On Deletion of Outsourced Data in Cloud Computing

Zhen Mo, Qingjun Xiao, Yian Zhou, Shigang Chen
Department of Computer & Information Science & Engineering
University of Florida, Gainesville, FL 32611 Email:{zmo, qxiao, yian, sgchen}@cise.ufl.edu

***Abstract*—Data security is a major concern in cloud computing. After clients outsource their data to the cloud, will they lose control of the data? Prior research has proposed various schemes for clients to confirm the *existence* of their data on the cloud servers, and the goal is to ensure data integrity. This paper investigates a complementary problem: When clients delete data, how can they be sure that the deleted data will never resurface in the future if the clients do not perform the actual data removal themselves? How to confirm the *non-existence* of their data when the data is not in their possession? One obvious solution is to encrypt the outsourced data, but this solution has a significant technical challenge because a huge amount of key materials may have to be maintained if we allow fine-grained deletion. In this paper, we explore the feasibility of relieving clients from such a burden by outsourcing keys (after encryption) to the cloud. We propose a novel multi-layered key structure, called Recursively Encrypted Red-black Key tree (RERK), that ensures no key materials will be leaked, yet the client is able to manipulate keys by performing tree operations in collaboration with the servers. We implement our solution on the Amazon EC2. The experimental results show that our solution can efficiently support the deletion of outsourced data in cloud computing.**

I. Introduction

One major problem of cloud services is how to ensure the security of the data stored on servers. This problem stems from the fact that users lost physical control of their outsourced data, while they cannot fully trust the cloud service providers. For instance, the servers may be compromised, the providers may be forced to hand over data to law enforcement, or their employees who have access to the data may sell them under the table to competitors. To address such concern, much research work has been done to ensure the privacy and integrity of the outsourced data [13], [19], [10], [23], [12], [3]. One of their goals is to verify the *existence* of the data in its entirety on the cloud servers. This paper investigates a complementary problem that is important but has received much less attention: When users delete data in the cloud, how can they be sure that the deleted data will never resurface in the future if the actual data removal is preformed by someone else? How to confirm the *non-existence* of their data when the data is not in their possession?

In order for a client to render its data on a server inaccessible, a straightforward approach is to encrypt the data before outsourcing. The client keeps the encryption key, while the server keeps the encrypted data. To make sure that the data cannot be recovered in the future, the client needs to securely delete the key, while informing the server to delete the encrypted data. Regarding this approach, there is an important question to answer: How many keys to be used? *If we use one key to encrypt all data*, whenever we delete one data item, we have to re-encrypt all other data items with a new key because otherwise they would also become inaccessible after the old key is deleted. *If we assign one key to each file* [22], [18], there will be numerous keys if the number of files is large. Moreover, even if we only want to delete one block in a file, we will have to retrieve the entire encrypted file from the server, decrypt it, delete that block, remove the old key, choose a new key, and re-encrypt the entire file. Now, what about the outsourced is a large database? It is better modeled as a collection of data records, rather than files. We should be able to efficiently delete any record in the database without having to frequently re-encrypt other data that are not directly related to the deletion. To do that, we may assign one key to each record. Clearly, we should not keep that many keys at the client side because it defeats, at least partially, the benefit of outsourcing data to the cloud.

That translates the deletion of cloud data into a problem of key management, and the challenge is how to do so efficiently. The FADE system [22] introduces a trusted third party to help managing the keys. The role of a trusted third party in FADE is fundamentally different from that of a trusted third party in PKI. The former has to store keys. It is involved in all operations of data access, insertion and deletion. Hence, it is potentially a performance bottleneck and a single point of failure. The latter only signs the public keys, and it is not involved in actual data access. Moreover, if we cannot fully trust a cloud system under the concern that its servers may be compromised by external or internal attackers, we will naturally have the same concern on the servers of a trusted third party that holds our keys.

This paper explores the feasibility of permanently deleting data without involving a third party between clients and servers in a cloud system. We not only outsource the data but also outsource key materials that are used to encrypt the data to the cloud system. Yet we prevent any possibility for the cloud service providers or anyone who compromises the cloud servers to circumvent deletion or break data privacy. Our solution is based on a novel multi-layered key structure, called Recursively Encrypted Red-black Key tree (RERK), that ensures no key materials will be leaked. The client only maintains a small amount of metadata, and it is able to manipulate keys by performing tree operations in collaboration with the servers. A client may apply the RERK only to its security-sensitive information, while using conventional storage schemes for other bulk data. Modest addition of overhead by the RERK is justified by the peace of mind that its property of security-assured deletion can bring to the client. We have implemented our key management system in Amazon Elastic Compute Cloud (EC2). The evaluation results show that our system is efficient.

The rest of the paper is organized as follows: Section II discusses the prior art. Section III defines the problem and the adversary model. Section IV introduces a basic two party solution for deletion. Section V presents our new RERK solution. Section VII gives security analysis. Section VIII describes our

978-1-4799-5063-8/14 $31.00 © 2014 IEEE
DOI 10.1109/CLOUD.2014.54

implementation and presents the experimental results. Section IX draws the conclusion.

II. Related Work

Perlman proposes the first approach for assured file deletion in [16]. Together with the followup works [7], [15], [2], [1], [21], they form the so-called Ephemerizer family of solutions. Their goal is to ensure the privacy of past messages transferred between two parties, such as emails or SMS. The main approach is to encrypt each message with a data key, and the data keys whose expiration times are the same will be encrypted by an *ephemeral public key*. These public keys are managed by one or more trusted third parties, named "the ephemerizers." As the ephemeral private keys are only known to the ephemerizers, deleting one ephemeral private key will make the data keys encrypted by the corresponding public key unrecoverable. All Ephemerizer solutions require the trusted third parties to perform the deletion operations; if the third parties are down, certain operations will become unavailable. In addition, risks arise when the third parties are internally or externally compromised.

Instead of relying on centralized third parties to manage the keys, Geambasu $et\,al.$ design a decentralized approach called the Vanish [11] using a distributed hash table (DHT) [20]. The Vanish is vulnerable to Sybil attacks [9]. It has been proved by Wolchok $et\,al.$ [24] that attackers can continuously crawl the DHT and save each stored value before it expires. They can efficiently recover keys for more than 99% of the messages. Zeng $et\,al.$ propose SafeVanish [25] and Castelluccia $et\,al.$ design EphPub [5] to fix the security problems of Vanish.

However, Vanish was originally designed to ensure the privacy of past messages transferred between two parties. Some of its properties make it unsuitable for a cloud system: First, it protects data that only need to be available for hours or days, such as emails, SMSs, trash bin files, etc. Data in a cloud system may stay for months, years, or permanently. Second, Vanish assumes users know approximately the lifetime of their data, but that may not be the case in a cloud system. Third, Vanish is designed for data whose privacy is more important than accessibility. That is, users may not be able to access their data before the specified timeout in Vanish, which will not be generally acceptable to users of a cloud system. Finally, Vanish creates a Vanish Data Object (VDO) for each data item. Because VDOs have to be stored by clients and they cannot be outsourced to cloud servers, it creates a significant storage burden for users.

Most related is Tang's policy-based system named FADE [22], which is designed to perform assured file deletion for a cloud storage system. Their basic approach is to encrypt each file with a data key, which is in turn encrypted by a control key. There is a policy associated with each control key. The policies are managed by one or multiple trusted third parties. When the client deletes a file, it will revoke the corresponding policy and instruct the third party to remove the key. The problem with the FADE's third party has been discussed in the introduction. Arthur Rahumed $et\,al.$ design a system called FadeVersion [18], which supports both version control and assured deletion. But it needs users to manage the keys by themselves through a key escrow system. This may create a heavy burden on users because the volume of the keys can be huge if fine-grained deletion is required.

III. Preliminaries

A. System Model

A cloud system consists of two parties: (1) The *clients* are individual users or companies. They have a large amount of data to be stored, but do not want to maintain their own storage systems. By outsourcing their data to the cloud and deleting the local copies, they are freed from the burden of storage management. (2) The cloud *servers* have a huge amount of storage space and computing power. They offer resources to clients on a pay-as-you-go manner.

After putting data on cloud servers, the clients lost direct control of their data. They may query and retrieve their data, or change the data by sending requests to the servers. Upon receiving the requests, the servers will perform operations for insertion, modification, deletion, etc. Due to possible external/internal compromise, the clients cannot fully trust the servers. Hence, it is important for the cloud-system design to have built-in mechanisms that guard the security of clients' data against any misbehavior of the servers.

B. Problem Statement

We investigate the problem of *assured data deletion*, which guarantees that the data deleted from a cloud system will be permanently inaccessible. Our goal is to protect the forward privacy of deleted data. Suppose all data items are encrypted before sending to the cloud. Assured deletion requires that the encryption key of a data item must be unrecoverable once the data is deleted. This requirement becomes challenging if we want to outsource both data and keys that encrypt the data to the cloud, so as to keep the clients as storage-light as possible. In addition, we require a *two-party design* that does not involve any third party in the operations between clients and servers.

C. Adversary Model and Security Definition

Consider a data item D that is deleted by a client at time T from a cloud server. We adopt the worst-case adversary model that gives attackers the following capabilities: (1) they may have full control of the server at all time and (2) they may compromise the client's host after time T.

The first attacking capability reflects the possibility that the server may be compromised before T. Hence, the attackers have access to everything on the server, and they are able to control the actions of the server in response to the client's requests.

The second attacking capability reflects the possibility that the client's host may be compromised after T. In this case, the attackers have access to everything stored on the client side, including any key materials remained on the client.

We want to make sure that, under the above model, the attackers are unable to figure out the deleted data. However, if the attackers manage to compromise the client's host before T, they will know the data, which has not been deleted yet.

D. Security Definition

A solution for deleting data in a cloud system is secure if for an arbitrary time T all data that have been deleted before time T will be unrecoverable in polynomial time even when the Probabilistic Polynomial Time (PPT) adversary is able to gain full control of servers before T and full control of clients after T, assuming (1) the existence of a collision-resistant hash function such that the probability of finding two hash inputs that produce the same output or finding a hash input to produce a specific output in polynomial time is negligibly small, and (2) the existence of a symmetric encryption scheme that is secure against CPA (chosen plaintext attack) such that the probability of finding the plaintext of a ciphertext in polynomial time without knowing the encryption key is negligibly small, given the knowledge of a polynomial number of plaintext/ciphertext pairs.

IV. Straightforward Two-Party Solutions

We show that simple two-party solutions do not work for assured deletion. We use $\{\mathbf{m}\}_{\mathbf{k}}$ as the notation for encrypting m with key k.

A. A Master Key based Solution

Suppose the client has n data items, denoted as $M = \{m_1, m_2, ..., m_n\}$. The client selects a master K. From the master key, it derives a different key $k_i = PRF(K, i)$ for each data item, where PRF is a pseudo random function with two inputs and i is the index of item m_i, for $1 \leq i \leq n$. The client encrypts each data item with its corresponding key, $c_i = \{m_i h(m_i)\}_{k_i}, i \in [1, n]$, where h is a cryptographic hash function and $h(m_i)$ serves the purpose of integrity protection. After encryption, the client stores the master key and sends all ciphertext to the cloud.

The advantage of the solution is that the client only needs to store one master key. But if the client wants to delete m_i, it has to delete k_i, which means that it has to delete K. Otherwise, if K is not deleted and it is revealed at a later time (possibly by external attack that compromises the client's computer), k_i can be recovered through $PRF(K, i)$. So for each deletion, the client has to choose a new K, re-generate k_i and other data keys, and finally re-encrypt all data items.

B. An Individual Key based Solution

To address the above problem, the client may adopt a different solution. It generates a sequence of n independent keys, denoted as $K = \{k_1, k_2, ..., k_n\}$, where k_i is used to encrypt m_i, $1 \leq i \leq n$. The client stores all encrypted data at the server side and keeps the keys by itself.

If the client wants to delete the i^{th} data item m_i, it finds the i^{th} key k_i, permanently deletes k_i from local storage, and sends the server a request to delete c_i. Since the key k_i is known only by the client, deleting k_i will make c_i undecryptable.

The weakness of the above approach is that the client has to manage all the keys, which is a burden if the system has a large number of data items. See the introduction for more detailed discussion.

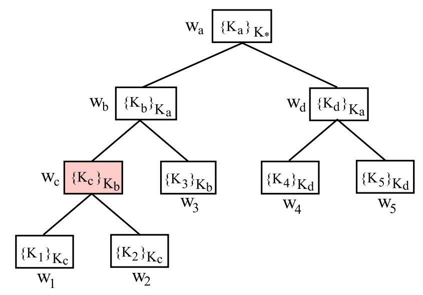

Fig. 1. A Recursively Encrypted Key tree (RERK) constructed on 5 keys

V. Deletion Based on Recursively Encrypted Key Tree

Before presenting our approach, we first introduce a novel data structure called *Recursively Encrypted Red-black Key tree* (RERK). The RERK design has the following four goals: (1) *Confidentiality* — after the keys are outsourced to the cloud, the RERK should be able to preserve the confidentiality of the keys. (2) *Integrity and correctness* — if the keys are lost by the cloud or a compromised cloud server does not send the client the correct key material, the client should be able to detect it. (3) *Efficiency* — the worst-case communication and computation cost of RERK operations are logarithmically bounded. (4) *Key assured deletion* — if the client wants to delete a key in RERK, the key will be made unrecoverable.

Before we describe the deletion algorithm in RERK, we must first explain how the RERK is constructed step by step for confidentiality, integrity, and efficiency, which are critical ingredients to set the stage for efficient assured deletion in a cloud environment.

A. Confidentiality

The client first constructs a red-black tree with n leaves. We denote the n leaves from left to right as w_1, w_2, ..., w_n. Each leaf w_i represents a key k_i in the sequence K. Recall that keys in K are used to encrypt data items. We call them *data keys*.

We use letter subscribes (such as w_a, w_b and w_c in Figure 1) to denote internal nodes, which helps distinguish them from leaves. For each internal node, the client randomly chooses an *auxiliary key* (used for encrypting other keys). Finally, the client arbitrarily picks a metakey k_*. Notice that the red color will show up as grey in black-n-white print in Figure 1 as well as other figures.

We use w_x to denote an arbitrary node in the tree, where x may be a number or a letter. Let k_x be the key of w_x, which may be a data key or an auxiliary key, depending on whether w_x is a leaf node or an internal node. Let p_x be the key of w_x's parent node. We define a value called *Encrypted Key* (EK) for node w_x as follows:

$$EK(w_x) = \begin{cases} \{k_x\}_{k_*} & \text{if } w_x \text{ is the root} \\ \{k_x\}_{p_x} & \text{otherwise} \end{cases} \quad (1)$$

It is the node's key encrypted by its parent's key, except for the root, whose key is encrypted by the metakey. An example is shown in Figure 1, where the EK value of each node is shown inside the box representing that node.

The client will then outsource the EK values of all nodes to the cloud. After that, it securely deletes all data/auxiliary keys and only keeps the metakey k_*.

All data/auxiliary keys are now stored in the cloud, but they are recursively encrypted from the root of RERK to the leaves. Only the client holds the metakey to decrypt them.

Key Lookup: When the client wants to look up for the i^{th} data key k_i, it will send a lookup request to the cloud server that handles this client. The server will reply with a node sequence from the i^{th} leaf node w_i to the root and their siblings in RERK. The siblings are used to ensure the integrity and correctness of the node sequence which will be explained next. By decrypting the keys recursively, the client can acquire the key k_i.

B. Integrity and Correctness

By recursive encryption, we minimize the amount of metadata that the client has to store, yet we are able to keep the confidentiality of the outsourced keys. However, a critical problem needs to be addressed before we can complete the tree construction: The client has no idea whether the key information sent back from the server is correct or not. Those EK values may have been corrupted or tampered intentionally by an intruder. So the client needs a mechanism to verify the integrity and correctness of the key information from the server.

The Merkle hash tree [14] has been widely used for integrity verification. However, we cannot directly combine the Merkle tree with our RERK because the former cannot verify index information: When a client wants to delete k_i, the server may send back the node sequence from another leaf node w_j to the root, which will pass the Merkle hash check and thus be able to trick the client to delete k_j instead. To address this problem, we adopt the rank idea [10] — which was originally applied to skip lists — into the Merkle tree construction.

Besides the EK value, each node w_x in the RERK carries two more values: a rank $r(w_x)$ and a tag $t(w_x)$. The rank is defined as the number of leaf nodes in the subtree rooted at w_x. For example, in Figure 1, $r(w_1)$ is 1, $r(w_b)$ is 3, and $r(w_a)$ is 5. The tag of a leaf node w_i is computed by hashing $EK(w_i)$ and $r(w_i)$, where a collision resistant hash function should be used. The tag of an internal node w_x is computed by hashing the concatenation of $EK(w_x)$, $r(w_x)$, and the tags of two child nodes. More specifically, let w_l and w_r be the child nodes of w_x, and we define

$$t(w_x) = h(EK(w_x)||r(w_x)||t\_(w_x)), \tag{2}$$

where $||$ is the concatenation operator and

$$t\_(w_x) = \begin{cases} NULL & \text{if } w \text{ is a leaf node} \\ h(t(w_l)||t(w_r)) & \text{otherwise} \end{cases} \tag{3}$$

Clearly, the tags are designed to implement the Merkle tree for integrity check of EK values and rank values. The ranks are designed to ensure that correct key information is returned from the server. The client outsources the ranks and tags of all nodes in the RERK to the cloud, while storing only the tag of the root.

After the client receives the node sequence from w_i to the root as well as their siblings, it verifies the integrity of the EK and rank information received from the server by re-computing the tags of the node sequence from w_i to the root. The client compares the re-computed tag of the root with the stored value. If they match, it confirms the integrity of the received EK and rank information, i.e., the EK and rank values are not tampered after being outsourced.

Next, from the rank values, the client can find out the number of leaves before w_i in the inorder traversal of RERK as follows: Initialize a variable v to zero. Walk through the node sequence (received from the server) backward from the root to w_i. When moving to a right child, add the rank of the left sibling to v. When moving to a left child, do nothing. After the walk-through is completed, v is the number of leaves before w_i in the inorder traversal of RERK. If v is equal to $i-1$, the client knows that the received w_i is the correct one; otherwise, the server has cheated.

We have shown above how to compute the index position of any data key in K by using the ranks of the left sibling nodes along the path from the root to the leaf in RERK. When we delete a leaf w_i, the ranks of the nodes on the path must be decreased by one, which automatically decreases the index position of all leaves after w_i by one. Similarly, when we insert a leaf node, the ranks of the nodes on the path to the root are increased by one, which automatically increases the index position of all leaves after the inserted one.

C. Efficiency

As the RERK is a red-black tree, two new values are defined for each node w_x in the RERK: a color $col(w_x)$ and a red-children counter $red(w_x)$. The color $col(w_x)$ is 1 if w_x is a red node, and it is 0 if w_x is a black node. The counter $red(w_x)$ is 0 (1 or 2) if w_x has no (one or two) red children. These values are set by the client during the tree operations but stored at the server. In order to ensure their integrity, they must be included in the tag computation together with the EK and rank values. We give the new tag definition as follows:

$$t(w_x) = h(EK(w_x)||r(w_x)||t\_(w_x)||col(w_x)||red(w_x)). \tag{4}$$

When nodes are inserted or deleted, RERK may become imbalanced. Then it will need rotation and color changing to re-balance the tree before the client re-computes and sends back the new EK values. The operations of red-black tree are highly efficient. It is easy to prove that re-balancing will only involve $O(\log n)$ nodes, and we will discuss the overhead issue after the key operations below.

Depend on different application scenarios, we can choose different self-balancing data structure to store EK values. Therefore, the red-black tree can be replaced by an AVL tree or a splay tree. In this paper, we assume that clients require frequent insertion and deletion. So according to the performance comparison in [17], we choose the red-black tree.

D. Key Operations

We present the deletion algorithm first. The confidentiality and integrity protection mechanisms embedded in RERK (Section V-A-V-B) ensure the correctness of this algorithm, as our security analysis will show. The red-black tree embedded in RERK (Section V-C) ensures its logarithmic worst-case overhead bound.

Key Deletion: Suppose the client wants to delete a data key k_i. It performs the following operations:

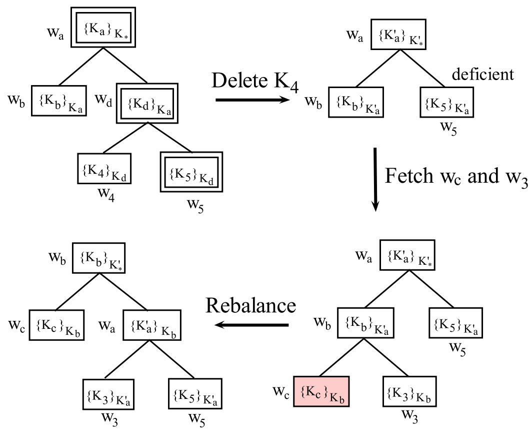

Fig. 2. Example for key deletion in the RERK. Double-boxes in the left top represent the node sequence from the leaf node to the root, and other nodes are their siblings.

1) The client looks up for the i^{th} key and the server returns the node sequence in RERK from the i^{th} leaf node w_i to the root and their siblings. The client constructs a partial RERK using these nodes and verifies their integrity and correctness through the embedded Merkle tree with rank information. After that, using the metakey k_*, it recursively decrypts all keys in the partial RERK.
2) The client removes the node w_i and replaces w_i's parent with its sibling node w_j. In the resulting partial RERK, it generates a new key for each node on the path from w_j's parent to the root. The tree will become imbalanced if a black node is removed (in our case, if w_i's parent is black), which triggers the standard algorithm for red-black tree re-balancing [6] in cooperation with the server. (Because the client only has a partial RERK, it may need to lookup another leaf node and retrieve $O(\log n)$ additional nodes from the server.) It is easy to prove that the red-black tree deletion only involves $O(\log n)$ nodes.
3) The client replaces the old metakey k_* with a new metakey k'_*. It re-computes the new EK values in its partial RERK using the new keys.
4) The client sends new values in its partial RERK back to the server and only keeps the new metakey k'_*.

An example is given in Figure 2, where k_4 (thus data item m_4) is deleted from the RERK in Figure 1. The left-top plot in Figure 2 is the partial RERK sent from the server to the client. After replacing w_d with w_5 and generating a new key for w_a, the RERK becomes imbalanced, as illustrated by the right-top plot. Following the standard re-balancing algorithm, the client needs to look up the key k_3 and fetch additional nodes w_c and w_3, as illustrated by the right-bottom plot. The result of re-balancing is shown by the left-bottom plot.

With k_* being permanently deleted by the client, even if the cloud server does not remove the EK value for k_i, there is no way for anyone to decrypt it for k_i. With k_i being unrecoverable, the corresponding data item m_i becomes unrecoverable even if the server does not remove the ciphertext c_i from its storage.

Key Insertion: While the RERK is designed to support assured deletion, we also need the insertion algorithm for completeness. Suppose the client wants to insert a data key k'_i at the i^{th} position. It performs the following operations:

1) The client looks up for the i^{th} key and the server returns the node sequence in RERK from the i^{th} leaf node w_i to the root and their siblings. The client constructs a partial RERK using these nodes and verifies their integrity and correctness. After that, using the metakey k_*, it recursively decrypts all keys in the partial RERK.
2) The client creates a new leaf node w'_i for k'_i. Then it replaces the node w_i with a new internal node w_{new} whose auxiliary key k_{new} is randomly selected. It assigns w'_i and w_i as the left and the right children of w_{new}. If the new node's parent (i.e., w_i's previous parent) is a red node, the tree will become imbalanced, which triggers the standard algorithm for red-black tree re-balancing [6]. Different from deletion, the partial RERK already contains all nodes for re-balancing.
3) The client re-computes the new EK values in its partial RERK using the new keys.
4) The client sends new values in its partial RERK back to the server.

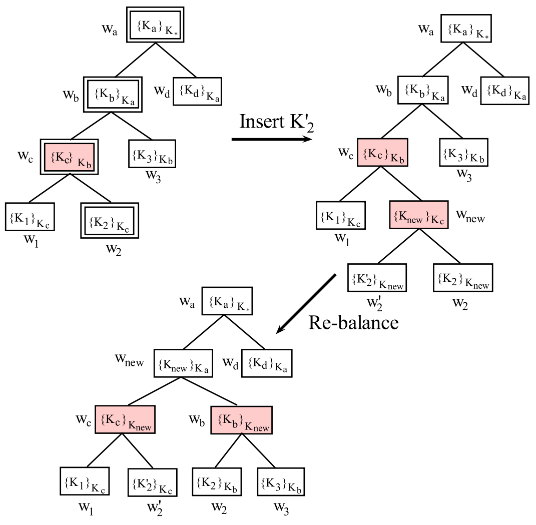

Fig. 3. Example for key insertion in the RERK

An example is given in Figure 3, where a new key k'_2 is inserted at the 2^{nd} position of the RERK in Figure 1. The left-top plot in Figure 2 is the partial RERK sent from the server to the client. After inserting the new internal node w_{new}, the partial RERK becomes imbalanced, as illustrated by the right-top plot. Based on the standard re-balancing algorithm, the client re-balance the partial RERK and the result of re-balancing is shown by the bottom plot.

VI. Assured Data Deletion Scheme

We give the set of algorithms that together form our assured data deletion scheme.

• $KeyGen(1^s) \rightarrow k$ is an algorithm executed by the client. It takes a security parameter 1^s as input and returns a symmetric key k. Note that k is randomly picked from the key space, so if the client runs this algorithm n times to generate n keys, these keys should be independent from each other.

• $Prepare(k_*, D) \rightarrow (\mathcal{RERK}, t_R, C)$ is an algorithm run by the client. It takes as input the metakey k_* and the user database D composed of a sequence of data items $\{m_i\}$, where i from 1 to n. The client first generates a sequence of keys, $k_1, k_2, ..., k_n$ using algorithm $KeyGen$, to encrypt data items m_1 through m_n into c_1 through c_n. Then the client constructs a RERK $\mathcal{RERK}$ based on the keys. Finally, the client outsources the RERK $\mathcal{RERK}$ and all ciphtertexts, $C = \{c_i\}$ to the server and only keeps the metakey k_* and the tag of the root of RERK t_R.

• $Lookup \rightarrow L$ is an algorithm run by the client. As input, it takes nothing as input and returns a lookup request $L = \{i\}$ to the server, where i is the index number of a data item.

• $GenProof(L) \rightarrow (c_i, P_i)$ is an algorithm run by the server. Upon receiving the lookup request, the server will first find the ciphertext c_i of the i^{th} data item. Then it generates a proof which contains a sequence of nodes from the i^{th} leaf node to the root and their siblings. Each node w_x in the sequence can be represented by a message $M_x = \{EK(k_x), r(w_x), red(w_x), t\_(w_x), col(w_x)\}$.

• $Verify(P_i, k_*, t_R) \rightarrow (\{true, k_i, \mathcal{RERK}_p\}, false)$ is an algorithm executed by the client. It takes the proof P_i, the metakey k_* and the tag of the root as input. The client will first verifies the integrity and correctness of the proof using the algorithm described in Section V-B. If the proof passes the verification, the client can construct a partial RERK $\mathcal{RERK}_p$, and extract the data key k_i by decrypting the EK values recursively. Otherwise, it will return false.

• $PrepareUpdate(i, info) \rightarrow R_U$ is an algorithm run by the client. The input contains the index of the block to be updated and the update related information $info = \{(insert, m'_i), delete\}$ which specifies what kind of update to perform. If the update action is insertion, then $info$ will include the new data item m'_i. The output contains the index of the block and the related information.

• $GenUpdateProof(R_U) \rightarrow P_U$ is an algorithm executed by the server. The server takes the update request R_U as input and outputs an update proof P_U. If the update request is insertion, then $P_U = P_i$, where P_i is the proof of the i^{th} leaf node. If the update request is deletion, as we mentioned in Section V-D, P_U may contain two proofs: P_i and P_j, where P_i is the proof of the i^{th} leaf node and P_j is the proof of another leaf node.

• $VerifyUpdateProof(P_U, k_*, t_R) \rightarrow (\{true, \mathcal{RERK}_p\}, false)$ is an algorithm run by the client. It verifies the update proof using the algorithm described in Section V-B. If the algorithm accepts, the client will construct a partial RERK $\mathcal{RERK}_p$ based on the proof. Otherwise, it will return false. Note that if the update action is deletion, P_U may contain two proofs. So the partial RERK $\mathcal{RERK}_p$ will include two sequences of nodes.

• $Update(\mathcal{RERK}_p, info) \rightarrow (\mathcal{RERK}'_p, k'_*, t'_R, c'_i)$ is an algorithm executed by the client. It takes the partial RERK $\mathcal{RERK}_p$ and the update related information $info$ as input. After performing the update on the partial RERK $\mathcal{RERK}_p$ using algorithm described in Section V-D, the client will return the new partial RERK $\mathcal{RERK}'_p$ to the server, delete local temporary files, and only store the new metakey k'_* and the new tag of root t'_R. If the update is insertion, the client will generate a new key k'_i to encrypt the new data item, $c'_i = \{m'_i\}_{k'_i}$ and return the ciphertext c'_i to the server.

VII. Security Analysis

Theorem 1: If there exist (1) a collision-resistant hash function which is used in the RERK construction and (2) an IND-CPA secure encryption scheme which is used to encrypt the outsourced data items, then the proposed assured deletion scheme is secure, i.e., for an arbitrary time T all data that have been deleted before time T will be unrecoverable in polynomial time even when the adversary is able to gain full control of servers before T and full control of clients after T.

Proof: We prove the theorem in two steps. First, we show that outsourcing the RERK tree is as good as keeping it locally because the probability for a compromised server to return an forged invalid proof (containing a required partial RERK tree) and successfully pass the verification algorithm $VerifyUpdateProof$ is negligibly small. Second, we show that if the adversary can recover the deleted text, it can break the encryption scheme used in the assured deletion scheme.

The partial RERK tree returned from a compromised server includes the node sequence from the root to the leaf w_i, as well as their sibling nodes. Refer to (4) and (3). Because the above Merkle tree construction is adopted to create parent-child hashing dependency by including the tags of child nodes in the hash input of any parent node, all nodal information must be truthful — the difficulty for the server to provide false nodal information without being detected is the same as breaking the security of the hash function used in Merkle tree. In other words, the probability for an invalid proof to pass the verification algorithm $VerifyUpdateProof$ is no greater than the probability of finding different input to produce the given hash output in the required partial RERK tree, which is negligibly small when a collision-resistent hash function is used. Moreover, as proved in [10], the rank value can uniquely determine the index of each key. Hence, the compromised server cannot cheat the client by returning another key in CMHT to pass the verification.

Next, given that the client has access to valid RERK, we show that a deleted data m_i will be unrecoverable by the adversary. Let k_i be the data key of m_i and c_i be the ciphertext. We consider three time phases. The first phase is from the creation of the data item to the beginning of the deletion. During this phase, the compromised server has the complete information about RERK. To know the data key, the compromised server needs to know the auxiliary key of the parent node in RERK. Recursively applying the same token, the compromised server needs to know the metakey in order to decrypt the root node of RERK. The metakey is however only known to the client (that is not compromised yet). Hence, the difficulty of acquiring k_i is the same as the difficulty of breaking the IND-CPA secure encryption scheme that RERK uses to recursively encrypt the auxiliary keys and the data items.

The second phase is from the beginning of the deletion to the accomplishment of deletion. We argue that if the client successfully deletes one key in RERK, the compromised server cannot recover the key even if it acquires the newest metakey after deletion. According to the deletion algorithm described in Section V-D, the client first deletes the key, then replaces all auxiliary keys of nodes on the path from the parent of the deleted key to the root and the metakey. Next it re-balance the partial RERK tree and encrypt all keys in the partial RERK tree transitively by using a new metakey k'_*, and permanently delete the old metakey. *Here, the important point is that the key sequence from the deleted key to the old metakey is not in the reconstructed partial RERK, and therefore k_i is never transitively encrypted by the new metakey k'_* through a sequence of intermediate auxiliary keys.* After the partial RERK is sent back to the server, since the partial tree does not carry any information about k_i, and all keys are randomly generated, no new information about k_i is revealed to the compromised server.

The third phase starts after the client has finished the deletion. The compromised server acquires k'_*, but not the original meta k_*, which has already been permanently deleted by the client. The compromised server only has the original ciphertexts of k_i transitively encrypted by k_*. The knowledge of k'_*, which has no relation with k_*, does not provide any help in decryption. Even the auxiliary keys used in the transitive encryption of k_i are totally replaced when k'_* is introduced in the second phase. Hence, if all keys in RERK are randomly generated, k'_* is useless to the decryption of any node in the sequence from the root to w_i and w_j in the original RERK before deletion.

Now suppose the adversary has a way to recover the deleted data item m_i with non-negligible probability. Based on the above analysis, $\mathcal{A}$ has no knowledge about the data key k_i that encrypts m_i, nor does it know the corresponding meta key k_* or any auxiliary key that leads to k_i. It only has the knowledge of ciphertext c_i. This means that the adversary can break the encryption scheme, which is against the theorem assumption that the adopted encryption scheme is IND-CPA secure. ■

VIII. Evaluation

We implement a cloud storage server on Amazon Elastic Compute Cloud (Amazon EC2) system. By purchasing an "instance" from Amazon EC2, we can completely control the remote resources and run the server programs on the instance.

We evaluate our solution in terms of communication and computational overhead. When a client performs deletion, lookup and insertion on a key, the server will send back $O(\log n)$ nodes in the RERK, where n is the total number of data items. Hence, the communication overhead is $O(\log n)$. It takes a constant time for the client to process each node. In addition, the red-black tree rotation has a complexity of $O(\log n)$. Hence, the overall computation overhead is also $O(\log n)$.

A. Experimental Setting

Our experiments are performed between two parties: the client and the server. We implement cloud storage servers on Amazon EC2. Each server instance has the following parameters: 2 virtual cores, each with 2 Compute Units; 7.5 GB RAM; 850 GB instance storage; Microsoft Windows Server 2008 R2 Base 64-bit. Note that although Amazon S3 provides cloud storage services, developers cannot directly run programs on Amazon S3. We use an ordinary desktop computer in our lab for the client, with the following configuration: Intel Core i7-3770 3.40 GHz, 8 GB RAM, 1 TB driver, and Windows 8 Professional 64-bit.

We use Secure Hash Algorithm-1 (SHA-1) [4] in the RERK. SHA-1 produces a 160-bit message digest. We choose Advanced Encryption Standard (AES) [8] to encrypt each data item and each key. AES has a key size of 128, 192, or 256 bits. In our implementation, we use 128-bit keys.

B. Communication Overhead

We measure the communication overhead between the client and the server through experiments, and the results are shown in Figure 4. The x-axis shows the total number of data items stored in the cloud in logarithmic scale. The y-axis shows the average communication overhead in KB. To measure the average communication overhead of deleting a data key, we tries to delete each data key in the RERK and count the number of bytes in the client message and the number of bytes in the server messages that carry the nodes involved. Similarly, we perform insertion and lookup on each data key and measure the average communication overhead among all keys.

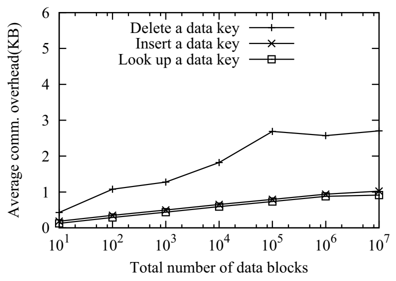

Fig. 4. Average Communication overhead between the client and the server. The x-axis shows the total number of data items in logarithmic scale. The y-axis shows the average communication overhead in KB.

Clearly, all measured communication overheads increase logarithmically with respect to the number of data items, demonstrating good scalability. For a data set of 10^6 items, the communication overhead can fit in a few IP packets of 1500 bytes each in most cases.

C. Computational Overhead

Next, we measure the computational overhead of the server and the client separately. On the server side, the main computational overhead is to construct server messages and send relevant nodes in the RERK to the client in these messages. The set of nodes to be sent only depends on the data key, regardless of what operation it is. On the client side, upon receiving the nodes from the server, it computes tags to verify the integrity of the information carried by the nodes, and uses ranks to determine if correct nodes are received. After that, the client performs the intended operation, whether it is deletion, insertion or lookup. It performs tree rotation if needed. Finally, it sends new information back to server. The client's computation overhead varies for different operations. Hence, we measure them separately.

1) Client Computation: Figure 5 shows the computational overhead of the client. We perform the experiments on the desktop computer mentioned above. The most costly operation is deletion, which is followed by modification, then insertion, and finally lookup (query). The difference is due to (1) re-computation of nodal information such as EK and tag and (2) re-balancing of the tree. Deletion requires significant overhead on both, whereas lookup requires neither. All overheads scale logarithmically with respect to the number of data items. When there are 10^7 items, it takes the client about 1.2ms to delete a key.

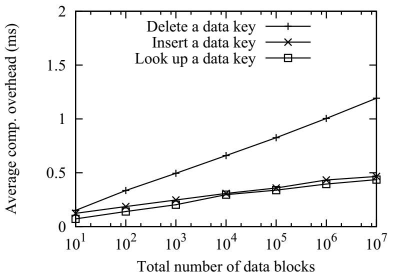

Fig. 5. Client computational overhead. The x-axis shows the number of data items in logarithmic scale. The y-axis shows the average computational time of the client.

2) Server Computation: We perform lookups on all keys. Figure 6 shows the average time it takes the server to process each lookup; the time for the server to handle other operations (deletion / insertion) is the same. Clearly, the computational overhead of the server increases logarithmically with respect to the total number of data items. When the number of data items is 10^7, it takes about 0.4ms to process a request. Our EC2 server has limited capacity. In real world, the cloud servers are expected to be much more powerful and should be able to process requests at much higher rates.

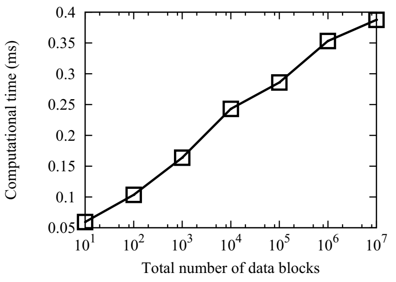

Fig. 6. Server computational overhead. The x-axis shows the number of data items in logarithmic scale. The y-axis represents the average time for the server to process a client request.

IX. Conclusion

The development of cloud computing brings a number of security problems. This paper presents a two-party solution for protecting the privacy of deleted data that has been outsourced by the clients to the cloud. The main challenges in addressing this problem are how to avoid burdening the clients with key management and how to make key outsourcing work in a two-party model. That is, we want to delegate the key management job to the cloud while preserving the confidentiality, integrity and correctness of the keys. We design a new data structure named Recursively Encrypted Red-black Key tree (RERK) to fulfill these design goals. We implement our solution on the Amazon EC2 system, and the evaluation results show that the proposed RERK is viable even with limited resources deployed in the experiments.

X. Acknowledgements

This work was supported in part by Cisco Systems, the US National Science Foundation under grant CNS-1115548, and the National Natural Science Foundation of China under grant 61170277.

References

[1] C. Arora and M. Turuani. Adding Integrity to the Ephemerizer's Protocol. *Proc. of AVoCS*, 2006.
[2] C. Arora and M. Turuani. Validating Integrity for the Ephemerizers Protocol with CL-Atse. *Formal to Practical Security*, 2009.
[3] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable Data Possession at Untrusted Stores. *Proc. of CCS*, 2007.
[4] J. Burrows. Secure Hash Standard. Technical report, DTIC Document, 1995.
[5] C. Castelluccia, E. D. Cristofaro, A. Francillon, and M. Kaafar. EphPub: Toward Robust Ephemeral Publishing. *Proc. of ICNP*, 2011.
[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. *The MIT Press, ISBN 0-262-03141-8, McGraw-Hill, ISBN 0-07-013143-0*, 1986.
[7] B. Crispo, M. Dashti, S. Nair, and A. Tanenbaum. A Hybrid PKI-IBC Based Ephemerizer System. *Proc. of EuroPKI*, 2009.
[8] J. Daemen and V. Rijmen. The Design of Rijndael: AES–the Advanced Encryption Standard. *Springer-Verlag, ISBN 3-540-42580-2, New York*, 2002.
[9] J. Douceur. The Sybil Attack. *Proc. of IPTPS*, 2002.
[10] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic Provable Data Possession. *Proc. of CCS*, 2009.
[11] R. Geambasu, T. Kohno, A. Levy, and H. Levy. Vanish: Increasing Data Privacy with Self-destructing Data. *Proc. of USENIX*, 2009.
[12] A. Juels and B. K. Jr. PORs: Proofs of Retrievability for Large Files. *Proc. of CCS*, 2007.
[13] S. Kamara and K. Lauter. Cryptographic Cloud Cstorage. *Proc. of FC*, 2010.
[14] R. Merkle. A digital signature based on a conventional encryption function. *Advances in Cryptology (CRYPTO)*, 1988.
[15] R. Perlman. File System Design with Assured Delete. *Proc. of SISW*, 2005.
[16] R. Perlman. The Ephemerizer: Making Data Disappear. *Information System Security*, 2005.
[17] B. Pfaff. Performance Analysis of BSTs in System Software. *Proc. of SIGMETRICS*, 2004.
[18] A. Rahumed, H. Chen, Y. Tang, P. Lee, and J. Lui. A Secure Cloud Backup System with Assured Deletion and Version Control. *Proc. of ICPPW*, 2011.
[19] H. Shacham and B. Waters. Compact Proofs of Retrievability. *Proc. of ASIACRYPT*, 2008.
[20] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-to-peer Lookup Service for Internet Applications. *Proc. of SIGCOMM*, 2001.
[21] Q. Tang. From Ephemerizer to Timed-Ephemerizer: Achieve Assured Lifecycle Enforcement for Sensitive Data. *Technical Report TR-CTIT-10-01*, 2010.
[22] Y. Tang, P. Lee, J. Lui, and R. Perlman. FADE: Secure Overlay Cloud Storage with File Assured Deletion. *Proc. of SecureComm*, 2010.
[23] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling Public Verifiability and Data Dynamics for Storage Security in Cloud Computing. *Proc. of ESORICS*, 2009.
[24] S. Wolchok, O. Hofmann, N. Heninger, E. Felten, J. Halderman, C. Rossbach, B. Waters, and E. Witchel. Defeating Vanish with Low-cost Sybil Attacks against Large DHTs. *Proc. of NDSS*, 2010.
[25] L. Zeng, Z. Shi, S. Xu, and D. Feng. SafeVanish: An Improved Data Self-Destruction for Protecting Data Privacy. *Proc. of CloudCom*, 2010.