1 Introduction

The Streamr Network is a decentralised publish-subscribe network. Users can publish infinite streams of data to the network, and other interested users can subscribe to these streams. Intermediate nodes in the peer-to-peer network help distribute the data from publishers to subscribers. As these data streams are sent over a public insecure network (the internet) and via a set of untrusted intermediate nodes, the only way to guarantee that only authorized subscribers can read the data is to use end-to-end multicast encryption. While encryption solves the problem of data confidentiality, it creates the problem of key management which is especially hard in the case of multicast encryption. How can we deliver the decryption keys to valid recipients only? How can we ensure that a user can no longer decrypt the data after their permission has been revoked?

Any key management solution should also be concerned with scalability in terms of network traffic and storage. Any data point to be published by the network should be encrypted with a single ciphertext such that it can be published as a single message on the stream and stored as a single message. But the most challenging problem we face when dealing with multicast encryption is revocation. Usually, the messages are encrypted and decrypted with a symmetric group key known by all the subscribers. But if at some point a subscriber must be revoked, we cannot make him "forget" the symmetric group key, so we have to make a new key available to the remaining subscribers. This process commonly known as rekeying can be very expensive in terms of traffic if the number of subscribers is very large and revocations are frequent. The naive solution sends a separate message containing the new group key to every remaining subscriber using asymmetric cryptography, thus it has O(N) complexity where N is the number of subscribers.

Research using logical key hierarchy (LKH)[1] has shown that solutions with O(log(n)) complexity exist. However, our protocol is based on research[2][3] using threshold cryptography that provides
a solution with $O(1)$ complexity. Our protocol allows to rekey with an arbitrary number of subscribers using only a single message. Such a scheme has many applications. IP multicast minimises the number of messages to send over the network using a minimum spanning tree. But in the case of encryption, it requires the message to be a unique ciphertext and a naive rekeying scheme would defeat the advantages of IP multicast if revocations are frequent. This applies to any kind of overlay network that requires confidentiality and the ability to revoke endpoints. TV channels also deal with this issue when a user’s subscription ends, especially in the case of satellite TV since the encrypted data is easy to get.

We will first describe in details what are the requirements for a multicast encryption protocol in a publish-subscribe network. We then present a first version of the protocol that makes use of a naive rekeying scheme for revocation, followed by the actual protocol that uses a rekeying scheme with $O(1)$ complexity. Finally, we will evaluate and compare the two schemes.

2 Specifications

In this section, we define formally what the multicast encryption protocol should allow us to achieve. The protocol is used by two types of participants:

- One single publisher denoted $p$.
- The subscribers, each denoted by $s_i$ and belonging to the set of subscribers $S$.

The publisher wants to deliver a stream of messages in real-time to the subscribers. Other messages containing metadata for key management are not transmitted on this stream but directly between the participants. The publisher $p$ is the only one sending encrypted messages on the stream, so he’s in full control of the keys used to encrypt them. In that sense, our protocol differs from other more general group communication protocols where each member of the group must play a part in some key agreement protocol. The publisher decides which subscribers in $S$ are allowed to read the sent messages, these subscribers are called valid subscribers and belong to a subset of $S$ denoted $S_V$. At any point, the publisher can decide to remove subscribers from $S_V$. The subscribers only receive messages from the publisher, we assume that there is no collusion or communication between them. At any point, any subscriber $s_i$ can send a message to the publisher to request to be a part of $S_V$. The protocol must ensure that the following properties hold:

1. For any plaintext $m$ sent as a ciphertext $c$ by $p$ at time $t$ on the stream, every subscriber $s_i \in S_V$ at time $t$ must be able to recover $m$ from $c$. 

2
2. For any plaintext $m$ sent as a ciphertext $c$ by $p$ at time $t$ on the stream, every subscriber $s_i \notin S_V$ at time $t$ must not be able to recover $m$ from $c$.

3. For any subscriber $s_i$ becoming a member of $S_V$ at time $t$, $s_i$ must not be able to decrypt ciphertexts sent before time $t - \delta t_1$.

4. For any subscriber $s_i$ excluded from $S_V$ at time $t$, $s_i$ must not be able to decrypt ciphertexts sent after time $t + \delta t_2$.

Ideally $\delta t_1$ and $\delta t_2$ would be 0, but ensuring these properties adds some computational overhead depending on $\delta t_1$ and $\delta t_2$. As we will see later in more details, the smaller $\delta t_1$ and $\delta t_2$ are, the more often these computations have to be executed. $\delta t_1$ and $\delta t_2$ are tradeoff parameters between security and performance.

3 Multicast Encryption with Naive Revocation

In this section, we describe a first multicast encryption protocol that revokes subscribers in the most simple and inefficient way: sending the new symmetric group key to every remaining valid subscriber using asymmetric cryptography. The actual protocol that makes use of a more efficient revocation scheme will be presented in Section 4.

3.1 Setup

Every participant $x$ to the protocol (the publisher $p$ and the subscribers in $S$) owns an Ethereum account which has three components:

- A private key denoted $\text{ethPriv}(x)$.
- A public key denoted $\text{ethPub}(x)$
- An Ethereum address denoted $A(x)$. It is derived from the public key and uniquely identifies the participant $x$.

The Ethereum accounts allow the participants to sign and verify the messages they send and receive. Note that we don’t need to establish the trust of the public key or the address of the account using a PKI or a Web of Trust technique because the public key is the identity. It doesn’t need to be linked to any kind of information about the participant. We denote the computation of a signature $s$ of a message $m$ by a participant $x$ with $s = \text{sign}(m, \text{ethPriv}(x))$. Every message sent by any participant is sent along with its signature and the address of the signer. For every message $m$ to send, any participant $x$ sends $m|\text{sign}(m, \text{ethPriv}(x))|A(x)$. But for the sake of simplicity, we will denote the message sent as $m$. 
In addition to the Ethereum account, every subscriber $s_i$ generates a public-private RSA key pair denoted $(rsaPub(s_i), rsaPriv(s_i))$. It will be used for standard asymmetric encryption. Theoretically, the Ethereum account itself could be used for encryption, but in practice it is only used to compute signatures and no standard implementation exists to encrypt and decrypt data using an Ethereum account. On the other hand, RSA is widely used for encryption and has standard implementations in all the most popular programming languages.

We will denote the RSA encryption and decryption of a message $m$ by a subscriber $s_i$ with the following: $c = E(m, rsaPub(s_i))$ and $m = D(c, rsaPriv(s_i))$.

Since we want to efficiently send multicast messages in real-time, we will guarantee their confidentiality with symmetric encryption using a group key common to the publisher and the valid subscribers. This ensures that the ciphertext is the same for every recipient and can easily be gossiped or routed with a multicast protocol. We will use AES-CTR as our symmetric encryption scheme as it is both secure and efficient. It doesn’t provide an HMAC like AES-GCM, which would be both unnecessary (we already guarantee authenticity and integrity with Ethereum signatures) and insecure (an HMAC would only guarantee that the message comes from a valid subscriber or the publisher since they all know the group key).

The publisher $p$ chooses a symmetric key $K$ that is going to be distributed to the valid subscribers (see Subsection 3.2). $K$ will be used by $p$ to encrypt the published messages and by the subscribers in $S$ to decrypt the consumed messages (see Subsection 3.3).

We will denote the encryption of messages $m$ and decryption of ciphertexts $c$ with a symmetric key $K$ by $c = AES(m, K)$ and $m = AES(c, K)$.

### 3.2 Joining

When a subscriber $s_i$ wants to join, meaning he wants to become a valid subscriber added to the set $S_V$, he sends his RSA public key $rsaPub(s_i)$ to the publisher $p$. This lets the publisher $p$ know that he can send encrypted messages to $s_i$ using $rsaPub(s_i)$. The trust of the RSA public key is guaranteed by the Ethereum signature and the address of $s_i$.

Upon reception of the RSA public key $rsaPub(s_i)$, $p$ decides if $s_i$ is eligible to be part of the set of valid subscribers $S_V$. In our case, this is done by checking that $s_i$ purchased the product, which is a subscription to the stream’s data. If so, $p$ adds $s_i$ to $S_V$, computes the encrypted group key $C = E(K, rsaPub(s_i))$ and sends it back to $s_i$.

$s_i$ can recover the symmetric group key $K = D(C, rsaPriv(s_i))$ and will be able to use it to decrypt the messages published on the stream.
3.3 Symmetric Encryption and Forward Secrecy

For every message \( m \) to publish, \( p \) sends \( c = AES(m, K) \) on the stream. Only the subscribers who know \( K \) will be able recover \( m = AES(c, K) \). The publisher \( p \) ensures that every valid subscriber \( s_i \) in \( S_V \) knows \( K \) by answering joining requests (see 3.2) if and only if the sender is either already in \( S_V \) or eligible to be part of \( S_V \).

The first and second properties described in Section 2 are guaranteed by this simple mechanism but not the third: A subscriber \( s_i \) initially considered invalid can eavesdrop messages encrypted with \( K \). If at some point \( t \), \( p \) decides to grant \( s_i \) access to the stream (\( s_i \) becomes a member of \( S_V \)) by answering his joining request, then \( s_i \) will come to know \( K \). He will be able to decrypt messages published from \( t \) onwards as expected, but also every eavesdropped message published before \( t \).

We can resist against such an attack by updating the key. Instead of having a single permanent key \( K \), \( p \) creates a first key \( K_0 \) and generates a new key on every message. For each message \( m_i \) to publish, \( p \) generates a new key \( K_{i+1} \) and sends \( c_i = AES(m_i|K_{i+1}, K_i) \). Every valid subscriber \( s_i \) knows the initial key \( K_0 \) from the joining request. So every ciphertext \( c_i \) can be decrypted using the current key \( K_i \) to recover \( m_i \) and \( K_{i+1} \) which will be used to decrypt the next ciphertext. Let be \( K_j \) the last generated key at time \( t \), any subscriber \( s_i \) who joins at that time will receive \( K_j \) which allows \( s_i \) to decrypt all future messages but not the past messages because they were encrypted with previous keys.

This key updating mechanism guarantees the third property even with \( \delta t_1 = 0 \) because the key is changed for every single message. But both the time overhead of generating a new key at every message and the space overhead of including the next key in every ciphertext can be reduced by changing the condition for updating the key. The publisher \( p \) is in control of updating the key after a certain amount of messages published or a certain amount of time. To guarantee the third property, the amount of time between two key updates has to be at most \( \delta t_1 \). If at some time \( t_1 \), \( p \) updates the key to be \( K_j \), a subscriber \( s_i \) can eavesdrop the ciphertexts sent after \( t_1 \) and send a joining request at time \( t_2 \). If the key has not been updated in between, \( s_i \) will receive \( K_j \) and will be able to decrypt all messages published between \( t_1 \) and \( t_2 \). The property is guaranteed if we always have \( t_2 - t_1 \leq \delta t_1 \).

3.4 Revocation

At this point, our protocol doesn’t guarantee the fourth property described in 2 because any subscriber \( s_i \) who obtains the symmetric group key can always decrypt future messages, even if the publisher \( p \) decides to remove \( s_i \) from \( S_V \). The process of removing a subscriber from the set of valid subscribers such that he can no longer decrypt messages is
called revocation. To revoke subscribers, the publisher needs to decide on a new symmetric group key and make it available only to the subscribers that are still considered valid. We call this operation rekeying.

In this first protocol, the rekeying mechanism sends the new key to every remaining subscriber using their RSA public key. To revoke a subscriber \(s_r\) that was removed from \(S_v\), the publisher \(p\) keeps in memory for every valid subscriber \(s_i\) its RSA public key \(rsaPub(s_i)\). \(p\) generates a new symmetric group key \(K_{\text{reset}}\) and sends it separately to every subscriber \(s_i\) in \(S_v\): \(C_{\text{reset}} = E(K_{\text{reset}}, rsaPub(s_i))\). Every valid subscriber \(s_i\) can recover the new group key with his RSA private key: \(K_{\text{reset}} = D(C_{\text{reset}}, rsaPriv(s_i))\), which \(s_r\) won’t be able to do because he didn’t receive anything and doesn’t know the other subscribers private keys.

This approach is the most simple and inefficient because for a system of \(N\) subscribers, it needs to individually create and send \(N\) different unicast messages. The complexity of a rekeying mechanism is defined by the number of messages exchanged for a given number of subscribers. Thus, this mechanism has \(O(N)\) complexity. Using logical key hierarchy[1], it is possible to have to rekey only the subset which the subscriber to revoke is in, and obtain \(O(\log(N))\) complexity. But there exists another approach, based on threshold cryptography, that allows to construct rekeying schemes with \(O(1)\) complexity. The protocol presented in the next section uses a rekeying mechanism based on this approach.

Another way to improve the efficiency of revocation is to rekey only once we have a given number of subscribers that need to be revoked or some amount of time has elapsed. In the scheme described above, when \(p\) decides to exclude \(s_r\) from \(S_v\), \(p\) immediately picks a new group key \(K_{\text{reset}}\) and triggers the rekeying mechanism. This ensures that if \(s_r\) is excluded at time \(t\), we will not be able to decrypt any messages sent after \(t\). This is ideal from a security point of view, but the fourth property described in 2 allows \(s_r\) to decrypt messages until time \(t + \delta t_2\). So instead of immediately changing the group key and rekeying, \(p\) can do it at time \(t + \delta t_2\) and the property will still hold. This allows \(p\) to exclude other subscribers from \(S_v\) between time \(t\) and time \(t + \delta t_2\) and revoke all of them in a single round at time \(t + \delta t_2\) instead of one round per subscriber to revoke.

4 Revocation with Threshold Decryption

This section improves the protocol described in Section 3 by replacing the naive \(O(N)\) revocation scheme with an efficient \(O(1)\) revocation scheme. Like in Section 3, we will describe the setup, joining and revocation phases, but we won’t discuss the symmetric encryption phase as it is not affected by this improvement.
4.1 Threshold Decryption

The revocation mechanism described in [2] relies on a cryptographic primitive called threshold decryption. In 4.1.1 we describe the interface of this primitive as a black box. This should be enough to understand the overall revocation mechanism. For readers interested in the inner workings of this primitive, 4.1.2 provides an in-depth explanation.

4.1.1 Definition

A threshold decryption scheme is a scheme that allows encryption of any plaintext with a single public key but requires at least $t$ out of $n$ different decryption keys to decrypt the ciphertext. Each of these $n$ keys can produce a decryption share from the ciphertext, then an algorithm can combine any set of at least $t$ decryption shares to return the plaintext. Commonly, the scheme provides a setup function to choose the threshold $t$ and create a fixed set of $n$ keys. As we will see later, our use case requires the ability to add a new key to the set of existing keys without modifying the required threshold, so the setup function can be simplified to simply fix the threshold and not create any keys yet. Some schemes also define a verification key to be able to check the integrity of every decryption share, we don’t need this functionality because authentication and integrity of any data is already guaranteed by digital signatures.

Our ($t$)-threshold decryption scheme provides the following functions:

- **setup($t$)**: fixes the threshold to $t$ and returns a public key $PK$ and some underlying secret object $P$ that allows to create secret partial decryption keys. More details about what $P$ is in 4.1.2.

- **generateKey($P$)**: creates and returns a new partial decryption key $sk$.

- **encrypt($m, PK$)**: encrypts the plaintext $m$ with the public key $PK$ and returns a ciphertext $c$.

- **decryptShare($sk, c$)**: computes a partial decryption share $s$ of the ciphertext $c$ using a secret partial decryption key $sk$.

- **combineShares($S, c$)**: $S$ is a set of at least $t$ decryption shares. Combines the shares in $S$ and $c$ to reconstruct and return the plaintext $m$. Fails if $|S| < t$.

4.1.2 El-Gamal-based Scheme

There are multiple schemes that implement threshold decryption, based on different asymmetric cryptosystems like El-Gamal[4][5], RSA[6][7] or Pallier[8]. Our scheme is based on El-Gamal and makes use of Shamir’s Secret Sharing (SSS)[9], which we need to introduce before diving in the details of our scheme.
Shamir’s Secret Sharing

SSS is based on the following idea: consider a line’s equation, any two points on this line will be sufficient to reconstruct the equation, but a single point won’t be enough. For a polynomial of the form \( ax^2 + bx + c \), three points are required to reconstruct it. More generally, a polynomial of degree \( k \) can be reconstructed with at least \( k + 1 \) points \((x_0, y_0), \ldots, (x_k, y_k)\) using Lagrange’s interpolation formula:

\[
P(x) = \sum_{i=0}^{k} y_i l_i(x)
\]

where each Lagrange basis polynomial \( l_i(x) \) is computed in the following way:

\[
l_i(x) = \prod_{j=0, j \neq i}^{k} \frac{x - x_j}{x_i - x_j}
\]

With a polynomial \( P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_k x^k \), we can evaluate \( P(0) = a_0 \). So any \( k + 1 \) points of \( P(x) \) are sufficient to find the value \( a_0 \). To find only \( a_0 \) and not all the coefficients, it can be shown that (1) and (2) can be optimised this way:

\[
a_0 = P(0) = \sum_{i=0}^{k} \prod_{j=0, j \neq i}^{k} \frac{x_j}{x_j - x_i}
\]

Based on this constatation, SSS allows to split a secret \( S \) in \( n \) parts such that knowing a threshold of at least \( t \) of these parts allows to reconstruct \( S \). We define the polynomial \( P(x) = S + \sum_{i=1}^{t-1} a_i x^i \) and \( n \) points \( p_i = (r_i, s_i = P(r_i)) \) where each \( r_i \) is chosen randomly. Any subset of \( t \) of these points will be enough to reconstruct \( P(x) \) and compute \( P(0) = S \).

Now that we introduced the SSS primitive, we will describe how our scheme implements the functions defined in 4.1.1.

public parameters

The elliptic curve used in our scheme is Curve25519[10]. It is a Montgomery curve over a prime field defined by \( p = 2^{555} - 19 \). Its base point \( G \ (x = 9) \) generates a cyclic subgroup of prime order \( n \). \( KDF \) denotes the key derivation function, which in our case is PBKDF2.

setup(t)

1. Sample \( a \) in \( \mathbb{Z}_n \)
2. Compute the public key \( P_K = aG \)
3. Sample $t - 1$ elements $a_i$ in $\mathbb{Z}_n$ where $1 \leq i \leq t - 1$

4. Define the polynomial $P$ such that $P(x) = a + \sum_{i=1}^{t-1} a_i x^i \mod n$

5. Return $P_K$ and $P$

addkey($P$)

1. Sample $id$ in $\mathbb{Z}_n$

2. Compute $k = P(id)$

3. Return $sk = (id, k)$

crypt($m$, $PK$)

1. Sample $r$ in $\mathbb{Z}_n$

2. Compute $C_1 = rG$

3. $K = KDF(rP_K) = KDF(arG)$

4. Compute $c_2 = AES(K, m)$

5. Return the ciphertext $c = (C_1, c_2)$

decryptshare($sk$, $c$)

1. We have $sk = (id, k)$ and $c = (C_1, c_2)$

2. Compute $D = kC_1$

3. Return $s = (id, D)$

combine($S$, $c$)

1. Each $s_i$ in $S$ has the form $(id_i, D_i)$ and we have $|S| \geq t$

2. We have $c = (C_1, c_2)$

3. For each $s_i$, compute $l_i = \prod_{j=1,j\neq i}^{t} \frac{id_j}{id_j - id_i}$ (See equation (3))
4. Compute \( d = \sum_{i=1}^{t} l_i D_i \)

5. Compute \( K = KDF(d) \)

6. Return \( m = AES(K, c_2) \)

**Proof of correctness of the combination**

We computed:

\[
d = \sum_{i=1}^{t} l_i D_i = \sum_{i=1}^{t} l_i k_i C_1 = \sum_{i=1}^{t} P(id_i) l_i C_1 = \left( \sum_{i=1}^{t} P(id_i) l_i \right) C_1 \tag{4}
\]

From equation (3), we know that \( \sum_{i=1}^{t} P(id_i) l_i = a \). So \( d = aC_1 = arG \). So we computed \( K = KDF(d) = KDF(arG) \) exactly like in the \( encrypt(m, PK) \) function.

### 4.2 Setup

Now that we described the threshold decryption primitive, we can dive in the multicast encryption protocol. Like presented in Subsection 3.1, every participant still has an Ethereum account and every subscriber \( s_i \) still has a public-private RSA key pair \( (rsaPub(s_i), rsaPriv(s_i)) \). The publisher \( p \) still generates an initial symmetric group key \( K_0 \) that is updated at its will (see Subsection 3.3).

In addition, the publisher \( p \) decides on a threshold value \( T \) that defines the maximum number of subscribers that \( p \) will be able to revoke with a single multicast message. We will see later what are the tradeoffs of this parameter \( T \). Then, \( p \) uses a threshold decryption scheme as described in 4.1 and calls \( setup(T+1) \) to generate a public key \( PK \) and a secret object \( P \) that allows to create secret partial decryption keys. \( p \) can now create \( T \) decryption keys by calling \( T \) times the function \( generateKey(P) \) and store them all in a set called \( SK_P \). At the same time, \( p \) initialises an empty set \( SK_S \) that will store one partial decryption key per subscriber.

### 4.3 Joining

Like in Subsection 3.2, any subscriber \( s_i \) that wants to become a valid subscriber sends his RSA public key \( rsaPub(s_i) \) to the publisher \( p \). Let \( K \) be the current symmetric group key used to encrypt and decrypt messages. In the first protocol joining phase described in Subsection 3.2, \( p \) was only sending \( K \) back to \( s_i \). But in this protocol, \( p \) also sends back a secret partial decryption key.
Elements in $SK_S$ are pairs $(s,k)$ where $s$ is a subscriber and $k$ is its associated secret partial decryption key. First, $p$ checks if there already exists in $SK_S$ a pair of the form $(s_i,k_i)$. If not, $p$ creates a new partial decryption key $k_i = \text{generateKey}(P)$ and adds the pair $(s_i,k_i)$ to $SK_S$. Then $p$ sends both the symmetric group key and the partial decryption key back to $s_i$ using the RSA public key to encrypt them: $C = E(K|k_i,rsaPub(s_i))$.

$s_i$ can decrypt both the symmetric key and his partial decryption key using his RSA private key: $K|k_i = D(C,rsaPriv(s_i))$. $s_i$ can now decrypt messages and the next symmetric group key using $K$. He can store $k_i$ locally, it will be used during revocation.

### 4.4 Revocation with a single message

The general idea of threshold-based revocation is that the publisher sends the encrypted group key along with $T$ decryption shares. Since it requires $T + 1$ shares to reconstruct the plaintext, every valid subscriber adds its own decryption share to the set of shares and is able to decrypt. To revoke a subscriber, the publisher computes a share with the subscriber’s key and adds the share to the set of $T$ shares. This way the revoked subscriber only has a total of $T$ different shares and is unable to decrypt the symmetric group key. We will now dive in the specifics.

Like in Subsection 3.4, assume a subscriber $s_r$ is removed from $S_V$ and $p$ wants to revoke $s_r$ by distributing a new key $K_{\text{reset}}$ to the remaining valid subscribers in $S_V$. $p$ can use the threshold decryption scheme to encrypt the new key: $C_{\text{reset}} = \text{encrypt}(K_{\text{reset}},PK)$. We will see later that this is actually not sufficient, but for the sake of simplicity let’s assume for the moment that the reset key $K_{\text{reset}}$ is simply encrypted with $PK$. $p$ then initialises an empty set of decryption shares $DS$ and looks up $(s_r,k_r)$ in $SK_S$. Using $k_r$, $p$ computes a decryption share of the encrypted group key $ds_r = \text{decryptShare}(k_r,C_{\text{reset}})$ that the subscriber to revoke $s_r$ would also be able to compute since he also knows $k_r$. $p$ adds $ds_r$ to the set $DS$ and picks $T - 1$ decryption keys at random in $SK_P$. These keys are not known to any subscriber, only to the publisher $p$. For each key $k_j$ among these $T - 1$ keys, $p$ computes the associated decryption share $ds_j = \text{decryptShare}(k_j,C_{\text{reset}})$ and adds $ds_j$ to $DS$.

$DS$ now contains a total of $T$ shares. $T - 1$ of these shares can only be computed by $p$. The last share in $DS$ is $ds_r$ and can be computed by both $p$ and $s_r$. The publisher can now send a single multicast message to all subscribers: $C_{\text{reset}}|DS$.

Every valid subscriber $s_i$ still in $S_V$ that receives this message can compute one additional decryption share with the decryption key $k_i$ obtained in the joining phase described in Subsection 4.3: $ds_i = \text{decryptShare}(k_i,C_{\text{reset}})$. $s_i$ can then add $ds_i$ to $DS$ which has now a total of $T + 1$ shares. According to the threshold set by the publisher in Sub-
section 4.2, DS now contains enough shares for $s_i$ to be able to recover the group key: $K_{\text{reset}} = \text{combineShares}(DS, C_{\text{reset}})$.

On the other hand, even if $s_r$ can also compute his own share $ds_r = \text{decryptShare}(k_r, C_{\text{reset}})$, this share $ds_r$ is already included in $DS$. So $s_r$ only has a total of $T$ different shares which is not enough to recover the group key using the $\text{combineShares}()$ function.

The problem with this simple scheme is that from now on $ds_r$ has to be included in every rekey message, otherwise $s_r$ could decrypt future reset keys. This means that the entire system will need a full reset after $T$ revocations, the publisher will need to create a new polynomial and send a new partial decryption key to every valid subscriber. A potential solution as described in [3] is to alternate between two instances of the threshold decryption scheme. We took another approach and solved this issue by adding another layer of encryption: the ciphertext sent by the publisher is $C_{\text{reset}} = \text{encrypt}(AES(K_{\text{reset}}, K), PK)$ where $K$ is the current group key.

Every valid subscriber $s_i$ can use the $T$ shares in the same way to recover $AES(K_{\text{reset}}, K)$. Since $s_i$ is a valid subscriber, he also knows the current group key $K$ and can decrypt $K_{\text{reset}}$. $s_r$ still knows $K$ but, as we saw above, $s_r$ doesn’t have enough shares to recover $AES(K_{\text{reset}}, K)$. Now let’s assume $p$ wants to revoke someone else after having revoked $s_r$. $p$ chooses a new reset key $K_{\text{reset}2}$ and computes $C_{\text{reset}2} = \text{encrypt}(AES(K_{\text{reset}2}, K_{\text{reset}}), PK)$. $p$ doesn’t include $ds_r$ in the new set of shares $DS$. $s_r$ can then eavesdrop $C_{\text{reset}2} \parallel DS$, compute and add $ds_r$ to $DS$ to obtain $T$ shares. $s_r$ recovers $AES(K_{\text{reset}2}, K_{\text{reset}})$ but he cannot recover the latest key $K_{\text{reset}2}$ because it is encrypted with $K_{\text{reset}}$, the reset key that $s_r$ was not able to recover from the rekey message.

In the scenario above, $p$ only wanted to revoke a single subscriber $s_r$. But this scheme actually allows to revoke up to $T$ subscribers at the same time using the same message. Let $S_R$ be the set of $R \leq T$ subscribers that $p$ wants to revoke. $p$ chooses $K_{\text{reset}}$ and computes $C_{\text{reset}}$ in the same way. For each subscriber $s_{r_i}$ among the $R$ subscribers in $S_R$ to revoke, $p$ looks up the subscriber’s decryption key $k_{r_i}$ in $SK_S$, computes the decryption share $ds_{r_i} = \text{decryptShare}(k_{r_i}, C_{\text{reset}})$ and adds it to the initially empty set $DS$. Now $DS$ contains $R$ decryption shares. $p$ picks $T - R$ keys in $SK_P$, for each key $k_j$, $p$ computes the associated decryption share $ds_j = \text{decryptShare}(k_j, C_{\text{reset}})$ and adds $ds_j$ to $DS$. Now $DS$ contains a total of $T$ shares as expected. $p$ sends $C_{\text{reset}} \parallel DS$ to the subscribers.

Every subscriber $s_{r_i}$ in $S_R$ will be able to compute a different share $ds_{r_i}$ already present in $DS$ and thus won’t be able to recover $K_{\text{reset}}$. The case where $R = 1$ corresponds to the first scenario we saw. In the other extreme case where $R = T$, $p$ doesn’t need to include other shares from keys picked in $SK_P$ since $DS$ already contains $T$ shares. This allows $p$ to wait until either $T$ subscribers need to be revoked or $\delta t_2$ has elapsed since the first subscribed needed to be revoked (see Section 2).
5 Performance Evaluation

The implementation is still in progress. Once it is completed, we will be able to evaluate in production the performance of the two revocation schemes and compare the obtained metrics.

6 Conclusion

We will conclude after the performance evaluation.

References


