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. **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

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

information about the original message [?]. Thus, it serves as a means for delegating decryption rights, opening up many possible applications that require of delegated access to encrypted data.

3. The Umbral PRE cryptosystem

In this section we present the Umbral PRE cryptosystem, defined by the following algorithms:

3.1. **Syntax.**

3.2. **The KEM construction.**

3.2.1. *Key Generation Algorithms.*

- KeyGen(): Sample $a \in \mathbb{Z}_q$ uniformly at random, compute g^a and output the keypair $(pk, sk) = (g^a, a)$.
- 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 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.

3.2.2. *Encapsulation and Decapsulation.*

- Encapsulate(pk_A): On input the public key pk_A, the encapsulation algorithm 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 $(K, capsule)$.
- CheckCapsule(*capsule*): On input a $capsule = (E, V, s)$, this algorithm examines the validity of the capsule by checking if the following equation holds:

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

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

- $\mathsf{Decapsulate}'(sk_B, capsule')$: On input the secret key $sk_B = b$, and a reconstructed capsule $capsule' = (E', V', X_A)$, the decapsulation algorithm $\mathsf{Decapsulate}'$ first computes $d = H_3(X_A, pk_B, X_A^b)$. 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). Next, it computes $K = \mathsf{KDF}((E' \cdot V')^d)$. Finally, it outputs K.

3.2.3. *Re-Encryption and Verification.*

- $\mathsf{CapsuleReenc}(kFrag, capsule)$: 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 capsule re-encryption algorithm $\mathsf{CapsuleReenc}$ first checks the validity of the capsule with $\mathsf{CheckCapsule}$ and outputs $\perp$ if the check fails. Otherwise, it computes $E_1 = E^{rk}$ and $V_1 = V^{rk}$, and outputs the capsule fragment $cFrag = (E_1, V_1, id, X_A)$.
- $\mathsf{Reconstruct}(\{cFrag_i\}_{i=1}^t)$: On input a set of t capsule fragments, each of them labeled as $cFrag_i$, the reconstruction algorithm $\mathsf{Reconstruct}$, first parses each fragment $cFrag_i$ as the tuple $(E_{1,i}, V_{1,i}, id_i, X_A)$. Let $I = \{id_i\}_{i=1}^t$. Next, it computes the values E' and V' as follows:

$$E' = \prod_{i=1}^{t} (E_{1,i})^{\lambda_{i,I}}, \qquad V' = \prod_{i=1}^{t} (V_{1,i})^{\lambda_{i,I}}, \qquad \text{where } \lambda_{i,I} = \prod_{j=1, j\neq i}^{t} \frac{id_j}{id_j - id_i}$$

  Finally, it outputs the reconstructed capsule (E', V', X_A).

3.3. **The KEM/DEM construction.**

- $\mathsf{Encrypt}(pk_A, M)$: On input the public key pk_A and a message $M \in \mathcal{M}$, the encryption algorithm $\mathsf{Encrypt}$ 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 $\mathsf{Decrypt}$ 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).
- $\mathsf{DecryptFrags}(sk_B, \{cFrag_i\}_{i=1}^t, encData)$: On input the secret key sk_B, a set of t capsule fragments, each of them labeled as $cFrag_i$, and the encrypted data $encData$, the fragments decryption algorithm $\mathsf{DecryptFrags}$ first computes

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.