Secure Keyless Multi-Party Storage Scheme

Pascal Lafourcade[1][0000−0002−4459−511X], Lola-Baie Mallordy[1][0009−0005−1308−4412], Charles Olivier-Anclin[1,2,4][0000−0002−9365−3259], and Léo Robert[3][0000−0002−9638−3143]

[1] Université Clermont Auvergne, LIMOS, CNRS, Clermont-Ferrand, France
[2] be ys Pay
[3] Université de Picardie Jules Verne, Amiens, France
[4] LIFO, Université d'Orléans, INSA Centre Val de Loire, Bourges, France

Abstract. Using threshold secret sharing, we propose a solution tailored for *forgetful* clients (*i.e.,* not required to keep any cryptographic secret) while accommodating the dynamic nature of multi-cloud deployments. Furthermore, we delegate the computation and distribution of shares to an intermediate server (proxy), effectively minimizing the client workload. We propose two variants of a keyless, space-efficient multi-cloud storage scheme named KAPRE and KAME. Our solution KAPRE requires less communications and computations, while KAME preserves data confidentiality against a colluding proxy. Our protocols offer robust guarantees for data integrity, and we demonstrate the proxy's ability to identify and attribute blame to servers responsible for sending corrupted shares during data reconstruction. We establish a comprehensive security model and provide proofs of the security properties of our protocols. To complement this theoretical analysis, we present a proof-of-concept to illustrate the practical implementation of our proposed scheme.

1 Introduction

Cloud storage services like Amazon S3, OVHcloud, or Google Drive are increasingly popular, both among companies and users to store large amounts of sensitive data. However, handing data over to a single third party often raises availability, integrity and confidentiality issues [28, 19]. The user does not want to neither lose access to its data in case of server failure, nor retrieve data that has been altered in any way (maliciously or not). It also should not have to reveal any of its sensitive content to the *Cloud Storage Provider* (CSP). Multi-cloud, the simultaneous use of several cloud services, counterbalances data centralisation [8]. It introduces redundancy in the stored data, hence providing availability and integrity even in the case of several server failures (*e.g.,* as in Strasbourg, France where several OVH servers were destroyed in a fire).

Numerous solutions exist to ensure data confidentiality within multi-cloud architecture. The seminal work of Shamir on secret sharing [29] showed an elegant solution to split a file into shares and to distribute them to a set of CSPs. It provides information-theoretic secrecy [22], meaning that no party can learn anything about the content of the data without the cooperation of the others. A perk of secret sharing is that its security relies on the need of other parties' cooperation rather than on the knowledge of a cryptographic secret. Hence, the security of the data does not rely on any cryptographic secret. Otherwise, this can lead to great loss, as in cryptocurrencies where losing the secret key results in the inability to access the money.

Despite its security, secret sharing is memory-consuming, as the shares must be as large as the secret. This means that storing a secret S of size $|S|$ requires $|S|$ storage space for each CSP, which scales poorly. Memory-efficient algorithms as Rabin's *Information Dispersal Algorithm* [25] (IDA) produce shares of optimal size, which depends on the number of shares needed to reconstruct the initial data. They achieve high fault-tolerance and small buffer size, but they were often not designed to provide confidentiality [26].

In multi-cloud setting, delegating the sharing and reconstruction of its data to a third party can benefit the user. For the user, it can be a hassle to communicate with each CSP, distribute its data and check the CSPs availability for download. This task is even more complicated if the user wants to make sure its data has not been altered during recovery and detect any corrupted share. All of this can be delegated to a proxy (a particular CSP with additional tasks). However, if the client does not trust the CSPs with its data content, it should not have to trust the proxy either. A solution can be to encrypt the data before sending it to the proxy with a block cipher [9]. But, as the key is needed for data retrieval, this shifts the problem from protecting the files to protecting the key: if the key is compromised, so is the data.

We tackle the problem of storage in multi-cloud infrastructure, where the client delegates most of the computations to an untrusted proxy, which handles the communications with the CSPs. The data must remain: (1) confidential, (2) available even if some of the CSPs are unavailable, (3) unaltered, any modification must be detected, and (4) liable to the entity that produce a fault. We consider a forgetful user, so no security values are stored between the upload and the download phases (*e.g.,* long-term keys or hashes for integrity). The keyless design not only simplifies the overall storage infrastructure but also eliminates the risk of key exposure, reducing the attack surface and enhancing the overall security.

Our Contributions. We design a *Keyless Multi-Party Storage* (KMPS) scheme, depicted in Fig. 1, to split client's data through $n+1$ shareholders (one proxy and n providers) such that only $k \leq n+1$ of them are needed for data recovery. We enforce data confidentiality with regards to the proxy and up to $k-1$ colluding CSPs. We prove the size of the shares is almost optimal ($|S|/k+\lambda$ for initial data of size $|S|$ and the security parameter λ). We expect a setup independent of any long-term key, nor any additional value to check data integrity. Thanks to the proxy, we minimise the computations and communication overhead on client-side. To our knowledge, there is no multi-cloud storage protocol delegating all the communications and most of computations to a proxy with a keyless client, while guaranteeing data's integrity without storing any additional value.

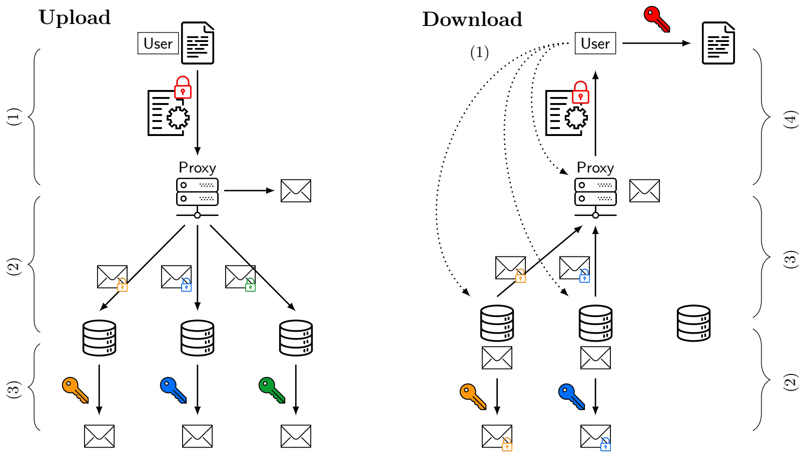

Fig. 1. Overview of KMPS upload and download phases for $k = 3, n + 1 = 4$ (Here we consider that the proxy also holds a share, just as the CSPs).

KMPS Threat model: Let k be the secret sharing threshold and n the number of providers. We define a generic model for a KMPS scheme formalizing strong security properties, namely:

$N-$**collusion-secrecy:** a proxy colluding with N providers cannot learn anything on the client's data. We assume the proxy follows the protocol during the upload.

Provider-secrecy: a collusion of $k - 1$ malicious providers cannot learn anything on the client's data.

User-Integrity: any collusion of the proxy with less than $k - 2$ malicious providers cannot alter the data without the user noticing. The proxy is assumed to follow the protocol during upload.

Accountability: if a share has been corrupted, the proxy will detect it and blame the corresponding CSP with overwhelming probability.

We leave open the design of a system that allows a malicious proxy to handle user data during upload, without compromising security and integrity.

Two protocols: We propose two KMPS variants, KAPRE (*Keyless Archivage with Proxy REencryption*) relying on homomorphic proxy re-encryption [11] and KAME (*Keyless Archivage with Multikey Encryption*) relying on homomorphic multikey encryption [24]. Both rely on Shamir's secret sharing [29] and an IDA [25]. Our two solutions balance in between efficiency and security. Indeed, KAPRE is round optimal in communications but some trust should be kept toward the proxy while KAME requires less trust in the proxy, but has an additional round of communication. We formally model and prove the security of our protocols. We prove that both schemes provide accountability and user-integrity. As for data's secrecy, we show that KAPRE has $0-$collusion secrecy and provider-secrecy while KAME achieves $(k - 2)-$collusion secrecy.

Implementation: We provide an implementation of our two solutions. This allows us to give an overview of the efficiency, and show a linear complexity on the user-side for both schemes.

External authentication. Our security model does not consider user's authentication. In download, any user could retrieve another user's data so we assume authentication throughout the protocol. Hence, our schemes are compatible with any pre-existing identity system such as mutual TLS [27] or other solutions [18].

Related Work. Usually, multi-cloud storage protocols security relies on the user encrypting its data before storing it, often with a symmetric encryption for cost efficiency [34, 23, 17]. This preserves data's privacy against the CSPs. The user must keep long-term keys, which shifts the concern of protecting data to protect the keys. To tackle this issue, the authors in [20] use a *Credentials-less Permission Mechanism.* While the user has access to its key, it uses it to actively block any demand for a new key generation. Hence, if the user loses its key, there is no one to block its request for a new one. However, this system can be broken if the attacker guesses whenever the user loses its key, and outruns the user in its request for a new key. Filecoin [4] is another example of using long-term keys. Filecoin is a decentralized storage network using blockchain to provide privacy, proof of spacetime and replication. Yet, the use of blockchain makes their protocol quite slow and prevents from deleting data. We prefer a solution which does not require the user to keep any cryptographic secret.

Some multi-cloud storage protocols use secret sharing to ensure confidentiality without permanent keys [9, 19]. However, the secret sharing must be done by a trusted party, *i.e.,* the user itself, hence no delegation is possible. Additionally, secret sharing is not memory efficient. The authors of [19] proposed an improved variant of Shamir's secret sharing achieving a storage ratio of about 2 to 4 times the initial data size, this is still far for the optimal ratio of n/k which we achieve. Krawczyk proposed a solution to obtain small shares by pairing an information-theoric secret sharing with erasure codes [14]. His scheme is used in both [8, 9]. Their protocol encrypts the user's data with a symmetric encryption, and distributes it with an erasure code for each CSP. The key is shared with a secret sharing. However, in these schemes all the sharing must be done by the user itself or a fully trusted third-party, when our solution relieves the user from these tasks. When multi-cloud protocols involve a proxy, it is usually a trusted-party [33, 23, 35]. Some protocols only consider server failure in their threat model [31, 21]. They aim for efficiency and data's availability, and do not consider confidentiality or trust issues. Some protocols do consider data's confidentiality w.r.t. individual cloud providers, but do not consider colluding CSPs [32]. Our model addresses all of these cases, as we consider the proxy as trustworthy as any other CSP.

Regarding integrity, most storage protocols also rely on keeping long-term values as hashes [34, 17], or long-term keys by signing the data [19, 35]. For example, the solution in [35] uses blockchain to provide integrity, which makes it quite slow. Also, the user must keep a secret key to sign its data, which we want

	Proxy	CSPs	Coll.	Keyless	Integrity
[34]	✗	✓	–	✗	Merkle Tree
[20]	✓	✓	✓	✗	✗
[32]	✗	✗	–	✗	✗
[23, 17]	✓	✓	✗	✗	Hashes [17]
[19, 9, 10]	✗	✓	–	✓	Signature [19],Quorum [9]
[33]	✓	✗	✗	✗	✗
[35]	✓	✓	✗	✗	Signature
KAPRE	✓	✓	✗	✓	auth. encryption/PRF
KAME	✓	✓	✓	✓	auth. encryption/PRF

Fig. 2. Comparison of existing storage schemes (Proxy: the file splitting is delegated to a proxy, CSP: confidentiality is preserved against CSPs colluding, Coll: confidentiality is preserved against the proxy colluding with CSPs, Keyless: the client does not hold permanent key, Integrity: whether/how data integrity is checked).

to avoid. We summarize all of these properties in Fig. 2, and compare previous works with ours. Finally, most of these works [34, 9, 33, 23, 17, 35] do not formally prove the security of their solutions, which we do in our long version [15].

Outline. In Section 2, we present a technical overview of our scheme. In Section 3, we present a generic model for KMPS schemes and their security properties. In Section 4, we propose two instantiations which differ in upload, and a common download. Security is discussed in Section 5, we only provide sketch proofs for space concern, the full proofs are available in [15]. In Section 6, we discuss our implementation for both schemes. Section 7 concludes and additional material is provided in Appendix A.

2 Technical Overview

Notations. We write negl for any negligible function in the security parameter λ. Sampling an element x uniformly from a set S is denoted $x \leftarrow\!\!\$\ S$. The set of possible outputs for any given inputs is denoted as $[\mathsf{Alg}(\cdot)]$. Also, by $\mathsf{P}\langle \mathsf{E}_1(i_1), \mathsf{E}_2(i_2)\rangle$, we denote the protocol P played between parties E_j, taking as input i_j. We denote concatenation of elements by $||$. By $\mathcal{C}$, we denote the challenger of a security experiment and by $\mathcal{A}$ the adversary, both being probabilistic polynomial-time algorithms.

Overview. To achieve small data shares, the user encrypts its data with an authenticated symmetric encryption [6]. The proxy distributes the encrypted data among the clouds with an IDA. Then, the key is shared between the clouds, with secret sharing as the data's confidentiality relies on the secrecy of the key.

Definition 1 (Secret Sharing [29]). *A* (k, n) secret sharing *scheme* SS *is given by:*

$\mathsf{SS.Split}(m, k, n) \rightarrow (s_1, \ldots, s_n)$*: on input a secret* m*, returns the shares* $(s_1, \ldots, s_n)$ *according to* n *and* k*,.*

```
$\mathsf{SS.Split}(m \in \mathbb{Z}_p, n, k)$
1 : $x_1, \ldots, x_n \leftarrow\!\!\$\ \mathbb{Z}_p^*$
2 : all distinct
3 : $a_1, \ldots, a_{k-1} \leftarrow\!\!\$\ \mathbb{Z}_p$
4 : $\forall i,\ y_i = m + \sum_{j=1}^{k-1} a_j x_i^j$
5 : **return** $\{x_i, y_i\}_{i=1}^{n}$
```

```
$\mathsf{SS.Rec}(k, (x_1, y_1), \ldots, (x_l, y_l))$
1 : **if** $l < k$: **return** $\perp$
2 : **for** $i \in [\![1, k]\!]$:
3 :     $\ell_i = \prod_{j \neq i, j=1}^{k} \frac{-x_j}{x_i - x_j}$
4 : **return** $\sum_{i=1}^{k} y_i \ell_i$
```

Fig. 3. Description of Shamir's secret sharing algorithms.

$\mathsf{SS.Rec}(k, s_1, \ldots, s_j) \rightarrow m$: *reconstructs the secret from the shares given that* $j \geq k$ *such that* $\mathsf{SS.Rec}(k, \mathsf{SS.Split}(m, k, n)) = m$.
It must achieve perfect secrecy (Fig. 8, Appendix A).

We use Shamir's secret sharing described in Fig. 3, which perfect secrecy is shown in [29]. As we want to delegate the sharing to the proxy, the user encrypts the key with an homomorphic encryption, such that the proxy can compute a secret sharing on the encrypted key resulting on encrypted shares of the key. Indeed as we want our setup to be independant of any long-term key, the CSPs need to store the key shares in plaintext. The key point of our constructions is using an asymmetric encryption $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ that commutes with a secret sharing. We model this property in the following definition:

Definition 2. *We say that* PKE *commutes with a secret sharing* SS *if it verifies* $\forall(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.KeyGen}(\lambda)$, $\forall m \in \{0,1\}^*, \forall k, n \in \mathbb{N}$ *such that* $k \leq n$ *and*
$\forall(s_1, \ldots, s_n) \in [\mathsf{SS.Split}(\mathsf{PKE.Enc}(m, \mathsf{pk}), n, k)]$, *we have*
$\forall \mathcal{I} \subseteq [\![1, n]\!], |\mathcal{I}| = k,\ \mathsf{SS.Rec}(k, \{\mathsf{PKE.Dec}(s_i, \mathsf{sk})\}_{i \in \mathcal{I}}) = m$.

As Shamir's secret sharing produces shares of a secret m which are linear relations in m and the random coefficients a_i, any additively homomorphic encryption scheme (integer scalar multiplication is given) PKE commutes with it by computing encrypted shares as:
$\mathsf{PKE.Enc}(m, \mathsf{pk}) + \sum_{j=1}^{k-1} \mathsf{PKE.Enc}(a_j, \mathsf{pk}) x_i^j \in \left[\mathsf{PKE.Enc}(y_i, \mathsf{pk})\right]$.
Reciprocally, recovering the encrypted secret is given by:
$\sum_{i=1}^{k} \mathsf{PKE.Enc}(y_i, \mathsf{pk}) \ell_i \in \left[\mathsf{PKE.Enc}\left(\sum_{i=1}^{k} y_i \ell_i = m, \mathsf{pk}\right)\right]$.
The shares of the key cannot be accessible to the proxy. This is where our two protocols differ: our first solution KAPRE uses proxy re-encryption [24, 12] to tackle this issue, while KAME uses multikey encryption [11, 16] (see Appendix A). At the end, our two protocols result in the same state, and have a common download phase. During download, the proxy checks the shares integrity using commitments computed by the user from key-homomorphic pseudorandom function (PRF) families [30, 3, 13] (see Appendix A). This avoids sending the user any corrupted data. As the proxy receives shares that have been re-randomized, usual methods like keeping hashes cannot be used to provide integrity.

3 Generic Model

Our Keyless Multi-Party Storage (KMPS) scheme involves three types of party: a User interacting with a Proxy which interacts with n *Cloud Storage Providers* CSP_i, for $i \in [\![1, n]\!]$. There are two phases: an *upload* where the user stores some data m interacting only with the proxy, and a *download* where at least $k-1$ of the CSPs cooperate with the proxy (gathering a total of k shares) to send back m to the user.

3.1 Multi-Party Storage scheme

We give in Fig. 5 a diagram describing an upload and a download.

Definition 3 (Keyless Multi-Party Storage). *A* Keyless Multi-Party Storage scheme KMPS *is a tuple* $\mathsf{KMPS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Transform}, \mathsf{Distrib}, \mathsf{Open}, \mathsf{Designate}, \mathsf{Hide}, \mathsf{Merge}, \mathsf{Recover})$ *of probabilistic polynomial time algorithms with:*

$\underline{\mathsf{Setup}(\lambda, n, k) \to \mathsf{param}}$*: sets the global parameters.*

The parameters param*,* n *and* k *are implicit parameters of all algorithms.*

$\underline{\mathsf{KeyGen}(\lambda) \to (\mathsf{pk}, \mathsf{sk})}$*: returns a key pair* $(\mathsf{pk}, \mathsf{sk})$ *according to* λ*.*

$\boxed{\textbf{\textit{Upload}}}$

$\underline{\mathsf{Transform}(m, \mathsf{pk}_0, \ldots, \mathsf{pk}_n) \to \mathsf{com}_\mathsf{U}, \mathsf{parts}_\mathsf{U}}$*: on input a message* m*, the proxy and providers' public keys, outputs* $\mathsf{parts}_\mathsf{U}$ *and a commitment* com_U *to* m*.*

$\underline{\mathsf{Distrib}(\mathsf{parts}_\mathsf{U}) \to \{h_i\}_{i=0}^n}$*: produces shares* $\{h_i\}_{i=0}^n$ *from* $\mathsf{parts}_\mathsf{U}$*,* h_0 *being the proxy's share and* $\{h_i\}_{i=1}^n$ *the providers shares.*

$\underline{\mathsf{Open}}$*: This is either an algorithm* $\mathsf{Open}(h_i, \mathsf{sk}_i) \to s_i$ *or a protocol between the providers and the proxy taking as input each party's shares* $\{h_i\}_{i=0}^n$ *and each party's secret keys* $\{\mathsf{sk}_i\}_{i=0}^n$*, outputting* $\{s_i\}_{i=0}^n$ *.*

We write $\underline{\mathsf{Upload}(m) \to \{s_i\}_{i=0}^n, \mathsf{com}_\mathsf{U}}$ *for the concatenation of the three previous algorithms with keys as implicit input and* Dec *executed for each share.*

$\boxed{\textbf{\textit{Download}}}$

$\underline{\mathsf{Designate}(\mathsf{pk}_1, \ldots, \mathsf{pk}_k) \to \mathsf{com}_\mathsf{D}, \mathsf{state}_\mathsf{U}, \{\mathsf{nc}_i\}_{i=1}^k}$*: The user computes for each provider involved in download an encryption* nc_i *of a nonce* n_i *under its public key* pk_i*, and a commitment* com_D *to the nonces. It also returns in* $\mathsf{state}_\mathsf{U}$ *the values needed for the final recovery.*

$\underline{\mathsf{Hide}(s_i, \mathsf{nc}_i, \mathsf{sk}_i) \to s'_i}$*: a provider or the proxy encrypts its share* s_i *with* nc_i*.*

$\underline{\mathsf{Merge}(\mathsf{com}_\mathsf{U}, \mathsf{com}_\mathsf{D}, \{s'_i\}_{i=0}^{k-1}) \to (\mathsf{parts}_\mathsf{D}, \mathcal{I})}$*: if there are invalid shares,* $\mathsf{parts}_\mathsf{D} = \perp$ *and* $\mathcal{I}$ *contains their indexes. Otherwise,* $\mathcal{I} = \perp$ *and* $\mathsf{parts}_\mathsf{D}$ *is computed from the shares.*

$\underline{\mathsf{Recover}(\mathsf{parts}_\mathsf{D}, \mathsf{state}_\mathsf{U}) \to m/\perp}$*: outputs* m *from* $\mathsf{parts}_\mathsf{D}$*,* $\mathsf{state}_\mathsf{U}$ *and the nonces,* $\perp$ *if the data has been corrupted.*

Note that between upload and download, the algorithms run by the user $\mathsf{Transform}$, $\mathsf{Designate}$ and $\mathsf{Recover}$ only require the public keys of the CSPs and the proxy (which can be updated between the two phases). This is what we mean by keyless: the user do not keep any cryptographic secret between these two phases.

3.2 KMPS Security Model

We give formal definitions of the security properties for a secure KMPS scheme.

Correctness. The composition of the algorithms of a KMPS scheme must allow to recover the original data from the $n+1$ shares (n for the providers and one for the proxy).

Definition 4 (Correctness). *A* KMPS *scheme is* correct *if it verifies the following* $\forall \lambda \in \mathbb{N}, \forall n \in \mathbb{N}^*$*,* $\forall m \in \{0,1\}^*$*:* $\forall k$ s.t. $k \leq n+1$, $\forall \mathsf{param} \in [\mathsf{Setup}(\lambda, n, k)]$,
$\forall i \in [\![0,n]\!]$, $\forall (\mathsf{pk}_i, \mathsf{sk}_i) \in [\mathsf{KeyGen}(\lambda)]$, $\forall \mathcal{I} \subseteq [\![0,n]\!]$ s.t. $|\mathcal{I}| = k$,
$\forall \mathsf{com}_\mathsf{U}, \mathsf{parts}_\mathsf{U} \in [\mathsf{Transform}(m, \{\mathsf{pk}_i\}_{i=0}^n)]$, $\forall \mathsf{com}_\mathsf{U}, \{h_i\}_{i=0}^n \in [\mathsf{Distrib}(\mathsf{parts}_\mathsf{U})]$,
$\forall \mathsf{com}_\mathsf{D}, \mathsf{state}_\mathsf{U}, \{\mathsf{nc}_i\}_{i\in\mathcal{I}} \in [\mathsf{Designate}(\{\mathsf{pk}_i\}_{i\in\mathcal{I}})]$,
$\forall i \in I, h'_i \in [\mathsf{Hide}(\mathsf{Open}(h_i, \mathsf{sk}_i), \mathsf{nc}_i, \mathsf{sk}_i)]$,
$\forall (\mathsf{parts}_\mathsf{D}, \mathcal{I}) \in [\mathsf{Merge}(\mathsf{com}_\mathsf{U}, \mathsf{com}_\mathsf{D}, \{h_i\}_{i\in I})]$, $\mathsf{Recover}(\mathsf{parts}_\mathsf{D}, \mathsf{state}_\mathsf{U}) = m$.

Data's Secrecy. We model the data's secrecy properties through indistinguishability games [30]. The adversary chooses two messages, and tries to guess which one is being uploaded and downloaded by the challenger. There are two properties depending on the adversary's capacities. The first one, *provider-secrecy*, considers the proxy is trusted. The adversary models $k-1$ colluding providers while the challenger simulates the other parties (game $\mathsf{Exp}_\mathcal{A}^{\mathsf{CSP}}$ in Fig. 4).

Definition 5 (Provider-Secrecy). *A* KMPS *scheme is* Provider-secret *if for any adversary* $\mathcal{A}$ *we have* $\mathsf{Adv}_\lambda^{\mathsf{CSP}}(\mathcal{A}) := \left|Pr\left[\mathsf{Exp}_\mathcal{A}^{\mathsf{CSP}}(\lambda) = 1\right] - 1/2\right| \leq \mathsf{negl}$.

The second property N*-collusion-secrecy*, depicted in game $\mathsf{Exp}_{\mathcal{A},N}^{\mathsf{Coll}}$ (Fig. 4), considers the proxy colluding with N CSPs as the adversary. The challenger plays the role of the user and the remaining $n-N$ honest CSPs. To model the fact that the proxy follows the protocol, the challenger executes the proxy's algorithms but reveals all intermediate values to the adversary. Note that $0-$collusion-secrecy means that data are confidential for a proxy not colluding with any provider.

Definition 6 ($N-$Collusion-Secrecy). *A* KMPS *scheme is* $N-$collusion-secret *if for any* $\mathcal{A}$ *we have* $\mathsf{Adv}_\lambda^{N-\mathsf{Coll}}(\mathcal{A}) := \left|Pr\left[\mathsf{Exp}_{\mathcal{A},N}^{\mathsf{Coll}}(\lambda) = 1\right] - 1/2\right| \leq \mathsf{negl}$.

Note that based on the above definition, an KMPS scheme cannot achieve $(k-2)-$collusion-secrecy unless it achieves provider-secrecy as the proxy is essentially a provider with more power (it also holds a share of the data). Our first protocol KAPRE achieves Provider-Secrecy and $0-$collusion-secrecy (Section 4.1), and our second one KAME achieves $(k-2)-$Collusion-Secrecy (Section 4.2).

Integrity and Accountability. First, we define integrity as the user's capacity to know whether or not its data has been altered. Here, the proxy is considered an adversary and is *malicious* only during the download phase.

This property is depicted in game $\mathsf{Exp}_\mathcal{A}^{\mathsf{INTG}}(\lambda)$ of Fig. 4. The adversary plays the role of $k-2$ malicious providers that would collude with a proxy during download phase (and honest during the upload phase). Hence, it controls $k-1$

$\mathsf{Exp}^{\mathsf{Coll}}_{\mathcal{A},N}(\lambda)$

```
1 : $b \leftarrow\!\!\$ \{0,1\}$
2 : $(\mathsf{pk}_i, \mathsf{sk}_i)_{i=N+1}^{n} \leftarrow \mathsf{KeyGen}(\lambda)$
3 : $m_0, m_1, \{\mathsf{pk}_i\}_{i=0}^{N} \leftarrow \mathcal{A}(\{\mathsf{pk}_i\}_{i=N+1}^{n})$
4 : $\textbf{if } |m_0| \neq |m_1| : \textbf{return } b$
5 : $\mathsf{com}_\mathsf{U}, \mathsf{parts}_\mathsf{U}$
        $\leftarrow \mathsf{Transform}(m_b, \mathsf{pk}, \{\mathsf{pk}_i\}_{i=0}^{n})$
6 : $\{h_i\}_{i=0}^{n} \leftarrow \mathsf{Distrib}(\mathsf{parts}_\mathsf{U})$
7 : $\{s_i\}_{i=N+1}^{n} \leftarrow \mathsf{Dec}(h_i, \mathsf{sk}_i)_{i=N+1}^{n}$
8 : $\mathsf{com}_\mathsf{D}, \mathsf{state}_\mathsf{U}, \{\mathsf{nc}_i\}_{i=0}^{k-1}$
        $\leftarrow \mathsf{Designate}(\{\mathsf{pk}_i\}_{i=0}^{k-1})$
9 : $\{s'_i\}_{i=N+1}^{k-1} \leftarrow \mathsf{Hide}(s_i, \mathsf{nc}_i, \mathsf{sk}_i)_{i=N+1}^{k-1}$
10 : $b' \leftarrow \mathcal{A}(\mathsf{com}_\mathsf{U}, \mathsf{parts}_\mathsf{U}, \{h_i\}_{i=0}^{n},$
        $\mathsf{com}_\mathsf{D}, \{\mathsf{nc}_i\}_{i=0}^{n}, \{s'_i\}_{i=N+1}^{k-1})$
11 : $\textbf{return } (b = b')$
```

$\mathsf{Exp}^{\mathsf{INTG}}_{\mathcal{A}}(\lambda)$

```
1 : $(\mathsf{pk}_i, \mathsf{sk}_i)_{i=k-1}^{n} \leftarrow \mathsf{KeyGen}(\lambda)$
2 : $m, \{\mathsf{pk}_i\}_{i=0}^{k-2} \leftarrow \mathcal{A}(\{\mathsf{pk}_i\}_{i=k-1}^{n})$
3 : $s_0, \ldots, s_n, \mathsf{com}_\mathsf{U} \leftarrow \mathsf{Upload}(m)$
4 : $\mathsf{com}_\mathsf{D}, \mathsf{state}_\mathsf{U}, \{\mathsf{nc}_i\}_{i=0}^{k-1}$
        $\leftarrow \mathsf{Designate}(\{\mathsf{pk}_i\}_{i=0}^{k-1})$
5 : $h_{k-1} \leftarrow \mathsf{Hide}(s_{k-1}, \mathsf{nc}_{k-1}, \mathsf{sk}_{k-1})$
6 : $\mathsf{parts}_\mathsf{D} \leftarrow \mathcal{A}(\mathsf{com}_\mathsf{U}, \mathsf{com}_\mathsf{D},$
        $h_{k-1}, \{s_i\}_{i=0}^{k-2}, \{\mathsf{nc}_i\}_{i=0}^{k-1})$
7 : $m' \leftarrow \mathsf{Recover}(\mathsf{parts}_\mathsf{D}, \mathsf{state}_\mathsf{U})$
8 : $\textbf{return } (m' \neq m) \wedge (m' \neq \perp)$
```

$\mathsf{Exp}^{\mathsf{CSP}}_{\mathcal{A}}(\lambda)$

```
1 : $b \leftarrow\!\!\$ \{0,1\}$
2 : $(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathsf{KeyGen}(\lambda)$
3 : $(\mathsf{pk}_i, \mathsf{sk}_i)_{i=k}^{n} \leftarrow \mathsf{KeyGen}(\lambda)$
4 : $m_0, m_1, (\mathsf{pk}_i, \mathsf{sk}_i)_{i=1}^{k-1} \leftarrow \mathcal{A}(\mathsf{pk}_0, \{\mathsf{pk}_i\}_{i=k}^{n})$
5 : $\textbf{if } |m_0| \neq |m_1| : \textbf{return } b$
6 : $\mathsf{com}_\mathsf{U}, \mathsf{parts}_\mathsf{U}$
        $\leftarrow \mathsf{Transform}(m_b, \mathsf{sk}_{\mathcal{U}}, \{\mathsf{pk}_i\}_{i=0}^{n})$
7 : $\{h_i\}_{i=0}^{n} \leftarrow \mathsf{Distrib}(\mathsf{parts}_\mathsf{U})$
8 : $\mathsf{com}_\mathsf{D}, \mathsf{state}_\mathsf{U}, \{\mathsf{nc}_i\}_{i=0}^{k-1}$
        $\leftarrow \mathsf{Designate}(\{\mathsf{pk}_i\}_{i=0}^{k-1})$
9 : $b' \leftarrow \mathcal{A}(\{h_i\}_{i=1}^{k-1}, \{\mathsf{nc}_i\}_{i=1}^{k-1})$
10 : $\textbf{return } (b = b')$
```

$\mathsf{Exp}^{\mathsf{ACC}}_{\mathcal{A}}(\lambda)$

```
1 : $(\mathsf{pk}_k, \mathsf{sk}_k) \leftarrow \mathsf{KeyGen}(\lambda)$
2 : $m, \{\mathsf{pk}_i\}_{i=1}^{k-1} \leftarrow \mathcal{A}(\mathsf{pk}_k)$
3 : $s_0, \ldots, s_n, \mathsf{com}_\mathsf{U} \leftarrow \mathsf{Upload}(m)$
4 : $\mathsf{com}_\mathsf{D}, \mathsf{state}_\mathsf{U}, \{\mathsf{nc}_i\}_{i=0}^{k-1}$
        $\leftarrow \mathsf{Designate}(\{\mathsf{pk}_i\}_{i=0}^{k-1})$
5 : $h_k \leftarrow \mathsf{Hide}(s_k, \mathsf{nc}_k, \mathsf{sk}_k)$
6 : $\{h_i\}_{i=1}^{k-1} \leftarrow \mathcal{A}(\{s_i\}_{i=1}^{k-1}, \{\mathsf{nc}_i\}_{i=1}^{k-1})$
7 : $(\mathsf{parts}_\mathsf{D}, \mathcal{I})$
        $\leftarrow \mathsf{Merge}(\mathsf{com}_\mathsf{U}, \mathsf{com}_\mathsf{D}, \{h_i\}_{i=1}^{k})$
8 : $\textbf{return } \exists i \text{ s.t. } (i \notin \mathcal{I})$
9 :     $\wedge \big(y_i \neq \mathsf{Hide}(s_i, \mathsf{nc}_i, \mathsf{sk}_i)\big)$
```

Fig. 4. Security Games: Provider-Secrecy, $N-$Collusion-Secrecy, User-Integrity and Accountability Games Games.

providers in total. The challenger emulates the user and the $n-k+2$ honest providers remaining. To win, the adversary's final answer must contain a forged $\mathsf{parts}_\mathsf{D}$ that would be accepted as the user's data during recovery, but corresponding to a modified $m' \neq m$ with non-negligible probability. We assume that the honest providers consistently return the correct shares, and the proxy cannot manipulate them to return incorrect data, such as shares from a different file. It can be achieved if the proxy demonstrates the legitimacy of its request to the providers.

Definition 7 (User-Integrity). *A* KMPS *scheme achieves* user-integrity *if for any adversary $\mathcal{A}$ we have* $\mathsf{Adv}^{\mathsf{INTG}}_{\lambda}(\mathcal{A}) := Pr\big[\mathsf{Exp}^{\mathsf{INTG}}_{\mathcal{A}}(\lambda) = 1\big] \leq \mathsf{negl}$.

We define accountability as the proxy's capacity to check the shares integrity, and blame the corresponding provider whenever there is a corrupted share. We

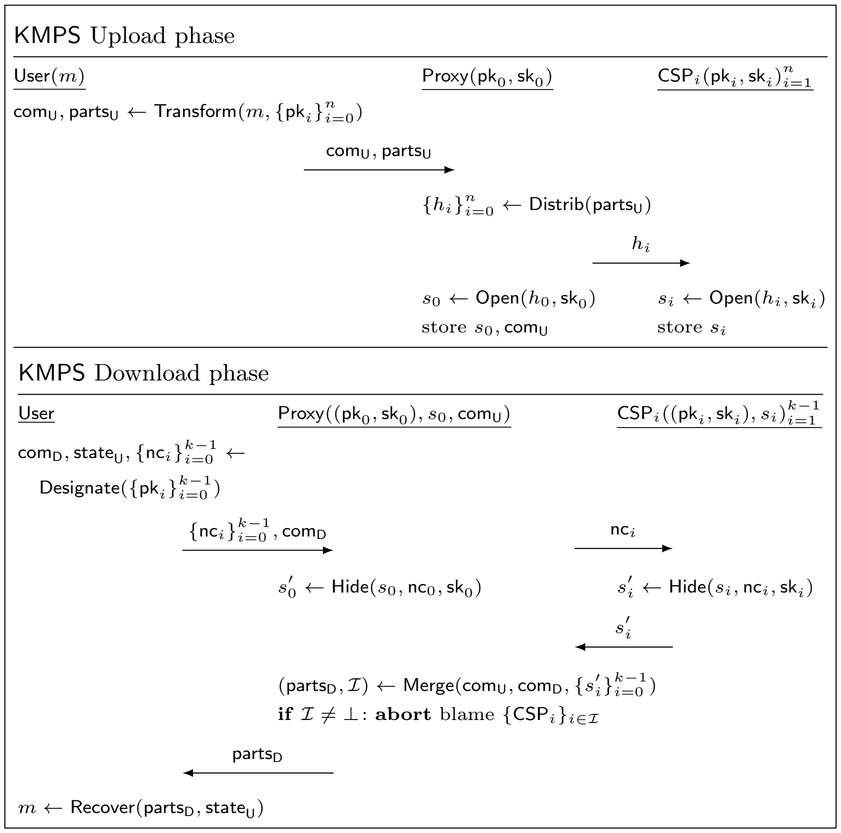
KMPS Upload phase
User(m)
Proxy($\mathsf{pk}_0, \mathsf{sk}_0$)
$\mathsf{CSP}_i(\mathsf{pk}_i, \mathsf{sk}_i)_{i=1}^{n}$
$\mathsf{com_U}, \mathsf{parts_U} \leftarrow \mathsf{Transform}(m, \{\mathsf{pk}_i\}_{i=0}^{n})$
$\mathsf{com_U}, \mathsf{parts_U}$
$\{h_i\}_{i=0}^{n} \leftarrow \mathsf{Distrib}(\mathsf{parts_U})$
h_i
$s_0 \leftarrow \mathsf{Open}(h_0, \mathsf{sk}_0)$
store $s_0, \mathsf{com_U}$
$s_i \leftarrow \mathsf{Open}(h_i, \mathsf{sk}_i)$
store s_i
KMPS Download phase
User
Proxy($(\mathsf{pk}_0, \mathsf{sk}_0), s_0, \mathsf{com_U}$)
$\mathsf{CSP}_i((\mathsf{pk}_i, \mathsf{sk}_i), s_i)_{i=1}^{k-1}$
$\mathsf{com_D}, \mathsf{state_U}, \{\mathsf{nc}_i\}_{i=0}^{k-1} \leftarrow$ $\mathsf{Designate}(\{\mathsf{pk}_i\}_{i=0}^{k-1})$
$\{\mathsf{nc}_i\}_{i=0}^{k-1}, \mathsf{com_D}$
nc_i
$s'_0 \leftarrow \mathsf{Hide}(s_0, \mathsf{nc}_0, \mathsf{sk}_0)$
$s'_i \leftarrow \mathsf{Hide}(s_i, \mathsf{nc}_i, \mathsf{sk}_i)$
s'_i
$(\mathsf{parts_D}, \mathcal{I}) \leftarrow \mathsf{Merge}(\mathsf{com_U}, \mathsf{com_D}, \{s'_i\}_{i=0}^{k-1})$
if $\mathcal{I} \neq \bot$: **abort** blame $\{\mathsf{CSP}_i\}_{i \in \mathcal{I}}$
$\mathsf{parts_D}$
$m \leftarrow \mathsf{Recover}(\mathsf{parts_D}, \mathsf{state_U})$

Fig. 5. Overview of a KMPS protocol.

model this property with the game $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{ACC}}$ of Fig. 4. The adversary plays the role of $k-1$ *colluding* CSPs during download aiming to alter the user's data. The challenger emulates the proxy, who wants to check which shares are corrupted. To win, the adversary's final answer $h_1, \ldots, h_{k-1}$ must contain at least one corrupted share that would be accepted as correct by the proxy, but corresponding to a modified $m' \neq m$ with non-negligible probability. We let the adversary choose the initial message.

Definition 8 (Accountability). *A* KMPS *scheme achieves* accountability *if for any adversary $\mathcal{A}$ we have* $\mathsf{Adv}_{\lambda}^{\mathsf{ACC}}(\mathcal{A}) := Pr\big[\mathsf{Exp}_{\mathcal{A}}^{\mathsf{ACC}}(\lambda) = 1\big] \leq \mathsf{negl}$.

4 KMPS instantiations

Keyless storage schemes are secure up to a given corruption level. First, we propose a KMPS scheme KAPRE (Section 4.1), offering protection against a corrupted proxy or $k-1$ colluding providers. It remains vulnerable to a proxy colluding with a provider. Secondly, at the cost of an additional round of communications between the proxy and the providers, we propose another scheme KAME (Section 4.2) robust against a proxy colluding with $k-2$ providers.

The resulting state of an upload is the same in both schemes, allowing a common download (Section 4.3).

4.1 KAPRE – Upload using Proxy Re-Encryption

Let $\mathsf{E} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be an authenticated symmetric encryption scheme with $\mathsf{IND\text{-}CPA}$ security [5]. Let p be a prime of at least λ bits. Our design of KAPRE upload relies on an additively homomorphic proxy re-encryption scheme $\mathsf{PRE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{ReKey}, \mathsf{ReEnc})$ and a key homomorphic pseudorandom function family $\mathcal{F} = \{F_s\}_{s \in \mathbb{Z}_p}$. As PRE is additively homomorphic, it commutes with Shamir's secret sharing (Definition 2). We give formal definitions of a authenticated symmetric encryption, proxy re-encryption scheme and its security property $\mathsf{PRE\text{-}IND}$, and a key homomorphic pseudorandom function family in Appendix A. We describe the protocol below while the algorithms (KeyGen, $\mathsf{Transform}, \mathsf{Distrib}, \mathsf{Open}$) are detailed in Fig. 6.

Consider a user willing to store some data m among n providers with a threshold recovery level of k shares. First, in $\mathsf{Transform}$ the user samples a random key recK to encrypt its data m as ct with the symmetric encryption E. Then, it prepares a Shamir's secret sharing by encrypting $a_0 \leftarrow \mathsf{recK}$ and $k-1$ random coefficients a_i with the additively homomorphic proxy re-encryption scheme as $\tilde{a}_i$. The user computes one share y_0 for the proxy in plaintext. Next, the user commits the coefficients used in $\mathsf{com_U}$ by evaluating the key homomorphic pseudorandom functions F_{a_i} at a same random point x. The user also computes re-encryption keys rk_i towards each provider, although this only needs to be done whenever a party changes its keys. Finally, the user sends $\mathsf{parts_U} \leftarrow (\{\tilde{a}_i\}_{i=0}^{k-1}, ct, \{\mathsf{rk}_i\}_{i=1}^{n}, y_0)$ and $\mathsf{com_U} \leftarrow (x, \{c_{a_i}\}_{i=0}^{k-1})$ to the proxy.

The proxy stores $\mathsf{com_U}$, which will be used in download to check the shares integrity. Now the proxy can split the data into shares with $\mathsf{Distrib}$. Confidentiality is ensured by the absence of any re-encryption key toward the proxy. It splits the encrypted data ct with an IDA in shares $\{r_i\}_{i=0}^{n}$. Then, from the homomorphically encrypted coefficients $\tilde{a}_i$, the proxy evaluates encrypted Shamir shares $(i+1, \tilde{y}_i)$ as described in Section 2. Essentially, the proxy plays the role of the dealer in Shamir's secret sharing, that computes the shares blindly as the values are hidden by the encryption PRE that it cannot decrypt. Due to the homomorphic property of PRE, the resulting shares are exactly the ones that would be computed by the dealer in classical Shamir's secret sharing, hence still benefitting from its perfect secrecy. The proxy sends to each provider a share of the form $h_i \leftarrow (i+1, \mathsf{PRE.ReEnc}(\tilde{y}_i, \mathsf{rk}_i), r_i)$ while it stores its share $(1, y_0, r_0)$. Finally, each provider decrypts $\tilde{y}_i$ and stores its share $s_i \leftarrow (i+1, y_i, r_i)$.

Under the assumption that no more than k providers may leak their share, recK is protected by Shamir's secret sharing perfect secrecy, and m is protected by E's security. On the other hand, the key is disposable as long as at least k parties are accessible. Ultimately, as Shamir shares are of the size of their secret and IDA like Rabin's [25] produces shares of size $1/k$ of the secret, each provider holds a share of size $\lambda + |ct|/k$, which is almost optimal[5].

4.2 KAME – Upload Using Multikey Encryption

Our protocol KAME's achieves improved security properties. This section highlights the differences between KAPRE and KAME's upload. KAME's algorithms are formally described in Fig. 6. We consider an additively homomorphic multikey encryption scheme $\mathsf{MKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{PartDec}, \mathsf{FinDec})$ (see Appendix A). Due to its homomorphic property, MKE also commutes with Shamir's secret sharing. The KAME scheme resembles KAPRE, where multikey encryption replaces proxy re-encryption. Each provider and the proxy possess a key pair for MKE. The polynomial coefficients a_i of Shamir's secret sharing are encrypted under all the public keys. This mechanism requires the cooperation of all parties to decrypt each share. Hence, the decryption process becomes a two-round protocol involving the providers and the proxy. This approach provides higher security guarantees, as even one honest party can prevent a dishonest decryption.

4.3 Common Download

Due to the common structure of the resulting shares from an upload (a pair of a share from Shamir's secret sharing of recK and an IDA share of ct), we propose a common download. Let $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be an IND-CPA asymmetric encryption scheme (see Appendix A for a full description).

Only k shares are needed for data recovery. The proxy and the providers use their public key $\mathsf{pk}_i^{\mathsf{PK}}$ from PKE. The process start with the client chosing k providers to participate in the download, and executing Designate. It samples k nonces, commits them in $\mathsf{com_D}$ and encrypts one with each $\mathsf{pk}_i^{\mathsf{PK}}$. These ciphertexts and $\mathsf{com_D}$ are sent to the proxy, which redirects each encrypted nonce to the designated provider. In Hide, each provider decrypts its nonce and adds it to its share $(i+1, y_i' := y_i + n_i)$ of recK (this is essentially a one-time pad).

After gathering the shifted shares from the providers, the proxy computes a secret sharing reconstruction $\mathsf{shiftK} \leftarrow \sum_{i=1}^{k} y_i' \ell_i$. First, the proxy does a batched verification on the shares by checking whether $\mathsf{recK} + \sum_{i=1}^{k} n_i \ell_i = \sum_{i=1}^{k} y_i' \ell_i$ with $F_{\mathsf{recK}}(x) \in \mathsf{com_U}$ stored in upload and $F_{n_i}(x) \in \mathsf{com_D}$, using the key-homomorphic property of $\mathcal{F}$. If the equality holds, the shares are correct and shiftK is sent back to the user along with the values ℓ_i and the reconstructed ct from the IDA. The user recover its data by shifting back $\mathsf{recK} \leftarrow \mathsf{shiftK} - \sum n_i \ell_i$ and decrypting ct with recK. By using an authenticated encryption [7], the decryption will fail if ct was altered. Otherwise, to detect which shares are corrupted, the proxy checks for each share if $y_i' - n_i = \sum_{j=0}^{k-1} a_j x_i^j$ from $c_{a_i} \in \mathsf{com_U}$ and $\mathsf{com_D}$, again using the key-homomorphic property of $\mathcal{F}$. Whenever

[5] As a ciphertext from a symmetric encryption usually is about the size of the plaintext, our protocol stores shares close to the optimal size of $|m|/k$, with an additional overhead of the size of the security parameter.

Upload of KAPRE Protocol

```
KeyGen(λ) :
1 : $(\mathsf{pk}^{\mathsf{PK}}, \mathsf{sk}^{\mathsf{PK}}) \leftarrow \mathsf{PKE.KeyGen}(\lambda)$
2 : $(\mathsf{pk}^{\mathsf{PR}}, \mathsf{sk}^{\mathsf{PR}}) \leftarrow \mathsf{PRE.KeyGen}(\lambda)$
3 : return $(\mathsf{pk}^{\mathsf{PK}}, \mathsf{pk}^{\mathsf{PR}}), (\mathsf{sk}^{\mathsf{PK}}, \mathsf{sk}^{\mathsf{PR}})$

Transform$(m, \perp, \mathsf{pk}_1^{\mathsf{PR}}, \ldots, \mathsf{pk}_n^{\mathsf{PR}})$ :
1 : $\mathsf{recK} \leftarrow \mathsf{E.KeyGen}(\lambda)$
2 : $ct \leftarrow \mathsf{E.Enc}(m, \mathsf{recK})$
3 : $(\mathsf{pk}^{\mathsf{PR}}, \mathsf{sk}^{\mathsf{PR}}) \leftarrow \mathsf{PRE.KeyGen}(\lambda)$
4 : $\{\mathsf{rk}_i\}_{i=1}^n \leftarrow \mathsf{PRE.ReKey}(\mathsf{sk}, \mathsf{pk}_i^{\mathsf{PR}})_{i=1}^n$
5 : $\tilde{a}_0 \leftarrow \mathsf{PRE.Enc}(\mathsf{recK}, \mathsf{pk}^{\mathsf{PR}})$
6 : $x \leftarrow\!\!\$\ D, c_{a_0} \leftarrow F_{\mathsf{recK}}(x)$
7 : for $i \in [\![1, k-1]\!]$ :
8 :     $a_i \leftarrow\!\!\$\ \mathbb{Z}_p, c_{a_i} \leftarrow F_{a_i}(x)$
9 :     $\tilde{a}_i \leftarrow \mathsf{PRE.Enc}(a_i, \mathsf{pk}^{\mathsf{PR}})$,
10 : $y_0 \leftarrow \mathsf{recK} + \sum_{i=1}^{k-1} a_i$
11 : return $\mathsf{com}_{\mathsf{U}} \leftarrow (\{\tilde{a}_i\}_{i=0}^{k-1}, ct, \{\mathsf{rk}_i\}_{i=1}^n,$
       $y_0), \mathsf{parts}_{\mathsf{U}} \leftarrow (x, \{c_{a_i}\}_{i=0}^{k-1})$

Distrib$\big(y_0, \{\tilde{a}_i\}_{i=0}^{k-1}, ct, \{\mathsf{rk}_i\}_{i=1}^n\big)$ :
1 : $\{r_i\}_{i=0}^n \leftarrow \mathsf{IDA.Split}(ct, n+1, k)$
2 : $\tilde{P}(x) \leftarrow \sum_{i=0}^{k-1} \tilde{a}_i x^i$
3 : for $i \in [\![1, n]\!]$ : $\tilde{y}_i \leftarrow \tilde{P}(i+1)$
4 :     $\tilde{y}_i \leftarrow \mathsf{PRE.ReEnc}(\tilde{y}_i, \mathsf{rk}_i)$
5 : return $(1, y_0, r_0), \{(i+1, \tilde{y}_i, r_i)\}_{i=1}^n$

Open$(\tilde{y}_i, \mathsf{sk}_i^{\mathsf{PR}})$ :
1 : return $y_i \leftarrow \mathsf{PRE.Dec}(\tilde{y}_i, \mathsf{sk}_i^{\mathsf{PR}})$
```

Upload of KAME Protocol

```
KeyGen(λ) :
1 : $(\mathsf{pk}^{\mathsf{PK}}, \mathsf{sk}^{\mathsf{PK}}) \leftarrow \mathsf{PKE.KeyGen}(\lambda)$
2 : $(\mathsf{pk}^{\mathsf{MK}}, \mathsf{sk}^{\mathsf{MK}}) \leftarrow \mathsf{MKE.KeyGen}(\lambda)$
3 : return $(\mathsf{pk}^{\mathsf{PK}}, \mathsf{pk}^{\mathsf{MK}}), (\mathsf{sk}^{\mathsf{PK}}, \mathsf{sk}^{\mathsf{MK}})$

Transform$(m, \mathsf{pk}_0^{\mathsf{MK}}, \ldots, \mathsf{pk}_n^{\mathsf{MK}})$ :
1 : $\mathsf{recK} \leftarrow \mathsf{E.KeyGen}(\lambda)$
2 : $ct \leftarrow \mathsf{E.Enc}(m, \mathsf{recK})$
3 : $\tilde{a}_0 \leftarrow \mathsf{MKE.Enc}(\mathsf{recK}, \{\mathsf{pk}_i^{\mathsf{MK}}\}_{i=0}^n)$
4 : $x \leftarrow\!\!\$\ D, c_{a_0} \leftarrow F_{\mathsf{recK}}(x)$
5 : for $i \in [\![1, k-1]\!]$ : $a_i \leftarrow\!\!\$\ \mathbb{Z}_p$
6 :     $\tilde{a}_i \leftarrow \mathsf{MKE.Enc}(a_i, \{\mathsf{pk}_i^{\mathsf{MK}}\}_{i=0}^n)$
7 :     $c_{a_i} \leftarrow F_{a_i}(x)$
8 : return $(\{\tilde{a}_i\}_{i=0}^{k-1}, ct), (x, \{c_{a_i}\}_{i=0}^{k-1})$

Distrib$(\{\tilde{a}_i\}_{i=0}^{k-1}, ct)$ :
1 : $\{r_i\}_{i=0}^n \leftarrow \mathsf{IDA.Split}(ct, n+1, k)$
2 : $\tilde{P}(x) \leftarrow \sum_{i=0}^{k-1} \tilde{a}_i x^i$
3 : for $i \in [\![0, n]\!]$ : $\tilde{y}_i \leftarrow \tilde{P}(i+1)$
4 : return $\{(i+1, \tilde{y}_i, r_i)\}_{i=0}^n$

Open$\langle \mathsf{CSP}_i(\tilde{y}_i, \mathsf{sk}_i^{\mathsf{MK}})_{i=1}^n, \mathsf{Proxy}(\tilde{y}_0, \mathsf{sk}_0^{\mathsf{MK}})\rangle$ :
1 : $\mathsf{CSP}_i : y_j^{(i)} \leftarrow \mathsf{MKE.PartDec}(\tilde{y}_j, \mathsf{sk}_i^{\mathsf{MK}})$
2 : $\mathsf{CSP}_i \rightarrow \mathsf{Proxy} : \{y_j^{(i)}\}_{j \neq i}$
3 : $\mathsf{Proxy} \rightarrow \mathsf{CSP}_i : \{y_i^{(j)}\}_{j \neq i}$
4 : $\mathsf{CSP}_i : y_i \leftarrow \mathsf{MKE.FinDec}(\{y_i^{(j)}\}_{j=0}^n)$
5 : Proxy :
       $y_0 \leftarrow \mathsf{MKE.FinDec}(\{y_0^{(j)}\}_{j=0}^n)$
```

Common Download

```
Designate$(\mathsf{pk}_1^{\mathsf{PK}}, \ldots, \mathsf{pk}_k^{\mathsf{PK}})$ :
1 : $n_1, \ldots, n_k \leftarrow\!\!\$\ \mathbb{Z}_p$
2 : $\mathsf{nc}_i \leftarrow \mathsf{PKE.Enc}(n_i, \mathsf{pk}_i^{\mathsf{PK}})$
3 : $c_i \leftarrow F_{n_i}(x)$
4 : return $\{c_i\}_{i=1}^k, \{n_i\}_{i=1}^k, \{\mathsf{nc}_i\}_{i=1}^k$

Hide$((x_i, y_i, r_i), \mathsf{nc}_i, \mathsf{sk}_i^{\mathsf{PK}})$ :
1 : $n_i \leftarrow \mathsf{PKE.Dec}(\mathsf{nc}_i, \mathsf{sk}_i^{\mathsf{PK}})$
2 : return $(x_i, y_i + n_i, r_i)$

Recover$((\mathsf{shiftK}, ct, \{\ell_i\}_{i=1}^k), \{n_i\}_{i=1}^k)$ :
1 : $\mathsf{recK} \leftarrow \mathsf{shiftK} - \sum_{i=1}^k n_i \ell_i$
2 : return $\mathsf{E.Dec}(ct, \mathsf{recK})$
```

```
Merge$((x, \{c_{a_i}\}_{i=0}^{k-1}), \{c_i\}_{i=1}^k, (x_i, y_i', r_i)_{i=1}^k)$ :
1 : $\mathsf{shiftK} \leftarrow \sum_{i=1}^k y_i' \ell_i$
2 : for $i \in [\![1, k]\!]$ : $\ell_i \leftarrow \prod_{j \neq i} \frac{-x_j}{x_i - x_j}$
3 : if $c_{a_0} + \sum_{i=1}^k c_i \ell_i = F_{\mathsf{shiftK}}(x)$ :
4 :     $ct \leftarrow \mathsf{IDA.Rec}(k, \{r_i\}_{i=1}^k)$
5 :     return $(\mathsf{shiftK}, ct, \{\ell_i\}_{i=1}^k)$
6 : else return $\mathcal{I}$ the set of
      indexes $i$ such that
      $c_i \neq F_{y_i'}(x) - \sum_{j=0}^{k-1} c_{a_j} x_i^j$
```

Fig. 6. Description of KAPRE and KAME uploads, and download algorithms.

the equality does not hold, the proxy blames the corresponding provider. The detailed process of the download algorithms is presented in Fig. 6.

The user can store the value x to compute the commitments to the nonces. Otherwise, as the proxy stores x in com_U during upload, the user can ask the proxy for x in the beginning of the download. Additionally, the user might not need to select which CSPs participate in the download, albeit at the cost of slightly more computations. Instead, it can send nonces for all the providers, allowing the proxy to determine which shares to utilize for reconstruction.

5 Security Analysis

In KAPRE, all shares are re-encrypted under pk_i using rk_i. It's important to highlight that the proxy does not possess any re-encryption key and must obtain its share in plaintext. Otherwise, this situation opens up a direct attack: if the client sends the proxy a re-encryption key rk_0, allowing it to decrypt its share, the proxy could re-encrypt all parts of recK using its own public key and subsequently decrypt it, thereby revealing the content of ct. This implies that only *0-collusion-secrecy* can be achieved for KAPRE in addition to *provider-secrecy.* We give sketch proofs that KAPRE indeed achieves these two properties in addition to *user-integrity* and *accountability*, and that KAME has $(k-2)$-*collusion-secrecy*, *user-integrity* and *accountability.* The full proofs are given in the long version [15]. Both schemes are correct, our implementation [15] demonstrate this.

We assume that the designated parties participating in download are always the $k-1$ first CSPs and the proxy, up to a permutation (the proxy is essentially a CSP with additional knowledge).

Theorem 1. *Assume that* PRE *is* $\mathsf{PRE\text{-}IND}$*,* E *is* $\mathsf{IND\text{-}CPA}$*,* $\mathcal{F}$ *is pseudorandom and* PKE *is* $\mathsf{IND\text{-}CPA}$*. Then* KAPRE *achieves 0-collusion-secrecy.*

Sketch Proof. The user's data m is encrypted as ct with recK under E. As recK is sent to the proxy encrypted under the homomorphic proxy re-encryption scheme PRE, the proxy cannot learn anything on m as E is $\mathsf{IND\text{-}CPA}$. During download, the shares are added to the nonces n_i, which value the proxy cannot guess as they are sent by the user encrypted by an $\mathsf{IND\text{-}CPA}$ encryption PKE. The proxy cannot learn anything from the commitments due to the pseudorandomness of the PRF family.

Theorem 2. *Assume that* E *is* $\mathsf{IND\text{-}CPA}$*. Then* KAPRE *has provider-secrecy.*

Sketch Proof. The providers cannot learn anything from their IDA shares of ct as it is encrypted under E with key recK. Their only information on recK comes from their Shamir share of recK, which do not leak anything due to Shamir's perfect secrecy.

Theorem 3. *Assume that* E *is* $\mathsf{IND\text{-}CPA}$*,* MKE *has internal security,* $\mathcal{F}$ *is pseudorandom and the encryption* PKE *is* $\mathsf{IND\text{-}CPA}$*. Then* KAME *has* $(k-2)$*-collusion-secrecy for a Shamir threshold* k*.*

Sketch Proof. The argument is the same as in Theorem 1, but this time the value of recK is protected in upload by the multikey encryption scheme MKE, and the honesty of the remaining providers.

As only the result of an upload is involved in the integrity and accountability games, we prove simultaneously that both schemes achieve these properties.

Theorem 4. *The schemes* KAPRE *and* KAME *achieve user-integrity under the assumption that* $\mathcal{F}$ *is pseudorandom,* PKE *is* $\mathsf{IND\text{-}CPA}$ *and* E *has authenticity.*

Sketch Proof. As one of the nonces remains unknown from the adversary, it cannot forge a false value such that when the user removes the sum $\sum n_i \ell_i$, the result is a valid ciphertext for E.

Theorem 5. *Both schemes have accountability given that* $\mathcal{F}$ *is pseudorandom.*

Sketch Proof. From Shamir's perfect secrecy, the adversary has no advantage to guess the value of recK, hence it has no advantage to provide shares verifying a linear relation with $F_{\mathsf{recK}}(x)$.

6 Instantiation and Experimental Results

We provide a proof-of-concept of both KAPRE and KAME in C++ available in [15]. We used this implementation to design benchmarks and evaluate the performance of our schemes.

Implementation details. We use the OpenFHE [2] and Crypto++ [1] libraries to implement both schemes. We use the OpenFHE implementation of a proxy re-encryption scheme built from BGV [24]. We use the multiparty BGV encryption provided by OpenFHE for multikey encryption. We chose for symmetric encryption E the AES-128 implementation of Crypto++, and we use the Crypto++ implementation of Rabin's IDA. The asymmetric encryption scheme in download is done with the BGV encryption of OpenFHE. As for the PRF family, we implemented the key-homomorphic PRF family described in [3], which is highly parallelizable (we did not do it).

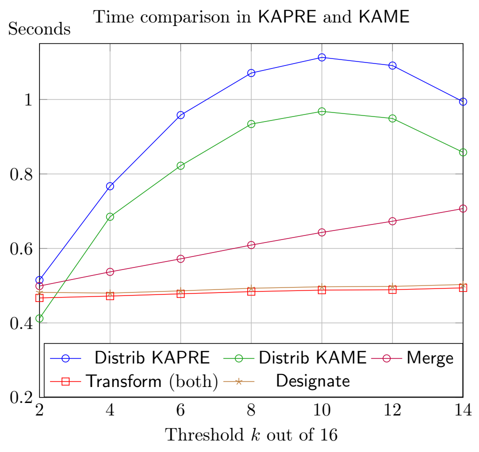

Fig. 7. Average execution time comparison for KAPRE and KAME algorithms (over 500 trials) in function of the threshold k for a number of providers $n = 15$. Open, Hide and $\mathsf{Recover}$ are not included as their times were negligible ($<$ 0.05s).

Benchmark results. Our tests were carried out on an Ubuntu 22.04.2 laptop equipped with an Intel i7-12800H processor of 4.8 GHz and 32 GB of RAM.

We upload and download messages of 1MB (to reflect the processing time of our algorithms instead of the encryption time of the data), for a threshold from $k = 2$ to 14 and a fixed number of providers $n = 15$ (only the complexity of $\mathsf{Distrib}$ depends on n, all the other algorithms have a constant time in n). We use the BGV parameters of OpenFHE that provide 128 bits of security.

Upload. We give in Fig. 7 the average times for the user to execute $\mathsf{Transform}$ and the proxy to execute $\mathsf{Distrib}$, over 500 trials. We suppose the re-encryption keys rk_i are already known of the user in KAPRE, as the user does not have to change its PRE key everytime. The time taken by the user in $\mathsf{Transform}$ is linear in k as it encrypts k Shamir coefficients, and is constant in n. The complexity is the same in both KAPRE and KAME, as it only changes the kind of key used to encrypt the Shamir coefficients. In $\mathsf{Distrib}$, the complexity of Shamir's secret sharing is linear both in k and n as computing one share costs k scalar multiplications (plus a re-encryption in KAPRE). The parabolic shape comes from the IDA which complexity is in $\mathcal{O}(nk - k^2)$ [26], hence the worst trade-off is for k close to $n/2$. In Open, the time for each provider is constant in KAPRE, and linear in n in KAME as each party computes $n + 1$ partial decryptions. We did not include these times in Fig. 7 as they are constant to 45ms per party in KAME and 1ms in KAPRE. However, even if $\mathsf{Distrib}$ takes more time in KAPRE as the proxy also computes n re-encryptions, we stress that KAME is less efficient in the sense that it has an additional round of communications for decryption, which communication time is not taken into account in our experiments.

Download. We give in Fig. 7 the average times to execute $\mathsf{Designate}$ and Merge over 500 trials. The time for $\mathsf{Designate}$ is linear in k, as the user encrypts a nonce of each provider involved in recovery. The time for each provider to compute Hide its share is constant (about 1ms). The time for the proxy to check the shares is linear in k as it computes a linear relation in the k commitments to check integrity. Finally, the time for $\mathsf{Recover}$ is negligible (about 1ms) as it is essentially one AES decryption.

7 Conclusion

We propose an efficient and scalable KMPS scheme addressing collusion amongst providers, and collusion with a curious proxy. Our scheme also achieves data integrity on the user (*integrity*) and proxy (*accountability*) sides, while allowing *forgetful* users. We design two implementations: our first solution KAPRE involving proxy re-encryption and our second one KAME using multi-key encryption schemes. The first one is round optimal in the upload phase, but only provides security against a proxy not colluding with any providers. The second one offers confidentiality against a proxy colluding with up to $k - 2$ malicious providers, but requires an additional round of communication.

References

1. Crypto++ library, https://github.com/weidai11/cryptopp
2. Al Badawi, A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., et al.: Openfhe: Open-source fully homomorphic encryption library. In: Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (2022)
3. Banerjee, A., Peikert, C.: New and improved key-homomorphic pseudorandom functions. In: CRYPTO 2014. pp. 353–370. Springer (2014)
4. Bauer, D.P.: Filecoin. Apress, Berkeley, CA (2022). https://doi.org/10.1007/978-1-4842-8045-4_8, `https://doi.org/10.1007/978-1-4842-8045-4_8`
5. Bellare, M., Desai, A., Jokipii, E., Rogaway, P.: A concrete security treatment of symmetric encryption. In: 38th Annual Symposium on Foundations of Computer Science. pp. 394–403. IEEE (1997)
6. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 531–545. Springer (2000)
7. Bellare, M., Rogaway, P., Wagner, D.: The eax mode of operation. In: Fast Software Encryption, 2004. pp. 389–407. Springer (2004)
8. Bessani, A.N., Correia, M., Quaresma, B., André, F., Sousa, P.: Depsky: Dependable and secure storage in a cloud-of-clouds. ACM Trans. Storage (2013)
9. Bessani, A.N., Mendes, R., Oliveira, T., Neves, N.F., Correia, M., Pasin, M., Veríssimo, P.: SCFS: A shared cloud-backed file system. In: Gibson, G., Zeldovich, N. (eds.) 2014 USENIX Annual Technical Conference. USENIX Association (2014)
10. Chase, M., Davis, H., Ghosh, E., Laine, K.: Acsesor: A new framework for auditable custodial secret storage and recovery. Cryptology ePrint Archive (2022)
11. Chen, L., Zhang, Z., Wang, X.: Batched multi-hop multi-key fhe from ring-lwe with compact ciphertext extension. In: TCC 2017. pp. 597–627. Springer (2017)
12. Cohen, A.: What about bob? the inadequacy of CPA security for proxy reencryption. In: Lin, D., Sako, K. (eds.) PKC 2019 - 22nd IACR. LNCS, Springer (2019)
13. Kim, S.: Key-homomorphic pseudorandom functions from lwe with small modulus. In: EUROCRYPT 2020. pp. 576–607. Springer (2020)
14. Krawczyk, H.: Secret sharing made short. In: Annual international cryptology conference. pp. 136–146. Springer (1993)
15. Lafourcade, P., Mallordy, L.B., Olivier-Anclin, C., Robert, L.: Implementation and long version, `https://hal.science/hal-04540895`
16. Lee, H., Park, J.: On the security of multikey homomorphic encryption. In: Albrecht, M. (ed.) 17th IMA International Conference, IMACC 2019. LNCS, Springer (2019). https://doi.org/10.1007/978-3-030-35199-1_12, `https://doi.org/10.1007/978-3-030-35199-1\_12`
17. Leila, M., Zitouni, A., Djoudi, M.: Ensuring user authentication and data integrity in multi-cloud environment. Hum. centric Comput. Inf. Sci. **10**, 15 (2020). https://doi.org/10.1186/s13673-020-00224-y, `https://doi.org/10.1186/s13673-020-00224-y`
18. Melki, R., Noura, H.N., Chehab, A.: Lightweight multi-factor mutual authentication protocol for iot devices. International Journal of Information Security **19**, 679–694 (2020)
19. Niknia, A., Correia, M., Karimpour, J.: Secure cloud-of-clouds storage with space-efficient secret sharing. J. Inf. Secur. Appl. (2021)

20. Orsini, C., Scafuro, A., Verber, T.: How to recover a cryptographic secret from the cloud. Cryptology ePrint Archive (2023)
21. Papaioannou, T.G., Bonvin, N., Aberer, K.: Scalia: an adaptive scheme for efficient multi-cloud storage. In: Hollingsworth, J.K. (ed.) SC 2012. IEEE/ACM (2012). https://doi.org/10.1109/SC.2012.101, `https://doi.org/10.1109/SC.2012.101`
22. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) CRYPTO '91. LNCS, Springer (1991)
23. Pietro, R.D., Scarpa, M., Giacobbe, M., Puliafito, A.: Secure storage as a service in multi-cloud environment. In: ADHOC-NOW 2017. LNCS, Springer (2017)
24. Polyakov, Y., Rohloff, K., Sahu, G., Vaikuntanathan, V.: Fast proxy re-encryption for publish/subscribe systems. ACM Transactions on Privacy and Security (TOPS) **20**(4), 1–31 (2017)
25. Rabin, M.O.: Efficient dispersal of information for security, load balancing, and fault tolerance. J. ACM (1989)
26. Resch, J.K., Plank, J.S.: AONT-RS: blending security and performance in dispersed storage systems. In: Ganger, G.R., Wilkes, J. (eds.) 9th USENIX Conference on File and Storage Technologies, 2011. USENIX (2011)
27. Rescorla, E.: RFC 8446: The transport layer security (TLS) protocol version 1.3 (2018)
28. Rocha, F., Correia, M.: Lucy in the sky without diamonds: Stealing confidential data in the cloud. In: IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W 2011) (2011)
29. Shamir, A.: How to share a secret. Commun. ACM (1979)
30. Shoup, V.: Sequences of games: a tool for taming complexity in security proofs. IACR Cryptol. ePrint Arch. (2004)
31. Singh, Y., Kandah, F., Zhang, W.: A secured cost-effective multi-cloud storage in cloud computing. In: 2011 IEEE Conference (INFOCOM WKSHPS) (2011). https://doi.org/10.1109/INFCOMW.2011.5928887
32. Stefanov, E., Shi, E.: Multi-cloud oblivious storage. In: Sadeghi, A., Gligor, V.D., Yung, M. (eds.) ACM, CCS, 2013. ACM (2013)
33. Sulochana, M., Dubey, O.: Preserving data confidentiality using multi-cloud architecture. Procedia Computer Science **50**, 357–362 (2015). https://doi.org/https://doi.org/10.1016/j.procs.2015.04.035, `https://www.sciencedirect.com/science/article/pii/S1877050915005360`, big Data, Cloud and Computing Challenges
34. Wilcox-O'Hearn, Z., Warner, B.: Tahoe: the least-authority filesystem. In: ACM international workshop on Storage security and survivability. pp. 21–26 (2008)
35. Witanto, E.N., Stanley, B., Lee, S.: Distributed data integrity verification scheme in multi-cloud environment. Sensors **23**(3), 1623 (2023). https://doi.org/10.3390/s23031623, `https://doi.org/10.3390/s23031623`

A Appendix

We give a formal definition of an authenticated symmetric encryption, a proxy re-encryption scheme, a multikey encryption scheme and their security properties. We also give the security game for secret sharing perfect secrecy [22].

Definition 9 (Authenticated Encryption [6]). *A symmetric encryption scheme* E *has authenticity if for any* $\mathcal{A}$ *in game* $\mathsf{Exp}^{\mathsf{AUTH}}_{\mathcal{A}}(\lambda)$ *(Fig. 8) we have:* $\mathsf{Adv}^{\mathsf{AUTH}}_{\lambda}(\mathcal{A}) := Pr\left[\mathsf{Exp}^{\mathsf{AUTH}}_{\mathcal{A}}(\lambda) = 1\right] \leq \mathsf{negl}$.

Definition 10 (Proxy Re-encryption [24]). *A* proxy re-encryption scheme PRE *is given by:*

$\mathsf{PRE.KeyGen}(\lambda) \to (\mathsf{pk}, \mathsf{sk})$ *: outputs a key pair according to* λ*.*

$\mathsf{PRE.Enc}(m, \mathsf{pk}) \to c$ *: on input* pk *and a message* m*, outputs a ciphertext.*

$\mathsf{PRE.Dec}(c, \mathsf{sk}) \to m$ *: returns a decryption* m *of* c*.*

$\mathsf{PRE.ReKey}(\mathsf{sk}_i, \mathsf{pk}_j) \to \mathsf{rk}_{i\to j}$*: returns a re-encryption key which allows to transform ciphertexts under* pk_i *into ciphertexts under* pk_j*.*

$\mathsf{PRE.ReEnc}(c_i, \mathsf{rk}_{i\to j}) \to c_j$*: on input a ciphertext encrypted under* pk_i *and a re-encryption key* $\mathsf{rk}_{i\to j}$*, returns a ciphertext encrypted under* pk_j*.*

It must be correct and have $\mathsf{PRE\text{-}IND}$ *security (Definition 11).*

Definition 11 ($\mathsf{PRE\text{-}IND}$ [12]). *The scheme* PRE *is* $\mathsf{PRE\text{-}IND}$ *if for any* $\mathcal{A}$ *in* $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{PRE\text{-}IND}}$ *(Fig. 8),* $\mathsf{Adv}_{\lambda}^{\mathsf{PRE\text{-}IND}}(\mathcal{A}) := \left|Pr\left[\mathsf{Exp}_{\mathcal{A}}^{\mathsf{PRE\text{-}IND}}(\lambda) = 1\right] - 1/2\right| \le \mathsf{negl}.$

Definition 12 (Multi-key Encryption [11]). *A* multi-key encryption scheme MKE *is given by:*

$\mathsf{MKE.KeyGen}(\lambda) \to (\mathsf{pk}, \mathsf{sk})$ *: outputs a key pair* $(\mathsf{pk}, \mathsf{sk})$*.*

$\mathsf{MKE.Enc}(m, \mathsf{pk}) \to c$ *: on input pk and a message* m*, outputs a ciphertext.*

$\mathsf{MKE.PartDec}(c, \mathsf{sk}_i) \to p_i$ *: returns a partial decryption* p_i *of* c*.*

$\mathsf{MKE.FinDec}(p_1, \ldots, p_n) \to m$*: given all the partial decryptions, outputs* m*.*

It must be correct and internally secure (Definition 13).

Definition 13 (Internal Security [16]). *The* MKE *has* internal security *if for any* $\mathcal{A}$ *in* $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{INT}}$ *(Fig. 8),* $\mathsf{Adv}_{\lambda}^{\mathsf{INT}}(\mathcal{A}) := \left|Pr\left[\mathsf{Exp}_{\mathcal{A}}^{\mathsf{INT}}(\lambda) = 1\right] - 1/2\right| \le \mathsf{negl}.$

Pseudorandom Function Family Consider three finite sets: D, R, and S, and let $\Gamma_{D,R}$ represent the set of all functions from D to R.

Definition 14 (Pseudo-random Function Family [30]). *Let* $\mathcal{F} = \{F_s\}_{s\in S}$ *be a family of keyed functions mapping* D *to* R*. The familly* $\mathcal{F}$ *is* pseudorandom *if* $\mathsf{Adv}_{\lambda}^{\mathsf{PRF}}(\mathcal{A}) := |Pr[s \leftarrow\$ S : \mathcal{A}^{F_s} = 1] - Pr[f \leftarrow\$ \Gamma_{D,R} : \mathcal{A}^{f} = 1]|$ *for any adversary* $\mathcal{A}$ *is negligible.*

Definition 15 (Key-Homomorphic PRF [3]). *A PRF family* $\mathcal{F} = \{F_s\}_{s\in S}$ *is* key-homomorphic *if* S *has a group structure and if there is a polynomial time algorithm that, given* $F_s(x)$ *and* $F_t(x)$*, outputs* $F_{s+t}(x)$*.*

$\mathsf{Exp}_{\mathcal{A}}^{\mathsf{INT}}(\lambda)$

```
$\mathsf{pk}_0 \leftarrow \mathcal{A}(\lambda), b \leftarrow\!\!\$\ \{0,1\}$
$(\mathsf{pk}_i, \mathsf{sk}_i)_{i=1}^{n} \leftarrow \mathsf{MKE.KeyGen}(\lambda)$
$i, m_0, m_1 \leftarrow \mathcal{A}(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$
$c \leftarrow \mathsf{MKE.Enc}(m_b, \mathsf{pk}_0, \ldots, \mathsf{pk}_n)$
$\forall j \neq i, p_j \leftarrow \mathsf{MKE.PartDec}(c, \mathsf{sk}_j)$
$b' \leftarrow \mathcal{A}(\{p_j\}_{j \neq i})$
return $(b = b')$
```

Oracle $\mathsf{ORk}(i, j)$

```
if $(i \in \mathsf{Hon} \wedge j \in \mathsf{Corr})$
    return $\perp$
return $\mathsf{PRE.ReKey}(\mathsf{sk}_i, \mathsf{pk}_j)$
```

Oracle $\mathsf{ORenc}(i, j, ct_i)$

```
if $(i \in \mathsf{Hon} \wedge j \in \mathsf{Corr})$
    return $\perp$
return $\mathsf{PRE.ReEnc}(ct_i, \mathsf{rk}_{i \to j})$
```

$\mathsf{Exp}_{\mathcal{A}}^{\mathsf{AUTH}}(\lambda)$

```
$k \leftarrow \mathsf{E.KeyGen}(\lambda)$
for $i \in [\![1, q]\!]$:
    $m \leftarrow \mathcal{A}, c_i \leftarrow \mathsf{E.Enc}(m, k)$
$c \leftarrow \mathcal{A}(\{c_i\}_{i=1}^{q})$
if $\exists i, c = c_i$: return $\perp$
return $\mathsf{E.Dec}(c, k) \neq \perp$
```

$\mathsf{Exp}_{\mathcal{A}}^{\mathsf{PRE\text{-}IND}}(\lambda)$

```
$nb_k \leftarrow 0, \mathsf{Hon}, \mathsf{Corr} \leftarrow \emptyset$
$b \leftarrow\!\!\$\ \{0,1\}$
$nb_k, \mathsf{Hon}, \mathsf{Corr} \leftarrow \mathcal{A}^{\mathsf{OKey}}(\lambda)$
$\mathsf{rk}_{i \to j} \leftarrow \mathsf{PRE.ReKey}(\mathsf{pk}_i, \mathsf{sk}_i)$
$i, m_0, m_1 \leftarrow \mathcal{A}^{\mathsf{ORk},\mathsf{ORenc}}(nb_k)$
if $i \in \mathsf{Hon}, c \leftarrow \mathsf{PRE.Enc}(\mathsf{pk}_i, m_b)$
else $c \leftarrow \perp$
$b' \leftarrow \mathcal{A}(c)$
return $(b = b')$
```

Oracle $\mathsf{OKey}(\lambda, b)$

```
$(\mathsf{pk}_{nb_k}, \mathsf{sk}_{nb_k}) \leftarrow \mathsf{PRE.KeyGen}(\lambda)$
increment $nb_k$
if $b = 0$: add $nb_k$ to $\mathsf{Corr}$
    return $(\mathsf{pk}_{nb_k}, \mathsf{sk}_{nb_k})$
add $nb_k$ to $\mathsf{Hon}$, return $\mathsf{pk}_{nb_k}$
```

$\mathsf{Exp}_{\mathcal{A}}^{\mathsf{PS}}(\lambda, k, n)$

```
$m_0, m_1 \leftarrow \mathcal{A}(\lambda)$
$b \overset{\$}{\leftarrow} \{0,1\}$
$\{s_i\}_{i=1}^{n} \leftarrow \mathsf{SS.Split}(m_b, n, k)$
$b' \leftarrow \mathcal{A}(s_1, \ldots, s_{k-1})$
return $(b = b')$
```

Fig. 8. Security games: $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{AUTH}}(\lambda)$ authenticated symmetric encryption, secret sharing perfect-secrecy $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{PS}}(\lambda, k, n)$, multi-key encryption internal security $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{INT}}(\lambda)$, and proxy re-encryption indistinguishability $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{PRE\text{-}IND}}(\lambda)$ with its oracles.