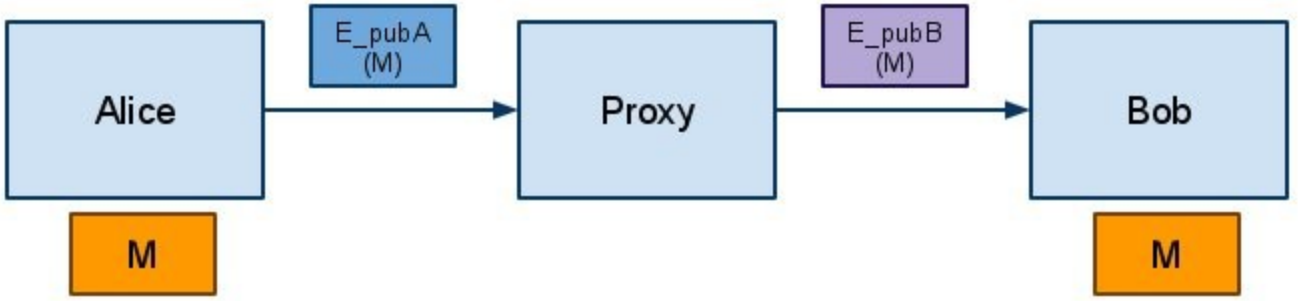
Proxy re-encryption

1. 使用 Bob 的公钥对 Alice 的明文进行加密
2. 将此密文发送给 Bob
3. Bob 对接收到的密文通过自己的私钥进行解密



代理重新加密允许代理将根据 Alice 的公钥加密出的密文转换成可以由 Bob 的私钥打开的密文

interactive and non-interactive

Interactive versions delegate access to a private key
non-interactive ones delegate access to a public key.

security properties

1. The proxy cannot see the plaintext unless it colludes with Bob.

2. The proxy cannot derive the secret key of Alice (even when the proxy colludes with Bob).

3. The scheme could be bi-directional (When Alice delegates to Bob, automatically Bob delegates to Alice. So, Alice and Bob need to have mutual trust for such schemes to work) or uni-directional (Alice can delegate to Bob without Bob having to delegate to her. Thus, the trust relationship between Alice and Bob does not need to be mutual).

4. The scheme could be transitive (Alice can delegate to Bob, and Bob can delegate to Tim in turn for example.) or non-transitive (Bob cannot delegate to Tim).

Shamir's Secret Share

(t,n)-threshold scheme

- 实现 (t,n) 门限方案的一个传统办法是，把这份文件的密钥拆成 C(n,t-1) 份，每个人持有 C(n-1,t-1) 份密钥。

- t 个点可以确定一个 (t-1) 次方程。于是我们可以构造一个 (t-1) 次方程，密码信息包含在其中。接着，把方程图像上的 n 个不同点的位置分别告诉 n 个人。这样，任意 t 个人在一起就可以凭借 t 个点的位置解出方程，但少一个人都不行。

Build this polynomial

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_{k-1}x^{k-1} \bmod(p)$$

Got K points

(x1,f(x1)),(x2,f(x2),…,(xk,f(xk)))

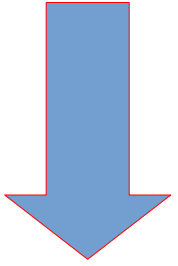
Also K key pairs

$$a_0 + a_1 x_1 + a_2 x_1 2 + \ldots + a_{k-1} x_1{}^{k-1} = f(x_1)$$

$$a_0 + a_1 x_2 + a_2 x_2 2 + \ldots + a_{k-1} x_2{}^{k-1} = f(x_2)$$

$$\vdots$$

$$a_0 + a_1 x_k + a_2 x_k 2 + \ldots + a_{k-1} x_k{}^{k-1} = f(x_k)$$



Lagrange 插值法或者范德蒙形式的线性方程组数值解方案

$$f(x) = \sum_{i=1}^{k} (f(x_i) \cdot \prod_{j \neq i}^{1 \leq j \leq k} \frac{(x - x_j)}{(x_i - x_j)})$$

NuCypher && Umbral

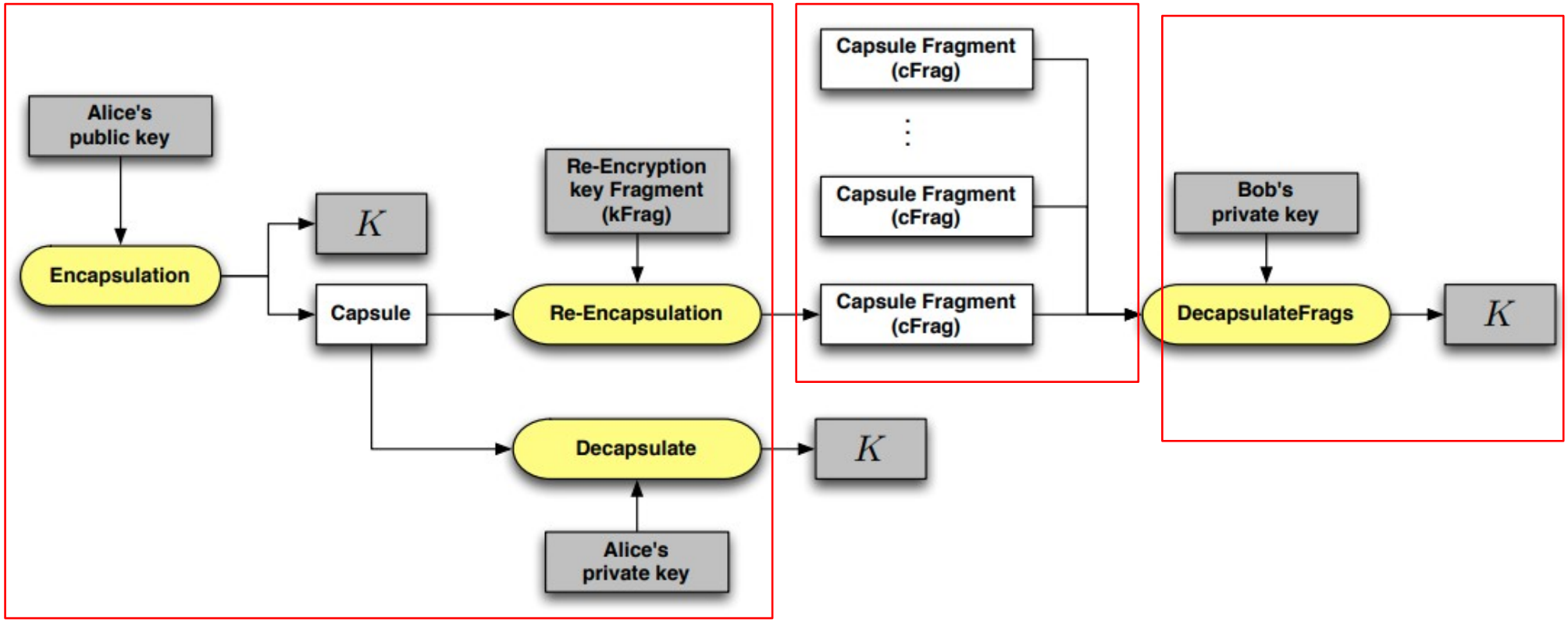


FIGURE 2. Main operation of Umbral KEM. Operations are shown in yellow, cryptographic keys in gray, and data in white

- Key Generation

```
from umbral import pre, keys, signing

# Generate Umbral keys for Alice.
alices_private_key = keys.UmbralPrivateKey.gen_key()
alices_public_key = alices_private_key.get_pubkey()

alices_signing_key = keys.UmbralPrivateKey.gen_key()
alices_verifying_key = alices_signing_key.get_pubkey()
alices_signer = signing.Signer(private_key=alices_signing_key)

# Generate Umbral keys for Bob.
bobs_private_key = keys.UmbralPrivateKey.gen_key()
bobs_public_key = bobs_private_key.get_pubkey()
```

- Key Generation
- Alice 端生成两组非对称加密的公私钥对
- Bob 端生成一组非对加密的公私钥对

- Encryption

```
# Encrypt data with Alice's public key.
plaintext = b'Proxy Re-Encryption is cool!'
ciphertext, capsule = pre.encrypt(alices_public_key, plaintext)

# Decrypt data with Alice's private key.
cleartext = pre.decrypt(ciphertext=ciphertext,
                        capsule=capsule,
                        decrypting_key=alices_private_key)
```

- Encryption

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

Capsule = (E , V , s)

- CheckCapsule

CheckCapsule($capsule$): On input a $capsule = (E, V, s)$, this algorithm examines the validity of the capsule by checking if the following equation holds:

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

$s = u + r \cdot H_2(E, V)$.

$E = g^r$ and $V = g^u$

$$g^s = g^u \cdot (g^r)^{H_2(E,V)}$$

$$= g^{u + r \cdot H_2(E,V)}$$

- Decapsulate

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.

$$
\begin{aligned}
K &= KDF((E \cdot V)^a) \\
&= KDF(g^{r \cdot a + u \cdot a}) \\
&= KDF(g^{a \cdot (r+u)}) \\
&= KDF((pk_A)^{r+u})
\end{aligned}
$$

- # Re-Encryption Key Fragments

When Alice wants to grant Bob access to open her encrypted messages, she creates re-encryption key fragments, or "kfrags", which are next sent to N proxies

```
# Alice generates "M of N" re-encryption key fragments (or "KFrags") for Bob.
# In this example, 10 out of 20.
kfrags = pre.generate_kfrags(delegating_privkey=alices_private_key,
                             signer=alices_signer,
                             receiving_pubkey=bobs_public_key,
                             threshold=10,
                             N=20)
```

Define a re-encryption key fragment $kFrag$ as the tuple $(id, rk, X_A, U_1, z_1, z_2)$.

(1) Sample random $x_A \in \mathbb{Z}_q$ and compute $X_A = g^{x_A}$
(2) Compute $d = H_3(X_A, pk_B, (pk_B)^{x_A})$. Note how d is the result of a non-interactive Diffie-Hellman key exchange between B's keypair and the ephemeral key pair (x_A, X_A). We will use this shared secret to make the re-encryption key generation of the scheme non-interactive.
(3) Sample random $t-1$ elements $f_i \in \mathbb{Z}_q$, with $1 \leq i \leq t-1$, and compute $f_0 = a \cdot d^{-1} \bmod q$.
(4) Construct a polynomial $f(x) \in \mathbb{Z}_q[x]$ of degree $t-1$, such that $f(x) = f_0 + f_1 x + f_2 x^2 + ... + f_{t-1} x^{t-1}$.
(5) Compute $D = H_6(pk_A, pk_B, pk_B^a)$
(6) Initialize set $KF = \emptyset$ and repeat N times:
    (a) Sample random $y, id \in \mathbb{Z}_q$
    (b) Compute $s_x = H_5(id, D)$ and $Y = g^y$.
    (c) Compute $rk = f(s_x)$
    (d) Compute $U_1 = U^{rk}$.
    (e) Compute $z_1 = H_4(Y, id, pk_A, pk_B, U_1, X_A)$, and $z_2 = y - a \cdot z_1$.
    (f) Define a re-encryption key fragment $kFrag$ as the tuple $(id, rk, X_A, U_1, z_1, z_2)$.
    (g) $KF = KF \cup \{kFrag\}$
(7) Finally, output the set of re-encryption key fragments KF.

- Re-Encryption

```
# Several Ursulas perform re-encryption, and Bob collects the resulting `cfrags`.
# He must gather at least `threshold` `cfrags` in order to activate the capsule.

capsule.set_correctness_keys(delegating=alices_public_key,
                             receiving=bobs_public_key,
                             verifying=alices_verifying_key)

cfrags = list()          # Bob's cfrag collection
for kfrag in kfrags[:10]:
  cfrag = pre.reencrypt(kfrag=kfrag, capsule=capsule)
  cfrags.append(cfrag)   # Bob collects a cfrag
```

it computes $E_1 = E^{rk}$ and $V_1 = V^{rk}$, and outputs the capsule fragment $cFrag = (E_1, V_1, id, X_A)$.

- Decryption by Bob

```
# Bob activates and opens the capsule
for cfrag in cfrags:
  capsule.attach_cfrag(cfrag)

bob_cleartext = pre.decrypt(ciphertext=ciphertext,
                            capsule=capsule,
                            decrypting_key=bobs_private_key)
assert bob_cleartext == plaintext
```

那么，在 Bob 端， Bob 知道哪些参数呢?

1. pkA ， pkB ， skB
2. cfags

$$(E_1, V_1, id, X_A)$$

(1) Compute $D = H_6(pk_A, pk_B, pk_A^b)$
(2) Let $S = \{s_{x,i}\}_{i=1}^{t}$, for $s_{x,i} = H_5(id_i, D)$. For all $s_{x,i} \in S$, compute:

$$\lambda_{i,S} = \prod_{j=1, j\neq i}^{t} \frac{s_{x,j}}{s_{x,j} - s_{x,i}}$$

(3) Compute the values:

$$E' = \prod_{i=1}^{t} (E_{1,i})^{\lambda_{i,S}} \qquad V' = \prod_{i=1}^{t} (V_{1,i})^{\lambda_{i,S}}$$

(4) Compute $d = H_3(X_A, pk_B, X_A^b)$. Recall that d is the result of a non-interactive Diffie-Hellman key exchange between B's keypair and the ephemeral key pair (x_A, X_A). Note also that the value X_A is the same for all the $cFrags$ that are produced by re-encryptions using a $kFrag$ in the set of re-encryption key fragments KF.
(5) Finally, output the symmetric key $K = \mathsf{KDF}((E' \cdot V')^d)$.

(1) Compute $D = H_6(pk_A, pk_B, pk_A^b)$

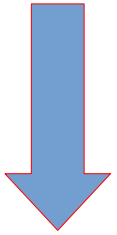
Alice

(5) Compute $D = H_6(pk_A, pk_B, pk_B^a)$

Bob

(1) Compute $D = H_6(pk_A, pk_B, pk_A^b)$

$$pk_B{}^a = g^{b \cdot a} = g^{a \cdot b} = pk_A{}^b$$



$$D_{Alice} = D_{Bob}$$

(2) Let $S = \{s_{x,i}\}_{i=1}^{t}$, for $s_{x,i} = H_5(id_i, D)$. For all $s_{x,i} \in S$, compute:

$$\lambda_{i,S} = \prod_{j=1, j\neq i}^{t} \frac{s_{x,j}}{s_{x,j} - s_{x,i}}$$

Alice

(a) Sample random $y, id \in \mathbb{Z}_q$

(b) Compute $s_x = H_5(id, D)$ and $Y = g^y$.

(c) Compute $rk = f(s_x)$

$$f_0 + f_1 s_{x_1} + f_2 s_{x_1}{}^2 + \ldots + f_{t-1} s_{x_1}{}^{t-1} = rk_1$$

$$f_0 + f_1 s_{x_2} + f_2 s_{x_2}{}^2 + \ldots + f_{t-1} s_{x_2}{}^{t-1} = rk_2$$

$$\vdots$$

$$f_0 + f_1 s_{x_{t-1}}{}^{t-1} + f_2 s_{x_{t-1}}{}^{t-1} + \ldots + f_{t-1} s_{x_{t-1}}{}^{t-1} = rk_{t-1}$$

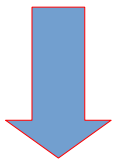
$$f_0 + f_1 s_{x_1} + f_2 s_{x_1}{}^2 + \ldots + f_{t-1} s_{x_1}{}^{t-1} = rk_1$$

$$f_0 + f_1 s_{x_2} + f_2 s_{x_2}{}^2 + \ldots + f_{t-1} s_{x_2}{}^{t-1} = rk_2$$

$$\vdots$$

$$f_0 + f_1 s_{x_{t-1}}{}^{t-1} + f_2 s_{x_{t-1}}{}^{t-1} + \ldots + f_{t-1} s_{x_{t-1}}{}^{t-1} = rk_{t-1}$$

$$f(x) = \sum_{i=1}^{k} (f(x_i) \cdot \prod_{j \neq i}^{1 \leq j \leq k} \frac{(x - x_j)}{(x_i - x_j)})$$



$$f(x) = \sum_{i=1}^{t-1} (rk_i \cdot \prod_{j \neq i}^{1 \leq j \leq t-1} \frac{(x - s_{x_j})}{(s_{x_i} - s_{x_j})})$$

(3) Compute the values:

$$E' = \prod_{i=1}^{t} (E_{1,i})^{\lambda_{i,S}} \qquad V' = \prod_{i=1}^{t} (V_{1,i})^{\lambda_{i,S}}$$

Re-Encryption

it computes $E_1 = E^{rk}$ and $V_1 = V^{rk}$, and outputs the capsule fragment $cFrag = (E_1, V_1, id, X_A)$.

$$E_{1,i} = E^{rk_i} \qquad V_{1,i} = V^{rk_i} \quad \Rightarrow \quad E' = \prod_{i=1}^{t} (E)^{rk_i \cdot \lambda_{i,S}} \qquad V' = \prod_{i=1}^{t} (V)^{rk_i \cdot \lambda_{i,S}}$$

(4) Compute $d = H_3(X_A, pk_B, X_A^b)$. Recall that d is the result of a non-interactive Diffie-Hellman key exchange between B's keypair and the ephemeral key pair (x_A, X_A). Note also that the value X_A is the same for all the $cFrags$ that are produced by re-encryptions using a $kFrag$ in the set of re-encryption key fragments KF.

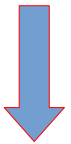
Alice

$$d = H_3(X_A, pk_B, (pk_B)^{x_A})$$

Bob

$$d = H_3(X_A, pk_B, X_A^b)$$

$$(pk_B)^{x_A} = g^{b \cdot x_A} = {X_A}^b$$



$$d_{Alice} = d_{Bob}$$

(5) Finally, output the symmetric key $K = \mathsf{KDF}((E' \cdot V')^d)$.

$$
\begin{aligned}
(E' \cdot V')^d &= \Big(\prod_{i=1}^{t} (E_{1,i})^{\lambda_{i,s}} \cdot \prod_{i=1}^{t} (V_{1,i})^{\lambda_{i,s}}\Big)^d \\
&= \Big(\prod_{i=1}^{t} (E_{1,i} \cdot V_{1,i})^{\lambda_{i,s}}\Big)^d \\
&= \Big(\prod_{i=1}^{t} (E \cdot V)^{rk_i \cdot \lambda_{i,s}}\Big)^d \\
&= (E \cdot V)^{d \cdot \sum_{i=1}^{t} (rk_i \cdot \lambda_{i,s})}
\end{aligned}
$$

$$f(x) = \sum_{i=1}^{t-1} (rk_i \cdot \prod_{1 \le j \le t-1, j \ne i} \frac{(x - s_{x_j})}{(s_{x_i} - s_{x_j})})$$

$$\lambda_{i,S} = \prod_{j=1, j \ne i}^{t} \frac{s_{x,j}}{s_{x,j} - s_{x,i}}$$

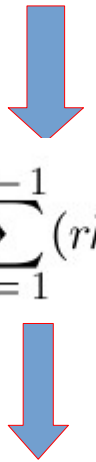
$$f_0 = \sum_{i=1}^{t-1} (rk_i \cdot \lambda_{i,S})$$

Alice

$$f_0 = a \cdot d^{-1} \ mod \ q$$

Bob

$$f_0 = \sum_{i=1}^{t-1} (rk_i \cdot \lambda_{i,S})$$



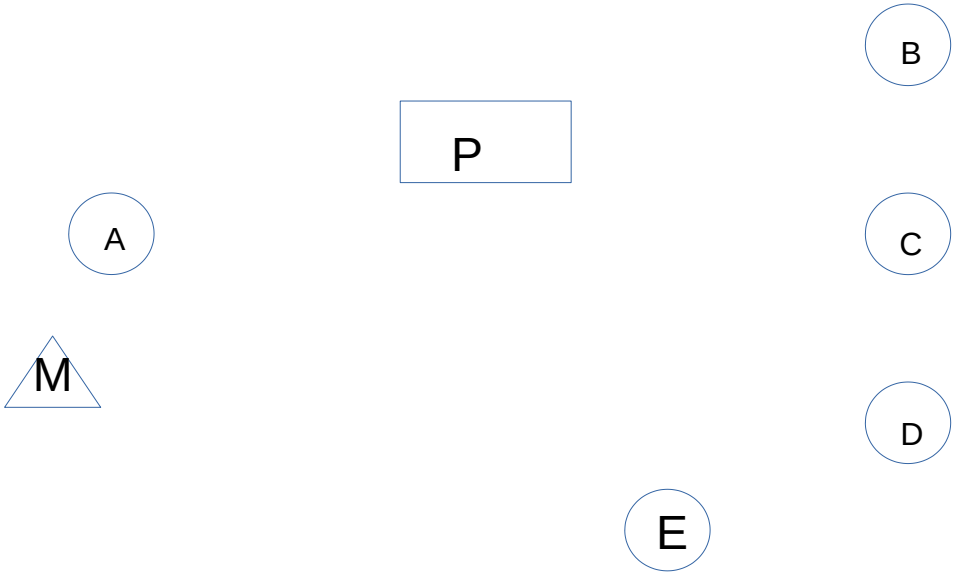
$$a = d \cdot \sum_{i=1}^{t-1} (rk_i \cdot \lambda_{i,S})$$

$$K_{Alice} = KDF((E \cdot V)^a) = KDF((E' \cdot V')^d) = K_{Bob}$$

PRE 和传统 PKE 对比

仅考虑单一代理。假设一个数据分享者 A 有 N 份文件需要分享（对应 N 个加密密钥），网络中有 M 个潜在的数据访问者（用户）。假设文件平均命中率是 x，即在一定周期内文件被平均请求的次数。

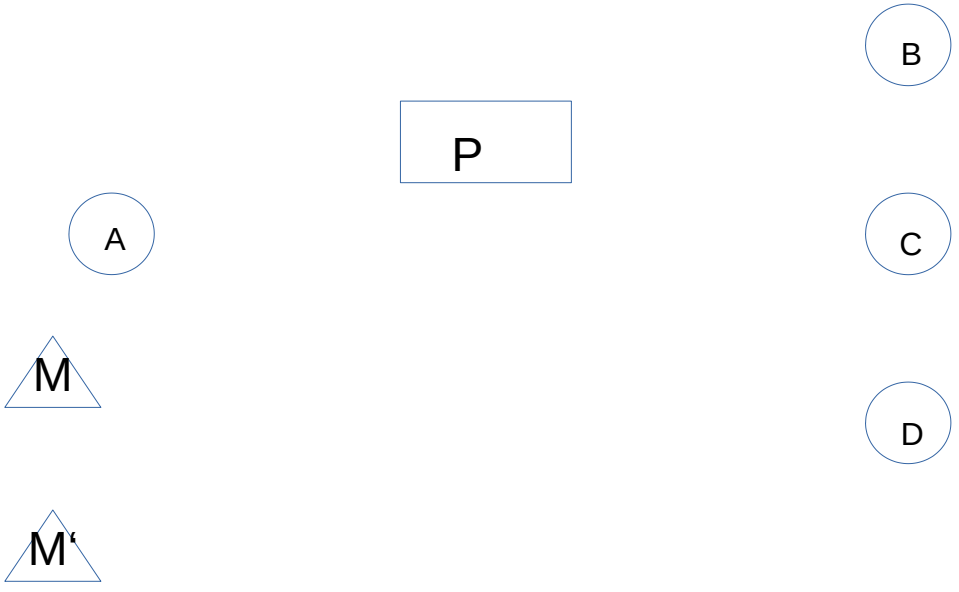

Bob
proxy
Alice
Cathy
M
David



新增加一名用户：

1. 对于新增的用户来说，他需要 A 对他进行授权，所以需要在 A 端计算生成新的 rk 。但是对于本端的所有同类的文件来说，是不需要进行其他操作的。

2. 但是对于不同类的文件来说，由于他们所使用的公私钥对是不同的，所以相对应的 rk 也是不同的，所以对于不同类的文件也需要重新计算新的 rk ，即本端需要进行 N 次计算



新增加一类文件：

1. 对于新增的一类文件来说，他需要新的公私钥对进行加密，所以需要生成新的 rk
2. 同时 rk 是与对端的用户相关的，所以新增加一类文件就需要在本端进行 M 次 rk 的计算

在静态场景下，传统做法有一个较长的初始化过程。在实际运用中，重加密方法需要较大的计算需求，但节约了一部分存储，两种方式差距不是特别明显。但一般情况下，文件系统的用户和内容都是动态的，此时重加密做法的优势明显。

具体来说，在增加用户时，传统做法需要本地对已有的所有加密密钥进行加密，可能花费很长时间。在增加文件时，重加密做法下，数据分享者只需要使用自己的公钥去加密对称密钥即可，而传统做法需要做 M 次非对称加密。更新文件加密密钥时，重加密做法的数据分析者只需要使用自己的公钥去加密对称密钥即可（无需再次生成重加密密钥），而传统做法需则需要重新执行初始化过程。另外，服务端拥有更强的计算能力，重加密算法把这些计算过程让服务端更多承担，尽量减少客户端的计算，符合架构设计原理。最后，重加密算法的弹性更好，一般来说， x 不会是一个太大的数，当文件访问稀疏时，在一开始就执行多次可能无用的加密是巨大的浪费。

1. 对于同一类明文，我们使用的相同的对称秘钥进行加密，与对端的用户无关。
2. 对于对端的不同用户，我们需要在本端针对不同的用户生成不同重加密秘钥，与本端文件无关。
3. 在做完授权的部分后，本端将不再参与其他过程，解放客户端的压力，使之成为一个非交互性的系统。