2014 ASE BIGDATA/SOCIALCOM/CYBERSECURITY Conference, Stanford University, May 27-31, 2014

# A Simple and More Versatile Authentication Path Computation Algorithm for Binary Hash Tree

Shuanghe Peng, Yingjie Qin, Zhige Chen
School of Computer and Information Technology, Beijing Jiaotong University
shhpeng@bjtu.edu.cn, 13125178@bjtu.edu.cn, 13120384@bjtu.edu.cn

## Abstract

Merkle tree, an important complete binary tree, is always used in theoretical cryptographic constructions such as one-time signature. How to get the authentication path of the leaf node is an important part of Merkle Signature Scheme. Currently, the computation of authentication path for Merkle tree is only applicable to complete binary tree, but not applicable to unbalanced tree.

We propose a new approach for computing the authentication path for binary hash tree by using two stacks. It is applicable not only to balanced hash tree, such as Merkle tree, but also other binary hash trees, such as skewed Merkle tree and Huffman Certification Revocation Tree. Therefore it is more versatile when compared with the previous algorithms. Besides, it is structurally very simple and easy to implement.

For a balanced hash tree with $N$ leaves, the time and space complexity are $2\log(N)$ and $2\log(N)$ respectively for the authentication path computation of a specific leaf node. As for the authentication path computation of a sequence of leaves, e.g. $M$ leaves, the average time complexity is $\frac{2M(2N-1)}{N}$ and space complexity is $2\log(N)$.

## 1 Introduction

Merkle tree[1] was always used in theoretical cryptographic constructions such as multiple one-time signatures associated to a single public key [2]. Since this introduction, a Merkle tree has been defined to be a complete binary tree[1]. In the Merkle tree each interior node value is a one-way function of the node value of its children. Merkle trees had found a wide range of applications in cryptography, such as certification revocation list verification [3], out-sourced databases verification and micro-payments [4].

In many applications, given an authentication path and a leaf, one can verify the integrity of the leaf with respect to the publicly known root value [5], where authentication path consists of the interior nodes that constitute the siblings on the path from the leaf to the root.

Computation of the authentication path is the key factor for generating a Merkle signature. The process of computing the authentication path of leaf node in Merkle tree is also called Merkle tree traversal [6, 7]. Merkle tree traversal was considered in a number of papers, e.g. [5-14].

The problem of Merkle tree traversal in [5-14] is that it is only applicable to balanced hash trees, e.g. complete binary trees. It can not be applicable to other binary hash trees. This shortage may causes some problems as follows.

In DPDP schemes, if Merkle tree is used for verification like the one described in [15], the dynamic data update operations, such as insertion and removal, can destruct the balance status of the Merkle tree. The updated Merkle tree must be re-balanced to make the Merkle tree traversal algorithm [5-14] applicable. It is inefficient in this way.

We call all binary trees with their interior node value being a one-way function of the node value of its children as binary hash tree. How to make the authentication path algorithm applicable to all binary hash trees is an open problem. We will focus on the problem of efficiently computing the authentication path for binary hash tree, which include not only balanced binary Merkle trees, but also other binary hash trees.

### 1.1 Related Work and Applications

Apart from Merkle tree, other binary hash trees also have wide application. Since other binary hash trees are not a balanced, algorithms of authentication path computation in [5-14] are not applicable to them.

In 2004, Farid F. Elwailly et al. presented a new scheme for efficient certificate revocation [16]. The core of their scheme was a novel construct termed a QuasiModo tree, which was like a Merkle tree but contained a length-2 chain at the leaves and also directly utilized interior nodes.

In 2004, Marek Karpinski and Yakov Nekrich [17] solved the problem of traversal skew Merkle trees, which are unbalanced ones, and designed an algorithm for it. But the skew Merkle tree is a modified Merkle tree with extra leaves attached to the right most part of the tree. The proposed traversal algorithm in [17] was not applicable to all binary hash trees.

In paper [18] Sunoh Choi et al. also used skewed tree to provide not only efficient top-k aggregation over distributed databases but also authentication on the top-k results.

In 2005, Yuan Xue [19] proposed a new variant of $CRT$(Certification Revocation Tree), namely $H-CRT$ (Huffman $CRT$) by using the idea of Huffman tree. In $H-CRT$ a searching probability $W_i$ is defined for every leaf node $A_i$. The node which has the most weight has the shortest path. More frequently queried response node was assigned shorter hash path. The processing method greatly reduced the average hash path length and further optimized the performance of certification revocation system. But the paper only considered the performance improvement of $CRT$, it didn't consider the computation of authentication path for $H-CRT$, which is also not a complete binary tree with leaves having different depth.

## 1.2 Our Contribution

We propose a technique for traversal of binary hash trees which is structurally very simple and versatile. It can be employed not only in Merkle tree but also other binary hash trees, such as skew Merkle tree, QuasiModo tree and H-CRT.

Our technique can be employed in any construction which can meet different authentication paths generation requirement of single or consecutive leaf pre-images according to different situations.

## 1.3 Organization

Section 2 describes preliminaries and notations used in our algorithm. Section 3 describes the main idea of our algorithm. Section 4 states our authentication path computation algorithm. Section 5 compares the new algorithm with classical algorithm [9], Szydlo from [7, 11] and fractal algorithm from [6]. Section 6 states our conclusion.

## 2 Preliminaries and Notations

In this part we are going to give a brief description of the binary hash tree traversal technique, but first we need to introduce some preliminaries and notations used in it.

**Binary Hash Tree.** We now describe binary hash tree. Binary hash tree has a feature that every node has two children nodes if it is an interior node. For an interior node $n$, let $LC(n)$ and $RC(n)$ denote its left and right child respectively. Let $||$ denote the concatenation operation, and $H()$ denote the hash function. Then,
$Value(n) = H(Value(LC(n))||Value(RC(n)))$.

Unlike the Merkle tree, binary hash tree can be both balanced and unbalanced, while Merkle tree can only be a complete binary tree.

**Authentication Path.** We will make use of the notion of the $AuPath$ as authentication path for a given node in the binary hash tree. For a node $n$, $AuPath(n)$ is the set of siblings of the nodes on the path from $n$ to the root. More formally, if we let $Sib(n)$ and $Parent(n)$ denote $n$'s sibling and parent respectively, then:

$$AuPath(n) = \begin{cases} \phi, & \text{if } n \text{ is the root,} \\ Sib(n) \bigcup AuPath(Parent(n)), & \text{otherwise.} \end{cases} \tag{1}$$

The notion of $AuPath$ is applicable not only to complete binary tree but also unbalanced ones, such as skew Merkle tree, $H-CRT$, and QuasiModo tree.

**Legend.** We use ellipse filled with dots to denote pre-order visited nodes, ellipse filled with slash to denote authentication path nodes of the pre-order visited nodes, diamond to denote leaf nodes during pre-order traversal.

**Definition 1:**[$stackNode$] A stack stores the currently visited nodes in pre-order traversal of the binary hash tree.

**Definition 2:**[$stackAuth$] A stack stores the authentication path nodes for the currently visited nodes of the binary hash tree.

**Definition 3:**[$node.flag$] The current status of the visited node of the binary hash tree is indicated by a flag.

$$node.flag = \begin{cases} 0, & \text{if } node\text{'s children have} \\ & \text{not been visited yet,} \\ 1, & \text{otherwise.} \end{cases} \tag{2}$$

**Definition 4:**[$n.keyword$] A keyword stores the value to help to make the hash tree become a binary sort tree. The keywords of all leaves are given, which are ordered from left to right, and keyword of the interior node is the median of its two children.

**Notations** is shown in table 1.

Table 1: Notations

| $Functions$ | $Description$ |
|---|---|
| $stack.init()$ | initialize an empty stack |
| $stack.empty()$ | judge the stack is empty or not |
| $stack.push()$ | push an element into the stack |
| $stack.pop()$ | pop an element from the stack |
| $stack.gettop()$ | get the top element of the stack |
| $stack.output()$ | output all the elements of the stack |
| $exchTop(stack_1, stack_2)$ | switch the top elements of two stacks |
| $clearFlag(n)$ | clear the flag of node $n$, i.e. $n.flag \leftarrow 0$ |

## 3 Main Idea

The idea of our algorithm is something like pre-order traversal of a tree [20].

A traversal starts at the root of the tree and visits every node in the tree exactly once. Pre-order traversal of a binary hash tree with a stack is described in algorithm 1.

**Algorithm 1** *Preorder Traversal of a Binary Hash Tree with a Stack*

```
Require:
1: root index
2: stack Stack
Ensure: Output: nodes of the tree
3: Push the root onto the Stack;
4: while the Stack is not empty do
5:     Pop a vertex off Stack, and write it to the output list;
6:     Push its children right-to-left onto Stack;
7: end while
```

As shown in Algorithm 1 the pre-order traversal process of a binary hash tree, after the top element of the stack is popped off the stack, right child and left child are pushed into the stack one after the other. Only one stack is used in the process.

The key difference of our algorithm from pre-order algorithm is that two stacks $stackNode$ and $stackAuth$ are used to store visited nodes and authentication path nodes respectively during the pre-order traversal.

**Authentication path computation**:In this part we'll describe our new algorithm to compute authentication path for binary hash tree.

Before computing the authentication paths of leaves in the tree, we need to build a hash tree. We can build the hash tree using different methods according to different application scenes. For example, if a complete binary tree is needed, we can use the algorithm in paper [8, 12] to build a Merkle tree. If a CRT tree is needed, we can use the algorithm in paper [17, 18, 19] to build it.

The approach for generating authentication path for the first given leaf can follow a bottom-up strategy or a top-bottom strategy.

In the bottom-up strategy, a parent pointer is needed in every node of the tree to get the path from the given leaf node to the root, and the correspondent sibling nodes along the path are stored in stack *stackAuth*.

Here we follow a top-bottom strategy to get the first given leaf node. In order to compute the authentication paths of leaves efficiently in time, we need to add a keyword field to each node of the tree during the process of building the hash tree, where the definition of keyword is shown in **Definition 4**. After adding the keyword field, the built hash tree becomes a binary sort tree and the computation of traversal path and authentication path is more destination-oriented.

When computing the authentication paths of leaves, three situations should be considered:

**A specific leaf, e.g.** $g$ : When computing the authentication path of a specific leaf $g$, we find the path from the root to the given leaf $g$ according to the **Search Algorithm** of binary sort tree and its authentication path is recorded in the other stack accordingly.

**A sequential of leaves e.g.** $[g_1, g_n\ ]$: To compute the authentication path of some specific sequential leaves, we find the path of the first leaf $g_1$ according to the **Search Algorithm** of binary sort tree. And then, we can find the path of the following leaves according to the pre-order traversal algorithm until the last given leaf $g_n$ is found. Their traversal path and authentication path can be computed during the pre-order traversal process.

**All leaves** : This is a special situation for the computation of a sequential of leaves, e.g. $[g_1, g_n]$, where $g_1$ is the leftmost leaf node of the tree, $g_n$ is the rightmost leaf node of the tree. In fact, the first situation is also a special case of the second one, where $g_1$ and $g_n$ are the same leaf node.

## 4 Our authentication path algorithm

At the beginning, authentication path for the first given leaf should be generated, which is described in the first *while* loop, lines 12-22 of algorithm 2. The goal of this *while* loop is to find the first given leaf according to the search algorithm in binary sort tree. The traversal path nodes and authentication path nodes are pushed into stack *stackNode* and *stackAuth* respectively.

**Algorithm 2** *Authentication Path $\prod$ Generation for a Series of Leaf Nodes between $\varphi$ and $\phi$ of Binary Hash Tree $T$, $\prod = genAuthPath(T, \varphi, \phi)$*

```
Require:
 1: root node $T$
 2: stack $StackNode$
 3: stack $StackAuth$
 4: start leaf index $\varphi$
 5: end leaf index $\phi$
Ensure: output authentication paths for leaf nodes between $\varphi$ and $\phi$
 6: $stackNode.init()$;
 7: $stackAuth.init()$;
 8: for all node $n$ of the hash tree $T$ do
 9:   $clearFlag(n)$;
10: end for
11: $curNode \leftarrow root$;
12: while $curNode.key <> \varphi.key$ do
13:   $curNode.flag = 1$;
14:   if $\varphi.key < curNode.key$ then
15:     $stackNode.push(curNode.lchild)$;
16:     $stackAuth.push(curNode.rchild)$;
17:   else
18:     $stackNode.push(root.rchild)$;
19:     $stackAuth.push(root.lchild)$;
20:   end if
21:   $curNode \leftarrow stackNode.getTop()$;
22: end while
23: while $!(StackNode.Empty())$ do
24:   $curNode \leftarrow StackNode.getTop()$;
25:   if $curNode$ is an interior node then
26:     if $curNode.flag == 0$ then
27:       $StackNode.push(curNode.lchild)$;
28:       $StackAuth.push(curNode.rchild)$;
29:       $curNode.flag == 1$;
30:     else
31:       $curAuth \leftarrow StackAuth.getTop()$;
32:       if $curNode$'s right brother is $curAuth$ then
33:         $exchTop(StackNode, StackAuth)$;
34:       end if
35:       if $curNode$'s left brother is $curAuth$ then
36:         $StackNode.Pop()$;
37:         $StackAuth.Pop()$;
38:       end if
39:     end if
40:   else
41:     call $StackAuth.output()$ to output all elements in the $StackAuth$;
42:     they are authentication path nodes for the $curNode$.
43:     if index of $curNode$ is $\phi$ then
44:       return;
45:     end if
46:     $curAuth \leftarrow StackAuth.getTop()$;
47:     if $curNode$'s left brother is $curAuth$ then
48:       $StackNode.Pop()$;
```

```
49:       StackAuth.Pop();
50:     else
51:       if curNode's right brother is curAuth then
52:         exchTop(StackNode, StackAuth);
53:       end if
54:     end if
55:   end if
56: end while
```

After that,as showed in every round in the second *while* loop, lines 23-56 of algorithm 2, we run a check if the top element of stack $stackNode$ is an interior node or not.

If it is an interior node, we should also check the node's flag. If its flag is 0, it denotes that its children have not been visited yet and its left child and right child should be pushed into the stack $stackNode$ and $stackAuth$ respectively. If its flag is 1, which means that its children have been pushed into the stacks, so either the top elements of these two stacks are exchanged or popped according to the relationship of these two elements.

If it is a leaf node, we first output all the elements in the stack $stackAuth$, they are the authentication path nodes of the currently visited leaf. After that we'll check if the visited leaf has right brother at the top of stack $stackAuth$ or not. If the visited leaf has right brother at the top of stack $stackAuth$, the top elements of these two stacks are exchanged. Otherwise, they are popped.

1. If the top element $curNode$ of stack $stackNode$ is an interior node:
   (a) If node $curNode.flag == 0$, which means that the children of $curNode$ haven't been visited yet, then left child and right child of node $curNode$ are pushed into the stack $StackNode$ and $StackAuth$ respectively. After that $flag$ of node $curNode$ is set to 1 to indicate that its children have been visited (**case 1**: lines 25-29 in algorithm 2),
   (b) If node $curNode.flag == 1$, which means that the children of node $curNode$ have already been visited.
      i. If node $curNode's$ right brother is on the top of stack $StackAuth$, top elements of stack $StackNode$ and $StackAuth$ are exchanged(**case 2**: lines 30-34 in algorithm 2),
      ii. If node $curNode's$ left brother is on the top of stack $StackAuth$, top elements of the two stacks $StackNode$ and $StackAuth$ are popped(**case 3**: lines 35-38 in algorithm 2),
2. If the top element $curNode$ of stack $stackNode$ is a leaf node, its authentication path nodes are in stack $StackAuth$ and they are output:
   (a) If node $curNode$ has left brother on the top of stack $StackAuth$, then top elements of the two stacks $StackNode$ and $StackAuth$ are popped, which means "go back in pre-order traversal" (**case 4**: lines 47-49 in algorithm 2),
   (b) If node $curNode$ has right brother on the top of stack $StackAuth$, then top elements of stack $StackNode$ and $StackAuth$ are exchanged (**case 5**: lines 51-53 in algorithm 2).

## 4.1 Example

We take the complex situation for example. Consider the unbalanced binary hash tree with 8 leaves as shown in Figure 1. We'll compute the authentication path of leaves between $J$ and $M$, i.e.$[J, M]$.

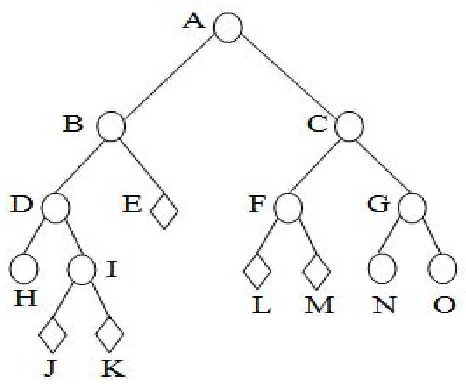

Figure 1: Example, an unbalanced binary hash tree with 8 leaves

In order to make it clear, we would explain the example step-by-step. The steps are No.(1) to No.(14) in Figure 2 to Figure 8.

At the beginning, two stacks $StackNode$ and $StackAuth$ are empty, which is described at the left side of Figure 2, i.e. step No.(1). The first given leaf node $J$ should be found which is described in Figure 2 to Figure 4. The corresponding pseudo-codes are in the first *while* loop of algorithm 2.

According to the first *while* loop of algorithm 2, root $A's$ left child $B$ and right child $C$ are pushed into the stack $StackNode$ and $StackAuth$ respectively, which is described at the right side of Figure 2, i.e. step No.(2).

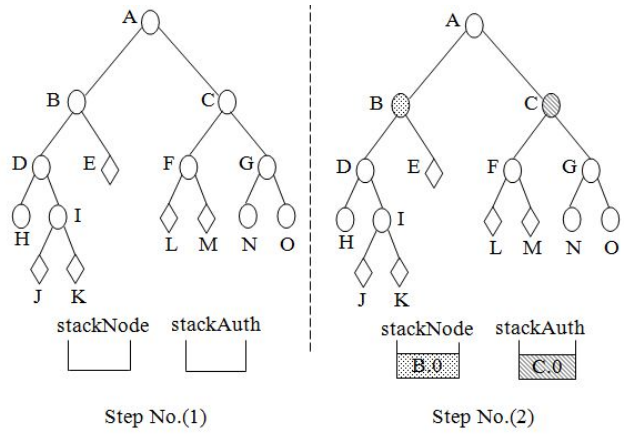

Figure 2: Root $A's$ left child $B$ and right child $C$ are pushed into the stack $StackNode$ and $StackAuth$ respectively.

Then nodes $\{D, I, J\}$ are pushed into stack $StackNode$, and nodes $\{E, H, K\}$ are pushed into stack $StackAuth$. The process is described as step No.(3) to step No.(5) in Figure

3 and Figure 4. After the first given leaf $J$ is found, the status of stack $StackNode$ and $StackAuth$ is shown at the left side of Figure 4, i.e. step No.(5).

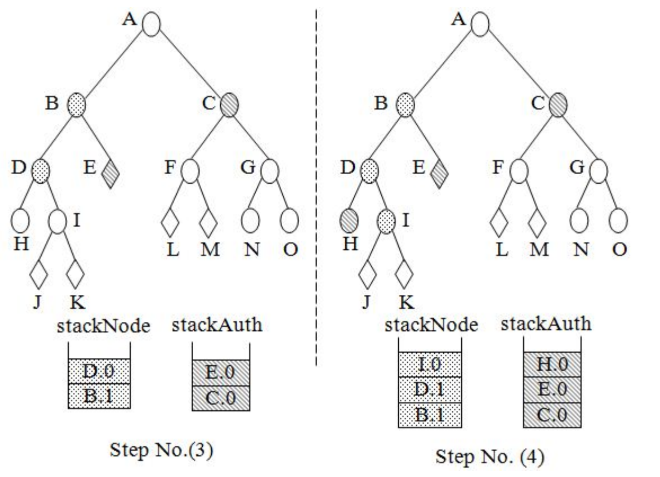

Figure 3: Two stacks status are updated when top element of $stackNode$ is an interior node with flag 0.

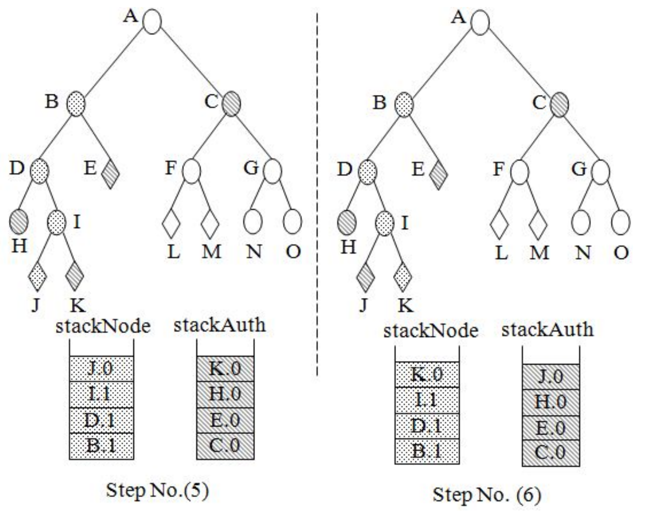

Figure 4: Authentication path nodes $\{K, H, E, C\}$ are output for leaf node $J$. Then the stacks are updated when top element of $stackNode$ is a leaf node, and its right brother is at the top of stack $stackAuth$.

In the second stage of the algorithm 2, the authentication path for the rest of the given leaf nodes $\{K, E, L, M\}$ should be generated, which is described as step No.(6) to step No.(14) in Figure 4 to Figure 8. The corresponding pseudo-codes are in the second *while* loop of algorithm 2.

**Five cases** of the second *while* loop in the algorithm 2 will be described in the following. In order to make it clear, we would explain the example step-by-step, so the cases would be explained in time order, not in numerical order.

When the first *while* loop of algorithm 2 is finished, the top element of stack $stackNode$ is $J$, which is a leaf node, as shown at the left side of Figure 4, i.e. step No.(5). The authentication path nodes $\{K, H, E, C\}$ are output for leaf $J$. **This is the case 5: Top element of** $stackNode$ **is a leaf node, and its right brother is at the top of stack** $stackAuth$**.** After the authentication path nodes are output for leaf $J$, the top element of stack $stackAuth$ is $K$, it is the right brother of node $J$. The top elements of stack $stackNode$ and $stackAuth$ are exchanged according to **case** 5 of Algorithm 2. After this operation, the stacks and the corresponding binary hash tree are illustrated at the right side of Figure 4, i.e. step No.(6).

As shown at the right side of Figure 4, i.e. step No.(6), the top element of stack $stackNode$ is $K$, it is a leaf node. The authentication path nodes $\{J, H, E, C\}$ are output for leaf $K$. It has left brother on the top of stack $stackAuth$, so **this is the case** 4**: Top element of** $stackNode$ **is a leaf node, and it has left brother at the top of stack** $stackAuth$**.** The top elements of stack $stackNode$ and $stackAuth$ are popped according to **case** 4 of Algorithm 2. After this operation, the stacks and the corresponding binary hash tree are illustrated at the left side of Figure 5, i.e. step No.(7).

As shown at the left side of Figure 5, the top element of stack $stackNode$ is $I$, it is an interior node with flag 1. It has left brother on the top of stack $stackAuth$, so **this is the case 3: Top element of** $stackNode$ **is an interior node with flag** 1 **and has left brother on top of stack** $stackAuth$. The top elements of stack $stackNode$ and $stackAuth$ are popped according to **case** 3 of Algorithm 2. After this operation, the stacks and the corresponding binary hash tree are illustrated at the right side of Figure 5, i.e step No.(8).

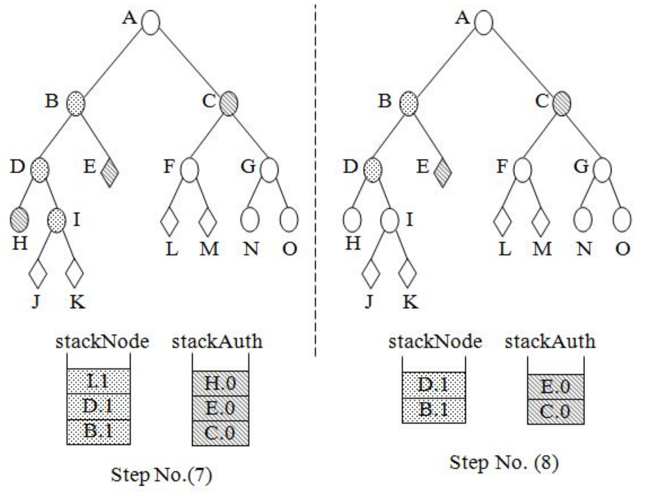

Figure 5: Two stacks status are updated when top element of $stackNode$ is an interior node with flag 1 and it has left brother on top of stack $stackAuth$.

As shown at the right side of Figure 5, the top element of stack $stackNode$ is $D$, it is an interior node with flag 1. The top element of stack $stackAuth$ is $E$ and $E$ is the right brother of node $D$. **This is the case** 2**: Top element of** $stackNode$ **is an interior node with flag** 1**, and its right brother is at the top of stack** $stackAuth$**.** The top elements of stack $stackNode$ and $stackAuth$ are exchanged according to **case** 2 of Algorithm 2. After this operation, the stacks and the corresponding binary hash tree are illustrated at the left side of Figure 6, i.e. step No.(9).

When status of stack $stackNode$ and $stackAuth$ as shown

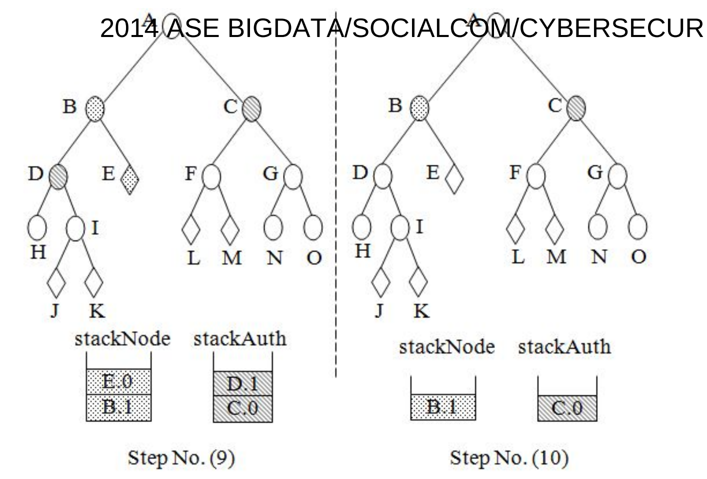

Figure 6: Authentication path nodes $\{D, C\}$ are output for leaf node $E$. Then the stacks are updated when top element of $stackNode$ is a leaf node, and its left brother is on the top of stack $stackAuth$.

in step No.(9) are reached, authentication path of leaf $E$ are output. The top element of stack $stackNode$ is $E$ with flag 0, and it has left brother at the top of stack $stackAuth$. The top elements of these two stacks are popped according to case 4 of the algorithm 2, which is described at the right side of Figure 6, i.e. step No.(10).

After that the top element of stack $stackNode$ is $B$, it is an interior node with flag 1, and it has right brother at the top of stack $stackAuth$. The top elements of these two stacks are exchanged according to case 2 of the algorithm 2. The status is shown at the left side of Figure 7, i.e. step No.(11).

As shown at the left side of Figure 7, the top element of stack $stackNode$ is $C$. It is an interior node with flag 0, **this is the case** 1**: Top element of** $stackNode$ **is an interior node with flag** 0. So its left child and right child are pushed into the stack $stackNode$ and $stackAuth$ respectively according to **case** 1 of algorithm 2. After this operation, the stacks and the corresponding binary hash tree are illustrated at the right side of Figure 7, i.e. step No.(12).

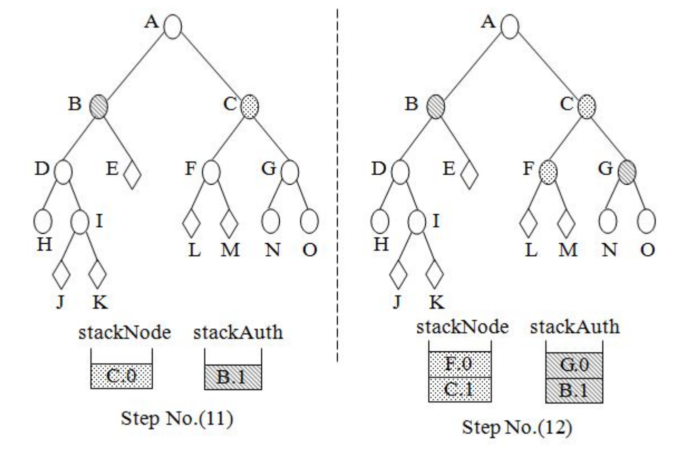

Figure 7: Two stacks status are updated when top element of $stackNode$ is an interior node with flag 0.

As shown at the right side of Figure 7, the top element of stack $stackNode$ is $F$. It is an interior node with flag 0, **this is the case** 1**: Top element of** $stackNode$ **is an interior node with flag** 0. So its left child and right child are pushed into the stack $stackNode$ and $stackAuth$ respectively according to **case** 1 of algorithm 2. After this operation, the stacks and the corresponding binary hash tree are illustrated at the left side of Figure 8, i.e. step No.(13).

As shown at the left side of Figure 8, the top element of stack $stackNode$ is $L$. It is a leaf node, so its authentication path $\{M, G, B\}$ are output. As $L$ has right brother $M$ at the top of stack $stackAuth$, **This is the case** 5**: Top element of** $stackNode$ **is a leaf node, and its right brother is at the top of stack** $stackAuth$**.** So according to **case** 5 of algorithm 2, top elements of stack $stackNode$ and $stackAuth$ are exchanged. After this operation, the stacks and the corresponding binary hash tree are illustrated at the right side of Figure 8, i.e. step No.(14). $M$ is the last given leaf that has been reached, its authentication path $\{L, G, B\}$ are output.

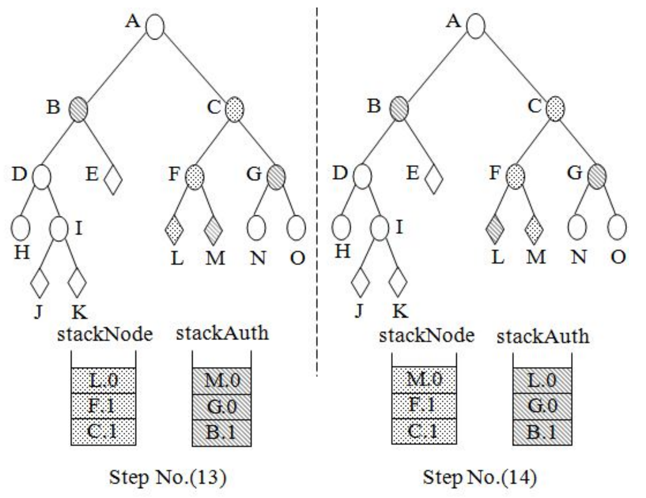

Figure 8: Authentication path nodes $\{M, G, B\}$ are output for leaf node $L$. Then the stacks are updated when top element of $stackNode$ is a leaf node, and its right brother is at the top of stack $stackAuth$.

## 4.2 Features of our algorithm

1. Our algorithm is designed to compute the authentication path of the binary hash tree. The application domains of the proposed algorithm include not only balanced binary hash tree e.g. Merkle hash tree but also unbalanced ones, such as H-CRT, skew Merkle tree and QuasiModo tree. In DPDP scheme, the proposed algorithm here still works even if the dynamic data update operations destruct the balance status of the Merkle tree. That is the reason why we call it a more versatile algorithm.

2. It is structurally very simple. Only two stacks $stackNode$ and $stackAuth$ are needed in the algorithm to store the currently visited node and its brother

nodes respectively during the pre-order traversal. Whatever leaf node is reached, all the elements in the *stackAuth* stack are its authentication path nodes.

3. The order of *push* operation of stack *stackNode* is the pre-order of the binary hash tree traversal. The leaves are visited according to the natural indexing, i.e. from left to right, which has the advantage that we can reduce the amount of computation, since that $leaf_i$ and $leaf_{i+1}$ share a large portion of their authentication paths.

4. Unlike the previous algorithms, we can compute the authentication path of some specific leaves by giving some parameters. It means that our algorithm is more flexible and we can compute authentication path in time.

5. Unlike the previous algorithms, the height of nodes in the binary hash tree need not be computed in our algorithm.

6. After our algorithm is finished, the order of the elements in the stack *stackAuth* is the same as what is needed to verify the leaf node by root node value.

### 4.3 Time and space analysis

For the case that the hash tree is a balanced one, we will analyze the time and space complexity here.

As in algorithm 2, we can compute authentication path for different situations by giving different parameters, one is only for a specific leaf, the other is for a sequence of leaves. Let $N$ indicate the number of all leaves in the binary hash tree, $M$ stand for the number of leaves involved in the authentication path computation. For a binary hash tree of maximal height $H, H = \log(N)$:

1. The computation for a specific leaf: The number of $Push$ operations is $2H = 2\log(N)$ when computing the authentication path of the first leaf, since there are two stacks i.e. *stackNode* and *stackAuth* being used. The number of $Push$ operations is much less than $2\log(N)$ for any other following leaf since the neighboring leaves share a large portion of their authentication paths. The number of $Push$ operations for the first leaf is $2\log(N)$, so the storage of $2\log(N)$ nodes is needed. Due to the fact that nodes in stacks are discarded when no longer needed, the storage of $2\log(N)$ nodes at most are needed during the computation. Therefore the time and space complexity for this case are both $2\log(N)$.

2. The computation for a sequence of leaves: According to the property of balanced binary tree, the number of all nodes (including leaf and interior nodes) in the tree is $2N - 1$. Since every node in the tree experiences a $Push$ operation and a $Pop$ operation during the pre-order traversal, the number of total $Push$ and $Pop$ operations is $2(2N - 1)$. The average time complexity is $\frac{2M(2N-1)}{N}$ when computing the authentication paths for $M$ sequential leaves. The space complexity here is also $2\log(N)$ as explained in the first situation.

## 5 Comparison

We now compare our algorithm with classic Merkle algorithm from [9], Szydlos algorithm from [7, 11] and fractal algorithm from [6] in two aspects, i.e. applicability and performance.

### 5.1 Applicability

As table 2 shows, our algorithm is applicable to all binary hash trees, while other algorithms in table 2 is only applicable to Merkle hash tree.

Table 2: Comparison of the applicability of our algorithm, classic Merkle algorithm from [9], Szydlos algorithm from [7, 11] and fractal algorithm from [6].

| Traversal Technique | Merkle Tree | Other Binary Hash Trees |
|---|---|---|
| Classic Merkle Traversal 1979 | $Y$ | $N$ |
| Logarithmic Traversal 2003 | $Y$ | $N$ |
| Logarithmic Traversal 2004 | $Y$ | $N$ |
| Fractal Traversal 2003 | $Y$ | $N$ |
| Our Algorithm 2014 | $Y$ | $Y$ |

### 5.2 Space and Time Requirements

Since these algorithms[6, 7, 9, 11] in table 2 are only applicable to Merkle tree, we only consider the Merkle tree here even if our algorithm is applicable to both balanced Merkle tree and other binary hash trees.

As Table 3 shows, when compared with the Classic Merkle Traversal and Fractal Traversal, our algorithm shows a nearly equal performance in time cost and a better performance in storage cost. When compared with the Logarithmic Traversal, our algorithm shows a nearly equal performance in both time cost and storage cost. It follows that our algorithm is an efficient one.

## 6 Conclusion

We propose a new algorithm for the computation of authentication path in a binary hash tree. Before the computation of authentication path, a binary sort hash tree should be built by adding a keyword field to every node. The main idea of our algorithm is to use two stacks to store visited nodes and its sibling nodes in pre-order traversal respectively. Unlike previous algorithms, the proposed algorithm is applicable to both balanced hash tree, such as Merkle

Table 3: Comparison of the space and time requirements of our algorithm, classic Merkle algorithm from [9], Szydlos algorithm from [7, 11] and fractal algorithm from [6].

| Traversal Technique | Time Cost | Storage Cost |
|---|---|---|
| Classic Merkle Traversal 1979 | $2\log(N)$ | $log^2(N)/2$ |
| Logarithmic Traversal 2003 | $\log(N)$ | $3log(N)$ |
| Logarithmic Traversal 2004 | $2\log(N)$ | $3log(N)$ |
| Fractal Traversal 2003 | $2\log(N)/\log(\log(N))$ | $1.5\log^2(N)/2\log(\log(N))$ |
| Our Algorithm 2014 | $2\log(N)$ | $2\log(N)$ |

tree, and unbalanced ones, such as skew Merkle tree and H-CRT. Besides, it is structurally very simple and functionally very flexible, when compared with previous algorithms. Its performance in time and storage is also good.

## 7 Acknowledgment

This work was supported in part by the Fundamental Research Funds for the Central Universities (No.2011JBM221 and No.2012JBZ010), the Program for New Century Excellent Talents in University under Grant No.NCET-11-0565, and the Program for Changjiang Scholars and Innovative Research Team in University under Grant No.IRT201206.

## References

[1] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function, Proceedings of Crypto '87, pp. 369-378.

[2] L. Lamport. Constructing Digital Signatures from a One Way Function, SRI International Technical Report CSL-98 (October 1979).

[3] S. Micali. Efficient Certificate Revocation, Proceedings of RSA '97, and U.S.Patent No. 5,666,416.

[4] C. Jutla and M. Yung. PayTree: amortized-signature for flexible micropayments, 2nd USENIX Workshop on Electronic Commerce, pp.213-221, 1996.

[5] Luis Carlos Coronado Garcia. On the security and the efficiency of the Merkle signature scheme, 2005, available at eprint.iacr.org/2005/192.pdf.

[6] Markus Jakobsson, Tom Leighton, Silvio Micali, and Michael Szydlo. Fractal Merkle tree representation and traversal. In Cryptographer 's Track at RSA Conference CT-RSA '03, volume 2612 of Lecture Notes in Computer Science, pp.314-326, 2003.

[7] Michael Szydlo. Merkle tree traversal in log space and time. 2003. Preprint, available at http://www.szydlo.com/.

[8] Ralph C. Merkle. A certified digital signature. In CRYPTO '89: Proceedings on Advances in cryptology, volume 435 of Lecture Notes in Computer Science, pp.218-238. Springer-Verlag, 1989.

[9] Ralph C. Merkle. Security, Authentication, and Public Key Systems, Stanford University, CA, USA, Technical Report No. 1979-1, 1979.

[10] Piotr Berman, Marek Karpinski, and Yakov Nekrich. Optimal trade-off for Merkle tree traversal. Theoretical Computer Science, 372(1): 26-36, 2007.

[11] Michael Szydlo. Merkle tree traversal in log space and time. In Advances in Cryptology EUROCRYPT 2004, volume 3027 of Lecture Notes in Computer Science,pp.541-554, 2004.

[12] Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle tree traversal revisited. PQCrypto, volume 5299 of Lecture Notes in Computer Science, pp.63-78. Springer, (2008).

[13] Xiuxia Tian, Yuan Peng, Hongjiao Li, Weimin Wei, Yong Wang. An Efficient Approach for Merkle Tree Traversal, Journal of Information & Computational Science 9:15 (2012) 4541-4549.

[14] Johannes Buchmann, Erik Dahmen, and Andreas Hulsing. XMSS-A Practical Forward Secure Signature Scheme based on Minimal Security Assumptions, PQCrypto, volume 7071 of Lecture Notes in Computer Science, pp.117-129. Springer, (2011).

[15] Qian Wang, Cong Wang et al. Enabling Public Auditability and Data Dynamics for Storage Security in Cloud Computing, IEEE Transactions on Parallel and Distributed Systems, 22(5),2011: 847-859.

[16] Farid F. Elwailly, Craig Gentry, Zulfikar Ramzan. QuasiModo: Efficient Certificate Validation and Revocation, In 7th International Workshop on Theory and Practice in Public Key Cryptography, Singapore, March 1-4, 2004.

[17] Marek Karpinski, and Yakov Nekrich. A Note on Traversing skew merkle trees, ECCC Report TR04-118, 2004.

[18] Sunoh Choi, Hyo-Sang Lim and Elisa Bertino. Authenticated Top-K Aggregation in Distributed and Outsourced Databases, In PASSAT,2012 International Confernece on Social Computing (SocialCom), pp.779-788, 2012.

[19] Yuan Xue,Yongbin Zhou, Jianfeng Guo, and Xizhen Ni. Hufman-based certificate revocation tree, Journal on Communications, Vol.2, No.2, 2005. pp.45-50.

[20] Weimin Yan, Weimin Wu. Data Structure, Tsinghua University Press, Beijing, China, 1997.