SECURE CLOUD COMPUTING: DATA INTEGRITY, ASSURED DELETION, AND MEASUREMENT-BASED ANOMALY DETECTION

By

ZHEN MO

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2015

To my parents and my wife. Without you I cannot reach so far. I love you!

ACKNOWLEDGMENTS

It has been a long journey for me to complete my dissertation and the subsequent Ph.D. After five years of hard work, finally I reach here. During this long journal, I received a lot of help from my teachers, friends and family. Without them, I don't know whether I can achieve it.

The first and the most important person I would like to thank is my advisor, Dr. Shigang Chen. Thank you so much for everything you've done to help me with my research. You've always been patient and willing to assist me with any problem I encountered. I sincerely appreciate all the work that you have done to make the Ph.D. process painless.

I want to give my special gratitude to my Ph.D. committee members. They are Dr. Jose Fortes, Dr. Sartaj Sahni, Dr. Ye Xia and Dr. Yuguang Fang. Thank you for your advices and support during my study at University of Florida. I also want to thank the researchers and colleagues in my research group. Their names are Ming Zhang, Tao Li, Yan Qiao, Wen Luo, Yian Zhou, Min Chen, Xi Tao, You Zhou, Qingjun Xiao, Zhiping Cai, and Liping Zhang.

Finally, I would like give my greatest thanks to my parents. Thank you for investing so much in raising me. Thank you for bringing me everything. Thank you for all the many, many happy memories. I also want to thank my wife, Jenny. Thank you for your love and support. I am really grateful and lucky to have you as my life partner.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

SECURE CLOUD COMPUTING: DATA INTEGRITY, ASSURED DELETION, AND MEASUREMENT-BASED ANOMALY DETECTION

By

Zhen Mo

May 2015

Chair: Shigang Chen
Major: Computer Engineering

Cloud computing brings security concerns. In this work, we focus on the following three security concerns in the cloud computing systems: the assured deletion problem, the data integrity problem and the measurement based anomaly detection problem.

The first concern is called the assured deletion problem which 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? In this work, we propose two methods to achieve two party fine-grained assured deletion in cloud computing systems.

Another major security concern in cloud computing is about the integrity of user data. By outsourcing data to an off-site storage system and removing local copies, cloud users are relieved from the burden of storage, but in the meantime lose physical control of their data. Can we trust the cloud service providers? Obviously we can't. Meanwhile, there is another complementary problem: When a client claims that the server has lost their data, how can we be sure that the client is correct and honest about the loss? It is possible that the client's meta data is corrupted or the client is lying in order to blackmail the server. In this work, we propose a non-repudiable data possession verification solution that protects both the client and the server.

The third security concern in cloud computing is how to detect network security threats. As the Internet moves into the era of big network data, it presents both opportunities and technical challenges for traffic measurement functions, such as flow cardinality estimation. We propose a new solution of virtual maximum likelihood sketches for cardinality estimation. It has four technical contributions: virtual sketches, virtual sketch vectors, a maximum likelihood method for cardinality estimation based on per-flow virtual sketch vectors, and a method to achieve differentiated estimation accuracy among multiple subsets of flows with different priorities.

CHAPTER 1
INTRODUCTION

1.1 Motivation

Cloud computing is a type of online service model. Instead of providing a product, cloud computing provides services in a pay-as-you-go manner. Developers and users do not need to know about the physical location and configuration of the system that delivers the services. They can easily and quickly adjust the resources to their needs. This elasticity of resources, without any pre-investment, attracts more and more people join the cloud computing. Although envisioned as a promising service model, cloud computing also brings security concerns. In this work, we focus on the following three security problems in the cloud computing: the assured deletion problem, the data integrity problem and the measurement based anomaly detection problem.

Assured Deletion Problem: In cloud storage systems, researchers usually try to verify the *existence* and *accessibility* of the data in its entirety on the cloud servers. On the other hand, we 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. In this work, we present two solutions to help the clients efficiently delete the data permanently.

Data Integrity Problem: Another major security concern in cloud computing is about the integrity of user data. By outsourcing data to an off-site storage system and removing local copies, cloud users are relieved from the burden of storage, but in the meantime lose physical control of their data. Can we trust the cloud service providers? Even if we assume that service providers will not deliberately hinder clients' correct access to their own data (after all this is the providers' lifeline business), involuntary security breaches may occur. For example, a provider may lose user data due to hardware failure, human mistakes or external intrusion. Not having the data, the provider may attempt to hide such an incident in order to save reputation. Meanwhile, there is another complementary problem: When a client claims that the server has lost their data, how can we be sure that the client is correct and honest about the loss? It is possible that the client's meta data is corrupted or the client is lying in order to blackmail the server. In addition, most previous work relies on sequential indices. However, the indices bring significant overhead to bind an index to each block. We propose to replace sequential indices with much flexible non-sequential *coordinates*. The binding

of coordinates to data blocks is performed through a *Coordinate* Merkle Hash Tree (CMHT). Based on CMHT, we can improve both the average and the worst-case update overhead by simplifying the updating algorithm.

Measurement-based Anomaly Detection Problem: The third security concern in cloud computing is how to detect network security threats. As the Internet moves into the era of big network data, it presents both opportunities and technical challenges for traffic measurement functions, such as flow cardinality estimation, which is to estimate the number of distinct elements in each flow. Cardinality estimation has important applications in intrusion detection, resource management, billing and capacity planning, as well as big data analytics. Due to the practical need of processing network data in high volume and high speed, past research has strived to reduce the memory overhead for cardinality estimation on a large number of flows. One important thread of research in this area is based on sketches. The representative work includes the FM sketches [29], the LogLog sketches [24], and the HyperLogLog sketches [28]. Each sketch requires multiple bits and many sketches are needed for each flow, which results in significant memory overhead. This work proposes a new method of virtual maximum likelihood sketches to reduce memory consumption. First, we design *virtual sketches* that use no more than two bits per sketch on average. Second, we design *virtual sketch vectors* that consider all flows together. Based on these new constructs, we design a flow cardinality solution with an online operation module and an offline estimation module. We also consider the problem of differentiated estimation that gives flows of high priorities better precision in their cardinality estimations. We implement the new solution and perform experiments to evaluate its performance based on real traffic traces.

1.2 Overview of The Dissertation

In this work, we first introduce the cloud computing system. Cloud computing has been gaining firm traction in the marketplace as major high-tech companies rush

to offer cloud services, such as Amazon, Google cloud storage, and Apple iCloud. There is obvious benefit of outsourcing data to the cloud and utilizing services in the cloud: Its professional data storage, backup, and processing services provide a high level of efficiency, reliability and economy-of-scale that individual users and companies can hardly match at a comparable cost. With a pay-as-you-go model, the users save money by paying only for the resources they actually use, while offloading the maintenance burden. They can easily adjust resources to their needs in real time. This elasticity in resource provision, without any pre-investment, are attracting more and more entrepreneurs and business companies to replace their IT infrastructure with a cloud-based one. Although envisioned as a promising service model, cloud storage also brings security concerns.

In Chapter 2, we study the assured deletion problem. 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) [48, 48, 56]. 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 [48, 48], which belong to the same class and are expected to be deleted together [48], or which share the same access policy [56]. 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.

If we adapt the prior work [48, 48, 56] 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. So in this work, we presents two solutions 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.

The first solution is based on key modulation function. The client stores a master key and the cloud server stores a set of modulators in a data structure named key modulation tree. We develop a novel key modulation function that allows the client to delete each individual data item without having to re-encrypt the rest of the file even though the master key has been changed. In the first solution, we view the integrity issues in data storage and data access as complementary problems that have been solved in [5, 26, 52, 58], which can provide proof of data possession in the cloud and allow the client to correctly access each data item.

Next, we present a second solution based on a novel multi-layered key 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.

In Chapter 3, we focus on the data integrity problem in cloud computing. This problem stems from the fact that the clients lost the physical control of their data. In order to solve this problem, many solutions are proposed [5, 6, 11, 25, 34, 52, 58]. However, most of the previous works [5, 6, 11, 34, 52] can only apply to static data files. Though Wang *et al.* propose a dynamic version of PoR model in [58] and Erway *et al.* present a dynamic PDP model in [25], unfortunately, the performance of their solutions are not tightly bounded. Accordingly, we first design a new Cloud Merkle B+ Tree (CMBT) to assist the verification procedure, whose *worst-case* computation/communication overhead for inserting/deleting/updating a data block is $O(\log n)$, comparing with $O(n)$ worst-case overhead in [25, 58].

Then, we find that most current designs involve sequential data indices. The user data is divided into a sequence of blocks, which are sequentially indexed from 1, 2, ..., to n. Index numbers are not needed for data access; data blocks are normally accessed based on filenames and byte offsets in the files. The indices are used in the data-possession verification process. For instance, each time the user will randomly select a subset of indices, and then it will challenge the cloud server to present a proof, showing that the server possesses the data blocks with the selected indices. To prevent the cloud server from cheating, the indices are cryptographically bound to the data blocks either explicitly through a homomorphic data signature [5, 52] or implicitly through an authentication data structure [25, 58]. Those signatures are used to generate the integrity proof. The problem of explicitly binding an index number in each signature is that it effectively prohibits dynamic update of user data. For example, if the user deletes a block with index i, the indices of all subsequent blocks are reduced by one and thus their signatures will have to be recomputed. To address this issue, other solutions implicitly bind indices to blocks through a remotely-stored cryptographic data structure,

e.g., skip list [25] or Merkle tree [58], where the index of a block is essentially its position among all leaves in the postorder traversal of the skip list or Merkle tree. Accordingly, in order to further bind performance of inserting/deleting and updating a data block in our solution, we present another new data structure named Coordinate Merkle Hash Tree (CMHT). Different from the existing designs, CMHT is constructed based on coordinate instead of indices. With the new design, we optimize the communication and computational overhead.

Next, we expands data integrity protection by covering an important complementary problem: When a user claims a data loss, how can we be sure that the user is correct and honest about the loss? If the meta data stored by the users is corrupted or if a user tries to blackmail the cloud by lying about data loss, how can the cloud prove its innocence? In order to solve this problem, we design a new meta data to realize the non-repudiation property that allows the cloud and the users to supervise each other, so that users are able to detect whether the cloud has lost their data and the cloud is able to fend off the false claims of data loss from users. Comparing with previous work, our solution can protect the rights of both sides.

In Chapter 4, we pay attention to the cardinality estimation in cloud computing. There are practical needs for reducing per-flow memory overhead in cardinality estimation. If the cardinality estimation function is implemented on a network processor chip, because the on-chip memory is typically small and the number of flows in modern networks can be very large, we will have to minimize per-flow overhead in order to accommodate more flows. In the previous application example of purchase associations, if there are hundreds of thousands of different products, the number of possible purchase associations (flows) can in tens of billions. For such a large number of flows, memory overhead may become a problem.

we proposes a new cardinality estimation method, called virtual maximum likelihood sketches, to reduce memory consumption by cardinality estimation on a large number of

flows. It embodies two ideas. The first idea is called *virtual sketches*, which use no more than two bits per sketch on average, while retaining the functional equivalence to an FM sketch. The second idea is called *virtual sketch vectors*, which combine the sketches of all flows into a mixed common pool. Together, these two ideas can drastically reduce the overall memory overhead. Based on virtual sketches and virtual vectors, we design a cardinality estimation solution with an online operation module and an offline estimation module. For a system where some flows are more important than others, we investigate the problem of differentiating estimation accuracy, depending on the priorities of the flows. We implement the new solution and perform experiments based on real traffic traces. The results demonstrate that the new solution can work reasonably well in very tight space and has the ability of differentiating flows of different priorities.

CHAPTER 2
ASSURED DELETION PROBLEM IN CLOUD COMPUTING

2.1 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.

2.2 Related Work

We discuss the related work in greater details. There are two categories. The first one is about file assured deletion. The second one is about hierarchical key tree.

2.2.1 File Assured Deletion

One approach to make data stored on remote servers disappear is called assured file deletion. It is first designed to ensure the privacy of the 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. Perlman proposes the first approach for assured file deletion in [48]. Together with the followup works [3, 4, 20, 47, 55], they form a so-called the Ephemerizer family of solutions. These solutions first encrypt each message with a data key. Then the

data keys whose expiration times are the same will be encrypted by a public key called *ephemeral public key*, which is managed by one or more trusted third parties, named "the ephemerizers." As the ephemeral private keys are only known to the ephemerizer, deleting one ephemeral private key will make the data keys encrypted by the corresponding ephemeral public key unrecoverable. So the messages encrypted by the data keys will become inaccessible. However, these solutions require trusted third parties to perform their deletion operations; if the third parties are down, certain operations will not be able to perform. Risks arise when the third parties are internally or externally compromised.

Following the Ephemerizer family of solutions, instead of relying on centralized third parties to manage the keys, Geambasu *et al.* design a decentralized approach called the Vanish [31]. In their solution, the sender first encrypts the data D with a random key K to obtain a ciphertext C. Then the sender uses threshold secret sharing [53] to split the key K into N shares $k_1, k_2, \ldots, k_N$. The parameter $threshold$ determines how many of the N shares are required to reconstruct the key. For example, if $N = 10$ and the $threshold$ is 5, then anybody can reconstruct the key after acquiring any 5 of the 10 shares. Next, the sender picks at random an access key L to generate N node indices $l_1, \ldots, l_N$, and store each share k_i on a node with index l_i in the distributed hash tables (DHTs) [54]. As the DHTs evolve dynamically with new nodes joining and old nodes leaving, each node in the DHTs has a lifetime. With disappearing of the nodes in DHTs, the keys will become "self-destructed" and accordingly the data will become unrecoverable. Finally the sender encapsulates L, C, N and $threshold$ into a Vanish Data Object (VDO) and sends the VDO to the receiver, protected by PGP or GPG. After receiving the VDO, the receiver can acquire the message as long as the VDO has not expired. However the Vanish is vulnerable to Sybil attacks [23]. It has been proved by Wolchok *et al.* [60] that attackers can continuously crawl the DHTs and save each

stored value before it expires. They can efficiently recover keys for more than 99% of the messages.

Researchers proposes different approaches to fix the security flaws of the Vanish. Castelluccia *et al.* design a solution named EphPub [13]. Their approach is built on the Domain Name System (DNS) and its caching mechanism. Instead of distributing the encryption keys into the DHTs, they stores each bit of the key into different DNS revolvers. As the cache entries have a fixed life time, the key information will be erased once the entry has expired. Compared with the Vanish, their approach is immune to Sybil attacks and easier to implement.

2.2.2 Key Management in Hierarchical Access Control

Akl and Taylor [2] investigate how to efficiently and securely entitle users with different rights to access classes organized in a hierarchical manner, where classes can be file folders and files in a file system. That is, if a user is entitled to access a certain class, the user obtains the access right to not only that class, but also its descendants in the hierarchy. After the seminal work of [2], there have been a large number of follow-up papers on key management in hierarchical access control [7, 9, 14–18, 22, 32, 51, 64]. We want to stress that the hierarchical key management in this work is designed for a different purpose: assured deletion, not hierarchical access control. With a different purpose, the requirements and the operations are also very different. Most notably, we use a key tree and they do not. The details of key operations and particularly the semantics of those operations share little common ground between our work and theirs.

Specially, Grolimund *et al.* [32] propose a cryptographic tree structure called Cryptree to realize hierarchical access control in file systems operating on untrusted storage. With the cryptographic tree, they can grant different users access rights to visit different files or folders without revealing the identities of other users. Although RERK and Cryptree are both tree like data structures, they are different in the following two aspects: First, as they have totally different design goals, they have different key

updating algorithms. For example, in Cryptree, when a key becomes dirty, they only need to replace it with a new key. But in RERK, modifying a key will make the changes from the leaf to the root. Second, Cryptree does not implement any re-balancing algorithm, so the worst case asymptotic complexity of lookup for a key will be $O(n)$. However, RERK is a self balanced red-black tree whose lookup and update complexity are tightly bounded by $O(\log n)$.

2.3 Straightforward Two-party Solutions

We first analyze two simple two-party solutions to give the motivation for our new approach of key modulation. We use $\{\mathbf{m}\}_{\mathrm{k}}$ as the notation for encrypting m with key k.

2.3.1 Master-Key Solution

Consider an arbitrary client file of n data items, denoted as $\{m_1, m_2, \ldots, m_n\}$. 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 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.

2.3.2 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 $\{k_1, k_2, \ldots, k_n\}$, 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.

2.4 Key Modulation Based Solution

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 2-1. The data keys are derived from the master key K and a set M of values called *modulators*. More

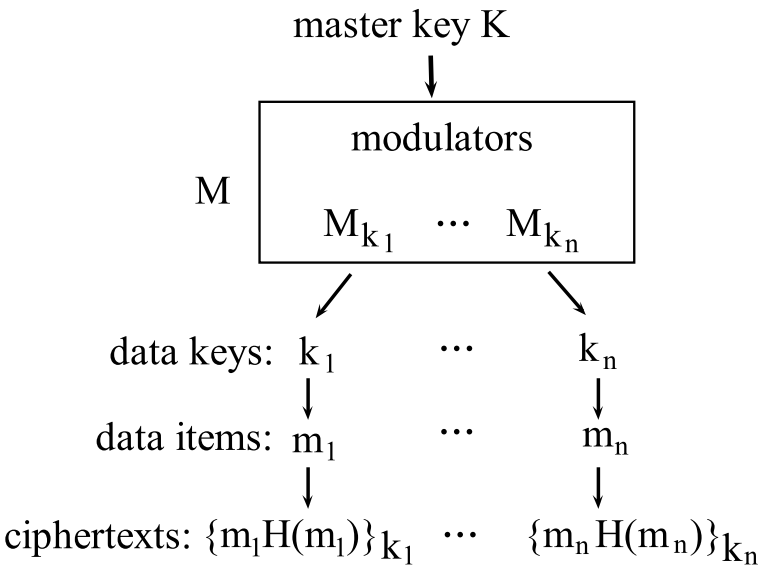

Figure 2-1. Key modulation

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.

2.4.1 Threat Model

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.)

2.4.2 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.

Modulated hash chain. We formulate the function $F(K, M_k)$ as a modulated hash chain. The classical hash chain has the following format [38]:

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

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

$$F(K, M_k) = H(...H(H(H(K \otimes x_1) \otimes x_2)... \otimes x_l), \tag{2–1}$$

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:

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

Let $S_k^{(l-i)}$, $0 \leq i \leq l$, be a suffix of M_k, containing the last $l - i$ modulators in M_k. In (2–1), if we treat $H(K \otimes x_1)$ as the new key, we have the following lemma:

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

Lemma 2. $F(K, M_k) = F(F(K, M_k^{(i)}), S_k^{(l-i)}),\ 0 \leq i \leq l$.

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

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

This completes the induction proof. □

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 3. *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*

$$x_i' = x_i \otimes F(K, M_k^{(i-1)}) \otimes F(K', M_k^{(i-1)}). \tag{2–3}$$

That is,

$$F(K, M_k) = F(K', M_k|x_i \to x_i'). \tag{2–4}$$

Proof. By Lemma 2, 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 (2–4), it suffices to prove

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

By (2–2), 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. □

Key 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 2-2. 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 2-2 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. 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 2.2).

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

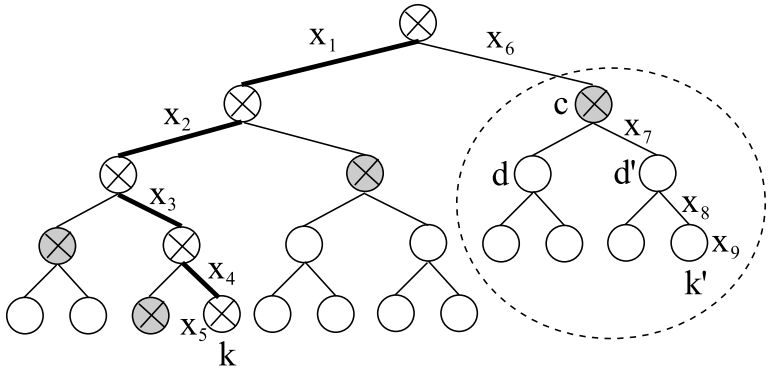

Figure 2-2. A modulation tree, where each leaf node is assigned a leaf modulator such as x_5, and each link is assigned a link modulator such as x_1 through x_4. The path $P(k)$ from the root to the leaf node k is drawn in bold lines; it is a graphical representation of $M_k = \langle x_1, x_2, x_3, x_4, x_5 \rangle$. 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.

security of this algorithm that ensures the deleted date key is unrecoverable even if the new master key is revealed in the future.

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

[1] Recall that we view the integrity issues in data storage and data access as complementary problems that have been solved in [5, 26, 52, 58], 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.

Figure 2-2, 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 2-2 for an example. The client computes

$$\delta(c) = F(K, M_c) \otimes F(K', M_c). \tag{2–5}$$

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:

$$\begin{aligned} x_{c,d} &:= x_{c,d} \otimes \delta(c) \\ x_{c,d'} &:= x_{c,d'} \otimes \delta(c), \end{aligned} \tag{2–6}$$

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:

$$x_c := x_c \otimes \delta(c), \tag{2–7}$$

where x_c is the leaf modulator of node c.

We have the following theorems. Theorem 2.1 ensures the correctness of our solution in not re-encrypting other data items after the master key is changed.

Theorem 2.2 ensures the security of our solution in making the outsourced data unrecoverable after deletion. Their proof can be found in the Section 2.4.5.

Theorem 2.1. *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 2.2. *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 2.2 shows that the server cannot temper with modulators to circumvent the deletion. See the Section 2.4.5 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 $F(K, M_k^{(i)})$, $0 \leq i \leq l$, in $O(\log n)$ time, where l is the size of M_k, which is $O(\log n)$. Similarly, we can recursively compute $F(K', M_k^{(i)})$, $0 \leq i \leq l$, 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 $F(K', M_k^{(i)})$, $0 \leq i \leq l$.

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

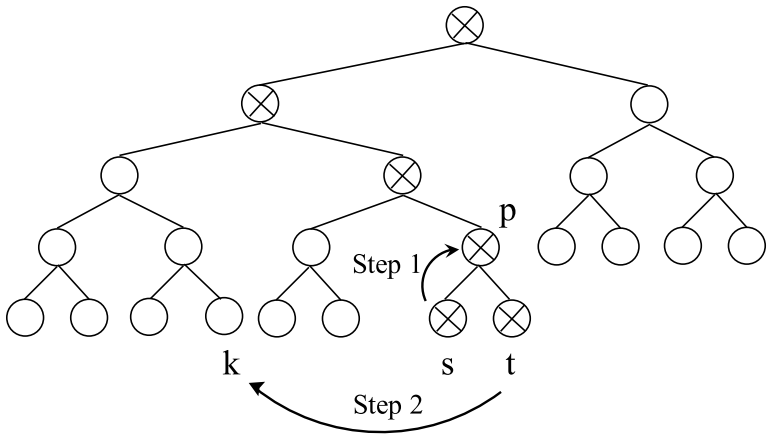

Figure 2-3. Balancing the tree after k is deleted. The server sends the nodes with cross inside to the client.

be the last leaf node at the last level of the modulation tree. See Figure 2-3. 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:

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

$$x'_s = F(K', M_p) \otimes H(F(K', M_p) \otimes x_{p,s}) \otimes x_s, \tag{2–8}$$

  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.

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

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:

$$x'_t = F(K', M_p + \langle x_{p,t} \rangle) \otimes F(K', M_{p'} + \langle x_{p',t} \rangle) \otimes x_t, \tag{2–9}$$

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.

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

where $M_p + \langle x_{p,t}, x_t \rangle$ is the list modulator for node t at its original location.
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. (4–2) and (2–9) require $O(\log n)$ hashes. Hence, the time complexity is also $O(\log n)$.

2.4.3 Access Modification and Insertion

Although the goal of this work 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.

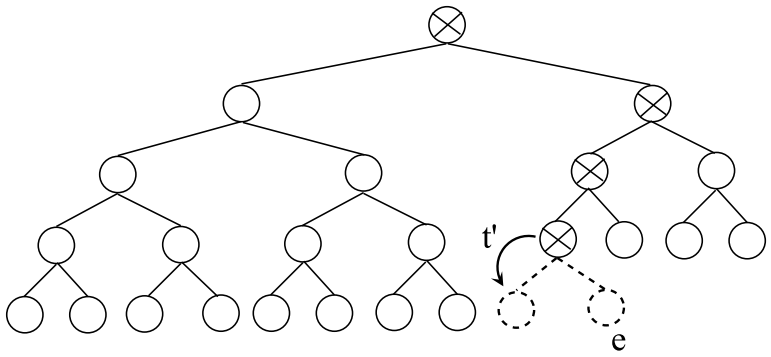

Figure 2-4. Balancing a new key e to the tree. The shape of the original tree does not have the dotted links and dotted nodes. The dotted links are created after the insertion.

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 2-4. 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 $x_{t'}$.

[3] Again we assume the correct return of requested item is enforced by another branch of research [5, 26, 52, 58], which ensures integrity in data storage and access.

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 (2.4.2), 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.

2.4.4 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.

2.4.5 Security Analysis

Security definition. 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*.

Security proofs. **Proof of LEMMA 1**: Let $S_k^{(l-i)}$, $0 \leq i \leq l$, be a suffix of M_k, containing the last $l-i$ modulators in M_k. In (2–1), if we treat $H(K \otimes x_1)$ as the new key, it becomes

$$F(K, M_k) = F(H(K \otimes x_1), S_k^{(l-1)}). \tag{2–10}$$

Next we prove by induction that

$$F(K, M_k) = F(F(K, M_k^{(i)}), S_k^{(l-i)}),\ 0 \leq i \leq l. \tag{2–11}$$

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

case of $i+1$ as follows:

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

Now according to (2–11), we have

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

Hence, in order to prove (2–4), it suffices to prove

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

By (2–2), 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. (2–5) can rewritten as

$$\delta(c) = F(K, M_{k'}^{(i-1)}) \otimes F(K', M_{k'}^{(i-1)}). \tag{2–12}$$

When the server receives $\delta(c)$, it performs either (2–6), which updates the modulator on the child link belonging to $P(k')$, or (2–7), 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 (2–6), (2–7) and (2–12), 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 3, we have

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

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, 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 (2–2) 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.

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 2-5, 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

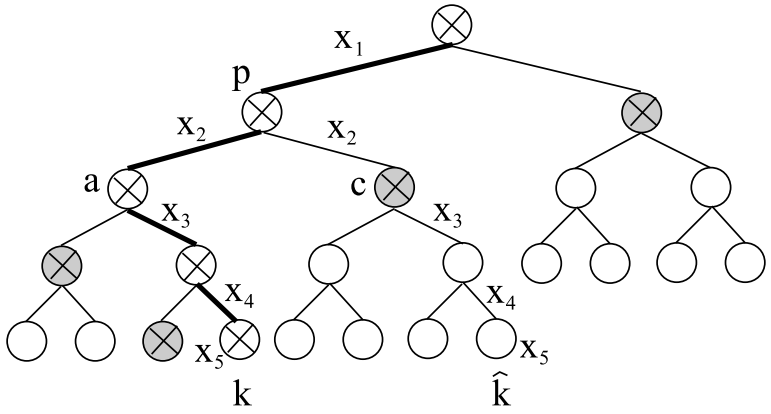

Figure 2-5. $MT^*(k)$ consists of nodes with cross inside. It contains $P(k)$ shown by bold lines and C shown by shaded nodes.

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 2-5. 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. □

2.4.6 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.

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) [12] 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) [21] 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 2-1. 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	Master-key	Individual-key	Our work
Client storage	$O(1)$	$O(n)$	$O(1)$
Comm./Comp.	$O(n)$	$O(1)$	$O(\log n)$

Table 2-2. Experimental comparison, including client storage overhead, communication overhead for deletion, and computation overhead for deletion.

Overhead	Master-key	Individual-key	Our work
Client storage	16 Bytes	1.53 MB	16 Bytes
Comm. overhead	391 MB	0	1.61 KB
Comp. overhead	5.5 minutes	almost 0	0.24 ms

Performance comparison. We compare our two-party solution with the master-key solution and the individual-key solution, which do not require a third party, either. The difference between our solution and those requiring third parties [48, 48, 56] is more fundamental than performance alone, as we have discussed their security problem under the threat model of this work 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.

- Complexity Comparison
  Table 4-1 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 $O(1)$.

- Experimental Comparison
  We further compare the deletion overhead of the three solutions through real experiment. The results are shown in Table 2-2. 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.

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 2-18, 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.

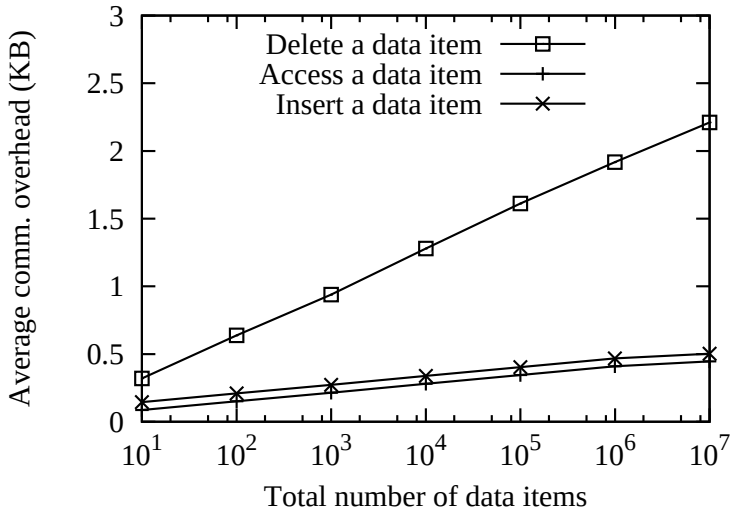

Figure 2-6. Communication overhead for deleting, inserting, or accessing a data item. It includes all information that the client sends or receives for an operation.

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.

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 2-7, 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.

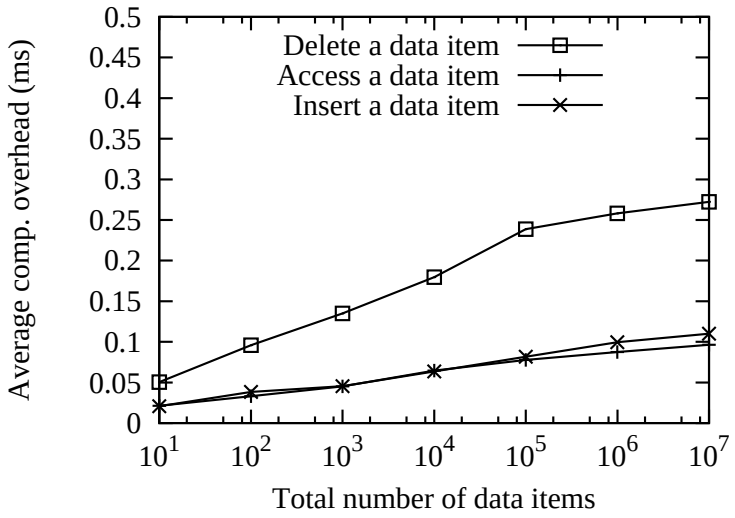

Figure 2-7. Client computation overhead for deleting, accessing, or inserting a data item.

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 2-3. 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%.

2.5 Recursively Encrypted Red-black Key Tree Based Solution

In Key modulation based solution, we consider the integrity protection as a solved problem. But in recursively encrypted key tree based solution, we involve the integrity protection into the solution.

Table 2-3. 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

2.5.1 Threat Model

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.

2.5.2 Recursively Encrypted Red-black 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.

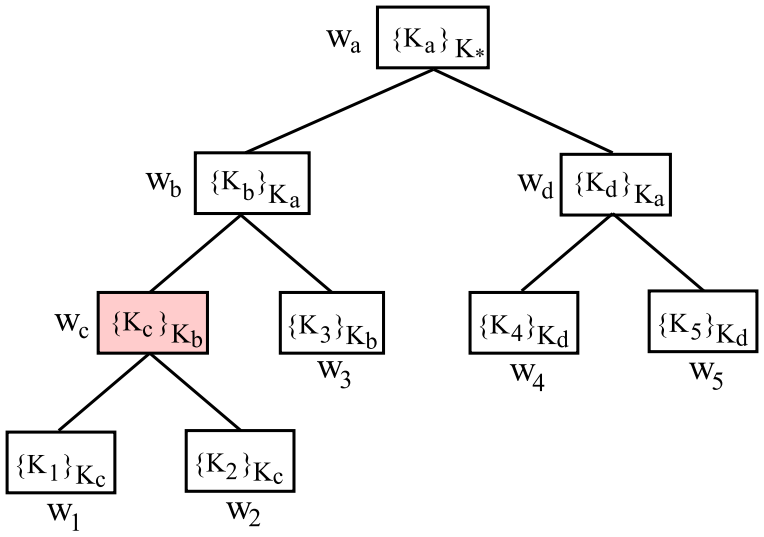

Figure 2-8. A Recursively Encrypted Key tree (RERK) constructed on 5 keys

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.

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 2-8) 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 2-8 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:

$$EK(w_x) = \begin{cases} \{k_x\}_{k_*} & \text{if } w_x \text{ is the root} \\ \{k_x\}_{p_x} & \text{otherwise} \end{cases} \tag{2–13}$$

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 2-8, 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.

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 [43] 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 [26] — 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 2-8, $r(w_1)$ 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

$$t(w_x) = h(EK(w_x)||r(w_x)||t\_(w_x)), \tag{2–14}$$

where $||$ is the concatenation operator and

$$t\_(w_x) = \begin{cases} NULL & \text{if } w \text{ is a leaf node} \\ h(t(w_l)||t(w_r)) & \text{otherwise} \end{cases} \tag{2–15}$$

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.

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:

$$t(w_x) = h(EK(w_x)||r(w_x)||t\_(w_x)||col(w_x)||red(w_x)). \tag{2–16}$$

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 work, we assume that clients require frequent insertion and deletion. So according to the performance comparison in [49], we choose the red-black tree.

2.5.3 Proof of Re-Balancing Complexity

In this section, we will prove that in order to finish the re-balancing, the client only needs to look up the server at most two times to retrieve $O(\log n)$ nodes.

Theorem 2.3. *For each update operation, e.g., insertion and deletion, the client can re-balance the RERK, and re-compute the metadata by looking up the server for at most two times and retrieving* $O(\log n)$ *nodes.*

Suppose a client has outsourced the RERK to a server. When the client looks up for a key, it can construct a partial RERK based on the node sequence returned from the server. Assume that $w_i, v_k, v_{k-1}, \ldots, v_1$ is the node sequence from the i^{th} leaf node w_i to the root v_1 and $w_j, v'_k \ldots, v'_2$ are their siblings. For each node v in the partial RERK, the client can acquire a set of values: $I_v = \{k_v, EK(v), r(v), col(v), t\_(v), t(v), red(v)\}$.

We first show that the client can perform the following three basic operations on the partial RERK.

- Color changing: The client can change the color of any node in the partial RERK and re-compute its tag.
- Rotation: For any three nodes in the partial RERK, the client can make two of them as the children of the third one and re-compute the tags of these three nodes.
- Index computation: For any internal node (except the root) in the partial RERK, if the internal node is a left (or right) child of its parent, the client can compute the smallest (or largest) index of the leaf node that belongs to the subtree whose root is the sibling of the internal node.

The above operations can be performed as follows:

- Color changing: If the client wants to change color of a node v, it can re-compute the new tag according to Equation 2–15 by changing $col(v)$ and re-computing the hash.
- Rotation: If the client tries to make two nodes v_l and v_r as the children of the third node v, it will first compute $t\_(v)$. Then it can compute the new tag of v based on $t\_(v)$.
- Index computation: Suppose the internal node is v_j, where $j \in [2, k]$ and the index of w_i is i. If v_j is the left (or right) child, the client can compute the smallest (or largest) index of the leaf node that belongs to the subtree whose root is v'_j by traversing the nodes from w_i to v_j, adding (or subtracting) the rank of their left siblings. Algorithm 1 describes the process.

Input: $l_{w_i}, l_{v_k}, l_{v_{k-1}}, \ldots, l(v_j), l_{w_j}, l_{v'_k}, \ldots, l_{v'_j}$
Output: smallest (or largest) index in subtree rooted at v_j's right (or left) sibling
$index = i$
if v_j *is the left child* **then**
  $index = index + 1$
  foreach x *from* k *to* $j-1$ **do**
    if $d(v_x) = 1$ **then**
      $index = index + r(v'_x)$
    end
  end
end
else
  $index = index - 1$
  foreach x *from* k *to* $j-1$ **do**
    if $d(v_j) = 0$ **then**
      $index = index - r(v'_x)$
    end
  end
end
return $index$

Algorithm 1: Index computation

Next, we show that re-balancing after insertion and deletion only involves $O(\log n)$ nodes.

Insert a key: If the client tries to insert a key by adding a new leaf node w'_i at position i, where $i \in [1, n]$. The client will first look up for the key w_i and construct a

partial RERK. According to the red-black insertion algorithms in [19], the client will generate a new red internal node v_{k+1} and make w_i' and w_i as its left child and right child. However, if v_k is also a red node, then the insertion will make the RERK tree become imbalanced.

There are 8 types of imbalance: LLr, LRr, RLr, RRr, LLb, LRb, RLb, RRb. The first letter L or R encodes the relationship between v_{k+1} and v_k. The second letter L or R shows the relationship between v_k and v_{k-1} and the last r or b stands for the color of v_k'. For example, if v_{k+1} is the left child of v_k, v_k is the left child of v_{k-1} and v_k' is a red node, then the imbalance type is LLr.

Imbalances of the type XYr (X and Y may be L or R) can be handled by color changing. If color changing causes further imbalance because v_{k-2} is also red, re-balancing will continue. If color changing does not cause further imbalance, we finish here.

Figure 2-9 shows the color changes performed for LLr and LRr imbalances. If v_{k-1} is the root, we do not change the color of v_{k-1}. Notice that the red color will show up as grey in black-n-white print in this and other figures.

Rotation and color changing can eliminate the imbalances of types XYb (X and Y may be L or R). Figure 2-10 shows the rotation and color changes for LLb and LRb imbalances. As rotation and color changing only involve the nodes in the partial RERK, the client does not need other information except for the partial RERK to perform the rotation.

Accordingly, imbalances caused by insertion can be eliminated by rotation and color changing. The client can perform the insertion by only one lookup which involves $O(\log n)$ nodes.

Delete a key: If the client tries to delete a key k_i by deleting a leaf node w_i from the RERK, the client will first look up for the key k_i, and construct a partial RERK. According to the red-black deletion algorithm [19], if the leaf node w_i is deleted, its parent v_k will

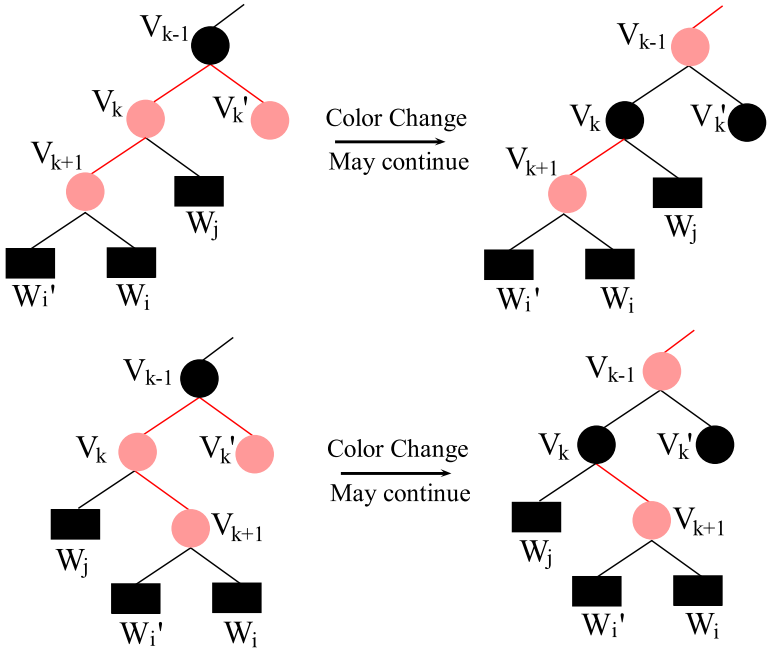

Figure 2-9. LLr and LRr color change

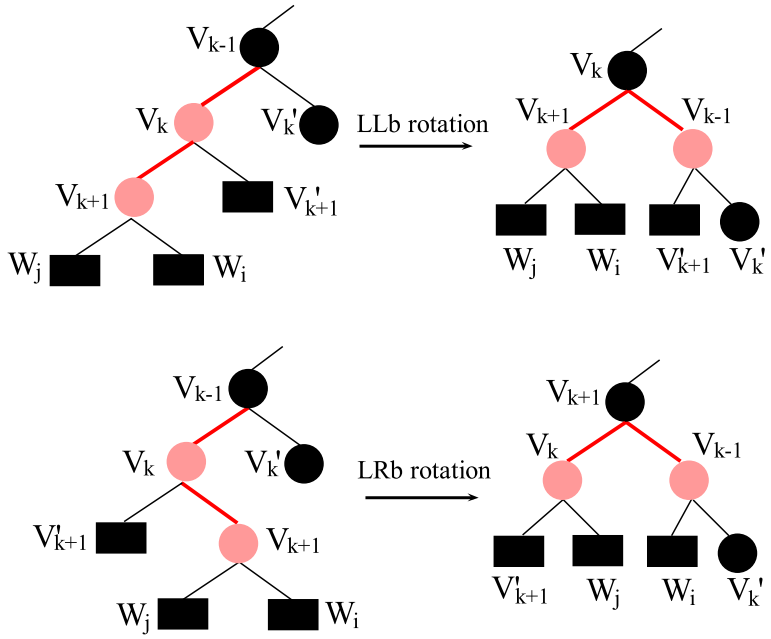

Figure 2-10. LLb and LRb rotation and color change

be deleted and replaced by its sibling w_j. As v_k is deleted, the subtree whose root is w_j becomes deficient. We denote the root of the deficient subtree as y; see Figure 2-11 for illustration.

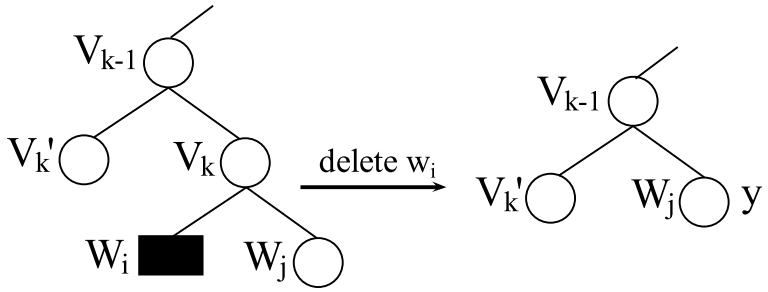

Figure 2-11. y is the root of the deficient subtree

The imbalance types can be categorized as the following five cases.

- Case 1: if y is a red node, changing the color of y can eliminate the imbalance. If y is the root, then the entire tree becomes deficient. No further work needs to be done.

- Case 2: *y and v'_k are both black, and v'_k has no red child.*
  If v_{k-1} is black, the client will need to change the color of v'_k and v_{k-1} becomes y. Re-balancing will continue. If v_{k-1} is red, changing color of v_{k-1} will eliminate the imbalance. Figure 2-12 illustrates this case.

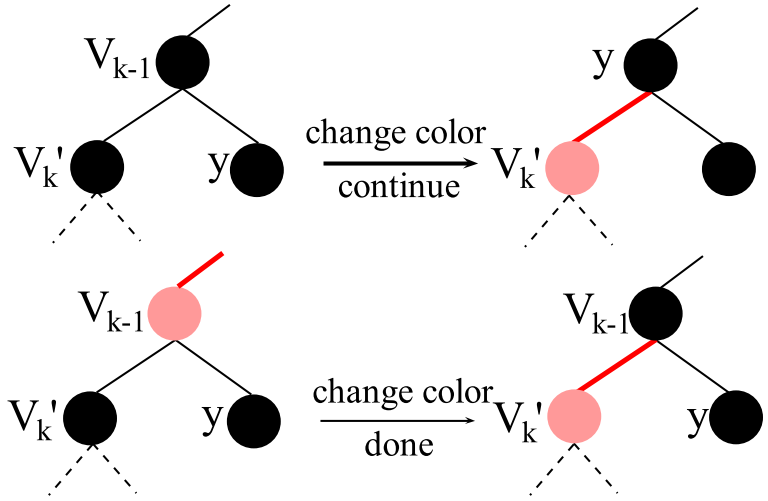

Figure 2-12. Case 2. y and v'_k are both black. v'_k has two black children.

- Case 3: If y and v'_k are both black, and v'_k has only one red child, then the client will need color changing and rotation to eliminate imbalance. Figure 2-13 illustrates this case. However, in this case, The client does not have enough node values to finish the re-balancing. For example, in the upper plot, it does not have the values of a and b. In the lower plot, it does not have the values of a, b, c and $v'\prime$. Therefore, In order to get enough values, it will first compute an index using

Algorithm 1 by setting $v_j = y$. Then it looks up another key with the computed index. After acquiring enough node values, the client can perform rotation and color changing.

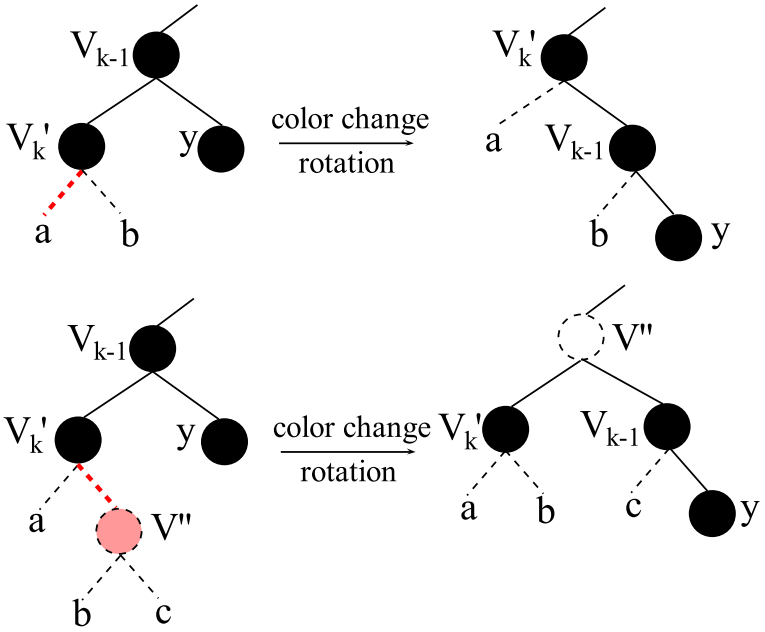

Figure 2-13. Case 3. y and v'_k are both black. v'_k has only one red child. Dotted line and cycle indicate that the client cannot acquire the values of the node based on this lookup.

- Case 4: If y and v'_k are both black, and v'_k has two red children. This case is similar to the lower plot of case 3 in Figure 2-13. The client will need to look up for a key again to eliminate the imbalance.

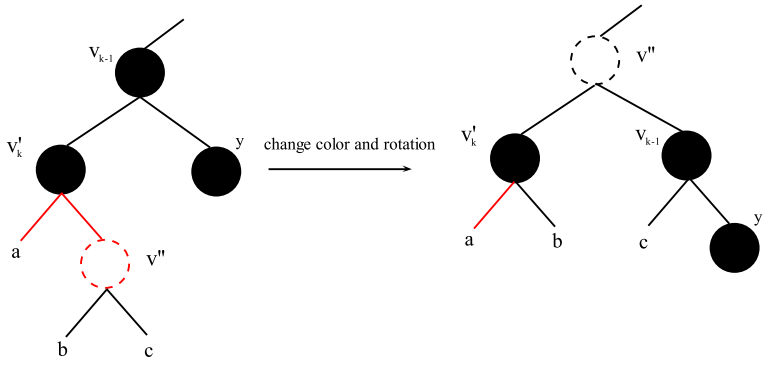

Figure 2-14. Case 4: y and v'_k are both black, and v'_k has two red child

- Case 5: If y is black but v'_k is red, there are totally four types of imbalances (See Figure 2-15). Similar to case 4, we need another lookup to finish the re-balancing.

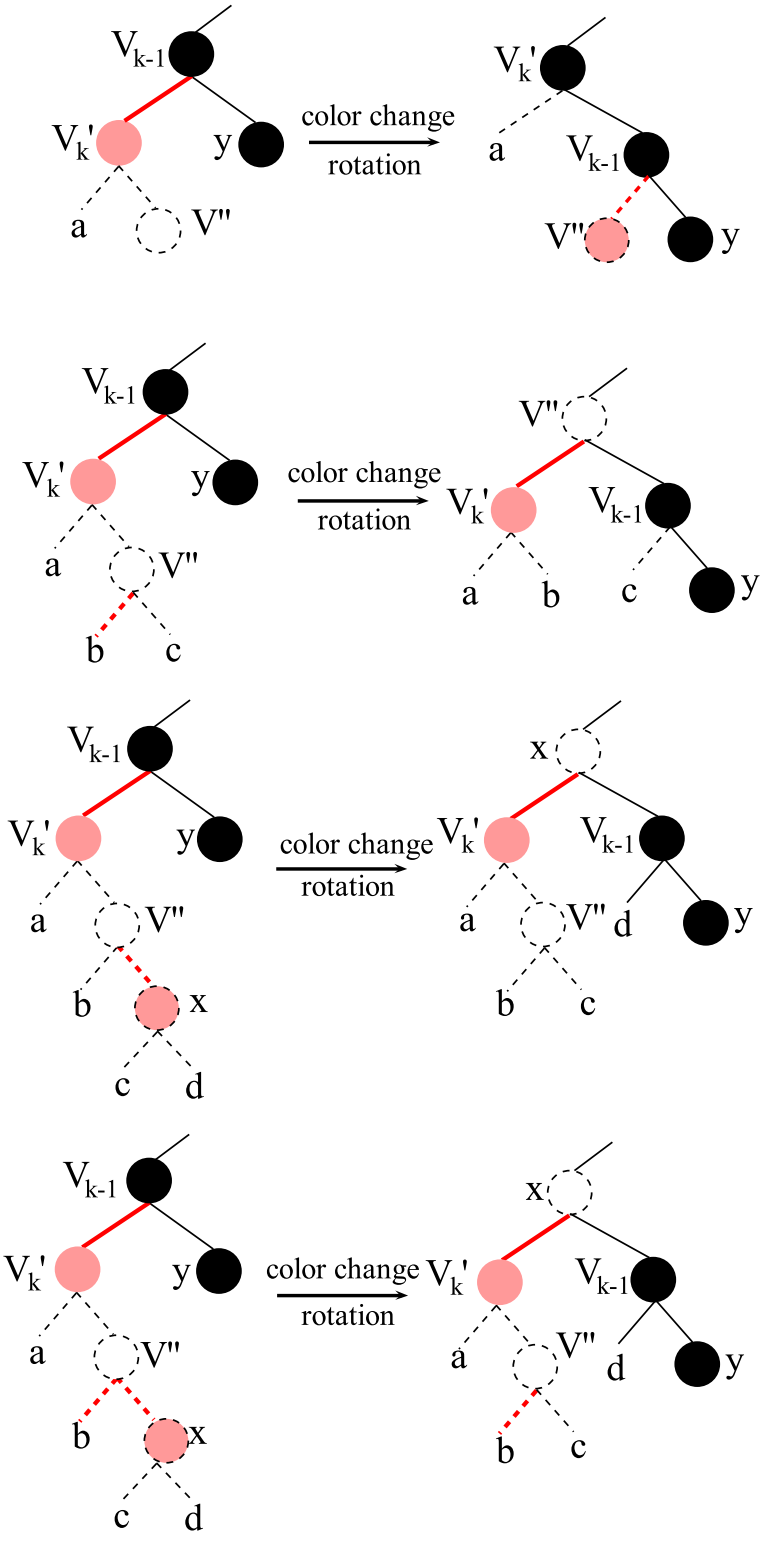

Figure 2-15. Case 5. y is black but v_k' is red

After the first lookup, if the deficient can be handled by case 1, the re-balancing involves $O(\log n)$ nodes. If the deficient can be solved by cases 3, 4 and 5,

then another lookup is needed. So the re-balancing involves $O(\log n)$. If the re-balancing will continue in case 2, the deficiency will finally be solved by another case. So obviously, The client can perform deletion with at most two lookups which involve $O(\log n)$ nodes.

2.5.4 Key Deletion and Insertion

We present the deletion algorithm first. The confidentiality and integrity protection mechanisms embedded in RERK (Section 2.5.2-2.5.2) ensure the correctness of this algorithm, as our security analysis will show. The red-black tree embedded in RERK (Section 2.5.2) 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:

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 [19] 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 2-16, where k_4 (thus data item m_4) is deleted from the RERK in Figure 2-8. The left-top plot in Figure 2-16 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

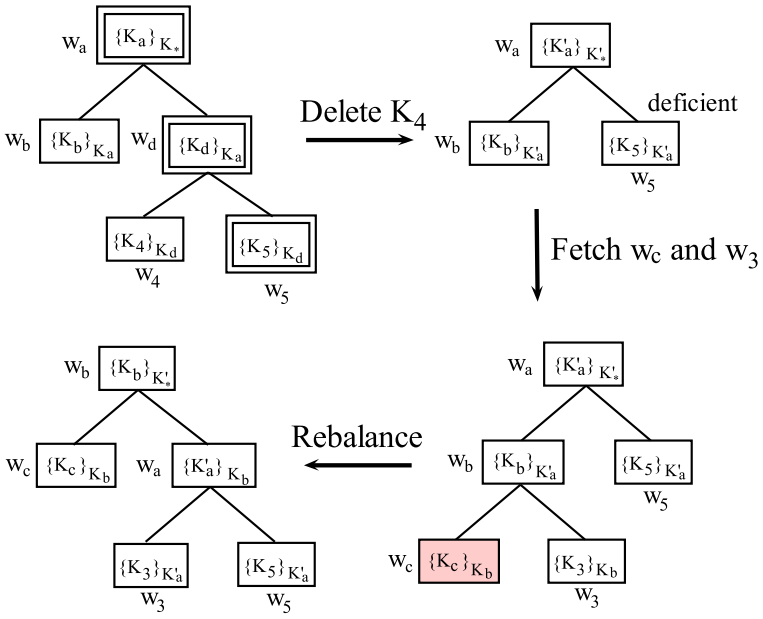

Figure 2-16. Example for key deletion in the RERK. Double-boxes in the left top represent the node sequence from the leaf node to the root, and other nodes are their siblings.

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

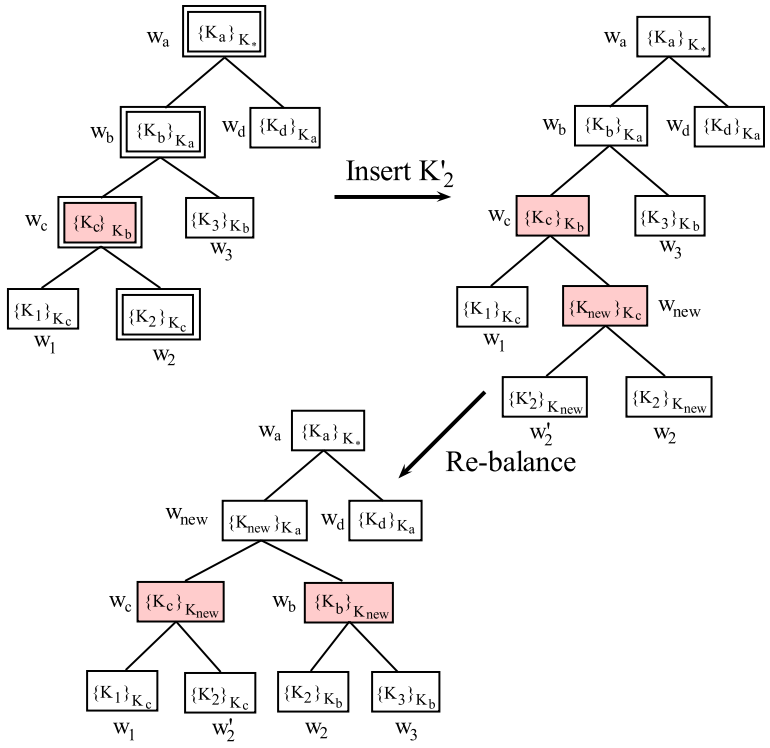

Figure 2-17. Example for key insertion in the RERK

(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 [19]. 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.

An example is given in Figure 2-17, where a new key k_2' is inserted at the 2^{nd} position of the RERK in Figure 2-8. The left-top plot in Figure 2-16 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.

2.5.5 Security Analysis

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.

Security proofs.

Theorem 2.4. *If there exist (1) a collision-resistant hash function which is used in the RERK construction and (2) an IND-CPA secure encryption solution which is used to encrypt the outsourced data items, then the proposed assured deletion solution 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 solution used in the assured deletion solution.

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 (2–16) and (2–15). 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 [26], 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 solution 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 2.5.4, 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 solution, which is against the theorem assumption that the adopted encryption solution is IND-CPA secure. □

2.5.6 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 computation 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 $O(\log n)$.

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) [12] in the RERK. SHA-1 produces a 160-bit message digest. We choose Advanced Encryption Standard (AES) [21] 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.

Communication overhead. We measure the communication overhead between the client and the server through experiments, and the results are shown in Figure 2-18. 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.

Clearly, all measured communication overheads increase logarithmically with respect to the number of data items, demonstrating good scalability. For a data set of

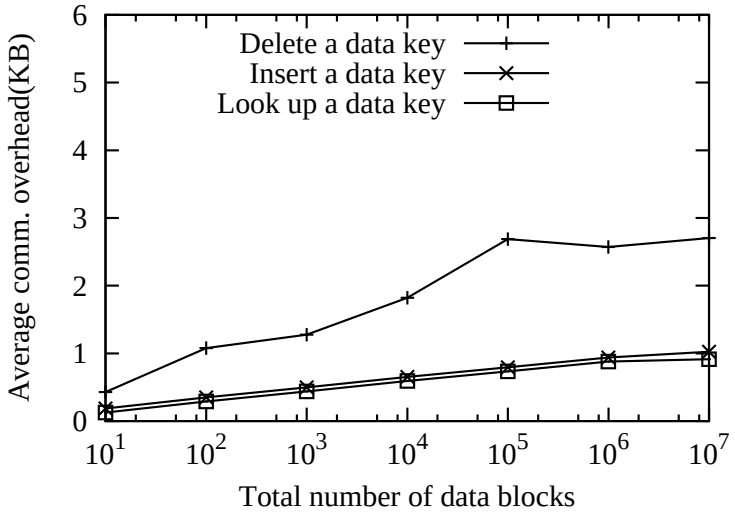

Figure 2-18. Average communication overhead between the client and the server. The x-axis shows the total number of data items in logarithmic scale. The y-axis shows the average communication overhead in KB.

10^6 items, the communication overhead can fit in a few IP packets of 1500 bytes each in most cases.

Computation overhead. Next, we measure the computation overhead of the server and the client separately. On the server side, the main computation 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.

Client Computation

Figure 2-19 shows the computation 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.

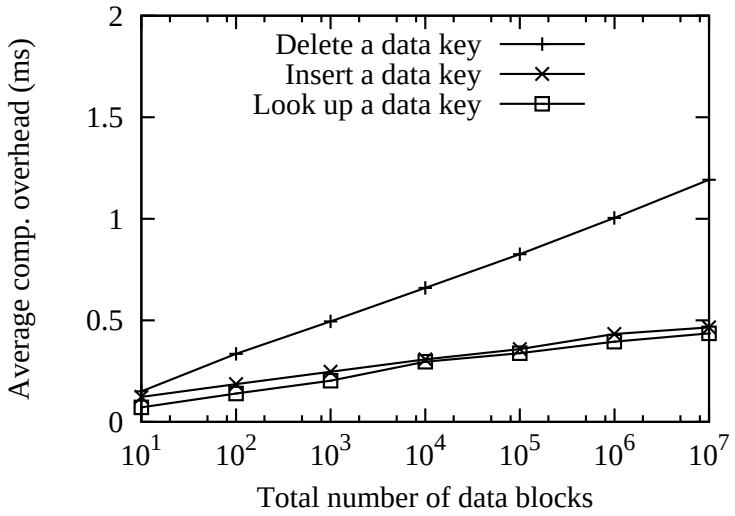

Figure 2-19. Client computation overhead. The x-axis shows the number of data items in logarithmic scale. The y-axis shows the average computational time of the client.

Server Computation

We perform lookups on all keys. Figure 2-20 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 computation 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.

2.6 Summary

This work presents two 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

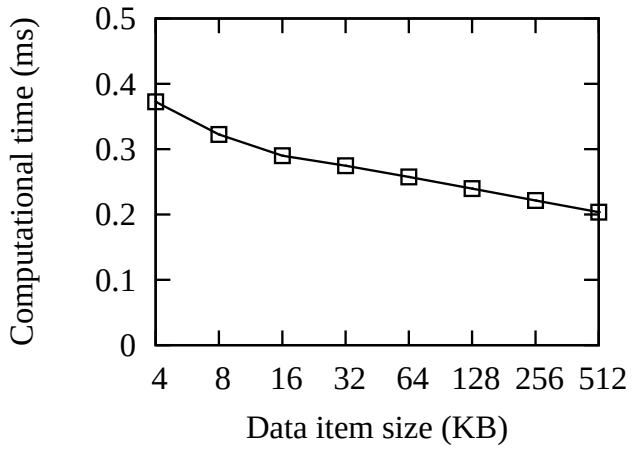

Figure 2-20. Server computation overhead. The x-axis shows the number of data items in logarithmic scale. The y-axis represents the average time for the server to process a client request.

allowing them to perform deletion on any data item in any file without causing significant overhead. Our solutions are based on a novel key modulation function and a recursively encrypted key tree. We prove their correctness and security, and implement it on the Amazon EC2 system.

CHAPTER 3
DATA INTEGRITY PROBLEM IN CLOUD COMPUTING

3.1 System Model

As shown in Figure 3-1, Our system model 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 lose direct control of their data. They access, update and check the integrity of their data by sending requests to the servers. Due to possible external/internal compromises, 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.

3.2 Related Work

Most existing solutions can be categorized along two research threads: Proof of Retrievability (PoR) [11, 34, 52, 58] and Provable Data Possession (PDP) [5, 6]. The first PoR solution is proposed by Juels and Kaliski in [34]. The first PDP solution is proposed by Ateniese et al. [5]. Both categories allow users to verify if the cloud correctly possesses their data. That is, by keeping some local meta data and verifying the proof returned from the cloud, users can (probabilistically) determine whether

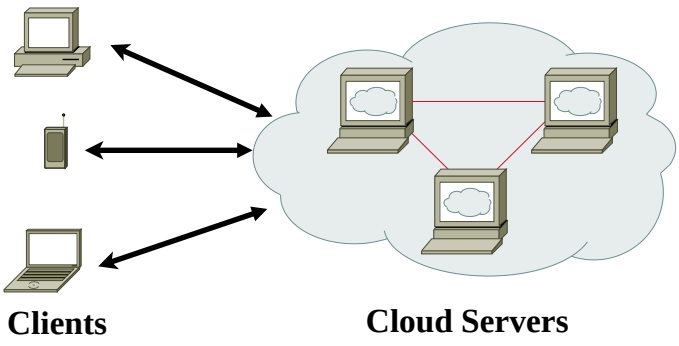

Figure 3-1. A cloud system.

their data are intact. The above solutions have two limitations: First, they either do not support dynamic data update or do so with significant overhead, particularly in terms of worst-case complexities. Second, they protect users from the cloud's misbehavior, but do not protect the cloud from the users' misbehavior. This work expands data integrity protection by covering an important complementary problem: When a user claims a data loss, how can we be sure that the user is correct and honest about the loss? If a user tries to blackmail the cloud by lying about data loss, how can the cloud prove its innocence?

Ateniese *et al.* [5] propose the first provable data possession (PDP) model to check the integrity of outsourced data. But their solution does not support data update. A followup work by Ateniese *et al.* [6] introduces a dynamic version of PDP. Unfortunately, it cannot support all types of data update operations.

Juels and Kaliski formalizes a solution called Proofs of Retrievability (PoR) to verify data integrity through "sentinel" blocks [34], but it does not support data update, either. Shacham *et al.* introduce an improved version of PoR called Compact PoR [52]. But still their solution is not designed to efficiently support dynamic data updates.

Following the work of [52], Erway *et al.* [25] propose a dynamic provable data possession solution (DPDP). Using a rank-based skiplist, their solution supports dynamic data update. However, the updating algorithm in their solution is not accurate. According to their solution, the cloud user only needs to query the server one time to finish the updating. Unfortunately, after each deletion, the user needs to reconstruct the skiplist. As the skiplist is stored on the server, querying once may not provide enough information for the user to finish the deletion. Accordingly, maintaining the sequential order among nodes at the bottom level of a skiplist makes updating (such as deletion) complicated. Moreover, the skiplist is a probabilistic data structure, whose worst-case overhead complexity is $O(n)$ [50], where n is the number of blocks.

Wang *et al.* [58] define a dynamic version of PoR based on the BLS signature and a sequenced Merkle Hash Tree (MHT) [42]. They use a modified BLS signature and a classical MHT construction to realize data-possession verification in cloud storage. After inserting or deleting data blocks, the tree will become unbalanced. Particularly, if the client keeps appending blocks at the end of the file, the height of the tree will increase linearly. As a result, the worst-case complexities for searching and updating are $O(n)$.

Zhang and Blanton take a different approach [62] that requires the client to locally record information about update history, using a balanced update tree whose size is $O(M)$, where M is the number of updates. Even though sequential indices are explicitly bound with blocks through MACs, the update tree allows the client to translate indexing information without having to re-computing MACs. The above approach however puts significant storage burden on the client. This work will follow the path of the prior work [5, 25, 34, 58] whose client storage requirement is a constant. See table 4-1 for a summary of the existing work.

Table 3-1. Summary of existing work

Features	Ateniese	J&K [34]	Shacham [52]	Wang [58]	Erway [25]
Dynamic updates	NO	NO	NO	YES	YES
Public verification	NO	NO	YES	YES	YES
Worst comm. complexity	O(1)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Worst comp. complexity	O(1)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Average comm. complexity	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$
Average comp. complexity	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$

3.3 Data Possession Verification and Basic Approach

Consider an arbitrary client and an arbitrary file F that the client outsources to the cloud. Suppose F consists of n data blocks, $\{m_1, m_2, \ldots, m_n\}$. Each block may contain a data key (ID) to allow lookup and access of a specific block of interest (for example, the record of a particularly employee indexed by the employee ID in a payroll file). The blocks do not necessarily have the same size. The problem is to design a data-possession verification solution that allows the client to (1) detect whether some of the blocks have been lost or corrupted at the cloud server, and (2) in the meantime make sure that the cloud can fend off false claims of client data loss.

Much existing work focuses on addressing the first part of the above problem based on a common basic approach [25, 52, 58]: The client randomly selects a subset of k blocks and queries the cloud for a proof, demonstrating that it possesses these blocks. After receiving the proof, the client verifies the proof using the meta data it has pre-computed and kept locally. If the received proof does not match what's expected from the meta data, the client claims that the cloud has lost some of its data. It is known that if the cloud loses k' data blocks, the probability of being detected after a single client query of k blocks is $1 - \frac{\binom{n-k'}{k}}{\binom{n}{k}}$ [5]. As an example, if 1% blocks are lost by the server, the client can achieve 99% detection probability by querying 460 blocks. If the above approach is performed periodically for l times, each time on an independent subset of 460 blocks, the detection probability will become $1 - (1 - 99\%)^l = 1 - 10^{-2l}$.

3.4 Enabling Efficient Dynamic Updating in Cloud Computing

Though Wang *et al.* propose a dynamic version of PoR model in [58] and Erway *et al.* present a dynamic PDP model in [25], unfortunately, the performance of their solutions are not tightly bounded. Accordingly, we design a new Cloud Merkle B+ Tree (CMBT) to assist the verification procedure, whose *worst-case* computation and communication overhead for inserting/deleting/updating a data block is $O(\log n)$, comparing with $O(n)$ worst-case overhead in [25, 58].

3.4.1 Cloud Merkle B+ Tree Based Design

BLS signature. Suppose the encoded file F is divided into n blocks: $m_1, m_2, \ldots, m_n$. For a bilinear map $e : G \times G \rightarrow G_T$, the private key and the public key are defined as $x \in \mathbb{Z}_p$ and $z = g^x \in G$ separately, where g is a generator of G. For each block m_i, where $i \in [1, n]$, we define the signature on the block m_i as $\sigma_i = [H(m_i)u^{m_i}]^x$. $H(m_i)$ is called the block tag, and u is another generator of G. We denote the set of signature as $\Phi = \{\sigma_i\}$, where $1 \leq i \leq n$.

CMBT construction. A merkle hash tree [42] has been widely used in checking memory integrity [30, 59] and certificate revocation [36, 46] because it is easy to realize and has $O(logn)$ complexity in both the worst case and the average case. However, directly using the classic merkle tree in cloud storage may cause an efficiency problem. So we develop an authenticated data structure based on a B+ tree and a merkle hash tree. We call it **Cloud Merkle B+ tree (**$CMBT$**)**. In our construction, we choose a B+ tree of order three[1] and require that each data node can store three elements at most.

We treat the sequence of block tags $H(m_1)$, $H(m_2)$,..., $H(m_n)$ as elements and insert them into a B+ tree sequentially, then we can get a B+ tree (see Figure 3-2), we will construct the $CMBT$ based on it.

For each node w in $CMBT$, we store six values:

[1] The B+ tree [33] is different from the B tree in following three aspects: 1. A B+ tree has two types of nodes - index nodes and data nodes. Index nodes store keys while data nodes store elements. But a B tree has only one type of node - data nodes 2. All data nodes in a B+ tree are linked together by a doubly linked list, but data nodes in a B tree are not linked. 3. The capacity of data nodes and index nodes can be different in a B+ tree, while the capacity of nodes in a B tree should be the same. For example, a B+ tree of order n means that the index nodes (except for the root node) can hold $n-1$ keys at most and hold $\lceil n/2 \quad - \quad 1 \rceil$ keys at least. But each data node can contain c elements at most and $\lceil c/2 \rceil$ elements at least. c and n can be different. The root node can hold n children at most and two children at least.

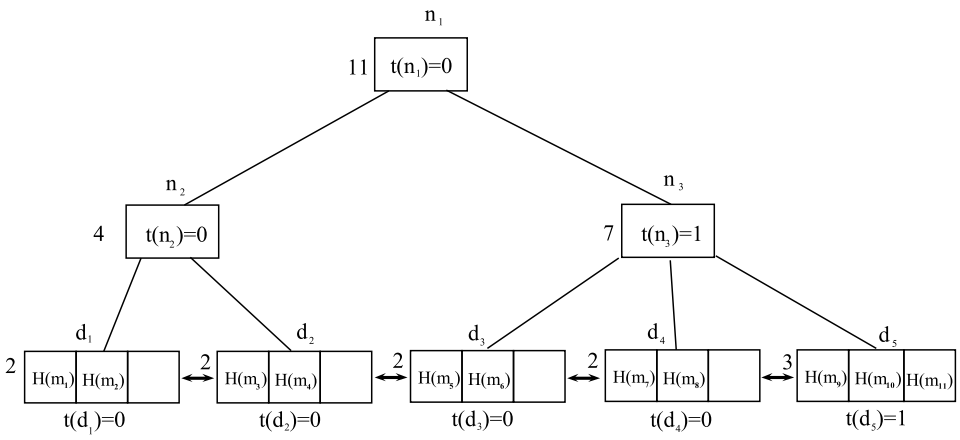

Figure 3-2. The cloud merkle B+ tree

- $left(w)$, $middle(w)$ and $right(w)$: For an index node, these three variables represent its left child, middle child and right child. If this node has only two children, then $right(w)$ will be NIL. For a data node, these three variants represent the elements it stores from left to right. If corresponding position has no element, NIL will be set.

- $rank(w)$: Rank of the node. For an index node w, $rank(w)$ stores the number of elements[1] that belong to the subtree whose root is w. For a data node w, $rank(w)$ stores the number of elements that belong to w. In Figure 3-2, we show the rank value for each node on the left side of each node. For example, the rank of node d_1 is 2 because from d_1 we can visit two elements $H(m_1)$ and $H(m_2)$.

- $t(w)$: We do not store keys in index node because we do not need to search the $CMBT$. Instead, we store the type of the node as $t(w)$. The definition of $t(w)$ shows as follows.
  For a node w in the tree:
  Definition 1.

$$t(w) = \begin{cases} 0 & \text{if } w \text{ has 2 children or contains 2 elements} \\ 1 & \text{if } w \text{ has 3 children or contains 3 elements} \end{cases}$$

  We also show the type value of each node in Figure 3-2.

- $v(w)$: The value of node. $v(w)$ is defined as follows:
  Definition 2.

$$v(w) = h(v(left(w))||v(middle(w))||v(right(w))||t(w)||rank(w))$$

  where $||$ means concatenation.
  Also for each element e that contains a block tag $H(m)$, we define the value of the element as follows:
  Definition 3.

$$v(e) = h(H(m))$$

With above definitions, the client can construct a $CMBT$ and get the value of the root R. Then the client will sign the root value $v(R)$ using its private key: $sig_{sk}(v(R)) \leftarrow (v(R))^{sk}$. Next the client will outsources the encoded file F, the block signature set Φ, the $CMBT$ and the root signature $sig_{sk}(v(R))$ to the server.

3.4.2 Compact Merkle Hash Tree Based Design

Although CMBT can tightly bind the worst case communication/computational overhead but performing tree rotation remotely requires significant information exchange between the client and the server. Our experiments reveal that its average update overhead is considerably higher than DPDP [25]. Accordingly, we present a new design based Compact Merkle Hash Tree to further optimize the performance.

Tag and signature.

For each data block, we define two auxiliary values needed by our data-possession verification solution. These values will be stored at the cloud server. Only the meta data for the whole file (defined later) will be kept by the client.

Block tag: For each block $m_i \in F$, we define its *tag* as $t_i = H(m_i)$, where H is a collision-resistant hash function, which will be discussed shortly. A tag is a fixed length representation of a data block (whose length may be arbitrarily set) in the data structure of CMHT.

Homomorphic Signature: The client chooses $N = pq$ where p and q are two large primes and g is an element of high order in $\mathbb{Z}_N^*$. The client keeps p and q, and sends N and g to the server. The signature for block m_i is defined as

$$\sigma_i = t_i \cdot g^{m_i} \mod N,$$

where t_i is the tag of m_i. Note that it is possible for our solution to use other homomorphic signatures such as BLS [10].

Tree construction. The client first constructs a red-black tree with n leaves. For convenience, we denote the n leaves from left to right as $w_1, w_2, \ldots, w_n$. The leaf node w_i represents the data block m_i.

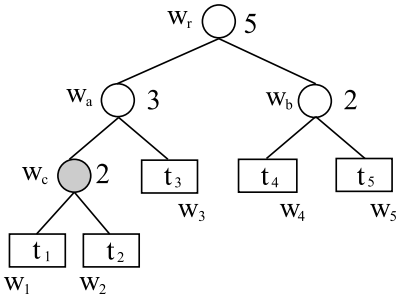

Figure 3-3. A Coordinate Merkle Hash Tree (CMHT) constructed for 5 blocks

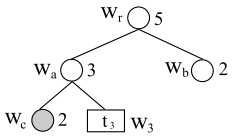

Figure 3-4. The partial CMHT from the root to w_3

Let w_x be an arbitrary node in the tree, where x may be a number (for leaf) or a letter (for internal node). To support the integrity verification, each node w_x in the CMHT needs to carry four values: a rank $r(w_x)$, a color $col(w_x)$, a red-children counter $red(w_x)$ and a label $l(w_x)$. The rank is defined as the number of leaf nodes in the subtree rooted at w_x. For example, in Figure 3-3, $r(w_1)$ is 1, $r(w_b)$ is 2, and $r(w_r)$ is 3. 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.

For the label value: As the leaf node of a red-black tree is always a black node with rank 1, if w_x is a leaf node representing a block m_i, its label is simply the tag t_i of the block; if w_x is an internal node whose two children are w_{left} and w_{right}, its label is computed by hashing the concatenation of the labels of the children.

$$l(w_x) = \begin{cases} H(l(w_{left})||l(w_{right})||r(w_x)||col(w_x)||red(w_x)) & \text{if } w_x \text{ is an internal node} \\ t_j & \text{if } w_x \text{ is a leaf node for block } m_j \end{cases}$$

Use of coordinates. Interestingly, the client does not need to keep track of the exact coordinate of each data block because it is not used in data access. The *only* purpose of introducing coordinates is to help the client detect data loss, which is

performed in a random-sampling way: Since the client knows n from its meta data, it knows the exact shape of the complete binary tree CMHT and thus knows the set of valid coordinates. The client challenges the server with a randomly selected subset of valid coordinates $\{c_i\}$, which corresponds to a subset of data blocks $\{m_i\}$ to be verified, where m_i is the block *currently* assigned with c_i. We stress that the whole purpose of designing CMHT is to make sure that the server will return the *correct* tags $\{H(m_i)||c_i\}$ that match the client's *current* meta data, more specifically, the root's label $l(w_r)$, using the standard Merkle tree operations (which prevent the server from cheating). After that, the server is required to produce a compact proof of its knowledge of blocks $\{m_i\}$ that match the blocks' fingerprint $\{H(m_i)||c_i\}$.

All current data blocks must be represented in the CMHT, but their specific coordinates (e.g., locations in the tree) are *not important* to data-possession verification because coordinates are randomly sampled and each block (its coordinate) has equal chance to be sampled.

3.5 Enabling Non-Repudiable Property in Cloud Computing

In order to realize non-repudiation, we need to design a new metadata. After the client constructs the CMHT, it sends the tree to the cloud server, together with F and Φ. The server verifies the labels on the tree. Let T be the current time stamp. We define the *meta data* as follows:

$$\mathcal{M} = \{l(w_r), T, n, \sigma_{\mathcal{M}}\},$$

which includes the root's label $l(w_r)$, the time stamp T, the total number n of blocks, and a digital signature [35], $\sigma_{\mathcal{M}} = Sign_{sk_s}(Sign_{sk_c}(l(w_r)||T||n))$, jointly signed by the client and the server using their private keys, sk_c and sk_s. The meta data is stored by the client and the server separately as $\mathcal{M}_c$ and $\mathcal{M}_s$. Both sides can use the other's public key to verify the signature on the meta data, which enforces authenticity and consistency between the two sides. Compared to the meta data in [52, 58], our meta data has the following two security properties.

Unforgeability: As the meta data includes the signature signed by both the client and the server, neither the client nor the server can forge the meta data.

Distinguishability: After each update of the tree, the client and the server will agree on a new time stamp and update the signature. As the time stamp T increases monotonically, if the two sides have dispute over which meta data is current, it is easy to resolve the dispute by authenticating the signatures with their public keys and comparing the time stamps.

The client only stores the meta data $\mathcal{M}_c$. Everything else, including CMHT, F and Φ, is outsourced to the server. The server needs to maintain an internal data structure to map between data blocks and their corresponding nodes in the CMHT. When the server stores the CMHT, it keeps a location field in each leaf node, specifying where the corresponding data block is stored. The server also keeps track of each data block's size and its coordinate in the CMHT, allowing flexible access of the node in the tree for any given data block.

3.6 Efficient Dynamic Data Possession Verification Solution with Non-repudiable Property

3.6.1 Problem Statement

Our objective is to design a *non-repudiable integrity verification solution* for cloud storage systems. It supervises not only the servers but also the clients. On the one side, the server cannot cheat the clients about data loss. On the other, clients cannot falsely claim that their data is lost. The cryptographic evidence produced by our solution should be non-repudiable by either the client side or the server side, when it is presented to the judicator.

In addition, we propose to replace indices with more flexible coordinates using a new data structure called coordinate Merkle hash tree that optimizes both worst-case and average-case performance for data updates and integrity verification.

Even though the presentation of our integrity verification solution will be based on a file F, the notation F can be generalized to be a single file, a part of a large file, a set of files, a data stream, or a segment of a data stream. Using key-based authenticated dictionaries, the proposed solution for a file can be extended for a file system consisting of many files with a directory structure in a similar way as in [25].

3.6.2 Threat Model

Server: We define the following semi-trust model for the server. In normal cases, the server will perform operations correctly, and will not deliberately delete or modify clients' data. But because of management errors, Byzantine failures or external intrusions, the server may lose or corrupt the hosted data inadvertently. When these errors happen, the service provider may try to hide the truth of data loss.

Client: We also define a semi-trust model for the client. We assume that most clients are honest but some may behave untruthfully. For example, some clients may falsely claim data loss in order to damage the reputation of a cloud service provider (possibly backed by competitors) or to blackmail the provider.

Judicator: We assume that the judicator is a honest third party which is trusted by both the client and the server.

3.6.3 Interaction Between Client and Server

Before we present the implementation of our solution, we give an overview of the interaction between the client and the server in the form of thirteen basic algorithms.

- $Gen(1^k) \rightarrow (pk, sk)$ is the algorithm in the digital signature solution Π defined by [35]. Gen is executed by both the client and the server to produce a pair of public and private keys. The client stores its private key sk_c and sends the public key pk_c to the server. The server stores its private key sk_s and sends the public key pk_s to the client.

- $Sign_{sk}(m) \rightarrow \sigma$ is the algorithm in the digital signature solution Π. It takes the private key sk and a message m as input, and outputs a signature σ on the message using the private key. In our solution, the client and the server apply this algorithm to jointly produce the signature $\sigma_{\mathcal{M}}$ of the meta data.

- $VerifySign_{pk}(m, \sigma) \rightarrow (TRUE, FALSE)$ is the algorithm in the digital signature solution Π. It takes as input the message and the signature. It verifies the correctness of the signature using the public key and outputs the result of the verification. It is used by both the client and the server to verify the authenticity of the meta data.

- $Prepare(sk_c, F) \rightarrow (\Phi, CMHT, Sign_{sk_c}(I(w_r)||T||n))$ is an algorithm run by the client. It takes as input the client's private key sk_c and a data file $F = \{m_i\}$. The output contains (1) a set of block signatures, $\Phi = \{\sigma_i\}$, (2) a CMHT constructed based on the block tags $\{t_i\}$, where $t_i = H(m_i)||c_i$, and (3) a digital signature on the meta data, $Sign_{sk_c}(I(w_r)||T||n)$. The client sends the output and F to the server.

- $GenMeta(sk_s, pk_c, sig_{sk_c}(I(w_r)||T||n)) \rightarrow \mathcal{M}_s$ is executed by the server. After verifying the correctness of the signature $Sign_{sk_c}(I(w_r)||T||n)$ using pk_c, the server signs the signature using its private key sk_s: $\sigma_{\mathcal{M}} = Sign_{sk_s}(Sign_{sk_c}(I(w_r)||T||n))$. Then the server sends the meta data $\mathcal{M}_s = \{I(w_r), T, n, \sigma_{\mathcal{M}}\}$ to the client.

- $Contract(pk_c, pk_s, \mathcal{M}_s, I(w_r)||T||n) \rightarrow \mathcal{M}_c$ is an algorithm run by the client. After verifying the correctness of the signature $\sigma_{\mathcal{M}}$ using pk_s and pk_c, the client deletes all local copies of data and only stores a copy of meta data $\mathcal{M}_c$, identical to $\mathcal{M}_s$.

- $GenChallenge(n) \rightarrow \mathcal{R}_k$ is an algorithm executed by the client. It takes n as input, and outputs a request $\mathcal{R}_k$ which contains a set of k randomly-selected coordinates as well as k randomly-selected constants. The client sends $\mathcal{R}_k$ to the server and asks the server to return a proof that it has the blocks whose coordinates are in $\mathcal{R}_k$.

- $GenProof(\mathcal{R}_k, CMHT, F, \Phi) \rightarrow P$ is executed by the server after receiving $\mathcal{R}_k$. The input contains the request $\mathcal{R}_k$, the CMHT, the file F, and the block signatures Φ. The server returns a proof P that allows the client whether it indeed has the blocks in the $\mathcal{R}_k$.

- $VerifyProof(\mathcal{R}_k, P, \mathcal{M}_c) \rightarrow (TRUE, FALSE)$ is an algorithm executed by the client. After receiving the proof P, the client can verify if the server possesses the blocks in $\mathcal{R}_k$ based on its meta data. It outputs $TRUE$ if the proof P passes the verification. Otherwise, it returns $FALSE$.

- $Judge(pk_s, pk_c, \mathcal{E}_c, \mathcal{E}_s) \rightarrow (ClientWin, ServerWin)$ is an algorithm executed by a judge during litigation after the client detects data loss but the server disputes that. It takes as input the public keys of the server and the client, the evidence from the client, $\mathcal{E}_c = \{\mathcal{R}_k, \mathcal{M}_c\}$, and the evidence from the server, $\mathcal{E}_s = \{P, \mathcal{R}_k, \mathcal{M}_s\}$. The requests $\mathcal{R}_k$ in both $\mathcal{E}_c$ and $\mathcal{E}_s$ must be the same. It decides whether the client wins or the server wins.

- $UpdateRequest() \rightarrow \mathcal{R}_U$ is an algorithm executed by the client. It outputs an update request $\mathcal{R}_U$ which contains an update $Order \in \{Modify, Insert, Delete\}$ and a block

Location in form of data key or byte offset. If the $Order$ is $Modify$ or $Insert$, $\mathcal{R}_U$ should contain a new block m^* and its signature σ^*.

- $Update(F, \Phi, CMHT, \mathcal{R}_U) \rightarrow P_{update}$ is an algorithm run by the server. After receiving the update request $\mathcal{R}_U$ from the client, the server takes F, Φ, and the CMHT as input. It performs the update, outputs a proof P_{update}, and sends the proof back to the client.

- $VerifyUpdate(P_{update}) \rightarrow (TRUE, FALSE)$ is executed by the client. It takes the proof P_{update} as input and outputs $TRUE$ if the the proof passes the verification. Otherwise, the client will return $FALSE$.

3.6.4 Solution Details

Our solution has three components. 1) *Preprocessing*: Before outsourcing a file to the server, the client generates the meta data that it keeps locally as well as the information that it outsources to the server together with the file. 2) *Data-possession Verification*: After outsourcing, the client will periodically check the integrity of its remotely-stored data.

Preprocessing. It includes four algorithms: Gen, $Prepare$, $GenMeta$ and $Contract$. Before outsourcing data to the server, the client generates the set Φ of block signatures and the CMHT. It agrees on a time stamp T with the server, and produces the meta data. Then, it outsources F, Φ, and the CMHT to the server, only keeping the meta data locally.

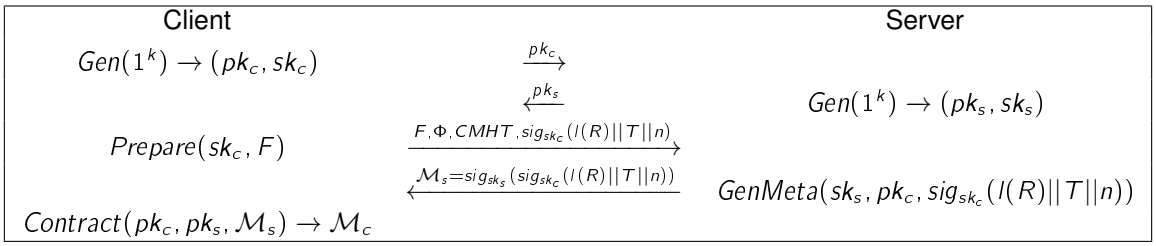

Figure 3-5. The procedure of preprocessing

Data-possession verification. It is performed periodically, including four algorithms: $GenChallenge$, $GenProof$, $VerifyProof$ and optionally $Judge$. The client queries the server with a randomly-choosing subset of coordinates, and the server

generates a proof in response. After verifying the proof, the client will return $TRUE$ or $FALSE$.

Figure 3-8 shows the procedure of data-possession verification.

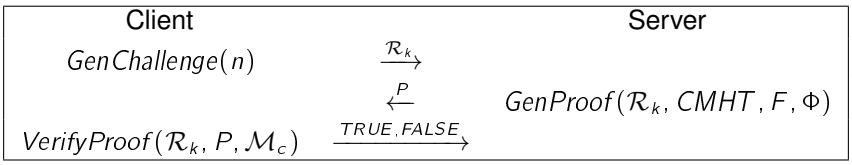

Figure 3-6. The procedure of data-possession verification

- *Generate a Request*: Knowing the value of n, the client knows the exact shape of the complete tree of CMHT. Hence, it knows all valid coordinates for leaf nodes. The client randomly selects $k(<< n)$ leaf nodes, whose coordinates are denoted as $\{c_{i_1}, c_{i_2}, \ldots, c_{i_k}\}$, for data blocks $\{m_{i_1}, m_{i_2}, \ldots, m_{i_k}\}$ that are represented by the selected leaf nodes. For convenience, we let $\Omega = \{i_1, i_2, \ldots, i_k\}$, and the set of selected coordinates is $\{c_i \mid i \in \Omega\}$. The corresponding data blocks are $\{m_i \mid i \in \Omega\}$. The client generates a request $\mathcal{R}_k = \{(c_i, v_i) \mid i \in \Omega\}$, where v_i is a constant. It sends $\mathcal{R}_k$ to the server.

- *Generate a Proof*: After receiving the request, the server generates a proof P. It computes
$$\mu = \sum_{i \in \Omega} v_i m_i, \qquad \sigma = \prod_{i \in \Omega} {\sigma_i}^{v_i} \mod N,$$
where σ_i is the block signature of m_i.
The proof P sent to the client consists of μ, σ, and a partial CMHT, denoted as Γ, consisting of the leaf nodes w_j with the selected coordinates c_i and the siblings of the nodes on the paths from the root to w_j. For each node in Γ, the server only needs to send the label of the node. If a node is a leaf, the label is simply the tag of the block represented by the leaf. Hence, the tags t_i of blocks with coordinates c_i are carried by Γ. For each coordinate c_i, the client produces a message $M_i = \{t_i, \gamma_i\}$, where $t_i = H(m_i)$ is the block tag whose coordinate is c_i, and $\{\gamma_i\}$ is the set of labels or block tags of the node siblings on the path from the leaf to the root. Note that if the sibling is a leaf node, γ_i is the block tag. Otherwise, γ_i is the label. For example, in Figure 3-3, if $c_i = c_1 = 001$, the message will be $M_1 = \{t_1, \gamma_1\}$, where $\gamma_1 = \{t_2, t_5, I(w_c)\}$.
Then the server sends the block tag set $S = \{H(m_i) : i \in I\}$ to the client, and generates a sequence of messages for each block tag $H(m_i)$ in S to prove its index number. Suppose $\{w_1, w_2, w_3, \ldots, w_h, w_{h+1}\}$ denotes the path from the root node to the element $H(m_i)$, where $i \in [i_1, i_k]$, h is the height of the CMHT and the node w_j is the parent of w_{j+1}. For each node w_j, $j \in [1, h]$, the server will provide a message M_j. With this message, the client can easily compute the value of w_j and eventually, the client can compute the value of the root node w_1.

- *Verify*: After receiving the proof P, the client will run the algorithm $VerifyProof$ to check the correctness of the proof. The client first verifies the integrity of the partial CMHT by the standard Merkle tree operations, which ensures the correctness of the tags t_i carried by the leaf nodes of Γ. The client then checks whether the following equation holds:

$$\sigma = (\prod_{i \in \Omega} {t_i}^{v_i}) \cdot g^{\mu} \mod N. \tag{3–1}$$

  Then the client will verify the correctness of the tags by checking the relationship between the tags and their coordinate values. As the CMHT is a complete binary tree, and the client stores the total number of blocks, n, it is easy for the client to determine the shape of the CMHT. For each coordinate value, the client can also uniquely determine a partial CMHT from the root to the leaf. Accordingly, based on each message, the client can compute the label values from the leaf to the root. If the label of the root is consistent with the $l(w_r)$ in $\mathcal{M}_c$, the coordinate is correct. Otherwise, the client will generate an evidence $\mathcal{E}_c$ and apply a judgment to the judicator.

- *Judge*: If the client detects data loss through $VerifyProof$ but the server disputes it, they may present their evidences to a court where $Judge$ is executed. After receiving the evidence $\mathcal{E}_c = \{\mathcal{R}_k, \mathcal{M}_c\}$ from the client and the evidence $\mathcal{E}_s = \{P, \mathcal{M}_s\}$ from the server, the judge first verifies the correctness of the proof P through $VerifyProof$ based on information from the client. Then it checks the signatures in $\mathcal{M}_c$ and $\mathcal{M}_s$. If both signatures are correct, it compares the time stamps to determine whose evidence is valid. Depending on whose time stamp is more recent, the judge decides the winner based on the algorithm in Fig. 3-7.

Updating. If the client wants to update a block, it first runs algorithm $UpdateRequest() \rightarrow \mathcal{R}_U$ to generate an update request and send the request to the server. Upon receiving the request, the server will update the block and execute the algorithm $Update(F, \Phi, CMHT, \mathcal{R}_U)$ to generate a proof P_{update}. The client will use the algorithm $VerifyUpdate(P_{update})$ to verify the update. If the update is correct, the client and the server will agree on a new meta data using algorithms $GenMeta$ and $Contract$.

- *Modification*: Suppose a client wants to change a data block m_i to m^*. It generates an update request $\mathcal{R}_U = \{Modify, Location, m^*\}$, and sends it to the server, which will replace the old data m_i with the new one m^*. Let c_i be the coordinate of m_i. The server constructs a partial CMHT (denoted as Γ) to the leaf node w_j at coordinate c_i, updates w_j with a new label $H(m^*)||c_i$, and recomputes the labels of the nodes on the path from w_j to the root. Let $l(w_{r'})$ be the new label of the root.

Input: pk_c, pk_s, $\mathcal{E}_c = \{\mathcal{R}_k, \mathcal{M}_c\}$, $\mathcal{M}_c = \{l_c(w_r), T_c, n_c, \sigma_{\mathcal{M}_c}\}$,
$\mathcal{E}_s = \{P, \mathcal{M}_s\}$, $\mathcal{M}_s = \{l_s(w_r), T_s, n_s, \sigma_{\mathcal{M}_s}\}$

---

1. **if** ($VerifyProof(\mathcal{R}_k, P, \mathcal{M}_c) = TRUE$)
2. **return** server as the winner;
3. **else**
4. **if** ($VerifySign_{pk_s,pk_c}(\sigma_{\mathcal{M}_c}, l_c(w_r)||T_c||n_c) = FALSE$)
5. **return** server as the winner;
6. **if** ($VerifySign_{pk_s,pk_c}(\sigma_{\mathcal{M}_s}, l_s(w_r)||T_s||n_s) = FALSE$)
7. **return** client as the winner;
8. **else**
9. **if** ($\sigma_{\mathcal{M}_c} = \sigma_{\mathcal{M}_s}$)
10. **return** client as the winner;
11. **else if** ($T_s > T_c$)
13. **return** server as the winner;
14. **else**
15. **return** client as the winner;

Figure 3-7. Algorithm for Judge

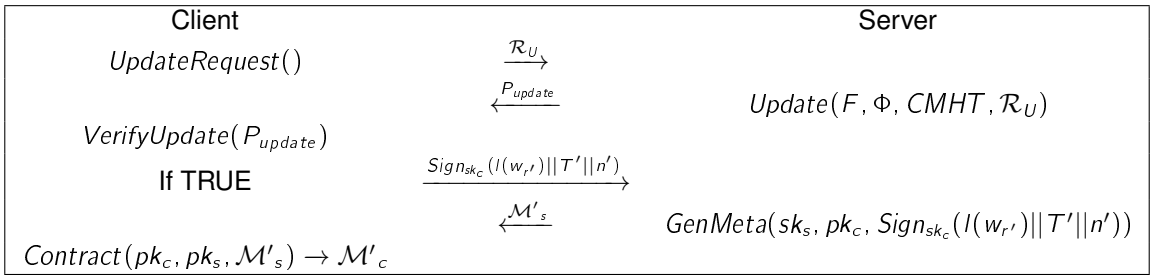

Figure 3-8. The procedure of data-possession verification

After that, the server generates a proof $P_{update} = \{\Gamma, c_i, l(w_{r'})\}$, where Γ is the partial CMHT to w_j *before modification*.
To ensure that Γ has the correct leaf node w_j for m_i, the client checks whether the received label of w_j (before modification) contains $H(m_i)$ and verifies the labels of the partial CMHT using the Merkle tree operations. Then it performs what the server does: replacing the label of w_j with $H(m^*)||c_i$ and re-computing the labels of nodes on the path to the root, whose new label is denoted as $l_c(w_{r''})$. The modification is successful only if $l_c(w_{r''})$ is equal to the received value of $l(w_{r'})$. In this case, the client will send the new homomorphic signature σ^* to the server and generate a new meta data with the server, where $\sigma^* = H(m^*)||c_i \cdot g^{m^*} \mod N$.

- *Insertion*: Suppose a client wants to insert a new block m^*. It will inform the server to insert m^* into the file at a specified offset location, insert a corresponding leaf node w^* into the CMHT, and update the meta data. The location in the CMHT where w^* will be inserted is determined by finding the coordinate c of the leftmost leaf node w_i with minimum level, where the *level* of a node is defined as the length of the path from the node to the root. We call w_i the *split node*; its location is where

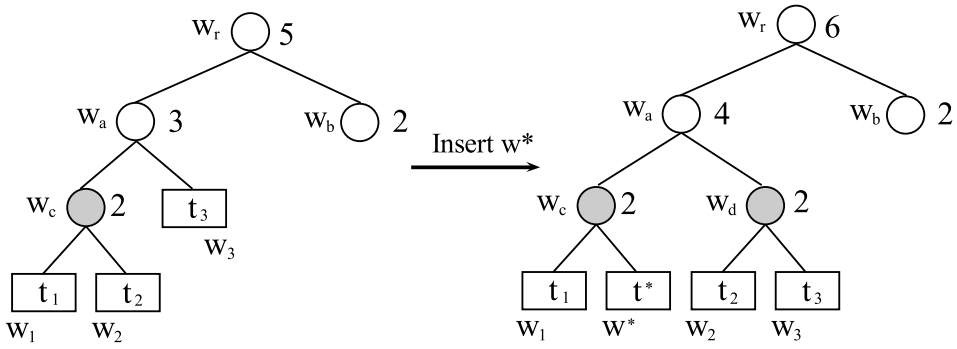

Figure 3-9. Insert a new leaf node w^* into the CMHT, where w_3 is the split node

we will insert w^*. See the left plot of Figure 3-9 for an example. Both the server and the client can independently determine c. Recall that the client knows the shape of the complete binary tree based on the value of n.
The client generates an update request $\mathcal{R}_U = \{Insert, Location, m^*, \sigma^*\}$, and send it to the server. After receiving the request, the server will insert the block into the file, constructs a partial CMHT (denoted as Γ) to the split node w_i at coordinate c, and then inserts a leaf node w^* as follows: replacing w_i with a new internal node w_d, and making w_i and w^* to be the left and right children of w_d, respectively. (See Figure 3-9 for a simple, illustrative example.) The server adds m^* and w^* into its data structure that maps between data blocks and their leaf nodes in the CMHT. Finally, it updates the labels of all nodes on the path from w^* to the root. Let $l(w_{r'})$ be the new label of the root.
The server generates a proof $P_{update} = \{\Gamma, l(w_{r'})\}$ and sends it to the client, where Γ is the partial CMHT to the split node before insertion. For each node in the partial CMHT, the server only needs to send its label. Recall that the client can independently determine c and thus know the exact shape of this partial CMHT. Using the labels of the nodes and following the standard Merkle tree operations, the client can verify the integrity of the partial CMHT. Next, the client re-performs the same insertion as the server does, and re-computes the label of the root $l(w_{r''})$ based on the partial CMHT after insertion. The insertion is successful only if $l(w_{r'})$ equals to $l(w_{r''})$. In this case, the client will agree on a new time stamp T' with the server, and together they will generate a new meta data with the new root label $l(w_{r'})$.

- *Deletion*: Suppose a client wants to delete a data block m_i. It sends an update request $\mathcal{R}_U = \{Delete, Location\}$ to the server, which deletes m_i from the file, reconstructs a partial CMHT (denoted as Γ) to a leaf node w_j which represents m_i at a certain coordinate c_i, and deletes w_j using the following algorithm: Let w_k be w_j's sibling node. There are two cases. (1) If w_j or w_k is the rightmost leaf node at the highest level, the server deletes w_j and replaces w_j's parent with w_k. (2) Otherwise, it finds the rightmost leaf node w' at the highest level, and expands Γ to that node. In this expanded partial CMHT that covers both w_i and w', the server replaces the parent of w' with its sibling and then moves w' to the location of w_j

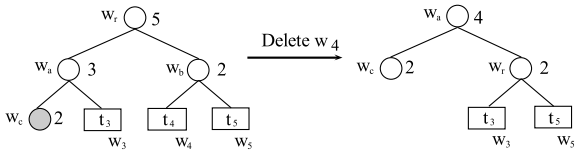

Figure 3-10. Delete the leaf node w_4 from the CMHT

after removing w_j from the tree. Hence, we call w' the *replacement node*. (See Figure 3-10 for a deletion example.) The server re-computes the labels in the partial CMHT from leaf(s) to the root. Let $l(w_{r'})$ be the new root value. Next, the server generates a proof $P_{update} = \{\Gamma, l(w_{r'})\}$, where Γ is the partial CMHT to w_j (possibly also to w') before deletion. Knowing the shape of the CMHT based on the value of n, the client can independently determine the coordinate of the replacement node for case (2) of the deletion algorithm. After receiving the proof, the client first verifies whether the label of w_j contains $H(m_i)$, verifies if the coordinate of the replacement node (if there is one in Γ) is correct, and then verifies the integrity of Γ through the Merkle tree operations. Finally, it performs the same deletion algorithm as the server does. Let $l_c(w_{r''})$ be the new label of the root that the client computes. The deletion is successful only if $l_c(w_{r'})$ equals to $l(w_{r''})$. In this case, the client will generates a new meta data with the server.

3.6.5 Client Caching

To improve the client's performance, we may cache the upper levels of the tree on the client side. Due to the nature of a complete binary tree structure, the upper levels account for a very small fraction of the whole tree, but it can significantly reduce the amount of computation and communication between the client and the server. For a file of 10^6 data blocks, if we cache the top 10 levels of CMHT at the client side (which accounts for less than 0.1% of the whole CMHT tree), we can reduce the communication overhead by half.

3.6.6 Security Analysis

Security definition. The proposed solution achieves the following security properties.

- *Storage Integrity*: with Theorem 3.1, we prove that our solution can detect loss of clients' data stored on the cloud. That is, if the cloud server loses data blocks and the client's request includes any of those blocks, with almost certainty the client will detect the loss by observing the proof from the server fails the verification.
- *Non-repudiable evidence against the client's false claim*: with Theorem 3.2, we prove that our solution can also supervise the behavior of the client. That is, if a

client makes false claim about data loss, the server can produce non-repudiable evidence that causes the $Judge$ algorithm by the judicator to always declare the server is the winner, regardless of what evidence the client provides.

Security proofs.

Theorem 3.1. *Suppose factoring $N = pq$ is polynomially infeasible for two sufficiently large primes p and q. Given a client request $\mathcal{R}_k$, if the server does not possess one or more data blocks whose coordinates belong to $\mathcal{R}_k$ (due to data loss or corruption), the probability for a proof P produced by the server in polynomial time to pass the client's integrity check $VerifyProof(\mathcal{R}_k, P, \mathcal{M}_c)$ is negligibly small.*

Proof. We prove the theorem by contradiction. Recall that $R_k = \{(c_i, v_i) \mid i \in \Omega\}$, and we use m_i to denote the data block that is represented by a leaf node in CMHT whose coordinate is c_i. Assume (1) the server does not possess one or more blocks in $\{m_i \mid i \in \Omega\}$, and yet (2) it has a polynomial method to produce a proof $P = (\mu, \sigma, \Gamma)$ that can pass the client's integrity check with non-negligible probability,

$$\sigma = (\prod_{i \in \Omega} t_i^{v_i}) \cdot g^{\mu} \mod N, \tag{3–2}$$

where $t_i = H(m_i)||c_i$. The integrity of the tags t_i is protected by the Merkle tree operations performed on Γ. Hence, unless the server has a way to break the collision-resistant hash function used by the CMHT, the correct tags for blocks in $\{m_i \mid i \in \Omega\}$ must be used in (3–2). This prevents the server from using other blocks not in $\{m_i \mid i \in \Omega\}$ and their tags to produce a proof that would make (3–2) hold.

Recall that μ is supposed to be set to $\sum_{i \in \Omega} v_i m_i$. But because the server does not have one or more of these blocks, it does not know the value of $\sum_{i \in \Omega} v_i m_i$. Hence, the probability for a chosen value μ to equal $\sum_{i \in \Omega} v_i m_i$ will be negligibly small if the data blocks are sufficiently large. In other words, with high probability,

$$\mu \neq \sum_{i \in \Omega} v_i m_i. \tag{3–3}$$

By definition, $\sigma = \prod_{i\in\Omega} {\sigma_i}^{v_i} \mod N$, and $\sigma_i = t_i \cdot g^{m_i} \mod N$. Applying them to (3–2), we have

$$\prod_{i\in\Omega}(t_i \cdot g^{m_i})^{v_i} = (\prod_{i\in\Omega} {t_i}^{v_i}) \cdot g^{\mu} \mod N$$

$$\prod_{i\in\Omega} g^{v_i m_i} = g^{\mu} \mod N$$

$$g^{\sum_{i\in\Omega} v_i m_i} = g^{\mu} \mod N.$$

That, together with (3–3), means we have found $\mu - \sum_{i\in\Omega} v_i m_i \neq 0$ such that $g^{\mu-\sum_{i\in\Omega} v_i m_i} = 1 \mod N$. Therefore, $\mu - \sum_{i\in\Omega} v_i m_i$ can be used to factor N, following Miller's Lemma [44].

The above analysis shows that if the server has a polynomial method to produce a proof that passes integrity check with non-negligible probability, then that method can factor N with non-negligible probability, which contradicts the theorem assumption that factoring a large integer N is polynomially infeasible. □

Theorem 3.2. *If the client makes false claim about data loss, the server is able to provide non-repudiable evidence that the client have lied.*

Proof. Let the client request be $R_k = \{(c_i, v_i) \mid i \in \Omega\}$, and we use m_i to denote the data block that is represented by a leaf node in CMHT whose coordinate is c_i. If the server possesses all blocks in $\{m_i \mid i \in \Omega\}$, it can correctly calculate $\mu = \sum_{i\in\Omega} v_i m_i$. The server can also correctly compute $\sigma = \prod_{i\in\Omega} {\sigma_i}^{v_i} \mod N$. Note that the correctness of the tags $t_i = H(m_i)||c_i$ and the signatures $\sigma_i = t_i \cdot g^{m_i} \mod N$ is verifiable by the server based on m_i. Hence, we have

$$\sigma = \prod_{i\in\Omega} {\sigma_i}^{v_i} = (\prod_{i\in\Omega} {t_i}^{v_i}) \cdot g^{\sum_{i\in\Omega} v_i m_i} = (\prod_{i\in\Omega} {t_i}^{v_i}) \cdot g^{\mu} \mod N.$$

Moreover, the correctness of the CMHT is also completely verifiable by the server through hashing. The server will not sign the meta data at any time when it finds that

the CMHT fails the integrity check based on its meta data $\mathcal{M}_s$. Hence, the partial tree Γ produced by the server will also pass the Merkle-tree operations.

In order for the client to make a successful false claim, it has to make sure that $Judge(pk_s, pk_c, \mathcal{E}_c, \mathcal{E}_s)$ does not execute Line 2, 5 or 13 in Figure 3-7 because otherwise the server will be declared as the winner. To avoid Line 2, the client must provide a false meta data $\mathcal{M}_c$, which is different from $\mathcal{M}_s$, because the latter (together with the proof P also provided by the server) will pass the data-possession verification in Line 1 as we have argued previously. Hence, $\mathcal{M}_c \neq \mathcal{M}_s$.

To avoid Line 5, the client must provide $\mathcal{M}_c$ that was signed by the server such that the signature verification in Line 5 can pass.

The server has signed both $\mathcal{M}_s$ and $\mathcal{M}_c$. Because the server always keeps the latest meta data as $\mathcal{M}_s$, $\mathcal{M}_c$ must have been signed earlier with a smaller timestamp. In this case, Line 13 will be executed, which still declares the server as the winner. Therefore, the evidence provided by the server, $\mathcal{E}_s = \{P, \mathcal{R}_k, \mathcal{M}_s\}$, will non-repudiably result in the judicator declaring the server as the winner when the client makes false claim, regardless of what evidence the client will provide. □

3.6.7 Evaluation

We evaluate the performance of our solution in two parts. First, we evaluate the performance of our solution without client-side cache. Compared with the DPDP in [25][1] , we show that our solution has better performance in both the communication overhead and the computational overhead. Then, we add client-side cache into our solution. Comparing with the result without the client-side cache, we show that the

[1] We don't compare our solution with MHT in [58] because as we mentioned in Section 3.2, MHT in [58] is an unbalanced binary tree while the CMHT is a complete binary tree. So the performance of CMHT will definitely be better than that of MHT.

client-side cache can reduce communication and computational overhead with small storage space requirement.

Note that we don't compare CMHT with CMBT in [45] and the MHT in [58] because: on the one side, the performance of DPDP is better than that of the CMBT. On the other side, as the MHT in is an unbalanced binary tree while the CMHT is a complete binary tree. So the performance of CMHT will definitely be better than the MHT.

Our experiments are performed on a desktop computer with Intel Core i7-3770 @3.40 GHz, 8 GB RAM, and a 2TB hard driver. Algorithms are implemented using C++. We implement the block signature and the hash function using the crypto library of OpenSSL version 1.0.1 [1].

The size of the file used in our experiments is 1GB. After dividing the file into blocks, we measure the communication and computational overhead incurred at the client side and the server side for performing data-possession verification and update operations. We evaluate the performance of our solution and compare it with DPDP under different block sizes. We let the client cache the upper half of the levels in the CMHT or in the skiplist of DPDP. The cached data is about 0.1% of the tree (or skiplist). For CMHT, when the block size is 1024KB, the total cached data is just 1.25KB; when the block size is 2KB, the total cashed data is 80KB, which is still very small comparing with the file size of 1GB. For DPDP, the amount of cached data is larger because each node in skiplist needs to store an extra rank number.

Note that DPDP does not address false client claims as our solution does. This is a qualitative difference not included in the quantitative comparison below.

Communication overhead. We first compare our solution and DPDP in terms of communication overhead for data-possession verification over a request $\mathcal{R}_k$ for k data blocks. As proved in [5], detecting a 1% file corruption with 99% confidence needs querying a constant number of 460 blocks. So we set $k = 460$. The dominating overhead is the proof sent from the server to the client. It is not affected by the number

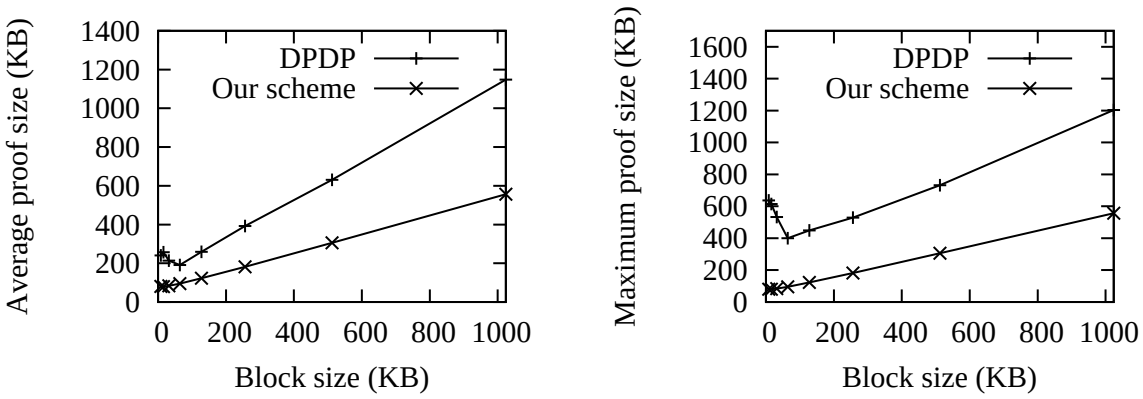

Figure 3-11. Comparing our solution (CMHT) and DPDP in terms of average or maximum communication overhead for data-possession verification with client cache

of corrupted blocks at the server. We measure both average overhead and maximum overhead. The former is the average over 100 independent runs, each run verifying 460 random selected blocks. The latter is the overhead for the case where $\mathcal{R}_k$ contains 460 leaf nodes with highest levels.

We present the experimental result in Figure 3-11. The x-axis shows the block size in KB. The y-axis shows the communication overhead in KB. The left plot presents the average overhead, and the right plot presents the maximum overhead. The overhead of our solution is consistently less than half of the overhead in DPDP. More specifically, our average overhead is $31\% \sim 50\%$ of DPDP's in the left plot, and our maximum overhead is $13\% \sim 46\%$ of DPDP's in the right plot. When the block size is 2KB, our solution reduces the average (maximum) overhead by 69% (87%) when comparing with DPDP.

Next, we measure the communication overhead between the client and the server for updating a data block. It includes all information sent by *UpdateRequest*, *Update*, and *VerifyUpdate*. We perform query, insert, delete, and modify once for every data block of the file to measure the average communication overhead. For all operations, when we work on one block, all other data blocks of the file are assumed to be present in the server, and so does their corresponding leaf nodes in CMHT. (Using deletion as an example, we will delete one block at a time. Before we delete the next block, we put

back the previously deleted one.) For insertion and modification, we do not account the transmission of the new block as overhead.

The experimental results are shown in Figure 3-12. The x-axis is the block size in logarithmic scale. The y-axis is the communication overhead in KB. The average overhead of our solution is significantly lower than that of DPDP — less than one third of it when the block sizes are relatively small. The gap for the maximum communication overhead is even larger, which we omit due to space limitation.

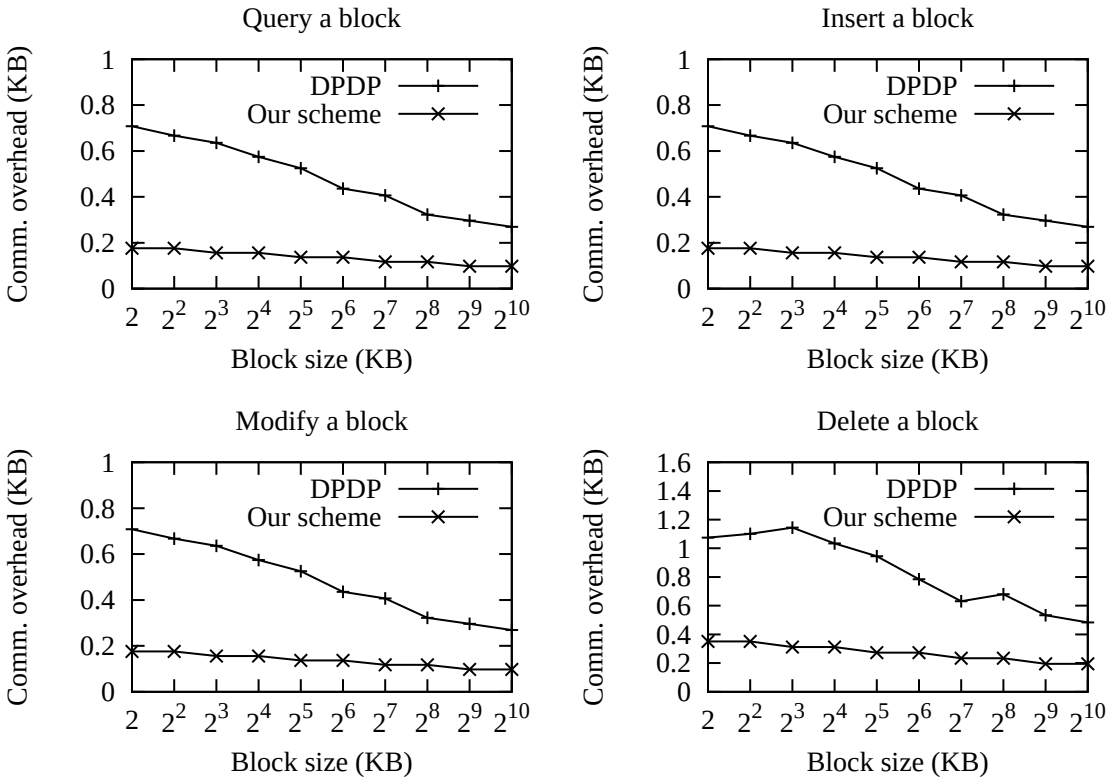

Figure 3-12. Comparing our solution and DPDP in terms of average communication overhead for updating a block with client cache

Computational overhead.

We measure the computational overhead for a client to verify a proof returned from the server for data-possession verification, including both the verification of (3–1) and the Merkle tree operations on the partial CMHT tree. We make 100 randomly-generated data possession verification requests and measure the average and maximum computational overhead per request. The results are presented in Figure 3-13. The

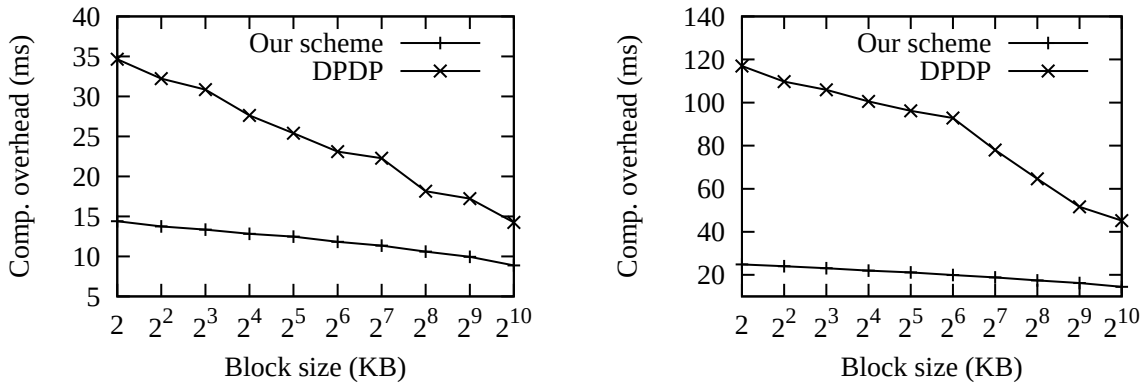

Figure 3-13. Average and maximum computational overheads by a client to verify a proof with client cache

x-axis shows the block size in logarithmic scale. The y-axis shows the computational overhead in second. Again, our solution performs better in the average case as well as the maximum case. More specifically, our solution reduces the average computational overhead by up to 58.5% and the maximum computational overhead by up to 78.7%, when comparing with DPDP.

3.7 Summary

The development of cloud storage systems brings a number of security problems. This work proposes a non-repudiable data possession verification solution that protects both the client and the server. The new solution makes sure that the server cannot cheat the client by lying about data loss, and the client cannot untruthfully claim data loss. We also design a new data structure named Coordinate Merkle Hash Tree (CMHT) to optimize the communication and computational overhead. We compare our solution with previous work through experiments, and the result shows that our solution has better performance.

CHAPTER 4
MEASUREMENT-BASED ANOMALY DETECTION IN CLOUD COMPUTING

4.1 Motivation

As the Internet moves into the era of big network data, it presents both opportunities and technical challenges for data flow measurement at both the core and the edge of the networks. This work focuses on a particular measurement function, counting *the number of distinct elements in each flow*, which is traditionally referred to as *flow cardinality* or *flow spread*. Flows and elements can be flexibly defined depending on application context. A few examples are given below.

- For scan detection, we can define each flow as all packets from the same source address and its elements as the destination addresses in the headers of the packets. The flow from a scanner has a large cardinality because it sends packets to many different destination addresses.
- For the gateway of a network to automatically identify its internal servers (possibly for resource alignment), it may regard all inbound packets to each internal address as a flow and the source addresses in the headers of the packets as elements. The flow to a server tends to have a larger cardinality than flows to other hosts because more clients communicate with the server. Moreover, as the gateway measures the cardinality of each flow periodically, it can detect potential DDoS attacks to a server if it finds the cardinality of the flow to the server jumps far beyond the normal value.
- In other applications, the flows do not have to be network traffic. For example, an online retailer may want to count the number of different customers that purchase one product after purchasing another one. These two products form a purchase association. We *logically* interpret all customers making the two purchases as a flow. Identifying purchase associations with large cardinalities help the retailer make effective followup suggestions to customers after they make their first purchases.

However, exact count for each flow will cause large memory and computation overhead. Because we count the number of *distinct* elements, in order to remove duplicates, the data structures may have to keep the elements that have been seen [63], which is costly. Note that counting the number of elements in each flow without removing duplicates is a related but different problem [39–41]. Fortunately, it is often

not necessary for applications such as those listed above. As an example, for scan detection, suppose the cardinalities of the flows from normal sources are in tens or fewer. If the cardinality of the flow from a scanner is in thousands, even with some estimation error, we can still separate it from the normal ones based on the estimated cardinality. Various methods have been proposed for estimating the cardinalities of flows [8, 27, 37, 57, 61]. One important thread of research in this area is based on sketches. The representative work includes the FM sketches [29], the LogLog sketches [24], and the HyperLogLog sketches [28], which have been implemented in real systems. In a typical implementation, a LogLog or HyperLogLog sketch uses 5 bits, and an FM sketch uses more. Multiple sketches (in hundreds) are needed for each flow to ensure estimation accuracy.

There are practical needs for reducing per-flow memory overhead in cardinality estimation. If the cardinality estimation function is implemented on a network processor chip, because the on-chip memory is typically small and the number of flows in modern networks can be very large, we will have to minimize per-flow overhead in order to accommodate more flows. In the previous application example of purchase associations, if there are hundreds of thousands of different products, the number of possible purchase associations (flows) can in tens of billions. For such a large number of flows, memory overhead may become a problem.

4.2 Related Work

This work is motivated from the famous FM sketches and probabilistic counting with stochastic averaging [29], which are designed to estimate the number of distinct elements in a multiset (or a flow in the context of this work). For each flow, the data structures consist of s FM sketches, denoted as $S[j]$, $0 \leq j < s$, each of which is a bitmap of I bits, denoted as $S[j][i]$, $0 \leq i < I$.

To record an element of the flow, we first perform a uniformly distributed hash function on the element to select one of the sketches. Without loss of generality,

we denote the selected sketch as $S[j]$. We then perform a geometrically distributed hash function on the element such that the probability for the output to be i is $2^{-(i+1)}$, $0 \leq i < l$. Let v be the output. We set the bit $S[j][v]$ to one. Duplicate elements will set the same bit and thus automatically removed since they have no impact on the values of the sketches.

After recording all elements of the flow, we can estimate the flow cardinality from the sketches as follows: Let k be the true cardinality of the flow, $\hat{k}$ be the estimated cardinality, and $f(S[j])$ be the number of leading ones in sketch $S([j])$. For example, if $S[j]$ is 1111010...0, then $f(S[j]) = 4$. A functional relation can be developed between k and the expected number of leading zeros in each sketch. Replacing the expected value with the measured average of $\frac{\sum_{j=0}^{s-1} f(S[j])}{s}$, the following estimation formula is derived in [29]:

$$\hat{k} = \frac{s2^{\frac{\sum_{j=0}^{s-1} f(S[j])}{s}}}{\theta}, \tag{4–1}$$

where $\theta = 0.77$ when k is sufficiently large. To ensure estimation accuracy, hundreds of sketches are often needed.

Instead of using the number of leading ones, the LogLog and HyperLogLog sketches [24, 28] develop their estimation formulas based on the highest index of any bit that is set to one in each sketch. Only $\log_2 l$ bits per sketch is needed to keep track of this index value. However, even though each sketch is smaller, hundreds of them are still needed to ensure accuracy.

We reduces per-flow memory usage in two ways. First, we develop virtual sketches, each of which uses no more than 2 bits on average. Second, we develop virtual sketch vectors, which are logically-separated but physically-shared data structures for a large number of different flows. Together, they can drive the memory usage down to the realm of one bit per flow.

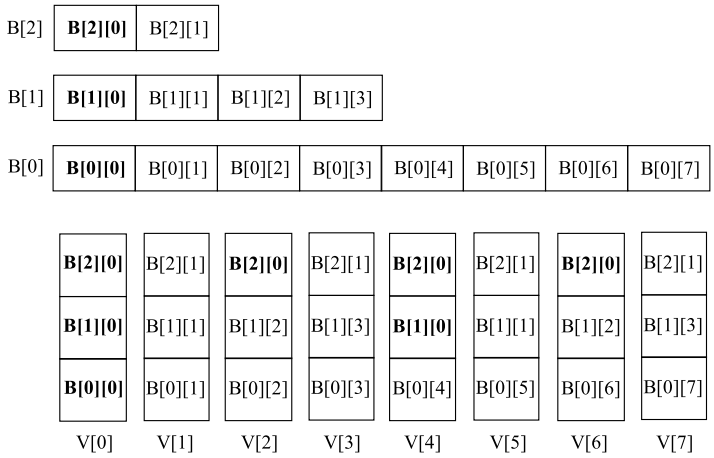

Figure 4-1. An illustrative example of constructing virtual sketches from the bit arrays with $l = 3$ and $m = 8$. The first bit in each bit array is shown in bold text. To construct virtual sketches, the bits in the arrays except for $B[0]$ must be reused. The figure shows that the bits in $B[2]$ are each used four times in the virtual sketches, and the bits in $B[1]$ are each used twice.

4.3 Virtual Sketches

4.3.1 Virtual Sketches

The available physical memory B is divided into l bit arrays, denoted as $B[i]$, $0 \leq i < l$, with the size of $B[i+1]$ being half of the size of $B[i]$. Let m be the number of bits in $B[0]$. The number of bits in $B[i]$ is $\frac{m}{2^i}$. The total number of bits in all bit arrays is $\sum_{i=0}^{l-1} m2^{-i} < 2m$.

The jth bit in $B[i]$ is denoted as $B[i][j]$, $0 \leq j < \frac{m}{2^i}$. With each bit $B[0][j]$, we construct a *virtual sketch* $V[j]$ of l bits, denoted as $V[j][i]$, $0 \leq i < l$, by taking one bit from every other array:

$$V[j][i] = B[i][j \bmod \frac{m}{2^i}]. \tag{4–2}$$

Because there are fewer bits in other arrays, their bits must be reused in multiple sketches. Figure 4-1 shows an example with $l = 3$ and $m = 8$. For instance, $V[0]$ consists of three bits, $B[0][0]$, $B[1][0]$ and $B[2][0]$, while $V[6]$ consists of three bits, $B[0][6]$, $B[1][2]$ and $B[2][0]$. They share $B[2][0]$.

In total, we construct m virtual sketches from fewer than $2m$ bits, using space fewer than 2 bits per sketch, more efficient than the existing sketches [24, 28, 29]. The m virtual sketches form a sketch pool, denoted as V.

4.3.2 Virtual Sketch Vector

For an arbitrary flow f, we select a number s of virtual sketches pseudo-randomly from the pool V to form a virtual sketch vector V_f for the flow, where s is a system parameter. For convenience, the sketches in the vector are also denoted as $V_f[j]$, $0 \leq j < s$, which will be used to record the elements in flow f.

There are different ways of selecting sketches from V to form V_f. One possible approach is described as follows: Let R be an array of s different constants that are randomly chosen. To select $V_f[j]$, we perform XOR on f and $R[j]$, and hash the result for an index to a sketch in V, where H is a hash function and f is the flow label. If the hash function requires a specific length of input and f has a different length, we can pad f or divide f into segments and XOR the segments such that the resulting length is appropriate. For simplicity, the formulas in the work assume that the hash function can take input in the length of f. Hence,

$$V[H(f \oplus R[j])] \rightarrow V_f[j], 0 \leq j < s, \tag{4–3}$$

where $\oplus$ is the XOR operator and $\rightarrow$ means "*is selected for*." An example of constructing virtual sketch vectors for two flows is given in Figure 4-2.

In (4–3), $V[H(f \oplus R[j])]$ should be $V[H(f \oplus R[j]) \bmod m]$. We omit "mod m" to simplify the notation. We will use $V_f[j][i]$ for the ith bit of $V_f[j]$, $0 \leq i < I$.

In theory, we can construct an arbitrary number of virtual sketch vectors from the same pool V to support an arbitrary number of flows. The vectors for different flows will share sketches in V. Sharing causes noise. As the elements of flow f are recorded by the sketches in vector V_f, because those sketches are also used by the vectors of other flows, it introduces noise into other vectors. The more the number of flows is, the

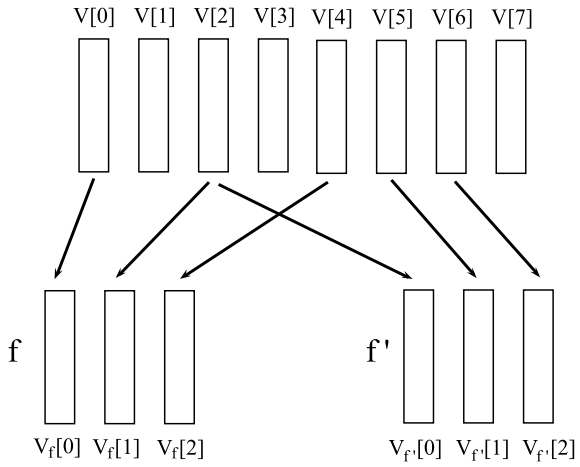

Figure 4-2. An illustrative example of constructing virtual sketch vectors from the common pool V with $s = 3$. Consider two flows, f and f'. Three sketches are randomly drawn from V to form V_f. The same happens for $V_{f'}$. The virtual sketch $V[2]$ is used in both vectors.

more the sharing of sketches will be, and the greater the noise will be. We will use the maximum likelihood method to remove the noise in each vector caused by other flows through sketch sharing.

4.4 Counting Distinct Elements in Network Flows

4.4.1 Online Operation

Consider a device processing an incoming stream of data from a large number of flows. The device may be a router processing the arrival packets which belong to different flows. A measurement function implemented on the router can provide estimations of flow cardinalities during each measurement period. The device may also be a server processing sale records and estimate the number of occurrences for each purchase association. See the introduction for application examples.

When processing the incoming data stream, the device extracts a sequence of flow/element pairs. We introduce a *recording probability* β, i.e., each flow/element pair has a probability of β to be recorded. To implement the recording probability, each flow/element pair is first sampled with a probability of $\frac{\beta}{1-2^{-l}}$; its reason will become clear shortly. Consider an arbitrary flow/element pair which is sampled. Let f be the flow label

and e be the element (which belongs to f). To record the element, the device does the following.

First, it pseudo-randomly selects a sketch from V_f. More specifically, it performs a hash $H(f \oplus e)$ in the range of $[0, s)$ and selects $V_f[H(f \oplus e)]$. If f and e have different lengths, we may pad e or divide e into segments and perform XOR on the segments such that its length is equal to that of f.

Second, the device chooses a bit from the selected sketch. To do so, it performs another hash $H'(f \oplus e)$. Let z be the number of leading zeros in $H'(f \oplus e)$. If $z \geq l$, e is not recorded. If $z < l$, the bit $V_f[H(f \oplus e)][z]$ is chosen and set to one for recording e. That is,

$$V_f[H(f \oplus e)][z] := 1, \tag{4–4}$$

where $:=$ is the assignment operator. By (4–3), it becomes

$$V[H(f \oplus R[H(f \oplus e)])][z] := 1. \tag{4–5}$$

By (4–2), it becomes

$$B[z][H(f \oplus R[H(f \oplus e)]) \bmod \frac{m}{2^l}] := 1. \tag{4–6}$$

Clearly, multiple occurrences of the same flow/element pair will cause the same bit to be set, Therefore, duplicate elements in a flow are filtered out automatically. Only distinct elements may be recorded by different bits. It is also possible that two distinct elements set the same bit. Such collision is considered in our later development of estimation formula.

We stress that V and V_f are logical concepts that will help us derive a formula for estimating the flow cardinalities. They are not physically constructed during the phase of online operation. The only physical data structures that the device maintain are $B[i]$, $0 \leq i < l$. The only operation per flow/element pair is sampling and then possibly assignment in (4–6). In the assignment, $\frac{m}{2^l}$ can be pre-computed. If m is chosen to be a

power of 2, the modulo can be accomplished by a right-shift operation. The two hashes, $H(f \oplus e)$ and $H'(f \oplus e)$, can be combined into one: Suppose the hash output of $H(f \oplus e)$ has 32 bits, $s = 256$, and $l = 8$. The first eight bits of $H(f \oplus e)$ will be sufficient for selecting a sketch from V_f. The remaining 24 bits can substitute $H'(f \oplus e)$ for selecting a bit from the sketch; in fact, only 7 bits are needed since $l = 8$. In summary, the computation of (4–6) requires two hashes and one memory access, plus some simple operations such as XOR and shift.

The probability for $z = i, \forall i \geq 0$, is $2^{-(i+1)}$. The probability for $z \geq l$ is 2^{-l}. Hence, the probability for e to be recorded is the sampling probability $\frac{\beta}{1-2^{-l}}$ multiplied by $(1 - 2^{-l})$, which gives β.

The probability for e to set the ith bit of a particular sketch, denoted as p_i, is

$$p_i = \frac{\beta}{1 - 2^{-l}} \times \frac{1}{s} \times 2^{-(i+1)}, \tag{4–7}$$

where $0 \leq i < l$. The reason is that e has a probability of $\frac{\beta}{1-2^{-l}}$ to be sampled, a probability of $\frac{1}{s}$ to select the sketch, and finally a probability of $2^{-(i+1)}$ to select the ith bit. The value of p_i decreases exponentially with respect to i.

4.4.2 Offline Estimation Based on Maximum Likelihood Method

After processing an incoming data stream, the online device offloads the bit arrays to a server for long-term storage and offline query. It will then reset its arrays for the next data stream.

The server can estimate the number n_i of flow/element pairs recorded in B_i, $\forall i \in [0, l)$. This may be done through probabilistic counting [**?**]:

$$n_i \approx -\frac{m}{2^i} \ln V_i, \tag{4–8}$$

where V_i is the fraction of bits in B_i that are zeros.

Given a flow label f under query, the server estimates its cardinality based on the stored bit arrays B_i. In this offline operation, the server first constructs the virtual sketch

vector V_f from the bit arrays. Combining (4–2) and (4–3), for $0 \leq j < s$, $0 \leq i < l$, we have

$$V_f[j][i] = B[i][H(f \oplus R[j]) \bmod \frac{m}{2^i}]. \tag{4–9}$$

Let k be the true cardinality of flow f. Consider an arbitrary bit $V_f[j][i]$. We derive the probability for the bit to be zero, which happens when (1) none of the k elements from f causes the bit to be set as one, and (2) none of the elements from other flows causes the bit to be set. Each element of f has a probability p_i in (4–7) to set $V_f[j][i]$ as one. The probability for none of the k elements from f to set it as one is $(1 - p_i)^k$.

$V_f[j][i]$ is a bit in $B[i]$. From (4–8), there are approximately n_i elements from all flows that set bits in $B[i]$. Among them, the number from flow f is approximately

$$k \times \frac{\beta}{1 - 2^{-l}} \times 2^{-(i+1)} = \frac{k\beta 2^{-(i+1)}}{1 - 2^{-l}}$$

because each element has a probability of $\frac{\beta}{1-2^{-l}}$ to be sampled and, regardless of which sketch is selected, it has a probability of $2^{-(i+1)}$ to set the ith bit in the sketch (which is a bit in $B[i]$). Hence, the number of elements from flows other than f, denoted as n_i', is approximately

$$n_i' \approx n_i - \frac{k\beta 2^{-(i+1)}}{1 - 2^{-l}}. \tag{4–10}$$

For an arbitrary element from another flow, the chance for $V_f[j]$ to happen to be a sketch in the vector of that flow and be selected for recording the element is $\frac{s}{m} \times \frac{1}{s} = \frac{1}{m}$. Hence, the probability for none of the n_i' elements from other flows to set $V_f[j][i]$ is $(1 - \frac{1}{m})^{n_i'}$. Summarizing the above analysis, we have the following formula for the probability of $V_f[j][i]$ being zero:

$$\phi_i = (1 - p_i)^k (1 - \frac{1}{m})^{n_i'}. \tag{4–11}$$

Let $u_i = s - \sum_{j=0}^{s} V_f[j][i]$, which is the number of zeros at the ith bits of all sketches in V_f. The likelihood function of observing $\{u_i \mid 0 \leq i < l\}$ with respect to k is

$$L(k) = \prod_{i=0}^{l-1} \binom{s}{u_i} \times \phi_i^{u_i}(1-\phi_i)^{s-u_i}. \tag{4–12}$$

We find an estimated cardinality $\hat{k}$ for flow f that maximizes $L(k)$. That is,

$$\hat{k} = \max_{0 \leq k \leq K} \{L(k)\}, \tag{4–13}$$

where K is the maximum flow size of interest. We may solve (4–13) numerically through exhaustive search, which is however computationally costly. In our experiments, we adopt a bi-section search method, producing the same results as the exhaustive search: Denote the search range as $[r_1, r_2]$. Initially set $r_1 = 0$ and $r_2 = K$. Let $r_3 = \lfloor \frac{r1+r_2}{2} \rfloor$. Compute $L(r_3)$ and $L(r_3 + 1)$. If $L(r_3)$ is greater than $L(r_3 + 1)$, set $r_2 = r_3$; if $L(r_3 + 1)$ is greater, set $r_1 = r_3$. Repeat the above procedure until $r_2 - r_1 \leq 1$. Finally, set $\hat{k}$ to be the one among r_1 and r_2 that produces a larger value of the likelihood function.

4.5 Differentiated Estimation Accuracy

Consider an online monitoring application where a gateway keeps track of the external scanning activities by measuring how many distinct destination addresses that each external source has contacted during each measurement period. All packets from the same source address constitute a flow, and the elements are destination addresses in the packet headers. The gateway may have access to a list of potentially malicious external hosts based on the past results from firewalls and other intrusion detection systems. Naturally, it is desirable to improve the accuracy in flow cardinality estimation for these flows over the background of other flows from addresses not in the list.

We introduce the problem of cardinality estimation with differentiated accuracy. Let $F = \{F_0, F_1, \ldots, F_{g-1}\}$ be the set of flows to be measured, which is divided into g subsets. Assume most flows belong to the base subset F_0. Their estimation accuracy serves as a baseline. Other subsets, F_v, $0 < v < g$, have increasingly higher requirements

on estimation accuracy. Each F_v is assigned an integer *priority number* a_v such that $a_0 = 1$ and $a_v > a_{v-1}$. The differentiated accuracy requirement is that the variance of the cardinality estimation $\hat{k}$ for a flow f in F_v should be only $\frac{1}{a_v}$ of the variance if the same flow had belonged to the base subset F_0.

To meet the differentiated accuracy requirement, we propose to measure the cardinality of flow f independently for a_v times. That is, we assign a_v virtual sketch vectors to f, and every element from f is probabilistically recorded for a_v times, each time by a different vector. Each vector will give an estimation with a baseline accuracy comparable to similar flows in F_0, which have a single vector per flow. When we average the a_v estimations for flow f, the variance of the average is $\frac{1}{a_v}$ of the variance for each individual estimation. More details are given below.

We assume that the device performing online operation is able to classify the incoming sequence of flow/element pairs into the correct subsets such that each flow/element pair, f and e, is associated with a priority number a. The classification is beyond the scope of this work. As an example, network traffic may be classified by pre-set ACLs or address lists.

If $a > 1$, we logically assign multiple virtual vectors to flow f, denoted as V_f^v, $0 \leq v < a$, which are constructed as follows:

$$V[H(f \oplus R[v \times a + j])] \rightarrow V_f^v[j], 0 \leq v < a, 0 \leq j < s, \tag{4–14}$$

where R is an array of $s \times a_{g-1}$ different constants. To record e, one bit from each V_f^v will be chosen and set to one. To do so, we need another array R' of a_{g-1} different constants. The two hash functions for selecting a bit in V_f^v are $H(f \oplus e \oplus R'[v])$ and $H'(f \oplus e \oplus R'[v])$. Following a process similar to that in Section 4.4.1, to record e, we can show that the device should set the following bits:

$$B[z_a][H(f \oplus R[H(f \oplus e \oplus R'[v])]) \bmod \frac{m}{2^i}] := 1, \ 0 \leq v < a, \tag{4–15}$$

where z_a is the number of leading zeros in $H'(f \oplus e \oplus R'[v])$.

During offline estimation, given a flow label f and its priority number a, we reconstruct the sketch vectors V_f^v, $0 \leq v < a$, from (4–14), compute an estimate $\hat{k}_v$ from each V_f^v, and return the average estimate:

$$\hat{k} = \frac{\sum_{v=0}^{a-1} \hat{k}_v}{a}. \tag{4–16}$$

4.6 Experiments

4.6.1 Experiment Setup

We evaluate the proposed virtual maximum likelihood sketches (VMLS) through experiments based on real traffic traces. We obtained inbound packet header traces that were collected through Cisco's NetFlow from the main gateway at University of Florida for five days. The source address in the packet header is used as flow label, and the measurement period is one day. Hence, each flow consists of all packets from the same source address during a day. The destination address in the packet header is used as element. For each flow, its cardinality is the number of distinct destination addresses that the source has sent packets to. One application for such per-flow measurement is scan detection as explained in the introduction.

Table 4-1. Traffic trace

	flows	flow/element pairs	avg cardinality
Day 1	3,558,510	10,048,129	2.82
Day 2	6,468,158	11,886,945	1.84
Day 3	5,189,371	11,858,928	2.29
Day 4	3,582,938	9,978,131	2.78
Day 5	4,007,256	10,702,677	2.67

Some statistics of the five-day traces are shown in Table 4-1. We use one day's trace as an example: It has 5,189,371 distinct source IP addresses, meaning that there are 5,189,371 flows. It has 11,858,928 distinct source/destination pairs (flow/element pairs). Hence, the average cardinality per flow is 2.29.

We implement the online operation module and the offline estimation module of the proposed VMLS. The online operation uses a default memory space of one bit per flow, i.e., the total number of bits in all bit arrays $B[i]$, $0 \leq i < l$, is equal to the number of flows in the trace. The existing FM, LogLog, hyperLogLog sketches cannot work under such a tight memory space.

Unless specified otherwise, the parameters of VMLS are set as follows: $l = 4$, $s = 250$, and the sampling probability is 50%. Increasing l to a larger value will extend the estimation range without adding much space overhead due to the exponentially decreasing nature of additional bit arrays. However, our traces do not have many large flows, and $l = 4$ is sufficient for the experiments.

4.6.2 Estimation Accuracy

The first experiment evaluates the estimation accuracy of VMLS with a single priority group, i.e., $g = 1$. For each one-day trace, we first apply the online operation module to record the elements of the flows. We then use the offline estimation module to compute an estimated cardinality for each flow. We then plot the five-day results in Figure 4-3. Each point in the figure represents a flow. Its x coordinate is the flow's true cardinality, k. Its y coordinate is the flow's estimated cardinality, $\hat{k}$. The closer a point is to the equality line $y = x$, the better the estimation accuracy will be. From the figure, we can see that the points cluster around the equality line, indicating reasonably good estimation accuracy even under a tight space of one bit per flow.

Figure 4-4 shows the relative standard error of the estimations. Because there are too few flows for some cardinality values, we compute the relative standard error by dividing the horizontal axis into measurement bins. In each bin, we compute the difference between the true cardinality and the estimated value for each flow, divide it by the true cardinality, square the result, add these squares for all flows, divide it by the number of flows minus one, and then take the squareroot. The figure shows that the relative standard error is large for small flows, but relatively small for large flows.

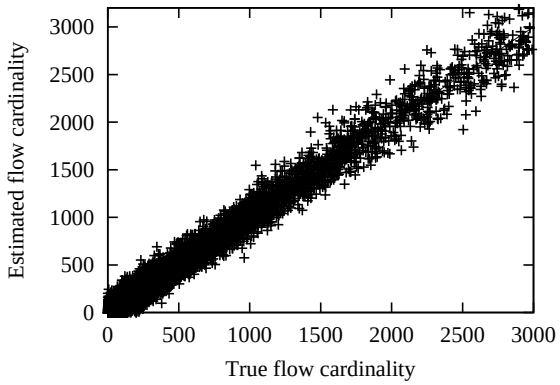

Figure 4-3. Estimation accuracy of virtual maximum likelihood sketches with a single priority in memory of 1 bit per flow

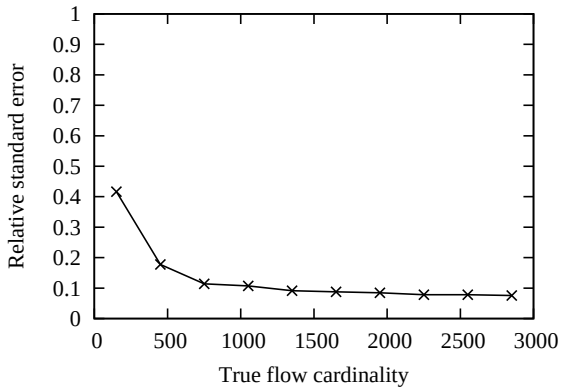

Figure 4-4. Relative standard error of the estimations with a single priority in memory of 1 bit per flow

It is below 10% for flows whose cardinalities are beyond 2,000. Even for small flows, although the relative standard error is large, the absolute error is still limited, which is evident from Figure 4-3, making it easy to separate large flows from small ones.

4.6.3 Differentiated Estimation Accuracy

The second experiment evaluates the differentiated estimation accuracy of VMLS with two priority groups, where $g = 2$, $a_0 = 1$ and $a_1 = 4$. 10% of all flows are randomly selected for the higher priority, and the remaining flows belong to the base priority. Figure 4-5 shows the estimation results of high-priority flows, and Figure 4-6 shows the results of base-priority flows. It can be seen that the estimation accuracy of the former is

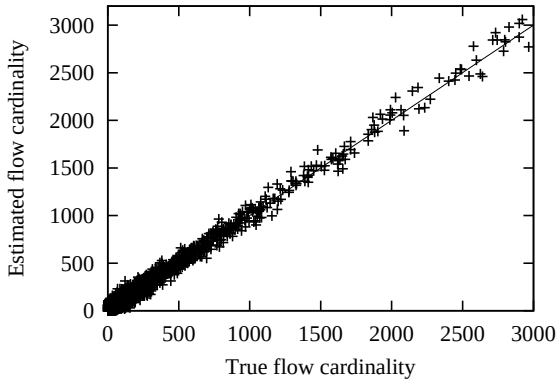

Figure 4-5. Estimation accuracy of higher priority flows with $g_1 = 4$ in memory of 1 bit per flow

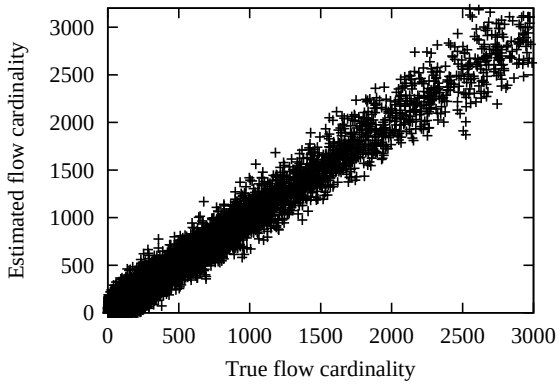

Figure 4-6. Estimation accuracy of base priority flows with $g_0 = 1$ in memory of 1 bit per flow

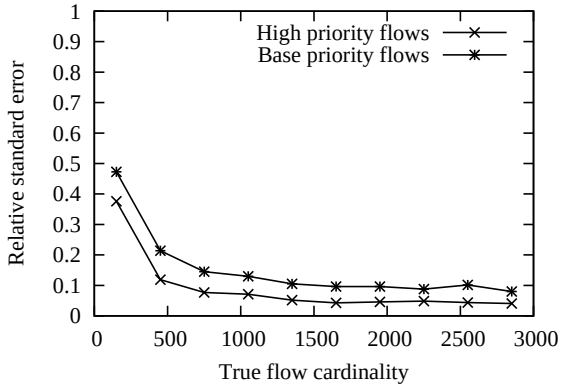

Figure 4-7. Relative standard error of the estimations with two priorities in memory of 1 bit per flow

much better than that of the latter; the points representing high-priority flows are much closer to the equality line.

A more direct comparison is given in Figure 4-7, which shows that the standard errors for high-priority flows are about half of the errors for the base-priority flows when the flow cardinalities are beyond 500. This conforms to the accuracy requirement specified by $a_1 = 4$: the variance of high-priority flows is $\frac{1}{4}$ of the variance of base-priority flows. Hence, the standard deviation (or error) of the former should be half of the latter.

4.6.4 Varied Memory Availability

The third experiment evaluates the performance of VMLS with two priority groups under different memory availability. Figure 4-8-4-10 shows the results when the memory of the online operation module is 0.5 bit per flow. Comparing with Figure 4-5-4-7, the estimation errors are considerably larger, indicating that the practical value of VMLS will begin to diminish when the available memory is too small for the online module.

Figure 4-8-4-10 shows the results when the memory of the online operation module is 3 bits per flow. Comparing with Figure 4-5-4-7, the estimation errors are measurably better. For example, for flows of cardinalities around 2,000, the relative standard error of the base priority is 9.6% under memory of 1 bit per flow, and it is lowered to 6.3% under memory of 3 bits per flow. This result indicates that better estimation accuracy can be achieved through memory increase for the online module.

4.7 Summary

This chapter proposes a new solution of virtual maximum likelihood sketches for cardinality estimation. It has four technical contributions: virtual sketches, virtual sketch vectors, a maximum likelihood method for cardinality estimation based on per-flow virtual sketch vectors, and a method to achieve differentiated estimation accuracy among multiple subsets of flows with different priorities. The experimental results based on real traffic traces demonstrate that the new solution produces cardinality estimations with reasonable accuracy in very tight memory space.

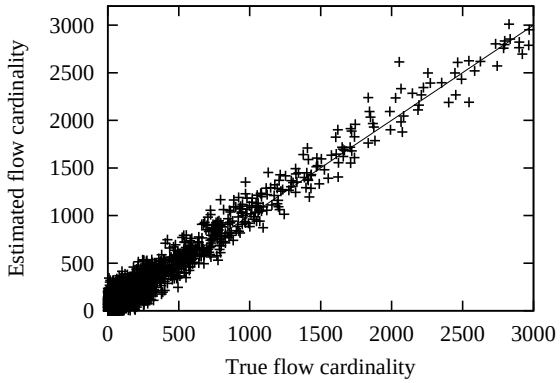

Figure 4-8. Estimation accuracy of higher priority flows with $g_1 = 4$ in memory of 0.5 bit per flow

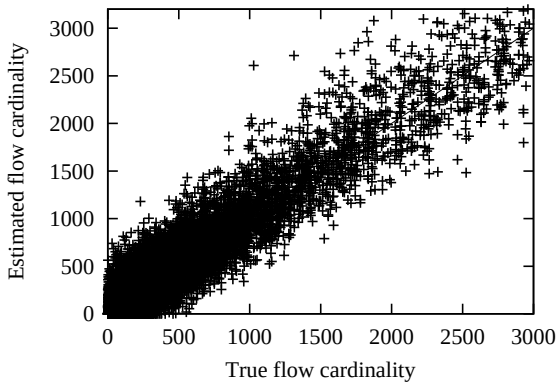

Figure 4-9. Estimation accuracy of base priority flows with $g_0 = 1$ in memory of 0.5 bit per flow

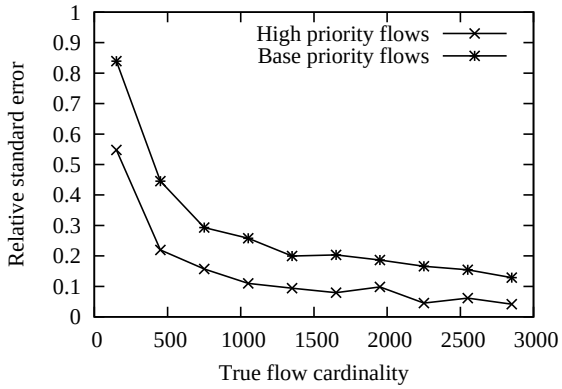

Figure 4-10. Relative standard error of the estimations with two priorities in memory of 0.5 bits per flow

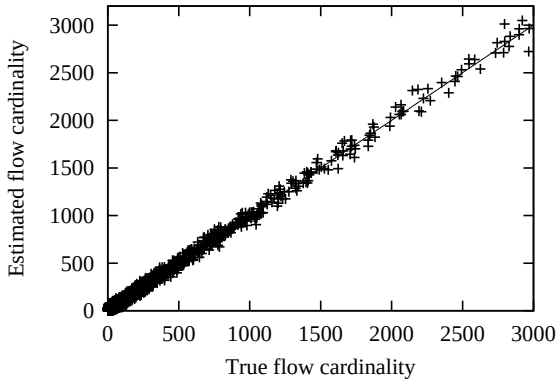

Figure 4-11. Estimation accuracy of higher priority flows with $g_1 = 4$ in memory of 3 bits per flow

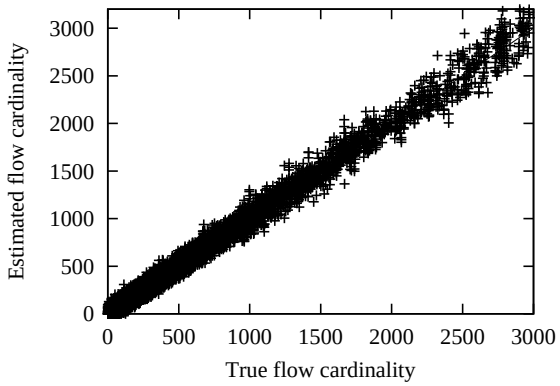

Figure 4-12. Estimation accuracy of base priority flows with $g_0 = 1$ in memory of 3 bits per flow

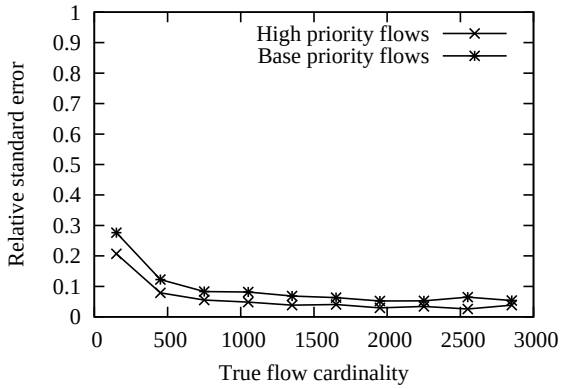

Figure 4-13. Relative standard error of the estimations with two priorities in memory of 3 bits per flow

REFERENCES

[1] "OPENSSL 1.0.1." *http://www.openssl.org/* (2013).

[2] Akl, S. and Taylor, P. "Cryptographic Solution to A Problem of Access Control in A Hierarchy." *ACM Transaction on Computer Systems* (1983).

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

[4] ———. "Validating Integrity for the Ephemerizers Protocol with CL-Atse." *Formal to Practical Security* (2009).

[5] Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., and Song, D. "Provable Data Possession at Untrusted Stores." *Proc. of CCS* (2007).

[6] Ateniese, G., Pietro, R. Di, Mancini, L. V, and Tsudik, G. "Scalable and Efficient Provable Data Possession." *Proc. of SecureComm* (2008).

[7] Ateniese, G., Santis, A. De, Ferrara, A., and Masucci, B. "Provably-secure Time-bound Hierarchical Key Assignment Schemes." *Proc. of CCS* (2006).

[8] Bar-yossef, Ziv, Jayram, T. S., Kumar, Ravi, Sivakumar, D., Trevisan, Luca, and Luca. "Counting Distinct Elements in a Data Stream." *Proc. of RANDOM: Workshop on Randomization and Approximation* (2002).

[9] Birget, J., Zou, X., Noubir, G., and Ramamurthy, B. "Hierarchy-Based Access Control In Distributed Environments." *Proc. of ICC* (2001).

[10] Boneh, D., Lynn, B., and Shacham, H. "Short Signatures from the Weil Pairing." *Proc. of ASIACRYPT* (2001).

[11] Bowers, K. D, Juels, A., and Oprea, A. "Proofs of Retrievability: Theory and Implementation." *Proc. of ACM CCSW* (2009).

[12] Burrows, J. "Secure Hash Standard." Tech. rep., DTIC Document, 1995.

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

[14] Chang, C., Kao, C., and Lin, C. "Efficient and Scalable Hierarchical Key Assignment Scheme." *Proc. of ACOS* (2006).

[15] Chang, C., Lin, I., Tsai, H., and Wang, H. "A Key Assignment Scheme for Controlling Access in Partially Ordered User Hierarchies." *Proc. of AINA* (2004).

[16] Chen, T., Chung, Y., and Tian, C. "A Novel Key Management Scheme for Dynamic Access Control in a User Hierarchy." *Proc. of COMPSAC* (2004).

[17] Chien, H. and ke Jan, J. "New Hierarchical Assignment without Public Key cryptography." *Computers and Security* (2003).

[18] Chou, J., Lin, C., and Lee, T. "A Novel Hierarchical Key Management Scheme based on Quadratic Residues." *Proc. of ISPA* (2004).

[19] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. "Introduction to Algorithms." *The MIT Press, ISBN 0-262-03141-8, McGraw-Hill, ISBN 0-07-013143-0* (1986).

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

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

[22] Das, A., Paul, N., and Tripathy, L. "Cryptanalysis and Improvement of An Access Control in User Hierarchy based on Elliptic Curve Cryptosystem." *Information Sciences* (2012).

[23] Douceur, J. "The Sybil Attack." *Proc. of IPTPS* (2002).

[24] Durand, Marianne and Flajolet, Philippe. "Loglog Counting of Large Cardinalities." *ESA: European Symposia on Algorithms* (2003): 605–617.

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

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

[27] Estan, C., Varghese, G., and Fish, M. "Bitmap Algorithms for Counting Active Flows on High-Speed Links." *IEEE/ACM Transactions on Networking (TON)* (2006).

[28] Flajolet, Philippe, Fusy, Eric, Gandouet, Olivier, and Meunier., Frdric. "HyperLogLog: The Analysis of a Near-optimal Cardinality Estimation Algorithm." *Proc. of AOFA: International Conference on Analysis Of Algorithms* (2007).

[29] Flajolet, Philippe and Martin, G. Nigel. "Probabilistic Counting Algorithms for Database Applications." *J. Comput. Syst. Sci.* (1985).

[30] Gassend, B., Suh, G., Clarke, D., Dijk, M. Van, and Devadas, S. "Caches and hash trees for efficient memory integrity verification." *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on* (2003).

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

[32] Grolimund, D., Meisser, L., Schmid, S., and Wattenhofer, R. "Cryptree: A Folder Tree Structure for Cryptographic File Systems." *Proc. of SRDS* (2006).

[33] Horowitz, E. and Sahni, S. *Fundamentals of data structures*. Computer science press, 1983.

[34] Juels, A. and Jr, B. Kaliski. "PORs: Proofs of Retrievability for Large Files." *Proc. of CCS* (2007).

[35] Katz, J. and Lindell, Y. *Introduction to Modern Cryptography*. 2007.

[36] Kikuchi, H., Abe, K., and Nakanishi, S. "Online certification status verification with a red-black hash tree." *IPSJ Digital Courier* (2006).

[37] Kumar, A., Xu, J., Wang, J., Spatschek, O., and Li, L. "Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement." *Proc. of IEEE INFOCOM* (2004, A journal version was published in IEEE JSAC, 24(12):2327-2339, December 2006).

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

[39] Li, T., Chen, S., and Ling, Y. "Fast and Compact Per-Flow Traffic Measurement through Randomized Counter Sharing." *Proc. of IEEE INFOCOM* (2011).

[40] Lieven, P. and Scheuermann, B. "High-Speed Per-Flow Traffic Measurement with Probabilistic Multiplicity Counting." *Proc. of IEEE INFOCOM* (2010).

[41] Lu, Y., Montanari, A., Prabhakar, B., Dharmapurikar, S., and Kabbani, A. "Counter Braids: A Novel Counter Architecture for Per-Flow Measurement." *Proc. of ACM SIGMETRICS* (2008).

[42] Merkle, R. "A Digital Signature Based on a Conventional Encryption Function." *Journal of CRYPTO* (1987).

[43] ———. "A digital signature based on a conventional encryption function." *Advances in Cryptology (CRYPTO)* (1988).

[44] Miller, G. "Riemann's Hypothesis and Tests for Primality." *Proc. of ACM STOC* (1975).

[45] Mo, Z., Zhou, Y., and Chen, S. "A Dynamic Proof of Retrievability (PoR) Scheme with O(logn) Complexity." *Proc. of IEEE ICC* (2012).

[46] Naor, M. and Nissim, K. "Certificate revocation and certificate update." *Selected Areas in Communications, IEEE Journal on* (2000).

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

[48] ———. "The Ephemerizer: Making Data Disappear." *Information System Security* (2005).

[49] Pfaff, B. "Performance Analysis of BSTs in System Software." *Proc. of SIGMETRICS* (2004).

[50] Pugh, W. “Skip lists: A Probabilistic Alternative to Balanced Trees.” *Communications of the ACM* (1990).

[51] Santis, A., Ferrara, A., and Masucci, B. “Efficient Provably-secure Hierarchical Key Assignment Schemes.” *Theoretical Computer Science* (2011).

[52] Shacham, H. and Waters, B. “Compact Proofs of Retrievability.” *Proc. of ASIACRYPT* (2008).

[53] Shamir, A. “How to share a secret.” *Communications of the ACM* (1979).

[54] Stoica, I., Morris, R., Karger, D., Kaashoek, M., and Balakrishnan, H. “Chord: A scalable Peer-to-peer Lookup Service for Internet Applications.” *Proc. of SIGCOMM* (2001).

[55] Tang, Q. “From Ephemerizer to Timed-Ephemerizer: Achieve Assured Lifecycle Enforcement for Sensitive Data.” *Technical Report TR-CTIT-10-01* (2010).

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

[57] Venkatataman, S., Song, D., Gibbons, P., and Blum, A. “New Streaming Algorithms for Fast Detection of Superspreaders.” *Proc. of NDSS* (2005).

[58] Wang, Q., Wang, C., Li, J., Ren, K., and Lou, W. “Enabling Public Verifiability and Data Dynamics for Storage Security in Cloud Computing.” *Proc. of ESORICS* (2009).

[59] Williams, D. and Sirer, E. “Optimal parameter selection for efficient memory integrity verification using merkle hash trees.” *Network Computing and Applications, 2004.(NCA 2004). Proceedings. Third IEEE International Symposium on* (2004).

[60] Wolchok, S., Hofmann, O., Heninger, N., Felten, E., Halderman, J., Rossbach, C., Waters, B., and Witchel, E. “Defeating Vanish with Low-cost Sybil Attacks against Large DHTs.” *Proc. of NDSS* (2010).

[61] Yoon, Myungkeun, Li, Tao, Chen, Shigang, and Peir, Jih-Kwon. “Fit a Compact Spread Estimator in Small High-Speed Memory.” *IEEE/ACM Transactions on Networking* (2011).

[62] Zhang, Y. and Blanton, M. “Efficient Dynamic Provable Possession of Remote Data via Balanced Update Trees.” *AsiaCCS* (2013).

[63] Zhao, Q., Xu, J., and Kumar, A. “Detection of Super Sources and Destinations in High-Speed Networks: Algorithms, Analysis and Evaluation.” *IEEE JASC* 24 (2006).10: 1840–1852.

[64] Zheng, Y., Hardjono, T., and Seberry, J. “New Solutions to the Problem of Access Control in A Hierarchy.” *Technical Report* (1993).

BIOGRAPHICAL SKETCH

Zhen Mo earned his Bachelor of Engineering degree in information security engineering from Shanghai Jiaotong University in 2007. He received his Master of Engineering degree in theory and new technology of electrical in 2010 from Shanghai Jiaotong University. In 2010 he joined the doctoral program in Computer Engineering at University of Florida. He received his Ph.D. from the University of Florida in the spring of 2015.