Privacy Enhanced Access Control for Outsourced Data Sharing

Mariana Raykova, Hang Zhao, and Steven M. Bellovin

Columbia University, Department of Computer Science,
New York, NY 10027-7003, USA,
{mariana,zhao,smb}@cs.columbia.edu

Abstract. Traditional access control models often assume that the entity enforcing access control policies is also the owner of data and resources. This assumption no longer holds when data is outsourced to a third-party storage provider, such as the *cloud*. Existing access control solutions mainly focus on preserving confidentiality of stored data from unauthorized access and the storage provider. However, in this setting, access control policies as well as users' access patterns also become privacy sensitive information that should be protected from the cloud. We propose a two-level access control scheme that combines coarse-grained access control enforced at the cloud, which allows to get acceptable communication overhead and at the same time limits the information that the cloud learns from his partial view of the access rules and the access patterns, and fine-grained cryptographic access control enforced at the user's side, which provides the desired expressiveness of the access control policies. Our solution handles both *read* and *write* access control.

1 Introduction

The emerging trend of outsourcing of data storage at third parties, such as the *cloud*, has recently attracted tremendous amount of attention from both research and industry communities. Outsourced storage make shared data and resources much more accessible as users can retrieve them anywhere from personal computers to smart phones. This alleviates data owner from the burden of data management and leaves this task to service providers with dedicated resources and more advanced techniques. By embracing the cloud computing solution, government agencies will drastically save budget and increase productivity by utilizing low-cost and maintenance-free services available on the Internet rather than purchasing, designing and installing new IT infrastructure themselves. Similar benefits could be realized in financial services, health care, education, etc.

Security remains the critical issue that concerns potential clients, especially for the banks and government sectors. A major challenge for any comprehensive access control solution for outsourced data is the ability to handle requests for resources according to the specified security policies to achieve confidentiality, and at the same time protect the users' privacy. Several solutions have been proposed in the past [5, 6, 9, 10, 12], but none of them considers privacy of the policies and users' access patterns as an essential goal. In this paper we propose a solution that addresses these privacy requirements and provides a mechanism to achieve a flexible level of privacy guarantees for the client. We introduces a two-level access control model that combines *fine-grained access control*, which supports the precise granularity for access rules, and *coarse-grained access control*, which allows the storage provider to manage access requests while learning only limited information from its inputs. This is achieved by arranging outsourced resources into units called *access blocks* and enforcing access control at the cloud only at the granularity of blocks. The fine-grained access control within each access block is enforced at the user's side and remains oblivious to the cloud. The mapping between files and access blocks is transparent to the users in the sense that they can submit file requests without knowing in what blocks the files are contained.

Moreover, existing solutions focus on read requests only and exploit the idea that data encryption implicitly enables read access control through the appropriate distribution of decryption keys. However, extending this idea to write access control is nontrivial since we cannot guarantee that a user given a specific access block will modify only the files to which he has write access. In this work, we also present a write access control solution, where the storage provider is able to verify that a user has write access to some file in the access block, without learning which file, and only then he would accept updates from the user.

2 Motivation

Traditional access control models often make an implicit assumption that the entity enforcing access control policies is also the owner of data. However, in many cases of distributed computing, this assumption no longer holds, and access control policies are enforced at points which should not have direct access to the data content itself, such as data outsourced to an untrusted third party. Hence we need to store data in encrypted form and enforce access control over the encrypted data. The setting of cloud computing falls into this category. The cloud servers are considered to be *honest* but *curious*. They will follow our proposed protocol in general, but try to find out as much information as possible based on their inputs. Therefore data confidentiality is not the only security concern.

First of all, access control policies defined by the data owner that govern who can have access to what data become private information with respect to the storage provider. For example, the fact that certain users share some of their data with other users or they stop the sharing can be suggestive about their business relationships. This problem is mitigated by the use of cryptography as an enforcement mechanism, which translates the access control problem into the question of key management for the decryption keys.

A more challenging task, that cannot be solved by data encryption alone, is to protect data access patterns from careful observations on the inputs of the storage provider. Even data is stored and transferred in an encrypted format, traffic analysis techniques can reveal sensitive information about the underlying data and the activities of users. For example, analysis on the length of encrypted traffic could reveal certain properties of the underlying data; access history to multiple data objects could disclose access habits and privileges of a particular user; access to the same data object from multiple users could suggest a common interest or collaborative relationship; a ranking of data objects popularity can also be built upon access requests that the cloud receives. A trivial solution is to return all encrypted data upon any data request. While this solution may not cause any confidentiality problem, if the cryptographic mechanism guarantees that a user can decrypt only data that he has access to, it brings prohibitive costs in several directions: data stored at a cloud provider most likely exceeds the storage capacity of any single user, transferring the data would incur enormous network usage as drastically increased latency; additionally, a user would need to spend time on cryptographic computations to find the appropriate piece of the data he needs. All these concerns rule out this obvious solution.

The question of hiding access pattern is challenging while avoiding work proportional to the total size of all stored files. There have been several cryptographic solutions that realize the notion of oblivious RAM and manage to achieve improved amortized complexity for queries while hiding access patterns [3, 8, 1, 4]. However, such solutions are highly interactive and still require communication polylogarithmic in the size of the database, which in the setting of large storage cloud providers, weak client devices and expensive network communication will not be practical (e.g., wireless network communication with limited bandwidth). Furthermore, they assume that the user submitting the query is the owner of all data, which does not fit into our scenario where access control is enforced on data shared by multiple users, not limited to the data owner.

An equally important, but often overlooked, aspect of access control for outsourced data is to enforce the *write* access. Existing solutions often made an implicit assumption that access to shard data always refer to *read* requests. However, an inevitable requirement of data sharing is to allow authorized write access, such as in a collaborative working environment, where co-workers are allowed to contribute to the same project. While data encryption naturally preserves authorization of the read access through key management, the procession of a decryption key implies authorized read access but not necessarily the write. Therefore, different cryptographic schemes are mandatory to manage read and write accesses separately. Moreover, access patterns on write operations should be hidden from the cloud provider to achieve better privacy. In addition, adding write operation into the current access control scheme complicates our problem as we do not assume any implications between read and write access rules. Thus a user may have both types of access (read and write) to a protected object or only one of them (read or write).

Therefore we summarize that a privacy enhanced access control solution for data sharing in outsourced storage needs to meet the following requirements:

1. it enables data confidentiality by implementing a fine-grained cryptographic access control mechanism;
2. it provides a practical and flexible data sharing scheme by supporting both read and write operations in the access control model;
3. it enhances data and user privacy by protecting access control rules and more importantly access patterns of users from the storage provider.

2.1 Two-level Access Control Model

We consider a scenario with a set of users that selectively perform data sharing among themselves on a set of resources using remote storage provided by the cloud. Three different roles are distinguished in the access control model: the *data owner* who creates data to be stored at the remote storage in an encrypted format and regulates who has what access to each data; the *data user* who may have read and write access to the protected data; the *cloud provider* that stores the encrypted data and responds to access requests. To provide confidentiality through data encryption while preserving privacy, we propose a two-leveled access control model, where data resources (files) are divided into units called *access blocks*.[1] These access blocks constitute the *coarse-grained level* view of the stored data. The cloud provider is presented with this view and enforce access control at this granularity. He is able to match an authorized request to an access block that contains the requested file. Upon a read request the cloud would provide the content of the entire matching block to the user. Upon a write request he shall accept only authorized updates for some content of that block and also obliviously match them to the corresponding files. At the fine-grained level, each access block consists of files owned by a single owner. Each data owner is responsible for distributing his files into blocks, and defines fine-grained access control policies that specify users' access rights to individual files.

To facilitate our discussion, consider a system with five users $U = \{A, B, C, D, E\}$. Let R_o denote the set of resources owner by user $o \in U$, and we have $R_A = \{r_1, r_2, r_3, r_4\}$, $R_B = \{r_5, r_6, r_7\}$ and $R_C = R_D = R_E = \emptyset$. An authorization policy P_o defined by data owner o at the fine-grained level is a set of tuples of form $\langle u, r, p\rangle$ $(p \in \{r, w\})$, which states a data user $u \in U$ is allowed to *read* (r) or *write* (w) to resource $r \in R$. In our example, we have:

1. $P_A = \{\langle A, r_1, r\rangle, \langle B, r_1, r\rangle, \langle C, r_1, r\rangle, \langle A, r_2, r\rangle, \langle B, r_2, r\rangle, \langle C, r_2, r\rangle, \langle A, r_3, r\rangle, \langle E, r_3, r\rangle, \langle A, r_4, r\rangle, \langle B, r_4, r\rangle, \langle C, r_4, r\rangle, \langle E, r_4, r\rangle, \langle A, r_1, w\rangle, \langle B, r_1, w\rangle, \langle C, r_1, w\rangle, \langle A, r_2, w\rangle, \langle B, r_2, w\rangle, \langle C, r_2, w\rangle, \langle A, r_3, w\rangle, \langle A, r_4, w\rangle, \langle D, r_4, w\rangle\}$;
2. $P_B = \{\langle A, r_5, r\rangle, \langle B, r_5, r\rangle, \langle B, r_6, r\rangle, \langle C, r_6, r\rangle, \langle D, r_6, r\rangle, \langle A, r_5, w\rangle, \langle B, r_5, w\rangle, \langle A, r_7, r\rangle, \langle B, r_7, r\rangle, \langle C, r_7, r\rangle, \langle D, r_7, r\rangle, \langle E, r_7, r\rangle, \langle C, r_5, w\rangle, \langle B, r_6, w\rangle, \langle D, r_6, w\rangle, \langle E, r_6, w\rangle, \langle A, r_7, w\rangle, \langle B, r_7, w\rangle, \langle C, r_7, w\rangle, \langle D, r_7, w\rangle, \langle E, r_7, w\rangle\}$.

Therefore, we can build the following set of access control lists (ACLs):

1. $acl\_read(r_1) = \{A, B, C\}$, $acl\_write(r_1) = \{A, B, C\}$;
2. $acl\_read(r_2) = \{A, B, C\}$, $acl\_write(r_2) = \{A, B, C\}$;
3. $acl\_read(r_3) = \{A, E\}$, $acl\_write(r_3) = \{A\}$;
4. $acl\_read(r_4) = \{A, B, C, E\}$, $acl\_write(r_4) = \{A, D\}$;
5. $acl\_read(r_5) = \{A, B\}$, $acl\_write(r_5) = \{A, B, C\}$;
6. $acl\_read(r_6) = \{B, C, D\}$, $acl\_write(r_6) = \{B, D, E\}$;
7. $acl\_read(r_7) = \{A, B, C, D, E\}$, $acl\_write(r_7) = \{A, B, C, D, E\}$.

Note that for each resource r owned by user o, we have $acl\_read(r) \cap acl\_write(r) \supseteq \{o\}$. That is the owner of a resource automatically entails both read and write access privilege. At the coarse-grained level, user A maintains two blocks $b_1 = \{r_1, r_2\}$ and $b_2 = \{r_3, r_4\}$, and user B maintains a single block $b_3 = \{r_5, r_6, r_7\}$.

[1] They are by no means related to file disk blocks.

3 Read Access Control

In this section, we present in detail the two-level access control scheme for read access only after describing the following techniques applied in our protocol.

3.1 Techniques

Fine-Grained Access Control Fine-grained access control is applied to files inside each access block to explicitly enforce access control rules. While the cloud provider is able to determine whether a user submits a legitimate request for some file within a block, he should remain oblivious to the access control rules defined for that file. To guarantee this property the access control view presented to the cloud treats blocks as entities, and the cloud grants a read access by providing the content of an entire block. Fine-grained access control is enforced by encrypting files per block under different keys, and the access control problem is mitigated to appropriate key distribution. Even a user receives the encrypted content of a block, he is able to decrypt only the files that he has access to. Access revocation requires re-encryption of the resource and re-distribution of the new key to the remaining authorized users. Our goal is to minimize the amount of work and interaction between users and the system upon policy updates.

The work of [10] proposes an encryption-based access control solution for outsourced data. Their key distribution is facilitated by the construction of a public tree structure that allows each user to derive file decryption keys using a secret, which he establishes once in the beginning. The leaf nodes in the tree represent initial secrets distributed to users when they join the system, and the internal nodes denote the file decryption keys derivable from leaf nodes using public tokens along a directed path. Any update of the access control rules entails a change in the tree. Access revocation requires re-encryption of affected files.

In our scheme, each user generates a public-private key pair, and the public key is used by data owners as initial secret to construct trees. Hence each user only needs to maintain one key in the system and the distribution of leaf nodes is implicit through the asymmetric key scheme. If some resources are only accessible by a single user, instead of encrypting files using a leaf node (i.e., the public key), we generate a symmetric key for file encryption, which is further encrypted under that public key to avoid expensive computation of asymmetric scheme. In that case, the initial secret needs to be explicitly distributed to an authorized user. Unlike in [10], key derivation tokens have to be protected. First a user's initial secret is generally available to anyone in the system. More importantly, the tree structure itself can reveal certain sensitive information to the cloud. For example, a user having access to one file will have access to all the files along a directed path. So the ontent of each node, a pointer to next node and the token to derive next key are all protected under the current encryption key. Thus we can achieve efficient key distribution without requiring any direct interaction between data owners and users beyond some initial set-up assuming only the cloud will be online all the time. A list of algorithms for key distribution and management are summarized in Figure 1.

Coarse-Grained Access Control The main goal to achieve at the level of coarse-grained access control is to enable the cloud provider to obliviously match

- **Publish**(r, o, e_o, acl): adds a resource r owned by o with a secret e_o and an access control list $acl = acl\_read(r)$ for read access.
- **Access_Read**(u, r, o): returns the encryption key for a resource r owned by o, if u is an authorized user.
- **Find_Chain**(u, r): finds the shortest chain of tokens from the secret key of user u to derive the decryption key for resource r.
- **Compute_Key**$(u, chain)$: derives the secret key for a user u given a chain of transition tokens.
- **Find_Resources**(u, r): finds the set of nodes that lie on any path from the user u to the node corresponding to resource r.
- **Update**(r, acl): if there is another resource with the same access control list acl, i.e., there is a node in the tree accessable exactly by a subset of users in acl, then encrypt r with the key contained in that node. Otherwise, encrypt r with a new key, add a new node containing this key to the tree and add appropriate edges to connect the new node to the users who have access to r. (Note that certain subgroups of the users in acl might already have a shared key through another node in tree, and in that case we connect to that node rather than all the users' nodes separately.)

Fig. 1: Algorithms for key distribution and management for fine-grained AC.

a user's request to an access block without learning which part of the block the user is authorized to access. In addition we provide *unlinkability* among multiple requests for the same resource even if coming from the same user, which further protects users' access patterns from the cloud provider. In order to achieve these goals we apply the predicate encryption scheme of [7]. Observing that in this scheme ciphertext can be re-randomized even without knowledge of the secret key, we define a re-randomization algorithm in Definition 1.

Definition 1. *A re-randomizable predicate encryption scheme consists of the following algorithms:*

- Setup(1^n)*: produces a master secret key* SK *and public parameters;*
- Enc$_{SK}(x)$*: encrypts an attribute* x *using key* SK*;*
- GenKey$_{SK}(f)$*: generate a decryption key* SK_f *associated with a function* f*;*
- Dec$_{SK_f}(c)$*: outputs* 1 *if the attribute encrypted in* $c = Enc_{SK}(x)$ *satisfies* f*, i.e.* $f(x) = 1$*, and output a random value, otherwise;*
- Rand(c)*: computes a new encryption* c' *of the value encrypted in* c *but with different randomness* without the secret key*.*

We present the predicate encryption scheme of [7] and the instantiation of the function Rand(c) for that scheme in Appendix A. This scheme handles a class of functions f, which includes polynomials of bounded degree. We use polynomial functions of the type $f(x) = (x - id_1) \cdots (x - id_n)$, to implement coarse-grained access control. Figure 2 present a list of algorithms to enforce access control on the block level granularity without revealing the exact files that are being accessed insider a block. The algorithm **File_Access_Check** grants access if the submitted access token matches any of the files in the block without revealing the file identity. The request token produced by **File_Access_Request** is an encryption that does not leak information about the file id it contains.

- **Block_Access_Setup:** data owner runs $Setup(1^n)$, publishes the public parameters and keeps the master secret key SK. For files $id_1, \ldots, id_n$ in each block, he computes $SK_f = \text{GenKey}_{SK}(f)$ for $f(x) = (x - id_1) \cdots (x - id_n)$ and sends SK_f to the cloud provider.
- **File_Access_Authorization:** data owner provides access to a file id by sending $c_{id} = \text{Enc}_{SK}(id)$ to an authorized user.
- **File_Access_Request:** user generates a token $t_{id} = \text{Rand}(c_{id})$ for file id.
- **File_Access_Check:** upon receiving a request token t, the cloud computes $\text{Dec}_{SK_f}(t)$ for each block, and returns those blocks that compute to 1.

Fig. 2: Algorithms for enforcing coarse-grained AC at the access block level.

3.2 Read Access Control Alone

We now present a complete access control solution, for read access only, that consists of the following algorithms:[2]

– **System Setup:** At the *fine-grained level*, files are distributed into access blocks. For each block, generate a tree graph by running **Publish**(r, o, e_o, acl) for each resource r owned by o with an initial set of ACLs. Encrypt resources using keys from internal nodes in the tree. At the *coarse-grained level*, compute parameters for a predicate encryption scheme. Each owner construct a separate tree graph over all resources he owns to distribute authorization tokens $SK_f = \text{GenKey}_{SK}(f)$ based on the initial ACLs.

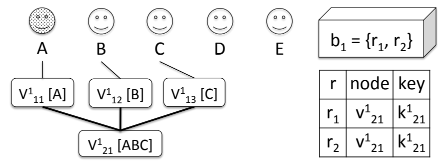
– **Access Authorization:** At the *fine-grained level*, add a leaf node containing the new user's public key to the corresponding tree graph with encryption keys. Update the graph by adding new internal nodes and appropriate edges if necessary. Update file encryptions if new internal nodes were added previously. At the *coarse-grained level*, perform similar operations with respect to the tree graph containing read access tokens.

– **Access Request:** At the *fine-grained level*, an authorized user u derives the decryption key from tree graph for resource r by calling **Find_Chain**(u, r), **Find_Resources** (u, r) and **Compute_Key**$(u, chain)$. At the *coarse-grained level*, he calls the same set of functions but to query the tree graph with access tokens and find token $c_{id} = \text{Enc}_{SK}(id)$ for the requested file id, and then submit a randomized token $t_{id} = \text{Rand}(c_{id})$ to the cloud.

– **Access Check:** At the *fine-grained level*, only authorized users can derive the correct decryption key for each file using the public tree structure. At the *coarse-grained level*, the cloud provider executes **File_Access_Check** to identify the block that contains the requested file.

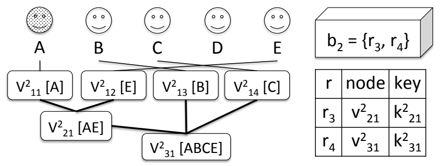
– **Access Rule Update:** At the *fine-grained level*, changes are applied immediately upon policy updates. If the policy update involves access revocation, the data owner changes the encryption of corresponding files. The data owner identifies the blocks affected by those files and updates their tree graphs with decryption keys. The changes at the *coarse-grained level* happen at longer intervals of time, the length of which would depend on the resources of the data owner. They involve updating of the tree graph with access tokens.

[2] Unless explicitly stated, all the actions are performed by individual data owners.



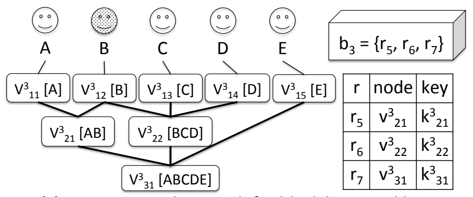
r	node	key
r_1	v^1_{21}	k^1_{21}
r_2	v^1_{21}	k^1_{21}

(a) Encryption policy graph for block b$_1$ owned by A



r	node	key
r_3	v^2_{21}	k^2_{21}
r_4	v^2_{31}	k^2_{31}

(b) Encryption policy graph for block b$_2$ owned by A

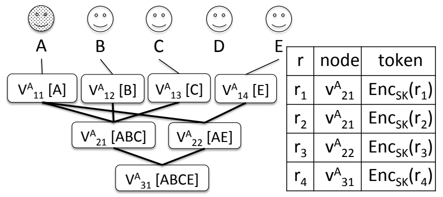


r	node	key
r_5	v^3_{21}	k^3_{21}
r_6	v^3_{22}	k^3_{22}
r_7	v^3_{31}	k^3_{31}

(c) Encryption policy graph for block b$_3$ owned by B

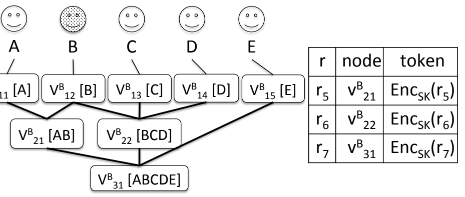
Fig. 3: Tree graphs of encryption policy for fine-grained AC on read access.

Given the example in Section 2, we construct one tree graph per block for file encryption keys at the fine-grained level in Figure 3. Each block stores files owned by a single user (shaded), and each file r is encrypted under a symmetric key. Leaf nodes, indexed as v^j_{1n} with block id j, store users' initial public keys. Key derivation paths are denoted using thick links connecting leaf nodes to internal ones. Each row in the table states resource r_i in block b_j encrypted under key k^j_{mn} at vertex v^j_{mn}. The tree structure significantly reduces the number keys that each user has to maintain, and enables encryption of different files with the same ACL under the same key. For example, key k^3_{31} for encrypting r_7 can be derived by all users, since there is a directed path from each user's initial secret to vertex v^3_{31}; resources r_1 and r_2 are encrypted under the same key k^1_{21} since $acl\_read(r_1) = acl\_read(r_2) = \{A, B\}$.



r	node	token
r_1	v^A_{21}	$Enc_{SK}(r_1)$
r_2	v^A_{21}	$Enc_{SK}(r_2)$
r_3	v^A_{22}	$Enc_{SK}(r_3)$
r_4	v^A_{31}	$Enc_{SK}(r_4)$

(a) Read access token graph for owner A



r	node	token
r_5	v^B_{21}	$Enc_{SK}(r_5)$
r_6	v^B_{22}	$Enc_{SK}(r_6)$
r_7	v^B_{31}	$Enc_{SK}(r_7)$

(a) Read access token graph for owner B

Fig. 4: Distribution of read access tokens $Enc_{SK}(r_i)$ for coarse-grained AC.

Figure 4 depicts a tree graph per data owner to distribute read access tokens at the coarse-grained level. Each row in the table states resource r_i is associated with access token $Enc_{SK}(r_i)$ stored at vertex v_{mn}. Unlike in Figure 3, each r_i is associated with an unique read access token encrypted on its *id*. For example, r_1 and r_2 are now given different access tokens at vertexes v_{21} and v_{22} respectively.

4 Write Access Control

Enforcing write access control presents more challenges, mainly for the fact that access control through data encryption does not apply to cases when data can be modified. Without revealing fine-grained access control policies, it is not guaranteed that a user will modify only files that he is granted write access to. An unauthorized user can overwrite and destroy data without being detected by the cloud, regardless of whether he has the read privilege. A trivial solution is to rely on the cloud provider to restrict the memory regions to which users may submit changes, which however reveals to the cloud access rules and access patterns.

The approach that we adopt is to record modifications of files in new regions of memory without overwriting previous content. The coarse-grained access control enforced by the cloud allows users to submit changes for files only if they can demonstrate write permission for some resource in that block, without revealing the exact content to be changed. At the fine-grained level, a public encryption scheme is used to separate read and write privileges by providing a key pair for each file. The only information that the cloud tags to each file change obliviously contains implicit information on file identifier. However, using a submitted write authorization token directly as an update identifier will enable users with only read access to copy and reuse it later to obtain write privilege. To prevent this undesired situation, we take advantage of the predicate encryption ciphertexts constituting access tokens, which allows us to use part of the token as identifier. We generate predicate that allows users with read access to identify relevant updates, but this identifier on its own is not sufficient to grant write access.

4.1 Techniques

File Encryption We apply an asymmetric encryption scheme to handle all possible combinations of read and write access to a file. Since such scheme is computationally expensive for large size of data, file content is still encrypted using a symmetric key (e.g., AES), which is further encrypted under the public key. Two trees are constructed for key distribution per block – one for the public (encryption) keys and the other for the private (decryption) keys. These two trees share the same set of internal nodes for an one to one correspondence between public and private key pair. Only files readable and writable by the same set of users can share the same public key pair.

Access Authorization Tokens Two trees are constructed by each data owner for the distribution of read and write access tokens respectively.

File Identifiers for Write Updates We observe that the write authorization token is a valid encryption for a predicate encryption that provides polynomials evaluation, and the structure of the encrypted plaintext for access to file id is a vector of the form $(1, id, id^2, \dots, id^n)$, where n is the number of files placed in a block. The structure of the ciphertext allows it to be split into parts where one part is an encryption of the vector $(1, id, id^2, \dots, id^k)$ $(k < n,\ n > 2)$, which is no longer a valid write access token for that file, but can still be used identify file updates for users with read privilege. This can be achieved using a decryption predicate for a polynomial of degree k that has id as a zero point. (See Appendix A for details.)

4.2 Integrated Read and Write Access Control

We realize the above proposal for the write access control enforcement and describe an integrated solution for both read and write access:

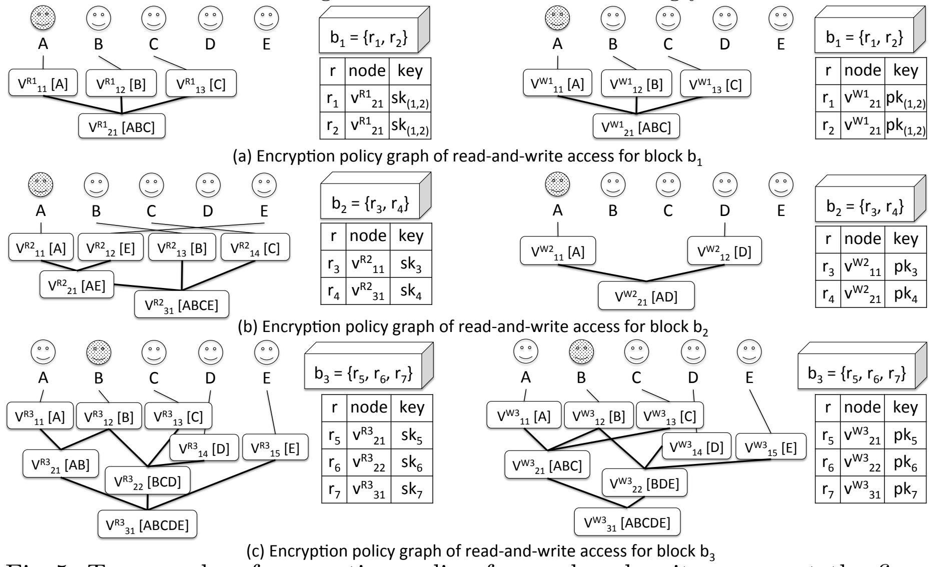
– **Setup:** At the *fine-grained level*, construct a key distribution tree per block based on read access rules. For each node in the tree, generate a public-private key pair (sk_n, pk_n), but only store the secret key sk_n. Construct another tree with the same set of nodes to store the public key pk_n, with edges determined by write access rule. For each file id generate a AES key sk_{id}^{aes} for encryption, and append to the ciphertext $\text{Enc}_{pk_n}(sk_{id}^{aes})$. At the *coarse-grained level*, each data owner generates a tree graph, where each node contains read access token $\text{Enc}_{pk'_{ra}}(id)$ and $SK_{x-id} = \text{GenKey}_{sk''-ra}(f)$ where $f(x) = x - id$ using predicate encryption with different keys. Similarly, construct another tree to distribute write access tokens $\text{Enc}_{pk_{wa}}(id)$.

– **Access Authorization:** At the *coarse-grained level*, extend the trees with read and write access tokens with new leaves for the new user and update the edges according to his read and write permissions. This may involve splitting of nodes and re-encrypting files with new keys if the user has read access only to a subset of files that have been encrypted with the same key.

– **Write Access Request:** At the *fine-grained level*, obtain the encryption key pk_n for the file to be updated from the write tree. Encrypt the new content for that file with key pk_n. At the *coarse-grained level*, submit to the cloud a re-randomized copy of the write authorization token for that file.

– **Write Access Check:** At the *fine-grained level*, a user can modify a file only if he has the encryption key and the write authorization token. Upon read he will check at the end of a block a list of updates with valid write access tokens. At the *coarse-grained level*, the cloud finds if there is a block for which the authorization token grants write access. The write access token is of the form $(C_0, \{C_{1,i}, C_{2,i}\}_{i=1}^{n})$, and the cloud uses the first components $(C_0, \{C_{1,i}, C_{2,i}\}_{i=1}^{2})$ as an identifier for updates appended to a block.

– **Write Access Rule Update:** Update per-block trees for encryption keys and the tree for distributing write access tokens accordingly.



r	node	key
r_1	v^{R1}_{21}	$sk_{(1,2)}$
r_2	v^{R1}_{21}	$sk_{(1,2)}$

r	node	key
r_1	v^{W1}_{21}	$pk_{(1,2)}$
r_2	v^{W1}_{21}	$pk_{(1,2)}$

(a) Encryption policy graph of read-and-write access for block b_1

r	node	key
r_3	v^{R2}_{11}	sk_3
r_4	v^{R2}_{31}	sk_4

r	node	key
r_3	v^{W2}_{11}	pk_3
r_4	v^{W2}_{21}	pk_4

(b) Encryption policy graph of read-and-write access for block b_2

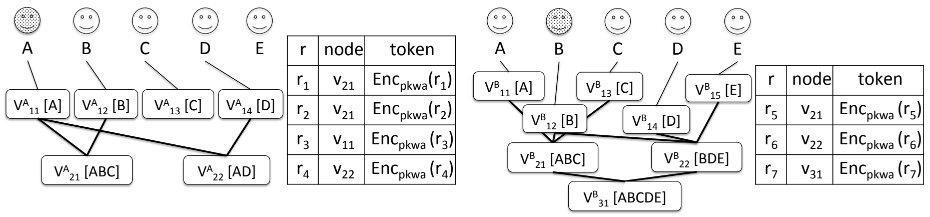
r	node	key
r_5	v^{R3}_{21}	sk_5
r_6	v^{R3}_{22}	sk_6
r_7	v^{R3}_{31}	sk_7

r	node	key
r_5	v^{W3}_{21}	pk_5
r_6	v^{W3}_{22}	pk_6
r_7	v^{W3}_{31}	pk_7

(c) Encryption policy graph of read-and-write access for block b_3

Fig. 5: Tree graphs of encryption policy for read and write access at the fine-grained level for each access block.

Following our example, we draw two trees per block for read and write access respectively at the fine-grained level in Figure 5. A public key pair is generated for each resource r_i, where pk_{r_i} is stored in the read tree (v_{mn}^{Rj}) and sk_{r_i} is stored



r	node	token
r_1	v_{21}	$Enc_{pkwa}(r_1)$
r_2	v_{21}	$Enc_{pkwa}(r_2)$
r_3	v_{11}	$Enc_{pkwa}(r_3)$
r_4	v_{22}	$Enc_{pkwa}(r_4)$

r	node	token
r_5	v_{21}	$Enc_{pkwa}(r_5)$
r_6	v_{22}	$Enc_{pkwa}(r_6)$
r_7	v_{31}	$Enc_{pkwa}(r_7)$

(a) Write access token graph for owner A   (a) Write access token graph for owner B

Fig. 6: Distribution of write access tokens $Enc_{pk_{wa}}(r_i)$ in coarse-grained AC.

in the write tree (v^{Wj}_{mn}). Different ACLs on read and write for the same resource entail different labels of user list, e.g., v^{R2}_{21} is labeled as $[ABCE]$ since $acl\_read(r4) = \{A, B, C, E\}$; whereas v^{W2}_{21} is labeled as $[AD]$ given $acl\_write(r4) = \{A, D\}$. In Figure 5(a), both trees share the same set of vertexes and edges, as $acl\_read(r_1) = acl\_read(r_2) = acl\_write(r_1) = acl\_write(r_2) = \{A, B, C\}$. Figure 6 depicts one tree graph per data owner for distributing the write authorization tokens $Enc_{pk_{wa}}(r_i)$ at the coarse-grained level. Each authorized write operation requires an additional update token SK_{x-r_i}, distributed the same way as read access tokens in Figure 4.

5 Analysis

5.1 Security Guarantees

Our two-leveled access control scheme provides the following privacy guarantees for data owners and users in the system:

Read Access For the privacy of the data owners, the cloud provider does not learn any of the content of the files that he stores. The cloud learns the frequency of access to particular blocks but not the exact files that have been accessed within a block. For users' privacy, the cloud provider cannot relate access requests to particular users', neither can he infer which requests were submitted from the same user. However, he can observe the block access pattern from the requests of all users. The data owner does not learn anything about the access requests for the data.

Write Access For privacy of the data owners, the cloud provider learns how often update requests are submitted for each block but without finding out which files have been written. Similarly to the read requests, write requests coming from the *users* are anonymous and unlinkable. Thus the cloud provider cannot learn anything about the access behavior of a particular user, but only a cumulative view over the requests from all users.

5.2 Performance Analysis

Read Access During setup, *data owners* compute of the authorization trees with decryption keys and access tokens. The work is proportional to the number of files in the database and the number of users. To authorize file access to a user, data owner updates the tree with decryption keys and the tree with access tokens: in the worst case proportional to the depth of the trees. To revoke file access of a user, data owner updates the tree with decryption tokens: in the worst case proportional to the depth of the tree. The updates for the tree with the access tokens can be executed at larger intervals of time to achieve better amortized efficiency for updates. For regular *users*, retrieving access tokens requires reading the coarse-level tree with access tokens for the data of a particular provide. Decryption keys retrieval can be proportional to the number of files that the user is authorized to access. However, once retrieved the access credentials for all files can be stored locally and used directly for subsequent requests. New credentials will have to be retrieved only when there has been an update of the access trees that has involved change of the credentials relevant for the particular user. For the *cloud provider*, in order to map an access request to a particular block the cloud provider will have to execute the **File_Access_Check** function for the submitted token and each block. This cost can be reduced if after the first retrieval of a file the user remembers the block id that contained the file, and the next time he need to access the same file he also submits this id. In this case the cloud provider would need to run a single check and verify that the block pointed in the access request does really contain the file of interest. Applying this optimization for efficiency reveals some additional information to the cloud, namely allows him distinguish first time access requests from repeated request, however, still without him being able to link requests to the same file. Further the user can choose whether to submit the block identifier that he has in repeated requests to weaken the additional leakage to the cloud.

Write Access From the perspective of a *data owner*, the enforcement of write access control requires duplication of the tree structures that were necessary for the read access control but this time with credentials necessary for the write access. This comes as an overhead in the setup phase when these structures are computed by the data owner and also each update of the access rules will necessitate update of both types of trees since the encryption and decryption (relevant for write and read access) need to be synchronized. Also periodically the data owner would need to process the blocks and compact the updates for each file back in its initial memory location. For a *user*, the size of the blocks that he receives will increase depending on the frequency of the updates for a block as well as the time period at which the data owner processes the blocks and brings the updates back in place. At read access the user would need to locate both the initial place of the file he is looking for as well as all updates that have been submitted for that file, and then reconstruct the most recent version of the file. The *cloud provider* would need to transfer larger blocks including both the original files as well as the updates. He would need to compute the identification tag for each authorized write update, which requires constant time.
Optimizations. Some optimizations that help improve the performance of the scheme are as follows. If the user has enough memory, he can cache both authorization tokens and decryption keys for multiple accesses of the same files. This optimization applies to the read and the write access tokens as well as the decryption key for read. The only exception is the encryption key for write access — the user should always derive the current public encryption key for the file, which he wants to update since if the key has been changed, he will not be able to detect it and will submit an invalid update. Similarly the user can cache the identifier of the block in which a file is located and use it in repeated requests,

which will save the search time at the cloud avoiding checks of all blocks. Further the user can trade-off the privacy guarantee for his request within its block for smaller communication overhead by revealing the exact memory address of the file after proving that he is authorized to access the block.

5.3 Discussion

Choosing the granularity for the access blocks in the read and write access control schemes affects the privacy guarantees for the scheme as well as its efficiency performance. The right granularity for each specific usage scenario will depend on the privacy and efficiency requirements for it, the expected patterns of access to the files and the expected frequency of access control rules' updates. The following points should be taken into consideration when choosing how to divide the files into access blocks: the size of a block should depend on the expected bandwidth of the clients and the acceptable delays for the system. Files that contain "complementary" information, i.e., a user is likely to access only one of a these files (e.g. a file to sell stocks, a file to buy stocks) should be located in the same block since their access pattern is highly sensitive. Data that requires often update should be split into smaller blocks since the size of those blocks will grow faster. Accessing files with frequently changing access rules will require derivation of the corresponding decryption keys, which is proportional to the number of files in the block, such files should be located in blocks with fewer items (that can still be of big size). Since the view of the cloud provider of the access requests amounts to the frequency at which each access block is matched, files that are expected to have high access rates should be distributed across different blocks.

6 Related Work

Existing access control solution in outsourced storage usually apply cryptographic methods by disclosing data decryption keys only to authorized users. [6] proposed a cryptographic storage system, called Plutus, which arranges files with similar attributes into filegroups, applying two-level file encryption and distinguishes read and write access. [5] designed a secure file system to be layered over insecure network and P2P file system, like NFS. Each file is attached a meta data containing the file's access control list. [12] defines and enforces fine-grained access control policies based on data attributes, and delegates most of computation tasks to untrusted cloud server without disclosing data content. [9] proposed a cloud storage system, called CloudProof, that enables meaningful security Service Level Agreements (SLAs) by providing a solution to detect violations of security properties, namely confidentiality, integrity, write-serializability, and read freshness. The problem presented in this paper shares some similarity with the proposals in [11, 2]. [11] introduced a practical oblivious data access protocol using pyramid-shaped database layout and an enhanced reordering techniques to ensure access pattern confidentiality. [2] proposed a shuffle index structure, adapting traditional B-tree, to achieve content, access and pattern confidentiality in the scenario of outsourced data. All those proposals focus on one or more aspects, such as scalability, efficiency, minimizing key distribution, etc., but none of them consider privacy issues as well as write access control.

7 Conclusion

We presented a two-level access control scheme enabling data sharing in outsourced storage, like the cloud environment. The fine-grained and the coarse-grained access control schemes complement each other to achieve both data confidentiality and privacy protection on access patterns. To the best of our knowledge, we are the first to handle both read and write access rights entailing a more practical data sharing solution. As the following work, we will conduct experiments on a full implementation of our scheme. As a more ambitious goal, we would like to further extend our scheme for a complete solution that guarantees both security and privacy protection for a remote file storage system.

References

1. Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious ram without random oracles. In *TCC*, 2011.
2. S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Efficient and private access to outsourced data. In *Proc. of the 31st International Conference on Distributed Computing Systems (ICDCS 2011)*, Minneapolis, Minnesota, USA, June 2011.
3. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):473, 1996.
4. Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *in Proc. International Colloquium on Automata, Languages and Programming, ICALP'11*, 2011.
5. Eu jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. Sirius: Securing remote untrusted storage. In *in Proc. Network and Distributed Systems Security (NDSS) Symposium 2003*, pages 131–145, 2003.
6. Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *USENIX Conference on File and Storage Technologies*, 2003.
7. Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology*, EUROCRYPT'08, 2008.
8. B. Pinkas and T. Reinman. Oblivious RAM Revisited. *Advances in Cryptology–CRYPTO 2010*, pages 502–519, 2010.
9. Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. Enabling security in cloud storage slas with cloudproof. In *in Proc. USENIX Annual Technical Conference ATC'11*, 2011.
10. S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi, and P. Samarati. Encryption-based policy enforcement for cloud storage. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems Workshops*, ICDCSW '10, pages 42–51, 2010.
11. Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 139–148, New York, NY, USA, 2008. ACM.
12. Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, 2010.

A Predicate Encryption and Extensions

We present the construction of predicate encryption of [7] as follows:

– Setup(1^n): Choose primes p, q and r and groups $\mathbf{G}_p$, $\mathbf{G}_q$ and $\mathbf{G}_r$ with generator g_p, g_q and g_r respectively. Let $\mathbf{G} = \mathbf{G}_p \times \mathbf{G}_q \times \mathbf{G}_r$. Choose $R_{1,i}, R_{2,i} \in \mathbf{G}_r$, $h_{1,i}, h_{2,i} \in \mathbf{G}_p$ uniformly at random for $1 \leq i \leq n$ and $R_0 \in \mathbf{G}_r$. The public parameters for the scheme are $(N = pqr, \mathbf{G}, \mathbf{G}_T, e)$. The public key PK and master secret key SK are defined as follows:
$$PK = (g_p, g_r, Q = g_q \cdot R_0, \{H_{1,i} = h_{1,i} \cdot R_{1,i}, H_{2,i} = h_{2,i} \cdot R_{2,i}\}_{i=1}^n),$$
$$SK = (p, q, r, g_q, \{h_{1,i}, h_{2,i}\}_{i=1}^n).$$
– Enc$_{SK}(x_1, \ldots, x_n)$: Choose randoms $s, \alpha, \beta \in \mathbf{Z}_N$, $R_{3,i}, R_{4,i} \in \mathbf{G}_r$ for $1 \leq i \leq n$, then output the following ciphertext:
$$C = \big(C_0 = g_p^s, \{C_{1,i} = H_{1,i}^s \cdot Q^{\alpha \cdot x_i} \cdot R_{3,i}, C_{2,i} = H_{2,i}^s \cdot Q^{\beta \cdot x_i} \cdot R_{4,i}\}_{i=1}^n\big).$$
– GenKey$_{SK}(v_1, \ldots, v_n)$: Choose randoms $r_{1,i}, r_{2,i} \in \mathbf{Z}_p$ for $1 \leq i \leq n$, $R_5 \in \mathbf{G}_r$, $f_1, f_2 \in \mathbf{Z}_q$ and $Q_6 \in \mathbf{G}_q$, then output $SK_{\mathbf{v}}$ that consists of
$$\Big(K = R_5 \cdot Q_6 \cdot \prod_{i=1}^n h_{1,i}^{-r_{1,i}} \cdot h_{2,i}^{-r_{2,i}},\ \{K_{1,i} = g_p^{r_{1,i}} \cdot g_q^{f_1 \cdot v_i}, K_{2,i} = g_p^{r_{2,i}} \cdot g_q^{f_2 \cdot v_i}\}_{i=1}^n\Big)$$
– Dec$_{SK_f}(c)$: The decryption algorithm outputs 1 if and only if
$$e(C_0, K) \prod_{i=1}^n e(C_{1,i}, K_{1,i}) \cdot e(C_{2,i}, K_{2,i}) = 1.$$

We define an algorithm called Rand(C) that re-randomizes any ciphertext produced by the predicate encryption. Given a ciphertext of form $(C_0, \{C_{1,i}, C_{2,i}\}_{i=1}^n)$, choose a random $s' \in \mathbf{Z}_N$ and output $C' = C_0 \cdot g_p^{s'}, \{C_{1,i} \cdot H_{1,i}^{s'}, C_{2,i} \cdot H_{2,i}^{s'}\}_{i=1}^n$. The resulting ciphertext is the same as freshly generated ciphertext for the encrypted value using random value $s + s'$, if s was the value used in C.

Now we look closely at the instantiation of the predicate encryption scheme that handles polynomial evaluation as its predicate. In this case the predicate $(v_1, \ldots, v_n)$ consists of the coefficients of the polynomial that is being evaluated and the attribute vector that is used for an evaluation point x is of the form $(1, x, x^2, \ldots, x^{n-1})$. The ciphertext for the encryption of $(1, x, x^2, \ldots, x^{n-1})$ has components $(C_0, \{C_{1,i}, C_{2,i}\}_{i=1}^n)$, where $C_{1,i}, C_{2,i}$ correspond to the vector point x^{i-1}. Thus we can view the first view components of the ciphertext $(C_0, \{C_{1,i}, C_{2,i}\}_{i=1}^2)$ as an encryption of the vector $(1, x)$ that can be used for evaluation of predicates that are linear functions.

We use the above observation in the instantiation of the tags that the cloud derives for each of the accepted write updates. He uses the token that the client has used to prove his write access to a particular block, which a predicate encryption ciphertext $(C_0, \{C_{1,i}, C_{2,i}\}_{i=1}^n)$, to derive identifier for the files with which the submitted update will be associated by taking the first part of the ciphertext $(C_0, \{C_{1,i}, C_{2,i}\}_{i=1}^2)$. This identifier cannot be used as a write access token since it is missing substantial part of the ciphertext, and no party without the master secret key can extend an identifier to a valid write token. Also any party that has read access to the file associated with the update will be given a key that would allow it to recognize the updates for that file. This key is the predicate corresponding to the linear function that evaluates to zero at the file id.