Membership Verification in Authenticating Dynamic Sets

http://dx.doi.org/10.3991/ijoe.v9i5.2973

Z.K.Wei[1], H.Y. Kim[1], Y.K. Kim[1], J.H. Kim[2]
[1] ETRI, Daejeon, Korea
[2] YoungDong Univesity, Chungbuk, Korea

***Abstract*—A file system for cloud storage has to guarantee integrity of its files. For this purpose, existing designs mainly rely on hash tree and RSA tree. A new design based on B+ tree construction is proposed in this paper. The new design performs better in dynamic authentication for integrity of file system and also relatively has low computational cost.**

***Index Terms*—B+ tree, dynamic authentication, membership query**

I. INTRODUCTION

Cloud storage provides some popular on-line services for archiving, backup, and even primary storage of files. It gathers different kinds of storage devices to work in cooperation by means of cluster application, grid computing technology and distributed file system. Cloud-storage providers offer users clean and simple file-system interfaces, abstracting away the complexity of direct hardware management. Amazon S3, IBM XIV and HP EXD9100 are well known examples. Using these cloud storage systems, users can rapidly deploy their data to the cloud and the cloud storage provider has to guarantee the security of their data. The security includes both confidentiality and integrity. Confidentiality which refers to identity authentication, data encryption and access control means that data should not be access by illegal users. Integrity means the data should not lose and be tampered with. Moreover, users can authenticate the integrity of their data which store in the cloud-storage System. This paper mainly presents a new construction which performs better when users want to authenticate the integrity of their data in dynamic sets.

When discussing authentication method for data integrity, making an abstract of the system and build the system model is essential. Two-party model and three-party model are the two kinds of system models in common use. Three-party model is composed of user, untrusted server and credible data source. In this model, data source send a digest with time stamp to user. The user issues query request and get data and proof for integrity authentication from untrusted server. Update operations perform in the data source which would send data information after update. In two-party model, there is not credible data source. Operations just occur between user and untrusted server. Figures 1 shows the two models.

II. RELATED WORK

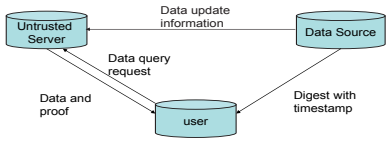
Authentication of data integrity means that the data, uploading from untrusted server by user, is valid and has not been tampered with. As known to all, traditional authentication method for data integrity is MAC and data signature, but it does not apply to cloud storage environment. In cloud storage environment, the archived files are very huge, so cloud-storage provider often divides these files into small buckets. Each bucket is an independent storage unit. When authenticating the data integrity, users need only some pieces of blocks, not the whole file. The archived file could be regarded in the cloud as a structured data set $\boldsymbol{S}$. As a result, integrity authentication actually could be taken as authenticating membership of $\boldsymbol{S}$. Furthermore, this kind of authentication could implement through authenticated data structure, such as hash tree, skip list and RSA tree. In the whole paper, $\boldsymbol{n}$ denotes the current number of elements of the data set.

A. Hash Tree

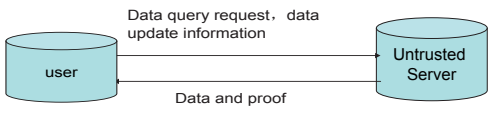
The hash tree scheme introduced by Merkle [1] which supports the construction of the data structure followed by query operations, but not updates operations (without complete rebuilding). The definition of label $\boldsymbol{f(v)}$ at each node v is as follows:

If v is a leaf, $\boldsymbol{f(v) = x}$, where x is stored at v; Else, $\boldsymbol{f(v) = h(f(a), f(b))}$ where $\boldsymbol{a}$ and $\boldsymbol{b}$ are the left and right child of $\boldsymbol{v}$ and $\boldsymbol{h}$ is a collision-resistant cryptographic hash function, such as MD5 or SHA1.

Suppose a query for an element $\boldsymbol{x}$: untrusted server always returns the proof, consisting of nodes labels which are along the path from x to the root together with all sibling nodes labels in that path. Credible data source returns the root label. The user recomputes the root label by hashing the labels in the proof in the appropriate order. If the value so obtained matches the one provided by the credible data source, and then the data obtained from the untrusted server is valid.



(a). Three-party model



(b). Two-party model

Figure 1. Two kinds of System Models

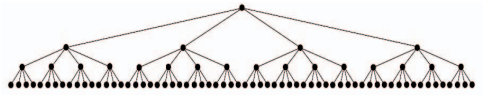


Figure 2. RSA tree

The hash tree scheme achieves $O(\log n)$ proof size, query time, update time and verification time. The disadvantages of this scheme are high computational cost and does not support update operation (without complete rebuilding).

B. Skip List

Goodrich and Tamassia [2] have designed a data structure based on skip lists. A skip list stores a set ***S*** of elements in a series of linked lists $S_0, S_1, \ldots, S_t$.The label of elements is computed by a commutative hashing .The base list, S_0, stores all the elements of S in order, as well as sentinels -∞ and +∞. Each successive list S_i, for i >=1, stores a sample of the elements from S_{i-1}. To define the sample from one level to the next, each element of S_{i-1} was chosen at random with probability 1/2 to be in the list S_i. The sentinel elements -∞ and +∞ are always included in the next level. The top level contains only the sentinels. This scheme possesses similar authentication process to that of hash tree scheme, as well, has high computational cost $O(\log n)$. Differently, skip list scheme can support update operations but with high algorithm complexity.

C. The RSA tree

The RSA tree scheme uses another cryptographic primitive, namely one-way accumulator [3]. This scheme based on RSA exponentiation function implements a secure one-way function that satisfies quasi-com mutativity [3]. Using the reference of solution to memory consistency [4], the scheme based on RSA tree was addressed in [5].

In this scheme, Let $0 < \varepsilon < 1$ be a constant and $S = [e1, e2, \ldots, en]$ be the set of elements to authenticate. A tree

is built $T(\varepsilon)$ on top of ***S***, which called the RSA tree of ***S***, such that: [5]

1. The leaves of the tree store the actual elements;
2. The tree consists of exactly $[1/\varepsilon]+1$ levels;
3. Every node of the tree has $O\left(n^{\varepsilon}\right)$ children;
4. Level i in the tree contains $O(n1-i\varepsilon)$ nodes (the leaves node of the tree lies in level 0).

Membership can be verified by using RSA accumulator [6] summarized set ***S***. The following figure 2 shows the RSA tree.

The process of Integrity authentication based RSA tree is similar to that of hash tree scheme. However, it is worth highlighting that RSA tree scheme achieves ***O(1)*** proof sizes, query time, and verification time. Unfortunately, this scheme cannot still support update operations without cyclical and complete rebuilding after a short time.

D. B+ Tree

The RSA tree scheme gains the lower time and space complexity. In this paper, An efficient and practical data structure, B+ tree, is introduced which factors away the disadvantage of update operation in RSA tree scheme, while maintaining some algorithms of that scheme and the lower time and space complexity $O(1)$.

A B+ tree in the form of a balanced tree is a type of tree which represents sorted data in a way that allows for efficient retrieval and update of records, each of which is identified by a key. The order of B+ tree, measuring the capacity of nodes, referred to here as b, a B+ tree has to meet the following conditions:

1) The root which is not the single leaf is allowed to have as few as two children.

2) The actual number of children for an internal node, referred to here as ***m***, is constrained for $\lceil b/2 \rceil \le m \le b$

3) An internal node, whose number of subtrees is ***n***, contains n keys and ***n*** pointers.

4) Leaf nodes which are always on the same level, have no children, but are constrained so that the number of keys must be at least $\lfloor b/2 \rfloor$ and at most $b - 1$.

5) Leaf nodes are linked together to form a link list, so that B+ tree can support fast process of range-search queries.

6) The tree has two pointers: they point to the root and the smallest leaf respectively.

The primary value of a B+ tree is in storing data for efficient retrieval and graceful adjustment after updates in particular, file systems. NTFS, ReiserFS, NSS, and JFS file systems all use this type of tree.

The B+ tree structure is an efficient means for storing a set ***S*** of elements. It supports the following operations:

Search (key): determine whether key is in ***S***. It supports random search from the root and sequential search from the smallest tree. The algorithm of random search is as follows:

```
/*  if found return the logical address of the record;   if not found return -1 */

int search(key)
{
   if (the tree is empty)  return -1;
   read_btidx_block(&rnode, bblock_hdr.root_bnum, fidx);  /* read b+ tree root into rnode */
   while (curlevel <tree_level - 1) {                     /* not reach the leaf level, still in the tree */
      int pointer = get_pointer(key, &rnode, &key_id);
      read_btidx_block(&rnode, pointer, fidx);
      curlevel++;                                          /* level down */
   }
   result = get_leaf_pointer(key, &rnode, &key_id);        /* get address */
   return result;
}
```

Insert (key): insert key into ***S***. the algorithm is as follows:

```
/*     * key:        key to be inserted;
       * address: addr, point to the record block     */

void insert(key_t key, int address)
{   search();
   if (key found)    return;
   if (leaf has empty space)              free_insert();
   else {   split_leaf_node();   update_inside_node(); }
}

/* update internal node when a key will be insert in this node */

void update_inside_node(int *key, int block_id)
{
   while (curlevel > 0) {                                  /*not reach root */
      curlevel--;
      get_pointer(*key, &node, &key_id);

      if (node.d < KEY_NUM) {                              /* have enough space to insert */
             free_insert_inside(*key, block_id, &node, key_id);
             return;
      } else {                                             /* do not have enough space, split */
             split_inside_node(key, block_id, key_id, &node, &new_node);
      }                                                    /*split into node and new_node */
   }
   if (curlevel == 0) {                                    /* need new a root node */
                                                                /* create the new root node */
   }
}
```

Delete (key): remove key from ***S***. the algorithm is as follows:

```
/*     * key:        key to be inserted;
       * address: addr, point to the record block     */

void insert(key_t key, int address)
{   search();
   if (key found)    return;
   if (leaf has empty space)              free_insert();
   else {   split_leaf_node();   update_inside_node(); }
}

/* update internal node when a key will be insert in this node */

void update_inside_node(int *key, int block_id)
{
   while (curlevel > 0) {                                  /*not reach root */
      curlevel--;
      get_pointer(*key, &node, &key_id);

      if (node.d < KEY_NUM) {                              /* have enough space to insert */
             free_insert_inside(*key, block_id, &node, key_id);
             return;
      } else {                                             /* do not have enough space, split */
             split_inside_node(key, block_id, key_id, &node, &new_node);
      }                                                    /*split into node and new_node */
   }
   if (curlevel == 0) {                                    /* need new a root node */
                                                                /* create the new root node */
   }
}
```

E. Organization of the Paper

The following is the organization of this paper. In Section 3, some necessary cryptographic and algorithmic ideas needed for our construction is presented. In Section 4, our design goals are introduced and our construction is developed. This paper proposes Authentication Method in dynamic sets in Section 5. And in Section 6, it is the conclusion of the paper.

III. PRELIMINARIES

Definition 1 (Hash Tables) [7]: Suppose ***n*** elements from a universe ***U*** are stored in a data structure so that constant look-up time can be expected. Set up a one-dimensional table ***T[1 . . .m]*** where ***m = O(n)***, fix a special function h : **U →{1, . . . ,m}**, such that for any two elements ***e1, e2 ∈ U , P[h(e1) = h(e2)] ⩽ 1/m***, and store element e in slot ***T(h(e))***. The probabilistic property that holds for ***h***, combined with the fact that ***h*** can be

computed in ***O(1)*** time, leads to the conclusion that there is always a constant expected number of elements that map to the same slot $i(1 \le i \le n)$ and therefore, look-up takes constant expected time.

Theorem 1 (Dynamic Hashing)[8]: For a set of size n, dynamic hashing can be implemented to use ***O(n)*** space and have ***O(1)*** expected query cost for membership queries and ***O(1)*** expected amortized cost for insertions or deletions.

Definition 2 (Two-Universal Hash Functions): Such functions were first introduced by Carter and Wegman [9]. A family ***H = {h : A → B}*** of functions is two-universal if, for all ***a1,a2 ∈ A,*** $a_1 \neq a_2$ and for a randomly chosen function h from ***H***, $P_h \in H\{h(a_1) = h(a_2)\} \le 1/|B|$.

Lemma 1 (Prime Representatives) [6]: Let ***H*** be a two-universal family of functions from $\{0,1\}^{3k}$ to $\{0,1\}^{k}$ and ***h ∈ H***. For any element $e_i \in \{0,1\}^{k}$, a prime $x_i \in \{0,1\}^{3k}$ can be computed so that $h(x_i) = e_i$, by sampling ***O(k2)*** times from the set of inverses $h^{-1}(e_i)$.

Definition 3 (Negligible Function) [6]: A real-valued function ***v(k)*** over natural numbers is ***neg(k)*** if for any nonzero polynomial ***p***, there exists m such that $\forall n > m, |v(n)| < 1/p(n)$.

Definition 4 (Strong RSA Assumption) [6]: Given an RSA modulus ***N*** and a random element ***x ∈ ZN***, it is hard (it happens with probability ***neg(k)***) for a computationally bounded adversary ***A*** to find ***y>1*** and a such that ***ay = x mod N***.

Definition 5 (RSA accumulator) [6]: Suppose a set of k-bit elements $S = \{e_1, e_2, ..., e_n\}$. Let ***N*** be a ***k′*** bit RSA modulus (***k′ >3k***), namely ***N=pq***, where ***p***, ***q*** are strong primes [9]. ***S*** can be efficiently represented with a ***k′*** bit integer, namely the integer $f(S) = g^{r(e_1)...r(e_n)} \bmod N$, where ***g ∈ QRN*** [10] and is a ***3k*** bit prime representative. This representation has the property that any computational bounded adversary A, that does not know ***ϕ (N)***, cannot find another set of elements ***S′ ! = S*** such that ***f (S′) = f(S)***, unless A breaks the strong RSA assumption.

IV. OUR CONSTRUCTION

A. Design Goals

The design of our construction should address the following goals:

1) Low computational cost: It achieves $O(1)$ proof sizes, query time, verification time and linear update time.

2) Better performance on authenticating dynamic sets (without compute rebuilding)

3) High security: the authenticity of the data should be verifiable with a high degree of reliability.

B. B+ Tree Constructions

The new authenticated data structure is built based on B+ tree. Let ***0<ε<1*** be a constant and $S = \{e_1, e_2, ..., e_n\}$ be the set which we would like to authenticate. Firstly store the elements of ***S*** in a hash table, referred to here as t, then build B+ tree ***T(ε)*** based on t, satisfying the following additional conditions:

1) The leaf nodes store the prime representatives of Hash value of elements in ***S***;

2) The number of levels is ***⌈1/ε⌉+1***;

3) The order of ***T(ε)*** is ;

4) Every node has $O(n^{\varepsilon})$ children.

In this construction, RSA accumulator is used to produce a short and computational proof to support the membership verification. Let ***l=⌈1/ε⌉*** and leaf nodes lie at level 0. given l ***RSA*** moduli ***N1,N2,…,Nl, l*** two-universal functions ***h1,h2,…,hl and g1,g2,…,gl(*** $g_i \in QR_{N_i}$ ***)***.

For every node v of ***T(ε)***, digest ***χ(v)*** is defined as follow:

If v lies at level 0, ***χ(v)=e*** ; Else $x(v) = g_i^{\prod_{u \in N(v)} r_i(x(u))} \bmod N_i$ [5];

where ***N(v)*** is the set of children of v and $r_i(\chi(v))$ is a representative of ***χ(v)***. [5];

In particular, ***χ(s)= χ(r)***, where s is the given set and ***r*** is the root of ***T(ε)***. Under the strong RSA assumption, the probability that a computationally bounded adversary ***A***, knowing only the RSA moduli N_i and g_i, ***1⩽ i ⩽ l***, can find another set $S_1 \neq S_2$ such that ***χ (S1) = χ (S2)*** is ***neg(k)***.

V. AUTHENTICATING DYNAMIC SETS

In this section, how the B+ tree authentication structure to verify membership in a dynamic set will be described. Let $S = \{e_1, e_2, ..., e_n\}$, a B+ tree is built based on S. the RSA moduli Ni and bases gi which are defined in section 3, $1 \leqslant i \leqslant l$, are public.

A. Queries and Verification

Suppose a query for an element x: untrusted server always returns a sequence of proof stored at the nodes which are along the path from the leaf containing the element x to the root .The user stores only the set digest $d = \chi(S)$ which achieved from the credible data source and recomputed χ(S) with the proof in the appropriate order . Let $v_0, v_2, ..., v_l$ be the path from x to the root, $r = v_l$. Let B(v) be the set of siblings of node v .The proof was defined as follow: $\pi_i = (\alpha_i, \beta_i)$ ($i = 1, ..., l$) where $\alpha_i = r_i(x(v_i))$ [5] and $\beta_i = g_i^{\prod_{u \in B(v_i)} r_i(x(u))} \bmod N_i$ [5]. The user verifies the membership by the following three equations:

$$h_1(\alpha_1) = x, \quad (1)$$

$$h_i\left(\alpha_i\right)=\beta_{i-1}^{\alpha_{i-1}} \bmod N_{i-1}, \quad (2)$$

$$d=\beta_l^{a_l} \bmod N_l. \quad (3)$$

If the three equations hold, the data obtained from the untrusted server is valid. Suppose that query time is the time to construct the proof and not the time to find for the element, on account of that can be achieved with hash table in $O(1)$ time. Due to precomputed proof and constant depth of the tree, both query time and verification time take $O(1)$. The size of proof is also $O(1)$.

B. Updates

When update occurs, digests of the nodes along path from the updated leaf to the root have to recompute.

In three-party model, updates performed at the data source and the information pass on to the untrusted server is as follows:

- Operation performed (insertion/deletion).
- Element x involved.
- Signed statement consisting of a timestamp and new digests in corresponding path.

Lemma 2 (*O(n log n)* Witness Updates) [11]. Let ***N*** be an RSA modulus. Given the elements $x_1, x_2, \ldots, x_n$, ***N*** and ***g***, without the knowledge of ***φ(N)***, $A_i=\prod_{j\neq i} x_j \bmod N_i$ for $i=1,2,\ldots,n$ can be computed in $O(n\log n)$ time.

The process of update to ***T(ε)*** can perform by means of algorithms introduced in section 2. Update operations only affect Concentrated part of ***T(ε)*** and need not rebuild it unless the increased leaf nodes or deleted leaf nodes are more than ***3n/4*** respectively. The authenticated information has size. Due to the constant depth of ***T(ε)***, the update time is also $O\left(n^{\varepsilon}\log n\right)$.

In two-party model, the untrusted server has to perform the update operation by itself. Suppose the user would like to insert x into ***T(ε)***: Instead of getting the new digest $d'=\chi\left(S'\right)$ from the untrusted server, the user should compute it locally, then sends ***x*** to the untrusted server. The untrusted server performs the insertion by means of algorithms introduced in section 2 and also updates the digests along the path from x to the root.

Subsequently, it sends the new digests, a_i', along the update path to the user. Then the user verifies the correctness of the update: it issues a query for ***y*** and the untrusted server returns ***y*** and its proof which defined in the 'Queries and Verification' section. If the following equations hold, the insertion is valid (i =1, 2,…, l).

$$h\left(\alpha'_1\right)=x, \quad (4)$$

$$h\left(\alpha'_2\right)=\beta_1^{\alpha'_1 r(x)} \bmod N_1, \quad (5)$$

$$h\left(\alpha_i'\right)=\beta_{i-1}^{\alpha'_{i-1}} \bmod N_{i-1}, \quad (6)$$

$$d'=\beta_{l-1}^{\alpha'_{l-1}} \bmod N_{l-1}. \quad (7)$$

Delete operations is similar to the insertion except for replace equation 5 of $h\left(\alpha'_2\right)=\beta_1^{\alpha_1 r(x)^{-1}} \bmod N_1$. ***T(ε)*** need not rebuild it unless the increased leaf nodes or deleted leaf nodes are more than ***3n/4*** respectively. The size of the verification proof is $O(1)$. Due to the constant depth of ***T(ε)***, the update time is also $O\left(n^{\varepsilon}\log n\right)$ time as that in the three-party model.

VI. CONCLUSION

In this paper, a new authentication construction for membership verification is presented based on B+ tree and RSA accumulator. Our construction performs better in authenticating dynamic sets without frequently rebuilding the data structure.

ACKNOWLEDGMENT

This work was supported by the IT R&D program of MKE/KEIT, KOREA. [10038768, the Development of Supercomputing System for the Genome Analysis].

REFERENCES

[1] R. C. Merkle, "A certified digital signature," in *Adv. Cryptology—CRYPTO'89 Proc.*, Springer New York, 1990, pp. 218-238. http://dx.doi.org/10.1007/0-387-34805-0_21

[2] M. T. Goodrich, R. Tamassia, "Efficient authenticated dictionaries with skip lists and commutative hashing," U.S. Patent App 416, 015, Oct., 18, 2000.

[3] J. Benaloh, M. d. Mare, "One-way Accumulators: A Decentralized Alternative to Digital Signatures," in *Workshop theory & appl. cryptographic tech. Adv. Cryptology*, Lofthus, Norway, 1994, pp. 274-285.

[4] C. Dwork, et al., "How Efficient Can Memory Checking Be," in *Proc. 6th Theory Cryptography Conf. Theory Cryptography*, San Francisco, CA, 2009, pp. 503-520. http://dx.doi.org/10.1007/978-3-642-00457-5_30

[5] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Authenticated hash tables", in *Proc. 15th ACM Conf. Comput. & Commun. Secur.*, Alexandria, USA, 2008, pp. 437-448.

[6] Goodrich M T, Tamassia R, Hasić J. An Efficient Dynamic and Distributed Cryptographic Accumulator*[M]//Information Security. Springer Berlin Heidelberg, 2002: 372-388.

[7] M. Dietzfelbinger, A. Karlin, and K. Mehlhorn, et al. "Dynamic perfect hashing: Upper and lower bounds," *SIAM J. Comput.*, vol. 23, no. 4, pp. 738-761, Nov., 1994. http://dx.doi.org/10.1137/S0097539791194094

[8] T. H. Cormen, C.E. Leiserson., R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed., Cambridge, England: MIT Press, 2001, pp. 128.

[9] J. L. Carter, M. N. Wegman, "Universal classes of hash functions," *J. Comput & Sys. Sci.*, vol. 18, no. 2, pp. 143-154, Sep., 1979.

[10] J. Camenisch, A. Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *Advances in Cryptology—CRYPTO 2002*, 20th ed., California,USA, Germany: Springer Berlin Heidelberg, 2002, pp. 61-76. http://dx.doi.org/10.1007/3-540-45708-9_5

[11] T. Sander, A. Ta-Shma, and M. Yung, "Blind, auditable membership proofs,"in *Financial Cryptography*, Springer Berlin Heidelberg, 2001, pp. 53-71. http://dx.doi.org/10.1007/3-540-45472-1_5

AUTHORS

Z. K. Wei, H.Y. Kim, and **Y.K. Kim** are with the Cloud Computing Research Department, ETRI, Daejeon, Korea.

J. H. Kim is with the Department of Computer & Information Engneering, YoungDong Univesity, Chungbuk, Korea.

Submitted 01 July 2013. Published as re-submitted by the authors 15 September 2013.