Secure and Flexible Data Sharing for Distributed Storage with Efficient Key Management

Liang Xue*, Dongxiao Liu*, Cheng Huang*, Xuemin (Sherman) Shen*, Weihua Zhuang*, Rob Sun†, Bidi Ying†
*Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada , N2L 3G1
†Huawei Technologies Canada, Ottawa, Canada, K2K 3J1

***Abstract*—In this paper, we propose a Secure and Flexible Data Sharing (SFDS) scheme for distributed storage, where data owners can outsource their data to a distributed storage network and share the data with authorized users. To preserve confidentiality, all data are encrypted by data owners' secret keys before being outsourced, and fine-grained access policies are enforced on the encrypted data (ciphertexts) to achieve flexible data sharing. Furthermore, based on the ciphertext puncturable encryption and the hierarchical identity-based encryption, we design an efficient key and ciphertext update mechanism, which enables data owners to update their secret keys and the corresponding ciphertexts periodically to deal with side-channel attacks and system vulnerabilities. Update tokens are constructed to directly derive new keys and ciphertexts. Through detailed security analysis, it is demonstrated that SFDS can achieve all three essential security properties, i.e., forward security, post-compromise security, and collusion attack resistance.**

***Index Terms*—Key management, ciphertext update, data sharing, access control.**

I. Introduction

Distributed data storage allows users to store their data repeatedly on a number of network nodes for better availability [1], [2]. Users can access the replica closest to them for low delay. Moreover, distributed storage provides good scalability, as more resources can be provisioned on-demand by adding computers to the storage network. An example of a distributed storage network is Filecoin, where many nodes can provide data storage services [3]. If a user chooses to store data in Filecoin, it first views available storage and the corresponding prices. Nodes in the network compete to win a storage contract. The user selects the affordable nodes and sends data to them. When the user needs to access its data, it first looks up nodes that store the data and retrieves them from the closest node. With distributed storage, users can store their data in an economical and reliable way.

For distributed data storage, there are many data security issues that need to be solved. Since data are stored at different nodes, data confidentiality should be guaranteed, and data owners should have the ability to determine who can access their data. Thus, data should be encrypted by data owners' secret keys before being outsourced. In addition, considering that data owners' secret keys may be exposed due to side-channel attacks and software vulnerabilities, which will pose a threat to data security, a data owner can update its key periodically to defend against the attacks. This provokes two questions on the key management. First, how to efficiently update the ciphertexts that are encrypted with the previous key. A naive approach is to download the ciphertexts from the storage nodes and re-encrypt them using the new key, which causes huge communication overheads for the data owner. Second, how to generate and distribute new keys to authorized users such that they can continue decrypting the updated ciphertexts. That is, an efficient key management mechanism is required for both data owners and users. For the security of the key management mechanism, both forward security and post-compromise security should be guaranteed. The former ensures that even if an adversary obtains the secret key of the current time period, it cannot decrypt a ciphertext that is encrypted by a key in a previous time period, and the latter ensures that the exposure of historical keys will not affect the security of the future ciphertext. In addition, the key management mechanism should be resistant to collusion attacks, that is, users cannot combine their keys to decrypt a ciphertext that they have no access to.

In this paper, we propose a secure and flexible data sharing (SFDS) scheme with efficient key management, where keys and ciphertexts can be updated periodically. In SFDS, data are encrypted with a decryptable time period and an access policy, i.e., there are two components in a private key, one corresponds to a time period, and the other one is correlated to attributes of a user. The two components from different users cannot be combined to carry out a collusion attack. For each time period, a time tag is built based on hierarchical identity-based encryption (HIBE), by which a data owner can use the current private key to produce the key for the next time period. For the key update of authorized users, the data owner generates a key update token, which can be encrypted and published on a bulletin board, such as blockchain, and only the authorized users can decrypt it and obtain the token. Based on the key update token, the users can update their private keys. For the ciphertext update, the data owner can generate a ciphertext update token and send it to the storage nodes, which can update the ciphertext based on the original ciphertext and the token. Note that if a data owner wants to delete a ciphertext, it can update the ciphertext without updating the key. This can achieve the right to be forgotten requirement of GDPR. The contributions of this paper are summarized as below:

- We propose a flexible data sharing scheme with key and ciphertext update that can guarantee forward security and

post-compromise security. Users in the system are represented by a set of attributes, and data can be encrypted with a predefined access policy and a decryptable time period. For ciphertext update, a data owner needs to send only a ciphertext update token to storage nodes for accomplishing the ciphertext update;
- SFDS can achieve efficient key management for data owners and users. In each time period, a data owner only needs to store a private key, which can be used to derive the subsequent keys. For key update of authorized users, the data owner generates a key update token for users, who can attain the new key by using the key update token. When a user is revoked, it cannot obtain the token;
- Under the decisional bilinear Diffie-Hellman exponent assumption, we demonstrate that SFDS can achieve desired security properties.

The remainder of this paper is organized as follows. In Section II, we introduce the system model and design goals. We describe the definitions, building blocks, and the design overview of SFDS in Section III. In Section IV, we present the detailed construction of SFDS. We demonstrate the security analysis in Section V, and simulation results are presented in Section VI. We review the related works in Section VII, and summarize this study in Section VIII.

II. System Model and Design Goals

In this section, we first describe our system model and then present the design goals.

A. System Model

As shown in Fig.1, we consider flexible data sharing with periodical ciphetext and key update in distributed storage, which involves data owners, data users, a trusted authority (TA), blockchain, and a distributed storage platform (DSP).

- Data owners – To reduce the burden of data management and achieve reliable and economic data storage, data owners store their data on a distributed storage platform, such as Storj and Filecoin. To preserve the data privacy, a data owner first encrypts its data before storing them in a distributed network. Moreover, the data owner wants to share its data to authorized users. Considering that secret keys may be leaked due to various attacks, the data owner can update the secret key and the ciphertext periodically, and publish the key update information on the blockchain to inform the authorized users, who can utilize the update token to generate new secret keys and access the shared data;
- Data users – A data owner can share its data to data users. In the system, each data user is associated with a set of attributes, which represent the decryption capability of the user. When a data user obtains a key update token, it can compute a new key based on the old secret key and the key update token;
- TA – Data owners and data users should register themselves with TA. After validating their identities, TA will create public and private keys for them according to their attributes and a submitted time tag. Note that TA is not involved in the key updates of data owners and data users;
- Blockchain – Blockchain can be a permissioned blockchain, such as Hyperleger Fabric, which is maintained by the system users [4]. Data owners can upload transactions to the blockchain, and data users can obtain the key update information on it;
- DSP – It can provide reliable and affordable distributed data storage for users. Data are stored on geographically diverse nodes and encrypted before being uploaded to the distributed network. DSP is honest and curious in our system, i.e., it will honestly store data and execute update operations, but would like to learn data plaintext.

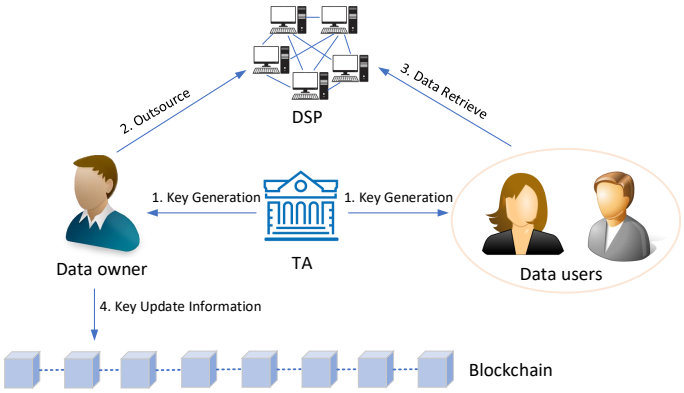

Fig. 1. System Model

B. Design Goals

In this work, the following properties should be satisfied:

- Flexible access control – A data owner can define fine-grained access policies for its data. Data users who satisfy the policies can decrypt the ciphertexts. When a data owner updates a secret key, authorized data users should also be able to update their keys, such that the ciphertext can still be decrypted;
- Efficient key management – Data owners and data users should store as few keys as possible. They do not need to store the secret keys for all time periods. The updated keys should be derivable from old keys;
- Data security – Unauthorized users should not be able to obtain the shared data. Moreover, forward security and post-compromise security should be satisfied, i.e., the exposure of the current key will not affect the security of ciphertexts encrypted with previous keys, and based on the current key and a key update token, one cannot compromise data security of the ciphertexts encrypted with the future keys.

III. Definitions, Building Blocks, and Design Overview of SFDS

In this section, we first define the algorithms in SFDS. Then, we present the building blocks for constructing SFDS. Finally, we give an overview on the design of SFDS.

A. Definition of SFDS

Our SFDS consists of eight algorithms: **Setup, KeyGen, KeyUpdate, Encrypt, Decrypt, CUpdateToken, CiphertextUpdate, KUpdateToken**.

- Setup – By taking as input the whole number of attributes (U) and the number of updates (T), TA outputs the pubic parameters (PK), and a master secret key (MSK);
- KeyGen – By taking as input MSK, a user's attribute set (S), and a time tag ($\vec{\tau}$), TA returns a private key (SK) for the user;
- KeyUpdate – By taking as input SK and the corresponding time tag ($\vec{\tau}_d$), a data owner outputs a private key (SK') for the next time period with time tag $\vec{\tau}_{d+1}$;
- Encrypt – By taking as input $params$, an access structure ($\mathbb{A}$), a message (K), and a decryptable time period whose time tag is $\vec{\tau}_d$, a data owner obtains a ciphertext (CT);
- Decrypt – The inputs of the algorithm include $params$, CT, $\mathbb{A}$, a decryptable time tag ($\vec{\tau}_d$), and SK of a user. If the user's attributes satisfy $\mathbb{A}$, and the time tag of SK is equal to $\vec{\tau}_d$, the user can decrypt CT, and obtain K;
- CUpdateToken – By taking as input $params$ and a time tag ($\vec{\tau}_d$), a data owner outputs a ciphertext update token (Λ);
- CiphertextUpdate – By taking as input CT with the decryptable time tag $\vec{\tau}_d$ and Λ, a storage node outputs a new ciphertext (CT') with the decryptable time tag $\vec{\tau}_{d+1}$;
- KUpdateToken – By taking as input $\mathbb{A}$ and a time tag ($\vec{\tau}_d$), a data owner outputs a key update token (Υ), with which one can update its SK to the next time period.

B. Building Blocks

1) Linear Secret Sharing Schemes (LSSS): A secret sharing scheme Π is called linear if the following conditions are satisfied [5]:

1) The share of each party is a vector.
2) There exists a share-generating matrix M for Π. Assume M has l rows and n columns. For all $i = 1, \ldots, l$, there is a function ρ that maps the i-th row of M to the party associated with row i, which is denoted as $\rho(i)$. Given a column vector $v = (s, y_2, \cdots, y_n)$, where s is the secret to be shared, and $y_2, \cdots, y_n$ are randomly selected, Mv is the shares of s according to Π. The element $(Mv)_i$ in the vector belongs to party ρ_i.

LSSS has the linear reconstruction property. That means, if a set of parties satisfy the policy corresponding to M, they can reconstruct the secret.

2) Hierarchical Identity-based Encryption (HIBE): In HIBE [6], identities are used to encrypt data and are constructed according to a tree. A user who has the private key corresponding to a high-level identity can derive the private key for a low-level identity. An HIBE scheme consists of four algorithms: 1) The Setup algorithm creates system parameters and a master key; 2) The Key Generation algorithm generates a private key for a given identity; 3) The Encryption algorithm is used to compute a ciphertext (C) based on an identity (ID) and a message; 4) Given the private key of ID and C, the Decryption algorithm can output the plaintext of C.

C. Design Overview

In our system, a data owner can encrypt its data and store them at a distributed network. The data can be encrypted with a predefined policy such that only authorized users can obtain the plaintext [7]. To defend against key-compromise attacks, the data owner updates the secret key periodically. Accordingly, the ciphertext is also updated, such that the data owner can continue decrypting it. For the ciphertext update, the data owner first generates a ciphertext update token, and transmits the token to the storage nodes, which will update the ciphertext according to the token. From the token, the storage nodes cannot attain any additional information about the data. Since the ciphertext is updated, the users cannot access it anymore. To grant them the ability to decrypt the new ciphertext, the data owner generates a key update token, and sends it to the users. If a user is revoked, it cannot obtain the key update token and the new secret key. For the key-update token distribution, to alleviate the communication overhead of the data owner, we use the blockchain to transmit the key update information. Specifically, when there is a ciphertext update, the data owner will upload a transaction, which includes the file name, the decryptable time period of the ciphertext, and the ciphertext of the key update token, which can only be decrypted by the unrevoked users.

Our scheme uses the puncture property of the ciphertext puncturable encryption, which enables the update of ciphertexts [8]. To achieve fine-grained data access and key update, we associate a private key with a special time period. Based on the idea of HIBE, a private key can be updated from a time period to the next time period. Data are encrypted with an access policy and a decryptable time period. A user can obtain the plaintext only if its attributes satisfy the defined policy, and it has the private key corresponding to the decryptable time period. For the secret key update, only the key components related to a time period need to be evolved.

We use a vector $\vec{\tau}$ to represent a time period. A time period can be one month or one year. For the first time period, $\vec{\tau}_1 = \{\tau_1\}$. For the second time period, $\vec{\tau}_2 = \{\tau_1, \tau_2\}$, and so on. For $i = 1, \ldots, T$, $\{\tau_1, \ldots, \tau_T\}$ are integers defined by the data owner, where T represents the number of updates or the number of time periods. When a private key is updated T times, the data owner can apply for a new key from TA.

IV. Detailed Construction of SFDS

Let G be a bilinear group, and its order be a prime (p). $e : G \times G \rightarrow G_T$ is a bilinear map. Assume the whole number of the attributes in the system is U.

For key generation, similar to the HIBE, users have hierarchical secret keys. We assume in time period d, the time tag of a secret key is $\vec{\tau}_d = (\tau_1, \tau_2, \ldots, \tau_d)$. Using the secret key $sk_{\vec{\tau}_d=(\tau_1,\tau_2,\ldots,\tau_d)}$ of a user, the user can derive the secret key of the next period, whose time tag is $\vec{\tau}_{d+1} = (\tau_1, \tau_2, \ldots, \tau_{d+1})$.

In our scheme, when a data owner wants to outsource his file f to a storage platform, it first encrypts the file using a symmetric encryption scheme, such as AES. Then, it encrypts the symmetric key (K), which is an element in group G, by using the following scheme.

Setup: To generate the public parameters for the users in the system, a trusted authority (TA) chooses a random generator $g \in G$, and randomly selects $\alpha, \beta \in Z_p$. Then, TA randomly chooses $F_0, F_1, \ldots, F_T \in G$, $h_1, \ldots, h_U \in G$, $\eta_1, \ldots, \eta_T \in G$, and $F'_2, \ldots, F'_T \in G$. The public parameters are $param = (g, g^\beta, e(g,g)^\alpha, F_0, \ldots, F_T, h_1, \ldots, h_U, \eta_1, \ldots, \eta_T, F'_2, \ldots, F'_T)$. The master secret key MSK is g^α.

KeyGen: The algorithm is run by TA. Let S be the attribute set of a user. The time tag of the user is $\vec{\tau}_d = \{\tau_1, \ldots, \tau_d\} \in Z_p$. TA first randomly chooses $t, v_{\tau_d} \in Z_p$, and generates a private key for the user as follows:

$$L = g^t, \qquad \forall x \in S \; K_x = h_x^t,$$

$$D_{0,\vec{\tau}_d} = g^{v_{\tau_d}}, \quad D_{1,\vec{\tau}_d} = g^\alpha g^{\beta t}(F_0 \prod_{i=1}^{d} F_i^{\tau_i})^{v_{\tau_d}},$$

$$\{Q_{j,\vec{\tau}_d} = (F'_j F_j)^{v_{\tau_d}}\}_{j=d+1,\ldots,T}.$$

The private key of the user for time period $\vec{\tau}_d$ is $SK_1 = \{L, \{K_x\}_{x\in S}, D_{0,\vec{\tau}_d}, D_{1,\vec{\tau}_d}, \{Q_{j,\vec{\tau}_d}\}_{j=d+1,\ldots,T}\}$.

KeyUpdate: A user who has the secret key with the time tag $\tau_d = \{\tau_1, \ldots, \tau_d\}$ can generate the secret key for time period with the tag $\vec{\tau}_{d+1} = \{\tau_1, \ldots, \tau_{d+1}\}$ as follows:

1) The user randomly chooses α', and generates $D_{0,\vec{\tau}_{d+1}} = D_{0,\vec{\tau}_d}$, $D_{1,\vec{\tau}_{d+1}} = D_{1,\vec{\tau}_d} Q_{d+1,\vec{\tau}_d}^{\tau_{d+1}} g^{\alpha'}$.
2) $Q_{d+1,\vec{\tau}_d}$ is deleted, i.e. $\{Q_{j,\vec{\tau}_d} = (F'_j F_j)^{v_{\tau_d}}\}_{j=d+2,\ldots,T}$.
3) Other components in the secret key remain the same.

Encrypt: In this algorithm, the data owner takes as input the public parameters $params$, the key $K \in G_T$ to encrypt, and an LSSS access structure (M, ρ), where M is an $l * n$ matrix, and ρ associates rows of the matrix M to attributes. We assume that for M, an attribute is associated with at most one row of M. To generate a ciphertext of K, the user first randomly selects $\vec{v} = (s, y_2, \ldots, y_n) \in Z_p^n$, and for $i = 1$ to l, it computes $\lambda_i = \vec{v} \cdot M_i$, where M_i is the i-th row of the matrix M. Let $\vec{\tau}_d = (\tau_1, \ldots, \tau_d)$ be the decryptable time period, where $d < T$. The data owner randomly chooses an $s \in Z_p$, and computes the ciphertext as follows:

$$C = K \cdot e(g,g)^{\alpha s}, \quad C' = g^s, \quad C'' = (F_0 \prod_{j=1}^{d} F_j^{\tau_j})^s,$$

$$for\ i = 1, \ldots, l,\ C_i = g^{\beta\lambda_i} h_{\rho_i}^{-s}, \quad \{Q'_j = (F'_j F_j)^s\}_{j=d+1,\ldots,T},$$

$$E' = \prod_{i=1}^{d} \eta_i^s, \quad \{E_j = \eta_j^s\}_{j=d+1,\ldots,T}.$$

The ciphertext is $CT_d = \{C, C', C'', C_1, \ldots, C_l, \{Q'_j\}_{j=d+1,\ldots,T}, E', \{E_j\}_{j=d+1,\ldots,T}\}$ along with the description of the $\{M, \rho\}$. CT_d means the time tag of the ciphertext is $\vec{\tau}_d$. After data encryption, the data owner sends the file storage information and time tag of the ciphertext to the authorized users. Users can apply for a private key from TA based on their attributes.

In the ciphertext, $\{Q'_j\}_{j=d+1,\ldots,T}$ are introduced to allow the ciphertext to be updated, so that the updated private key can decrypt the new ciphertext.

Decrypt: This algorithm takes as input the private key of a user, whose attribute set is S, and a ciphertext CT which is associated with an access structure (M, ρ). Assume the time tag corresponding to the private key is $\vec{\tau}_k = \{\tau_1, \ldots, \tau_k\}$. The ciphertext can be decrypted only if the attribute set S satisfies the access structure and the user has the private key for the same time period that the ciphertext is encrypted with, or the user can derive the private key for that time period. If these two conditions are both satisfied, the user defines the set $I \subset \{1, 2, \ldots, l\}$ as $I = \{i : \rho(i) \in S\}$. Then, the user computes the $\{\omega_i \in Z_p\}_{i\in I}$, which are constants that satisfy $\sum_{i\in I} \omega_i \lambda_i = s$, where λ_i are valid shares of the secret s.

To decrypt the ciphertext, the user first computes

$$\begin{aligned}\mu &= \prod_{i\in I} (e(L, C_i) e(C', K_{\rho(i)}))^{\omega_i} \\ &= \prod_{i\in I} e(g,g)^{t\beta\lambda_i\omega_i} \\ &= e(g,g)^{\beta st}.\end{aligned}$$

If $\vec{\tau}_k = \vec{\tau}_d$, where $\vec{\tau}_d$ is the time period corresponding to the ciphertext, the user has the private key components $\{D_{0,\tau_d}, D_{1,\tau_d}\}$ that can be used to decrypt the ciphertext. Then, the user calculates $K = C \cdot \mu \cdot e(D_{0,\vec{\tau}_d}, C'')/e(D_{1,\vec{\tau}_d}, C')$.

The correctness can be validated by:

$$\begin{aligned}&C \cdot \mu \cdot e(D_{0,\vec{\tau}_d}, C'')/e(C', D_{1,\vec{\tau}_d}) \\ &= \frac{C \cdot e(g,g)^{\beta st} \cdot e(g^{v_{\tau_d}}, F_0 \prod_{j=1}^{d} F_j^{\tau_j})^s}{e(g^s, g^\alpha g^{\beta t}(F_0 \prod_{j=1}^{d} F_j^{\tau_j})^{v_{\tau_d}})} \\ &= K.\end{aligned}$$

CUpdateToken: This algorithm is used to generate the ciphertext update token for the storage nodes. When the data owner wants to update the ciphertext with the time tag $\vec{\tau}_d = \{\tau_1, \ldots, \tau_d\}$ to the ciphertext with the time tag $\vec{\tau}_{d+1} = \{\tau_1, \ldots, \tau_{d+1}\}$ so that the updated private key can still decrypt the ciphertext, the data owner first randomly chooses an $s' \in Z_p$, and generates the ciphertext update token ϕ_d as $\phi_d = (g^{s'}, g^{\alpha'} \cdot \prod_{i=1}^{d} \eta_i^{s'})$, where α' is the random number chosen in the KeyUpdate algorithm. The data owner sends the ciphertext update token ϕ_d to the storage platform.

CiphertextUpdate: Assume the ciphertext stored on the platform is with the time tag $\vec{\tau}_d = \{\tau_1, \ldots, \tau_d\}$, With the update token ϕ_d, the storage platform first computes

$$\begin{aligned}\frac{e(C', g^{\alpha'} \cdot \prod_{i=1}^{d} \eta_i^{s'})}{e(g^{s'}, E')} &= \frac{e(g^s, g^{\alpha'} \cdot \prod_{i=1}^{d} \eta_i^{s'})}{e(g^{s'}, \prod_{i=1}^{d} \eta_i^s)} \\ &= e(g^s, g^{\alpha'})\end{aligned}$$

Then, the storage platform updates the ciphertext CT_d as follows:

1) $\overline{C} = C \cdot e(g^s, g^{\alpha'})$;
2) $\overline{C}'' = C'' \cdot (Q'_{d+1})^{\tau_{d+1}}$;
3) $\overline{E}' = E' \cdot E_{d+1}$;
4) Delete Q'_{d+1} and E_{d+1}.

KUpdateToken: This algorithm is used to generate the key update token for authenticated users. For the data owner, it can update the private key by choosing a random α', and update the private key as the algorithm KeyUpdate. Moreover, the data owner encrypts $g^{\alpha'}$ with the access structure corresponding to the authenticated users by using a ciphertext-policy attribute-based encryption scheme, and publishes the ciphertext CT_t, which is the ciphertext of key update token on the blockchain. Thus, the authenticated users can update their private key as the data owner, and use the private key to decrypt the updated ciphertext.

V. Security Analysis

In this section, we demonstrate that SFDS achieves the properties of secure access control, forward security, post-compromise security, and collusion-attack resistance.

In SFDS, to achieve fine-grained access control, data are encrypted with an access policy, which is defined over attributes in the system, before being outsourced. A user's private key is generated according to the attributes. A linear secret sharing scheme is used to generate a ciphertext, where the access structure (M, ρ) and a secret value s are embedded. Only users with attributes that satisfy the access policy can recover the secret s. Moreover, our scheme is built under the decisional bilinear Diffie-Hellman exponent (BDHE) assumption [9]. Due to the intractability of the BDHE problem, an adversary cannot break our scheme with an non-negligible probability.

To achieve forward security, we enable a data owner to update its private keys and the ciphertext periodically. Based on the idea of HIBE, we represent each time period with a time tag, which can be seen as the identity for each time period. According to the unidirectional property of key generation in HIBE schemes, given a private key corresponding to the current time period and an old ciphertext, an adversary cannot recover the private keys that corresponds to the previous time periods. Moreover, for the key update, we introduce a random element $g^{\alpha'}$, which makes the adversary cannot obtain the previous keys and decrypt the old ciphertexts.

For post-compromise security, when the data owner updates a private key, it randomly chooses α', and shifts the master secret key from g^{α} to $g^{\alpha+\alpha'}$. At the same time, the ciphertext is updated accordingly, i,e., the original ciphertext is multiplied by $e(g^s, g^{\alpha'})$. We can see that, an old key cannot decrypt the new ciphertext since the $\alpha-$component cannot match. Thus, our scheme can achieve post-compromise security.

For collusion-attack resistance, there are two components in a user's private key. Collusion attack means that users who do not satisfy the access policy of a ciphertext or have a different decryptable time tag with the ciphertext may combine their keys to obtain the plaintext. To avoid the collusion attacks, we tightly bind these two types of keys in a private key. To generate a key, TA randomly chooses t, which is used to compute both K_x, where $x \in S$, and $D_{1,\vec{\tau}}$. To decrypt a ciphertext, a user needs to first obtain $e(g,g)^{\beta st}$. This term can be dismissed only if it has the same $t-$element as $D_{1,\vec{\tau}}$. Thus, different users cannot collude to decrypt a ciphertext that they have no access to.

VI. Simulation Results

In this section, we numerically analyze the complexity of SFDS, and present the simulation results. The simulation is conducted on a laptop with the Intel(R) Core(TM) 2.9 GHz i7-7500U CPU and 8 GB RAM. We use the Miracl library and set the security parameter $\lambda = 128$ [10].

As described in section V, SFDS consists of 8 algorithms: Setup, KeyGen, KeyUpdate, Encrypt, Decrypt, CUpdateToken, CiphertextUpdate, and KUpdateToken. Among these algorithms, Setup and KeyGen are performed by TA. In Setup algorithm, TA needs only 1 $pairing$ and 2 exp operations. For the key generation, the computation cost is related to the size of S and the number of elements in vector $\vec{\tau}$. With l attributes in S and d elements in $\vec{\tau}$, TA needs $(T + l + 5)$ exp and $(T + 2)$ mul operations to generate a private key for the user. For data owner, to update its private key, it needs only 2 exp and 2 mul operations. For ciphertext generation, we omit the symmetric encryption, as it is fast and the computational cost depends on the size of data. To encrypt a symmetric key K, when there are three elements in the decryptable time period $\vec{\tau}_c$ and l rows in the access matrix M, the data owner needs to perform $(2T + 2l + 3)$ exp and $(T + c + 1)$ mul operations.

To recover a symmetric key K from a ciphertext, if a data owner or a user has κ attributes that satisfy the access policy of the ciphertext, it needs $(2\kappa + 2)$ $pairing$, κ exp, and $(2\kappa + 2)$ mul operations. For the generation of a ciphertext update token, if the current time period is $\vec{\tau}_d = \{\tau_1, \ldots, \tau_d\}$, the overhead for the data owner to compute the token is $(d+2)$ exp + d mul. The data owner also needs to generate a key update token, which is used to derive new keys for the authorized users. Assuming there are l rows in the access matrix, the computational cost for the key update token is $(3l + 2)$ exp + $(l + 1)$ mul. For the ciphertext update, when the decryptable time period of the ciphertext is $\vec{\tau}_d = \{\tau_1, \ldots, \tau_d\}$, the storage nodes need to perform 1 mul and 2 $pairing$ operations to finish the ciphertext update.

In the experiments, we set the whole number of time periods as 10, and the number of attributes in the system as 20. We evaluate the time overhead of TA when the number of attributes of a data owner is 20, and the time period is set to 1. Table I shows that the computational cost for key generation is about 16.4 ms. For encryption of a symmetric key, when the number of rows in the access matrix is 20, and the decryptable time period is the first time period, the computational cost is 57.5 ms. For data decryption, when the number of attributes used in decryption is 14, the decryption computational cost is 63.25 ms. For key and ciphertext update, the computational

cost for a data owner to generate a new key is 0.8 ms. With a ciphertext update token, the computational overhead for the ciphertext update is a constant, which is 42.6 ms. The complexity of SFDS and the experiment results are shown in Table I.

TABLE I. Complexity Comparison

Algorithms	Complexity	Computational Cost
KeyGen	$(T+l+5)\ exp + (T+2)\ mul$	16.4 ms
KeyUpdate	$2\ exp + 2\ mul$	0.8 ms
Encrypt	$(2T+2l+3)\ exp + (T+c+1)\ mul$	57.5 ms
Decrypt	$(2\kappa+2)\ pairing + \kappa\ exp + (2\kappa+2)\ mul$	63.25 ms
CUpdateToken	$(d+2)\ exp + d\ mul$	8.8 ms
CiphertextUpdate	$1\ mul + 2\ pairing$	42.6 ms

VII. Related Work

A. Attribute-based Encryption

Goyal et al. define the concept of attribute-based encryption (ABE) and propose two complementary forms of ABE: key-policy ABE, where ciphertext are encrypted with attributes, and users' private keys are generated based on formulas over these attributes, and ciphertext-policy ABE, where attributes are used to describe users' credentials and access policies over the attributes are attached to ciphertexts [11], [12]. Considering ABE is a one-to-many primitive, a user with the access privilege can share its secret key to others without being identified [13]. To solve this issue, Liu et al. construct a CP-ABE scheme that can support traitor tracing [14]. They propose an ABE template and demonstrate that any ABE scheme that satisfies the template can be transformed to a blackbox traceable ABE scheme.

Existing ABE schemes with user revocation can achieve fine-grained data sharing and key update. However, there is no time validity that can be predefined for the shared data.

B. Updatable Encryption

Updatable encryption allows a secret key and a ciphertext to be updated periodically [15], [16]. Specifically, a secret key is generated at each epoch, and a conversion key can be created to convert a ciphertext that is encrypted under k_u, which is the secret key at epoch u, to a ciphertext encrypted under k_{u+1}, which is the key at epoch $u+1$. Updatable encryption can be ciphertext-dependent, where the generation of an update token needs the involvement of a part of ciphertext, that is, the update token is related to the specific ciphertext to be updated [17]. Updatable encryption can also be ciphertext independent, where we can generate an update token without a ciphertext, and an update token can be used to update any ciphertext from epoch u to $u+1$ [18].

Most existing UE schemes employ symmetric encryption primitives to achieve the functionality and cannot support flexible data access control. In SFDS, fine-grained access policies can be enforced on the data, and efficient key updates and ciphertext updates are supported.

VIII. Conclusion

In this paper, we have proposed a secure and flexible data sharing scheme (SFDS) for distributed storage with efficient key management. SFDS can enable data owners to store their data at a distributed network with flexible access controls and update the ciphertexts and the respective keys periodically. Our scheme can achieve forward security, post-compromise security, and defend against collusion attacks. With the fine-grained data sharing mechanism, SFDS may shed light on further research of blockchain-based data management. For our future work, we aim to design a flexible data sharing and updating scheme with shorter key size.

References

[1] D. Vorick and L. Champine, "Sia: Simple decentralized storage," *Nebulous Inc*, 2014.

[2] D. Berdik, S. Otoum, N. Schmidt, D. Porter, and Y. Jararweh, "A survey on blockchain for information systems management and security," *Information Processing & Management*, vol. 58, no. 1, p. 102397, 2021.

[3] "Filecoin," Website, https://filecoin.io/.

[4] X. Shen, C. Huang, D. Liu, L. Xue, W. Zhuang, R. Sun, and B. Ying, "Data management for future wireless networks: Architecture, privacy preservation, and regulation," *IEEE Network*, vol. 35, no. 1, pp. 8–15, 2021.

[5] B. Waters, "Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization," in *Proc. International Workshop on Public Key Cryptography*, 2011, pp. 53–70.

[6] D. Boneh, X. Boyen, and E.-J. Goh, "Hierarchical identity based encryption with constant size ciphertext," in *Proc. International Conference on Theory and Applications of Cryptographic Techniques*, 2005, pp. 440–456.

[7] J. K. Liu, T. H. Yuen, P. Zhang, and K. Liang, "Time-based direct revocable ciphertext-policy attribute-based encryption with short revocation list," in *Proc. International Conference on Applied Cryptography and Network Security*, 2018, pp. 516–534.

[8] D. Slamanig and C. Striecks, "Puncture'em all: Stronger updatable encryption with no-directional key updates." *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 268, 2021.

[9] L. Rotem and G. Segev, "Algebraic distinguishers: from discrete logarithms to decisional uber assumptions," in *Proc. Theory of Cryptography Conference*, 2020, pp. 366–389.

[10] "Miracl library," Website, https://github.com/miracl/MIRACL.

[11] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proc. ACM conference on Computer and communications security*, 2006, pp. 89–98.

[12] L. G. Cecile Delerablee and D. Pointcheval, "Key-policy ABE with delegation of rights," 2021, https://eprint.iacr.org/2021/867.pdf.

[13] S. Wang, H. Wang, J. Li, H. Wang, J. Chaudhry, M. Alazab, and H. Song, "A fast CP-ABE system for cyber-physical security and privacy in mobile healthcare network," *IEEE Transactions on Industry Applications*, vol. 56, no. 4, pp. 4467–4477, 2020.

[14] Z. Liu, Q. Huang, and D. S. Wong, "On enabling attribute-based encryption to be traceable against traitors," *Computer Journal*, vol. 64, no. 4, pp. 575–598, 2021.

[15] A. Lehmann and B. Tackmann, "Updatable encryption with post-compromise security," in *Proc. International Conference on Theory and Applications of Cryptographic Techniques*, 2018, pp. 685–716.

[16] Y. Jiang, "The direction of updatable encryption does not matter much," in *Proc. International Conference on Theory and Application of Cryptology and Information Security*, 2020, pp. 529–558.

[17] L. Chen, Y. Li, and Q. Tang, "CCA updatable encryption against malicious re-encryption attacks," in *Proc. International Conference on Theory and Application of Cryptology and Information Security*, 2020, pp. 590–620.

[18] M. Klooß, A. Lehmann, and A. Rupp, "(R) CCA secure updatable encryption with integrity protection," in *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques*, 2019, pp. 68–99.