

# [Draft 2] NuCypher KMS: Decentralized key management system

Michael Egorov\* and MacLane Wilkison†  
*NuCypher*

David Núñez‡  
*NICS Lab, Universidad de Málaga, Spain*

(Dated: August 6, 2017)

NuCypher KMS is a decentralized Key Management System (KMS) that addresses the limitations of using consensus networks to securely store and manipulate private, encrypted data [1]. It provides encryption and cryptographic access control, performed by a decentralized network, leveraging proxy re-encryption [2]. Unlike centralized KMS as a service solutions, it doesn't require trusting a service provider. NuCypher KMS enables sharing of sensitive data for both decentralized and centralized applications, providing security infrastructure for applications from healthcare to identity management to decentralized content marketplaces. NuCypher KMS will be an essential part of decentralized applications, just as SSL/TLS is an essential part of every secure web application.

---

\* [michael@nucypher.com](mailto:michael@nucypher.com)

† [maclane@nucypher.com](mailto:maclane@nucypher.com)

‡ [dnunez@lcc.uma.es](mailto:dnunez@lcc.uma.es)

## CONTENTS

I. Introduction	3
A. Background	3
II. Architecture	4
A. Cryptographic primitives	4
1. Symmetric encryption	4
2. Public-key encryption	4
3. Proxy re-encryption	4
B. A brief review of re-encryption schemes	6
C. Signing encrypted messages	7
D. Re-encryption nodes	7
III. Network security	9
A. Pseudo-anonymity	10
B. Split-key re-encryption	10
C. Challenge protocol	10
D. Relevancy of possible threats to different use cases	11
E. Hardware-enforced security	12
IV. Performance considerations	12
V. Functionality	12
A. Local encryption library and daemon	12
B. Sharing short secrets	13
C. Sharing files and hierarchical data	13
D. Sharing encrypted streams	13
E. Time-based and condition-based policies	14
F. Key rotation	14
VI. Token economics	14
A. Token distribution	15
VII. Use cases	15
A. Sharing encrypted files (“Decentralized Dropbox”)	15
B. End-to-end encrypted group chat (“Encrypted Slack”)	15
C. Patient-controlled electronic health records (EHR)	16
D. Decentralized digital rights management (DDRM)	16
E. Blind identity management	16
F. Secret credentials management for scripts and backend applications	16
G. Shared credentials and enterprise password management	16
H. Mandatory access logging	16
I. Mobile device management (MDM) and revocation	17

VIII. Summary	17
IX. Acknowledgements	17
References	17

## I. INTRODUCTION

NuCypher KMS is a decentralized key management system (KMS), encryption, and access control service. It enables private data sharing between arbitrary numbers of participants in public consensus networks, using proxy re-encryption to delegate decryption rights in a way that cannot be achieved by traditional symmetric or public-key encryption schemes. Native tokens are used to incentivize network participants to perform key management and access delegation/revocation operations.

### A. Background

A key management system (KMS) is an integrated approach for generating, distributing, and managing cryptographic keys for devices and applications. A KMS includes the backend functionality for key generation, distribution, and rotation as well as the client functionality for injecting, storing, and managing keys on devices [3].

As the root of trust, it's critical that a KMS is appropriately configured, managed, and protected. Historically, this has meant deploying a KMS on-premises in hardware security modules (HSM) [4] or using tools like HashiCorp's Vault [5]. However, this requires a high degree of technical sophistication as well as upfront capital investment. To ease the technical burden and provide more competitive pricing, vendors like Amazon CloudHSM [6], Google Cloud KMS [7], Azure Key Vault [8] and TrueVault [9] have begun offering KMS as a service. However, KMS as a service offerings necessitate placing an undue level of trust in the service provider, which may be inappropriate for security-critical applications.

Public consensus networks, such as Bitcoin and Ethereum, are a promising solution to this centralization problem. But the limitations of public consensus networks in performing cryptographic operations that involve the manipulation of secret data are well-established [1]. Consensus networks employ a volunteer network of nodes, which is subject to constant churn and not as reliable as central infrastructure when it comes to availability and enforcing access management rules.

NuCypher KMS uses a decentralized network to remove the reliance on central service providers, proxy re-encryption for cryptographic access controls, and a token incentive mechanism to ensure reliability, availability, and correctness.

## II. ARCHITECTURE

### A. Cryptographic primitives

#### 1. Symmetric encryption

Symmetric or secret key encryption requires users to know a common secret key. For convenience, we refer to this common secret key as DEK (data encryption key).

Two operations are defined for symmetric ciphers:

$$c = \text{encrypt}_{sym}(dek, d); \quad (1)$$

$$d = \text{decrypt}_{sym}(dek, c); \quad (2)$$

where  $d$  is plaintext data, and  $c$  is ciphertext (encrypted data).

The most useful symmetric key encryption algorithms for our purposes are AES (because it's normally hardware-accelerated) [10] and Salsa20 [11].

Symmetric block ciphers can operate in different modes. We use modes of operation that yield probabilistic encryption (such as GCM for AES), to guarantee strong semantic security. For simplicity, we omit details about the particular modes of operations, treating the value of *nonce* related to the mode of operation as part of the ciphertext  $c$ .

#### 2. Public-key encryption

Public-key encryption (PKE) is a type of encryption where two parties (a sender and a receiver) exchange information without any required common secret. Every participant has a key pair (a public key  $pk$  and a secret/private key  $sk$ ). If the sender has a key pair  $sk_s/pk_s$  and the receiver has a key pair  $sk_r/pk_r$ , the sender can encrypt a message with the receiver's public key, and the receiver can decrypt with his secret key.

Hybrid cryptosystems can be created that combine the efficiency of symmetric encryption with the convenience of PKE. A hybrid cryptosystem encryption flow defines the functions:

$$dek = \text{random}(); \quad (3)$$

$$c = \text{encrypt}_{sym}(dek, d); \quad (4)$$

$$edek = \text{encrypt}_{pke}(dek, pk_r). \quad (5)$$

The pair  $(edek, c)$  is used to transform the encrypted data. The associated decryption by the receiver defines the functions:

$$dek = \text{decrypt}_{pke}(edek, sk_r); \quad (6)$$

$$d = \text{decrypt}_{sym}(dek, c). \quad (7)$$

#### 3. Proxy re-encryption

Proxy re-encryption (PRE) [2, 12] is a type of public-key encryption (PKE) that allows a proxy entity to transform ciphertexts from one public key to another, without learning anything about the underlying message (Fig. 1).

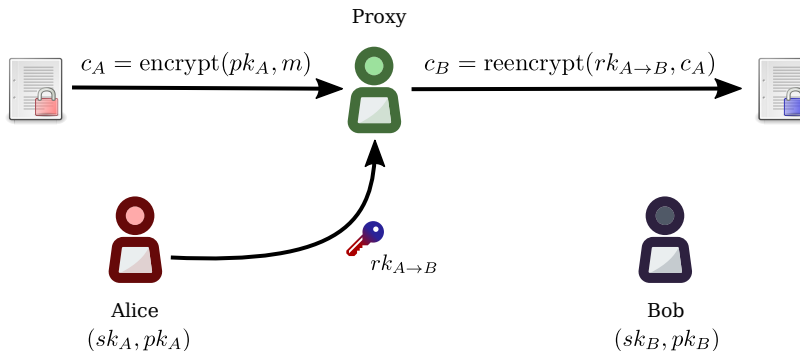


FIG. 1: Main actors and interactions in a PRE environment

In a typical proxy re-encryption scenario, there are several actors. Alice, the data owner, has public key  $pk_A$ . Any entity that knows this key can produce encrypted data that only her can decrypt, using her secret key  $sk_A$ . Suppose that a data producer encrypts a message  $m$ , with Alice’s public key  $pk_A$ , resulting in ciphertext  $c_A$ . Alice decides to delegate access to message  $m$  to Bob, who has the key pair  $(pk_B, sk_B)$ . To do so, Alice creates a re-encryption key:

$$rk_{A \rightarrow B} = \text{rekey}(sk_A, pk_B). \quad (8)$$

Importantly, in single-use, unidirectional proxy re-encryption schemes, this re-encryption function is one way, and  $rk_{A \rightarrow B}$  cannot be decomposed into its component parts (at least, without also knowing  $sk_A$  or  $sk_B$ ). All it can do is re-encrypt  $c_A$  such that it is transformed into  $c_B$ :

$$c_B = \text{reencrypt}(rk_{A \rightarrow B}, c_A). \quad (9)$$

Bob can then decrypt  $c_B$  using his secret key  $sk_B$ .

Compared to existing PKE protocols which are ideal for 1-to-1 communication, PRE is more scalable for N-to-N communication with arbitrary numbers of data producers and consumers. It doesn’t require knowing the recipient of a message in advance, as the re-encryption token can be created and applied at any point. This makes it well-suited for distributed systems such as blockchain, IoT, and big data [13].

There are many proxy re-encryption algorithms with different properties. For the first version of NuCypher KMS we choose the most simple (and performant) one, derived from BBS98 [14]. However, sometimes we want to delegate re-encryption to multiple nodes, in order to split the trust between them and apply time-based or conditional re-encryption policies. In this case we use the AFGH scheme [15]. Quantum-resistant NTRU may also be used if the need arises [16, 17].

The BBS98 algorithm creates a re-encryption key out of two secret keys, rather than a secret+public key. However, we don’t want the sender to know the receiver’s secret key. To get around this problem, we randomly generate an ephemeral key pair  $sk_e/pk_e$ . Then, access delegation looks like:

$$sk_e = \text{random}(); \quad (10)$$

$$rk_{A \rightarrow B} = \text{rekey}(sk_A, sk_e); \quad (11)$$

$$sk'_e = \text{encrypt}_{pk_e}(pk_B, sk_e). \quad (12)$$

The re-encryption node also needs to tell the receiver his ephemeral key. So, in addition to  $rk_{A \rightarrow B}$ , the receiver will get  $sk'_e$ , which will be stored by a re-encryption node along with the re-encryption key. The decryption by the receiver

will then look like:

$$sk_e = \text{decrypt}_{pk_e}(sk_B, sk'_e); \quad (13)$$

$$dek = \text{decrypt}_{pk_e}(edek, sk_e); \quad (14)$$

$$d = \text{decrypt}_{sym}(dek, c). \quad (15)$$

## B. A brief review of re-encryption schemes

There are different properties which re-encryption schemes may enjoy. In this subsection, we consider several of them relevant to our use cases. A full survey with consideration of all the known proxy re-encryption algorithms and their properties was published recently [12].

One of the important properties to consider is whether an algorithm is interactive or not. “Interactive” means that a re-encryption key is computed out of two secret keys:

$$re_{ab} = \text{rekey}(sk_a, sk_b).$$

“Non-interactive” means that one needs to know only the owner’s private key and the delegatee’s public key:

$$re_{ab} = \text{rekey}(sk_a, pk_b).$$

Examples of interactive algorithms are: BBS98 [14], our ECIES re-encryption (based on BBS98), and LWE-based reencryption [18]. An example of a “non-interactive” algorithm is AFGH [15]. Even though it initially seems that we are only interested in non-interactive algorithms, we can tweak interactive algorithms to share with a public rather than a private key on a protocol level using ephemeral keys (Eqs. 10-12).

Another property which is interesting is whether the algorithm is uni-directional or bi-directional. Bi-directionality means that it is possible to compute  $re_{ba}$  from just  $re_{ab}$ . In uni-directional algorithms that is impossible. Bi-directional algorithms are effectively uni-directional when using ephemeral keys (Eqs. 10-12). Nevertheless, in Sec. VC we show that uni-directionality can be very convenient for making complex hierarchical data shareable in a scalable way. An example of a bi-directional algorithm is BBS98 [14].

Proxy re-encryption algorithms can also be single-hop or multi-hop. “Multi-hop” means that if we have  $re_{ab}$  and  $re_{bc}$ , these two re-encryption keys can be applied in series to convert ciphertext  $c_a$  to a ciphertext  $c_c$  without the participation of  $b$ :

$$c_c = \text{reencrypt}(re_{bc}, \text{reencrypt}(re_{ab}, c_a)).$$

Sometimes, it is even possible to compute  $re_{ac}$ , which for BBS98 is:

$$re_{ac} = re_{ab} \cdot re_{bc}.$$

“Single-hop” means that if  $c_b$  is obtained via re-encryption, it is impossible to re-encrypt it further. Multi-hop schemes are useful for key rotation (Sec. VF) and for our hierarchical data sharing scheme (Sec. VC). The ephemeral key trick (Eqs. 10-12) is, effectively, single-hop (although key rotation is still possible because ephemeral keys are only created when data is shared with other parties). Examples of multi-hop algorithms include BBS98 [14] and the LWE-based algorithm [18]. The AFGH algorithm [15] is single-hop.

Another seemingly important property is collusion resistance. Collusion resistance means that having just  $re_{ab}$  and  $sk_b$  it is impossible to derive  $sk_a$ . A lack of collusion-resistance makes it possible to obtain  $sk_a$  from  $re_{ab}$

and  $sk_b$ . At first glance this seems to be very much desired for security. However, even with collusion-resistant algorithms, if an attacker has both  $re_{ab}$  and  $sk_b$ , he is quite capable of decrypting data encrypted by  $sk_a$  whenever he wants. Collusion-resistance only becomes important when the same key is used for signing and for encryption, but it's good practice to use separate key pairs for both functions. However, if any other key is derived from  $sk_a$ , then a collusion-resistant algorithm should be used. For our purposes, it appears unimportant whether an algorithm is collusion-resistant because we don't derive any subkeys from keys which are used to encrypt data. Collusion-resistant algorithms include AFGH [15] and LWE-based re-encryption [18]. Non collusion-resistant algorithms are BBS98 [14] and ECIES re-encryption.

We may need to make it impossible to identify producers and/or consumers of the data from re-encryption keys (Sec. III A). Often, a proxy who knows all public keys in the system is able to deduce this information. Imagine there is an unknown re-encryption key  $re_{xy}$  which known to the proxy. If the proxy iterates through all  $a$  and  $b$  in the system, it can discover that for some pair  $(a, b)$ :

$$\text{reencrypt}(re_{xy}, \text{encrypt}(pub_a, \text{'any text'})) = \text{encrypt}(pub_b, \text{'any text'}).$$

If such a pair is found, then it is proven that  $x = a$  and  $y = b$ . Many schemes are vulnerable to this attack [14, 15]. However, the LWE-based re-encryption [18] enjoys the property of anonymity.

Finally, as for any cryptosystem, notions of CPA-security [19] (security against chosen-plaintext attacks) and CCA-security [20] (security against chosen-ciphertext attacks) are applicable to proxy re-encryption. All of the algorithms we have mentioned so far are CPA-secure. However AFGH [15], ECIES, and LWE [18] are also CCA-secure while BBS98 [14] is not.

### C. Signing encrypted messages

In public-key encryption algorithms, anyone can encrypt using  $pub_a$ . This is useful, but it also allows for malicious users of the network to encrypt data as if it were created by  $A$ . So the data has to be signed in order to prove the identity of the original data owner to the recipient.

However, we want to make it possible to anonymize the protocol (Sec. ??) because a public digital signature that authenticates the owner of the data also raises the possibility of a re-encryption node attempting to extort money from the owner. As such, it makes sense to:

- include a digital signature as part of the plain text, so that it is impossible to verify the signature until the ciphertext is decrypted,
- use different key pairs for signing and for encryption (especially considering the lack of collusion-resistance of some re-encryption cryptosystems).

### D. Re-encryption nodes

When the data is stored in a cloud or decentralized storage, it is encrypted with the data owner's (*sender*) key  $pk_s$  (Fig. 2). The data itself is encrypted with a random symmetric key  $dek$ , with one key per file. The  $dek$ , encrypted with  $pk_s$  is attached to the encrypted data. This combination  $(edek, c)$  can be stored anywhere - in IPFS, Swarm, S3, or any kind of decentralized or centralized storage.

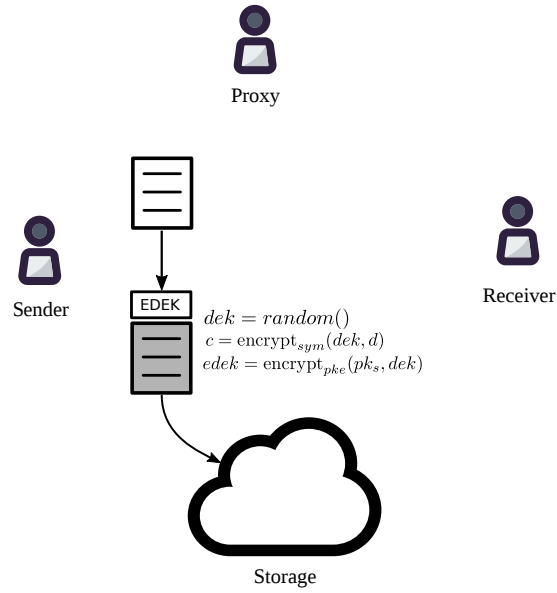


FIG. 2: Architecture: encryption

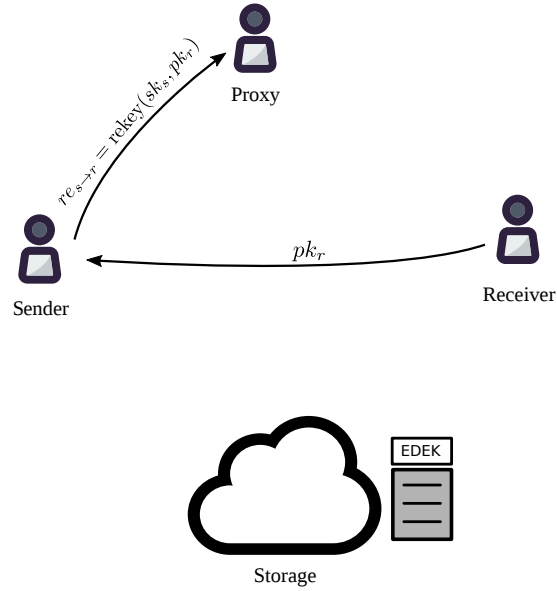


FIG. 3: Architecture: access delegation

When storing the data, the user to whom we delegate access is not necessarily known in advance. First, the receiver should show the sender his public key (Fig. 3). It often makes sense for the public key to correspond to an address in the Ethereum network (then it will be shown once a transaction is made from that address to make a payment for a digital content subscription, for example). The sender generates a re-encryption key  $re_{s \rightarrow r}$  (including an encrypted random ephemeral key when needed) and sends it to a random re-encryption node, selected according to proof-of-stake out of the active nodes in a decentralized network. The case where multiple nodes are selected for redundancy or security will be discussed later. The nodes which have shared data of user *sender* with a user *receiver* register this information in the network.

When the receiver wants to decrypt data shared with him, he first downloads that data from storage or an encrypted



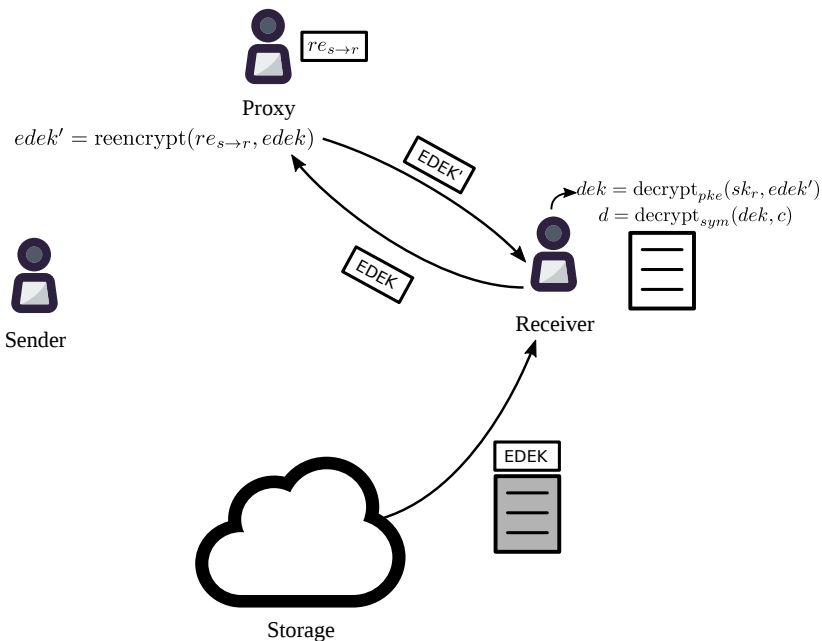


FIG. 4: Architecture: decryption

stream (Fig. 4). He separates out  $edek$  from the message and sends  $edek$  to the network of re-encryption nodes and finds active re-encryption nodes which can share the data of the sender with the receiver (those which have re-encryption key(s)  $re_{s \rightarrow r}$ ). The receiver asks the node(s) that has the re-encryption key to transform  $edek$  to  $edek'$  and uses his own secret key  $sk_r$  to decrypt it and obtain DEK. Now, he can use DEK to decrypt the bulk of the data.

### III. NETWORK SECURITY

In the network, there are multiple re-encryption nodes which apply access management policies.

Proxy re-encryption allows NuCypher KMS to split the trust between access management and decryption rights, without introducing an always-online always-trusted entity (such as a traditional key management system). Miners never see plaintext data, or anything which allows them to decrypt the data. They are solely responsible for storing re-encryption keys and applying re-encryption functions.

The first risk of this model is collusion between a miner and a reader of the data. If the miner gives the reader re-encryption keys for the data shared with him, the data can be decrypted at any time by the data reader, circumventing any conditional or time-based constraints. We counteract this threat in multiple ways: pseudo-anonymity of re-encryption keys, split-key proxy re-encryption, and a challenge protocol. In addition, we apply economic incentives for fair operation, described in Sec. VI.

The second risk is nodes malfunctioning (returning fake data instead of performing re-encryptions). We solve this problem using a challenge protocol.

The third risk is nodes colluding with each other to perform 50% attacks. This risk is usually deadly for multi-party computations [21] (such as Enigma [22]). However, in our case the attacker only gains the ability to wrongfully apply re-encryption policies, not to decrypt data or to grant access to a user who hasn't been granted access to the data. Ideally, the system should be as decentralized as possible, however 50% attacks don't compromise the confidentiality

of the data, just like 50% attacks in proof-of-work cryptocurrencies don't give an attacker the ability to move funds.

### A. Pseudo-anonymity <sup>1</sup>

It is highly beneficial for the security of the system that re-encryption nodes do not know what is it they are re-encrypting. This prevents them from knowing which re-encryption keys to perform collusion attacks on (and trying to collude with all the network participants is infeasible when the network is decentralized). But pseudo-anonymity of re-encryption keys also enables us to run a challenge protocol (Sec. III C).

In order to achieve pseudo-anonymity, we propose the following protocol. If each recipient of the data has a key pair  $s_r/p_r$ , and the path or identifier of the data shared is  $n$ , each re-encryption key has a unique ID  $i_r$  associated with it:

$$nonce = \text{random}(), \tag{16}$$

$$i_r = \text{HMAC}(p_r || n, nonce), \tag{17}$$

where HMAC is a hash-based message authentication code [23]. The value of  $i_r$  is unique and impossible to guess without knowing a random nonce. The nonce should be told to the recipient of the data once the data is shared, so that the recipient can form a proper request to apply the re-encryption key related to the data shared.

Re-encryption nodes store re-encryption keys  $re$  in an in-memory key-value store  $i_r \rightarrow re$ . The messages can be conveniently transmitted over Ethereum's Whisper protocol [24] which conveniently supports anonymity of encrypted messages.

### B. Split-key re-encryption

Imagine that a re-encryption node decides to re-encrypt data immediately rather than to apply conditional policies as instructed. A split-key proxy re-encryption scheme can be used to solve this problem.

Instead of one re-encryption key, m-of-m re-encryption keys can be used to produce "re-encryption shares." These shares can be combined client-side. An m-of-m scheme exists for AFGH [15] encryption. A collusion attack here would require  $m$  miners *and* the reader of the data.

A threshold-based m-of-n scheme, similar to what will soon be published by NICS Labs from University of Málaga, appears to be the most appropriate for this task.

### C. Challenge protocol

There is a risk of miners returning random numbers instead of correctly re-encrypting data. Since the data is private, users of the system cannot publish this data and their key as proof that the miner has cheated.

Provided we have pseudo-anonymity, a miner cannot distinguish between a "true re-encryption" and a re-encryption of random data. So, we can produce a number of "fake" re-encryption keys which are designed specifically to challenge the miners. If a miner cheats, the data and the key for this challenge aren't associated with any private data.

---

<sup>1</sup> We are completely reviewing this subsection. Most likely, the actual anonymization protocol will be based on re-encryption mixnets.

The miners should show the hashes of data before and after re-encryption to the network. If this re-encryption was a challenge and the miner has cheated, challengers can present a proof that non-sensitive keys related to this challenge should actually produce a different re-encryption result, and the miner’s collateral deposit can be awarded to the challenger.

The system should also intentionally produce a number of “wrong re-encryptions”, in order to incentivize challengers to operate, as pointed out by Truebit [25].

Special consideration should be given to protecting re-encryption keys from leaking. The following challenge protocol is proposed.

When accepting responsibility for a re-encryption key, a miner expects to get a fee  $f$  over time  $T$ , so the owner of the data deposits  $f$  coins. The miner should also put up collateral  $c$  which will be forfeit if leaks the re-encryption key is leaked.

If a challenger proves that the miner has leaked a re-encryption key, the challenger should be rewarded. However, the data owner may challenge the miner in order to fraudulently collect the challenge reward. We make this “self-challenge” infeasible. If the challenge has happened after time  $t$ , the challenger will get  $\alpha ft/T$  coins, where  $\alpha < 1$ . The data owner in this case gets  $(1 - t/T)f$  coins returned. The collateral and the rest of the fee gets seized for the benefit of the other participants of the network, with the total amount of  $c + (1 - \alpha)t/T$ .

There also should be no incentive for the owner of the data to fake-challenge the miner instead of revoking the policy. So, in a “correct” revocation, the owner of the data gets  $(1 - t/T)f$  coins back, and the miner gets  $c + ft/T$  coins, where  $c$  is the collateral which was staked.

#### D. Relevancy of possible threats to different use cases

In a mobile device management use case (Sec. VIII), the most important thing is to revoke access from a lost or stolen device before the data is compromised. A possible attack is a user who takes the device and colludes with the relevant miners. As such, there must be no way for miners to identify a user, and vice versa. Another possible attacks is a group of miners revoking access and demanding additional payment to re-encrypt. However, there is no incentive to do so since the owner of the data can easily re-grant access to that mobile device. Another possible threat is a mining node keeping re-encryption keys for a long time beyond the life of the policy, waiting for someone to attack the end-user device and collude with the mining node. In order to prevent this threat, it is important that the mining node cannot figure out if the data is valuable or not, and a good way to do it is to anonymize the data owner and the data itself. In other words, we prevent the mining node from discovering which data the *edek* corresponds to.

Decentralized DRM (Sec. VIID) assumes that once content (a file or a piece of video) is decrypted, it has been purchased, so access revocation is not really a problem in most DRM use cases. However, if a node knows that the content is very expensive, they may attempt to approach the buyer and ask him to pay them a cheaper price for the content, cutting out the original seller. To prevent this, we should anonymize the recipient of the data. It would also help to hide the exact pricing information from the mining node while still allowing it to verify the necessary amount was paid using zk-SNARKs [26].

When NuCypher KMS is used to secure access to files (Sec. VIIA) or messages, both granting and revocation of access are important. So full anonymization is highly desirable. Possible attacks include recipients of the data bribing miners to continue having access after it should have been revoked and miners extorting fees from the owner when it is critical to revoke access. Anonymization appears to be an important part of making such attacks infeasible.

### E. Hardware-enforced security

If miners misbehave, they risk losing their collateral deposit. However, rather than purposeful ill-intent, miner nodes could be the victim of a third-party attack. In order for miners to prevent their nodes from being compromised, they can use commonly accessible secure hardware, such as the latest generation of Intel CPUs (Skylake+) which support SGX, or NVidia GPUs.

Intel SGX technology [27] promises to run *any* computations in a secure environment. It was previously proposed to have a decentralized network for managing secrets relying on the SGX technology rather than fairness of miners or re-encryption [28].

Alternatively, holding and applying re-encryption keys can be done inside GPUs. It was shown that GPUs can serve as trusted platform modules with limited capabilities (although they can certainly execute re-encryption) [29].

## IV. PERFORMANCE CONSIDERATIONS

### V. FUNCTIONALITY

#### A. Local encryption library and daemon

NuCypher KMS can be interfaced from a traditional, centralized application. In Python, sharing a file in IPFS would look like:

```
import nkms
nkms.connect() # Using default config
path = 'ipfs://QmTkzDw.../to_the_moon.avi'
nkms.share('0xab12...', path, time=86400)
```

If there is no library client available, there could be a local API server, similar to how geth is used to interact with the Ethereum network.

Reading this file would be:

```
import nkms
client = ipfsapi.Client(...)
nkms.connect() # Using default config
path = 'ipfs://QmTkzDw.../to_the_moon.avi'
edata = client.cat(path)
data = nkms.decrypt(edata, path)
print(data)
```

The “decrypt” function splits *edata* into *edek* and the actual encrypted data, requests the kms network to transform *edek* into *edek'*, decrypts *edek'* with the receiver’s private key and decrypts data with the obtained *dek*.

The preliminary version of the API will include the following functions:

- connect — connect to the decentralized network, taking configuration from arguments or a config file;
- write — encrypt data with a public key corresponding to the owner of the file and save in a storage backend;

- read — download data from the storage backend, ask the decentralized network to re-encrypt, and decrypt with our private key;
- delete — delete file and re-encryption keys associated with it;
- decrypt — decrypt data which we’ve already read;
- split-edek — low level function to separate out encrypted symmetric encryption key from the data;
- share/renew/revoke — create a permission to read all the data we own, or a subset of it, based on file path or policy. The policy can include time limits and other conditions;
- read-policies/update-policies/delete-policies — read and change all the access policies we’ve created.

### B. Sharing short secrets

Low-level functionality allows encrypting and delegating access to binary secrets (such as database credentials) or groups of these secrets without storing them in individual files. A possible backend for storing simple secrets like these could be a hierarchical config file (in YAML or JSON format) stored in IPFS, or even simply in the same instance image. Client-side software can parse this file and ask to re-encrypt only the secrets it needs. There could be per-field and per-subfield re-encryption keys for granular permissions.

### C. Sharing files and hierarchical data

When working with files, each file and each directory is encrypted under its secret key. Each file has its DEK stored encrypted by its secret key, the directory’s secret key, and so on, up to the root folder of the user. When a directory is shared, a re-encryption key is created only for the shared directories.

The method above requires storing as many EDEKs as there are levels in the tree structure. If we use a multi-hop, uni-directional algorithm (LWE being the only one [18]), we can have re-encryption keys from the bottom levels to the top levels (Fig. 5). When access is granted to a directory, all its files and sub-directories have a path to be re-encrypted under the directory key, and then re-encrypted for the recipient:

$$edek_b = \text{reencrypt}(re_{xb}, \text{reencrypt}(re_{1x}, edek_1))$$

### D. Sharing encrypted streams

Sharing encrypted streams with multiple users is ideal for applications such as a decentralized Netflix, where third parties routing traffic aren’t permitted to see the content.

The principle is simple: each block of data in the stream is encrypted with a random DEK, and EDEK is produced using a per-channel public key. However, it takes time to make a roundtrip with NuCypher KMS. Therefore, it makes sense to also include an EDEK of the next block of data together with the previous block.

As usual, consumers of the stream have their own key pairs and ask miner nodes to make EDEKs decryptable by the consumers. Miners can also ensure the consumer actually paid for the subscription and refuse to re-encrypt if they did not.

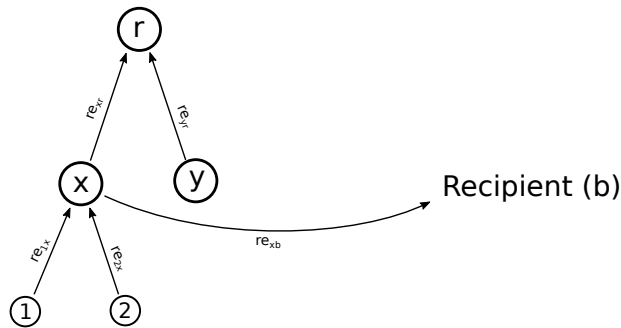


FIG. 5: Sharing hierarchical structures (such as files and folders)

### E. Time-based and condition-based policies

While we don't trust miners with encrypted data or keys, we still trust them to control the duration of storing a re-encryption key. The simplest policy is time-based: re-encryption is allowed only during the specified time interval and, if there is no time in the future when re-encryption is allowed, the re-encryption key should be removed.

More complex policies can be created. Re-encryption can be allowed on a condition — for example, pending the completion of a certain transaction. This allows applications like pay-per-content DRM, or storing secret data as an escrow for financial transactions.

### F. Key rotation

Many proxy re-encryption algorithms can be applied multiple times, BBS98 [14] included. Thus, proxy re-encryption can be used for key rotation. Key rotation allows all the EDEKs encrypted with an old key to become encrypted with a new key.

The owner of the data needs to produce re-encryption keys  $re_{v1 \rightarrow v2}$  between two versions of secret keys, essentially sharing data with her future self. Now, re-encryption nodes (assuming the encrypted storage is public, such as IPFS) will download EDEKs and apply the transformation. For this operation, no collusion risks exist since the “sender” and “receiver” of the data are, essentially, the same person.

## VI. TOKEN ECONOMICS

Protocol economies consist of a network of miners that contributes work to provide a scarce resource and that is rewarded when said resource is consumed. In NuCypher KMS, miners are re-encryption nodes. Anyone can become a miner and their rewards are differentiated based on the amount of re-encryption operations provided. Access to the scarce re-encryption services must be controlled and allocated to the highest value uses. The mechanism by which we achieve this is NuCypher KMS Token (NKMS). NKMS is both the reward miners get for contributing work and the price consumers (owners of the data) pay for access to re-encryption services. Vitaly, the token also incentivizes correctness of computation and security of the system.

### A. Token distribution

All the NKMS Tokens exist from the time of creation of the system on mainnet.

Miners can hold a number of re-encryption keys proportional to the number of tokens they hold as a collateral deposit, locked by a smart contract. The miners are paid both for providing re-encryption services and making themselves available to re-encrypt data. The miners can be paid either by the owner of the data, or by users of the data. The latter is more relevant for usecases where DAOs distribute content under decentralized DRM.

If miners become unavailable, they lose their “availability” compensation for the time period.

If miners cheat and provide wrong re-encryptions, they lose a fraction of their security deposit (Sec. III C).

We can disincentivize miners from leaking re-encryption keys. Anyone can challenge a miner with a hash of a re-encryption key and, if the re-encryption key is proven to have been leaked, the miner forfeits his collateral valued in NKMS.

An interesting property of NuCypher KMS is that security improves as the number of network participants grows. As additional re-encryption nodes enter the network, the lower the chance of collusion. This improves both the security and censorship-resistance of the system, providing powerful network effects and meaningful first mover advantages.

Finally, in order to test the security of the network, we can put bounty secrets — private keys for wallets with some crypto-currency. Anyone is free to hack the system and take the bounty. The fact of the bounty being taken would be proof of compromised security. This same mechanism can be used to warn of a potential data leak by individual users of the network, by signalling that they may have exposed their key to the data.

## VII. USE CASES

NuCypher KMS provides the infrastructure for a variety of applications that require sharing of sensitive data as a basic functionality. The ability to condition decryption operations on public actions on the consensus network, such as the publication of certain messages, payments made between specific parties, and other events, enables a range of applications including:

### A. Sharing encrypted files (“Decentralized Dropbox”)

Files can be encrypted client-side and stored in decentralized filesystems like Swarm [30], IPFS [31], Sia [32], or Storj [33], or centralized ones like S3. The files can be easily shared with approved third-parties by providing a re-encryption token based on the third-party’s public key. The third-party’s access permission can be easily revoked by removing the re-encryption token from the network.

### B. End-to-end encrypted group chat (“Encrypted Slack”)

PRE is an ideal primitive for end-to-end encrypted group messaging, in which multiple participants require read and write access to a channel. Members can easily be added or removed to the chat by issuing or revoking a re-encryption token. This avoids the overhead of encrypting and sending messages multiple times, individually for each participant.

### **C. Patient-controlled electronic health records (EHR)**

A patient-controlled EHR can be created in which the patient owns their data and encryption keys, as opposed to centralized systems like Epic. Again, the data can be stored centrally or in a decentralized backend. When the patient wants to share their encrypted data with a hospital or insurance company, they issue a re-encryption token, which grants temporary access to the third-party.

### **D. Decentralized digital rights management (DDRM)**

Cryptographic access controls can act as a decentralized DRM. Access controls can be embedded into the encryption itself so that they follow the data wherever it goes. Conditional re-encryption tokens can be controlled by a smart contract and released only upon payment. Services like a decentralized Netflix or an encrypted marketplace selling software, apps, photos, and other digital content can now be built using NuCypher KMS.

### **E. Blind identity management**

A blind identity management service can be constructed using NuCypher KMS. Identities can be encrypted client-side and stored with the identity management provider. Users can create re-encryption keys for approved applications. The service re-encrypts identity credentials for said third-party applications, without the identity provider ever having access.

### **F. Secret credentials management for scripts and backend applications**

NuCypher KMS is ideal for the storage of any secrets, such as sensitive environment variables, database credentials, and API keys. For scripts, a re-encryption token can be generated for the duration of a script, then revoked. For example, developers can safely store encrypted database credentials on GitHub, giving temporary access to these credentials once an instance is deployed. Even if the GitHub repository is public, the credentials cannot be used by an unauthorized person.

### **G. Shared credentials and enterprise password management**

NuCypher KMS can manage shared credentials that employees use to access web services. An audit log can be built to monitor who accesses what secrets. When an employee leaves, it is easy to revoke access or even roll keys.

### **H. Mandatory access logging**

In some corporate and enterprise settings, clients must publish access logs for sensitive files. This requires that each file access be recorded and conditional re-encryption can be used to mandate these logging rules.



## I. Mobile device management (MDM) and revocation

In an enterprise MDM setting, re-encryption tokens can be created for valid devices. When a device is lost or retired, or an employee leaves the organization, the re-encryption token can be deleted to revoke the device's access. This avoids the problem of re-organizing hierarchical key trees.

## VIII. SUMMARY

NuCypher KMS is a decentralized key management service and cryptographic access control layer for the blockchain and decentralized applications. Developers and enterprises alike can leverage it to create highly-secure applications in healthcare, financial services, and more. By bringing private data sharing and computation to the public blockchain, NuCypher KMS enables everything from encrypted content marketplaces to secret credentials management to patient-controlled electronic health records.

## IX. ACKNOWLEDGEMENTS

We would like to thank Isaac Agudo Ruiz from University of Málaga for his help with proxy re-encryption algorithms. We also thank Dave Evans at the University of Virginia for advising our company throughout its lifetime and his help with regards to the current state of the art in multi-party computation. And Stefano Bernardi, Tom Ding, and many others for their help on token economics.

- 
- [1] Gabriel Kaptchuk, Ian Miers, and Matthew Green, “[Managing secrets with consensus networks: Fairness, ransomware and access control](#),” Cryptology ePrint Archive, Report 2017/201 (2017).
  - [2] Wikipedia, “[Proxy re-encryption — Wikipedia, the free encyclopedia](#),” (2017).
  - [3] Wikipedia, “[Key management — Wikipedia, the free encyclopedia](#),” (2017).
  - [4] Wikipedia, “[Hardware security module — Wikipedia, the free encyclopedia](#),” (2017).
  - [5] HashiCorp Inc., “[Vault by hashicorp](#),” (2017).
  - [6] Amazon Inc., “[Aws cloudhsm](#),” (2017).
  - [7] Alphabet Inc., “[Cloud key management service](#),” (2017).
  - [8] Microsoft Inc., “[Key vault](#),” (2017).
  - [9] TrueVault Inc., “[Hipaa compliant api & secure data store](#),” (2017).
  - [10] Wikipedia, “[Advanced encryption standard — Wikipedia, the free encyclopedia](#),” (2017).
  - [11] Wikipedia, “[Salsa20 — Wikipedia, the free encyclopedia](#),” (2017).
  - [12] 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).
  - [13] NuCypher, “[Modern data security for hadoop](#),” (2017).
  - [14] Matt Blaze, Gerrit Bleumer, and Martin Strauss, “Divertible protocols and atomic proxy cryptography,” in *Advances in Cryptology — EUROCRYPT’98: International Conference on the Theory and Application of Cryptographic Techniques Espoo, Finland, May 31 – June 4, 1998 Proceedings*, edited by Kaisa Nyberg (Springer Berlin Heidelberg, Berlin, Heidelberg, 1998) pp. 127–144.

- [15] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger, “Improved proxy re-encryption schemes with applications to secure distributed storage,” *ACM Trans. Inf. Syst. Secur.* **9**, 1–30 (2006).
- [16] Wikipedia, “[Ntruencrypt — Wikipedia, the free encyclopedia](#),” (2017).
- [17] David Nuñez, Isaac Agudo, and Javier Lopez, “Ntruencrypt: An efficient proxy re-encryption scheme based on ntru,” in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS ’15 (ACM, New York, NY, USA, 2015) pp. 179–189.
- [18] Le Trieu Phong, Lihua Wang, Yoshinori Aono, Manh Ha Nguyen, and Xavier Boyen, “Proxy Re-Encryption Schemes with Key Privacy from LWE.” *IACR Cryptology ePrint Archive* **2016**, 327 (2016).
- [19] Wikipedia, “[Chosen-plaintext attack — wikipedia, the free encyclopedia](#),” (2017).
- [20] Wikipedia, “[Chosen-ciphertext attack — wikipedia, the free encyclopedia](#),” (2017).
- [21] Vitalik Buterin, “[Secret sharing daos: The other crypto 2.0](#),” (2014).
- [22] Guy Zyskind, Oz Nathan, and Alex Pentland, *Enigma: Decentralized Computation Platform with Guaranteed Privacy*, Tech. Rep. (2015).
- [23] Wikipedia, “[Hash-based message authentication code — wikipedia, the free encyclopedia](#),” (2017).
- [24] “[Whisper overview](#),” .
- [25] Jason Teutsch and Christian Reitwießner, “A scalable verification solution for blockchains,” (2017).
- [26] Consensys, “[Introduction to zk-SNARKs with examples](#),” (2017).
- [27] Wikipedia, “[Software guard extensions — wikipedia, the free encyclopedia](#),” (2017).
- [28] Gabriel Kaptchuk, Ian Miers, and Matthew Green 0001, “Managing Secrets with Consensus Networks: Fairness, Ransomware and Access Control.” *IACR Cryptology ePrint Archive* **2017**, 201 (2017).
- [29] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis, “PixelVault: Using GPUs for Securing Cryptographic Operations,” *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* , 1131–1142 (2014).
- [30] Victor Trón, Aron Fischer, and Daniel Varga, “[Smash-proof: auditable storage for swarm secured by masked audit secret hash](#),” (2016).
- [31] Juan Benet, “[IPFS — content addressed, versioned, P2P file system](#),” (2014).
- [32] Nebulous Inc., “[Sia: Simple decentralized storage](#),” (2014).
- [33] Storj Labs Inc., “[A peer-to-peer cloud storage network](#),” (2016).