UMBRAL: A THRESHOLD PROXY RE-ENCRYPTION SCHEME

DAVID NUÑEZ

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

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 m 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 decentralized architecture of NuCypher KMS.

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 Python called pyUmbral [5], instantiated over elliptic curve secp256k1.

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).

E-mail address: david@nucypher.com.

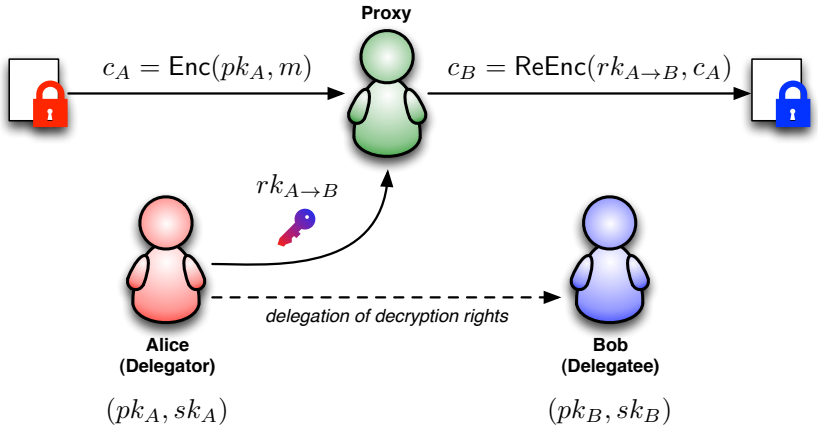

FIGURE 1. Main actors and interactions in a PRE environment

2.2. **A brief introduction to Proxy Re-Encryption.** 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 key* that enables this process [6]. 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.

This definition is oblivious to the specific properties of PRE schemes [**?**]

Directionality: A PRE scheme is *unidirectional* if the re-encryption keys enable the transformation of ciphertexts only in one direction, from delegator to delegatee, and is *bidirectional* otherwise.

Number of hops: We say a PRE scheme is *single-hop* (or *single-use*) if a re-encrypted ciphertext cannot be re-encrypted again, while it is *multi-hop* (or *multu-use*) if ciphertexts are re-encryptable multiple times.

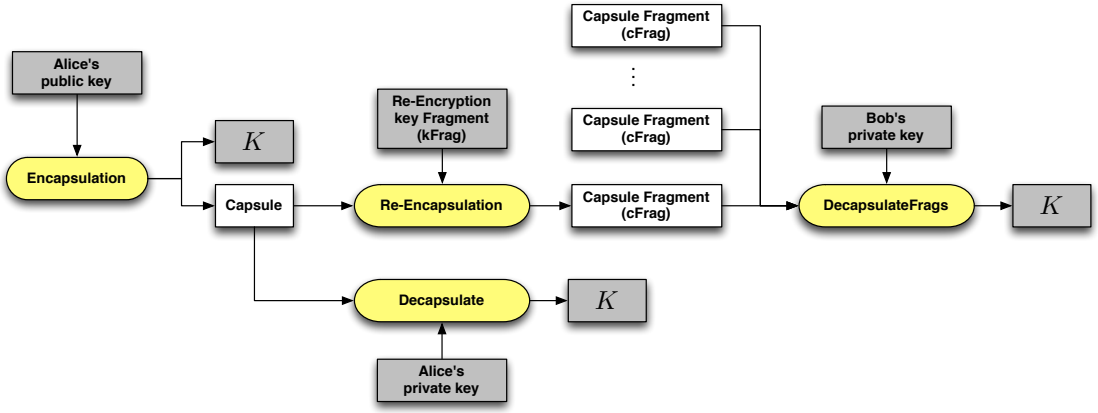

FIGURE 2. Main operation of Umbral KEM. Operations are shown in yellow, cryptographic keys in gray, and data in white

Interactivity: If the secret key of the delegatee is not needed in the re-encryption key generation process, then the scheme is *not interactive* (i.e., since he does not have to participate in the process). Otherwise, we say it is *interactive*.

See [6, Section 3.3] for a more detailed description of these and other PRE properties.

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.

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 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.*

- $\mathsf{KeyGen}()$: The key generation algorithm KeyGen outputs a pair of public and secret keys (pk_A, sk_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 m, the re-encryption key generation algorithm $\mathsf{ReKeyGen}$ computes N fragments of the re-encryption key between A and B, each of them named $kFrag$.

3.1.2. *Encapsulation and Decapsulation.*

- $\mathsf{Encapsulate}(pk_A)$: On input the public key pk_A, the encapsulation algorithm $\mathsf{Encapsulate}$ a symmetric key K and a *capsule* that allows to derive again (i.e., "decapsulate") the symmetric key K.
- $\mathsf{Decapsulate}(sk_A, capsule)$: On input the secret key sk_A, and an original *capsule*, the decapsulation algorithm $\mathsf{Decapsulate}$ outputs the symmetric key K, or $\perp$ if the capsule is invalid.

3.1.3. *Re-Encapsulation and Fragments Decapsulation.*

- $\mathsf{ReEncapsulation}(kFrag, capsule)$: On input a re-encryption key fragment $kFrag$, and a *capsule*, the re-encapsulation algorithm $\mathsf{ReEncapsulation}$ outputs the capsule fragment $cFrag$, or $\perp$ if the the process fails.
- $\mathsf{DecapsulateFrags}(sk_B, \{cFrag_i\}_{i=1}^{m}, capsule)$: On input the secret key sk_B, and a set of m capsule fragments or $cFrags$, the fragments decapsulation algorithm outputs the symmetric key K, or $\perp$ if the decryption 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 $\hat{H} : \mathbb{G}^2 \to \mathbb{Z}_q$, $H_3 : \mathbb{G}^3 \to \mathbb{Z}_q$, 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:

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

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 $(pk, sk) = (g^a, a)$.
- $\mathsf{ReKeyGen}(sk_A, pk_B, N, m)$: 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 m, 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 $e_{ni}, e_x \in \mathbb{Z}_q^*$ and compute $P_{ni} = g^{e_{ni}}$ and $P_x = g^{e_x}$.
  (2) Compute $d = \hat{H}(P_{ni}, pk_B, (pk_B)^{e_{ni}})$. Note how d is the result of a non-interactive Diffie-Hellman key exchange between B's keypair and the ephemeral key pair (e_{ni}, P_{ni}). We will use this shared secret d to make the re-encryption key generation process non-interactive.
  (3) Sample random $m-1$ elements $f_i \in \mathbb{Z}_q^*$, with $1 \leq i \leq m-1$, and compute $f_0 = a \cdot d^{-1} \bmod q$.
  (4) Construct a polynomial $f(x) \in \mathbb{Z}_q[x]$ of degree $m-1$, such that $f(x) = f_0 + f_1 x + f_2 x^2 + ... + f_{m-1} x^{m-1}$.
  (5) Compute $D = \hat{H}(P_x, pk_B, (pk_B)^{e_x})$
  (6) Let $\ell = \lceil \log_2(q) \rceil$. Initialize set $KF = \emptyset$ and repeat N times:
    (a) Sample random $y \in \mathbb{Z}_q^*$ and $id \in \{0,1\}^\ell$.
    (b) Compute $x = \hat{H}(id, D)$ and $Y = g^y$.
    (c) Compute $rk = f(x)$

(d) Compute $U_1 = U^{rk}$.
(e) Compute $z_1 = \hat{H}(Y, id, pk_A, pk_B, U_1, P_{ni}, P_x)$, and $z_2 = y - a \cdot z_1$.
(f) Define a re-encryption key fragment as:

$$kFrag = (id, rk, U_1, P_{ni}, P_x, z_1, z_2)$$

(g) Set $KF = KF \cup \{kFrag\}$

(7) Finally, output the set of re-encryption key fragments KF.

3.2.3. *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 $s = u + r \cdot \hat{H}(E, V)$. 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:

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

- 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.

3.2.4. *Re-Encapsulation and Fragments Decapsulation.*

- ReEncapsulate($kFrag, capsule$): On input a re-encryption key fragment $kFrag = (id, rk, U_1, P_{ni}, P_x, z_1, z_2)$, 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 $cFrag = (E_1, V_1, id, P_{ni}, P_x)$.
- DecapsulateFrags($sk_B, pk_A, \{cFrag_i\}_{i=1}^m$): On input the secret key $sk_B = b$, the original public key $pk_A = g^a$, and a set of m capsule fragments, being each of them $cFrag_i = (E_{1,i}, V_{1,i}, id_i, X_A)$, the fragments decapsulation algorithm DecapsulateFrag does the following:
  (1) Compute $D = \hat{H}(P_x, pk_B, (P_x)^b)$
  (2) Let $\mathcal{S} = \{x_i\}_{i=1}^m$, for $x_i = H_5(id_i, D)$. For all $x_i \in \mathcal{S}$, compute:

$$\lambda_{i,\mathcal{S}} = \prod_{j=1, j\neq i}^{m} \frac{x_j}{x_j - x_i}$$

  (3) Compute the values:

$$E' = \prod_{i=1}^{m} (E_{1,i})^{\lambda_{i,\mathcal{S}}} \qquad V' = \prod_{i=1}^{m} (V_{1,i})^{\lambda_{i,\mathcal{S}}}$$

  (4) Compute $d = \hat{H}(P_{ni}, pk_B, (P_{ni})^b)$. Recall that d is the result of a non-interactive Diffie-Hellman key exchange between B's keypair and the ephemeral key pair (e_{ni}, P_{ni}). Note also that the value P_{ni} 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.
  (5) Finally, output the symmetric key $K = \mathsf{KDF}((E' \cdot V')^d)$.

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. We require the DEM to be an authenticated encryption with associated data algorithm, which we will denote as AEAD. Note also 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 AEAD to M with key K, with $capsule$ as associated data. 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 AEAD with key K and $capsule$ as associated data, which results in 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 $C' = (cFrag, encData)$.
- $\mathsf{DecryptFrags}(sk_B, \{C'_i\}_{i=1}^m)$: On input the secret key sk_B, a set of m re-encrypted ciphertexts $C'_i = (cFrag_i, encData)$, the fragments decryption algorithm DecryptFrags first decapsulates the $cFrags$ with $\mathsf{DecapsulateFrags}(sk_B, \{cFrag_i\}_{i=1}^m)$ to produce key K, and decrypts ciphertext $encData$ using the decryption function of AEAD with key K and $capsule$ as associated data, which results in 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.

4. Providing Proofs of Re-Encryption Correctness

To prove correctness of re-encryption, the proxy uses a non-interactive zero-knowledge proof of discrete logarithm equality that shows that both E_1 and V_1 are exponentiations of, respectively, E and V for the same exponent, and that this exponent is the same used in U_1 with respect to U. Since U is a public parameter of the system, and U_1 is signed by Alice and attached to the proof, then this proofs that $E_1 = E^{rk}$ and $V_1 = V^{rk}$, given that $U_1 = U^{rk}$. In other words, let $dlog_B(X)$ be the discrete logarithm of $X \in \mathbb{G}$ with respect to element $B \in \mathbb{G}$; the re-encryption correctness proof shows that $dlog_E(E_1) = dlog_V(V_1) = dlog_U(U_1)$.

4.1. **Producing proofs of re-encryption correctness.** This extends the ReEncapsulate algorithm to include a proof of correctness in the resulting $cFrag$. We also support the addition of an optional arbitrary input aux to be added to the proof, and which can be used as metadata for the re-encryption request (e.g., in the NuCypher KMS this would include the proxy identifier, a timestamp, etc.).

Let the re-encryption key fragment be $kFrag = (id, rk, U_1, P_{ni}, P_x, z_1, z_2)$, and the input $capsule = (E, V, s)$. The resulting capsule fragment is $cFrag = (E_1, V_1, id, P_{ni}, P_x)$. A correctness proof π for $cFrag$ is generated as follows:

(1) Sample random $\tau \in \mathbb{Z}_q^*$
(2) Compute the values: $E_2 = E^\tau \qquad V_2 = V^\tau \qquad U_2 = U^\tau$
(3) Compute the hash value $h = \hat{H}(E, E_1, E_2, V, V_1, V_2, U, U_1, U_2, aux)$
(4) Compute $\rho = \tau + h \cdot rk$
(5) Output the proof $\pi = (E_2, V_2, U_2, U_1, z_1, z_2, \rho, aux)$

The result of the extended ReEncapsulate algorithm with correctness guarantees is the tuple $(cFrag, \pi)$.

4.2. **Verifying proofs of re-encryption correctness.** This extends the DecapsulateFrags algorithm to check the attached proof for each $cFrag$.

Let the input *capsule* be the tuple (E, V, s). For each $cFrag = (E_1, V_1, id, P_{ni}, P_x)$ and proof $\pi = (E_2, V_2, U_2, U_1, z_1, z_2, \rho, aux)$:

(1) Check that the signature (z_1, z_2) of $kFrag$ is correct:

$$z_1 \stackrel{?}{=} \hat{H}(g^{z_2} \cdot (pk_A)^{z_1}, id, pk_A, pk_B, U_1, P_{ni}, P_x)$$

(2) Compute the hash value $h = \hat{H}(E, E_1, E_2, V, V_1, V_2, U, U_1, U_2, aux)$

(3) Check that the following equations hold:

$$E^{\rho} \stackrel{?}{=} E_2 \cdot E_1^h$$

$$V^{\rho} \stackrel{?}{=} V_2 \cdot V_1^h$$

$$U^{\rho} \stackrel{?}{=} U_2 \cdot U_1^h$$

5. Notes concerning Umbral reference implementation (pyUmbral)

5.1. **Choice of elliptic curve.** 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 and it is widely used in the blockchain ecosystem; we are exploring other curve choices that could improve performance.

5.2. **Hash functions.** As described in Section 3.2.1, Umbral requires several hash functions, such as $\hat{H} : \mathbb{G}^2 \to \mathbb{Z}_q$, $H_3 : \mathbb{G}^3 \to \mathbb{Z}_q$, and $H_4 : \mathbb{G}^3 \times Z_q \to \mathbb{Z}_q$, which behave as random oracles that output an element of $\mathbb{Z}_q$. Since for elements of both $\mathbb{G}$ and $\mathbb{Z}_q$ there exists efficient encodings to bit string (e.g., compressed representation in the case EC points), in pyUmbral we implement these hash functions from a common hash function $H' : \{0,1\}^* \to \mathbb{Z}_q$. For example, for the case of $\hat{H}$, we can define it as $\hat{H}(g_1, g_2) = H'(\mathsf{encode}(g_1)||\mathsf{encode}(g_2))$.

This reduces the problem to the definition of H'. Let n be the order of the group induced by the curve in use, X a byte string of arbitrary size, and digest a cryptographic hash function with a digest size $> 64 + \log_2(n)$ (e.g., for a 256-bit curve, we use BLAKE2b with a digest size of 512 bits). The output h_X of $H'(X)$ is computed as:

$$h_X \Leftarrow 1 + \mathsf{int}(\mathsf{digest}(X)) \bmod (n-1)$$

In pyUmbral, this is implemented by the class method `CurveBN.hash()`. For reference, in the case of curve secp256k1 and BLAKE2b with a digest size of 512-bits:

$X =$ `616263` (ASCII string "*abc*")

$n =$ `fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141`

$\mathsf{digest}(X) =$ `ba80a53f981c4d0d6a2797b69f12f6e94c212f14685ac4b74b12bb6fdbffa2d1`
`7d87c5392aab792dc252d5de4533cc9518d38aa8dbf1925ab92386edd4009923`

$h_X =$ `63b4973dd623699fe9b344da6ddd77fa5dd60413a21f6ad810186602d05dd1a4`

5.3. **Symmetric encryption.** As per the authenticated encryption scheme, we use Chacha20-Poly1305 as provided by cryptography.io [**?**].

5.4. **Key derivation.** For the KDF, we use HKDF with BLAKE2b as hash function, with a digest size of 64 bytes.

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] NuCypher. pyumbral. `https://github.com/nucypher/pyumbral`, 2018.

[6] David Nuñez, Isaac Agudo, and Javier Lopez. Proxy re-encryption: Analysis of constructions and its application to secure access delegation. *Journal of Network and Computer Applications*, 87:193–209, 2017.