UMBRAL: A THRESHOLD PROXY RE-ENCRYPTION SCHEME

DAVID NUÑEZ

NICS Lab, University of Malaga, Spain
&
NuCypher Inc.

Abstract. This document describes the Umbral proxy re-encryption scheme, as used by NuCypher KMS [1]. Umbral is a threshold proxy re-encryption scheme following a Key Encapsulation Mechanism (KEM) approach. It is inspired by ECIES-KEM [2], and the BBS98 proxy re-encryption scheme [3]. With Umbral, Alice (which in the generic name for data owners in NuCypher KMS) can delegate decryption rights to Bob for any ciphertext intended to her, through a re-encryption process performed by a set of N semi-trusted proxies. When at least t of these proxies (out of N) participate by performing re-encryption, Bob is able to combine these independent re-encryptions and decrypt the original message using his private key. The name "Umbral" comes from the Spanish word for "threshold", emphasizing this characteristic of the scheme, given its central role in the NuCypher KMS architecture.

1. Introduction

NuCypher KMS [1] is a decentralized key management system (KMS), encryption, and access control service. It uses proxy re-encryption to delegate decryption rights, enabling this way the private sharing of data between arbitrary numbers of participants in public consensus networks, without revealing data keys to intermediary entities.

Umbral is a threshold proxy re-encryption scheme loosely inspired by ECIES-KEM [2] (since the Umbral KEM is constructed similarly as in ECIES) and the BBS98 proxy re-encryption scheme [3], although with several improvements to make it non-interactive, unidirectional, and most importantly, verifiable with respect to re-encryption. Finally, the threshold functionality of Umbral reuses ideas from Shamir's Secret Sharing [4], although applied to the context of proxy re-encryption.

We provide a reference implementation in [**?**], instantiated over an elliptic curve group.

2. Preliminaries

2.1. **Notation.** Although the additive notation is the norm when dealing with elliptic curve cryptography, in this document we adopt the multiplicative notation to express the operations in the elliptic curve group, which is the usual approach in the proxy re-encryption literature (where schemes are usually defined in generic groups).

2.2. **A brief introduction to Proxy Re-Encryption.** (TODO: General description of proxy re-encryption, properties, etc)

Proxy re-encryption is a special type of public-key encryption that permits a proxy to transform ciphertexts from one public key to another, without the proxy being able to learn any information about the original message; to do so, the proxy must be in possession of a *re-encryption*

E-mail address: dnunez@lcc.uma.es.

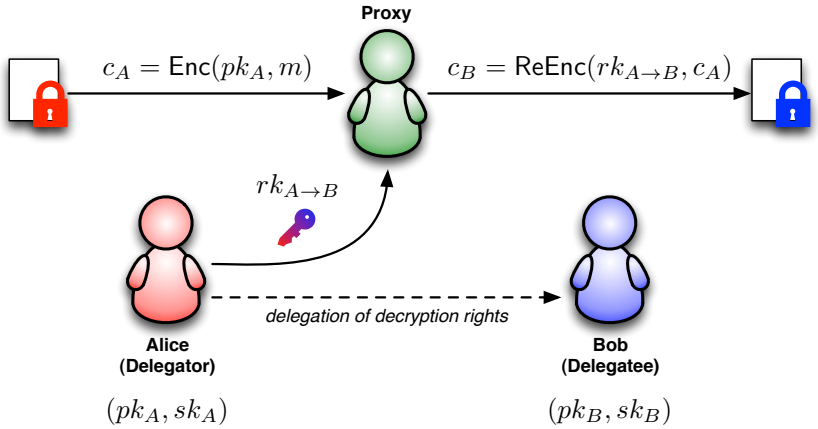

FIGURE 1. Main actors and interactions in a PRE environment

key that enables this process [**?**]. Thus, it serves as a means for delegating decryption rights, opening up many possible applications that require of delegated access to encrypted data. In the PRE literature, the parties involved are usually labeled in terms of a relationship of delegation, namely:

- Delegator: This actor is the one that *delegates* his decryption rights using proxy re-encryption. In order to do this he creates a re-encryption key, which he sends to the proxy. We usually refer to the delegator as "Alice".
- Delegatee: The delegatee is granted a delegated right to decrypt ciphertexts that, although were not intended for him in the first place, where re-encrypted for him with permission from the original recipient (i.e., the delegator). This actor usually takes the name "Bob".
- Proxy: It handles the re-encryption process that transforms ciphertexts under the delegator's public key into ciphertexts that the delegatee can decrypt using his private key. The proxy uses the re-encryption key during this process, and does not learn any additional information.

Figure 1 depicts the main actors in a PRE environment and their interactions. Since PRE is a special type of PKE, users also have a pair of public and private keys, as shown in the figure. Hence, anyone that knows a public key is capable of producing ciphertexts intended for the corresponding recipient; conversely, these ciphertexts can only be decrypted using the corresponding decryption key. The distinctive aspect is that ciphertexts can be re-encrypted in order to be decrypted by a different private key than the one originally intended.

3. The Umbral PRE cryptosystem

In this section we present the Umbral PRE cryptosystem. However, since Umbral is designed following the KEM/DEM approach, our focus will be in the Umbral KEM, since the DEM part is not affected by the "re-encryption" process. Note that when referring to "re-encryption" we are actually dealing with the transformation of the KEM ciphertexts (or "capsules"), so technically, it appears it is more appropriate to call this process "re-encapsulation". This would lead to the natural sequence of encapsulation/re-encapsulation/decapsulation, as shown in Figure 2. When possible we will use the term "re-encapsulation", although we will continue to use "re-encryption" in some contexts such as "re-encryption keys", since in the end Umbral KEM will be used as part of a full-fledged proxy re-encryption scheme.

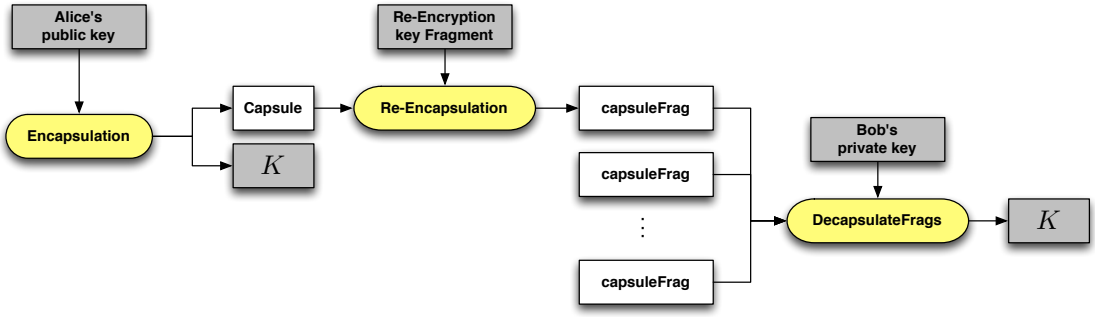

FIGURE 2. Main operation of Umbral KEM. Operations are shown in yellow, cryptographic keys in gray, and data in white

In this section we will first describe the syntax of the Umbral KEM; next, we present its construction; and finally, its integration with a DEM (i.e., a symmetric encryption algorithm) to produce the Umbral proxy re-encryption scheme.

3.1. **Syntax of the Umbral KEM.** The following is a description of the basic functions provided by Umbral KEM. For clarity we have categorized these functions in different groups according to their functionality.

3.1.1. *Key Generation Algorithms.*

- KeyGen(): The key generation algorithm KeyGen outputs a pair of public and secret keys (pk_A, sk_A).
- ReKeyGen(sk_A, pk_B, N, t): On input the secret key $sk_A = a$, the public key of the intended delegatee $pk_B = g^b$, a number of fragments N, and a threshold t, the re-encryption key generation algorithm ReKeyGen computes N fragments of the re-encryption key between A and B, each of them named $kFrag$.

3.1.2. *Encapsulation and Decapsulation.*

- Encapsulate(pk_A): On input the public key pk_A, the encapsulation algorithm Encapsulate a symmetric key K and a *capsule* that allows to derive again (i.e., "decapsulate") the symmetric key K.
- Decapsulate$(sk_A, capsule)$: On input the secret key sk_A, and an original *capsule*, the decapsulation algorithm Decapsulate outputs the symmetric key K, or $\perp$ if the capsule is invalid.
- DecapsulateFrags$(sk_B, \{cFrag_i\}_{i=1}^{t})$: On input the secret key sk_B, and a set of t capsule fragments or $cFrags$, the algorithm outputs the symmetric key K, or $\perp$ if the decryption fails.

3.1.3. *Re-Encapsulation.*

- ReEncapsulation$(kFrag, capsule)$: On input a re-encryption key fragment $kFrag$, and a *capsule*, the re-encapsulation algorithm ReEncapsulation outputs the capsule fragment $cFrag$, or $\perp$ if the the process fails.

3.2. **The Umbral KEM construction.**

3.2.1. *Setup and public parameters.*

- $\mathsf{Setup}(sec)$: The setup algorithm first determines a cyclic group $\mathbb{G}$ of prime order q, according to the security parameter *sec*. Let $g, U \in \mathbb{G}$ be generators. Let $H_2 : \mathbb{G}^2 \to \mathbb{Z}_q$, $H_3 : \mathbb{G}^3 \to \mathbb{Z}_q$, and $H_4 : \mathbb{G}^3 \times Z_q \to \mathbb{Z}_q$ be hash functions that behave as random oracles. Let $\mathsf{KDF} : \mathbb{G} \to \{0,1\}^\ell$ be a key derivation function also modeled as a random oracle, where ℓ is according to the security parameter *sec*. The global public parameters are represented by the tuple:

$$params = (\mathbb{G}, g, U, H_2, H_3, H_4, \mathsf{KDF})$$

For simplicity, we will omit the public parameters from the rest of the functions.

3.2.2. *Key Generation Algorithms.*

- $\mathsf{KeyGen}()$: Sample $a \in \mathbb{Z}_q$ uniformly at random, compute g^a and output the keypair $(pk, sk) = (g^a, a)$.
- $\mathsf{ReKeyGen}(sk_A, pk_B, N, t)$: On input the secret key $sk_A = a$, the public key of the intended delegatee $pk_B = g^b$, a number of fragments N, and a threshold t, the re-encryption key generation algorithm $\mathsf{ReKeyGen}$ computes N fragments of the re-encryption key between A and B as follows:
  (1) Sample random $x_A \in \mathbb{Z}_q$ and compute $X_A = g^{x_A}$
  (2) Compute $d = H_3(X_A, pk_B, (pk_B)^{x_A})$. Note how d is the result of a non-interactive Diffie-Hellman key exchange between B's keypair and the ephemeral key pair (x_A, X_A). We will use this shared secret to make the re-encryption key generation of the scheme non-interactive.
  (3) Sample random $t-1$ elements $f_i \in \mathbb{Z}_q$, with $1 \leq i \leq t-1$, and compute $f_0 = a \cdot b^{-1} \bmod q$.
  (4) Construct a polynomial $f(x) \in \mathbb{Z}_q[x]$ of degree $t-1$, such that $f(x) = f_0 + f_1 x + f_2 x^2 + ... + f_{t-1} x^{t-1}$.
  (5) Initialize set $KF = \emptyset$ and repeat N times:
    (a) Sample random $y, id \in \mathbb{Z}_q$ and compute $Y = g^y$ and $rk = f(id)$
    (b) Compute $U_1 = U^{rk}$
    (c) Compute $z_1 = H_4(X_A, U_1, Y, id)$, and $z_2 = y - a \cdot z_1$
    (d) Define a re-encryption key fragment $kFrag$ as the tuple $(id, rk, X_A, U_1, z_1, z_2)$
    (e) $KF = KF \cup \{kFrag\}$
  (6) Finally, output the set of re-encryption key fragments KF.

3.2.3. *Encapsulation and Decapsulation.*

- $\mathsf{Encapsulate}(pk_A)$: On input the public key pk_A, the encapsulation algorithm $\mathsf{Encapsulate}$ first samples random $r, u \in \mathbb{Z}_q$ and computes $E = g^r$ and $V = g^u$. Next, it computes the value $s = u + r \cdot H_2(E, V)$. The derived key is computed as $K = \mathsf{KDF}((pk_A)^{r+u})$. The tuple (E, V, s) is called *capsule* and allows to derive again (i.e., "decapsulate") the symmetric key K. Finally, the encapsulation algorithm outputs $(K, capsule)$.
- $\mathsf{CheckCapsule}(capsule)$: On input a $capsule = (E, V, s)$, this algorithm examines the validity of the capsule by checking if the following equation holds:

$$g^s \stackrel{?}{=} V \cdot E^{H_2(E,V)}$$

- $\mathsf{Decapsulate}(sk_A, capsule)$: On input the secret key $sk_A = a$, and an original $capsule = (E, V, s)$, the decapsulation algorithm $\mathsf{Decapsulate}$ first checks the validity of the capsule with $\mathsf{CheckCapsule}$ and outputs $\perp$ if the check fails. Otherwise, it computes $K = \mathsf{KDF}((E \cdot V)^a)$. Finally, it outputs K.

- $\mathsf{DecapsulateFrags}(sk_B, \{cFrag_i\}_{i=1}^t)$: On input the secret key $sk_B = b$, and a set of t capsule fragments, being each of them $cFrag_i = (E_{1,i}, V_{1,i}, id_i, X_A)$. Note that the value X_A is the same for all the $cFrags$ that are produced by re-encryptions using a $kFrag$ in the set of re-encryption key fragments KF.
  (1) Let $I = \{id_i\}_{i=1}^t$. For all $id_i \in I$, compute $\lambda_{i,I} = \prod_{j=1, j\neq i}^{t} \frac{id_j}{id_j - id_i}$
  (2) Compute the values:
  $$E' = \prod_{i=1}^{t} (E_{1,i})^{\lambda_{i,I}} \qquad V' = \prod_{i=1}^{t} (V_{1,i})^{\lambda_{i,I}}$$
  (3) Compute $d = H_3(X_A, pk_B, X_A^b)$. Recall that d is the result of a non-interactive Diffie-Hellman key exchange between B's keypair and the ephemeral key pair (x_A, X_A).
  (4) Finally, output the symmetric key $K = \mathsf{KDF}((E' \cdot V')^d)$.

3.2.4. *Re-Encapsulation.*

- $\mathsf{ReEncapsulate}(kFrag, capsule)$: On input a re-encryption key fragment $kFrag = (id, rk, X_A, U_1, z_1, z_2)$, and a $capsule = (E, V, s)$, the re-encapsulation algorithm ReEncapsulate first checks the validity of the capsule with CheckCapsule and outputs $\perp$ if the check fails. Otherwise, it computes $E_1 = E^{rk}$ and $V_1 = V^{rk}$, and outputs the capsule fragment $cFrag = (E_1, V_1, id, X_A)$.

3.3. **The KEM/DEM construction.** Extending Umbral KEM with a DEM results in a full-fledged proxy re-encryption scheme. As such, this defines encryption and decryption algorithms, rather than encapsulation and decapsulations. Note how the re-encryption algorithm actually does not involve any symmetric encryption operation. We omit the key generation algorithms since they are not changed in the extension.

- $\mathsf{Encrypt}(pk_A, M)$: On input the public key pk_A and a message $M \in \mathcal{M}$, the encryption algorithm Encrypt first computes $(K, capsule) = \mathsf{Encapsulate}(pk_A)$. $encData$ is the result of applying the authenticated encryption algorithm AEnc to M with key K. Finally, it outputs the ciphertext $C = (capsule, encData)$.
- $\mathsf{Decrypt}(sk_A, C)$: On input the secret key sk_A and a ciphertext $C = (capsule, encData)$, the decryption algorithm Decrypt computes the key $K = \mathsf{Decapsulate}(sk_A, capsule)$, and decrypts ciphertext $encData$ using the decryption function of the authenticated encryption algorithm AEnc to obtain message M if decryption is correct, and $\perp$ otherwise. Finally, it outputs message M (or $\perp$ if decryption was invalid).
- $\mathsf{ReEncrypt}(kFrag, C)$: On input a re-encryption key fragment $kFrag$ and a ciphertext $C = (capsule, encData)$, the re-encryption algorithm ReEncrypt applies ReEncapsulate to the $capsule$ to obtain a $cFrag$, and outputs the re-encrypted ciphertext $C' = (cFrag, encData)$.
- $\mathsf{DecryptFrags}(sk_B, \{C'_i\}_{i=1}^t)$: On input the secret key sk_B, a set of t re-encrypted ciphertexts $C'_i = (cFrag_i, encData)$, the fragments decryption algorithm DecryptFrags first decapsulates the $cFrags$ with $\mathsf{DecapsulateFrags}(sk_B, \{cFrag_i\}_{i=1}^t)$ to produce key K, and decrypts ciphertext $encData$ using the decryption function of the authenticated encryption algorithm AEnc to obtain message M if decryption is correct, and $\perp$ otherwise. Finally, it outputs message M (or $\perp$ if decryption was invalid). Note that the symmetric ciphertext $encData$ is the same for all the C'_i that are re-encryptions of the same ciphertext C.

3.4. **Parameters of Umbral instantiation in NuCypher KMS.** The only restriction that the Umbral cryptosystem imposes on the choice of EC curve is that it should generate a group of prime order, since we need to compute inverses modulo the order of this group. In our current

setting, we use the secp256k1 curve, since it fulfills this latter requirement; we are exploring other curve choices that could improve performance.

As per the authenticated encryption scheme, we use PyNaCl's SecretBox implementation, which in turn uses Salsa20-Poly1305.

For the KDF, we use HKDF with SHA-512 as hash function.

References

[1] Michael Egorov, MacLane Wilkison, and David Nuñez. Nucypher KMS: decentralized key management system. *CoRR*, abs/1707.06140, 2017.

[2] American National Standards Institute (ANSI) X9.F1 subcommittee. ANSI X9.63 Public key cryptography for the Financial Services Industry: Elliptic curve key agreement and key transport schemes, July 5, 1998. Working draft version 2.0.

[3] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. *Advances in Cryptology—EUROCRYPT'98*, pages 127–144, 1998.

[4] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[5] Victor Shoup. ISO 18033-2: An emerging standard for public-key encryption. `http://shoup.net/iso/std6.pdf`, December 2004. Final Committee Draft.

[6] IEEE P1363a Committee. IEEE P1363a / D9 — standard specifications for public key cryptography: Additional techniques. `http://grouper.ieee.org/groups/1363/index.html/`, June 2001. Draft Version 9.

[7] V Gayoso Martínez, L Hernández Encinas, and A Queiruga Dios. Security and practical considerations when implementing the elliptic curve integrated encryption scheme. *Cryptologia*, 39(3):244–269, 2015.

[8] R. Canetti and S. Hohenberger. Chosen-ciphertext secure proxy re-encryption. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 185–194. ACM, 2007.

[9] Tink. `https://github.com/google/tink`, 2017.