2014 IEEE 34th International Conference on Distributed Computing Systems

Two-Party Fine-Grained Assured Deletion of Outsourced Data in Cloud Systems

Zhen Mo Yan Qiao Shigang Chen
Department of Computer Science, University of Florida, Gainesville, FL 32611, USA

***Abstract*—With clients losing direct control of their data, this paper investigates an important problem of cloud systems: 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 guarantee inaccessibility of deleted data when the data is not in their possession? Using a novel key modulation function, we design a solution for two-party fine-grained assured deletion. The solution does not rely on any third-party server. Each client only keeps one or a small number of keys, regardless of how big its file system is. The client is able to delete any individual data item in any file without causing significant overhead, and the deletion is permanent — no one can recover already-deleted data, not even after gaining control of both the client device and the cloud server. We validate our design through experimental evaluation.**

I. Introduction

Cloud storage systems have gained firm traction in the marketplace, but they also bring concern from the fact that users lost physical control of their outsourced data, while they cannot fully trust the cloud service providers, as the servers may be compromised, the providers may be forced to hand data over to law enforcement, their employees who have access to the data may sell it under the table to marketers or corporate competitors, to give a few examples. To address such concern, much research work has been done to ensure the integrity of the outsourced data [1], [2], [3], [4]. One of their goals is to verify the *accessibility* of the data 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 ensure the *inaccessibility* of their data when the data is not in their possession? One straightforward solution is to encrypt data before outsourcing. The client keeps the encryption key, while the server keeps the encrypted data. To make sure that data cannot be recovered in the future, the client only needs to securely delete the key. But this simple approach has serious problems when the outsourced file system is large:

• *If the client uses one key to encrypt all files*, whenever it deletes anything from any file, it will have to re-encrypt everything else with a new key because otherwise the whole file system would become inaccessible after the old key is deleted.

• *If the client uses a different key for each file*, there will be numerous keys for the client to manage, which may become a serious burden particularly for light-weight client devices such as tablets. More importantly, the client has to change the key of a file even when it deletes just one data item, e.g., a retired employee record from a large roster, an erroneous entry of a sensor data file, a sensitive transaction in a secret financial book, an email from a mail backup file, or one record from a large database file. To make one data item inaccessible, the client has to retrieve the entire encrypted file from the server, decrypt it, remove one item, permanently delete the old key, and choose a new key to re-encrypt the entire file.

• To avoid the above overhead, one way is to *assign a different key to each data item in each file*, but the number of keys may become astronomical. Especially when the data-item size is comparable to the key size, as the volume of keys rivals the volume of data itself, the benefit of outsourcing diminishes.

The prior art relies on two approaches to relieve the key management burden from the client. The first approach is to shift key management burden to third-party servers (also called ephemerizers for timed deletion) [5], [6], [7]. Assured deletion relies on the security of third-party servers. However, if we cannot fully trust the cloud service providers, shouldn't we place the same benefit of doubt on the third-party servers? For example, if a Federal agency has a court order to force the cloud and the third party to surrender the data and keys of a company under investigation, no matter how hard the client tries to delete its data, it will be useless. (In comparison, our solution will ensure the effectiveness of deletion up to the moment of the client's device being seized.) Moreover, the third-party servers cause issues of performance degradation and availability because their key service is needed for all data operations.

The second approach is to reduce the number of keys by using each key to protect multiple files which should be deleted at the same future time [5], [6], which belong to the same class and are expected to be deleted together [6], or which share the same access policy [7]. This approach cannot support efficient fine-grained deletion on individual data items of each file in general-purpose file systems, because any such deletion will require an old key to be replaced by a new key and all files under that old key to be re-encrypted.

Now if we adapt the prior work [5], [6], [7] for a two-party solution by merging the function of the third party to the client and support fine-grained deletion by letting the client keep one key for each data item, then the problem of too many keys comes back. The objective of this paper is to design a new solution for two-party fine-grained assured deletion. It does not rely on any third-party server, yet the client only keeps one or a small number of keys, regardless of how big the file system is. The client should be able to delete each individual data item without causing any other data to be re-encrypted, and the deletion is permanent — no one can recover already-deleted data, not even after gaining control of both the client device and the cloud server.

We take two steps to design our solution. In the first step, we let the client keep one master key for every file. We develop a novel key modulation function that allows the client to delete

1063-6927/14 $31.00 © 2014 IEEE
DOI 10.1109/ICDCS.2014.39

each individual data item without having to re-encrypt the rest of the file even though the master key has been changed. In the second step, we outsource the master keys of all files to the cloud so that the client only needs to keep one or a small number of higher-level control keys.

The rest of the paper is organized as follows: Section II defines the problem and the threat model. Section III motivates for our idea of key modulation. Section IV describes our new solution for two-party fine-grained assured deletion. Section V explains how to outsource the master keys of files to the cloud. Section VI presents the experimental results. Section VII discusses the prior art. Section VIII draws the conclusion.

II. Preliminaries

A. System Model

A cloud system consists of two parties: (1) *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) 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 retrieve their data or change the data by sending requests to the servers. Upon receiving the requests, the servers will perform operations of deletion, insertion, modification, and data access.

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. Moreover, we want to implement *fine-grained deletion* which allows us to safely and efficiently delete small data items in large files without having to re-encrypt the files. Finally, we require a *two-party solution* that does not involve any third party in the operations between clients and servers.

C. Threat Model and Security Definition

Consider a data item that is deleted from the cloud by a client at time T. We adopt the worst-case threat 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 device after time T.

The first attacker 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 requests.

The second attacker capability reflects the possibility that the client's device may be compromised after T. In this case, the attackers have access to everything stored on the client side, including any key materials that the client has. (If the attackers manage to compromise the client's device before T, they will know the data, which has not been deleted yet.)

For the security definition, a solution for deleting data in a cloud system is secure *if all data that have been deleted before time T will be provably 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, assuming the existence of a collision-resistant hash function such that it is polynomially infeasible to find two hash inputs that produce the same output or find a hash input to produce a specific output*.

D. Scope of This Work

The scope of this work is solely about assured deletion. We view the integrity issues in data storage and data access as complementary problems that have been solved in [1], [2], [3], [4], which can provide proof of data possession in the cloud and allow the client to correctly access each data item. The above schemes are originally designed for outsourcing data in plaintext. It is trivial to make them work on encrypted data.

In addition, we are not concerned with other types of security problems that the attackers may inflict. For example, if an attacker compromises the cloud server and does not perform the required operations, our solution can still protect the privacy of successfully-deleted data, but it does not promise to guard against other types of damage on accessibility and integrity of client data that have not been deleted. In fact, once an attacker has full control of the server, other than damaging data accessibility by not following the required operations, it may simply erase client data to make it inaccessible. Solutions to such problems are beyond the scope of this paper.

Our design is suited for users that have large amounts of outsourced data and yet look for light-weight client solution (that may possibly be operated from mobile devices). For users with limited data, there are simpler solutions such as ones discussed below.

III. Motivation

We first consider a single file, and delay multi-file multi-user discussion to Section V. Below we analyze two simple two-party solutions to give the motivation for our new approach of key modulation. We use $\{\mathbf{m}\}_\mathbf{k}$ as the notation for encrypting m with key k.

A. Master-key Solution

Consider an arbitrary client file of n data items, denoted as $\{m_1, m_2, ..., m_n\}$. The client selects a master key K. From the master key, it derives a different data key $k_i = PRF(K, i)$ for each item m_i, where PRF is a pseudo random function. The client encrypts each data item with its corresponding key, $\{m_i H(m_i)\}_{k_i}, i \in [1, n]$, where H is a collision-resistant hash function. Given two different inputs, the hash outputs will be different with practically-assured high probability. $H(m_i)$ 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 above master-key solution is that the client only needs to store one key. However, if the client wants to delete a data item m_j, it must delete the master key K in order to delete k_j. If K is not deleted and it is revealed at a later time (possibly due to external attack

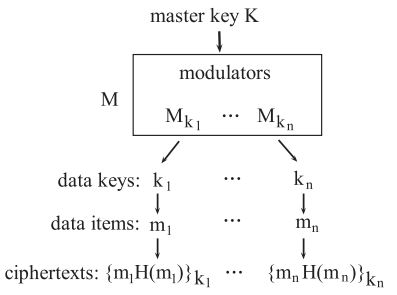

Figure 1. Key modulation

that compromises the client's computer), k_j can be recovered through $PRF(K, j)$. But the problem is that, if the master key is changed, the keys for all other data items are changed, too. Hence for each deletion, after choosing a new master key K', the client has to re-generate all remaining data keys $PRF(K', i)$ and re-encrypt all remaining data items, with $O(n)$ communication/computation overhead.

B. Individual-key Solution

To address the above problem, the client may adopt a different solution. It generates a sequence of n independent keys, denoted as $\{k_1, k_2, ..., k_n\}$, where k_i is used to encrypt m_i, $1 \leq i \leq n$. The client sends all encrypted data to the cloud while keeping the keys by itself.

If the client wants to delete a data item m_j, it finds the corresponding key k_j, deletes it permanently from local storage, and sends the server a request to delete c_j. Since k_j is known only by the client, deleting k_j will make c_j undecryptable, regardless of whether the client removes c_j timely or not.

The advantage of the individual-key solution is that its communication/computation overhead for deletion is $O(1)$, but its weakness is that the client may have to keep too many keys, particularly when the size of each data item is comparable to the key size — in this case, the total volume of keys rivals the data itself, which would defeat the purpose of outsourcing.

C. Our Approach of Key Modulation

Can we design a new approach to avoid the problems of the above solutions, while obtaining the benefits of both: *small client storage overhead* and *small deletion overhead*? To achieve the former, we let the client only keep a master key K. On the one hand, all data keys must be derived from this master key, and when any data key k is deleted, the master key K must be deleted in order to make sure that k is not recoverable in the future. On the other hand, even as the master key is changed to a new value K', we want *other data keys to stay the same, such that the client does not have to re-encrypt all other data items* after one item is deleted. This requirement necessarily means that the data keys are not determined solely by the master key in the way that the previous solution of $PRF(K, i)$ does.

Our idea is called *key modulation*, as illustrated in Figure 1. The data keys are derived from the master key K and a set M of values called *modulators*. More specifically, each data key k is determined by K and a unique subset M_k of modulators through a one-way function $k = F(K, M_k)$. The master key is stored at the client, while the modulators are stored in the cloud, unencrypted. To delete k, the client will permanently delete K and choose a new master key K'. In addition, it will adjust the values of $O(\log n)$ modulators in $M - M_k$ such that all other data keys k' stay the same, i.e., $k' = F(K', M'_{k'}) = F(K, M_{k'})$, where $M_{k'}$ is the subset of modulators for k' before deletion and $M'_{k'}$ is the same subset after deletion (with one modulator having a new value).

For the deleted key k, since we do not change any modulator in M_k, $F(K, M_k) \neq F(K', M_k)$. Therefore, even if the new master key K' is compromised in the future, the deleted key $k = F(K, M_k)$ is not recoverable after K is permanently deleted.

The challenge is to design a key modulation function F such that after one data key is deleted and the master key is changed, we can modify $O(\log n)$ modulators to keep the remaining $(n-1)$ data keys unchanged.

IV. Key Modulation Function

The design of our key modulation function has three components: (1) the formula of the function $F(K, M_k)$, (2) how to select the modulators M_k for each data key k, and (3) which modulators to change and how to change for each deletion. This section will present the design of these components.

A. Modulated Hash Chain

We formulate the function $F(K, M_k)$ as a modulated hash chain. The classical hash chain has the following format [8]:

$$H(...H(H(H(K)))...).$$

We treat M_k as an ordered list of modulators, denoted as $\langle x_1, x_2, ..., x_l \rangle$. A modulated hash chain is defined as follows:

$$F(K, M_k) = H(...H(H(K \otimes x_1) \otimes x_2)... \otimes x_l), \quad (1)$$

where $\otimes$ is the XOR operator and H is a one-way, collision-resistant hash function that produces pseudo-random output. Let $\emptyset$ be an empty list and $M_k^{(i)}$, $0 \leq i \leq l$, be a prefix of M_k, containing the first i modulators in M_k. An equivalent recursive definition of the modulated hash chain is given below:

$$\begin{aligned} F(K, \emptyset) &= K; \\ F(K, M_k^{(i)}) &= H(F(K, M_k^{(i-1)}) \otimes x_i), \forall\ 1 \leq i \leq l. \end{aligned} \quad (2)$$

After a single modulator in M_k is changed from x_i to x'_i, we denote the resulting list as $M_k | x_i \rightarrow x'_i$.

Lemma 1: The output of a modulated hash chain $F(K, M_k)$ will stay the same after the master key is changed from K to K' and the value of a single modulator x_i, $1 \leq i \leq l$, is changed to

$$x'_i = x_i \otimes F(K, M_k^{(i-1)}) \otimes F(K', M_k^{(i-1)}). \quad (3)$$

That is,

$$F(K, M_k) = F(K', M_k | x_i \rightarrow x'_i). \quad (4)$$

The proof is straightforward and can be found in the appendix.

B. Modulation Tree

We organize all modulators in a tree structure, based on which we will determine a subset M_k for each data key k. The hierarchical tree structure allows us to share modulators among the data keys in such a way that we only need to modify $O(\log n)$ modulators in order to keep $(n-1)$ keys unchanged after deleting a data key.

Before outsourcing a file F of n data items to the cloud, the client randomly picks a master key K and then builds a *modulation tree*, which is a complete binary tree with each internal node having two children and each leaf node representing a data key. The client assigns each leaf node a randomly-selected *leaf modulator*, and assigns each link in the tree a *link modulator*, as illustrated in Figure 2. No modulator is assigned to any internal node.

Each leaf node encodes a data key. For convenience, we refer to the leaf that encodes key k as "node k". Let $P(k)$ be the path from the root to node k. The client turns the link modulators along the path and the leaf modulator at the end of the path into an ordered list M_k, and computes a data key $k = F(K, M_k)$ by applying the modulated hash chain. Note that the path $P(k)$ is essentially a graphical representation of the modulator list M_k, as illustrated in Figure 2 by the bold lines.

Each data key k is used to encrypt a data item m in F. The ciphertext is $\{m||r, H(m||r)\}_k$, where r is a globally unique number that is appended to m to make it unique. To generate r, the client maintains a global counter whose value is increased whenever the client inserts a new block to any file. In the rest of the paper, we will simply treat $m||r$ as m, knowing that there are no identical data blocks after the appendix of r. The ciphertext may be stored at the leaf node, and a double linked list can be used to keep an order amongst the encrypted data items. It may also be stored in a separate data structure, and pointers are used to map between the leaf nodes and the corresponding ciphertexts.

The client keeps the master key and sends the modulation tree as well as all ciphertexts to the cloud. One requirement is that all modulators in the tree should have different values. As the modulators are randomly selected by the client, if the size of modulators is large enough (such as 160 bits), the chance of collision will be exceedingly small. However, if the client ever selects a duplicate modulator during deletion/insertion, as the tree is now in the cloud, the server should inform the client to re-perform the operation with a different modulator. If the server does not do so, there will be no harm to assured deletion because the client will refuse to operate on duplicate modulators from the server (see the proof of Theorem 2).

C. Modulator Adjustment Algorithm for Deletion

We describe a *modulator adjustment algorithm* that modifies $O(\log n)$ modulators to keep all data keys except for the deleted one unchanged after the client changes the master key. We also prove the security of this algorithm that ensures the deleted date key is unrecoverable even if the new master key is revealed in the future.

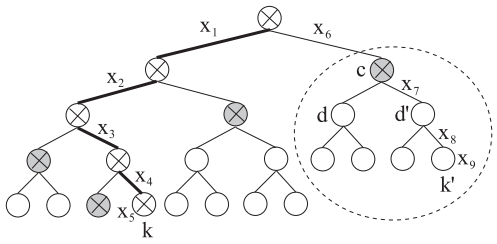

Figure 2. A modulation tree, where each leaf node is assigned a leaf modulator such as x_5, and each link is assigned a link modulator such as x_1 through x_4. The path $P(k)$ from the root to the leaf node k is drawn in bold lines; it is a graphical representation of $M_k = \langle x_1, x_2, x_3, x_4, x_5 \rangle$. The nodes in the cut C are shaded. $M_c = \langle x_6 \rangle$ is a prefix of $M_{k'}$ for any leaf k' in the sub-tree rooted at c.

Suppose the client wants to delete a data item m,[1] which is identified by a record ID that is carried by m, the byte offset in the file,[2] or other means of indexing. The client sends the cloud server a request, including the ciphertext of m and indexing information (such as a record ID). The server finds the encrypted item in its storage and the corresponding leaf node k in the modulation tree. It constructs a sub-tree of size $O(\log n)$, denoted as $MT(k)$, consisting of nodes on the path from the root to leaf k and the siblings of these nodes. The set of siblings, denoted as C, serves as a $(n-1)$-cut that separates all $(n-1)$ leaf nodes other than node k from the root, as illustrated by Figure 2, where $MT(k)$ consists of nodes with cross inside and C consists of shaded nodes. The client expects all modulators in $MT(k)$ to have different values. Otherwise, it will not accept $MT(k)$ for further operation.

The server sends $MT(k)$ to the client, including only the modulators. The client extracts M_k from the path $P(k)$, computes $k = F(K, M_k)$, and uses k to decrypt the ciphertext into $\{mH(m)\}$. Only if the decryption is successful, i.e., the hash of m matches $H(m)$ from the ciphertext, the client accepts $MT(k)$.

The client updates the master key from K to K', but it will not change any link modulators in $MT(k)$. Let $P(c)$ to be path from the root to a node $c \in C$, and M_c be the list of link modulators along $P(c)$. Note that M_c is a prefix of $M_{k'}$ for any data key k' encoded by a leaf node within the sub-tree rooted at c. See the circled sub-tree in Figure 2 for an example. The client computes

$$\delta(c) = F(K, M_c) \otimes F(K', M_c). \tag{5}$$

The client sends $\{\delta(c) \mid c \in C\}$ back to the sever. For each node c in the cut C, if it is an internal node, the sever adjusts the modulators on its child links, (c, d) and (c, d'), as follows:

$$\begin{aligned} x_{c,d} &:= x_{c,d} \otimes \delta(c) \\ x_{c,d'} &:= x_{c,d'} \otimes \delta(c), \end{aligned} \tag{6}$$

[1]Recall that we view the integrity issues in data storage and data access as complementary problems that have been solved in [1], [2], [3], [4], which can provide proof of data possession in the cloud and allow the client to correctly access each data item.

[2]The server divides the byte offset by the item size to identify which data item should be deleted. If variable item sizes are allowed, the size of each data item is stored with the ciphertext, such that the cloud server may sequentially scan the encrypted items and accumulate the sizes until the specified offset is reached.

where ":=" is the assignment operator, $x_{c,d}$ is the link modulator on (c, d), and $x_{c,d'}$ is the link modulator on (c, d'). If c is a leaf (i.e., the sibling of k), the server adjusts the leaf modulator:

$$x_c := x_c \otimes \delta(c), \tag{7}$$

where x_c is the leaf modulator of node c.

We have the following theorems. Theorem 1 ensures the correctness of our solution in not re-encrypting other data items after the master key is changed. Theorem 2 ensures the security of our solution in making the outsourced data unrecoverable after deletion. Their proof can be found in the appendix.

Theorem 1: For an arbitrary leaf node k, after the master key is changed and the modulator adjustment algorithm is performed on $MT(k)$, all data keys remain unchanged except for the key k.

Theorem 2: Suppose the key modulation function F uses a collision-resistant hashing function H. That is, it is polynomially infeasible to find two hashing inputs that produce the same output or find a hash input to produce a specific output. For an arbitrary leaf node k, after the master key is changed and the modulator adjustment algorithm is performed on $MT(k)$, the data key k becomes unrecoverable in polynomial time even if the new master key is revealed in the future.

We want to point out that the proof of Theorem 2 shows that the server cannot temper with modulators to circumvent the deletion. See the appendix for details.

The size of $MT(k)$ sent from the server to the client is $O(\log n)$. The size of $\{\delta(c) \mid c \in C\}$ sent from the client to the server is also $O(\log n)$. Hence, the communication complexity of the modulator adjustment algorithm is $O(\log n)$. The computation overhead is dominated by the computation of $\{\delta(c) \mid c \in C\}$, which can be done in $O(\log n)$ time: Because $F(K, M_k^{(i)}) = H(F(K, M_k^{(i-1)}) \otimes x_i)$, we can recursively compute $F(K, M_k^{(i)})$, $0 \le i \le l$, in $O(\log n)$ time, where l is the size of M_k, which is $O(\log n)$. Similarly, we can recursively compute $F(K', M_k^{(i)})$, $0 \le i \le l$, in $O(\log n)$ time. Any node c in the cut C is the sibling of a node on $P(k)$. That means M_c has a prefix of $M_k^{(j)}$ for some $j \in [0, l)$, plus one extra link modulator x_*. Hence, $F(K, M_c)$ can be computed in $O(1)$ time from $H(F(K, M_k^{(j)}) \otimes x_*)$. Similarly, $F(K', M_c)$ can also be computed in $O(1)$ time. The size of C is $O(\log n)$. Overall, it will take $O(\log n)$ time to compute $\{\delta(c) \mid c \in C\}$, including the time spent on $F(K, M_k^{(i)})$ and $F(K', M_k^{(i)})$, $0 \le i \le l$.

D. Balancing Algorithm

In order to keep the worst-case performance at $O(\log n)$, we want to restore the modulation tree back to a complete binary tree after deletion. Let t be the last leaf node at the last level of the modulation tree. See Figure 3. After we delete node k, we will move t to the location of node k.

The server sends the client the path $P(t)$ from the root to node t, together with the sibling s of t. The client extracts M_s from $P(s)$, which is the same path as $P(t)$ except for the last link. Let p be the parent of s and t. The client extracts M_p from $P(p)$, which is the sub-path of $P(s)$ without the last link. The balancing algorithm has two steps:

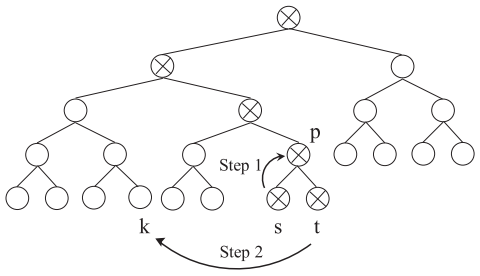

Figure 3. Balancing the tree after k is deleted. The server sends the nodes with cross inside to the client.

• Step 1: *remove node t from the tree*: The client computes a new leaf modulator for node s as follows:

$$x'_s = F(K', M_p) \otimes H(F(K, M_p) \otimes x_{p,s}) \otimes x_s, \tag{8}$$

where x_s is the original modulator for node s, and $x_{p,s}$ is the link modulator on (p, s). The client sends x'_s to the server, which removes node t from the tree, replaces parent p with node s, and assigns x'_s as the new leaf modulator for node s. Below we prove that the data key encoded by node s will stay unchanged. The new key is computed from the modulators in M_p and x'_s. Let "+" be the operator that concatenates two lists.

$$\begin{aligned}
&F(K', M_p + \langle x'_s \rangle) \\
&= H(F(K', M_p) \otimes x'_s) && \text{by (2)} \\
&= H(F(K', M_p) \otimes F(K', M_p) \otimes H(F(K, M_p) \otimes x_{p,s}) \otimes x_s) \\
&= H(H(F(K, M_p) \otimes x_{p,s}) \otimes x_s) \\
&= H(F(K, M_p + \langle x_{p,s} \rangle) \otimes x_s) && \text{by (2)} \\
&= F(K, M_p + \langle x_{p,s}\ ,\ x_s \rangle) && \text{by (2)} \\
&= F(K, M_s),
\end{aligned}$$

where M_s is the original modulator list for s before removal of t.

• Step 2: *insert node t to the place of node k*: This step is performed only if node t is not node k. Let p' be the parent of node k in $MT(k)$, and $M_{p'}$ be the list of modulators extracted from $P(p')$. The client randomly selects a link modulator for (p', t). The client computes a new leaf modulator for node t as follows:

$$x'_t = F(K, M_p + \langle x_{p,t} \rangle) \otimes F(K', M_{p'} + \langle x_{p',t} \rangle) \otimes x_t, \tag{9}$$

where $x_{p,t}$ is the link modulator on (p, t) when t is at its original location. The client will send $x_{p',t}$ and x'_t to the server. Below we prove that the data key encoded by t remains the same after it is moved to the new location.

$$\begin{aligned}
&F(K', M_{p'} + \langle x_{p',t}\ ,\ x'_t \rangle) \\
&= H(F(K', M_{p'} + \langle x_{p',t} \rangle) \otimes x'_t) && \text{by (2)} \\
&= H(F(K', M_{p'} + \langle x_{p',t} \rangle) \otimes F(K, M_p + \langle x_{p,t} \rangle) \otimes \\
&\qquad F(K', M_{p'} + \langle x_{p',t} \rangle) \otimes x_t) \\
&= H(F(K, M_p + \langle x_{p,t} \rangle) \otimes x_t) \\
&= F(K, M_p + \langle x_{p,t}\ ,\ x_t \rangle) && \text{by (2),}
\end{aligned}$$

where $M_p + \langle x_{p,t}, x_t \rangle$ is the list modulator for node t at its original location.

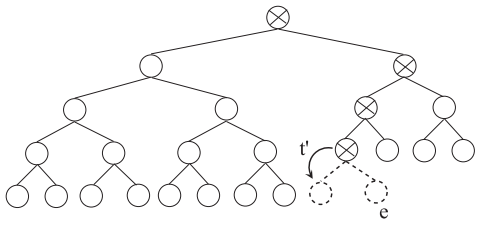

Figure 4. Balancing a new key e to the tree. The shape of the original tree does not have the dotted links and dotted nodes. The dotted links are created after the insertion.

The communication complexity of the balancing algorithm is $O(\log n)$, including $P(t)$ of size $O(\log n)$ from the server to the client and $O(1)$ modulators from the client to the server. Eq. (8) and (9) require $O(\log n)$ hashes. Hence, the time complexity is also $O(\log n)$.

E. Access, Modification, and Insertion

Although the goal of this paper is to support assured deletion, a practical system should also allow access, modification and insertion of outsourced data. For the purpose of completeness, we discuss these issues below.

To access a data item, the client makes a request to the server with appropriate indexing information such as a record ID or byte offset. The server identifies the encrypted item and the corresponding leaf node k in the modulation tree.[3] From the path $P(k)$, the server extracts a modulator list M_k. It sends the ciphertext and M_k to the client, which computes a data key $k = F(K, M_k)$, and uses the key to decrypt the ciphertext into $mH(m)$. The client computes the hash of m and compares with $H(m)$ from the ciphertext. They will match only if the key is correct.

To modify a data item, the client first fetches the item from the server using the access procedure above. It modifies the item, re-encrypts it using the same data key, and sends the ciphertext back to the server.

To insert a data item m', the client makes a request to the server for inserting a new leaf in the modulation tree. Let t' be the location in the tree where the insertion happens. For a full binary tree, the server sets t' to be the first leaf node at the last level; if there are leaves at the last two levels, the server sets t' to be the first leaf at the second-to-last level. See Figure 4. The server sends the client the path $P(t')$ from the root to node t'. Let $M_{t'}^-$ be $M_{t'}$ without the last modulator $x_{t'}$.

The client replaces node t' with a new internal node p, and sets t' and a new leaf e as the children of p. It assigns random modulators to leaf e and links (p, t') and (p, e). It reassigns a new leaf modulator to node t' as follows:

$$x'_{t'} = F(K, M_{t'}^-) \otimes F(K, M_{t'}^- + \langle x_{p,t'} \rangle) \otimes x_{t'}.$$

Following the same method as used in (IV-D), it can be shown that the data key encoded by node t' is unchanged with the new leaf modulator (after the insertion of p). Next, the client computes the data key encoded by the new leaf e, i.e., $F(K, M_{t'}^- + \langle x_{p,e}\ ,\ x_e \rangle)$. It then uses the key to encrypt m'. The client sends the following information to the server: the encrypted new item, the modulators for (p, t'), (p, e), t', and e, as well as the location (such as byte offset) in the file where the ciphertext should go. The server updates the modulation tree, stores the ciphertext, and keeps the mapping between node e and the ciphertext.

The communication/time complexity is $O(\log n)$ for access, modification, and insertion.

[3]Again we assume the correct return of requested item is enforced by another branch of research [1], [2], [3], [4], which ensures integrity in data storage and access.

V. Managing Master Keys for Large File Systems

Even though only one master key is needed for each file, the number of files in a large file system can be enormous, which means the number of master keys to be maintained by the client may still represent a problem. One solution is to outsource both data keys and master keys to the cloud through two levels of modulation trees. Each file has a master key and a modulation tree. If we treat all master keys as the data items of a meta file, we can introduce a so-called meta modulation tree and use a higher-level control key as the master key of the meta file. To access a file, the client will first use the control key to access the meta modulation tree in order to retrieve the master key for the file, and then use the master key to access the modulation tree of that file. Deleting a master key from the meta modulation tree will make the entire file unrecoverable. Deleting a data item of a file involves two steps: first deleting the data key from the modulation tree of the file, and then modifying the master key of the file in the meta modulation tree. Instead of storing just a single control key, the client may also divide the master keys of all files into groups based on the directory structure or file types, and use a separate control key and a corresponding meta modulation tree for each group.

If a client has many users sharing the same file system, the master keys (or control keys) may be stored in a shared local secure storage for users to access. Alternatively, the client may designate a local proxy server to manage these keys. When a user wants to operate on data, its request is redirected to the proxy, which will act on the user's behalf to access or update the data before forwarding the data to the user.

VI. Experimental Results

We use experiments to evaluate the practicality of our solution and compare it with other solutions in terms of client storage overhead, communication overhead, and computation overhead. The communication overhead consists of all information that the client receives and sends for an operation, but the overhead does not include the data item itself if the operation is to access (fetch) a data item. The computation overhead measures the time that the client spends on a certain operation; note that most computation is done at the client side in our solution (as well as in other solutions). The end-to-end access delay is not measured because it is not unique to our approach but a consequence of using remote cloud storage.

A. Implementation

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) [9] in the modulated hash chain. SHA-1 produces a 160-bit message digest. Each modulator is also 160-bit long. We choose Advanced Encryption Standard (AES) [10] to encrypt each data item. AES has a key size of 128, 192, or 256 bits. In our implementation, we use 128-bit keys, taken from the output of the key modulation function.

Table I
COMPLEXITY COMPARISON, INCLUDING CLIENT STORAGE COMPLEXITY, COMMUNICATION COMPLEXITY FOR DELETION, AND COMPUTATION COMPLEXITY FOR DELETION, WHERE THE LATTER TWO ARE COMBINED IN THE SAME ROW BECAUSE THEY HAVE THE SAME BIG-O VALUES.

complexities \ solutions	master-key	individual-key	our work
client storage	$O(1)$	$O(n)$	$O(1)$
communication/computation	$O(n)$	$O(1)$	$O(\log n)$

Table II
EXPERIMENTAL COMPARISON, INCLUDING CLIENT STORAGE OVERHEAD, COMMUNICATION OVERHEAD FOR DELETION, AND COMPUTATION OVERHEAD FOR DELETION.

overhead \ solutions	master-key	individual-key	our work
client storage	16 Bytes	1.53 MB	16 Bytes
communication overhead	391 MB	0	1.61 KB
computation overhead	5.5 minutes	almost 0	0.24 ms

B. Performance Comparison

We compare our two-party solution with the master-key solution and the individual-key solution (Section III), which do not require a third party, either. The difference between our solution and those requiring third parties [5], [5], [7] is more fundamental than performance alone, as we have discussed their security problem under the threat model of this paper in the introduction. Moreover, they use one key to protect multiple files, and therefore do not support efficient fine-grained deletion. If they are used to delete individual data items, their performance will be similar to the master-key solution, assuming each of their keys protects one file.

1) Complexity Comparison: Table I gives the complexity comparison for one file of n data items. The master-key solution has $O(1)$ client storage complexity but $O(n)$ communication/computation complexities. The individual-key solution has $O(1)$ communication/computation complexities but $O(n)$ client storage complexity. In comparison, our approach has both low client storage complexity of $O(1)$ and low communication/computation complexities of $O(\log n)$.

If we consider a file system of m files. The client storage complexity of the master-key solution will be $O(m)$, but that of our solution will still be $O(1)$.

2) Experimental Comparison: We further compare the deletion overhead of the three solutions through real experiment. The results are shown in Table II. Suppose the size of each data item is 4KB (typical sector size of newer hard disks) and the total number of data item is 10^5. The master-key solution and our solution only need to store a master key of 16 bytes. But the individual-key solution has to store 10^5 keys of 1.53MB in total; note that 1.53MB is the storage overhead for one file (in order to support fine-grained deletion), and the file system may have numerous files.

The master-key solution has a communication overhead of 391MB and a computation overhead of 5.5 minutes in order to retrieve and re-encrypt the entire file. In comparison, our solution has a communication overhead of 1.61KB and a computation overhead of just 0.24 ms.

C. Communication Overhead

Next, we validate the practicality of our modulation tree by measuring the scalability of our solution in communication overhead from small file size (10 data items) to large size (10^7 items). The results are presented in Figure 5, where the x-axis shows the total number of data items in logarithmic scale, and the y-axis shows the average communication overhead in KB. To measure the communication overhead of deletion or access, we perform the operation on each data item once and take the average overhead. Insertion into the modulation tree always happens at the same location in the tree, and averaging is not necessary.

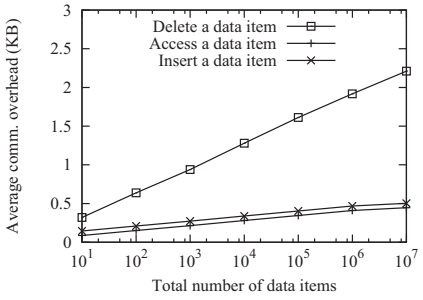

Figure 5. Communication overhead for deleting, inserting, or accessing a data item. It includes all information that the client sends or receives for an operation.

The communication overhead for deletion is modest even for very large files of 10^7 items, and the overhead for access or insertion is much lower. Clearly, all measured communication overheads increase logarithmically with respect to the number of data items, demonstrating good scalability.

D. Computation Overhead

We further validate the practicality of our modulation tree by measuring the scalability of our solution in computation overhead from small file size (10 data items) to large size (10^7 items). The results are presented in Figure 6, where the x-axis shows the number of data items in logarithmic scale, and the y-axis shows the average client computational time in ms. The computation overhead for deletion is small, under 0.3ms for very large files of 10^7 items. The overhead for access

and insertion is again much smaller. All measured computation overheads increase logarithmically with respect to the number of data items.

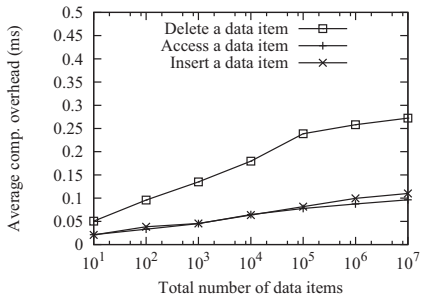

Figure 6. Client computation overhead for deleting, accessing, or inserting a data item.

E. Whole File Access Overhead

It is a common operation for a client to fetch a whole file from a remote file system. With our solution, the client will take the extra steps of fetching the entire modulation tree and computing all data keys from the tree, which causes communication/computation overhead. Fetching the file itself and decrypting the file are normal, necessary expenses that have to be taken in any encrypted cloud-based file system, and therefore do not count as overhead due to the design of our solution.

We define the *communication overhead ratio* as the communication overhead divided by the size of the file, and the *computation overhead ratio* as the time of computing all data keys from the modulation tree divided by the time of decrypting the file. The size of each data item is 4KB. The experimental results are shown in Table III. Both the communication overhead ratio and the computation overhead ratio are largely insensitive to the file size. The former is less than 1%, and the latter is less than 0.3%.

Table III
WHOLE FILE ACCESS OVERHEAD

no. of data items	comm. ratio	comp. ratio
10	0.0093	0.0004
10^2	0.0097	0.0024
10^3	0.0098	0.0025
10^4	0.0098	0.0025
10^5	0.0098	0.0027
10^6	0.0098	0.0027
10^7	0.0098	0.0027

VII. RELATED WORK

Most related is Tang's policy-based system named FADE [7], which is designed for assured file deletion in a cloud storage system. Their approach is to associate each policy with a control key maintained by a third party, and use the control key to protect the data keys that encrypt files assigned with that policy. Deleting a control key will make all files with the corresponding policy inaccessible. The key service from the third party is needed by all data operations. If the third party is compromised, deletion will no longer be secure. Rahumed $et\,al.$ design a system called FadeVersion [11], which combines policy-based deletion with version control.

Perlman proposes the first solution for timed deletion, where all messages to be expired at the same future time will be protected by the same key stored at a third party called the Ephemerizer [5]. The goal is to ensure the privacy of past messages transferred between two parties, such as emails or SMS. As these messages may be cached on the servers, one may want the assurance that they will be inaccessible after an expiration time. Together with followup work [12], [5], [13], [14], [15], they form the so-called Ephemerizer family of solutions. All Ephemerizer solutions require third parties to provide the key service for all data operations.

Instead of relying on centralized third parties to manage the keys, Geambasu $et\,al.$ design a decentralized approach called Vanish [16], where the sender first encrypts its message and then distributes key shares [17] to nodes through a distributed hash table (DHT) [18]. The DHT evolves dynamically with new nodes joining and old nodes departing from a P2P network, and the message will become inaccessible if enough key shares are removed. The sender encapsulates the ciphertext of the message and necessary information for locating key shares into a Vanish Data Object (VDO) and sends the VDO to the receiver for data access. Vanish is vulnerable to Sybil attacks [19]. Zeng $et\,al.$ propose SafeVanish [20] and Castelluccia $et\,al.$ design EphPub [21] to fix the security problems of Vanish.

Vanish and its followups were 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.

Also related is the work on secure local deletion such as [22]. The proposed solution shares superficial similarity with key revocation in broadcast encryption such as [23], where a revoked user cannot decrypt future messages but is not prevented from decrypting past messages.

VIII. CONCLUSION

This paper presents a two-party fine-grained solution for protecting the privacy of deleted data that has previously been outsourced by clients to the cloud. The main challenge is how to avoid burdening clients with a large number of keys, yet allowing them to perform deletion on any data item in any file without causing significant overhead. Our solution is based on a novel key modulation function which is derived from modulated hash chains and a modulation tree. We prove its correctness and security, and implement it on the Amazon EC2 system. The evaluation results show that it carries small overhead for remote deletion, insertion and file access. The proposed solution, to the best of our knowledge, is the first

two-party scheme that protects the confidentiality of the "dead" (deleted) data in a cloud system.

IX. 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] H. Shacham and B. Waters, "Compact Proofs of Retrievability," *Proc. of ASIACRYPT*, 2008.

[2] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, "Dynamic Provable Data Possession," *Proc. of CCS*, 2009.

[3] 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.

[4] 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.

[5] R. Perlman, "File System Design with Assured Delete," *Proc. of SISW*, 2005.

[6] ——, "The Ephemerizer: Making Data Disappear," *Information System Security*, 2005.

[7] Y. Tang, P. Lee, J. Lui, and R. Perlman, "FADE: Secure Overlay Cloud Storage with File Assured Deletion," *Proc. of SecureComm*, 2010.

[8] L. Lamport, "Password Authentication with Insecure Communication," *Communications of the ACM*, 1981.

[9] J. Burrows, "Secure Hash Standard," Tech. Rep., 1995.

[10] J. Daemen and V. Rijmen, "The Design of Rijndael: AES–the Advanced Encryption Standard," *Springer-Verlag, ISBN 3-540-42580-2, New York*, 2002.

[11] 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.

[12] B. Crispo, M. Dashti, S. Nair, and A. Tanenbaum, "A Hybrid PKI-IBC Based Ephemerizer System," *Proc. of EuroPKI*, 2009.

[13] C. Arora and M. Turuani, "Validating Integrity for the Ephemerizers Protocol with CL-Atse," *Formal to Practical Security*, 2009.

[14] Q. Tang, "From Ephemerizer to Timed-Ephemerizer: Achieve Assured Lifecycle Enforcement for Sensitive Data," *Technical Report TR-CTIT-10-01*, 2010.

[15] C. Arora and M. Turuani, "Adding Integrity to the Ephemerizer's Protocol," *Proc. of AVoCS*, 2006.

[16] R. Geambasu, T. Kohno, A. Levy, and H. Levy, "Vanish: Increasing Data Privacy with Self-destructing Data," *Proc. of USENIX*, 2009.

[17] A. Shamir, "How to share a secret," *Communications of the ACM*, 1979.

[18] 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.

[19] 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.

[20] L. Zeng, Z. Shi, S. Xu, and D. Feng, "SafeVanish: An Improved Data Self-Destruction for Protecting Data Privacy," *Proc. of CloudCom*, 2010.

[21] C. Castelluccia, E. D. Cristofaro, A. Francillon, and M. Kaafar, "EphPub: Toward Robust Ephemeral Publishing," *Proc. of ICNP*, 2011.

[22] J. Reardon, S. Capkun, and D. Basin, "Efficient Secure Deletion for Flash Memory," *Proc. of USENIX Security Symposium*, 2012.

[23] A. Fiat and M. Naor, "Broadcast Encryption," *Proc. of CRYPTO*, 1993.

Appendix. Proofs

Proof of LEMMA 1: Let $S_k^{(l-i)}$, $0 \le i \le l$, be a suffix of M_k, containing the last $l-i$ modulators in M_k. In (1), if we treat $H(K \otimes x_1)$ as the new key, it becomes

$$F(K, M_k) = F(H(K \otimes x_1), S_k^{(l-1)}). \tag{10}$$

Next we prove by induction that

$$F(K, M_k) = F(F(K, M_k^{(i)}), S_k^{(l-i)}),\ 0 \le i \le l. \tag{11}$$

(11) holds when $i = 0$ because $M_k^{(0)} = \emptyset$, $S_k^{(l)} = M_k$, and $F(K, \emptyset) = K$ by definition (2). The inductive assumption is that (11) holds for a certain value i. We prove the case of $i+1$ as follows:

$$\begin{aligned}
&F(F(K, M_k^{(i+1)}), S_k^{(l-i-1)}) \\
&= F(H(F(K, M_k^{(i)}) \otimes x_{i+1}), S_k^{(l-i-1)}) && \text{by (2)} \\
&= F(F(K, M_k^{(i)}), S_k^{(l-i)}) && \text{by (10)} \\
&= F(K, M_k) && \text{by inductive assumption}
\end{aligned}$$

Now according to (11), we have

$$F(K, M_k) = F(F(K, M_k^{(i)}), S_k^{l-i}),$$
$$F(K', M_k | x_i \to x_i') = F(F(K', M_k^{(i)} | x_i \to x_i'), S_k^{l-i}).$$

Hence, in order to prove (4), it suffices to prove

$$F(K, M_k^{(i)}) = F(K', M_k^{(i)} | x_i \to x_i').$$

By (2), we have $F(K, M_k^{(i)}) = H(F(K, M_k^{(i-1)}) \otimes x_i)$, and

$$\begin{aligned}
&F(K', M_k^{(i)} | x_i \to x_i') \\
&= H(F(K', M_k^{(i-1)}) \otimes x_i') \\
&= H(F(K', M_k^{(i-1)}) \otimes x_i \otimes F(K, M_k^{(i-1)}) \otimes F(K', M_k^{(i-1)})) \\
&= H(F(K, M_k^{(i-1)}) \otimes x_i) = F(K, M_k^{(i)}).
\end{aligned}$$

This completes the proof. □

Proof of THEOREM 1: Consider an arbitrary leaf node k' (other than k). The path $P(k')$ from the root to node k' must

pass a node c in the cut C. Node c divides $P(k')$ into a sub-path from the root to c and a sub-path from c to leaf k', which correspond to a prefix M_c of the modulator list $M_{k'}$ and a suffix, respectively. Hence, $M_c = M_{k'}^{(i-1)}$ for a certain value of i, where $1 < i \leq l$ and l is the number of modulators in $M_{k'}$. The suffix, denoted as $S_{k'}^{(l-i+1)}$, contains the last $(l-i+1)$ modulators in $M_{k'}$, including the leaf modulator of node k'. Hence, Eq. (5) can rewritten as

$$\delta(c) = F(K, M_{k'}^{(i-1)}) \otimes F(K', M_{k'}^{(i-1)}). \tag{12}$$

When the server receives $\delta(c)$, it performs either (6), which updates the modulator on the child link belonging to $P(k')$, or (7), which updates the leaf modulator if c is a leaf node. In either case, the updated modulator belongs to $M_{k'}$, and it is right after the prefix $M_{k'}^{(i-1)}$. Hence, it is also denoted as x_i. Based on (6), (7) and (12), the new value of this modulator is

$$x_i' = x_i \otimes F(K, M_{k'}^{(i-1)}) \otimes F(K', M_{k'}^{(i-1)}).$$

No other modulator in $M_{k'}$ has been changed. By Lemma 1, we have

$$F(K, M_{k'}) = F(K', M_{k'} | x_i \rightarrow x_i').$$

The data key k' remains unchanged. □

Proof of THEOREM 2: There are two cases: i) the server sends correct $MT(k)$ to the client, and ii) the server sends incorrect $MT(k)$. According to the threat model in Section II-C, we assume that an attacker may have compromised the server before deletion, allowing it to send incorrect information to the client, and that the attacker may also compromise the server after deletion, allowing it to learn the new master key T' (but not the old one T). Let l be the size of M_k.

Case i): The modulator adjustment algorithm does not change any modulator in $MT(k)$. Since the path $P(k)$ from the root to node k is entirely in $MT(k)$, the algorithm does not change any modulator in M_k, which is extracted from $P(k)$.

We prove $F(K, M_k) \neq F(K', M_k)$ w.h.p by contradiction. Suppose $F(K, M_k) = F(K', M_k)$. By (2) we have

$$H(F(K, M_k^{(l-1)}) \otimes x_l) = H(F(K', M_k^{(l-1)}) \otimes x_l),$$

which means $F(K, M_k^{(l-1)}) = F(K', M_k^{(l-1)})$ w.h.p; otherwise, we would have found two different hash inputs that produce the same output. Recursively applying the same token, we have $F(K, M_k^{(1)}) = F(K', M_k^{(1)})$, which is $H(K \otimes x_1) = H(K' \otimes x_1)$. We have found two different inputs, $K \otimes x_1$ and $K' \otimes x_1$, producing the same hash output, contradicting with the theorem assumption. Therefore, it must be true that $F(K, M_k) \neq F(K', M_k)$ w.h.p. Note that even if $F(K, M_k) = F(K', M_k)$ occurs in practice (whatever low probability it is), the client can simply pick a different K' such that $F(K, M_k) \neq F(K', M_k)$.

Because $F(K, M_k) \neq F(K', M_k)$, knowing K' will not help an attacker figure out $k = F(K, M_k)$ after K is permanently deleted and thus unknown. This is because if the attacker had a polynomial way to hash K' and some modulators into key k, it would break the assumption that it is polynomially infeasible to find a hash input for a specific output.

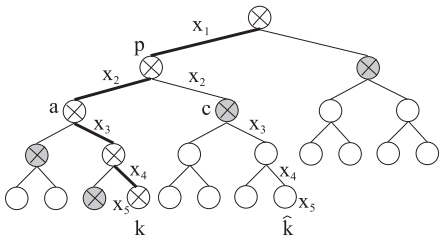

Figure 7. $MT^*(k)$ consists of nodes with cross inside. It contains $P(k)$ shown by bold lines and C shown by shaded nodes.

Case ii): Suppose an attacker controls the sever to send incorrect $MT(k)$. To begin with, the server can send $MT(k')$ for a different leaf node k', and try to trick the client into deleting k', while keeping other keys (including k) unchanged under a new master key K'. After K' is revealed, the attacker would be able to recover k. However, according to the modulator adjustment algorithm, after the client receives $MT(k')$, it computes the data key $k' = F(K, M_{k'})$, which will not be able to correctly decrypt the ciphertext $\{mH(m)\}_k$. Consequently, the client will reject $MT(k')$. Let $MT^*(k)$ be what the client actually receives. To avoid being rejected, $MT^*(k)$ must contain the correct path $P(k)$ from the root to the leaf, carrying correct M_k to produce the correct key k in order to correctly decrypt $\{mH(m)\}_k$.

Since $MT^*(k)$ consists of $P(k)$ and the cut C, if $P(k)$ must be correct, it will leave C the only place that the server can manipulate. As illustrated in Figure 7, the server can replace the modulators on the path $P(\hat{k})$ to a different leaf node $\hat{k}$ with those of M_k. By doing so, the key encoded by node $\hat{k}$ becomes the same as the key by node k. After k is deleted, if the key encoded by node $\hat{k}$ is kept unchanged, the deleted key is recoverable. In general, no matter how the server changes the modulators outside of $P(k)$, as long as $F(K, M_k) = F(K, M_{\hat{k}})$, we must have $M_k = M_{\hat{k}}$ because otherwise we would have two different sets of hash inputs that produce the same output.

Suppose path $P(\hat{k})$ intersects with path $P(k)$ at node p. See Figure 7. Let a be the child node of p on path $P(k)$, and c be the child node on path $P(\hat{k})$. Node c is a sibling of node a, and thus it belongs to the cut C. It follows that both link (p, a) and link (p, c) belong to $MT^*(k)$. Because $M_k = M_{\hat{k}}$, the modulators on these two links must be the same, which violates the requirement that all modulators should be different, and hence the client will reject $MT^*(k)$.

Combining the above two cases, if the server sends correct $MT(k)$, the deleted key k will be unrecoverable; if the server sends incorrect $MT(k)$ to make k recoverable, the client will reject the received $MT^*(k)$, which prevents the modulator adjustment algorithm from being executed. Hence, the theorem is proved. □