

K2C: Cryptographic Cloud Storage With Lazy Revocation and Anonymous Access ^{*}

Saman Zarandioon¹, Danfeng (Daphne) Yao², and Vinod Ganapathy¹

¹ Department of Computer Science, Rutgers University,
Piscataway, NJ 08854

{samanz, vinodg}@cs.rutgers.edu

² Department of Computer Science, Virginia Tech,
Blacksburg, VA 24060

danfeng@cs.vt.edu

Abstract. Security and privacy concerns hinder the adoption of cloud storage and computing in sensitive environments. We present a user-centric privacy-preserving cryptographic access control protocol called *K2C* (Key To Cloud) that enables end-users to securely store, share, and manage their sensitive data in an untrusted cloud storage anonymously. *K2C* is scalable and supports the lazy revocation. It can be easily implemented on top of existing cloud services and APIs – we demonstrate its prototype based on Amazon S3 API.

K2C is realized through our new cryptographic key-updating scheme, referred to as *AB-HKU*. The main advantage of the *AB-HKU* scheme is that it supports efficient delegation and revocation of privileges for hierarchies without requiring complex cryptographic data structures. We analyze the security and performance of our access control protocol, and provide an open source implementation. Two cryptographic libraries, Hierarchical Identity-Based Encryption and Key-Policy Attribute-Based Encryption, developed in this project are useful beyond the specific cloud security problem studied.

Keywords: Cloud, Untrusted Storage, Access Control, Mashup, Security, Web.

1 Introduction

In industries such as health-care, insurance and financial organizations, which deal with sensitive data, the question of how to ensure data security and privacy in cloud environments is crucial [18, 27] and even of legal concerns. For example, in the health-care industry the privacy and security of protected health information (PHI) need to be guaranteed according to HIPAA (Health Insurance Portability and Accountability Act)[1] requirements.

To take advantage of public clouds, data owners must upload their data to commercial cloud providers which are usually outside of their trusted domain. Therefore, they need a way to protect the confidentiality of their sensitive data from cloud providers.

^{*} This work has been supported in part by DHS CCICADA and NSF grants CNS-0831186, CNS-0953638, CNS-0831268, CNS-0915394, CNS-0931992, and CNS-0952128.

Moreover, in many cases, data owners also play the role of content provider for other parties. Following the naming convention used in [28, 30], we refer to the parties that consume data owner’s data as *data consumers* or *end-users*. For example, a healthcare provider (data owner) may need to let a medical doctor (data consumer) access medical record of his patient. Even a data consumer may recursively play the role of data owner on its own. A medical doctor may want to share part of his patient’s medical record with his secretary or nurse. Therefore, there is a need for a decentralized, scalable and flexible way to control access to cloud data without fully relying on the cloud providers.

In this paper we design and implement a scalable, user-centric, and privacy-preserving access control framework for untrusted cloud storage. Our framework protects the confidentiality and integrity of stored data as well as the privacy of end-users. It is also implementable on top of existing cloud services and APIs. (Design goals in more details are presented in Section 3.1).

Traditional access control techniques are based on the assumption that the server is in the trusted domain of the data owner and therefore an omniscient reference monitor can be used to enforce access policies against authenticated users. As we indicated earlier, in cloud-based services this assumption usually does not hold and therefore these solutions are not applicable. Cryptographic access control techniques designed for shared/untrusted file systems are potential candidates for clouds. In these approaches, the data stored on untrusted storage is encrypted and the corresponding decryption keys are disclosed only to the authorized users. Therefore, the confidentiality of data is protected against untrusted storage as well as unauthorized users.

However, the existing solutions [22, 23, 24] have scalability limitations that hinder their adoption in the cloud-storage settings. For example, until recently finding a cryptographic approach that simultaneously supports fine-granularity, scalability, and data confidentiality was an open problem. In [30], Shucheng Yu et al addressed this open problem by introducing a novel protocol which closes this gap. Another scalability issue, which we address in this paper, is related to access revocation. To eliminate re-encryptions required as part of access revocation, a technique called *lazy revocation* is widely adopted by existing cryptographic filesystems [11, 25, 26]. Lazy re-encryption delays required re-encryptions until the next write access². In practice, lazy revocation eliminates extra re-encryptions as write access requires the client to re-encrypt the data anyway. Therefore, lazy revocation significantly improves the performance at the cost of slightly lowered security. To support lazy revocation, cryptographic access control protocols need to use a *key-updating scheme* which provides *key regression*. Key regression enables a user holding a new key to derive an older key.

Despite the recent developments on untrusted cloud storage, current key-updating schemes are still inadequate in terms of usability and efficiency. Specifically, existing key-updating schemes [11], especially for access hierarchies, are not scalable as they require complex data structures such as cryptographic trees [22] or linked lists [24] (section 2). These cryptographic data structures need to be updated after each revocation. Since most of the existing cloud storage services have very simple APIs which

² Lazy re-encryption, adopted by [30], delays re-encryptions till next (read or write) access. Since in regular workloads read accesses are significantly more than write accesses, the performance gain by lazy revocation is drastically more than that of lazy re-encryption.

allow only storing and updating attribute-value pairs, implementation of existing key-updating schemes on top of existing commercial clouds is inefficient and unscalable; thus, they are not suitable for our framework.

We introduce a new key-updating scheme called *AB-HKU* which is scalable and also supports access hierarchies without requiring complex data structures. Our *AB-HKU* scheme enables us to support lazy revocation without requiring any change in the existing cloud APIs. We also introduce a new signature scheme for Key-Policy Attribute-Based Encryption [21] called *AB-SIGN*. We then apply these new cryptographic schemes to achieve scalable and anonymous information sharing in existing commercial cloud storage services. We provide an implementation of the proposed protocols and perform extensive experimental evaluation on cloud storage environments. Our technical contributions are summarized as follows:

- We introduce a new scalable and secure key-updating scheme for access hierarchies.
- We design and implement a scalable and privacy-preserving access control framework for existing untrusted cloud services. Our framework supports lazy revocation and access hierarchies.
- We present a signature scheme for Key-Policy Attribute-Based Encryption [21]. Using our signature scheme, users can prove that they own a key that its policy satisfies with a set of attributes, without revealing their identity or credentials.
- We provide the first open source implementation of cryptographic libraries for Hierarchical Identity-Based Encryption [5] and Key-Policy Attribute-Based Encryption [7] schemes. They are useful beyond the specific cloud storage problem studied.

Our paper is organized as follows. In Section 2 we introduce our new key-updating scheme and prove its security. In Section 3 we explain our access control protocol and discuss its features and security guarantees. Then, in Section 4 we discuss the implementation details of our cryptographic libraries and access control framework and evaluate its performance. In Section 5 we discuss the related work. Conclusions and future work are given in Section 6.

2 Key Updating Schemes For Access Hierarchies

In this section we present an efficient secure key-updating scheme that supports hierarchies. First, we provide a formal definition for secure key-updating schemes for hierarchical access. Then, we give a concrete construction of a key-updating scheme based on the use of attribute-based encryption scheme. Our solution supports both key revocation and hierarchical delegation of secret access keys. Our secure cloud storage framework for easy sharing and revocation, described in Section 3, is built based on those two key properties.

2.1 Background

Lazy revocation, first introduced in Cephues [19], is a technique which reduces the overhead of revocation at the price of slightly lowered security [22]. When a user's read

access right on a file is revoked, lazy revocation allows to postpone re-encryption of that file until the next change. Lazy revocation has been adopted by all majors cryptographic file systems [11, 25, 26]. However, it also causes fragmentation of encryption keys in access hierarchies. Therefore, a user receiving the most recent key of an access hierarchy should be able to compute the older keys in order to decrypt files that are not yet re-encrypted by the most recent key, a capability that is referred to as *key regression* [20]. *Key-updating schemes* [11] are cryptographic schemes which support *key regression*.

Another key management issue that we need to address is related to access hierarchies. A user owning access key of a specific hierarchy class should be able to decrypt all objects belonging to that hierarchy as well as all lower hierarchies. Key management schemes for hierarchies generate keys that satisfy this requirement. Key-updating schemes enable users to move backward in time dimension and decrypt data objects encrypted by older keys, whereas key management schemes for hierarchies let users traverse space forward and decrypt data objects encrypted by keys which correspond to lower hierarchies. Access control protocols that are coupled with folder structure of file system ([22]) and need support for lazy revocation, require schemes that let the users simultaneously traverse time backward and space forward. For example, a user holding the most recent version of an access key for folder $/a$ should be able to decrypt a file located at $/a/b/c$ which is encrypted by an older key.

In [11], Backes et al formalize key-updating schemes. They also analyze and evaluate existing protocols that support key regression, but none of these protocols support hierarchies. In [14, 15], Blanton et al formalize key management schemes for hierarchies, study existing protocols and introduce an efficient protocol for managing keys in hierarchies. But all of these schemes and protocols are static with respect to time, as they do not support key-updating/regression. *Therefore, none of these schemes are capable of handling key regression and hierarchies simultaneously.*

To our knowledge, the only work on key regression (lazy revocation) in hierarchies is [22], in which Grolimund et al introduced the concept of Cryptree, a tree constructed by symmetric and asymmetric cryptographic links. In Cryptree, a user holding a clearance key pointing to a folder can traverse a sequence of cryptographic links to derive access keys to all of its sub-folders and files. Moreover, the structure of the Cryptree lets the protocol delay re-encryption of data until the next update; thus supports lazy revocation. However, for the reasons that we explain in Section 5, the complexity of required data structure and its high performance cost for large volume of data makes its implementation on top of existing cloud services unscalable and inefficient.

2.2 Model and Definitions in HKU Scheme

In this section we present a formal definition for Hierarchical Key Updating (HKU) Schemes and its security. Let $T = (V, E, O)$ be a tree that represent a hierarchical access structure. More general access class hierarchies in which partially ordered access classes are represented by a DAG are studied in [15]. In our work, we are only interested in a special case where DAG is a tree. Each vertex v_i in $V = \{v_0, v_1, \dots, v_n\}$ corresponds to an access class. v_0 is the root and an edge $e = (v_i, v_j) \in E$ implies that v_i class is the parent of class v_j .

For example, *top secret*, *secret*, *confidential*, and *unclassified* form a hierarchy of access classes, where the root *top secret* access class is the parent of the *secret* access class. In a more complex access tree, a parent access class may have two or more child access classes. For example, a root *Enterprise* access class may have *Marketing*, *Manufacturing*, and *R & D* as its child access classes. We sometimes refer to access class as class, and use terms *node*, *vertex* and *access class* interchangeably.

O is a set of sensitive data objects, each object o is associated with exactly one access class $\mathcal{V}(o)$. In this model, any subject that can assume access rights at class v_i is also permitted to access any object assigned to a vertex which is the descendant of v_i .

The following definitions introduce the concept of time into our model.

Definition 1. *The local time at vertex v_i is an integer t_i that increases (elapses) every time access rights of a subject to that class is revoked.*

Definition 2. *The global time associated with node v_i is a vector $\tau_i = (t_0, \dots, t_j, \dots, t_i)$ where t_j is the local time of j^{th} vertex on the path from root to vertex v_i on the access tree T .*

Two instances of global time are comparable only if the vertices that they belong to are identical or one of them is the ancestor of the other one; We say $\tau_i < \tau_j$ iff τ_i and τ_j are comparable and all common components of τ_i are less than the corresponding components in τ_j . Similarly, we define comparative operators $=$, $>$, \leq , and \geq .

Definition 3. *A Hierarchical Key-Updating (HKU) Scheme consists of a root user and end users. An end user may be a reader, a writer, or both. There are five polynomial time algorithms $HKU = (\text{Init}, \text{Derive}, \text{Encrypt}, \text{Decrypt}, \text{Update})$ defined as follows.*

- $\text{Init}(1^k, T)$ is a randomized process run by the root user which takes as inputs a security parameter k and an access hierarchy tree T and then generates and publishes a set of public parameters Pub and outputs the root key $K_{v_0, \perp}$. It also initializes the state parameters including the value of local time at each vertex.
- $\text{Derive}(T, K_{(v_i, \tau_i)}, v_j)$ is a randomized process run by the root user, reader or writer which using the private key $K_{(v_i, \tau_i)}$ of v_i at time τ_i derives a private key of target class v_j at its current global time τ_j according to T . Derive computes the requested key only if v_i is an ancestor of v_j and $\tau_j = \tau_i$; otherwise, it outputs null (\perp).
- $\text{Revoke}(T, v_i)$ run by the root user, reader or writer, increments the local time t_i of v_i by one, updates other state variables, and returns the updated tree T' .
- $\text{Encrypt}(T, o_k)$ is a randomized algorithm called by writer that encrypts the data object o_k and returns the encrypted object C .
- $\text{Decrypt}(K_{(v_i, \tau_i)}, C)$ is a deterministic process run by reader which takes a key and an encrypted object as inputs and returns the corresponding object in plaintext. This function can decrypt C only if it belongs to the same or a descendant of the access class that the key belongs to and the time that o_k is encrypted at is less than or equal to τ_i ; otherwise, it outputs null (\perp).

Definition 3 is a generalization of the definition of key-updating schemes in [11] and the definition of key allocation schemes for hierarchies in [15]. If we assign to T

a tree of depth 1 where its leaves are a set of groups (i.e, remove heirarchies), our definition reduces to a key-updating scheme defined in [11] and if we remove the update process and the time dimension, our scheme reduces to key allocation scheme for hierarchies defined in [15]. Intuitively, a hierarchical key-updating scheme is secure if all polynomial time adversaries have at most a negligible advantage to break the ciphertext encrypted with the current-time key of a target class, assuming that the adversaries do not belong to higher (ancestor) target classes in the hierarchy, or possess keys for earlier time periods. The formal definition of the security model of hierarchical key-updating schemes is provided in the technical report [31]. In this model the adversary chooses her target at the beginning of the game and then adaptively queries the scheme.

2.3 AB-HKU Scheme

In this section, we present a concrete construction for *HKU* scheme called *AB-HKU*. This scheme is based on the use of bilinear map and the difficulty of the Bilinear Diffie-Hellman problem. Our solution is realized on top of the Key-Policy Attribute-Based Encryption scheme (KP-ABE) [21] and invokes KP-ABE operations including *SETUP_ABE*, *KEYGEN_ABE* for private key generation, *ENCRYPT_ABE* for data encryption, and *DECRYPT_ABE* for decryption.

– *Init*($1^k, T$)

1. The root user runs the *SETUP_ABE* process with 1^k as security parameter to generate ABE public parameters and the master key MK. Publishes the ABE public parameters as Pub_{abe} .
2. Calls *KEYGEN_ABE* procedure using MK as the secret key and “ $L_0 = v_0$ ” as its policy. Outputs the result as the root key ($K(v_0, \perp) = \text{KEYGEN_ABE}(\text{MK}, L_0 = v_0)$).
3. To each vertex in T adds a local time variable t_i initialized to zero.

– *Derive*($T, K(v_i; \tau_i), v_j$) is run by a user (root user, reader, or writer) with secret key $K(v_i; \tau_i)$ at time τ_i to obtain the private key for node v_j .

If class v_j is not a descendant of class v_i , or the time τ_i is not equal to current time τ_j associated with v_j , then return null. Otherwise, denote (u_1, u_2, \dots, u_n) as the list of vertices in the path from v_i to v_j ; denote $(t_{u_1}, t_{u_2}, \dots, t_{u_n}, t_{v_j})$ on T as the list of current local time values of intermediate vertices (including v_j); and let d represent the depth of v_i .

The user performs the following operations.

1. Construct the access tree \mathcal{T}' which corresponds to the following Boolean expression: ($L_d.v$ attribute represents vertex in d -th level, $L_d.t$ represents its current local time and \wedge is conjunction operator.)

$$\begin{aligned}
 & (L_{(d+1)}.v = u_1 \wedge \dots \wedge L_{(d+n)}.v = u_n \\
 & \quad \wedge L_{(d+n+1)}.v = v_j) \wedge \\
 & (L_{(d+1)}.t \leq \tau_{u_1} \wedge \dots \wedge L_{(d+n)}.t \leq \tau_{u_n} \\
 & \quad \wedge L_{(v_j)}.t \leq \tau_{v_j})
 \end{aligned} \tag{1}$$

This Boolean expression restricts access to objects that belong to node v_j or its descendants and are created at current time or before.

2. Denote the access tree of $K_{(v_i, \tau_i)}$ by \mathcal{T} . Using the procedure for delegation of private key in [21], add the access tree \mathcal{T}' to the root of $K_{(v_i, \tau_i)}$, increase its threshold by one, update and calculate the private parameters associated to the root according the protocol. In implementation section we provide more details on this procedure.
 3. Output the resulting access tree and parameters as a private key $K_{(v_j, \tau_j)}$ for v_j .
- *Encrypt*(T, o_k): Denote v_i as the access class that object o_k belongs to. ($v_i = \mathcal{V}(o_k)$). Denote $(v_0, u_1, u_2, \dots, u_n, v_i)$ as v_i 's path and $\tau_i = (t_{v_0}, t_{u_1}, t_{u_2}, \dots, t_{u_n}, t_{v_i})$ as its current time according to T . A writer encrypts o_k as follows.
1. Set the attribute set γ as follows. The attribute set is used as the public key for encryption.

$$\gamma = \{L_0.v = v_0, \dots, L_n.v = v_n, L_{n+1}.v = v_i, \\ L_0.t = t_{v_0}, \dots, L_n.t = t_{u_n}, L_{n+1}.t = t_{v_i}\} \quad (2)$$

2. Use ABE encryption procedure to encrypt o_k with attribute set γ and return the resulting encrypted object. ($C = \text{ENCRYPT_ABE}(\text{Pub}_{\text{abe}}, \gamma, o_k)$).
- *Decrypt*($K_{(v_i, \tau_i)}, C$). The reader decrypts as follows.
1. If the encrypted object C does not belong to the same access class v_i as the key $K_{(v_i, \tau_i)}$ or one of its descendants, or the time when C is encrypted is later than the time τ_i when the key is generated, then return null (\perp).
 2. Otherwise, run ABE decryption procedure and return its result as output ($o_k = \text{DECRYPT_ABE}(K_{(v_i, \tau_i)}, C)$).
- *Revoke* (T, v_i) is run by a user to increment the local time of v_i by one and then returns the updated tree T' .)

The correctness of our HKU scheme follows the correctness of the key policy ABE scheme [21] and is omitted here.

Theorem 1. *Assuming the hardness of the Decisional BDH, AB-HKU is a secure hierarchical key-updating scheme.*

Proof. Proof is presented in the technical report [31].

3 K2C Protocol

We describe the application of our hierarchical key-updating scheme in realizing a secure and scalable cloud access control protocol that supports easy sharing and revocation on hierarchically organized resources. We also analyze the security of our protocol.

3.1 Design Goals

Below we list the design goals of our *K2C* access control protocol:

- *Security*: Our protocol must protect the confidentiality and integrity of stored data against cloud providers and unauthorized end-users. Meaning that the stored data should be readable for authorized users only and any unauthorized change to the data should be prevented or detectable.
- *Privacy-preserving*: Access rights of a specific end-user as well as his usage trends should not be visible to other users or cloud service providers.
- *Efficiency and Scalability*: To avoid unjustified cost of re-encryption, the protocol should support lazy revocation. Also, the complexity of operations should be independent of number of data objects and users in the system. This ensures that the protocol will not affect the scalability of existing cloud services.
- *Flexibility*: The protocol should allow data owners and end-users to organize and manage their data in hierarchies similar to conventional file systems. Directories also represent access class hierarchies, users who have access to a directory/folder also assume the same access to all files and directories below that directory. Also, they should be able to grant/revoke part of their access rights to/from other users in a decentralized and scalable manner.
- *Simplicity and Extensibility*: Last but not the least, the protocol should be simple enough to be efficiently implementable on top of existing commercial cloud APIs.

We assume end-users have secure communication channels, limited computation power and storage required for authenticating each other and performing client-side key distribution in a synchronous or asynchronous manner.

Security Model. We assume that the root user, representing the data owner, is trusted. The cloud providers are honest-but-curious (aka semi-honest), who follow the protocol and faithfully execute the operations, but may actively attempt to gain additional knowledge, such as the sensitive data stored in the cloud. An adversary may attempt to perform unauthorized read or write access against the stored data, or attempt to learn the identities of readers or writers. For example, end-users may try to perform unauthorized read or write operations on stored data objects. To perform their attacks, unauthorized users may use their existing access keys for other objects and categories or cooperate with other unauthorized users and cloud providers to derive/guess credentials required to perform unauthorized access. Similarly, cloud providers may try to read or modify stored data or learn about the identities of the end users. Cloud providers may collude with each other or some unauthorized end-users to break the security of *K2C*. We assume communication channels between participants are secure (e.g., SSL).

3.2 A signature scheme for KP-ABE

K2C requires a signature scheme to 1) enable the readers to verify the integrity of data and ensure that it is produced by an authorized writer, 2) enable the cloud service providers to validate incoming requests and block unauthorized accesses. However, the original paper which introduces KP-ABE [21] does not present any signature scheme. In this section we introduce an attribute-based signature scheme called *AB-SIGN* which

enables the verifier to ensure that a signature is produced by a user whose access policy is satisfiable by a set of attributes without learning the signer’s identity.

Definition 4. *AB-SIGN is a signature scheme for key-policy attribute-based encryption that its signing and verification methods are defined as follows. Let’s say that the signer has a key K for policy P , and wants to sign message \mathcal{M} . The verifier needs to verify that the signature is generated by a signer whose key policy satisfies attribute set A .*

Signing: *From K derive a key (K') which corresponds to a policy which is the concatenation of P and $@S = \mathcal{M}$. Send the derived key to the verifier as the signature.*

Verification: *Generate a random token and encrypt it using the attribute set $A \cup \{@S = \mathcal{M}\}$ and then decrypt the result using a key which is equal to the signature. If the result is equal to the original token the signature is valid (i.e. the attribute set A satisfies the signer’s key policy.)*

To prevent an attacker from using the signature method to derive a valid access key, we need to reserve the attribute ‘@S’ for signature. The security of *AB-SIGN* scheme in terms of unforgeability follows immediately from the security of *KP-ABE* scheme.

3.3 Protocol Description

In this section we provide the details of *K2C* protocol. The protocol runs between the root user, end-user (reader or writer), and the cloud providers. The root user may be a system administrator in the data owner’s organization, who can specify the access privileges of end-users. The end-users may further delegate their access privileges to other individuals for easy sharing. We achieve the revocation of privilege by encoding the validity period in the private keys of users and advancing time with respect to the target hierarchy or data object. Another advantage of our *K2C* framework for use in cloud storage is the support of anonymous access.

As illustrated in Figure 1, *K2C* requires three repositories: *Meta-data Directory*, *Data Store* and *Key-store*.

- **Meta-data Directory:** All meta-data associated with hierarchies and data objects are maintained in this repository. *K2C* requires two properties for each object: *Read Access Revision (RAR)* and *Write Access Revision (WAR)*. These two properties play the role of local time in *AB-HKU* for read and write access, respectively. In order to compute Read/Write Access Revision Vectors (which correspond to global time instances in *K2C*), the cloud provider that hosts *Meta-data Directory* needs to provide an API for querying *RAR* and *WAR* values of multiple directories in a single request. All existing cloud-based databases such as Amazon SimpleDB [3], Microsoft Azure SQL [9], and Google’s AppEngine [4] database (Bigtable [17])) satisfy this requirement and therefore qualify to host a *K2C* *Meta-data Directory*. For our experiments we use Amazon SimpleDB [3].
- **Data Store:** This repository contains the actual content of each data object. Any cloud key-value based storage system such as Amazon S3 [2] can be used as *K2C* *Data Store*. In *K2C*, *keys* are hierarchical path name of data objects and *values* are the actual content of corresponding data objects. Cloud key-value storage

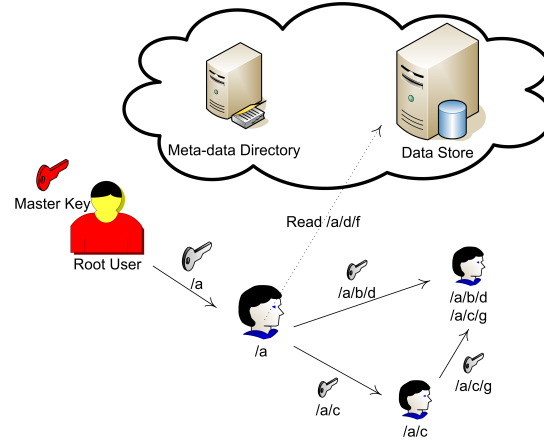


Fig. 1. Illustration of all major participants of *K2C*. Following *K2C* protocol, end-users can enforce access control on their own data without fully trusting or relying on the cloud providers. In *K2C*, keys are distributed and managed in a distributed fashion. Solid arrows represent access delegation.

providers are tuned for high throughput and low storage cost; these features make them a good candidate for *K2C* Data Store ³.

- **Key-store:** All read/write access keys of end-users are kept in their secure local repository called Key-store.

Initial Setup: To setup *K2C*, the root user needs to follow the steps listed below:

1. Sign up for cloud services required for hosting Meta-data Directory and Data Store.
2. Run *Init* procedure according to *AB-HKU* scheme to generate public parameters and the master key.
3. Save the master key and public parameters in the root's Key-store.
4. Share the public parameters with the cloud service providers that support *K2C* request authorization.
5. Create an entry in Meta-data Directory that corresponds to the root directory. The WAR and RAR numbers of the root directory entry are initialized to zero.

Basic Operations: There are four basic operations in our protocol: write, read, delegate, and revoke. Each basic operation leads to calls to Meta-data directory and/or Data Store. We present the high-level steps involved in these operations below. Other operations such as create/remove/rename for directories and data objects can be defined similarly. *K2C* requires that each request is signed by user's access key for the target object using our *AB-SIGN* operation. This requirement enables cloud providers which support *K2C* to block unauthorized request. We refer to this property as *K2C* request authorization.

³ Note that using key-value storage for Meta-data Directory is not efficient as computing WAR/RAR vector leads to multiple calls to the cloud storage system.

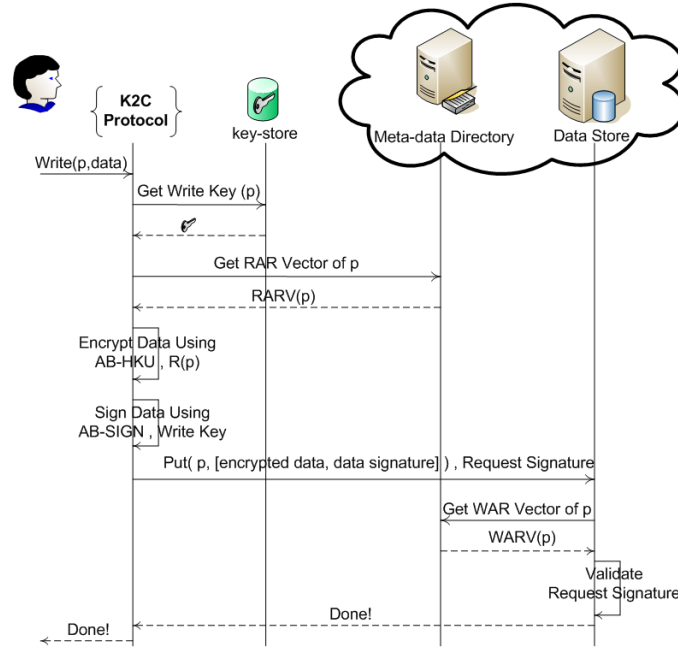


Fig. 2. Write operation.

- **Write:** To write into a specific data object, the end-user needs to perform the following steps (See also Figure 2).
 1. Retrieve the required write access key from the local Key-store.
 2. Query Meta-data directory to get read access revision (RAR) vector of the target object.
 3. Using *AB-HKU* scheme, encrypt the data by the retrieved RAR vector and its path.
 4. Using *AB-SIGN* scheme, sign the data by his write access key.
 5. Construct a key-value pair where the key is equal to the path of data object and the value is the encrypted data and corresponding signature. Store the pair in Data Store.
- **Read:** To read a specific data object stored using *K2C* protocol, the end-user needs to do the following (See also Figure 3). To ensure the data is produced by an authorized writer, the reader needs to validate the corresponding signature using *AB-SIGN* signature scheme. Then the reader can decrypt the data using its read access key and *AB-HKU* scheme.
 1. Retrieve the required read access key from the local Key-store.
 2. Using *AB-HKU* scheme and the read access key, decrypt the encrypted data.
 3. Using *AB-SIGN* signature scheme, validate the signature to ensure that data is produced by a user who has the proper write access.
 4. Return the decrypted data.

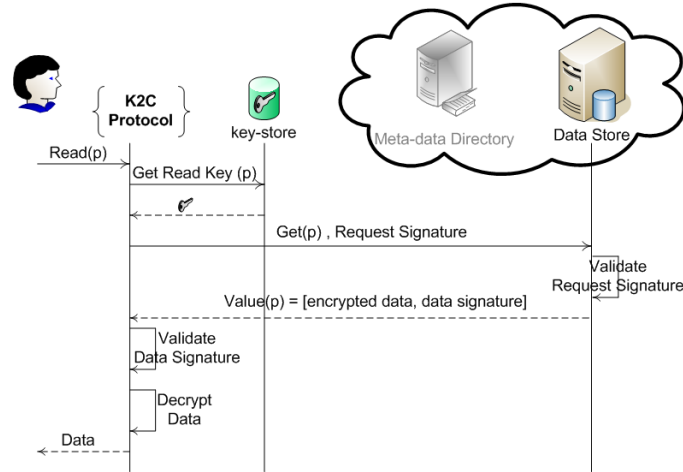


Fig. 3. Read Operation.

- **Delegation:** Delegation operation can be run by a user to authorize another user a subset of his access privileges. It requires three input parameters: the identity of the delegatee, the resource path, and access type (read/write). The steps required for this operation are listed below:
 1. From the local Key-store, get the access key that matches the target resource path and access type.
 2. Query Meta-date Directory to get the read/write access revision (RAR/WAR) vector of target resource.
 3. Run *Derive* operation, as defined in *AB-HKU* scheme, to generate the required access key.
 4. Send the generated access key to the delegatee through a secure communication channel.
- **Revocation:** To revoke a user's access on a specific directory or data object, the authorized user needs to make a signed request to the Meta-data Directory to increase the corresponding access revision number. To ensure the integrity of access revision numbers, these entries should be signed by the requester.

3.4 Security Analysis

In this section we state the security guarantees provided by *K2C* protocol. More detailed proofs and analysis can be found in our full version [31].

Confidentiality: Our solution ensures that only the users who have the most recent version of the access key of the data object or one of its ancestor directories can decrypt it. The confidentiality of stored data is protected under our protocol because writers always encrypt the data objects by their path and most recent read access revision (RAR) vector according to *AB-HKU* scheme. The cloud provider or other unauthorized users

cannot gain any information that helps them to guess the access key of unauthorized data objects.

Collusion-resistance: of KP-ABE guarantees that unauthorized users and malicious cloud service providers cannot collude to guess access key to an unauthorized data object.

Integrity: The integrity of stored data is preserved. This guarantee is realized by requiring writers to sign the data by their write access key using *AB-SIGN* scheme. We require readers to validate writer’s signature to ensure that it is produced by an authorized writer (i.e. a user with write access to that data object or on of its parent directories). Because meta-data entries stored in the Meta-data Directory are also required to be signed by the end-users, any unauthorized change in Meta-data Directory is detectable by the reader.

Anonymity: The end users are anonymous to each other and to the cloud providers. The signatures used in the our authorization do not contain any identify information. During the course of protocol, the end-users do not reveal any information about their credentials. *AB-SIGN* signatures bound to the data objects and requests, include only attributes related to the location and global time of those objects.

4 Implementation and Evaluation

We give our implementation of *K2C* framework and the required cryptographic libraries. We present our experimental results on accessing Amazon cloud storage [2, 3] using *K2C* framework.

4.1 Key-Policy Attribute-Based Crypto Library

To support lazy-revocation and hierarchies, *K2C* uses our *AB-HKU* scheme that is based on Key-Policy Attribute-Based encryption scheme [21]. But, we were not able to find any implementation of KP-ABE ⁴. Therefore, we develop a general KP-ABE cryptographic library and release it as an independent open source project [7]. In this section, we provide a short overview of this library.

Our library implements the KP-ABE scheme. We also fix a non-trivial limitation existed in the construction of [21]. KP-ABE is a *large universe construction*, meaning that it does not require the attributes to be fixed during the initialization process. However, the maximum number of attributes should be known in advance – a limitation which is not desirable in many practical cases. To overcome this limitation, we adopt the random oracle model [12] and replace function $T(X)$ (used in the *Setup* phase) by a secure hash function. This modification also improves the efficiency of the library. Therefore, our library does not put any limitation on the number of attributes that can be used in the system. We support numerical attributes and comparisons [13].

⁴ The open source implementation of Ciphertext-Policy Attribute-Based Encryption (CP-ABE) [13] which was presented in the original paper [13] is available at [8]. However, CP-ABE is not applicable in our protocol.

In our library, policies are defined recursively and represented using an S-Expression (LISP-like expression) as follows:

$$\left\{ \begin{array}{ll} a = v & a \text{ is a symbolic attribute} \\ (c a v) & a \text{ is a numerical attribute \&} \\ & c \text{ is a comparative operator} \\ ([t \mid \text{and} \mid \text{or}] p_1 p_2 \dots p_n) & \text{Composite policy} \end{array} \right.$$

The first and second types correspond to a simple policy for symbolic and numerical attributes, respectively. The third type represents policies which are composed of a set of policies (i. e. p_1, p_2, \dots, p_n) preceded by a threshold. Composed policies get satisfied only if the number of satisfied policies in that list is more than or equal to the specified threshold. The threshold can be one of the following three items: an integer threshold $t \in [1, n]$, and, or. For example, `(and role=manager (> age 18))` is a composite policy which gets satisfied only when the value of attribute `role` is equal to `manager` and the value of the numerical attribute `age` is greater than 18. More implementation details are presented in the technical report [31].

4.2 K2C Framework and Performance Evaluation

In this section we present the high-level architecture of *K2C* framework. We also provide some experimental results that show its performance in an existing commercial cloud storage.

Simplicity and extendability are two major design goals of *K2C* framework. *K2C* framework is independent of any specific cloud provider. It has two simple interfaces which abstract away the details of the cloud providers: `IDataStore` and `IMetadataDirectory`. A new cloud service provider can be supported easily by implementing these interfaces. Out of the box, *K2C* framework comes with a data store driver for Amazon S3 and a meta-data directory driver which uses Amazon SimpleDB. To make it easier for the developers to learn and use our framework, we expose its services through a set of APIs which are very similar to the Java APIs for accessing the file system. The source code is published at [6]. KeyStore).

To evaluate the performance of *K2C* framework/protocol, we used the default drivers (Amazon S3 [2] data store driver and Amazon SimpleDB [3] meta-data directory driver) and ran our experiments on a machine with the following configuration: Intel Core 2 CPU, 2.53GHz, 2.90 GB RAM, Microsoft Windows XP 2002 SP2.

Figure 4 shows the time required for users in different access levels to perform read and write operations against the data objects of size 1KB belonging to directories in different hierarchical levels. Reported costs for each operation include computational time required for cryptographic operations (Symmetric and KP-ABE) as well as round trip time for HTTP calls to the cloud servers. As these diagrams show, users with higher level access (e.g. the root user) can perform read and write operations more efficiently, a property which is normally desirable⁵. Access time increases linearly as the access level of users decreases. Also this figure shows that access time for both read and write

⁵ In Cypree, high-level users have higher access time as they have to traverse longer cryptographic lists to find the access key (See Section 5).

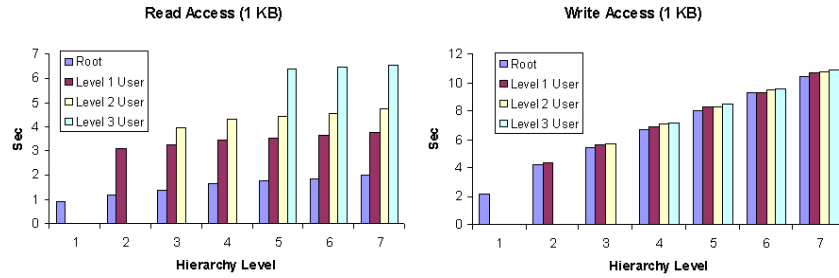


Fig. 4. Cost of read/write operation on objects which belong to different hierarchy levels performed by users with different access levels. The size of data objects are 1 KB.

operations increases linearly as the hierarchy level of object increases. The reason behind this linear increase is that encryption/deception time in KP-ABE is a function of number of attributes and the number of attributes associated with data objects linearly increases as their hierarchy level increases. Another observation is that write operations are more expensive than reads. This overload for write operations is partly due to an extra http call to Meta-data Directory required for retrieving the latest RAR vector.

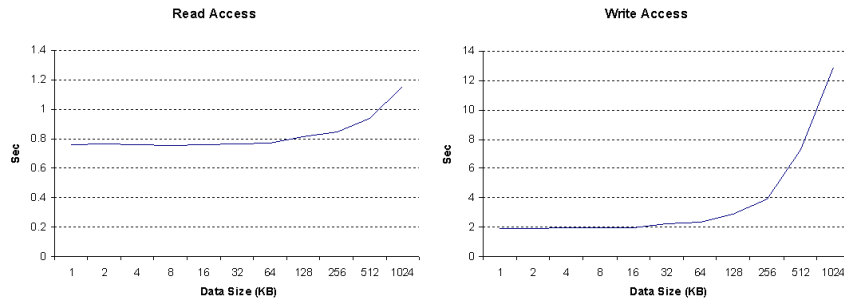


Fig. 5. Read/write access time for the root user as the size of data objects increase.

Figure 5 shows the cost of read and write access as the size of data object increases. Since in our experiments the actual data is encrypted using the symmetric-key encryption scheme AES (Advanced Encryption Standard) and only the key is encrypted using KP-ABE, these statistics reflect the time required by AES to encrypt the data as the data size increases.

5 Related Work

There are two general key management approaches which are used in the existing cryptographic file systems: 1) classic access control list (e.g., [22, 23]) requires maintaining a key list along with each file. This approach supports fine-granularity but is not scalable. 2) grouping files and assigning the same access key to each group (e.g., [24]). This approach is more scalable but provides coarse-grained access control. This trade-off makes these solutions unsuitable for clouds where we need a fine-grained and scalable access control mechanism. In [30], Shucheng Yu et al introduced a novel approach which addresses this trade-off by proposing a fine-grained and scalable access control protocol. Their solution uses *lazy re-encryption* to statistically reduce the number of re-encryptions required after access revocation. They use *proxy re-encryption* (PRE) [16] to off-load the task of re-encryption to cloud servers. In our solution, we adopt *lazy revocation* to eliminate these re-encryptions. In [29], Xiong et al introduce a protocol for securing end-to-end content distribution when delivery services are involved.

Lazy revocation was first introduced in Cephues [19] to eliminate re-encryption required for each revocation at the cost of slightly lowered security. Lazy revocation, which is widely being used in recent cryptographic file systems [24, 25, 26], requires a key-updating scheme to support key regression. Key-updating schemes are studied and formalized in [11]. In [22], Grolimund et al introduced Cryptree which can support access hierarchies and lazy revocation simultaneously. However, due to the explicit and physical dependency of these links, file system operations – especially revocations – require updating large number of these cryptographic links. For example, the revocation of write privilege requires updating $O(n)$ keys, where n is the number of data objects contained in that folder and its sub-folders. Therefore, revocation of write access for a folder containing many files is relatively slow as all the links that connect to the contained sub-folders and files need to be updated.

Moreover, in Cryptree, since key derivation requires traversing cryptographic links, key derivation time is a function of distance of data objects to the folder that the user has access to. Therefore, users with access to high-level folders (e.g. root user) have slower read access. For a specific user read access time depends on the location of the data object, but intuitively we expect the read access time to be independent of the location of the data object. Another limitation of this approach is that Cryptree does not support the delegation of administrative rights and assumes that granting and revoking access rights are done by a single administrator, an assumption which is usually unrealistic in the context of Cloud Storage, as we expect non-centralized administration of data. In this paper we introduced a scalable key updating scheme for hierarchies which addresses these shortcomings and enables us to build a cryptographic access control supporting lazy revocation.

6 Conclusion and Future Work

We presented a novel key-updating scheme that can be used to enhance the scalability and performance of cryptographic cloud storages by adopting lazy revocation. We also designed a new digital signature scheme that enables cloud providers to ensure that

requests are submitted by authorized end-users, without learning their identities. Using our key-updating and signature scheme, we developed, implemented, and evaluated a scalable cryptographic access control protocol for hierarchically organized data. We plan to improve the efficiency of *K2C*, and to enhance our access control protocol by using proxy re-encryption [10] to off-load key distribution task to the cloud [30]. We are also investigating application of our key-updating scheme in existing cryptographic file systems and webtops [32].

7 Acknowledgments

The first author would like to thank the help of professors at Bahai Institute for Higher Education (BIHE) and Gurpreet Singh.

References

1. 104th United States Congress. Health Insurance Portability and Accountability Act of 1996 (HIPPA), <http://aspe.hhs.gov/admsimp/pl1104191.htm>.
2. Amazon S3 . <http://aws.amazon.com/s3/>.
3. Amazon SimpleDB . <http://aws.amazon.com/simpledb/>.
4. Google App Engine. <http://appengine.google.com>.
5. HIBE Crypto Library. <https://sourceforge.net/projects/hibe>.
6. K2C Framework. <https://sourceforge.net/projects/key2cloud/>.
7. KP-ABE Crypto Library. <https://sourceforge.net/projects/kpabe>.
8. Open Source Implementation of CP-ABE. <http://acsc.cs.utexas.edu/cpabe/>.
9. SQL Data Services/Azure Services Platform. <http://www.microsoft.com/azure/data.mspx>.
10. Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. In *NDSS*, pages 29–43, 2005.
11. Michael Backes, Christian Cachin, and Alina Oprea. Secure Key-Updating for Lazy Revocation. In *Research Report RZ 3627, IBM Research*, pages 327–346. Springer, 2005.
12. Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security, CCS '93*, pages 62–73. ACM, 1993.
13. John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy, SP '07*, pages 321–334, Washington, DC, USA, 2007. IEEE Computer Society.
14. Marina Blanton. Key Management in Hierarchical Access Control Systems, 2007. PhD Thesis, Purdue University, Aug. 2007.
15. Marina Blanton, Nelly Fazio, and Keith B. Frikken. Dynamic and Efficient Key Management for Access Hierarchies. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2005.
16. Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In *EUROCRYPT*, pages 127–144. Springer-Verlag, 1998.
17. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th symposium on Operating systems design and implementation - volume 7*, pages 205–218, 2006.

18. Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, pages 85–90, New York, NY, USA, 2009. ACM.
19. Kevin Fu. Group sharing and random access in cryptographic storage file systems. Technical report, Masters thesis, MIT, 1999.
20. Kevin Fu, Seny Kamara, and Tadayoshi Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *NDSS*, 2006.
21. Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 89–98, New York, NY, USA, 2006. ACM.
22. Dominik Grolimund, Luzius Meisser, Stefan Schmid, and Roger Wattenhofer. Cryptree: A folder tree structure for cryptographic file systems. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 189–198, Washington, DC, USA, 2006. IEEE Computer Society.
23. Eu jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. Sirius: Securing remote untrusted storage. In *NDSS*, pages 131–145, 2003.
24. Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage, 2003.
25. Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A framework for evaluating storage system security. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.
26. Paul Stanton, William Yurcik, and Larry Brumbaugh. Protecting multimedia data in storage: A survey of techniques emphasizing encryption. In *IS and T/SPIE International Symposium Electronic Imaging / Storage and Retrieval Methods and Applications for Multimedia*, pages 18–29, 2005.
27. Hassan Takabi, James B.D. Joshi, and Gail-Joon Ahn. Security and Privacy Challenges in Cloud Computing Environments. *IEEE Security and Privacy*, 8:24–31, 2010.
28. Qian Wang, Cong Wang, Jin Li, Kui Ren, and Wenjing Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 355–370, Berlin, Heidelberg, 2009. Springer-Verlag.
29. Huijun Xiong, Xinwen Zhang, Wei Zhu, and Danfeng Yao. CloudSeal:End-to-End Content Protection in Cloud-based storage and delivery services. In *Securecomm*, 2011.
30. 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, pages 534–542, Piscataway, NJ, USA, 2010. IEEE Press.
31. Saman Zarandioon, Danfeng Yao, and Vinod Ganapathy. K2C: Cryptographic Cloud Storage With Lazy Revocation and Anonymous Access. Technical report, Rutgers University. DCS-tr-688.
32. Zaranux, <http://zaranux.com/>, Saman Zarandioon.