsule γ_K.
- $K \leftarrow \mathsf{DecapsulateFrags}(sk_B, pk_A, \{cFrag\}_{i=1}^{>t})$: On input the secret key sk_B of user B, the public key pk_A of user A and at least t $cFrag$s, algorithm DecapsulateFrags outputs K (note that it is the same K of algorithm Encapsulate).

Figure 1 highlights the flow of the procedures which we explain in details in Section 5.1

3.5. Key redistribution mechanisms

Most threshold cryptosystems, and particularly secret sharing schemes, assume parties will take good care of their share. In fact, if some party P_i loses a share, it is generally said that P_i is corrupted or faulty. No further analysis on P_i is done, since from the point of view of the threshold cryptosystem no single party is important as long as the majority or minority of them is still honest, depending on the access structure of the cryptosystem.

On the other hand, real world deployments of these systems have to deal with such problems. For example in Proactive Secret Sharing schemes [22, 23] the participants refresh (or *rotate*) their key shares periodically in order to avoid these kinds of problems or at least mitigating them. The process known as key-refresh or key-rotation.

However, in proactive secret sharing schemes, the access structure is not changed: the set of parties required for threshold secret sharing are the same before and after the key-refresh. Therefore the only way to extend or shrink the access structure once it is in place is by performing a new distribution of the shares. This is costly, since it requires DH to recompute all the shares. Consequently, new approaches have been proposed in the literature to deal with this issue.

Among those proposals, one that is beneficial for the goal of this paper is the process of *redistribution* of shares. Unlike key refreshing schemes, a redistribution of shares is performed by the *AS*, supports the change of the access structure and requires no input by the DH (beside some authorization if needed by the general system).

In this proposal and proof of concept[3], we use redistribution method as presented by Desmedt *et al.* in [21]. The method leverages a SSS scheme once more and treats each share as a secret on its own. More formally, given a (t, n)-SSS scheme with shares $s_1, \ldots, s_n$:

- $s'_1, \ldots, s'_m \leftarrow \mathsf{DesRedistr}(s_1, \ldots, s_n)$: On input $t \leq k \leq n$ shares from a (t, n)-SSS for secret s, algorithm DesRedistr outputs m secret shares $s'_1, \ldots, s'_m$ such that k of them are needed to reconstruct s.

In practice DesRedistr transforms a (t, n)-SSS into a (k, m)-SSS.

It is easy to see that if $k = t$ and $m > n$ then DesRedistr adds a new party to the access structure, while if $t < m < n$ then DesRedistr removes party from the access structure. We will see in Section 6.1 the constraints on k and m related on t and n. A working example tailored to our purposes is presented in Algorithm 3.

[3] A working implementation of the key redistribution for KeRePRE can be found at https://github.com/disnocen/umbral-rs/blob/master/src/internal/keyredistrib.rs

4. Key redistribution within the Umbral system

Since Umbral does not use a key redistribution mechanism, then it is impossible to perform key rotation, addition or deletion. Consequently, it is impossible for us to apply Umbral as is. In this section we show how we extend the Umbral functions to create a real-world ready threshold PRE and in Section 5 we describe how the system can be used in a decentralized PDS to perform actions equivalent to key deletion, and key addition, beside simple refresh. We use the terminology as explained in Section 3.1.

We assume that either DH or DR triggers a key redistribution. Note that this triggering may be part of a notification system in the application that asks DH or DR if they want to act to mitigate a potential threat (such as a share corruption in one of the operator nodes). We show a representation of the dynamics of the extended system in Figure 1.

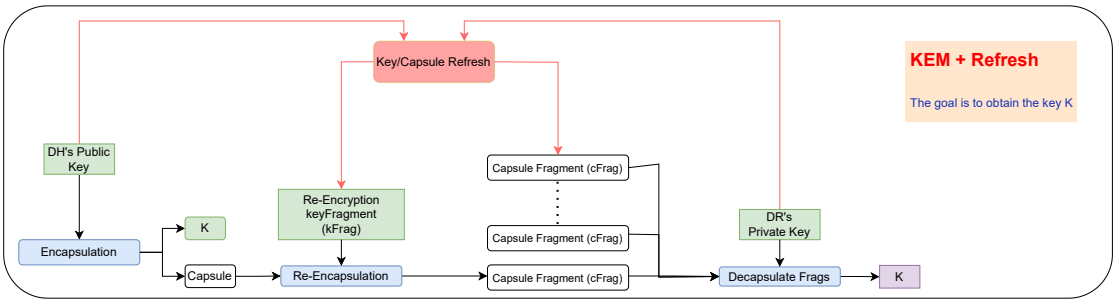

Figure 1: The image represents the Umbral work flow with our key redistribution extension (in red). Either DH or DR can trigger a key redistribution procedure. The nodes in the threshold proxy re-encryption operate the $kFrag$ and $cFrag$ redistribution.

The key-redistribution mechanism in KeRePRE is split into two algorithms: the first algorithm deals with the $kFrag$s while the second deals with the $cFrag$s. Here we deal with the $kFrag$ redistribution. the $cFrag$ update with the new share $kFrag'$ is easier to understand and less relevant for the comprehension of the whole system. the interested reader can find it in Appendix B.

4.1. Kfrags redistribution

A $kFrag$ redistribution is a procedure that transforms a $kFrag$ into a $kFrag'$. More formally:

- $(id, rk', opt) \leftarrow \mathsf{kFragRedistr}((id, rk, opt))$: On input a $kFrag$, the kFragRedistr algorithm outputs a new $kFrag$ with the updated re-encryption key share.

In particular kFragRedistr focuses on the update of the rk component into a rk' component. The complete kFragRedistr algorithm for a (t, n) threshold cryptosystem performed by each party $P_i, i = 1 \ldots n$ is presented in Algorithm 3.

To do it we use the Desmedt routine DesRedistr introduced in Section 3.5 (lines 7-14 of Algorithm 3). Instead of passing the id to the LsssPrep routine as done in Algorithm 1, we pass the hashed id hid instead: this change is done to maintain compatibility with the original

Algorithm 3 kFragRedistr for a (t, n) threshold scheme

Require: $kFrag_i = (id_i, rk_i, U^{rk_i}...), D, sid$

1: Get id_j from each party P_j
2: **for** $j = 1 \dots n$ **do**
3: **if** $j \neq i$ **then**
4: Compute $hid_j = H(D, id_j)$
5: **end if**
6: **end for**
7: $\{s_{i,j}\}_{\substack{j=1 \\ j\neq i}}^{n} \leftarrow \mathsf{LsssPrep}(\{hid_j\}_{\substack{j=1 \\ j\neq i}}^{n}, rk_i)$
8: **for** $j = 1 \dots n$ **do**
9: **if** $j \neq i$ **then**
10: P_i sends share $(id_i, s_{i,j})$ to party P_j
11: **end if**
12: **end for**
13: Wait for all shares $\{(id_j, s_{j,i})\}_{\substack{j=1 \\ j\neq i}}^{n}$ from parties P_j
14: $rk' \leftarrow \mathsf{LsssRec}(\{(id_j, s_{j,i})\}_{\substack{j=1 \\ j\neq i}}^{n})$
15: Erase rk_i and $s_{i,j}\ \forall j$
16: Output $kFrag'_i = (id_i, rk'_i, U^{rk'_i}...)$

Umbral protocol and does not affect the security of the Desmedt key-redistribution protocol. On the other hand, note that we pass the actual id as parameter in the LsssRec at line 14.

5. Use Case

In this section we describe a system that fully exploits our proposed scheme, i.e., a decentralized PDS. Our proposal consists of the design of a PDS that addresses two main issues: the lack of transparency in managing personal information and the inability to access and make personal data interoperable.The PDS can be intended as the first step toward this aim, relying on a new user-centered model for managing personal data, where storage is decoupled from the application logic. Furthermore, providers of personal data apps and data intermediaries can exploit PDS to prove their compliance with regulations (e.g., GDPR) [24]. The resulting PDS system we propose is compliant with the GDPR, thus protecting users' data, and it promotes a transparent personal data sharing. The use of DLTs and DFS is of paramount importance in our system architecture. DLTs provide the technological guarantees for trusted data management and sharing, as they can offer a fully auditable decentralized access control policy management and evaluation. In the view of the GDPR, this makes it possible to check whether the involved actors comply with the regulation or not, e.g., the trace personal information sharing can help data processors, and controllers easily demonstrate their compliance transparently. As concerns DFS, its combined use with DLT allows overcoming the typical scalability and privacy issues of the latter while preserving the benefits of decentralization. In practice, DFS is leveraged for

storing the actual data outside the DLT, i.e., through "off-chain" storage, and tracing all the data references in the DLT (i.e., "on-chain").

The general idea is straightforward: a data subject is the user of a personal device that generates different kinds of personal data; a data holder (which can be the data subject itself) stores and maintains such data in a PDS and a cryptographically immutable reference is stored in a DLT. In the following, we illustrate the choices behind using such components and their interactions.

5.1. Components Design

5.1.1. Cryptosystem and Wallet

We assume each actor has a unique pair of asymmetric keys obtained via KeyGen (see Definition 1) In particular DH has key-pair (sk_{DH}, pk_{DH}), DR has key-pair (sk_{DR}, pk_{DR}), and SP_i has key-pair (sk_{SP_i}, pk_{SP_i}).

A data holder DH encrypts personal data pd using a symmetric key $K \in \mathcal{K}$. Then, the key is placed in a capsule γ through the Encapsulate routine with input pk_{DH} (See Section 3.4).

5.1.2. (Decentralized) File Storage component

In our design, personal data is kept in a DFS associated with a data subject, i.e., the set of encrypted personal data referring to the subject that are stored in a DFS. The use of a P2P network and data replication mechanisms inherent in a DFS, make it so that the storage of personal data is decoupled from both the DLT and the personal device to provide wider data availability. This allows having different DLTs and/or services to refer to the same data storage system and facilitates the creation of a PDS in the perspective of data portability.

Uniquely referenced means that the resource containing the piece of personal data, e.g., data contained in the decentralized file storage, should be identified by making use of a specific protocol to keep the content unmodified for verifiability. Specifically, instead of referring to a resource "normally", i.e., *"resource-name-1"*, we make use of the resource content hash digest, e.g., by using the IPFS content id or CID *"QmdmQXB2m...KxDu7Rgm"*. This is in line with the fact that if the content was a specific one at the time of storing the piece of data in the PDS, an audit must verify that, subsequently, the file may have been altered.

5.1.3. Distributed Ledger Technology component

The primary use of the DLT is the execution of smart contracts implementing personal data access control. Access to the personal data stored in PDS can be allowed by the data holder through smart contracts. These access control smart contracts encode the eligibility of data access through a data structure, namely an access control list (ACL). In practice, each piece of encrypted personal data $epd \in \mathcal{E}$ is referenced in a specific smart contract through its on-chain hash pointer hp_{epd}. Thus, the smart contract stores a subset of the HP set held in the DFS, i.e., $HP^{\textit{on-chain}} \subseteq HP$. The ACL can be (i) directly modified by the DS or (ii) indirectly modified by the data holder DH based on a policy set by the subject. Once a recipient is listed in the ACL, i.e., authorized to access some content, the DLT providers and authorization servers can verify this

information through the ACL stored in the ledger. Eventually, when the recipient demonstrates to own the address listed in the ACL, servers release the $\gamma_{K_{pd}}$ capsule that includes the k_{pd} content key needed to decrypt the encrypted data epd.

5.1.4. Capsule Distribution Mechanism

The first operation consists of storing the encrypted data epd in the PDS and obtaining the reference to the data, i.e., the hash pointer hp_{epd}. Then the access control smart contract is updated with the hash pointer hp_{epd}. While this can be considered a setup, the first real capsule distribution phase occurs when the holder shares the capsule $c_{k_{pd}}$ associated to epd with n authorization servers AS. The holder creates n fragments of the capsule such that $c_{k_{pd}} = \sum_i^n c_{k_{pd}}^i$ and each server receives its own capsule fragment. Here the sum operation represents the fragment aggregated function associated with the SS or TPRE methods, which is discussed in the next paragraph.

A data recipient may be allowed to access the stored data simply because the holder adds pk_{DR} to the ACL in the smart contract The second capsule distribution operation comes after the data access request made by the recipient to the authorization servers. The data access request is composed of these elements $\{pk_{DR}, \textit{contractAddr}, hp_{epd}, \textit{sign}\}$, where *contractAddr* is the address of the smart contract containing the ACL. The *sign* element is the signature of a challenge-response message to be signed with sk_{DR}, to allow each server to identify the data recipient: this way the recipient proves that it owns the secret key sk_{DR}. Upon receiving the data access request and verified DR's response, t authorization servers check the ACL for the presence of pk_{DR}. If pk_{DR} is in the ACL, then each of the t servers releases its owned capsule fragment to the recipient. Finally, DR obtains K_{pd} via DecapsulateFrags.

5.2. Adding new members

Assuming KeRePRE grows in its user base, it is important for the system to scale accordingly. On the one hand the ACL management can not scale: the ACL is managed by a smart contract, therefore scaling that part means dealing with the topic of blockchain scaling, which is outside the scope of the paper. On the other hand, it is possible to add new authorization servers to the PDS management. Using terminology from Section 3.1 this means that new ASs have to be added to the access structure.

To see how it is possible, assume the current access structure for a PDS is (t, n), i.e. t ASs among n are needed to perform the ReEncapsulate algorithm for DR so that DR can obtain K via the DecapsulateFrags algorithm on the $\{cFrag\}_{i=1}^{\geq t}$ (see Section 3.1). Then, a new node SP_{n+1} can be added by performing kFragRedistr to create a $(t, n+1)$ access structure. Specifically, with reference to Algorithm 3, if is possible to derive hid_j for $j = 1, \ldots, n+1$ and LsssPrep can be called for $\{hid\}_{j=1}^{n+1}$. Furthermore, it is easy to see how the access structure can be incremented not just by 1, but for arbitrary $\nu > 0$ using the same method and in just one iteration create a new access structure of $(t, n+\nu)$.

5.3. Members removal

It easy to imagine that in the course of operations, at least a subset of nodes becomes faulty or are compromised. While ascertain *when* a AS has become malicious is outside the scope of the paper, we focus on how to deal with such cases.

As in Section 5.2, assume a current access structure of (t, n). We split the explanation into two parts: we first deal with the case where there are still m honest nodes with $t \leq m < n$ and then we deal with the case where $m < t$.

If there are m honest nodes, $t \leq m < n$, then it is possible to perform kFragRedistr involving only those m honest nodes and excluding the $n - m$ malicious ones. Since the m nodes are honest by hypothesis, then we can trust them to perform data deletion, and therefore exclude the $n - m$ malicious nodes forever, as we explain in Section 6.1. This procedure creates a new access structure of (t, m).

On the other hand, if $m < t$, then the system is highly compromised and it is impossible to have a secure redistribution mechanism at this point, see proof of Theorem 1.

6. Analysis

6.1. Security

The main innovation of the system relies on the extension of the threshold proxy re-encryption to accommodate the decentralized and encrypted data management with a dynamic access structure. For this reason the security analysis is focused on this part.

In particular, we want to prove the security of member addition (Section 5.2) and member deletion (Section 5.3). We start with a definition for security in the context of a share-redistribution scheme. Recall that an *access structure* (t, n) means that at least t parties out of n are needed to reconstruct a secret s.

Definition 2. *Let* Redist *be a share redistribution scheme form an access structure $\Sigma = (t, n)$ to a access structure $\Sigma' = (k, m)$ for a secret s, with $m \geq t$. Let $\mathcal{P}$ be the set of parties for Σ, $\mathcal{P}'$ the set of parties in Σ' such that $|\mathcal{P} \cap \mathcal{P}'| \geq t$. Then* Redist *is secure if after its run parties from the set $\mathcal{P} \setminus \mathcal{P}'$ are not able to reconstruct the secret s anymore.*

We are now ready to state:

Theorem 1. *In the hypothesis of Definition 2, if $2t > n$, then* kFragRedistr *as presented in Algorithm 3 is a secure share redistribution scheme from $\Sigma = (t, n)$ to $\Sigma' = (k, m)$ with $m \geq t$ and $k \leq m$.*

We give a sketch of the proof below. A complete proof will appear in the complete version of this work.

Proof. The proof strongly follows the work of Desmedt *et al.* [21], since the routine kFragRedistr is inspired by it.

First of all, note that a change from an access structure $\Sigma = (t, n)$ to a access structure $\Sigma' = (k, m)$ is feasible if and only if there are still are at least t honest parties, otherwise it is

impossible to reconstruct the secret s in the first place. This is equivalent to ask for a honest majority since $t > \lfloor \frac{n}{2} \rfloor$. Furthermore, note that if $m < t$ then it is not possible to reconstruct the secret, since t is the least amount of number of parties in $\mathcal{P}$ needed to reconstruct s according to Σ. If all the constraints are satisfied, then kFragRedistr is equivalent to the system of Theorem 1 in [21]. Consequently, it is possible to apply Corollary 3 of [21] and conclude that it is sufficient that all the honest parties in $\mathcal{P}$ erase rk_i and $s_{i,j}\ \forall j$ to guarantee that parties in $\mathcal{P} \setminus \mathcal{P}'$ can not reconstruct secret s. Parties are required to do this operation in Line 15 of Algorithm 3. □ □

6.2. Data Protection

Under the GDPR, data controllers (the authorization servers in our use case) are required to implement appropriate security measures to ensure the confidentiality and integrity of personal data (Article 32). One of the recommended security measures is the use of encryption to protect the data from unauthorized access. However, if an encryption key is leaked, it can pose a significant risk to the confidentiality and integrity of personal data. In such a scenario, data controllers must take immediate action to comply with the GDPR (Articles 33 and 34). We argue that KeRePRE improves the efficiency of some steps that need to be performed by the controller in such a case. The main step is the one referred to as **Implement corrective measures**, i.e., based on the findings of the security review, the data controller should implement corrective measures to prevent future breaches; this may include strengthening its encryption protocols and improving access controls. With KeRePRE, the authorization servers are facilitated in the provision of key deletion and addition procedures, such that they can comply with implementing corrective measures.

7. Conclusions and Future Works

In this paper we showed how a threshold proxy re-encryption system may be used to obtain a decentralized and secure personal data storage once extended with key redistribution. We showed also how a DLT may be used to also decentralize the management of accesses and their delegations.

While an implementation of the key re-distribution is available online[4], in the future we aim to complete the implementation of the whole system.

References

[1] M. Zichichi, S. Ferretti, G. D'Angelo, V. Rodríguez-Doncel, Personal data access control through distributed authorization, in: 2020 IEEE 19th International Symposium on Network Computing and Applications (NCA), IEEE, 2020, pp. 1–4.

[2] Y. Chen, B. Hu, H. Yu, Z. Duan, J. Huang, A threshold proxy re-encryption scheme for secure iot data sharing based on blockchain, Electronics 10 (2021) 2359.

[3] X. Chen, Y. Liu, Y. Li, C. Lin, Threshold proxy re-encryption and its application in blockchain, in: Cloud Computing and Security: 4th International Conference, ICCCS 2018,

[4]The implementation can be found at https://github.com/disnocen/umbral-rs

Haikou, China, June 8–10, 2018, Revised Selected Papers, Part IV 4, Springer, 2018, pp. 16–25.
[4] D. Nuñez, Umbral: A Threshold Proxy Re-Encryption Scheme, Technical Report, NuCypher Inc., 2018.
[5] J. Aumasson, A. Hamelink, O. Shlomovits, A survey of ECDSA threshold signing, IACR Cryptol. ePrint Arch. (2020) 1390. URL: https://eprint.iacr.org/2020/1390.
[6] M. M. Merlec, Y. K. Lee, S.-P. Hong, H. P. In, A smart contract-based dynamic consent management system for personal data usage under gdpr, Sensors 21 (2021) 7994.
[7] M. Koscina, D. Manset, C. Negri, O. Perez, Enabling trust in healthcare data exchange with a federated blockchain-based architecture, in: IEEE/WIC/ACM International Conference on Web Intelligence-Companion Volume, 2019, pp. 231–237.
[8] European Parliament, Regulation (eu) 2016/679, 2016.
[9] M. Egorov, M. Wilkison, D. Nuñez, Nucypher kms: Decentralized key management system, arXiv preprint arXiv:1707.06140 (2017).
[10] P. Bai, S. Kumar, K. Kumar, O. Kaiwartya, M. Mahmud, J. Lloret, Gdpr compliant data storage and sharing in smart healthcare system: a blockchain-based solution, Electronics 11 (2022) 3311.
[11] M. Naz, F. A. Al-zahrani, R. Khalid, N. Javaid, A. M. Qamar, M. K. Afzal, M. Shafiq, A secure data sharing platform using blockchain and interplanetary file system, Sustainability 11 (2019) 7054.
[12] V. Ortega, F. Bouchmal, J. F. Monserrat, Trusted 5G vehicular networks: Blockchains and content-centric networking, IEEE Vehicular Technology Magazine 13 (2018).
[13] M. Jemel, A. Serhrouchni, Decentralized access control mechanism with temporal dimension based on blockchain, in: 2017 IEEE 14th International Conference on e-Business Engineering (ICEBE), IEEE, 2017, pp. 177–182.
[14] D. Hawig, C. Zhou, S. Fuhrhop, A. S. Fialho, N. Ramachandran, Designing a distributed ledger technology system for interoperable and general data protection regulation–compliant health data exchange: a use case in blood glucose data, Journal of medical Internet research 21 (2019) e13665.
[15] E. Y. Chang, S.-W. Liao, C.-T. Liu, W.-C. Lin, P.-W. Liao, W.-K. Fu, C.-H. Mei, E. J. Chang, Deeplinq: distributed multi-layer ledgers for privacy-preserving data sharing, in: 2018 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR), IEEE, 2018, pp. 173–178.
[16] Z. Yan, G. Gan, K. Riad, Bc-pds: protecting privacy and self-sovereignty through blockchains for openpds, in: 2017 IEEE Symposium on Service-Oriented System Engineering (SOSE), IEEE, 2017, pp. 138–144.
[17] K. Popovic, Z. Hocenski, Cloud computing security issues and challenges, The 33rd International Convention MIPRO (2010) 344–349.
[18] K. Ren, C. Wang, Q. Wang, Security challenges for the public cloud, IEEE Internet Computing 16 (2012) 69–73. doi:10.1109/MIC.2012.14.
[19] S. Sundareswaran, A. Squicciarini, D. Lin, Ensuring distributed accountability for data sharing in the cloud, IEEE Transactions on Dependable and Secure Computing 9 (2012) 556–568. doi:10.1109/TDSC.2012.26.
[20] M. Blaze, G. Bleumer, M. Strauss, Divertible protocols and atomic proxy cryptography, in:

K. Nyberg (Ed.), Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding, volume 1403 of *Lecture Notes in Computer Science*, Springer, 1998, pp. 127–144. URL: https://doi.org/10.1007/BFb0054122. doi:10.1007/BFb0054122.

[21] Y. Desmedt, S. Jajodia, Redistributiong secret shares to new access structures and its applications, Technical Report ISSE TR-97-01, George Mason University, 1997.

[22] V. Nikov, S. Nikova, On proactive secret sharing schemes, in: H. Handschuh, M. A. Hasan (Eds.), Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers, volume 3357 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 308–325. URL: https://doi.org/10.1007/978-3-540-30564-4_22. doi:10.1007/978-3-540-30564-4_22.

[23] A. Herzberg, S. Jarecki, H. Krawczyk, M. Yung, Proactive secret sharing or: How to cope with perpetual leakage, in: D. Coppersmith (Ed.), Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27-31, 1995, Proceedings, volume 963 of *Lecture Notes in Computer Science*, Springer, 1995, pp. 339–352. URL: https://doi.org/10.1007/3-540-44750-4_27. doi:10.1007/3-540-44750-4_27.

[24] M. Zichichi, S. Ferretti, G. D'Angelo, V. Rodríguez-Doncel, Data governance through a multi-dlt architecture in view of the gdpr, Cluster Computing 25 (2022) 4515 – 4542. doi:10.1007/s10586-022-03691-3.

A. Umbral and KeRePRE Comparison

We present in Table 1 how the actors involved in KeRePRE relate with the proxy re-encryption scheme in Umbral.

KeRePRE	Umbral
Data Subject	N/A
Data Holder	Alice
DFS Provider	N/A
Authorization Server	Proxy Re-encryption Node
Data Recipient	Bob

Table 1
Comparison of the names between the Umbral project and our extension KeRePRE

B. cFrag

As mentioned in Section 3.4, for each $kFrag$ there is a $cFrag$. The latter is used by DR to recover the data after a Re − Encryption has been performed by the node operators.

Algorithm 4 shows how a $cFrag$ is updated. Note that cFragRefresh must be performed after kFragRedistr since knowledge of the new rk' is necessary to operate the update of the $cFrag$. To see why the change works note that:

Algorithm 4 cFragRefresh for a (t, n) threshold scheme

Require: $cFrag_i, rk_i, rk'_i, sid$

1: $(E_i, V_i, id_i, X_A) \leftarrow cFrag_i$
2: $E'_i = E_i^{rk'_i/rk_i}$
3: $V'_i = V_i^{rk'_i/rk_i}$
4: Output $cFrag'_i = (E'_i, V'_i, id_i, X_A)$

$$E'_i = E_i^{\frac{rk'_i}{rk_i}} = (E^{rk_i})^{\frac{rk'_i}{rk_i}} = E^{rk'_i} \tag{1}$$

so E'_i is constructed as if it came directly from the ReEncryption function. Moreover, the change does *not* require the nodes operators to know E (which would be unfeasible because of the discrete logarithm problem). All these considerations works similarly for V'_i.